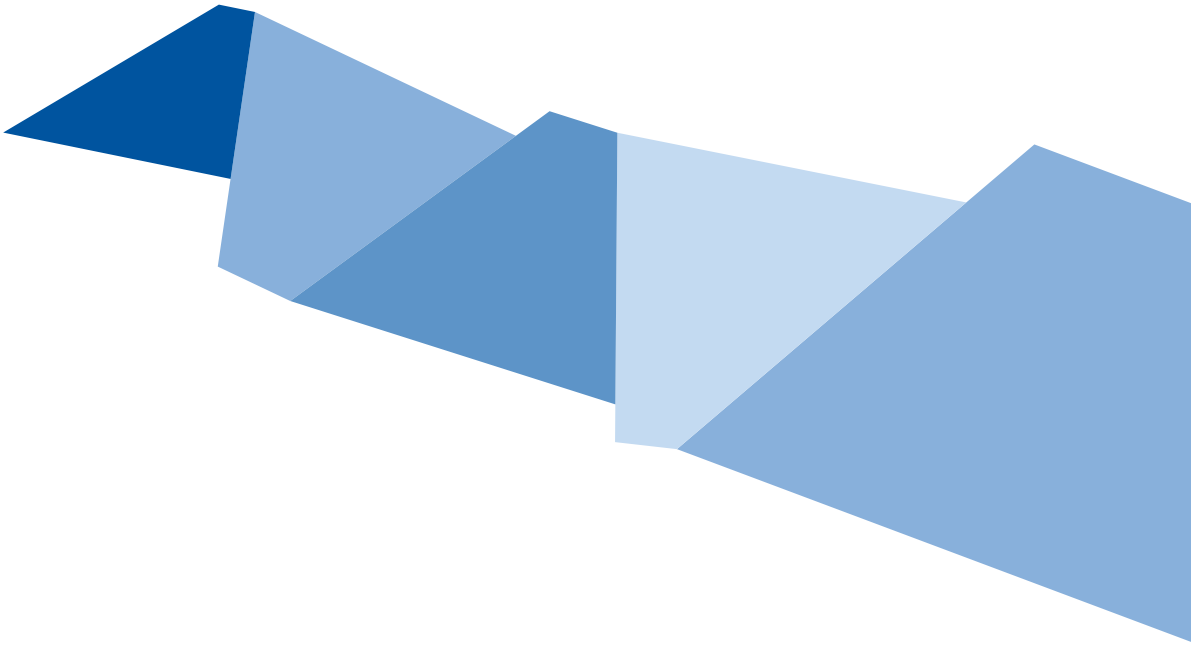

Real-World Deployment and Evaluation of Synchronous Programming in Reactive Embedded Systems

Matthias Terber

Department of Computer Science

Technical Report

Aachener Informatik-Berichte (AIB) | ISSN 0935-3232 | AIB-2018-05
RWTH Aachen University | Department of Computer Science | November 2018

The publications of the Department of Computer Science of *RWTH Aachen University* are in general accessible through the World Wide Web.

`http://aib.informatik.rwth-aachen.de/`

Real-World Deployment and Evaluation of Synchronous Programming in Reactive Embedded Systems

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der
RWTH Aachen University zur Erlangung des akademischen Grades
eines Doktors der Ingenieurwissenschaften genehmigte Dissertation

vorgelegt von

Matthias Terber, M. Sc.
aus Gießen

Berichter: Universitätsprofessor Dr.-Ing. Stefan Kowalewski
Universitätsprofessor Dr. rer. nat. Bernhard Rumpe

Tag der mündlichen Prüfung: 29. Oktober 2018

Matthias Terber
Lehrstuhl Informatik 11
terber@embedded.rwth-aachen.de

Aachener Informatik Bericht AIB-2018-05

Herausgeber: Fachgruppe Informatik
RWTH Aachen University
Ahornstr. 55
52074 Aachen
GERMANY

ISSN 0935-3232

Abstract

Pervasive smart devices link embedded concerns to information technology in a single, resource-constrained system. Both domains have quite opposite computation characteristics – *reactive* versus *transformational*. Due to C's dominance in industry, the reactive part is usually based on conventional sequential programming which lacks domain-specific support making the solution hard to program, comprehend and maintain. Synchronous languages might be a promising solution in order to facilitate software engineering and improve software quality. However, to date, they are restricted to very specific industrial niches; real-world deployments and evaluations are rarely reported in literature.

This thesis conducts a case study that examines the feasibility and suitability of the synchronous approach based on a real-life smart device. Focusing on its reactive concerns, it elaborates the engineering challenges and quality issues of the existing production code. By taking advantage of the synchronous language Céu, it outlines a synchronous reimplementation, thereby illustrating the deployment of synchronous programming and how to reconcile it with the transformational part of the system. Architectural considerations and best practices are provided for developers in order to effectively apply the synchronous language concepts. Furthermore, it shows the applicability of established object-oriented software design patterns and how to implement reproducible unit tests for reactive code.

Several qualitative discussions treat the software engineering and quality benefits gained by the synchronous reimplementation compared to the existing production code. A code analysis uses the separation of concerns, the scattering of interfaces and the code size as performance indicators in order to quantitatively substantiate the results. A user study confirms that reactive behavior is easier to implement and comprehend using the synchronous approach.

This thesis represents a proof of concept which demonstrates the feasibility and suitability of synchronous programming in resource-constrained, real-life industrial embedded applications that are exposed to reactive and transformational concerns likewise. By using synchronous programming, we were able to recover fundamental software engineering principles while, at the same time, fulfill the strong resource limitations – a combination that is known to be hard to achieve. Finally, we believe that our work generally suggests a practicable way of improving embedded software quality in industrial applications.

Zusammenfassung

Allgegenwärtige *Smart Devices* verbinden den eingebetteten Anwendungsbereich mit Informationstechnologie in einem einzigen System unter Einsatz begrenzter Ressourcen. Beiden Domänen liegen dabei ganz unterschiedliche Berechnungscharakteristiken zu Grunde – *reaktiv* versus *transformierend*. Durch die große Dominanz der Programmiersprache C im industriellen Bereich, werden die Probleme der reaktiven Domäne üblicherweise mit konventioneller, sequentieller Programmierung adressiert. Diese bietet jedoch keine spezifische Unterstützung für die Beschreibung von reaktivem Verhalten, wodurch Entwicklung, Verständlichkeit und Wartbarkeit der Software erschwert werden. Synchrone Programmiersprachen könnten eine vielversprechende Lösung darstellen, um den Entstehungsprozess von Software zu vereinfachen und deren Qualität zu verbessern. Bis heute ist der Einsatz von synchroner Programmierung jedoch nur auf sehr spezielle industrielle Anwendungsfälle beschränkt; ihre praktische Anwendung und deren Nutzen werden in existierender Literatur kaum behandelt.

In dieser Arbeit wird eine Fallstudie durchgeführt, welche die Anwendbarkeit und Eignung des synchronen Programmierparadigmas anhand eines konkreten, existierenden Smart Devices aus der Industrie untersucht. Mit Fokus auf die reaktive Domäne werden die technischen Herausforderungen für die Entwicklung sowie die Qualitätsprobleme der bestehenden Softwarelösung herausgearbeitet. Basierend auf der synchronen Sprache Céu wird eine synchrone Neuimplementierung skizziert. Diese veranschaulicht den Einsatz von synchroner Programmierung und zeigt deren Integration in den transformierenden Teil des Gesamtsystems. Dem Entwickler werden architektonische Überlegungen und bewährte Vorgehensweisen an die Hand gegeben, um die synchronen Sprachkonzepte effektiv zu nutzen. Außerdem wird die Anwendbarkeit von etablierten objekt-orientierten Softwareentwurfsmustern demonstriert und aufgezeigt, wie sich reproduzierbare Tests für reaktiven Code implementieren lassen.

Mehrere qualitative Diskussionen behandeln die Vorteile für Softwareentwicklung und -qualität, welche sich durch die synchrone Neuimplementierung im Vergleich zur existierenden Lösung ergeben. Eine Codeanalyse verwendet die Trennung der Belange (*separation of concerns*), die Verteilung der Komponentenschnittstellen (*scattering of interfaces*) sowie die Codegröße (*code size*) als Indikatoren, um die qualitativen Ergebnisse quantitativ zu belegen. Weiterhin bestätigt eine Nutzerstudie, dass reaktives Verhalten mit dem synchronen Ansatz einfacher zu implementieren und zu verstehen ist.

Diese Arbeit ist ein Konzeptnachweis für die Umsetzbarkeit und Eignung von synchroner Programmierung in ressourcenbegrenzten, eingebetteten Systemen im industriellen Kontext, die sowohl mit reaktiven als auch transformierenden Berechnungen konfrontiert sind. Durch den Einsatz des synchronen Paradigmas wurden grundlegende Prinzipien der Softwareentwicklung wieder anwendbar ohne dabei die starken Ressourcenanforderungen zu verletzen – eine Kombination, die bekanntermaßen schwer zu erreichen ist. Schließlich glauben wir, dass diese Arbeit grundsätzlich einen praktikablen Weg aufzeigt, wie sich die Qualität eingebetteter Software in industriellen Anwendungen verbessern lässt.

Acknowledgments

I would like to thank the Bosch Thermotechnik GmbH, particularly Dr.-Ing. Jürgen Hötzel, for the opportunity to perform this industry-related research.

I would like to express my sincere gratitude to my supervisor Prof. Dr.-Ing. Stefan Kowalewski for accepting me as a doctoral candidate, the trust in me, the freedom given to my work, the helpful and constructive participation and the close cooperation during the entire doctorate period. My thanks also go to my advisor Dr. rer. nat. Mark Hönig from Bosch Group for proof reading this thesis and the fruitful discussions. Also, I would like to thank Prof. Dr. rer. nat. Bernhard Rumpe for accepting to be a co-referee of my doctorate thesis and the interest in my work.

Furthermore, I would like to express my deepest thanks to Franz-Josef Grosch and Dr. rer. nat. Friedrich Gretz from Bosch Corporate Research for their considerable guidance, helpful advices, valuable information and constant support during the development of this thesis. Also, I thank Prof. Dr.-Ing. Jens Brandt for his suggestions and useful hints during the first phase of my work.

Moreover, I would like to thank Dr. Francisco Sant'Anna for answering my countless questions when I familiarized myself with the semantics and the deployment of the programming language Céu. My thanks also go to Prof. Dr. Berthold Franzen for the opportunity to conduct the user study at the University of Applied Sciences in Gießen.

Many thanks go to all my dear colleagues from Bosch Thermotechnik GmbH in Lollar, in particular Christian Berger, Marc Immel, Frank Schäfer and Harald Klinkel, for their support with all problems related to soft- and hardware development and providing a pleasant working atmosphere.

Last but not least, I am very thankful to my parents and my brother for their steady encouragement and support during the entire period of my studies.

Thank you very much!

Matthias Terber
18th November, 2018, Lollar

Contents

List of Tables

List of Figures

Listings

List of Abbreviations

API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
CAN	Controller Area Network
CFS	Completely Fair Scheduler
CRC	Cyclic Redundancy Check
EMS	Energy Management System
FIFO	First-In, First-Out
GALS	Globally Asynchronous, Locally Synchronous
HIL	Hardware-In-the-Loop
HMI	Human Machine Interface
HTTP	Hypertext Transfer Protocol
I/O	Input/Output
IDE	Integrated Development Environment
LED	Light-Emitting Diode
LIFO	Last-In, First-Out
LOC	Lines of Code
M-Bus	Meter-Bus
OSEK	Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen
POSIX	Portable Operating System Interface
RAM	Random Access Memory
REST	REpresentational State Transfer
ROM	Read Only Memory
RTS	Real-Time Scheduler

SIL	Software-In-the-Loop
UUT	Unit Under Test
VDX	Vehicle Distributed eXecutive
WCET	Worst-Case Execution Time

1 Introduction

The past decades have seen a steady and world-wide increasing dissemination of embedded computer systems in virtually all areas of everyday life. Popular slogans such as Internet of Things, Cloud Computing, Augmented Reality, Ambient Intelligence, Big Data and Industry 4.0 are just some of today's drivers towards this unstoppable trend. Usually, these efforts produce so called *smart devices* in various fields, for example consumer electronics, automotive and aviation solutions, industrial applications, telecommunication services and medical devices. All these embedded systems have in common that they link traditional embedded concerns to conventional information technology in a single device. However, these problem domains appear to have quite different computation characteristics. On the one hand, the traditional embedded domain typically realizes the control part. Control-intensive tasks are periodically or sporadically prompted by the environment and have to respond in a timely manner. Conventional information technology, on the other hand, mostly focuses on data processing which is usually not time-critical but demands computation-intensive tasks. Following Harel and Pnueli [HP85], we use the terms *reactive* and *transformational* in order to distinguish these two types of systems. It seems reasonable that this diversity requires domain-specific hard- and software support in order to promote simple and concise solutions.

Looking back, it becomes apparent that embedded hardware made substantial progress in terms of computation power and storage capacity. Due to mass production, microcontrollers have become cheaper ever since and at the same time provide a widely extended range of domain-specific functionalities. Today's low-cost 32-bit devices, for instance, may be equipped with dedicated cryptographic acceleration, real-time coprocessors, floating point units and various communication interfaces. When it comes to embedded software, in contrast, it appears that distinctive changes have been less frequent within the last decades.

> "Regardless of how advanced our [embedded] products may be, our methods for designing them are almost medieval." [Tur12, p. 24]

The introduction of the programming language C [KR78] in the 1970s – one of the most fundamental advances – replaced unstructured, hardware-dependent assembler code. Having appeared almost half a century ago, C intends to provide a lightweight, easy to learn and relatively low-level system programming language based on imperative, sequential, single-threaded control flow. While in the following years a number of higher-level alternatives have popped up, for example C++ or Real-Time Java, to date they have not conquered the embedded systems domain. Due to a number of practical

advantages [NM12] C has become *the* de facto standard for industrial, embedded applications world-wide [BM06; EJ09; Sak12; EE15]. However, C's conventional sequential programming generally does not contribute any domain-specific language support. Together with the rapid increase of system complexity many companies nowadays have run into software quality problems [LT09]. In particular, this becomes apparent when considering the smart devices mentioned above. While C's sequential style seems reasonably suitable for the transformational part, reactive control behavior sets a different significantly challenge.

Reactive concerns require continuous event- and time-based interaction with the system's physical environment [HP85]. The conventional sequential model of computation, however, does not contribute any language-level support facilitating this kind of cooperation. Workarounds in C usually deploy callbacks for event handling which force embedded system developers to deal with manual stack [Ady+02; Kas07] and state [Kas07] management. By this, implementing reactive behavior in C becomes a challenging and error-prone task [Bai+13]. Actually, these workarounds are known to encourage the violation of fundamental software engineering principles, for example information hiding, separation of concerns and modularization, making the solution hard to program, comprehend and maintain [MO12]. On this account, the reactive programming paradigm [Bai+13] has been proposed as a general solution. Reactive languages and frameworks provide proper computing abstractions which integrally take care of event handling logic and state management, thereby considerably reducing the burden faced by developers. However, it appears that, for industrial real-world embedded applications, the vast majority of approaches is not applicable since they cannot keep up with C's practical advantages.

An exception are synchronous languages [Ben+03] which have been specifically designed for modeling, specifying, validating and implementing reactive real-time embedded systems. They provide promising features such as deterministic concurrency as well as bounded memory and bounded reaction time. While some visual notations for synchronous languages have found their way in successful industrial use, the SCADE [Est14] tool for instance, their deployments are limited to the fringe group of highly safety-critical systems. Apart from that, synchronous programming might be generally a suitable and useful solution to provide domain-specific support for the implementation of reactive concerns in broad industrial embedded applications. This points to a potential improvement of the overall embedded software quality in systems such as smart devices that cover reactive and transformational concerns at the same time. However, existing literature introduces the synchronous paradigm but does rarely report and evaluate its deployment in a real-world application. In particular, current research does not consider the applicability and integration of the synchronous technology in an industrial use case and lacks a detailed investigation of its effect on software engineering and software quality.

1.1 Objectives

The main objective of this thesis is to study whether synchronous programming is a feasible and suitable approach in order to simplify software engineering and improve software quality of resource-constrained, real-world industrial embedded applications that are exposed to transformational and reactive concerns at the same time.

In the scope of this thesis we conduct a case study that examines an existing industrial smart device – a heating gateway [Bos17] – developed and marketed by Bosch Thermotechnik GmbH. The gateway is subject to strong resource limitations as well as soft real-time requirements and is completely based on conventional sequential programming. For synchronous programming, we take advantage of the third-party, C-based, synchronous, imperative, reactive programming language Céu [San+13].

In particular, we have to examine the gateway's existing production code with respect to engineering challenges and quality issues in order to illustrate today's actual state and establish a baseline for our comparison. We must identify the reactive concerns in order to have a suitable basis for deploying and evaluating the synchronous paradigm. We have to demonstrate the actual deployment of synchronous programming to

- show its feasibility in a real-world industrial use case
- illustrate the reconciliation of reactive, synchronous code and transformational, asynchronous code
- help developers through general considerations, guidelines and best practices for effectively applying synchronous language constructs

We have to perform a comparison that evaluates the effect of the domain-specific language support of synchronous programming with respect to software engineering and quality.

Note on Intellectual Property The gateway implementation comprises intellectual property of Bosch Group. Thus, we can neither publish complete communication protocols nor the original code base. However, presented code chunks are derived from production code as close as possible and explicitly retain the core issue on focus. Time specifications and numerical values are distorted but retain their order of magnitude.

1.2 Contributions

The main contributions of this thesis are as follows:

Analysis of existing production code We provide a software engineering analysis of an existing resource-constrained, real-life industrial smart device which completely relies on conventional sequential programming.

- We identify the transformational and reactive concerns of the industrial use case and work out their computation characteristics in detail. We consider how to reconcile both domains.
- We reveal how the lack of domain-specific language support for reactive concerns actually manifests in production code. The analysis covers reactive key concerns such as event handling, concurrency and temporal behavior.

Synchronous reimplementation We outline a corresponding reimplementation in Céu that aids as a proof of concept for synchronous programming in a real-world industrial use case.

- We provide architectural considerations and practical demonstrations on how to reconcile the reactive, synchronous and the transformational, asynchronous part of the system.
- We present a guideline for developers on how to suitably choose and deploy synchronous abstraction entities in Céu depending on the functionality to be realized. Further, we illustrate how they can be used in order to apply a divide-and-conquer strategy to the architectural design of synchronous code.
- We demonstrate the applicability of established object-oriented design patterns in resource-constrained embedded applications by taking advantage of Céu's abstraction entities.
- We illustrate a best practice for specifying and performing reproducible unit tests for reactive concerns that allow black-box and white-box testing likewise. In this context, we provide a classification of different possible deployment approaches of Céu's abstraction entities with respect to their testability.
- We provide some general hints that developers should be aware of when deploying synchronous code in Céu.

Comparative evaluation We provide a qualitative and quantitative evaluation that compares the new synchronous re-implementation to the existing production code with respect to software engineering and software quality.

- We provide several qualitative discussions that comparatively consider software engineering aspects and software quality.
- We provide a code analysis that extracts different quantitative performance indicators from both implementation approaches in order to allow an objective comparison, too.
- For an evaluation of the synchronous paradigm based on practical experiences from embedded software developers, we conduct a user study targeting undergraduate students of Computer Science for Engineers. We consider them to be the next generation embedded software developers which might be potential users of synchronous programming.

1.3 Thesis Outline

The rest of this thesis is organized as follows: In Chapter 2, we describe preliminaries of this work, including basic information about the industrial use case, the synchronous model of execution and the programming language Céu. In Chapter 3, we work out the transformational and reactive concerns of the industrial use case. In Chapter 4, we analyze the existing production code and elaborate the engineering challenges and quality issues that emerge if reactive concerns are implemented using conventional sequential programming. In Chapter 5, we present a corresponding synchronous reimplementation of the reactive concerns and discuss its effect on software engineering and quality. In Chapter 6, we substantiate our qualitative comparison from Chapters 4 and 5 in quantitative terms. In Chapter 7, we summarize this thesis and outline possible issues for future work.

1.4 Bibliographic Notes

Some parts of this thesis are based on work already presented in earlier publications. The identification of the gateway's problem domains and their respective computation characteristics as well as the domain-oriented software architecture based on Céu and Rust have been published in [Ter16]. A qualitative and quantitative comparison of the existing byte layer state machine and its reimplementation in Céu as well as considerations for the choice and integration of Céu have been published in [Ter17].

1.5 Related Work

Most of the existing literature about synchronous programming either introduces one of the three main synchronous languages Esterel [BS91], Signal [Le +91] and Lustre [Hal+91] or presents an overview of the synchronous paradigm in general [BB91; Hal93; Ben+03; STP05; CRT07]. However, related work focuses on language features, compilation and program verification rather than software engineering and software quality. Also, the adoption in a concrete industrial application is never examined.

Some research takes advantage of synchronous programming in order to develop new reactive libraries and frameworks. Poigne et al. [Poi+98] present a workbench that allows to mix different synchronous languages for a single application. Motika and Hanxleden [MH15] develop a synchronous extension for the programming language Java. Furthermore, there are some synchronous language deployments for specific use cases such as control applications [SG01], device drivers [BMM11] and audio processing [BJ13]. However, above work only relies on non-industrial sample applications.

It appears that the deployment and engineering-related evaluation of synchronous programming in real-life industrial applications has only superficially been considered by existing research papers. Murakami and Sethi [MS92] use Esterel to reimplement a telecommunication application. Their explanations focus on introducing selected language constructs. Also, they compare C and Esterel implementations with respect

to source and object code size. Code structuring capabilities are incidentally mentioned but not investigated in detail. Andre and Peraldi [AP93] generally reveal potential industrial applications for synchronous languages but do not examine a concrete use case. Jagadeesan et al. [JPV95] present a case study on the adoption of Esterel for a switching system which provides telecommunication services. Amongst others, they mention advantages of Esterel for software development with respect to abstraction and structuring capabilities. However, those explanations are not concretely exemplified nor are they compared to the existing industrial implementation. Benveniste et al. [Ben+03] and Halbwachs [Hal05] generally highlight the successful adoption of synchronous languages in industry but do not consider any concrete use case.

Finally, the synchronous programming language Céu appears in several scientific papers [SIR12; San+13; San13; SIR15; San+16]. The existing literature indicates some engineering benefits but does not consider a concrete deployment and evaluation in a real-word industrial system.

2 Preliminaries

In this chapter we introduce some preliminaries for this research. While the first section outlines the industrial use case under investigation, the following ones introduce the basic concepts of the synchronous model of execution and the synchronous programming language Céu.

2.1 Heating System Remote Control

The *heating gateway* [Bos17] is an embedded, real-world industrial smart device marketed by Bosch Thermotechnik GmbH. It provides remote control and monitoring of heating appliances via the Internet. Among others, it allows to extend the traditional Human Machine Interface (HMI) by feature-rich apps and service software, send notifications about malfunctions of the heating appliance and log history data to keep track of temperatures, operating states and error messages for instance. While Internet connectivity relies on the Hypertext Transfer Protocol (HTTP), Bosch's Energy Management System (EMS) fieldbus interfaces local heating devices using a proprietary communication protocol. By this, the gateway acts as a bridge between the heating domain and internet-ready devices such as smartphones, tablet PCs, notebooks and so on.

The EMS Fieldbus Heating appliances are interconnected via the EMS fieldbus for in-house communication purpose. Physically, the fieldbus is a two-wire connection that supports half-duplex communication. The group of bus members is composed of a single, designated *master* device and multiple *slaves* – the gateway is one of the slaves. Participants follow a request-response communication discipline based on frames in order to exchange information. A frame is a structured sequence of bytes terminated by a special end-of-frame character. Essentially, it is composed of address fields for the unique hardware address of the source and destination device, the actual payload data and a Cyclic Redundancy Check (CRC) sum. Figure 2.1 provides a corresponding illustration.

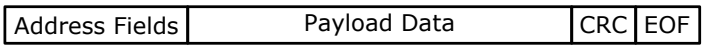

Figure 2.1: Basic fieldbus frame structure

Fieldbus access is controlled by the master using a time-slicing policy based on token passing. Therefore, each slave has two different operation modes. In *Passive*

Mode – the default – the device only responds to incoming requests from other fieldbus members. In contrast, during *Active Mode*, the device sends requests itself. Thus, only a device in Active Mode can initiate a request-response cycle on the fieldbus. In order to avoid collisions, only one participant is allowed to be in Active Mode at the same time. To achieve this, a token is passed by the master in sequence. As soon as a slave device receives the token it switches from Passive to Active Mode. By this, the master logically grants fieldbus access to the corresponding slave. The slave device switches back to Passive Mode and returns the token if it has no more requests to send or its communication time slot – denoted as *token time* (see below) – has expired. Due to the physical transmission method, each byte sent by a slave device is mirrored by the master. With respect to timing, the fieldbus communication protocol specifies the following time intervals (see Figure 2.2):

- t_m (*mirror time*): The time gap between a byte transmission and the reception of its corresponding mirror.
- t_i (*idle time*): The time gap between two consecutive, incoming frame bytes.
- t_r (*response time*): The time gap between the transmission of a request frame and the reception of a response frame.
- t_t (*token time*): The duration of a slave remaining in Active Mode.
- t_{ta} (*token active time*): The first time interval of t_t. During this period, the slave may transmit new requests.
- t_{tp} (*token passive time*): The second time interval of t_t. During this period, the slave is only allowed to wait for pending responses.

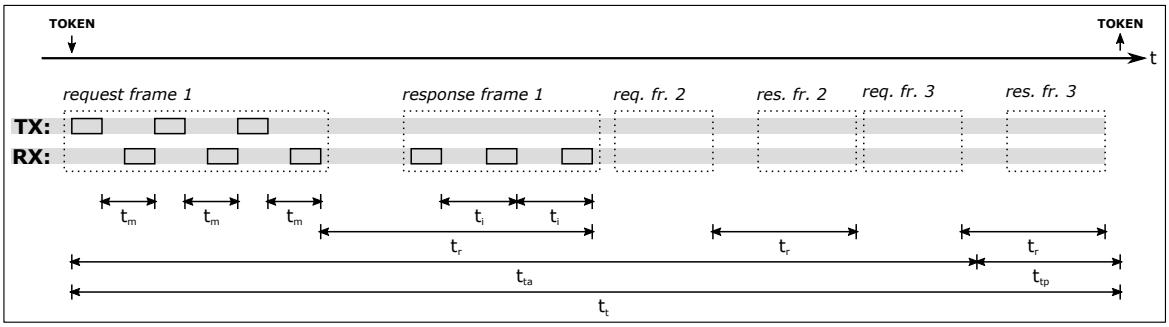

Figure 2.2: Fieldbus communication time intervals of a slave in Active Mode (TX: outgoing bytes, RX: incoming bytes)

Above time intervals are bounded by the maximum values presented in Table 2.1.

Time interval	Maximum value
t_m	42 ms
t_i	350 ms
t_r	225 ms
t_{ta}	800 ms
t_{tp}	200 ms
t_t	$t_{ta} + t_{tp}$

Table 2.1: Timing constraints for fieldbus communication

2.2 Synchronous Model of Computation

The fundamental model of computation for software is given by the pure, sequential execution of instructions [Neu93]. A concrete sequence of instructions aids in solving a certain problem. Following Sant'Anna et al. [San+13], we use the term *trail* to generally denote such a single line of execution. Due to their inherent concurrency [HP85], reactive embedded applications often require to run several trails concurrently in order to handle different, simultaneous concerns. Concurrent trail execution may be performed in an *asynchronous* or a *synchronous* manner [San09].

Asynchronous Concurrency In the prevailing *asynchronous model*, the trails are the leader of execution – they are in charge of their control flow. Conceptually, trails are blind to the surrounding system. They just keep running, independently of what happens outside. Thus, in order to cooperate, they have to explicitly synchronize their data and control flow occasionally. The decision if, when and how synchronization with the surrounding system takes place is not enforced by the environment but taken internally by each individual trail. For this reason, the asynchronous model is considered *trail-centric* [San09].

In this approach, the asynchronism leads to inherent non-deterministic behavior since it is impossible to predict the order in which concurrent trails interleave their execution [Lee06]. After each system restart, for example, a different scheduling may be performed depending on the current system state [San09].

Inter-trail communication is usually realized by shared-memory access or message passing. Both approaches take time and hence introduce a delay between data transmission and reception. Consequently, data obtained by the receiver may reflect a past state of the sender. This makes it difficult to achieve a global consensus about the system state. Thus, trails generally have a divergent vision of their environment and of each other. Also, race conditions must be manually avoided by explicit synchronization which entails the risk of concurrency bugs such as deadlocks for instance [Lee06; Lu+08; San09].

Synchronous Concurrency The *synchronous model* aims to overcome above limitations by combining synchrony with deterministic concurrency. In this approach, control

is inverted making the environment the leader of execution – trails must run at its pace and in permanent synchrony [San09]. Therefore, it relies on the *synchronous hypothesis* [STP05] which divides time and system behavior into a discrete sequence of non-overlapping computation steps that are commonly denoted as *reactions* [Ben+03; CRT07]. Essentially, the hypothesis makes three major assumptions [BB91; BG92]:

1. "*Output is synchronous with input.*" [BB91, p. 1277] A reaction is instantaneous and hence takes no time with respect to the external environment. Thus, it is atomic in any possible sense and the environment remains invariant during it.

2. "*Internal actions are instantaneous.*" [BB91, p. 1277] A trail takes no time with respect to other trails. They react instantly to each other.

3. "*Communications are performed via instantaneous broadcasting.*" [BB91, p. 1277] The propagation of data takes no time and is always visible to all trails.

Following this approach, trails run in a lock-step manner and are continuously in sync with each other and the environment. Thus, they have a global consensus about the system state.

Inside each reaction, the instantaneous property prohibits any side-effects that are visible across concurrent trails. The status of every signal or variable must be established and defined before they are read. By this, all concurrent trails use the same consistent data making the outcome independent from their execution order. Consequently, the behavioral propagation inside each reaction is deterministic – in particular for concurrent concerns [STP05].

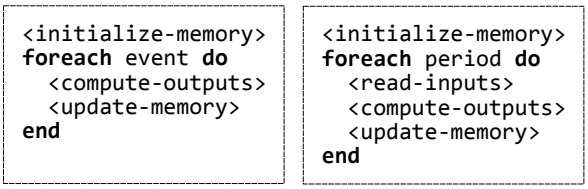

Figure 2.3: Two common synchronous execution schemes [Ben+03, p. 65]

Reactions are triggered by changes in the system's environment. There are two common synchronous execution schemes for this. In the *event-driven* scheme (see Figure 2.3a), each change manifests in an input event which triggers a reaction. This means that there is one reaction for each change. In the *sample-driven* scheme (see Figure 2.3b), a predefined (physical) time interval causes the system to cyclically execute a reaction, thereby polling the environment for changes [San09]. In this approach, several changes may be processed in a single reaction.

> "In an event-driven system, at least one input event is required to produce a reaction; in a sample-driven system, reactions are triggered by the clock ticks." [Ben+03, p. 69]

Irrespective of the adopted scheme, trails are idle by default. Once a reaction is required, they awake, perform their computations and return to idle. Thus, computations take place only at discrete points in (physical) time. Since the environment enforces synchronization and determines the operating speed, the synchronous model is considered *environment-centric* [San09]. Finally, Table 2.2 highlights the main differences between the synchronous and the asynchronous approach.

	Synchronous	Asynchronous
Control	environment	trail
Synchronization	implicit	explicit
	permanent	occasional
Communication	instantaneous	delayed
	broadcast	side effects (in shared memory)
		addressed (in message passing)
Determinism	deterministic	non-deterministic

Table 2.2: Concurrency models comparison [San09, p. 18]

Motivation of the Synchronous Approach Program correctness and efficiency are of outermost importance in safety-critical, resource-constraint embedded real-time systems. Meeting these requirements demands appropriate, domain-specific language support based on solid mathematical foundations. This provides the ability to reason formally about the system operation which facilitates formal verification and allows to prove certain aspects of the system's runtime behavior [Ben+03]. Note that in this context determinism is an indispensable feature since it allows to predict and reason about the system's runtime behavior at development time.

The synchronous model of execution is based on a common mathematical framework that combines synchrony with deterministic concurrency. In particular, it divides time and system behavior into a sequence of discrete instants (synchrony) whereby in each instant the behavior of concurrent computations is well-defined (deterministic). This approach is pervasive in mathematics and engineering, for example in automata theory, discrete-time dynamical systems and synchronous digital hardware logic. Those disciplines give access to a large corpus of universally recognized mathematical models such as the Mealy machines and the digital circuits. These models provide supporting foundations for efficient optimization, compilation, and formal verification techniques [Ben+03; STP05]. Timing analysis for instance – an important tool with respect to real-time constraints – is significantly easier in the synchronous approach. It only requires to check that the Worst-Case Execution Time (WCET) of a single reaction is smaller than the minimal time distance between two consecutive reactions [CRT07].

Adoption on Language-Level Historically, first implementation schemes of the synchronous model in embedded programming have emerged from programming practices of control and electronic engineers [CRT07]. Digital microprocessors have seen a rapid

adoption in the early eighties, thereby replacing the usual analog devices. Within a relatively short period of time, embedded system engineers were faced with a new technology that required to deal with instruction sets and system programming – concepts to which they were not accustomed. As a consequence, early embedded software usually followed the very simple program structures presented in Figure 2.3 which resemble the low-level `main`-loop approach for event-driven programming in conventional languages [CRT07; San09].

In the following years, two different high-level programming styles for synchronous systems have evolved [Ben+03]. In the declarative *data flow style*, supported by Lustre [Hal+91] and Signal [Le +91] for instance, data is represented by time-varying values which are linked by operators, thereby forming a dependency graph. On each clock tick, changes to values are automatically propagated throughout the network without explicitly programming. This style focuses on declaring dependencies between data, thereby making control flow implicit. In the imperative *control flow style*, supported by Esterel [BS91] for instance, synchronous code is organized by traditional basic control structures such as sequence, branch and iteration as well as parallelism. In this approach, the control flow is reflected by the program structure.

The above mentioned high-level languages are considered as the three main synchronous languages [Ben+03]. Their development has focused on the specification and verification of reactive embedded real-time hard- and software. Some of them have found their successful way into industrial use, for example in form of the SCADE (ANSYS) and Sildex (TNI- Valiosys) tools in case of Lustre and Signal respectively. However, their deployments are generally restricted to very specific industrial niches [San09].

2.3 The Programming Language Céu

The synchronous programming language Céu[1] [San+13] aims to offer a high-level and better suited alternative to C for the development of reactive, concurrent programs. Therefore, it provides *structured synchronous reactive programming* which augments classical structured programming with continuous environment interaction, thereby assuming the synchronous hypothesis (see Section 2.2). Céu targets the domain of control-intensive, resource-constrained, reactive embedded real-time systems in particular [San+13; SIR15]. In the following, we introduce its fundamental concepts.

Operation Purpose Primarily, Céu's language design focuses on expressiveness for reactive concerns and static safety guarantees while meeting runtime and memory requirements imposed by the embedded context [San13]. Event handling, physical time and concurrency – key concerns of the reactive domain – are integral parts of Céu and hence can be described in a very concise and readable way. In order to ensure responsiveness, the language constructs explicitly bound the execution time for each

[1] We used version 0.12b for our work (`https://github.com/fsantanna/ceu/tree/v0.12b`).

reaction; the same applies to the total amount of memory required during runtime. Thus, infinitely running reactions and memory overflows are prevented at compile time by language design instead of coding conventions. This constitutes an essential aspect of Céu's safety concept. Also, note that finite memory and time is required to preserve the synchronous hypothesis.

The intention of Céu is not to replace the industry standard C for embedded systems. Instead, it aims to seamlessly integrate into an existing C environment in order to facilitate the implementation of its reactive concerns. Actually, every Céu program is compiled into a single-threaded state machine in C. To simplify system integration and allow reuse of existing software, the execution of native C code is supported on language level.

Language Design Céu is a text-based, imperative programming language which is strongly influenced by Esterel [San13]. Compared to conventional sequential programming, it provides three major language extensions [San+13; SIR15]:

1. *Synchronous control statements*: For event handling, an (a) **`await`**-statement allows to suspend the currently running trail until the specified event occurs. For concurrency, a (b) **`par`**-block (parallel block) provides a structured composition of several concurrent trails. Along with this, an (c) orthogonal abortion mechanism is introduced to determine how concurrent trails rejoin. An **`or`**-abortion (**`par/or`**) terminates the whole **`par`**-block as soon as at least one of the covered trails has run to completion. In contrast, an **`and`**-abortion (**`par/and`**) requires all included trails to complete. Finally, a single **`par`** never rejoins.

2. *Adoption of physical time*: For specifying temporal behavior, physical time is adopted by the notion of *wall-clock time*. That is, "the passage of time from the real world, measured in hours, minutes, etc." [San13, p. 36] This allows to express time as a physical quantity in the code. For example, **`await`** `10ms` suspends the current trail for ten milliseconds.

3. *Object-like abstraction entities*: In order to abstract sequences of synchronous control statements, Céu provides the notion of *organisms* that encapsulate synchronous code with an object-like interface.

With respect to event handling, Céu distinguishes between two types of events. *External events* are used to interact with the environment respectively the surrounding host system (see System Integration below). External input events, for example **`input void`** `EIN`, are emitted by the environment and handled by the program in form of reactions. External output events, for example **`output u8`** `EOUT`, are emitted by the program and handled by the environment. *Internal events*, for example **`event void`** `notify`, are emitted and handled by the program internally. They can be used for coordinating computations of concurrent trails inside the same reaction. While a First-In, First-Out (FIFO) policy (queue-like) is used for processing external input events, internal ones follow a Last-In,

First-Out (LIFO) policy (stack-like) instead. By this, emitting an internal event behaves similar to calling a subroutine.

Furthermore, Céu relies on an efficient memory layout. In particular, it does not allocate a stack per trail but manages data in one fixed memory slot. The basic idea is that memory for concurrent trails must coexist whereas statements in sequence can reuse it. Mapping this concept to C is done by packing memory for blocks in parallel in a `struct` while blocks in sequence reside in a `union`. This allows to switch between different data interpretations during runtime, not depending on garbage collection [San13].

Program Execution Program execution in Céu is entirely event-driven and hence follows the scheme depicted in Figure 2.3a where input events map to reactions in a one-to-one relationship. The adoption of physical time is realized by event handling, too. Therefore, Céu internally implements its own *wall clock* that accounts for the passage of time. Occasionally, it is advanced by the environment through special wall clock input events which provide the amount of elapsed physical time as payload. Based on this, each Céu program behaves as follows [San13]:

1. On system start, the program performs the boot reaction in a single trail.
2. Active trails execute until they await an event or terminate. This step is always bounded in time and memory. The usage of **`par`**-blocks may spawn new trails.
3. The program goes idle; the environment takes control.
4. The occurrence of a new input event causes all trails awaiting that event to awake. Then, it goes to step 2.

Due to the synchronous hypothesis, a program conceptually takes no time on step 2 and is always idle on step 3. In practice, if a new input event arrives while a reaction is currently running (step 2), Céu demands to enqueue it to run in the next reaction. This relaxes the rigorous semantics of the synchronous hypothesis (see Section 2.2) where all computations are entirely performed before the next event occurs.

Whenever multiple trails are active in the same reaction, for example they have awaited the same event, Céu schedules them in the order they appear in the program text – at most one trail is running at any time. Also, Céu allows side-effects on shared variables across concurrent trails in the same reaction. Note that this implementation of the synchronous hypothesis is very Céu-specific and clearly deviates from the classical synchronous model. In Section 5.5, we will come back to this issue.

In order to illustrate Céu's scheduling strategy, the execution of the example code in Listing 2.1 is depicted in Figure 2.4 [San13]. The program performs the boot reaction and forks into three trails. Following the lexical order of their declarations, they execute as follows ($t0$ in Figure 2.4):

1. Trail 1 runs up to the **`await A`** (line 4).
2. Trail 2 runs up to the **`await B`** (line 8).
3. Trail 3 runs up to the **`await A`** (line 12).

```
1  input void A, B, C; // three external input events
2  par/and do          // trail 1
3    <...>             // <...> represents non-awaiting code
4    await A;
5    <...>
6  with                // trail 2
7    <...>
8    await B;
9    <...>
10 with                // trail 3
11   <...>
12   await A;
13   <...>
14   await B;
15   par/and do        // trail 3
16     <...>
17   with              // trail 4
18     <...>
19   end
20 end
```

Listing 2.1: A Céu program to illustrate program execution [San13, p. 27]

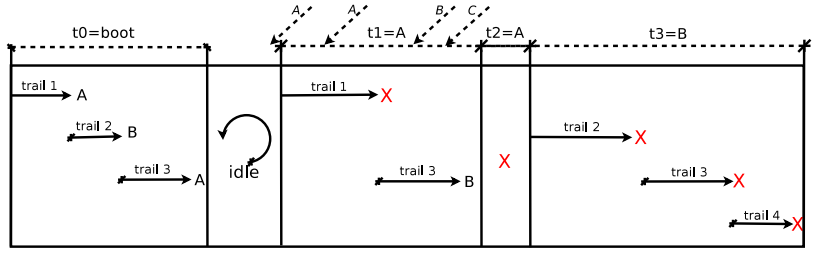

Figure 2.4: A sequence of reactions for the program in Listing 2.1 [San13, p. 28]

Since there are no other trails pending, the reaction terminates and the program remains idle until the input event `A` occurs ($t1$ in Figure 2.4):

1. Trail 1 awakes, runs and terminates (line 5).
2. Trail 2 remains suspended, as it is not awaiting `A`.
3. Trail 3 runs up to **`await`** `B` (line 14).

During the reaction $t1$, new occurrences of events `A`, `B` and `C` happen and are enqueued to be handled sequentially in the next reactions. Since `A` happened first, it is used in the next reaction. However, since no trail is awaiting it, an empty reaction is performed ($t2$ in Figure 2.4). The next reaction dequeues the event `B` ($t3$ in Figure 2.4):

1. Trail 2 awakes, runs and terminates.
2. Trail 3 forks in two and they both terminate immediately.

Finally, the **`par/and`** rejoins causing the whole program to terminate too. The pending occurrence of event `C` is discarded and does not trigger a reaction.

System Integration Céu depends on a host platform which emits the external input events and accepts the external output events. Independent of how Céu code is organized across source files, the Céu compiler always generates a single state

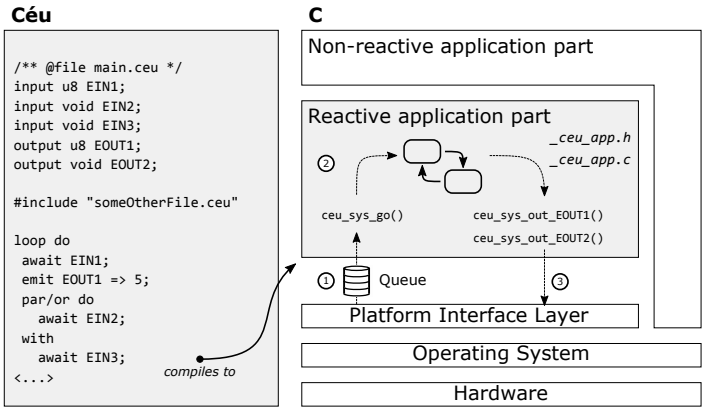

Figure 2.5: System integration of Céu code

machine implementation in C which must interface the host platform. Therefore, as depicted in Figure 2.5, the state machine provides an interface function `ceu_sys_go` for all input events, for example `EIN1`, and a handler function for each output event, for example `ceu_sys_output_EOUT1` in case of `EOUT1`. To actually realize the event handshakes with the host platform, Céu requires a thin platform interface layer in C– a one-time implementation overhead. If the business logic complexity grows, the platform interface code keeps constant.

Input events are triggered by the host platform and buffered into a queue (1). They are dequeued sequentially and passed to `ceu_sys_go`. This performs the reaction and gradually advances the state machine (2). An output event causes the state machine to execute the corresponding handler function holding the assigned C platform interface code (3).

3 Identification of Reactive Concerns

The deployment of synchronous programming is a fundamental design decision which must be considered at the very first beginning of software development. This is due to the fact that it determines the fundamental model of computation. An inappropriate computation model can considerably complicate software engineering (see Chapter 4) and hence should be carefully selected according to the domain-specific problems to solve. In this chapter, we consider the gateway's different problem domains and the characteristics of their respective computations. We aim to identify those parts of the entire application that are exposed to reactive concerns and hence are suitable for deploying the synchronous paradigm.

3.1 Exposed Problem Domains

Figure 3.1 presents a top-level view of the gateway's system architecture. Services and parameters of the heating domain are mapped to *resources* of a REpresentational State Transfer (REST)ful application programming interface. One resource, for example, is the set point for the room temperature. An embedded web server accepts GET and PUT requests via HTTP in order to perform read and write accesses on that resources. By this, the gateway actually allows the user to interact with his local

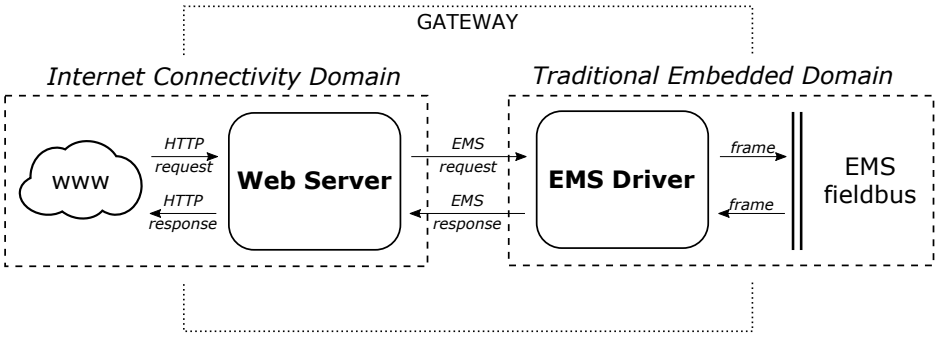

Figure 3.1: Top-level view of the gateway's system architecture

heating appliance by remote. He may use any internet-ready mobile device world-wide to send, for example, a PUT request which increases the set point, thereby adapting the current room temperature at home. Resources that cover services provided by the gateway itself, for example the current values of its analogue inputs, the firmware version or the logged history data, are called *internal resources*. They are stored in the gateway's internal Random Access Memory (RAM), Read Only Memory (ROM) or

flash memory. In contrast, configuration parameters or monitor values of the heating appliance, for example the current room temperature, are provided by the surrounding heating ecosystem and denoted as *external resources*. While internal resources can be directly read or written by the gateway itself, external ones must be indirectly accessed using the EMS fieldbus (see Section 2.1). A dedicated fieldbus driver software is used by the web server to actually communicate with the heating appliance. Incoming HTTP requests are translated by the web server into EMS requests which are delegated to the fieldbus driver. The latter finally exchanges frames with the fieldbus in order to process them. Above functionality requires the gateway to deal with Internet and fieldbus communication at the same time. It appears that both belong to very different problem domains that generally place different demands on software development.

The Internet communication is part of the *information technology domain*. It typically comprises a high level of abstraction based on client-server architectures where data modeling and processing is paramount. Safety is usually associated to data rather than functions. This leads to the notion of (data) security which covers, for instance, approaches for cryptography, authentication and authorization. Due to short product life cycles and newly emerging technologies, the environment of Internet applications is highly dynamic and hence requires recurring updates. Software development usually favors high reuse of existing standard solutions, for example network stacks, cryptography libraries or web frameworks. Generally, the Internet connectivity domain relies on massively parallel computations which are not time-critical and do not require deterministic execution.

The fieldbus communication belongs to the *embedded domain*. In contrast, it imposes a low level of abstraction that requires to deal with hardware architectures and physics. Functional safety is mandatory, particularly in safety-critical systems. Data is usually transferred in clear text. The environment of embedded applications is often static for a longer period of time. A heating appliance, for example, typically runs for at least 10 to 15 years without any modifications. Embedded applications almost always require customized and special software solutions which rely on company-specific intellectual property. The embedded domain is typically based on time-critical computations that demand deterministic execution.

3.2 Domain-Specific Computation Characteristics

In the following, we consider the requirements of the domain-specific computations in more detail.

Internet Communication (Information Technology Domain) Every incoming GET or PUT request is handled by the web server for processing. In particular, the following steps are required: The request has to be (1) received, (2) decrypted, (3) parsed, (4) interpreted and (5) executed. Subsequently, the response must be (6) built, (7) encrypted and finally (8) transmitted. Internet connectivity is highly concurrent. Several client devices may be connected to the gateway simultaneously. As a consequence, the web

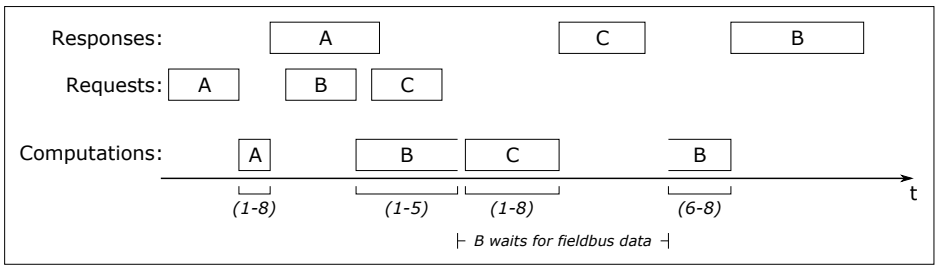

Figure 3.2: Computations for Internet communication

server may have to accept and deal with several requests concurrently. Due to HTTP, the individual requests are inherently independent. They do not share any global data or state nor do they demand any communication or synchronization among them. Processing of steps 1 to 8 does never induce any direct side effects across requests. A single request is completely isolated from the others.

Figure 3.2 illustrates the processing of three requests `A` to `C`. Some requests, for instance `A`, can be answered instantaneously, in negligible time, that is. For example, this is true for internal resources that are located in the gateway's RAM and ROM memory. Requests for external resources, for instance `B`, in contrast, involve fieldbus communication. This adds delays due to the medium access protocol and response times of other bus participants. Moreover, de- and encryption are computation-intensive tasks. Depending on the size of response data, the required processing time can additionally increase. Thus, depending on the concrete resource, answering a GET or PUT request may generally last from just a few milliseconds up to several seconds. However, in case of several concurrent requests, for example `B` and `C`, long-lasting ones such as (`B`) must not block those that can be served quickly (`C`). Otherwise requests from one device can considerably delay responses to another device, thereby introducing a time-related dependency. Strict sequential request processing is consequently not possible. Instead, a concurrent execution is required in order to allow requests to overtake each other. For example, `B` cannot proceed because it is waiting for fieldbus data. Thus, `C` is answered meanwhile. Once the data are available, processing of `B` continues. In favor of short response times this approach is not deterministic. The order in which concurrent requests interleave their processing depends on several parameters such as the inquired resource, the current internal state of the web sever and the underlying scheduling policy.

Furthermore, an individual HTTP request is not subject to any real-time requirements. The response is transmitted by the web server as soon as the data are ready. Client devices and the user respectively have to wait for the feedback accordingly. By this, the processing unit – the web server– determines the operating speed.

Fieldbus Communication (Embedded Domain) Fieldbus participants follow a request-response communication discipline based on frames in order to exchange information (see Section 2.1). For frame processing the fieldbus driver performs the following steps: An incoming frame has to be (1) received, (2) parsed (3) interpreted

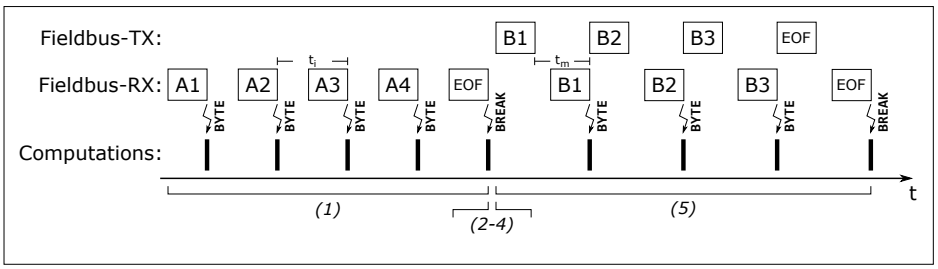

Figure 3.3: Computations for fieldbus communication

and (4) executed. Depending on whether the gateway is currently in Passive or Active Mode, execution requires to either store the provided data or build a response frame. In the latter case, the response needs to be (5) transmitted. Fieldbus communication does not use any cryptography. Due to the static, predefined frame structure parsing and interpreting rely on simple comparisons and conditional code execution. Storing payload data and building the response frame involve only access to local gateway memory and do not require any computation-intensive algorithms. Thus, steps 2 and 3 can be seen as instantaneous. Due the baud rate of the fieldbus, frame reception and frame transmission, in contrast, last up to several hundred milliseconds depending on the frame length.

Figure 3.3 illustrates a request-response communication scenario where request frame `A` is answered by response frame `B`. Frame reception (1) requires to merge the stream of incoming single bytes into a complete frame. Each byte is checked for content and time accuracy. The content may be erroneous due to collisions or disturbing signals on the fieldbus. In addition, the idle time t_i between two consecutive bytes belonging to the same frame must not exceed the upper limit of 350 ms. Content and time valid bytes are stored, invalid ones are discarded. Once the end-of-frame character has been received, the capture completes and the stored frame is interpreted (2). Subsequently, it is executed (3) and may trigger a frame transmission in response, for example because another device requested a gateway-internal resource. In contrast to Internet communication, the inquiring device demands a response within 225 ms. If the limit is exceeded the current request-response cycle will fail. Frame transmission (4) requires to send a single frame byte-wise onto the bus. Due to the physical transmission method, each byte is mirrored by the master device. Before proceeding with the next byte transmission, the content and time accuracy of the mirror byte is checked too. The mirror time t_m must not exceed an upper limit of 42 ms. If the mirror is invalid the transmission fails and is aborted immediately.

As depicted in Figure 3.3, the long-lasting actions of frame reception (1) and transmission (4) are composed of simple reactions to the incoming bytes. For each one, the fieldbus driver only has to decide whether to store or discard it. For a complete frame, the fieldbus driver decides whether a transmission is required in response or not. All those reactions as well as steps 2 and 3 rely on simple comparisons and conditional code execution without any computation-intensive tasks. Thus, all computations involved in fieldbus communication can be seen as instantaneous. However, in contrast to

GET and PUT requests, the reaction to the current byte may influence reactions to future bytes. For example, a malformed frame byte causes the remaining, future frame bytes to be discarded since the whole frame is already known to be invalid. Thus, there is a dependency across byte reactions which requires a global state. Also, bytes must be deterministically processed in the same sequential order they are received. Furthermore, note that above time limits impose soft real-time requirements on fieldbus communication. If a communication cycle fails this does never cause any harm. It leads to a corresponding retry in future. However, in order to provide a high quality of service fieldbus access should usually succeed. Thus, timeliness is important.

To sum it up, fieldbus communication is characterized by simple *switching behavior*. That means that instantaneous reactions successively advance the fieldbus driver's internal state depending on the incoming characters. There are no computation-intensive tasks involved. Each character receipt can be seen as an external fieldbus event (see Figure 3.3). In particular, fieldbus communication is based on the following reception events:

(1) `BYTE`, indicates a frame byte with valid content.

(2) `BREAK`, indicates an end-of-frame character.

(3) `ERROR`, indicates a character with malformed content.

Note that information on timing accuracy is not provided on event basis and requires additional care.

3.3 Conclusion

Although Internet and fieldbus communication both rely on a request-response discipline, they place very different demands on the underlying model of computation. In the Internet connectivity domain, computations are inherently independent and stateless. Concurrent computations interleave their execution in a non-deterministic order. Furthermore, they may last for a longer period of time and are never subject to hard timing constraints. In contrast, the fieldbus communication requires to perform reactions on event occurrence. These reactions comprise a high interdependency and hence require global state. Reactions have to be provided in a strict sequential, deterministic order. Due to their negligible processing time, they can be seen as instantaneous computations which are subject to real-time requirements.

Harel and Pnueli [HP85] introduce a wording for this kind of dichotomy: *transformational* and *reactive* systems. An Internet communication application has a transformational character. It accepts plain text in the HTTP request format as input, transforms it, for example into a binary representation that can be handled by the fieldbus driver and vice versa, and produces plain text in the HTTP response format as output. The fieldbus driver software, on the contrary, is repeatedly prompted by the fieldbus and has to continuously react to the external, incoming characters. By this, it maintains an

ongoing interaction with its physical environment based on events. With respect to Section 2.2, the following becomes apparent:

First, thread-based, sequential programming which relies on asynchronous execution seems to be particularly suitable for the transformational Internet communication. In the traditional multi-threaded approach each incoming HTTP request is processed by a dedicated worker thread typically taken from a thread pool. Second, event-based, reactive programming which relies on synchronous execution seems to fit the reactive fieldbus communication. Third, deploying synchronous programming for Internet communication too is not expected to provide any software engineering benefits. On the contrary, it seems reasonable, that, in an approach that relies on a single model of computation, only part of the domain-specific problems is easy to solve. The remainder must be tackled using a technology that lacks appropriate support. In Chapter 4, we elaborate the corresponding engineering and quality implications in more detail. Fourth, it seems that, in a perfect world, an uncomplicated implementation that tackles such oppositional problem domains relies on several, appropriate models of computation. However, this is generally impossible in today's C-dominated, monolingual approaches. On this account, we propose a multilingual software architecture in Section 5.1.1.

Finally, our work deliberately deploys and evaluates the synchronous paradigm for the reactive fieldbus driver only.

Note on State-of-the-art Internet Technology Some of today's Internet connectivity applications escape from the traditional multi-threaded approach due to its poor efficiency and scalability when dealing with thousands of concurrent requests at the same time. Instead, they favor event-driven alternatives, for example *node.js* [TV10], where a single thread handles all incoming requests simultaneously by taking advantage of non-blocking, asynchronous Input/Output (I/O) operations. This introduces the ambiguous term *reactivity* to the information technology domain and hence generally relativizes its transformational character. However, reactive web solutions are typically not applicable and also not needed in resource-constrained embedded systems. Embedded Internet applications are usually not exposed to massive parallelism – they hardly have to deal with more than a handful of concurrent requests at the same time. The gateway application, for instance, supports a maximum of three concurrent requests only and hence favors the simplicity of the traditional multi-threaded model. For this reason, in the embedded domain, we still consider Internet communication to be transformational rather than reactive.

4 Analysis of the Existing Asynchronous Implementation

The existing gateway software is entirely written in C. This is true for Internet and fieldbus communication likewise. Features of an embedded operating system expand C's conventional sequential tool set by thread-like, concurrent, asynchronous execution. However, in Chapter 3, we considered this model of computation to be particularly inappropriate for implementing inherently reactive concerns such as the fieldbus driver.

In this chapter, we justify our considerations by elaborating the induced software engineering challenges and drawbacks as well as their manifestation in the existing fieldbus driver code base. Our results serve as the baseline for the comparative evaluation of the event-based, reactive, synchronous paradigm in Chapters 5 and 6 respectively.

First, in Section 4.1, we provide an overview about the fieldbus driver's fundamental architecture, functionalities and underlying technologies. Second, in Sections 4.2 to 4.4, we investigate its implementation, thereby focusing on the following aspects: (a) control and handling of events, (b) concurrency and (c) temporal behavior. Those concerns are intrinsic to reactive, physical systems [HP85; Lee05] and constitute key issues in software design [BF14, ch. 2]. Finally, in Section 4.5, we conclude our results.

4.1 Overview

Figure 4.1 illustrates the architectural design of the fieldbus driver. It is composed of three layers – byte, frame and data – that handle different concerns with respect to the EMS communication protocol. We assume, for simplicity, that each layer resides in a single source file, for example *ems_byte_layer.c*, although this is usually not the case. For event processing, the byte and the frame layer are centered around the state machines `blState` and `flState` respectively. In addition, they interact with a set of timers in order to realize temporal behavior. Data exchange between layers relies on shared memory represented by the flat array `buffer` and the queue `requests`. Byte layer code is mainly executed in the interrupt service routines `RxChar` and `TxChar`, frame layer code in cyclic, high frequency task $Task_{High}$ ($\mathcal{T}_H$) and data layer in cyclic, low and medium frequency tasks $Task_{Low}$ ($\mathcal{T}_L$) and $Task_{Medium}$ ($\mathcal{T}_M$).

4.1.1 Reactive Functionalities

Physically interfacing the fieldbus is done by a serial communication device in hardware. It independently manages the bit-wise reception and transmission of bytes from and to

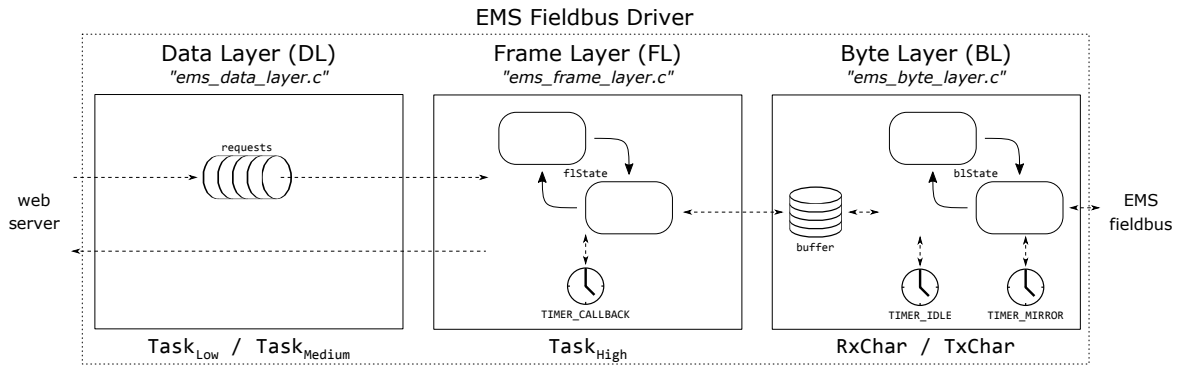

Figure 4.1: Overview of the fieldbus driver software architecture in C

the fieldbus respectively. In this context, it also provides the three fundamental, external reception events `BYTE`, `BREAK` and `ERROR` described in Section 3.2. By this, the serial communication device actually realizes the reactive interface to the fieldbus driver's physical environment. The three software layers implement a set of functionalities that process above events. In general, we will refer to functionalities that involve event handling as *reactive functionalities*.

Byte Layer (BL) This layer manages the byte-wise reception and transmission of frames. Therefore, it implements the reactive functionalities *single frame reception* ($\texttt{Receive}_{\texttt{BL}}$) and *single frame transmission* ($\texttt{Transmit}_{\texttt{BL}}$). Both are always mutually exclusive and can be divided into consecutive reactions to above events. $\texttt{Receive}_{\texttt{BL}}$– the default operation – merges the stream of incoming single bytes into a complete frame. $\texttt{Transmit}_{\texttt{BL}}$ decomposes a given frame into single bytes and triggers their transmission. The byte layer does not interpret frame content in any way. The essential steps for $\texttt{Receive}_{\texttt{BL}}$ and $\texttt{Transmit}_{\texttt{BL}}$ are:

$\texttt{Receive}_{\texttt{BL}}$	$\texttt{Transmit}_{\texttt{BL}}$
1. Capture frame start	1. Transmit frame bytes
2. Capture remaining frame bytes	2. Transmit end-of-frame character

Frame Layer (FL) This layer determines whether the device currently operates in Passive or Active Mode (see Section 2.1) and implements the respective behavior. Therefore, it realizes the reactive functionalities *respond to request frame* ($\texttt{Respond}_{\texttt{FL}}$) and *send request frame* ($\texttt{Request}_{\texttt{FL}}$). Both take advantage of above byte layer functionalities. $\texttt{Respond}_{\texttt{FL}}$– the default operation – is performed in Passive Mode. It accepts an incoming request frame from another device and provides a proper acknowledgment. $\texttt{Request}_{\texttt{FL}}$ is performed in Active Mode. It sends a request frame to another device and awaits the corresponding feedback. Both functionalities interpret the frame content to distinguish between read request, read response, write request and write response frames. The essential steps for $\texttt{Respond}_{\texttt{FL}}$ and $\texttt{Request}_{\texttt{FL}}$ are:

$\texttt{Respond}_{\texttt{FL}}$	$\texttt{Request}_{\texttt{FL}}$
1. Receive request frame	1. Get next request frame from queue `requests`
2. Process request frame	2. Transmit request frame
3. Build response frame	3. Wait for response frame
4. Transmit response frame	4. Receive response frame
	5. Process response frame

Data Layer (DL) This layer interfaces between the fieldbus and the remaining application. It accepts EMS requests from the web server, for instance, and translates them into corresponding request frames. A single EMS request may lead to several request frames. Request frames can only be transmitted during Active Mode (by $\texttt{Request}_{\texttt{FL}}$). Thus, the data layer buffers them into the dedicated queue `requests`. Also, it provides EMS responses back to the web server.

4.1.2 Underlying Technologies

In the domain of embedded systems a number of practical reasons [Tan12] leads to the choice for the programming language C [KR78]. It provides imperative, sequential programming which relies on a single-threaded control flow. Its basic set of flow control mechanisms covers the sequential, branch and iteration structures [Hoa+87]. Reactive concerns, however, require additional support for control and handling of events, concurrency and temporal behavior. Those features are not provided by sequential programming. On this account, the sequential tool set is extended by usage of an "Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen (OSEK)/Vehicle Distributed eXecutive (VDX)" [ISO05] compliant operating system. The deliberate choice for OSEK responds to the strong resource limitations (512 KB ROM, 124 KB RAM, 48 MHz) imposed by the gateway's hardware platform.

Operation Purpose and Requirements The OSEK/VDX operating system standard [ISO05] defines a single-core operating system that aims to provide a uniform environment for automotive control unit software. Accordingly, its design responds to demands emerging from the automotive domain. Those applications require to handle several independent concerns simultaneously. Most of them are characterized by computation-intensive control tasks. That is, a set of control parameters is continuously adapted over time according to changes in the system's physical environment. Therefore, several differential equations are periodically recalculated. A single calculation cycle requires negligible processor time since there are no long-lasting and complex algorithms involved. However, due to the high update frequency, for example 500 recalculations per second, control tasks finally lead to high processing load. As a consequence, automotive applications typically favor a time-triggered execution policy based on fixed periods. While mimicking the computation behavior of control tasks this also improves timing predictability. Due to the safety-critical context, the latter is mandatory to meet hard real-time requirements and achieve reliability. In order to tackle cost sensitivity, automotive implementations are additionally subject to stringent

resource requirements. Thus, the OSEK system design consistently aims at a minimum utilization of RAM, ROM and processor time making it feasible even for low-end 8-bit microprocessors at a code footprint of 1 to 10 kilobytes [Mar11, p. 191]. With respect to reactive concerns, the OSEK standard contributes features to the following service groups: (1) task management, (2) synchronization, (3) interrupt management and (4) alarms.

Support for Control and Handling of Events OSEK promotes an efficient, event-driven model of execution based on the notion of *tasks* and *interrupt service routines*. Both entities allow to perform application code on event occurrence. A task holds a sequence of event handling instructions which can be activated for execution one or several times during run-time. Each activation spawns a new execution instance which obtains a new run-time context at the beginning of execution time and discards it on completion. Activation can be done by calling a system service or by alarm expiration (see below). A task's life cycle is generally determined by the state model depicted in Figure 4.2. While being *suspended* a task is deactivated and cannot run. A *ready* task has been activated and waits for allocation of the processor. During *running* state, the task's instructions are actually performed. Finally, an event mechanism provides an additional *waiting* state. The latter causes a task's control flow to block until a predefined event occurs.

However, management of tasks with waiting states is, in principle, more complex and hence requires more system resources [ISO05, p. 16]. In particular, waiting tasks do not drop their run-time context since they have not yet run to completion. Unable to continue execution, they remain in the system stack for a potentially considerable amount of time [Mar11, p. 192]. Thus, separate stack space is mandatory for each task which increases RAM usage. On this account, OSEK distinguishes between full-featured *extended tasks* and limited, resource-saving *basic tasks*. As illustrated in Figure 4.2, basic tasks share the same life cycle except the waiting state. Specifically, they cannot block their control flow for awaiting events. Thus, the event mechanism is not available for basic tasks. This deliberate limitation reduces RAM utilization by enabling *stack sharing* across tasks [Mar11, p. 192]. In order to allow a high reuse of system resources and meet real time requirements, basic tasks demand a run-to-completion semantic. That is, after activation they are intended to terminate and release their resources as fast as possible – long-lasting tasks are generally discouraged.

> "In this way, the [basic] task behaves like a function, which allocates a frame on the stack, runs, and then cleans the frame." [Mar11, p. 193]

With respect to their event handling semantics, interrupt service routines behave similar to basic tasks. However, since interrupts are reserved for highly time-critical reactions, the need for negligible execution time is of great importance. While basic tasks are generally intended for recurring events and cyclic polling or updating, interrupt service routines particularly address sporadic reactions.

Due to the gateway's strong resource limitations, the memory overhead of extended tasks is not tolerable. Thus, the implementation is restricted to efficient basic tasks

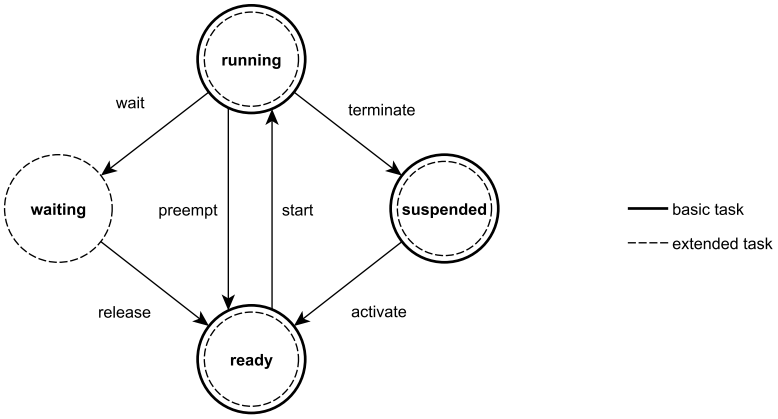

Figure 4.2: OSEK task state model [ISO05, p. 17–18]

and interrupt service routines. To sum it up, their execution instances conceptually spawn a new trail (see Section 2.3) which is subject to the following constraints:

Constraint 1: It obtains a new run time context at start of execution time and discards it on completion.

Constraint 2: It must run to completion as fast as possible; long-lasting computations are forbidden.

Constraint 3: It cannot block its control flow waiting for an event to occur (as a consequence of Constraint 2).

We will further denote this provided model of execution by *one-shot trails*.

Support for Concurrency In OSEK, tasks and interrupt service routines are concurrent units of execution. The OSEK operating system allows several execution instances of tasks or interrupt service routines to coexist during runtime. If multiple tasks are in the ready state at the same time, they are competing for processor time. In this case, the OSEK operating system software performs asynchronous, preemptive scheduling. That is, the scheduler organizes the sequence in which concurrent tasks interleave their execution. Related preemption decisions are based on static, user assigned task priorities.

Inter-task synchronization is explicit and provided by two means. On the one hand, resource management takes advantage of semaphores in order to coordinate access to shared resources. A priority ceiling protocol avoids priority inversion. On the other hand, event control enables a task to suspend execution until an event occurs. This mechanism can be effectively used to synchronize the control flow between multiple concurrent tasks. However, as mentioned above, event-based synchronization is reserved to extended tasks only. Consequently, except for semaphores, basic tasks only comprise synchronization points at the beginning and end of each execution instance.

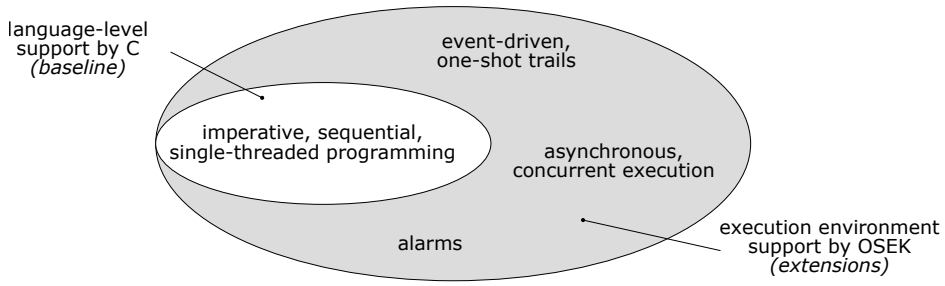

Figure 4.3: Underlying technologies provided by C and OSEK

Interrupt service routines have priority over tasks. Their scheduling is performed by hardware and not covered by the OSEK specification.

Support for Temporal Behavior Interfacing with physical time is provided by the notion of *alarms* in OSEK. Alarm management is a service for processing recurring events. The basic idea is to count the occurrences of a certain event and automatically trigger a notification – called alarm – as soon as a predefined counter value is reached. This approach is based on an implementation-specific counter, managed by the operating system, that is successively incremented on each event occurrence. Event source may be, in principle, any application-specific trigger, for example an interrupt of a serial communication device in order to notify about received messages. However, to interface with the time domain, OSEK demands at least one counter that is sourced by a timer. The latter is configured to generate an event at regular time intervals depending on the required resolution. Based on this master clock, the counter measures the passage of time according to the timer's frequency.

Alarms can be defined to expire once (*single alarm*) or periodically (*cyclic alarm*). Their expiration may cause the operating system to (1) activate a task, (2) execute a callback function or (3) set an event. Option 3, however, is only provided for extended tasks. The notion of alarms essentially maps the passage of physical time to events. By this, it provides an effective link between the physical time domain and OSEK's event-driven execution model. The combination of cyclic alarms which periodically activate basic tasks particularly responds to the time-triggered execution policy of short-lived computation tasks.

Summary To sum it up, Figure 4.3 illustrates the underlying technologies in use. C provides (1) imperative, sequential, single-threaded programming on language-level. OSEK extends this baseline by (2) event-driven, one-shot trails, (3) asynchronous, concurrent execution and (4) alarms.

4.2 Control and Handling of Events

Events indicate that something notable has happened in the system's physical environment. Some of them require a proper response. Performing such kind of system

(s)eq./(c)onc.	s	s	s	s	s	s	s	s	c	c	c	c	c	c	c	c
(o)ne/(m) loc.	o	o	o	o	m	m	m	m	o	o	o	o	m	m	m	m
(p)ass./(a)ct.	a	a	p	p	a	a	p	p	a	a	p	p	a	a	p	p
(b)l./(n)on-bl.	b	n	b	n	b	n	b	n	b	n	b	n	b	n	b	n
Strategy	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Table 4.1: Combination possibilities for event handling

reaction is known as event handling. In order to do that, two essential steps are required. First, the system must observe the event. This is done by awaiting its occurrence somewhere in the application code. Second, once the event has been detected, the proper reaction code must be executed. This basic procedure is common for all reactive applications independent of their concrete hard- and software circumstances. However, corresponding implementations may follow quite different strategies and hence impose different implications on software engineering and quality.

In the following, we explain the fundamental event handling strategy adopted by the existing fieldbus driver software and how it manifests in the code. Subsequently, we provide a qualitative discussion.

4.2.1 Fundamental Strategy

Based on the technologies presented in Section 4.1.2, we explore the solution space for implementing event handling. Different strategies are generally imaginable. First, an event can be processed sequentially in a single trail or concurrently in several trails (see Section 2.2). Second, as depicted in Figure 4.4, in each trail it can be awaited at one or multiple code locations. Third, checking for an event occurrence can be done using either a polling (active) or an inversion of control (passive) approach. Fourth, the control flow of the corresponding trail may or may not block until the event occurs. Table 4.1 presents the resulting set of all 16 possible combinations. However, due to the technologies in use, only a subset is usefully applicable.

Strategies 9 to 16 Handling a certain event in several trails considerably increases implementation complexity. In particular, due to the asynchronous, independent execution model, the developer is faced with the burden to globally coordinate event processing across trails. On the one hand, trails require to have a common understanding of when the event has occurred in the system and is ready to

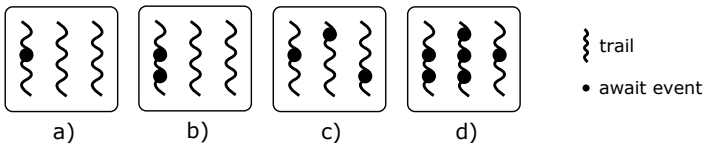

Figure 4.4: Locations awaiting an event: a) One trail, one location. b) One trail, several locations. c) Several trails, one location. d) Several trails, several locations

get handled (start of reaction). On the other hand, there has to be a common understanding of when processing has been completed by all trails involved (end of reaction). Thus, at least two synchronization points are mandatory to realize an event reaction. However, due to asynchronous trail execution, additional effort is required in order to explicitly implement such kind of inter-trail synchronization. Actually, an application specific event handling management logic is required for global coordination. Moreover, the developer has to define the order in which involved trails execute, for example in an interleaving or sequential fashion. Apart from event handling management logic, the developer has to take care of access to shared resources in order to prevent race conditions. Therefore, synchronization mechanisms with all their disadvantages (see Section 4.3) have to be used.

Strategies 6, 8 We consider waiting at multiple code locations within a trail to be useful only if a sequential event processing is intended. That is, at a certain location we wait for a specific event to occur, then execute the code below for realizing the corresponding reaction. Subsequently, we wait at the next location and so on. Obviously, this approach requires to block the control flow of the corresponding trail. This is in contradiction to Constraint 2 and 3 of one-shot trails. Consequently, we consider strategies 6 and 8 to be not usefully applicable in practice.

Strategies 1, 3, 5, 7 Blocking approaches are in contradiction to Constraint 2 and 3 of one-shot trails and hence have to be discarded.

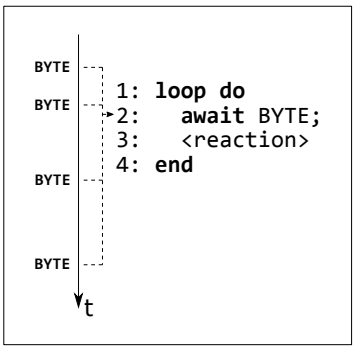

Figure 4.5: Programming scheme of the single entry point model

Finally, it appears that only strategies 2 and 4 remain. Figure 4.5 illustrates their common programming scheme. In the entire application there is solely one trail and code location awaiting BYTE in a non-blocking fashion (line 2). Irrespective of which strategy (active or passive) is eventually adopted, the assigned reaction code is executed subsequently (line 3). Once the reaction has been completed the procedure is repeated. Consequently, all occurrences of BYTE enter the same code location for getting processed. Thus, hereinafter we refer to this as the *single entry point model*. In fact, this is the usual approach adopted in resource-constrained embedded systems.

The single entry point model demands to map an event reaction to a trail in a one-to-one relationship. This leads to the traditional callback approach adopting the popular observer pattern [Gam+95]. However, callbacks are known to induce implementation effort on top of business logic. First, due to Constraint 2 and 3, each instance can only handle a single event occurrence. All occurrences enter application code at the same location. In order to individualize event reactions, for example treat the first occurrence of BYTE in a different way than the second one, *manual state management* [Kas07] is required. Second, Constraint 1 eliminates any event history across reactions. Consequently, application data and progress must be retained by *manual stack management* [Ady+02; Kas07].

4.2.2 Implementation Outline

In Figure 4.6, we provide an outline of today's byte layer implementation featuring $\texttt{Receive}_{\texttt{BL}}$ and $\texttt{Transmit}_{\texttt{BL}}$. It takes advantage of OSEK's interrupt service routines, thereby relying on the single entry point model. Arrows 1 to 8 illustrate the logical control flow performed for one run of $\texttt{Respond}_{\texttt{FL}}$. That is, the fieldbus driver first executes $\texttt{Receive}_{\texttt{BL}}$ to receive a frame and then $\texttt{Transmit}_{\texttt{BL}}$ to transmit the response. We assume the "good case" without any communication errors.

External fieldbus events BYTE, BREAK and ERROR enter application code via the common interrupt service routine `RxChar` (lines 4 to 50). In addition, interrupt service routine `TxChar` (lines 52 to 62) provides an internal event triggered on character transmission. Manual state and stack management are addressed by deploying a flat, handwritten state machine and global variables respectively. In particular, `buffer` (line 1) retains the last frame received or the next frame to send across event reactions while `blState` (line 2) stores the state machine's current state. In order to advance the state machine, `RxChar` first selects the current event (lines 6, 35 and 48), then the current state (lines 7 and 36) and finally the transition to take. Variable `notify` (line 3) aids as a flag to synchronize byte and frame layer control flow. In Section 4.3.1, we return to that in more detail. Note that `notify` actually resides in the frame layer but is made accessible using `extern`.

$\texttt{Respond}_{\texttt{FL}}$ starts with $\texttt{Receive}_{\texttt{BL}}$ and waits for the start of frame in `IDLE` (1). On arrival (line 8), the byte is stored (line 9) and the state is set to `RECEIVING` (line 10). The latter encodes that the frame start has been captures successfully. Control flow advances (2) to collect the remaining bytes (line 14). The receipt of the end-of-frame character causes the control flow to move (3) and state is changed to `COMPLETE` (line 38), thereby indicating that $\texttt{Receive}_{\texttt{BL}}$ has been successfully finalized. `notify` is set (line 39) to signal the completed frame capture to the frame layer.

Concurrently (not depicted in Figure 4.6), frame layer task $\mathcal{T}_H$ (see Section 4.1) monitors `notify` and executes in response. It parses the captured frame, builds the response and stores it to `buffer` (see Section 4.3.1). To start $\texttt{Transmit}_{\texttt{BL}}$, $\mathcal{T}_H$ triggers the transmission of the first response frame byte and forces `blState` into `TX_BYTE`. By this, control flow implicitly moves from `RxChar` to `TxChar` (4). This triggers a ping-pong procedure between both interrupt service routines. Once byte transmission completed,

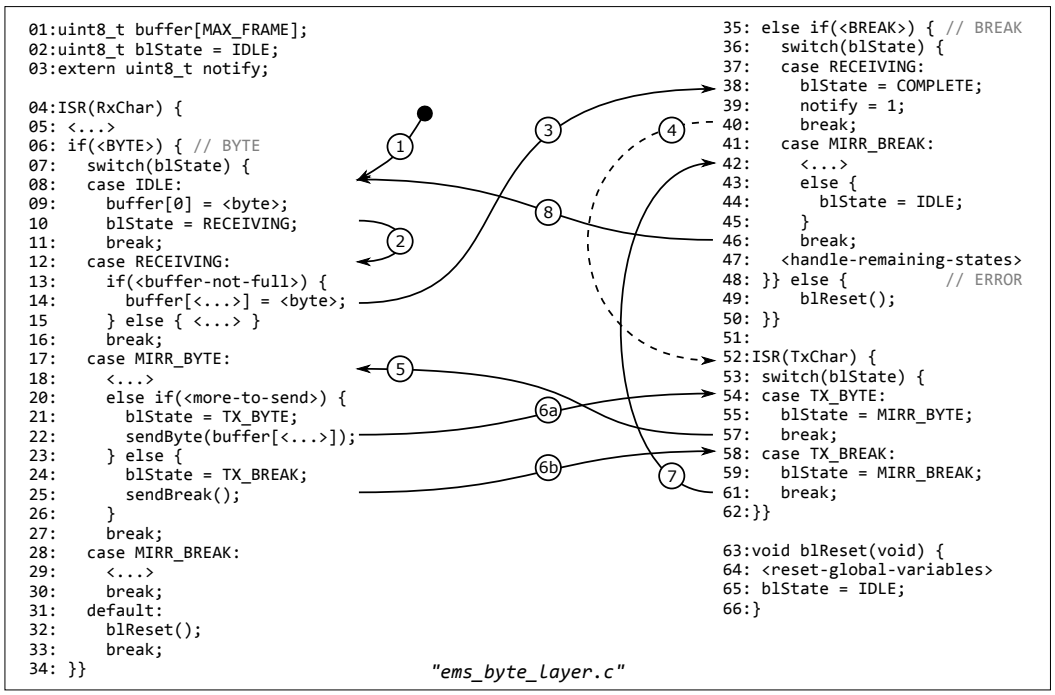

Figure 4.6: Outline of the byte layer implementation featuring $\texttt{Receive}_{\texttt{BL}}$ and $\texttt{Transmit}_{\texttt{BL}}$ (arrows 1 to 8 depict the logical control flow performed for one run of $\texttt{Respond}_{\texttt{FL}}$)

`TxChar` advances `blState` to `MIRR_BYTE` (line 55) to await the mirror, thereby returning control flow to `RxChar` (5). If the response frame has not been transmitted completely yet (line 20) `blState` changes to `TX_BYTE` again (line 21) and the next byte is sent (line 22). This moves control flow back to `TxChar` (6a). If all frame bytes have been transmitted (line 23), the state is set to `TX_BRK` (line 24) instead and the end-of-frame character is transmitted (line 25). Control flow advances to `TxChar` (6b) causing `blState` to switch to `MIRR_BRK` (line 59). Control flow returns to `RxChar` awaiting the end-of-frame character mirror (7). On reception (line 41), $\texttt{Transmit}_{\texttt{BL}}$ and consequently $\texttt{Respond}_{\texttt{FL}}$ complete. Finally, the state machine returns to $\texttt{Receive}_{\texttt{BL}}$ (line 44) waiting for the next frame to arrive (8).

4.2.3 Discussion

Above state machine approach "generally leads to excellent and measurable performances; a reaction is a 'linear' piece of code (neither loop nor recursivity, no interrupt, no overhead due to process management), whose maximal execution time can be accurately bounded." [Hal93, p. 3] However, while automata are indispensable for many application areas, they seem to sacrifice software engineering principles in favor of addressing embedded constraints.

Sequential programming in C provides three layers of abstraction given by instructions, functions and source files. Figure 4.7 illustrates their adoption in above state

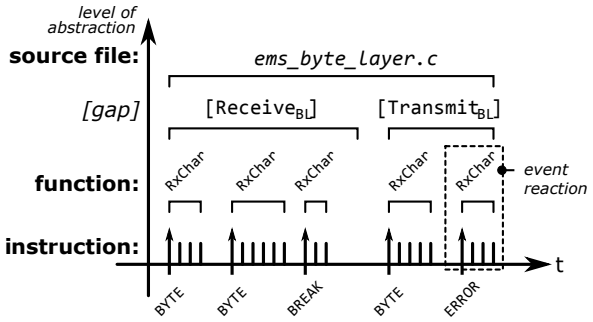

Figure 4.7: Mapping of C's abstraction ladder to the adopted event processing

machine approach. `RxChar` behaves like a function and abstracts over single event reactions while source file *ems_byte_layer.c* abstracts over all reactions performed for fieldbus communication. Reactive functionalities $\texttt{Receive}_{\texttt{BL}}$ and $\texttt{Transmit}_{\texttt{BL}}$ cover a respective subset of event reactions – more than one but less than all. However, it seems that sequential programming comprises an *abstraction gap* for event sequences. In particular, $\texttt{Receive}_{\texttt{BL}}$ and $\texttt{Transmit}_{\texttt{BL}}$ cannot be mapped to functions. Note that functions provide function-oriented software decomposition by design. This supports the deployment of most important software design principles, for example modularization, separation of concerns and encapsulation [BF14, ch. 2], on language level. Their inapplicability causes the loss of any above language-level support for implementing $\texttt{Receive}_{\texttt{BL}}$ and $\texttt{Transmit}_{\texttt{BL}}$ respectively. It appears, that this leads to a number of implications.

Torn and Convoluted Control Flow The logical control flow is inevitably torn across several function calls. This demands global variables and global state. The usage of global variables entails a well-known line-up of engineering disadvantages on its own [SK13]. Global `buffer`, for example, is not thread-safe but shared between byte and frame layer. In Section 4.3.1, we elaborate the implementation and mental effort required for synchronizing access to `buffer`. Furthermore, only detailed knowledge about the byte and frame layer state machine's switching logic reveals whether `buffer` is currently in use and, if so, whether it is used for a read or write operation. This makes it difficult to reason about its current state which is used and modified by a certain function, thereby generally complicating debugging and testing.

Apart from that, sequential programming lacks language-level support for implementing automata. The adopted workaround relies on arbitrary complex combinations of conditional code execution to successively filter for the transition to take. The original code of `RxChar` encompasses up to seven deep levels of nesting. Also, due to the sequential arrangement of conditional code blocks, the logical control flow seems to "jump" between branches across reactions (see arrows in Figure 4.6). Consequently, it appears that the workaround just relaunched low-level `GOTO`-like execution semantics into the high-level programming language C. The combination of a tangling control flow, deep nesting and global variables requires a great deal of mental effort to establish

the cognitive link between the technical EMS communication protocol specification, which favors timing diagrams, and its manifestation in the code. For example, mutual exclusion of $\texttt{Receive}_{\texttt{BL}}$ and $\texttt{Transmit}_{\texttt{BL}}$ is not obvious. Each functionality manifests in several states. State switches may, in principle, appear in any arbitrary order allowing $\texttt{Receive}_{\texttt{BL}}$ and $\texttt{Transmit}_{\texttt{BL}}$ to interleave.

Unsuitable Encapsulation In general, a single reactive functionality may rely on different events to fulfill its task while a single event may contribute to different reactive functionalities. Aspiring a function-oriented software design [BF14, ch. 2] reasonably requires code encapsulation *per functionality* as depicted in Figure 4.8. This abstracts all the code responsible for processing any event according to the current functionality to be performed. However, due to the single entry point model, functions in C only allow encapsulation *per event* which appears to be the orthogonal approach. This abstracts all the code responsible for processing a certain event depending on the current functionality to be performed.

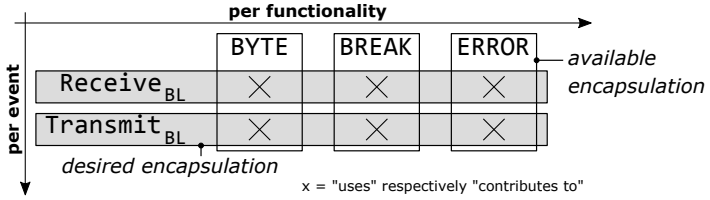

Figure 4.8: Orthogonal encapsulation approaches

Encapsulation on a per-event basis is generally a suitable approach in order to separate the complexity of a single event reaction into smaller auxiliary functions such as `blReset` (lines 63 to 66). This allows to shrink the overall size of the state machine implementation in `RxChar`, thereby improving readability and avoiding code redundancy, for example for default transitions. However, note that this scatters, at the same time, the state machine's switching logic across several functions making the effect and traceability of involved state transitions hard to understand.

Furthermore, it inevitably merges the concerns of different reactive functionalities. Thus, we cannot usefully break the complexity of fieldbus communication into separated, smaller entities. Eventually, this leads to a monolithic software design centered around a complex state machine in `RxChar` which features $\texttt{Receive}_{\texttt{BL}}$ and $\texttt{Transmit}_{\texttt{BL}}$ likewise. The lack of separation of concerns prevents functional isolation. This eliminates any option for independent and parallel programming of $\texttt{Receive}_{\texttt{BL}}$ and $\texttt{Transmit}_{\texttt{BL}}$ by different developers. Any change to $\texttt{Receive}_{\texttt{BL}}$ may potentially interfere with $\texttt{Transmit}_{\texttt{BL}}$ and vice versa. Note that this also affects testing. If one functionality is slightly modified, regression tests require to re-test the whole automaton. Also, an observed phenomenon cannot be easily assigned to either $\texttt{Receive}_{\texttt{BL}}$ or $\texttt{Transmit}_{\texttt{BL}}$ which complicates debugging.

Lack of Hierarchical Composition Conceptually, $\texttt{Respond}_{\texttt{FL}}$ and $\texttt{Request}_{\texttt{FL}}$ are composed reactive functionalities that rely on the byte layer. They determine execution start and termination of $\texttt{Receive}_{\texttt{BL}}$ and $\texttt{Transmit}_{\texttt{BL}}$ in order to fulfill their task. The latter both, in their turn, just provide a basic service without knowing anything about either $\texttt{Respond}_{\texttt{FL}}$ or $\texttt{Request}_{\texttt{FL}}$. This unidirectional dependency allows to structure software in a hierarchical fashion, thereby reducing the complexity on each abstraction level and promoting modularity. For non-reactive concerns, this is trivial to implement in C. Sequential programming allows to simply call a (sub-)function within another function, thereby supporting the unidirectional coupling between caller (for example frame layer) and called (for example byte layer) on language-level. This instantiates the called, passes the parameters, starts the called, waits for its termination and provides the return value to the caller. Also, if the caller terminates the called is automatically aborted too. However, due to the lack of a suitable encapsulation entity, this language-level feature is lost in the reactive domain.

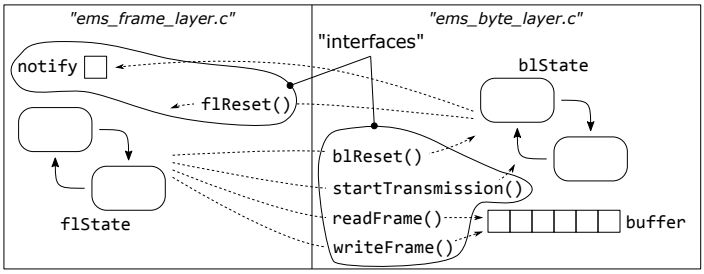

Figure 4.9: Interaction between byte and frame layer

As a result, Figure 4.9 reveals that the fieldbus driver implementation comprises a rather flat architecture. Instead of a hierarchical structuring, the byte and frame layers conceptually act on the same level. Their corresponding state machines run concurrently so that any hierarchical dependency must be implemented manually. This includes parameter passing, synchronization of control flows, reinitialization, execution of default transitions and propagation of termination. Therefore, both layers provide an "interface" based on a loose collection of functions which perform side effects on global variables. For example, there is no language support for letting a sequence of `RxChar` executions provide a return value, in order to notify about a successful frame capture. With respect to the byte layer, `readFrame` and `writeFrame` access `buffer` in order to exchange frames between the layers. `startTransmission` triggers $\texttt{Transmit}_{\texttt{BL}}$ by setting `blState` to `TX_BYTE`. `blReset` is used to reset the byte layer state machine in case of errors or termination of $\texttt{Respond}_{\texttt{FL}}$ and $\texttt{Request}_{\texttt{FL}}$ respectively. With respect to the frame layer, `notify` is used as a synchronization flag while `flReset` allows to reset the frame layer state machine in case of communication errors.

The lack of a well-defined, non-scattered interface makes it hard to delimit the layer implementations against each other. Due to the state machines mutual influences, $\texttt{Receive}_{\texttt{BL}}$ and $\texttt{Transmit}_{\texttt{BL}}$ seem to be intertwined with $\texttt{Respond}_{\texttt{FL}}$ and $\texttt{Request}_{\texttt{FL}}$. This generally complicates reuse and makes reasoning about how layers actually interact

during runtime a challenge. For example, it is not clear in which order function calls and accesses to global variables must be performed in order to make the approach work. There is no local, smooth, compiler-checked call which allows to hide all the complexity involved in layer interaction. Instead, each involved step, in principle, must be verbosely recorded, for example based on prose text or diagrams, making source code documentation a time-consuming and daunting task. However, it appears that – if at all – this is often not thoroughly done and hence may lead to a lack of understanding and incorrect usage. In Section 4.3.1, we exemplarily elaborate the required implementation and mental effort in more detail.

4.3 Concurrency

Concurrency is concerned with decomposing software into separated units of execution that run concurrently. As presented in Section 4.1.2, OSEK provides the notion of basic tasks and interrupt service routines on this purpose. The design for concurrency is usually motivated by two reasons [Hal93]. First, *physical concurrency* allows to increase performance or reliability by executing code on parallel or distributed hardware architectures. Second, *logical concurrency* provides a convenient and natural way to compose a system as a set of parallel, cooperating components. Logical and physical concurrency are not necessarily the same. Remember that OSEK is designed as a single-core operating system and that the gateway's hardware platform relies on a single-core processor. Thus, our application comprises only logical concurrency.

Specifically, the fieldbus driver is composed of the byte, frame and data layers which run concurrently. In order to fulfill their task, interaction between the layers is required. However, due to OSEK's asynchronous scheduling, their execution may interleave, in principle, in any non-deterministic order. Thus, the key challenge in asynchronous, concurrent programming is to guarantee correct interaction for all possible interleavings. Any order that is not explicitly ruled out is allowed. The developer is faced with the burden of pruning away the non-determinism. Therefore, points of synchronization between the concurrent execution entities must be introduced explicitly. Since most developers think sequentially, synchronizing concurrent code is difficult and error-prone [Lee05; Lee06; Lu+08]. Lu et al. [Lu+08] classify related concurrency issues in two main categories: *deadlock* and *non-deadlock* bugs. In our application, the only locking mechanism in use are OSEK's semaphores for resource management. They adopt the priority ceiling protocol [ISO05, p. 31] which prevents priority inversion and deadlocks by design. However, this deals with only part of the synchronization problem – issues not related to deadlocks still remain. According to Lu et al. [Lu+08], non-deadlock bugs usually manifest in two simple bug patterns:

Atomicity Violation "The desired serializability among multiple memory accesses is violated. (i.e. a code region is intended to be atomic, but the atomicity is not enforced during execution.)" [Lu+08, p. 332]

Order Violation "The desired order between two (groups of) memory accesses is flipped. (i.e. A should always be executed before B, but the order is not enforced during execution.)" [Lu+08, p. 332]

In the following, we present the strategies adopted in the fieldbus driver implementation to ensure the required atomicity and execution order. Subsequently, we provide a qualitative discussion. Therefore, we consider two different approaches. The first relies on manual synchronization without any OSEK support. The second takes advantage of OSEK's semaphores.

4.3.1 Synchronization without Operating System Support

While the byte layer code is executed sporadically in interrupt service routine `RxChar`, the frame layer code runs repeatedly in a cyclic high frequency basic task $\mathcal{T}_H$. Both concurrent units of execution perform cooperative interaction. This means that they voluntarily pass control among each other according to a common execution policy. $\texttt{Respond}_{\texttt{FL}}$ accepts a request frame captured by $\texttt{Receive}_{\texttt{BL}}$ and provides a response frame to be sent by $\texttt{Transmit}_{\texttt{BL}}$. $\texttt{Request}_{\texttt{FL}}$, in its turn, provides a request frame to be sent by $\texttt{Transmit}_{\texttt{BL}}$ and accepts a response frame captured by $\texttt{Receive}_{\texttt{BL}}$. Thus, for $\texttt{Respond}_{\texttt{FL}}$ the frame layer executes in response to the byte layer; for $\texttt{Request}_{\texttt{FL}}$ it is vice versa. Due to the strong resource limitations, data exchange is based on shared memory communication. The dedicated byte array `buffer` is used for passing a single frame up and down the layers. Byte and frame layer require to synchronize their control flow – $\texttt{Respond}_{\texttt{FL}}$, for example, must wait for $\texttt{Receive}_{\texttt{BL}}$ to complete – and access to the shared memory in order to avoid data races. In OSEK, semaphores prevent a task or interrupt service routine to enter the running state if the required resource, for example `buffer`, is currently locked. Thus, the fieldbus driver deliberately abstains from locking mechanisms in `RxChar` in order to keep it responsive.

Required Atomicity and Execution Order A frame read or write operation performed on `buffer` is composed of a sequence of single byte accesses. Each byte access is intrinsically atomic due to the microcontrollers hardware architecture. However, in between any two byte accesses frame read and write operations may interleave in any arbitrary order.

For $\texttt{Receive}_{\texttt{BL}}$ and $\texttt{Transmit}_{\texttt{BL}}$, a single frame read or write operation is spread across several execution instances of `RxChar`. For writing, this is due to the fact that bytes arrive successively over time and hence are stored through separated reactions (see Section 4.2.2). The same applies for reading due to the mirroring mechanism (see Section 2.1). Interrupt service routines have priority over tasks. However, in between any two consecutive runs of `RxChar`, frame layer code may access `buffer` and hence interleave the current read or write operation.

For $\texttt{Respond}_{\texttt{FL}}$ and $\texttt{Request}_{\texttt{FL}}$, a single frame read or write operation is performed in a single execution instance of $\mathcal{T}_H$ using a loop. However, due to the higher priority of byte layer code, access to `buffer` in the frame layer may be interleaved at any time too.

In order to ensure data consistency, frame read and write operations must be atomic. This means that once an operation on `buffer` has been started it must not interleave with another one. If frame reading and writing interleave, the read operation will return an incomplete, corrupted frame. If two writing operations interleave, the content of `buffer` depends on their execution order but generally leads to data loss and corruption. The only harmless interleaving is between two reading operations since they do not perform any side effects on the buffer. Finally, for atomicity, they have to follow a single-writer-multiple-readers policy [CHP71]. Note that any other code that does not access `buffer` may interleave without any implications.

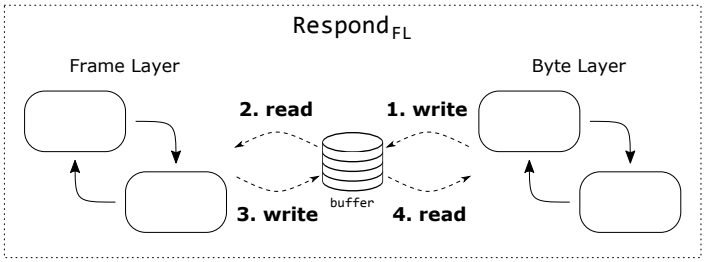

Figure 4.10: Order of accesses to `buffer` required for $\texttt{Receive}_{\texttt{BL}}$

Figure 4.10 illustrates the required execution order of read and write operations for $\texttt{Respond}_{\texttt{FL}}$. First, the byte layer performs a write operation to store the captured request frame (1). Second, the frame layer processes the frame, thereby performing a read operation (2). Third, the frame layer builds the response frame and performs a write operation to store it to `buffer` (3). Forth, the byte layer performs a read operation to transmit the response (4). For $\texttt{Request}_{\texttt{FL}}$, the order is essentially inverted. Thus, the application imposes an alternating, sequential execution semantic for read and write operations across byte and frame layer. This cooperative access policy automatically ensures atomicity.

Implementation Outline Synchronization for $\texttt{Respond}_{\texttt{FL}}$ and $\texttt{Request}_{\texttt{FL}}$ follows a similar approach. Thus, for simplicity, we exemplify $\texttt{Respond}_{\texttt{FL}}$ only. Figure 4.11 illustrates how above requirements manifest in the code. Accesses to `buffer` are highlighted bold.

At start of $\texttt{Respond}_{\texttt{FL}}$, the byte layer logically has access control for `buffer` and executes $\texttt{Receive}_{\texttt{BL}}$. Task $\mathcal{T}_H$ cyclically executes callback `taskHighCallback` (lines 4 to 8). The actual frame layer code which handles a captured frame is located in `handleFrame` (lines 10 to 19). The latter is only executed once the flag `notify` is set to `1` (line 5). Since `notify` is `0` by default (line 1), frame layer code is inactive and waits for $\texttt{Receive}_{\texttt{BL}}$ to complete.

The byte layer initiates the first write operation (1) in state `IDLE` by storing the frame start (line 28). Writing continues in `RECEIVING` by storing the remaining frame bytes (line 33). On end-of-frame character (line 46), the first write operation completes. Subsequently, the byte layer state machine switches to `COMPLETE` (line 49). In `COMPLETE`, any incoming byte is discarded (line 43). The byte layer cannot leave this state on its

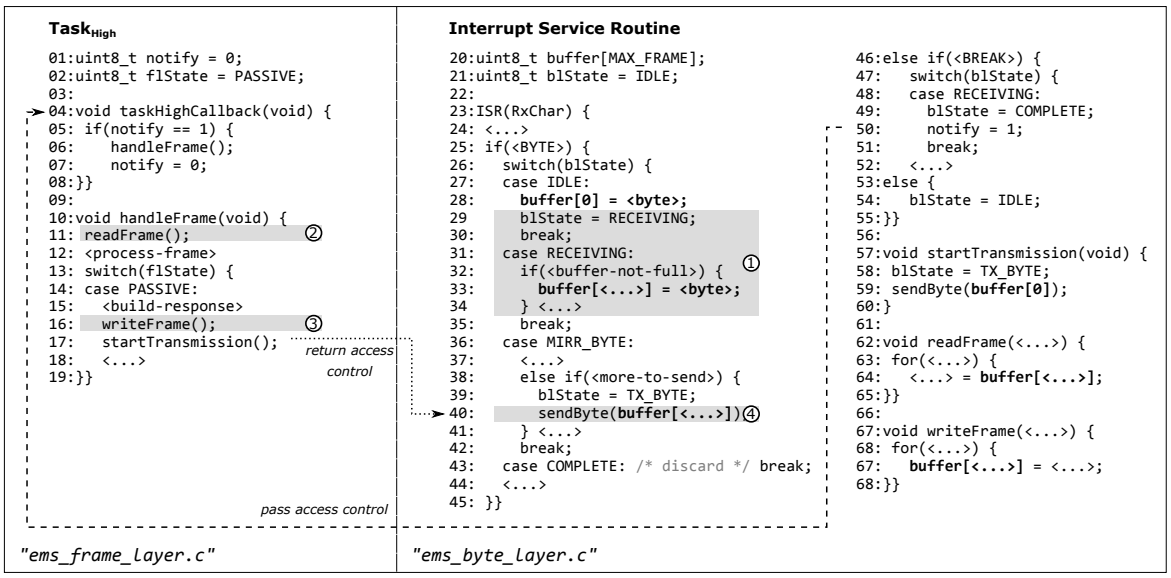

Figure 4.11: Synchronization without operating system support in C

own account. Instead, it requires the frame layer to switch the byte layers state. By this, the byte layer deactivates itself in order to prevent any accidental access to `buffer` while the frame layer code is running. At the same time, it sets `notify` in order to pass access control to the frame layer and activate it for frame processing (line 50).

The Frame layer, in its turn, actually executes `handleFrame` (line 6). In `handleFrame`, function `readFrame` (line 11) performs the first read operation (2) in order to copy the frame to a local buffer. Subsequently, the copied frame is processed (line 12) and the response frame built (line 15). Then, function `writeFrame` copies the response frame to `buffer` (line 16), thereby performing the second write operation (3). Finally, the frame layer calls `startTransmission` (line 17), thereby forcing the byte layer into `TX_BYTE` (line 58) and transmitting the first byte of `buffer` (line 59). This reactivates the byte layer state machine and initiates $\texttt{Transmit}_{\texttt{BL}}$, thereby returning access control back to the byte layer. Once `handleFrame` terminates, flag `notify` is reset (line 7). By this, the frame layer deactivates itself again. The byte layer performs the second read operation (4) to transmit the frame byte-wise onto the bus (line 40). On completion, the byte layer returns to `IDLE` respectively $\texttt{Receive}_{\texttt{BL}}$, thereby restarting the procedure.

Discussion Reasoning about synchronization in the manual approach generally requires a considerable amount of mental effort. This is due to the fact that synchronization does not manifest in a single code location. There is no concise language primitive, for example like `synchronized` in Java, that explicitly indicates a certain variable being thread-safe. Instead, the manual approach highly relies on implicit knowledge and is scattered across functions and source files:

1. The switching logic of the byte layer state machine ensures sequence (1,4).

2. The sequential execution of `readFrame` (line 11) and `writeFrame` (line 16) in $\mathcal{T}_H$ as well as the switching logic of `flState` ensure the sequence (2,3).

3. `notify` aids as a synchronization variable for the handshake between byte and frame layer (line 5), thereby ensuring the sequence (1,2).

4. `blState` aids as a synchronization variable for the handshake between frame and byte layer (line 17 respectively 58), thereby ensuring the sequence (3,4).

This makes it hard to identify and localize points of synchronization in the code and to grasp how they actually work. Solely reviewing the code of `handleFrame`, for example, does not reveal that calling `readFrame` (line 11) in that particular situation is safe. Quite the contrary, since the byte layer executes in `RxChar` which has priority over the frame layer's $\mathcal{T}_H$, it appears reasonable to assume that `readFrame` may be interrupted at any time. Only because we know that the byte layer's state machine is inactive – so to speak – while frame layer code executes, we do not have to care about any further synchronization. This means that while developing `handleFrame`, assumptions about the implementation of `RxChar` are made. This induces two major drawbacks:

First, synchronization takes advantage of implicit knowledge that does not directly manifest in the code but on a higher level of abstraction such as the state machine logic respectively the EMS communication protocol. This makes reasoning about the correctness of synchronization a challenge. The following cognitive interlude should make this apparent:

If we consider, for example, the access to the global synchronization variables we might question whether it is possible that a transmission initiation (line 17) gets lost if the frame and byte layers try to set `blState` at the same time. Is simultaneous access to `blState` even possible? As argued above, the byte layer code is usually inactive in `COMPLETE` while `startTransmission` executes. However, in case of `ERROR` (line 53), it returns to `IDLE` (line 54), thereby getting reactivated. Since `blState` is an 8-bit variable, access is intrinsically atomic due to the microcontrollers hardware architecture. `RxChar` has priority over $\mathcal{T}_H$. Thus, it may happen that `startTransmission` sets `blState` to `TX_BYTE` (line 58) and subsequently gets preempted by `RxChar` due to the receipt of `ERROR`. `RxChar`, in its turn, resets `blState` to `IDLE` (line 54) and terminates. Then, `startTransmission` continues to send the first byte of the response frame (line 59) and terminates too. In this scenario, the remaining frame bytes are never transmitted because in `IDLE` the incoming mirror bytes do not trigger the next byte transmissions. Consequently, $\texttt{Transmit}_{\texttt{BL}}$ fails. It seems as if synchronization is not correct. However, due to the EMS communication protocol, the following holds true. If we are in the situation that we want to transmit a frame, we either own the fieldbus (in case of $\texttt{Request}_{\texttt{FL}}$) or another device is currently waiting for our reply (in case of $\texttt{Respond}_{\texttt{FL}}$). In both cases, the protocol ensures that there is no communication meanwhile and hence no erroneous byte can be received. Thus, this case should never happen. Nevertheless, in the unlikely event it happens, for example because of an interfering signal on the bus, the byte layer state machine resets to `IDLE` and waits for a new transmission trigger by the frame layer. The frame layer, in its turn, implements a timeout in order to notice that the transmission has

not been successful. The timeout causes a retransmission of the same response frame in future. Finally, simultaneous access, in principle, may appear but will not cause any harm. This exemplary consideration should make apparent the imposed cognitive challenge required for reasoning about the correctness of interleavings between byte and frame layer code – it appears to be a very daunting and error-prone task.

> "[...] humans are quickly overwhelmed by concurrency and find it much more difficult to reason about concurrent than sequential code. Even careful people miss possible interleavings among simple collections of partially ordered operations." [SL05, p. 56]

Second, the assumptions about the implementation of `RxChar` clearly introduce interdependencies between the byte and the frame layer code. This makes the synchronization approach fragile since it only works by correct correspondence of all involved features. Calling `readFrame` and `writeFrame` is not thread-safe in general – only under a specific set of preconditions. The behavior of both state machines must perfectly match, for example with respect to self-deactivation. Consequently, they can not be easily developed independently and the slightest modification of their switching logic may cause the entire synchronization to fail. Thus, we cannot easily use `readFrame` in another execution context, for example in a concurrent logging service that creates a protocol of all received frames for debugging purpose. This would require to manually extend synchronization to the logging task or to hook the logging function into $\mathcal{T}_H$ at the correct location. Moreover, the approach of mutual state machine activation requires to take great care. The scenario in which both state machines are inactive at the same time must be explicitly ruled out when designing the state transitions. Otherwise the fieldbus driver might become inoperative. Finally, note that synchronization variables `notify` and `blState` have global scope. Their deployment causes operations in the byte layer to affect the behavior in the frame layer and vice versa. This creates additional mutual dependencies and untracked interactions between the separated software layers. In software engineering, this is generally considered an irregularity, called "action at a distance" [SK13].

4.3.2 Synchronization with Operating System Support

While the frame layer code is executed in a cyclic high frequency basic task $\mathcal{T}_H$, data layer code runs in a cyclic medium frequency basic task $\mathcal{T}_M$ as well as in a cyclic low frequency basic task $\mathcal{T}_L$. All three concurrent units of execution perform competing interaction. Specifically, they compete for access to a shared memory given by a frame queue `requests`. The queue aids as a buffer for storing request frames that have to be send by future runs of $\texttt{Request}_{\texttt{FL}}$. Competing means that each execution unit tries to obtain access on demand without coordinating with the others. $\texttt{Request}_{\texttt{FL}}$ consumes request frames from the queue and performs their transmission. The data layer produces request frames by two means. First, $\mathcal{T}_M$ processes HTTP requests from the web server and uses the data layer to sporadically create and enqueue corresponding request frames

on demand (event-triggered). Second, $\mathcal{T}_L$ runs an update service that periodically creates and enqueues request frames for fieldbus data which require regular updates (time-triggered). The frame and the data layers require to synchronize their access to the shared memory in order to avoid data races. Synchronization is implemented using the semaphore locking mechanism provided by the OSEK operating system.

Desired Atomicity and Execution Order The queue `requests` is implemented as a circular buffer. Enqueue and dequeue operations require multiple steps in order to modify the involved data fields as well as the read and write indices. In contrast to Section 4.3.1, both operations perform side effects on `requests`. If any two operations interleave this generally leads to data loss as well as invalid read and write pointers. Thus, at any time, there is only a single task execution instance allowed to work on the queue. This requires enqueue and dequeue operations to be atomic.

Accesses to `requests` may appear in any arbitrary order as long as they are atomic. $\mathcal{T}_M$ and $\mathcal{T}_L$– the producers – enqueue new request frames on demand. $\mathcal{T}_H$– the consumer – dequeues frames during $\texttt{Request}_{\texttt{FL}}$. If the queue is empty, there is nothing to do for $\texttt{Request}_{\texttt{FL}}$ and it will terminate immediately. If the queue is full, the oldest request is overwritten.

Implementation Outline Figure 4.12 illustrates how above considerations manifest in the code. The callback functions `taskHighCallback` (lines 35 to 39), `taskMediumCallback` (lines 1 to 7) and `taskLowCallback` (lines 8 to 14) are cyclically executed by $\mathcal{T}_H$, $\mathcal{T}_M$ and $\mathcal{T}_L$ respectively. Accesses to `requests` are highlighted bold. The data layer holds the queue (line 15) and its related indices (lines 16 and 17).

$\text{Task}_{\text{Medium}}$ + Task_{Low}

```
01:void taskMediumCallback(void) {
02: <...>
03: if(<HTTP-inquires-EMS-data>) {
04:   enqueueFrame(<...>);
05: }
06: <...>
07:}
```

"web_server.c"

```
08:void taskLowCallback(void) {
09: <...>
10: if(<update-required>) {
11:   enqueueFrame(<...>);
12: }
13: <...>
14:}
```

"update_service.c"

```
15:tst_Frame requests[MAX_QUEUE];
16:uint8_t readIndex;
17:uint8_t writeIndex;
18.
19:void enqueueFrame(<...>) {
20: <...>
21: GetResource(LOCK_QUEUE);
22: requests[writeIndex] = <...>;
23: writeIndex++;
24: <...>
25: ReleaseResource(LOCK_QUEUE);
26:}
27:
28:void dequeueFrame(<...>) {
29: <...> = requests[readIndex];
30: readIndex++;
31: <...>
32:}
```

"ems_data_layer.c"

$\text{Task}_{\text{High}}$

```
33:uint8_t flState = PASSIVE;
34:
35:void taskHighCallback(void) {
36: if(notify == 1) {
37:    handleFrame();
38:    notify = 0;
39:}}
40:
41:void handleFrame(void) {
42: processFrame();
43: switch(flState) {
44: case ACTIVE:
45:   <...>
46:   dequeueFrame();
47:   <...>
48:}}
```

"ems_frame_layer.c"

Figure 4.12: Synchronization with OS support in C

The web server and update service may enqueue frames at any time on demand (line 4 and 11). Enqueuing is implemented by `enqueueRequestFrame` (lines 19 to 26). The implementation takes advantage of the semaphore feature provided by OSEK for

resource management. First, it performs a system call in order to acquire the lock for `requests` (line 21). On success, OSEK guarantees that the calling task has exclusive access. Second, the new frame is stored (line 22). Third, the write pointer is adapted accordingly (line 23). Forth, the lock is released in order to provide access to other tasks (line 25).

The frame layer may dequeue frames at any time on demand too (line 46). Dequeuing is implemented by `dequeueRequestFrame` (lines 28 to 32). First, it reads the next frame from the queue (line 29). Second, it adapts the read index accordingly, thereby deleting the previously read frame from queue. Note that no lock acquisition is performed. This asymmetric locking approach needs some explanation.

Task $\mathcal{T}_H$, $\mathcal{T}_M$ and $\mathcal{T}_L$ have different execution priorities. The higher the execution rate the higher the priority. Thus, $\mathcal{T}_H$ has priority over $\mathcal{T}_M$ and $\mathcal{T}_L$ while $\mathcal{T}_M$ has priority over $\mathcal{T}_L$. Without semaphores, this leads to the following possible preemption scenarios: (1) $\mathcal{T}_M$ preempts $\mathcal{T}_L$, (2) $\mathcal{T}_H$ preempts $\mathcal{T}_L$ and (3) $\mathcal{T}_H$ preempts $\mathcal{T}_M$. $\mathcal{T}_H$ can never be preempted. The enqueue operation is semaphore protected. Thus, $\mathcal{T}_L$ and $\mathcal{T}_M$ compete for the same lock. Once $\mathcal{T}_L$ has successfully acquired the lock, it cannot be preempted by $\mathcal{T}_M$ anymore, thereby preventing (1). However, the dequeue operation is not semaphore protected. Thus, it seems as if scenarios (2) and (3) are still possible – which is in fact not the case. As mentioned in Section 4.1.2, semaphore handling in OSEK relies on a priority ceiling protocol. In our application the ceiling priority is statically configured to the one of $\mathcal{T}_H$. Once a task successfully acquired the semaphore it inherits the ceiling priority. This means that if $\mathcal{T}_L$ or $\mathcal{T}_M$ managed to obtain the lock they temporarily have the same priority as $\mathcal{T}_H$ until they release the lock. As a consequence, $\mathcal{T}_H$ can no longer preempt $\mathcal{T}_L$ and $\mathcal{T}_M$, thereby eliminating (b) and (c). Calls to `I_OS_SemaLock` in `dequeueRequestFrame` would be unnecessary and lead to additional runtime. Finally, this approach guarantees atomicity for en- and dequeuing operations.

Discussion Compared to the manual approach in Section 4.3.1, taking advantage of OSEK's semaphore feature considerably reduces the implementation complexity required for synchronization. `GetResource` and `ReleaseResource` allow to explicitly identify the critical section (lines 21 to 25), thereby making it automatically thread-safe. However, it appears that synchronization still does not manifest locally. While the code seems to be more concise and readable, there is still a line-up of conditions that must hold true in order to obtain correctness for all possible interleavings of $\mathcal{T}_H$, $\mathcal{T}_M$ and $\mathcal{T}_L$. Table 4.2 presents all possible preemption scenarios P1 to P6 and their corresponding preventions strategies in above implementation. The related conditions that make these prevention strategies actually work essentially fell in two groups:

The first group is composed of OSEK-related configuration parameters involving the scheduling policy, the assignment of task priorities and the priority ceiling protocol. These configurations are not locally tied to the code but reside in external, independent files, thereby inducing a cognitive distance. In particular, we identify the following conditions:

1. Preemption decisions are based on static task priorities. This is a fundamental requirement.
2. $\mathcal{T}_M$ always has priority over $\mathcal{T}_L$. This is required for P1.
3. $\mathcal{T}_H$ always has priority over $\mathcal{T}_M$. This is required for P2 and P4.
4. Occupation of `requests` leads to inheritance of the ceiling priority according to [ISO05, p. 31]. The ceiling priority is set to that of $\mathcal{T}_H$. This is required for P5 and P6.
5. $\mathcal{T}_H$, $\mathcal{T}_M$ and $\mathcal{T}_L$ do not enter the running state if `requests` is currently occupied. This is to avoid deadlocks.

The second group is composed of constraints imposed to the application code for correct semaphore handling. In particular, we identify the following conditions:

6. $\mathcal{T}_M$ and $\mathcal{T}_L$ acquire the semaphore before access to `requests` has been started and release the semaphore after access to `requests` has been completed. This is required for P3, P5 and P6.
7. $\mathcal{T}_M$ and $\mathcal{T}_L$ release the semaphore before task termination. Otherwise the behavior is unspecified according to OSEK. In the worst case, `requests` might be locked forever [ISO05, p. 73] making the driver software inoperative.
8. `dequeueFrame` is only called in $\mathcal{T}_H$. This is to guarantee that `dequeueFrame` can never be preempted.

Further, the OSEK specification provides an own section [ISO05, p. 72] about design hints for usage of `GetResource` and `ReleaseResource` system calls. In particular, they should follow the following guidelines in order to guarantee correct resource handling:

9. Calls to `GetResource` and `ReleaseResource` should be placed in the same functional level.
10. Calls to `GetResource` and `ReleaseResource` should directly encapsulate the access to the resource in a pairwise fashion.
11. Nested resource occupation must follow a LIFO policy. That is they have to be released in reversed order of their occupation.

It becomes apparent that both, the operating system configuration and the application code, must perfectly match in order to make access to `requests` thread-safe. Basically, its like a contract between the developer and the system wherein the system guarantees synchronization if the developer follows the rules. Similar to the manual synchronization

	Scenario	Prevention Strategy
P1	$\mathcal{T}_L \rightarrow \mathcal{T}_M$	$\mathcal{T}_M$ has priority over $\mathcal{T}_L$.
P2	$\mathcal{T}_L \rightarrow \mathcal{T}_H$	$\mathcal{T}_H$ has priority over $\mathcal{T}_L$.
P3	$\mathcal{T}_M \rightarrow \mathcal{T}_L$	$\mathcal{T}_L$ is lock-protected.
P4	$\mathcal{T}_M \rightarrow \mathcal{T}_H$	$\mathcal{T}_H$ has priority over $\mathcal{T}_M$.
P5	$\mathcal{T}_H \rightarrow \mathcal{T}_L$	$\mathcal{T}_L$ inherits ceiling priority.
P6	$\mathcal{T}_H \rightarrow \mathcal{T}_M$	$\mathcal{T}_M$ inherits ceiling priority.

Table 4.2: Possible preemption scenarios and their prevention strategy ($A \rightarrow B$: A tries to preempt B)

approach it comprises a high fragility since the slightest deviation causes the whole approach to fail. Thus, it is not robust towards changes. For example, if task priorities are modified in the external configuration files, the application code might be no longer synchronized. Note that the application code does not even detect that change. Consequently, the solution is very specialized for that particular constellation of system and application code. This rigidity makes it difficult to port onto another platform. Last but not least it is worth to mention that the whole synchronization approach is only valid on a single-core processor (as supposed by OSEK). If two or more tasks are exposed to physical concurrency, for example on a multi-core system, mutual exclusion is no longer guaranteed.

4.4 Temporal Behavior

Temporal behavior is intrinsic to physical systems. Essentially, it requires to perform computations, for example calculations or decisions, depending on the passage of time. According to Burns and Wellings [BW90] interfacing physical time must be generally possible in the context of specifying (a) delays, (b) timeouts and measuring (c) elapsed time. The conventional sequential model of computation, however, does not account for time in any way. Imperative languages such as C only allow to define the order of actions but not their timing [Lee05].

> "No widely used programming language integrates a way to specify timing requirements or constraints." [Lee05, p. 85]

Time passes in the system's physical environment while application code is executed. Due to the lack of proper computing abstractions, time appears to elapse independently and asynchronously from the program's point of view. Thus, in order to interface physical time, the key challenge is to (temporarily) synchronize the code and the time domain on demand. On this account, two typical strategies can be distinguished:

First, in the passive approach (inversion of control) a code chunk is executed once a specified amount of time has been elapsed. By this, the time information is provided implicitly. This strategy is directly supported by OSEK's alarm mechanism. Second, in the active approach (polling) the running code retrieves a counter value that represents the amount of elapsed time. By this, the information is provided explicitly on inquiry. However, OSEK does not provide a standardized Application Programming Interface (API) to access counters directly [ISO05, p. 36]. Thus, active time interfacing requires an indirect application-specific implementation. The fieldbus driver uses software timers for that purpose. Those timers belong to the application software and take advantage of OSEK's periodic system tick interrupt that triggers each millisecond. For measuring the elapsed time in milliseconds it increments a variable on each call, thereby enabling (c). The variable's current value can be read by the fieldbus driver. Multiple software timers may be registered and started concurrently.

In the following, we investigate how above strategies manifest in the fieldbus driver implementation. Therefore, we consider the implementation of delays (a) and timeouts (b) which adopt the passive and active approach respectively.

4.4.1 Delays

Sometimes it is necessary to deliberately suspend code execution until a certain amount of time has been passed. This artificial slowdown is usually required in order to account for speed differences between the embedded system and its physical environment. In contrast to timeouts (see Section 4.4.2), the delay interval must elapse. Thus, a delay is conceptually in sequence to the program's control flow. In the following, we investigate the delay implementation of the fieldbus driver which adopts the passive synchronization strategy.

Implementation Outline Write broadcast requests are not acknowledged. After transmission, the fieldbus driver may, in principle, immediately proceed with the next request without waiting for a response. However, in order to give other bus participants enough time for reception and processing, the fieldbus driver has to delay the next transmission for 250 ms. Figure 4.13 outlines the corresponding implementation approach. Accesses to OSEK's alarm feature are highlighted bold.

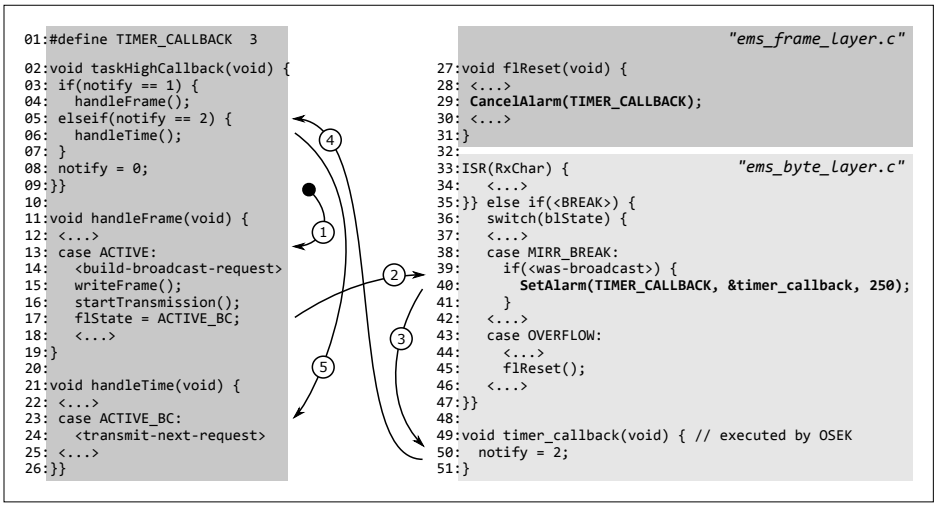

Figure 4.13: Outline of a delay implementation in C

First (1), the frame layer builds the broadcast request, copies it down to the byte layer and triggers its transmission (lines 14 to 16). `flState` is set to `ACTIVE_BC` which indicates the broadcast transmission (line 17). Once the transmission completes (2), the byte layer executes `SetAlarm` (line 40) in order to setup an OSEK alarm that triggers in 250 ms. Thereby, callback function `timer_callback` is passed and executed by OSEK once `TIMER_CALLBACK` expires (3). It sets `notify` to `2` (line 50) which notifies the frame layer code (4), thereby causing the execution of `handleTime` (line 21 to 26) (5). This finally leads to the transmission of the next request (line 24).

Discussion The expiration of `TIMER_CALLBACK` conceptually triggers an internal event which manifests in the assignment to `notify` (line 50). The corresponding processing is performed by $\mathcal{T}_H$ in (callback function) `handleTime` (lines 21 to 26). Consequently, the latter requires to introduce manual stack and state management for the same reason as discussed in Section 4.2. At the same time, it demands to reason about synchronization. Global variable `notify` is asynchronously accessed by three concurrent contexts of execution. $\mathcal{T}_H$ performs read and write access for event processing (line 3, 5 and 8), `RxChar` performs write access for notifying about a received frame (`notify = 1`, see Section 4.3.1) and finally an OSEK interrupt service routine executes `timer_callback` (lines 49 to 51), thereby performing a write access to notify (line 50) about the expiration. Thus, it appears that the passive approach, to a certain extend, introduces the problems of event handling (see Section 4.2) and synchronization (see Section 4.3.1) at the same time. As Figure 4.13 reveals this is particularly true for the convoluted control flow and the scattering of delay logic. Note that the delay is initiated by `RxChar` (line 40), triggered by OSEK (line 50) and finally processed by $\mathcal{T}_H$ (line 6). The behavior of all involved components must match in order to make the entire approach work. In addition, this also makes it hard to establish the cognitive link between the code locations that indicate the start and end of the time interval.

Furthermore, due to the single entry point model, we cannot block the control flow of a one-shot trail in order to await the passage of time. Consequently, delays must be implemented using a concurrent approach although they are conceptually in sequence with the program's control flow. This seems to be in contradiction to the intuitive notion of a delay and additionally makes it difficult to understand how it actually manifests in the code.

Moreover, correctly canceling a started alarm is of great importance. For example, if the frame layer state machine is reset in `flReset` due to a communication error (see Section 4.2.3) without canceling the alarm, the time event will be triggered at some future point in time, thereby causing `handleTime` to run. In the best case, the frame layer state machine just ignores the time event in its current state. In the worst case, it performs a reaction that most probably does not fit to the current communication progress, thereby unnecessarily causing an error.

4.4.2 Timeouts

Specifying a timeout means to intend a certain event or event sequence to occur within a specified time interval. In contrast to delays (see Section 4.4.1), a timeout interval should usually not elapse. Thus, a timeout is conceptually concurrent to the program's control flow and used for monitoring purpose. In the following, we investigate the timeout implementation of the fieldbus driver which adopts the active synchronization strategy.

Implementation Outline In Figure 4.14, we outline how timeout specifications for the idle time t_i and the mirror time t_m manifest in the code. The two software timers

TIMER_IDLE (line 1) and TIMER_MIRROR (line 2) measure t_i and t_m respectively. Access to the timers are highlighted bold.

"ems_byte_layer.c"

```
01:#define TIMER_IDLE   0
02:#define TIMER_MIRROR 1

03:ISR(RxChar) {
04:  if(getCount(TIMER_IDLE)==ELAPSED) {
05:    blReset();
06:  }
07:  start(TIMER_IDLE, 350);
08:  if(<BYTE>) {
09:    switch(blState) {
10:    <...>
11:    case MIRR_BYTE:
12:      if(getCount(TIMER_MIRROR)==ELAPSED) {
13:        blReset();
14:      }
15:    <...>
16:  }} else if(<BREAK>) {
17:    switch(blState) {
18:    <...>
19:    case MIRR_BREAK:
20:      if(getCount(TIMER_MIRROR)==ELAPSED) {
21:        blReset();
22:      }
23:    <...>
24:  }}

25:ISR(TxChar) {
26:  stop(TIMER_MIRROR);
27:  switch(blState) {
28:  case TX_BYTE:
29:    blState = MIRR_BYTE;
30:    start(TIMER_MIRROR, 42);
31:    break;
32:  case TX_BREAK:
33:    blState = MIRR_BREAK;
34     start(TIMER_MIRROR, 42);
35:    break;
36:}}

37:void blReset(void) {
38:  <...>
39:  stop(TIMER_IDLE);
40:  stop(TIMER_MIRROR);
41:  <...>
42:}
```

Figure 4.14: Outline of a timeout implementation in C

On the receipt of the first frame byte, getCount retrieves the current counter value of TIMER_IDLE (line 4). Since this is the first run of RxChar TIMER_IDLE has not been started yet. Consequently, it is not elapsed and blReset (line 5) is skipped. In line 7, TIMER_IDLE is actually started with the corresponding time interval of 350 ms. In the following runs of RxChar, line 4 always checks whether the timeout occurred in between two received characters. If this is the case, blReset stops TIMER_IDLE (line 39) and resets the state machine.

Once the transmission of the first frame byte has been completed, TxChar first stops TIMER_MIRROR (line 26). Then, it is started with the corresponding time interval of 42 ms (lines 30 and 34). If the mirror is received (lines 11 and 19), TIMER_MIRROR is checked for expiration (lines 12 and 20). If this is the case, blReset stops TIMER_MIRROR (line 40) and resets the state machine.

Discussion In the active approach, each timeout interval is assigned to a dedicated timer which demands manual management. This requires to have an explicit identity, for example TIMER_IDLE and TIMER_MIRROR, which allows separated interfacing. The developer must determine the maximum number of different time intervals at development time and deal with allocation and registering according operating system resources. Timer management is performed through a loose collection of functions, for example start, getCount and stop, which may be called, in principle, in any arbitrary order. The developer must coordinate accesses to the corresponding timer across reactions in order to guarantee their correct execution order. For example, if stop is accidentally called in between start and getCount the check for timing accuracy will fail and hence leads to wrong behavior. A single timeout specification, however, does not manifest locally

in the code. According timer accesses are scattered across different code locations in interrupt service routines and functions. This may easily lead to unintentional interleaving of different timeout intervals. Also, their lexical order within the code does not correspond to their intended order of execution. Thus, it appears that manual timer management is similar to accessing a global variable and hence comprises the same line-up of disadvantages [SK13] making it a daunting and error-prone task.

Furthermore, there is no obvious coupling between the timeout interval and the associated action. In particular, due to the scattered interface, it is hard to comprehend which lines of code are actually protected by a certain interval. Actually, the contained action is torn across different functions too. `TIMER_MIRROR`, for example, comprises several start- (lines 30 and 34) and endpoints (lines 26 and 40). In between any consecutive calls of `start` and `stop` an arbitrary amount of byte layer and non-byte layer code may run. Also, the processing of a timeout is not enforced. If `getCount` is not evaluated in the correct reaction at the correct code location, a timeout may silently appear without any impact on the behavior. Abortion must be manually taken care of. For example, if `ERROR` causes abortion of the current run of $\texttt{Transmit}_{\texttt{BL}}$, the current mirror timeout interval must be aborted too (line 40).

Moreover, the specified time intervals (lines 7, 30 and 34) have no unit since time is not supported as a physical quantity. This means that the defined values depend on the timer's frequency and hence the operating system configuration. If the configuration changes, timeout handling will no longer work. Consequently, the application code must be adapted accordingly.

Finally, timers cannot be nested. Thus, it is not possible to describe hierarchical dependencies between time intervals. For example, the response timeout t_r conceptually contains several instances of t_i and t_m respectively. If t_r expires, the current instances of t_i and t_m have to be automatically aborted too since they have become irrelevant and require restart on the next run of $\texttt{Transmit}_{\texttt{BL}}$ or $\texttt{Receive}_{\texttt{BL}}$ respectively. However, this hierarchical nesting is not mappable to timers and must be implemented manually, for example by correctly calling the reset functions of the corresponding state machine as indicated in Section 4.3.1.

4.5 Conclusion

Traditional sequential programming does not provide any language-level support for the implementation of event handling, concurrency and temporal behavior which are intrinsic to reactive, physical systems. This inevitably demands an underlying execution platform that adds the required capabilities. In order to facilitate development, non-trivial embedded applications usually escape from programming on the bare metal by taking advantage of an operating system. Its features particularly respond to the strong resource limitations of the embedded domain. By the common notion of tasks, interrupt service routines and alarms, the OSEK operating system, for example, provides a framework that allows to hook up several application code chunks for event- and

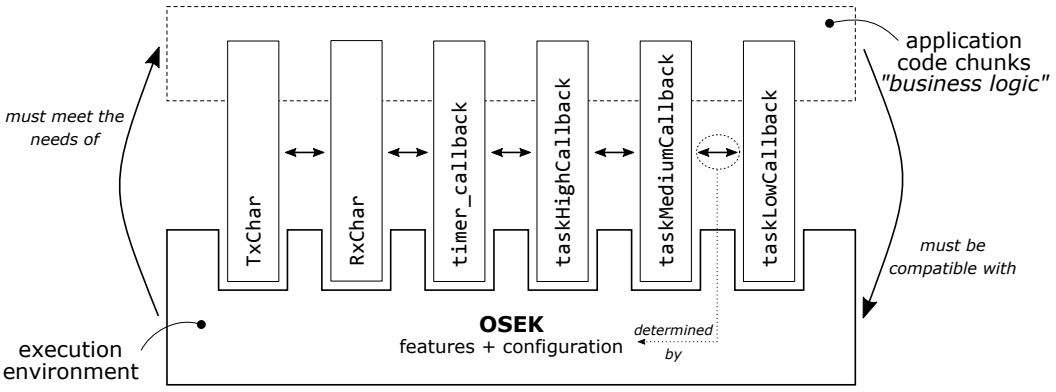

Figure 4.15: Dependency between application code and execution environment

time-triggered, asynchronous, concurrent execution (see Figure 4.15). As a whole, these interacting units realize the fieldbus communication.

By this, the business logic conceptually disintegrates into several loose fragments tied together by the surrounding execution environment. Based on a line up of configurations, for example scheduling policy, task priorities, priority ceiling protocol and timer intervals, the OSEK scheduler determines how these entities actually execute and interleave at runtime. Also, OSEK imposes clear guidelines and restrictions for the usage of operating system and application functions throughout the code. Thus, in order to ensure correctness for all possible interleavings and make the application work, on the one hand, the implementation must be compatible to the framework's features and their respective configuration. On the other hand, the features and their configuration must meet the needs of the application. It appears that this makes the application code and the execution environment firmly intertwined. In particular, note that the logic of how concurrent application components correctly execute and cooperate in order to fulfill their supreme aim mostly does not manifest in the application's business logic itself but is scattered across a set of external conditions. This non-locality requires to navigate back and forth between multiple places in a source file (or even between multiple source files) in order to see and comprehend everything the code actually does. With respect to software engineering, this approach appears to give rise to two major drawbacks:

First, it makes software development particularly difficult since those application code chunks are generally hard to program, comprehend and maintain. This is mainly due to the torn and convoluted control flow, the loss of function-oriented decomposition and hierarchical composability as well as the need for explicit synchronization and manual timer management.

Second, the final software product is fragile. If the configuration changes the code will no longer work as expected. As a consequence, it cannot be easily ported or reused without either adapting the application code accordingly or ensuring that the entire complexity of application-external conditions is preserved.

5 Deployment and Qualitative Evaluation of Synchronous Programming

Fieldbus communication is the major reactive concern of the gateway system. In Chapter 3, we considered event-based, synchronous-reactive programming a suitable approach which is expected to lead to a simple and concise solution. In this chapter, we justify our considerations by actually deploying the synchronous paradigm for fieldbus communication. Therefore, we reimplement the fieldbus driver using the synchronous-reactive programming language Céu and elaborate its qualitative, engineering-related benefits compared to the asynchronous approach outlined in Chapter 4.

First, in Section 5.1, we provide architectural considerations about the system integration of the synchronous paradigm and the internal structure of our synchronous reimplementation. Second, in Sections 5.2 and 5.3, we demonstrate how Céu enables function- and object-oriented software designs in the reactive domain. Third, in Section 5.4, we consider Céu's additional language-level support for testing. Finally, in Section 5.5, we retrospectively discuss important points to consider when deploying the synchronous paradigm.

Choice of Céu The reactive paradigm [Bai+13] proposes a number of solutions which address the problems with callbacks and inversion of control: (1) data flow languages, (2) functional reactive programming – a modernized data flow style – , (3) reactive extensions to general-purpose languages and (4) synchronous languages. The data flow style has been adopted in prominent industry tools, namely Matlab/Simulink, LabView and SCADE which promote visual, model-driven development. They particularly address the subdomain of highly safety-critical, hard real-time systems, for example automotive, avionics, railways or nuclear power plants. Those applications mostly rely on computation-intensive control tasks known from control theory (see Section 4.1.2) and hence excellently map the declarative, data-centric style.

Switching behavior (see Section 3.2), however, focuses on control flow rather than data flow. Above approaches provide graphical state machines for developing control-centric concerns. For illustration purpose, in Figure 5.1, we manually derived a graphical automaton based on the implementation of $\texttt{Respond}_{\texttt{FL}}$ outlined in Section 4.2.2. It appears, that even such a simple functionality leads to a quite complex graphical representation which manifests in 8 states and 32 transitions. We are aware that features like hierarchy, default- and history states significantly help to reduce the complexity

induced by flat automata. Nonetheless, the fundamental components of state machines – states, transitions and events – focus on the specification and visualization of single event reactions rather than their linear flow. This makes state explicit while control flow remains implicit. In Figure 5.1, the gray highlighted transitions, for instance, illustrate the hidden control flow for one error-free run of $\texttt{Respond}_{\texttt{FL}}$. We believe that this approach is hardly better to understand than the textual `switch-case` adoption.

> "[...] the human design and maintenance of automata turns out to be very difficult and error-prone. Nontrivial automata are difficult to draw and impossible to understand when not drawn." [BG92, p. 90]

For this reason, in accordance with Sant'Anna et al. [SIR15], we believe that the text-based, sequential, imperative programming style is more suitable for switching behavior and hence more intuitive to developers. Also, state machines are not robust towards changes – "the slightest modification in the system specifications may involve a complete modification and rewriting of the automaton." [Hal93, p. 3]

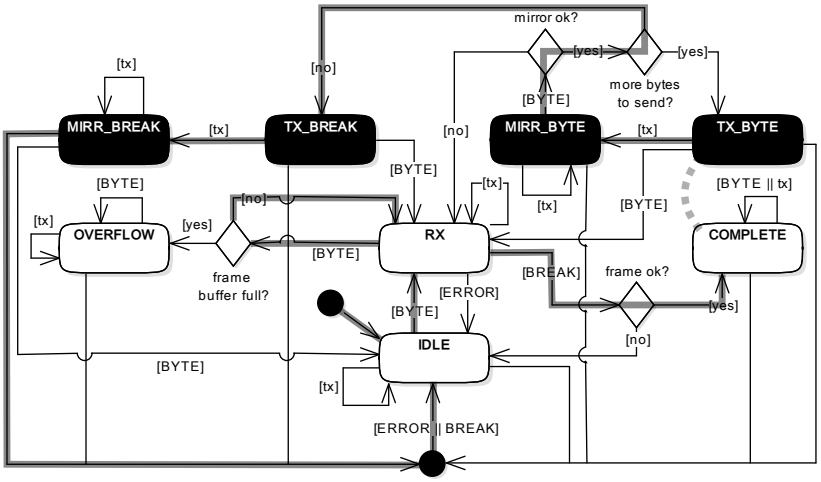

Figure 5.1: Graphical representation of the byte layer state machine

Furthermore, the vast majority of reactive extensions is based on general-purpose languages [Bai+13] which are generally unsuitable for embedded systems. A number of embedded-oriented technologies emerged in the context of wireless sensor networks. Examples include Céu [San+13], nesC [Gay+03], Protothreads [Dun+06] and others. Sant'Anna et al. [San+13; San13] provide an extended list as well as a qualitative comparison. Céu outperforms existing approaches with respect to language features which reduce code complexity and increase safety. Above all, this is due to its synchronous model of execution which provides deterministic, bounded execution and safe shared memory access. Also, to our knowledge, Céu seems to be the only representative of the synchronous language family [Ben+03] that provides a text-based, imperative programming style as well as an up-to-date and freely accessible compiler. Moreover, it only imposes a small memory overhead compared to hand-crafted event-driven code

and its performance is comparable to nesC – an event-driven extension for C– making it suitable even for constrained devices [San+13].

Hard- and Software Platform For our deployment, we pick a BeagleBone Black development board[1] running a Linux operating system. It provides a 1 GHz processor, 512 MB RAM and 4 GB flash memory. This platform is more powerful than the existing embedded microcontroller which executes an OSEK operating system. However, this does not invalidate any of our considerations below. Exemplified Céu code will run on the existing gateway platform likewise. Only the thin platform interface layer (see Section 2.3) needs to be adapted accordingly. For example, OSEK's recurring basic tasks can be used to advance Céu's generated state machine. Each task execution instance performs a single reaction which is bounded in time and memory. Our choice is deliberately done for two reasons:

First, Linux provides a feature-rich and stable software playground for experimentation which enables rapid prototyping. In particular, this involves operating system support for process management and inter-process communication as well as diagnosis tools for monitoring purposes.

Second, Bosch Group considers a platform switch towards a Linux-based gateway system in order to take advantage of existing software solutions and simplify development. We aim to take this as a chance to actually integrate our exploratory work into future production code.

Functional Tests In order to actually show the operability of our synchronous reimplementation we perform functional tests. Therefore, we connect our development platform running the fieldbus driver in Céu to a real heating appliance which is composed of an EMS master and two slave devices. At the same time, an EMS diagnostic device from Bosch Group is connected to the fieldbus, too. This allows to independently monitor and check the fieldbus communication performed by our synchronous reimplementation. We log the traffic on the fieldbus for about 14 hours. The synchronous code behaves correctly for the whole duration of the test.

5.1 Architectural Considerations

Asynchronous execution seems inherent in embedded software. Considering development on the bare metal, microcontroller hardware provides interrupt service routines which asynchronously execute to the main program. Operating systems additionally promote the notion of asynchronous tasks or threads. Thus, it seems reasonable to generally consider an embedded application being asynchronous by default. Deploying a synchronous language in that globally asynchronous context consequently requires to locally prune away asynchronism. Every Céu application constitutes a synchronous island within the entire asynchronous system. The synchronous hypothesis (see Section 2.2) does

[1] `http://beagleboard.org/black`

only hold within those synchronous components. This approach is also known as the Globally Asynchronous, Locally Synchronous (GALS) model of computation [SIR12]. All operations within a Céu program are performed synchronously while the remainder execute asynchronously. The GALS execution must be reflected in the architectural application design in order to allow seamless integration. Actually, this requires to consider two different architectures:

First, the global architecture of the entire system which enables to combine the sequential, asynchronous and the reactive, synchronous domain in a single application. Second, the local architecture which internally structures the synchronous subsystem.

In our work, we address both concerns. Following a top-down approach, we first present our domain-oriented system architecture in Section 5.1.1 which links the synchronous fieldbus driver and the asynchronous web server. Although our work focuses on the synchronous paradigm, we deliberately provide a short outline of the web server's implementation too in order to illustrate how the diversity of both domains manifests in the architectural design. Subsequently, in Section 5.1.2, we illustrate how the synchronous fieldbus driver code is actually interfaced by its surrounding C environment. Finally, in Section 5.1.3, we demonstrate and explain the fieldbus driver's internal design.

5.1.1 Domain-Oriented System Architecture

In the course of designing a software architecture the entire system is generally decomposed into individual components. An architectural style [BF14, ch. 2] is applied in order to define relations and constraints among them. Typically, this leads to a pure logical structuring of source code without taking the domain-specific computation requirements (see Section 3.2) into account. The resulting system design relies on a single programming language respectively model of computation and hence is flat. If different problem domains are involved, only part of the domain-specific problems are easy to solve (see Chapter 4). On this account, we propose a domain-oriented software architecture. It allows the deployment of different programming languages based on appropriate computation models and domain-specific features in order to provide best implementation support.

Internet and fieldbus communication are disjoint concerns to the greatest possible extend. While the web server answers HTTP requests, the independent fieldbus driver reacts to the received characters. Only in case of an HTTP request for an external resource, a handshake is required to exchange the corresponding EMS request and EMS response respectively. In order to manifest this separation of concerns in the software architecture, we organize its high-level into layers – a major architectural style [BF14, ch. 2]. As illustrated in Figure 5.2, Internet communication is assigned to Layer 1, fieldbus communication to Layer 2. Our layered design takes advantage of two independent Linux processes P_{L1} and P_{L2} for the following reasons:

First, it enables usage of different programming languages as demanded above. Second, it allows selective scheduling prioritization. For example, in order to address domain-specific timing requirements, we assign the fieldbus driver process P_{L2} to

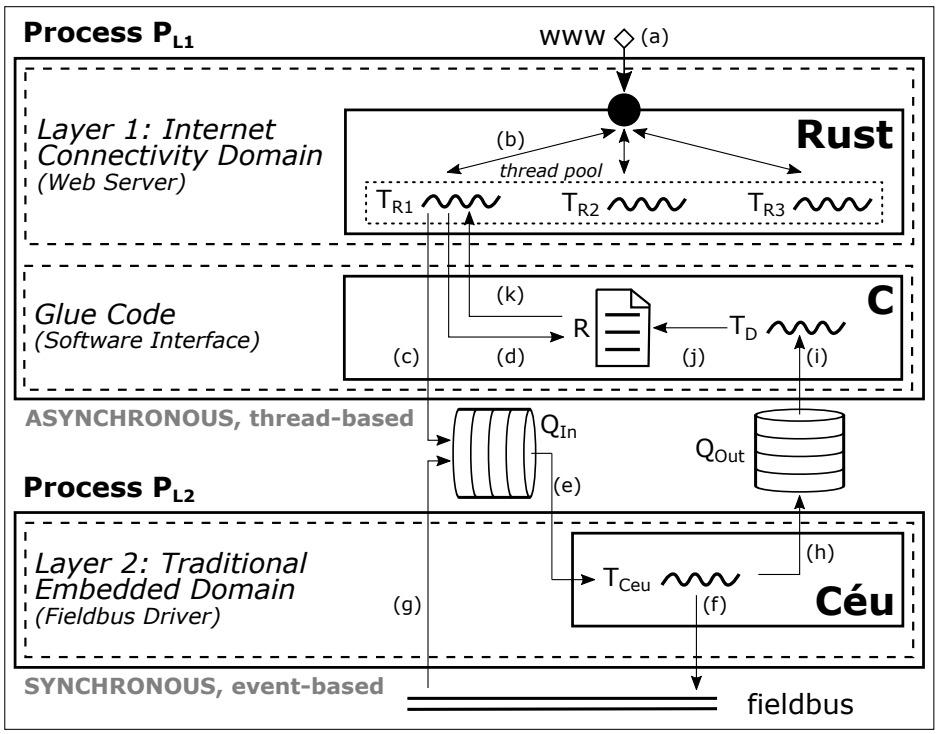

Figure 5.2: Domain-oriented, layered software architecture with Céu and Rust

the Linux Real-Time Scheduler (RTS) while the web server process P_{L1} runs in the conventional Completely Fair Scheduler (CFS). Third, Linux processes independently run in distinct memory spaces. Thus, their execution may never interfere and they can be easily restarted for error recovery which increases robustness. This also complicates a compromised web server to attack the fieldbus driver, thereby becoming an important security aspect. Forth, it allows execution on different processor cores to fulfill real-time requirements (not necessary in our application). Fifth, the web server and the fieldbus driver can be independently developed, tested and evaluated.

Finally, a *glue code* serves as a lean software interface between web server and fieldbus driver. Essentially, it manages the inter-layer handshake based on two thread-safe Linux message queues Q_{In} and Q_{Out} provided for inter-process communication.

Layer 1: Web Server in Rust With respect to the web server, we aim for a programming language that allows a more comfortable and safe implementation approach than C. String handling, for instance, is a major task for Internet connectivity but poorly supported and hence error-prone in C. At the same time, the language should be efficient and low-level enough to run on a resource-constrained, embedded device. We consider Rust[2] a feasible candidate for the following reasons. Rust is a system programming language focusing on safety, speed and concurrency [Rus]. It intrinsically supports thread-based, sequential programming and asynchronous execution by design –

[2]https://www.rust-lang.org/

remember that C is only single-threaded. Thus, RUST's fundamental computation model fully matches the requirements for Internet communication elaborated in Section 3.2. Furthermore, RUST aims to achieve efficient, low-level hardware accesses and high-level programming features at the same time by so called "zero-cost abstractions". This combination makes it generally attractive for Internet applications. Mozilla Research, for example, uses RUST for developing the new web browser engine Servo [And+15]. In addition, several existing RUST-frameworks support the development of web servers. In [Ter16], we exemplarily illustrate how we use the web framework Iron[3] on that purpose.

Our RUST web server comprises a fixed-size thread pool, for example T_{R1-3} (see Figure 5.2). Each incoming HTTP request (a) is automatically received, decrypted, parsed and assigned to one of the non-busy threads (b) for further processing, for example T_{R1}. Subsequently, T_{R1} executes all the code required to interpret, execute and build the corresponding HTTP response. For execution of an external resource request, it calls the glue code in order to enqueue the EMS request into Q_{In} (c). By this, T_{R1} maps the EMS request to an input event for the fieldbus driver. Finally, T_{R1} registers the submitted request and itself with the glue code (further explanation below) and suspends until the response is ready (d). Once the data are ready, T_{R1} resumes and builds the HTTP response. The web server takes care of encryption and transmission.

Threads naturally provide interleaved request processing. Long-lasting ones that inquiry EMS data, are suspended and resumed once the data are ready. Thus, they do not consume any processing time while waiting and can be easily overtaken by quicker ones.

Layer 2: Fieldbus Driver in Céu We use the programming language CÉU in order to deploy the synchronous paradigm for the driver. The auto-generated CÉU state machine code runs in a single, dedicated thread T_{Ceu}. It successively reads the EMS requests from its input event queue Q_{In} (e), performs event-based processing (f), (g) and returns the corresponding EMS responses as output events back to output queue Q_{Out} (h). Its implementation is exemplified in the following sections in more detail.

Glue Code in C Code in RUST cannot directly interface CÉU code and vice versa. However, both languages have the ability to execute native C code. Also, the Linux Portable Operating System Interface (POSIX) standard enables to perform system calls in C. For this reason, we use C as the lowest common denominator between web server, fieldbus driver and operating system.

An HTTP request for an external resource, for example in T_{R1}, causes the web server to call the glue code for enqueuing the request in Q_{In}. T_{R1} subsequently blocks until the data is provided by the fieldbus driver. A dedicated dispatcher thread T_D in the web server retrieves the EMS response from Q_{Out} (i) and forwards it to the waiting T_{R1} (j). On the one hand, the web server may create several concurrent EMS requests limited by the size of the thread pool. On the other hand, there is a single fieldbus driver for processing. All EMS requests are buffered in Q_{In} and all EMS responses are returned

[3] https://docs.rs/iron/0.5.1/iron/

in Q_{Out}. In order to enable the correct assignment of response to thread, the glue code uses a register list R. The glue code generates a unique numerical identifier for each request and passes it to the fieldbus driver. Also, the identifier is stored in R together with the thread identification number. The fieldbus driver returns the request identifier in the response. The glue code looks up the corresponding thread in R, resumes it and forwards the reply. Once T_{R1} gets resumed, it finally reads the EMS response from R (k).

5.1.2 Interfacing Synchronous Code

As indicated above, Céu's synchronous code – represented by the auto-generated state machine in C– is executed by the C environment in thread T_{Ceu}. In order to make this possible, Céu code must define an event-based interface first. In this section, we illustrate the interface declaration in Céu and its actual execution in C.

Declaration of the Céu Code Interface The Céu application defines the event-based interface used for interacting with its C environment. The declaration, as presented in Listing 5.1, is done at the beginning of the top-level fieldbus driver implementation. First, we link to the fieldbus by declaring the input events `BYTE` (line 2), `BREAK` (line 3) and `ERROR` (line 4) as identified in Section 3.2. Also, we need to specify the output events `TX_BYTE` (line 5) and `TX_BREAK` (line 6) which allow Céu to pass a transmission request for a single byte or end-of-frame character to the Linux operating system. The payload of `BYTE` and `TX_BYTE` provides the byte value received and to be sent respectively. Second, we link to the web server by declaring the input event `EMS_REQUEST` (line 8) and output event `EMS_RESPONSE` (line 9). While the former indicates an incoming EMS request from the web server, the latter is used to return the corresponding EMS response. Their payload carries user-defined structures which contain all the required information for the associated request and response respectively.

```
1 // Interface to Fieldbus
2 input u8 BYTE;                     // received frame byte with valid content
3 input void BREAK;                  // received end-of-frame character
4 input void ERROR;                  // received character with malformed content
5 output u8 TX_BYTE;                 // trigger frame byte transmission
6 output void TX_BREAK;              // trigger end-of-frame character transmission
7 // Interface to Web Server
8 input _data_request EMS_REQUEST;   // incoming EMS request
9 output _data_response EMS_RESPONSE; // outgoing EMS resqonse
```

Listing 5.1: Declaration of the event-based interface to Céu code

Execution in C The dedicated fieldbus driver thread T_{Ceu} executes the simplified code illustrated in Listing 5.2. We import Céu's auto-generated state machine code (line 2). In `main` (lines 3 to 29) variable `app` (line 4) stores the entire state machine runtime

context including the required working memory. Line 7 indicates the initialization of the state machine as well as the platform interface code. The `while`-loop (lines 9 to 26) successively advances the state machine in three steps:

```
1  #define ceu_out_wclock_set(us) wclock_next = us;
2  #include "_ceu_app.c"
3  int main(<...>) {
4    tceu_app app; // holds the generated state machine
5    byte CEU_DATA[sizeof(CEU_Main)];
6    app.data = (tceu_org*) &CEU_DATA;
7    <initialization>
8    /* Successively advance the state machine. */
9    while (app.isAlive) {
10     // Step 1: Block until next event or timeout.
11     dequeue(<input-queue>, <event-id-and-payload>, wclock_next);
12     // Step 2: Perform time-triggered reaction chain.
13     ceu_sys_go(&app, CEU_IN__WCLOCK, &delta_time);
14     // Step 3: Perform event-triggered reaction chain.
15     switch (<event-id>) {
16       case CEU_IN_BYTE:
17         ceu_sys_go(&app, CEU_IN_BYTE, <received-byte>); break;
18       case CEU_IN_BREAK:
19         ceu_sys_go(&app, CEU_IN_BREAK, NULL); break;
20       case CEU_IN_ERROR:
21         ceu_sys_go(&app, CEU_IN_ERROR, NULL); break;
22       case CEU_IN_EMS_REQUEST:
23         ceu_sys_go(&app, CEU_IN_EMS_REQUEST, <request-data>); break;
24       default: /* nothing */ break;
25     }
26   }
27   <de-initialization>
28   return 0;
29 }
30 // Handler functions for output events.
31 int ceu_sys_output_TX_BYTE(<the-byte-to-send>) {
32   <...> aio_write(<byte-to-send>); <...>
33 }
34 int ceu_sys_output_TX_BREAK(void) {
35   <...> aio_write(<end-of-frame>); <...>
36 }
37 int ceu_sys_output_EMS_RESPONSE(<...>) {
38   <enqueue-response-in-output-queue>
39 }
```

Listing 5.2: Execution of CÉU's state machine

First, we call `dequeue` (line 11) in order to retrieve the next event from the input queue Q_{In}. If Q_{In} is empty, `dequeue` blocks and suspends T_{Ceu} until at least one event has been enqueued. This releases the processor if no event is ready for processing. In addition, `dequeue` also returns once the timeout interval specified by `wclock_next` has been expired. By this, `dequeue` provides event- and time-triggered execution of CÉU code. At the end of each reaction the state machine notifies when the next reaction must be performed in future due to the passage of physical time. Therefore, it calls the C macro `ceu_out_wclock_set` (line 1), thereby setting `wclock_next` to the according number of microseconds. Thus, either an event occurs meanwhile or the passage of physical time causes a new reaction to run on timeout at the latest. This approach

wakes up T_{Ceu} only on demand and not, for example, in a periodic fashion which saves processor time.

Second, we perform a time-triggered reaction (line 13). Therefore, we call `ceu_sys_go` and provide the event id `CEU_IN__WCLOCK`. This indicates to the CÉU state machine that physical time has been passed while `delta_time` provides the number of microseconds that have been passed since the last reaction. On the one hand, this updates CÉU's internal wall clock (see Section 2.3) by the amount of `delta_time`. On the other hand, it executes any time-dependent transitions which got triggered due to the passage of `delta_time`. This is how CÉU actually interfaces physical time. In order to retrieve `delta_time`, we take advantage of Linux' system clocks.

Third, we perform an event-triggered reaction depending on the retrieved event – if any (lines 15 to 25). In case of BYTE (line 16), for example, we execute `ceu_sys_go`, thereby providing the corresponding event id `CEU_IN_BYTE` and the received byte value as payload (line 17). The similar approach is adopted for the remaining fieldbus events (lines 19 and 21) as well as for the EMS request event indicated by id `CEU_IN_EMS_REQUEST` (line 22).

Above steps are repeated as long as the state machine respectively the fieldbus driver is active (`app.isAlive == 1`). Otherwise, the loop exits the state machine, the platform interface code is deinitialized (line 27) and `main` returns, thereby finally terminating T_{Ceu}.

Output events cause the state machine to call one of the corresponding handler functions of the platform interface code (lines 37 to 39). In order to preserve the synchronous hypothesis, they must not contain any blocking or long-lasting code. However, taking the example of `TX_BYTE`, the actual byte transmission requires more than one millisecond. This is due to the baud rate of the fieldbus. Consequently, if we use a blocking Linux system call such as `write` for transmission, the entire CÉU code remains blocked for that amount of time too. In general, this requires to introduce an additional output queue to decouple the long-lasting byte transmission from the reactive code. On this account, we take advantage of Linux's asynchronous I/O system calls. Function `aio_write` (line 32) only enqueues the write request into a Linux-internal driver queue and returns immediately, thereby keeping the state machine reactive. We apply the same approach for `TX_BREAK` (line 35).

Finally, it is worth to mention that the worst-case amount of memory required for executing the CÉU state machine is determined at compile time in line 5. `CEU_Main` holds a complex, auto-generated structure that contains the maximum of variables, trails, organisms, pools and so on that may exist in parallel during runtime (see Section 2.3). The statically allocated memory is subsequently assigned to the state machine `app` (line 6).

5.1.3 Fieldbus Driver Architecture

As mentioned in Section 5.1.1, designing a software architecture is about decomposing the entire system into individual components. Those components typically rely on abstraction entities provided by the programming language in use. In order to explore

the solution space, we first consider the available entities of abstraction in Céu as well as their general usage strategies. Furthermore, we provide a best practice on how to chose the suitable deployment strategy depending on the intended use case. Finally, we present and explain our adoption for designing the fieldbus driver architecture.

Available Abstraction Entities In the synchronous domain, Céu's language design consistently distinguishes between code abstraction within and across reactions (see Listing 5.3). For organizing and structuring code within a single reaction, Céu provides *functions* which are equivalent to their C counterparts (lines 2 to 4). A function starts execution and runs to completion in the same reaction. In particular, it cannot be used for event processing. For this reason, it must not contain any synchronous control statements (see Section 2.3).

```
1  // Function Declaration
2  function (<list-of-parameters>)=><return-value> <function-name> do
3    <implementation>
4  end
5  // Organism Declaration
6  class <organism-name> with
7    <interface-declaration>
8  do
9    <execution-body>
10 end
```

Listing 5.3: Declaration of functions and organisms

In order to hide implementation details across reactions, Céu additionally introduces the concept of *organisms*, thereby providing a new object-like level of abstraction which encapsulates data and associated control flow (lines 6 to 10). A class of organisms describes a public interface for correct and smooth calls (line 7) and an execution body defining its runtime behavior (line 9). Variables (knowledge), functions (skills) and internal events (stimuli) may be encapsulated within the execution body or published by the interface. In contrast to functions, an organism is a concurrent unit of execution. Once instantiated, its execution body runs concurrently to the remaining program. Thus, organism instantiations are non-blocking. An organism can provide an integer return value on termination or by references through its interface like in C. The execution body can hold any valid Céu code particularly including synchronous control statements. By this, an organism can suspend its execution until a future event occurs and resumes subsequently. This enables organisms to outlive several events and hence to encapsulate their reactions within a single abstraction entity. Consequently, organisms effectively close the abstraction gap discussed in Section 4.2.

General Deployment Approaches In Listing 5.4, we illustrate the fundamental usage of functions and organisms. Using functions in Céu is similar to C and hence straightforward (1). We call function `add` in order to abstract the code responsible for

adding two integer values and returning the result (line 2). In addition, we identify three possible approaches on how to deploy organisms.

```
1  //---(1) Function
2  var u16 sum = add(1, 7);
3
4  //---(2) Organism: function-like
5  // verbose call with explicit identity
6  var Org o with
7    <...>
8  end;
9  await o;
10 // concise, anonymous call
11 do Org with
12   <...>
13 end;
14
15 //---(3) Organism: object-like, static
16 var Org o with
17   <...>
18 end;
19 <...>
20 <...> = o.x;
21
22 //---(4) Organism: object-like, dynamic
23 pool Org[10] orgs;
24 spawn Org in orgs with
25   <...>
26 end;
```

Listing 5.4: Possible deployments of code abstraction entities

First, they can be used in a *function-like* fashion (2). Therefore, we first create an instance `o` of organism `Org` (lines 6 to 8). Then, **`await`** `o` (line 9) blocks the calling trail until `o` has run to completion. This prunes away `o`'s inherent concurrency and hence imposes sequential execution. If we do not need an explicit identity for further reference, for example for accessing the organism's interface variables after its termination, we can also use the concise **`do`**-instantiation (lines 11 to 13). This creates an anonymous organism and waits for its termination automatically. By this, an organism behaves like a "reactive subroutine" [SIR15, p. 36] devised for continuous input.

Second, they can be used in an *object-like, static* fashion (3). Therefore, we apply the same approach as above in order to create an instance `o` of organism `Org` (lines 16 to 18). However, we do not await its termination. Instead, `o` runs concurrently to any code of the calling trail (line 19). Since the organism is statically bound to variable `o`, we can easily refer to it in the future – independent of whether `o` is still alive or has already run to completion (line 20). By this, the static assignment allows future interaction and to keep control over the organism's lifetime. For example, if `o` goes out of scope, the corresponding organism immediately terminates.

Third, they can be used in an *object-like, dynamic* fashion (4). In contrast to the static approach, dynamic organisms are, in principle, not distinguishable – they have no explicit identity. Instead, they are aggregated and live in a common *pool*. A pool aids as a container for organisms of the same class and is declared using Céu's **`pool`**-keyword.

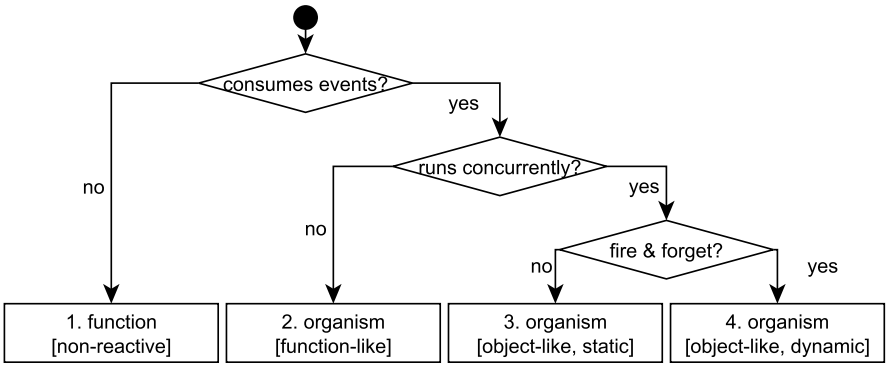

Figure 5.3: Best practice for choosing and deploying abstraction entities in Céu

Since dynamic organisms are not statically assigned to an individual variable, their lifetime is bound to the pool instead. In general, this enables dynamic organisms to outlive their scope of instantiation. We first declare a pool `orgs` which may hold up to 10 organisms of class `Org` (line 23). Then, we use Céu's **`spawn`**-statement in order to create a new, anonymous instance of `Org` in `orgs` (lines 24 to 26). All organisms in a pool execute concurrently.

Best Practice for Choosing a Suitable Deployment Approach In order to implement a certain functionality, one has to choose the suitable of the four approaches above. On this account, we provide a general best practice in Figure 5.3. The first decision that has to be taken is whether the functionality involves event handling or not. If not, we refer to this as *instantaneous code* which starts and terminates in the same reaction. In this case, use a function (1). Else, we denote this as *reactive code* which suggests the deployment of an organism. The second decision is whether the reactive code shall run concurrently to the calling trail or not. If sequential (non-concurrent) execution is intended, use an organism in a function-like fashion (2). Else, the concurrent execution may follow a fire-and-forget discipline or not. Fire-and-forget means that, after the code has started execution, we never need to explicitly refer to it again. If future referencing is required, use an organism in an object-like, static fashion (3). Else, apply an object-like, dynamic approach (4).

Adoption for Designing the Fieldbus Driver Architecture In Chapter 3, we considered the environment of traditional embedded applications static while Internet connectivity exposes a dynamic nature. The requirements for the fieldbus driver show this dichotomy too since it must serve the web server and the fieldbus at the same time. On the one hand, the fieldbus is a physical piece of hardware that outlives any execution instance of the fieldbus driver software. The number of connected fieldbuses is fixed and known at development time. Thus, we consider the fieldbus a *real entity*. On the other hand, EMS requests do not map to any physical resource. They spawn and disappear during execution of the fieldbus driver. The number of concurrent EMS requests may

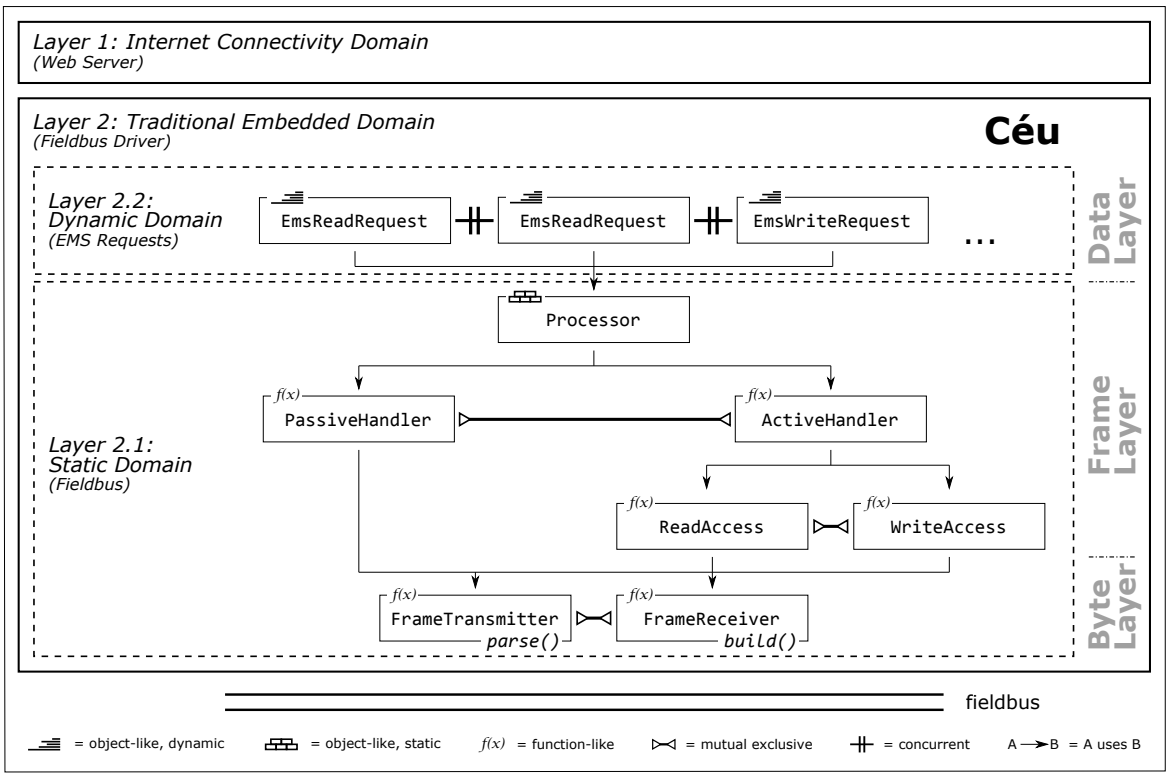

Figure 5.4: Layered architecture of the fieldbus driver in Céu

vary during runtime. Thus, we consider them *virtual entities*. Consequently, the fieldbus driver must deal with real and virtual entities at the same time in a reactive fashion.

As mentioned by Sant'Anna et al. [SIR15], it is a usual approach in embedded software design to map real entities to a static piece of code in a one-to-one relationship. Virtual entities, in contrast, usually require dynamic allocation on demand in order to handle their individual life cycles. As depicted in Figure 5.4, we consistently separate these different concerns into a static and a dynamic layer in our fieldbus driver architecture.

In the static Layer 2.1, we aim to hide all the complexity involved in fieldbus communication in a single abstraction entity on top-level. Since communication requires event handling, we implement an organism `Processor` on that purpose. Fieldbus interaction has to be performed continuously, independent of whether there are any EMS requests or not. Thus, `Processor` and the management of EMS requests must run concurrently. `Processor` represents the fieldbus in software. Thus, for realizing EMS requests, recurring referencing is mandatory. For these reasons, we choose an object-like, static approach (3) which bounds an instance of `Processor` to a dedicated variable that has application lifetime. Furthermore, we decompose the complexity of `Processor` into six organisms which all adopt a function-like approach (2). `PassiveHandler` and `ActiveHandler` implement $\texttt{Respond}_{\texttt{FL}}$ and $\texttt{Request}_{\texttt{FL}}$ respectively, thereby realizing Passive and Active Mode. Both are mutually exclusive and hence must run in sequence within `Processor`. `PassiveHandler` uses `FrameReceiver` and `FrameTransmitter` in order to perform $\texttt{Receive}_{\texttt{BL}}$ and $\texttt{Transmit}_{\texttt{BL}}$.

Again, both are mutually exclusive and must run in sequence within `PassiveHandler`. Internally, `FrameReceiver` takes advantage of function (1) `parse` in order to hide the complexity of parsing a received frame. `FrameTransmitter` takes the same approach by using function `build` to create the frame's raw byte sequence for transmission. `ActiveHandler` might perform a read or write access on the fieldbus which manifests in dedicated organisms `ReadAccess` and `WriteAccess` respectively. For the same reason as above, they require sequential execution. Both take advantage of `FrameReceiver` and `FrameTransmitter` to fulfill their task. On the lowest level, `FrameReceiver` and `FrameTransmitter` implement $\texttt{Receive}_{\texttt{BL}}$ and $\texttt{Transmit}_{\texttt{BL}}$ for single frame reception and transmission.

In the dynamic Layer 2.2, we aim to assign a single abstraction entity to each individual EMS request. Since EMS requests rely on event handling too, we take advantage of two organisms `EmsReadRequest` and `EmsWriteRequest` which distinguish between read and write EMS requests respectively. However, in contrast to the fieldbus, EMS requests are subject to a fire-and-forget policy. Once an organism is instantiated and assigned, it behaves like an independent unit of execution which is responsible for processing the request. It should execute concurrently to the remaining program and automatically disappear once the corresponding EMS request has been processed completely. For these reasons, we adopt an object-like, dynamic approach (4) in this case.

We outline the implementation of above organisms in the following sections.

5.2 Function-Oriented Design

Function-oriented design is a classical method of software engineering. It aids in breaking a complex problem into smaller parts that are easier to comprehend, program and maintain. Therefore, it focuses on identifying the major software functionalities and performs their stepwise refinement in a hierarchical top-down manner [BF14, ch. 2]. In this section, we consider how Céu's language-support addresses the drawbacks elaborated in Chapter 4 and particularly recovers function-oriented decomposition in the reactive domain.

5.2.1 Basic Functions

Basic functions do not rely on any other function. If we consider the entire function-oriented system as a tree, basic functions are the leafs. With respect to our fieldbus driver architecture in Section 5.1.3, `FrameReceiver` and `FrameTransmitter` are basic functions that contain the fundamental knowledge on how to react on the external fieldbus events `BYTE`, `BREAK` and `ERROR`. In this section, we outline the implementation of these organisms and demonstrate how they introduce encapsulation to the reactive domain, thereby recovering from `GOTO`-like code execution.

Implementation of $\texttt{Receive}_{\texttt{BL}}$ In Listing 5.5, we exemplify organism `FrameReceiver` featuring $\texttt{Receive}_{\texttt{BL}}$. Through its public interface it provides the captured and parsed

frame `fRx` (line 7) to the caller. The execution body (lines 8 to 49) contains the implementation of the private function `parse` (lines 9 to 22) and the actual, reactive behavior of `FrameReceiver` (lines 25 to 48).

```
1  /**
2   * Captures and parses a single frame.
3   * \param[out] fRx  high-level description of the received and parsed frame
4   * \return          0 success, -1 buffer overflow, -2 idle timeout, -3 malformed byte
5   */
6  class FrameReceiver with
7    var Frame& fRx;
8  do
9    function (u8[MAX_FRAME]& fRaw, Frame& fParsed)=>void parse do
10     var u8 crcOk = checkCRC(&fRaw);
11     <...>
12     if crcOk then
13       if <is-read> then
14         fParsed = Frame.MESSAGE_READ();
15         <...>
16       else/if <is-write> then
17         fParsed = Frame.MESSAGE_WRITE();
18         <...>
19     else
20       fParsed = Frame.INVALID();
21     end
22   end
24   // Step 1.1: Capture start of frame.
25   var u8[MAX_FRAME] fRaw;
26   var u8 bRx = await BYTE;
27   fRaw = [] .. fRaw .. [bRx];
28   // Step 1.2: Capture remaining bytes.
29   loop do
30     par/or do // Trail A
31       var u8 bRx = await BYTE;
32       if $fRaw == MAX_FRAME then
33         escape -1;
34       end
35       fRaw = [] .. fRaw .. [bRx];
36     with      // Trail B
37       await BREAK;
38       break;
39     with      // Trail C
40       await ERROR;
41       escape -3;
42     with      // Trail D
43       await 350ms;
44       escape -2;
45     end
46   end
47   parse(&fRaw, &this.fRx);
48   escape 0;
49 end
```

Listing 5.5: Implementation of organism `FrameReceiver`

We use a Céu vector `fRaw` to store the frame bytes (line 25). A vector is an array augmented by automatic and runtime-checked length management which provides detailed information about invalid memory accesses. This increases safety and considerably simplifies debugging. The vector's capacity is equal to the maximum frame length and statically defined by the pre-processor macro `MAX_FRAME` at compile time. By taking

advantage of Céu's **await**-statement we wait for the start of the next frame (line 26). A frame start needs to be a valid byte. Thus, **await** BYTE explicitly awaits BYTE only. Any other occurrence of BREAK and ERROR is automatically discarded. On BYTE, the trail resumes and **await** returns the received byte value which is temporarily stored in bRx. Then, we append the first frame byte to fRaw (line 27).

In contrast to the first, the remaining frame bytes must be checked for content and time accuracy. The approach is similar for every byte. Thus, it is reasonable to use Céu's **loop** (lines 29 to 46) in order to iterate over all further bytes. Since fieldbus frames do not contain any length information, we do not know the length of the current frame in advance. On this account, we deploy an infinite loop which exits once the end-of-frame character is hit. In the loop, we have to consider all possible communication scenarios for each incoming character. Therefore, we take advantage of Céu's **par**-block (lines 30 to 45) which splits the original main trail into four concurrent trails. This allows to await and handle the related fieldbus events concurrently.

The first trail (lines 31 to 35) handles a valid byte. We await BYTE and temporarily store the byte value in bRx (line 31). If fRaw is already full (line 32), this indicates a communication error and causes $\texttt{Receive}_{\texttt{BL}}$ to fail. In this case, we use **escape** to immediately terminate the entire FrameReceiver and return a proper error code (line 33). Else, the byte is appended to the vector (line 35). Due to the orthogonal **or**-abortion mechanism, all concurrent trails subsequently rejoin and the next loop iteration starts. This respawns all concurrent trails.

The second trail (lines 37 to 38) handles an end-of-frame character. We await BREAK (line 37) and execute **break** (Line 38). The latter immediately terminates all trails of the surrounding **par**-block and regularly exits the loop – the frame capture has been successfully completed. Consequently, the control flow advances to line 47.

The third trail (lines 40 to 41) handles an erroneous byte. We await ERROR (line 40). Since a malformed byte invalidates the whole frame $\texttt{Receive}_{\texttt{BL}}$ fails. Thus, we immediately terminate FrameReceiver using **escape**, thereby returning an error code (line 41).

The fourth trail (lines 43 to 44) finally checks for correct inter-byte timing. We take advantage of Céu's adoption of physical time in order to await the passage of the idle time t_i (line 43). If none of BYTE, BREAK and ERROR occur within 350 ms, **await** 350ms will return and **escape** (line 44) immediately terminates FrameReceiver due to the timing error.

To sum it up, the infinite loop (lines 29 to 46) keeps running as long as the fieldbus driver receives valid bytes and the vector fRaw is not full. While a communication error – buffer overflow (line 32), malformed byte (line 40) or timing error (line 43) – terminates the entire FrameReceiver, an end-of-frame character (line 37) leads to a regular loop exit only.

After a successful capture, execution continues in line 47. We call function parse in order to parse the raw byte sequence in fRaw. Lines 9 to 22 illustrate its implementation which relies on simple comparisons and conditional code execution. Note that parse calls another function checkCRC for verifying the frame content (line 10). Subsequently, parse populates the interface variable fRx of FrameReceiver through fParsed with the parsed frame data (lines 12 to 21). Finally, **escape** 0 regularly terminates FrameReceiver (line 48).

Implementation of `Transmit`$_{BL}$ In Listing 5.6, we exemplify organism `FrameTransmitter` featuring `Transmit`$_{BL}$. Through its public interface it obtains a high-level, structured description `fTx` (line 7) of the frame to send from the caller. The execution body (lines 8 to 47) contains an implementation of the private function `build` (lines 9 to 17) and the actual, reactive behavior of `FrameTransmitter` (lines 20 to 47).

```
1  /**
2   * Builds and transmits a single frame.
3   * \param[in] fTx  high-level description of the frame to send
4   * \return 0 success, -1 transmission error, -2 mirror timeout, -3 malformed mirror
5   */
6  class FrameTransmitter with
7    var Frame& fTx;
8  do
9    function (Frame& fToBuild, u8[MAX_FRAME]& fBuilt)=>void build do
10     if fToBuild.ACK_OK then
11       fBuilt = [] .. fBuilt .. [ACK_OK];
12     else/if fToBuild.MESSAGE_READ then
13       <...>
14     else/if fToBuild.MESSAGE_WRITE then
15       <...>
16     end
17   end

19   // Step 4.1: Transmit frame bytes.
20   var u8[MAX_FRAME] fRaw;
21   build(&fTx, &fRaw);
22   loop/MAX_FRAME i in $fRaw do
23     var int r = emit TX_BYTE => fRaw[i];
24     if r != 0 then
25       escape -1;
26     end
27     par/or do
28       var u8 mirror = await BYTE;
29       if mirror != fRaw[i] then
30         escape -3;
31       end
32     with
33       await BREAK;
34       escape -3;
35     with
36       await ERROR;
37       escape -3;
38     with
39       await 42ms;
40       escape -2;
41     end
42   end
43   // Step 4.2: Transmit end-of-frame.
44   var int r = emit TX_BREAK;
45   <handle-BREAK-mirror-likewise>
46   escape 0;
47 end
```

Listing 5.6: Implementation of organism `FrameTransmitter`

First, we transmit the frame bytes. Vector `fRaw` is used to store the byte sequence to be sent (line 20). Before transmission, the high-level description `fTx` must be serialized to a raw byte sequence. Therefore, we call function `build` (line 21). Lines 9 to 17 illustrate the corresponding conversion. Subsequently, the byte sequence in `fRaw` is

ready for transmission. Transmitting a single byte onto the fieldbus requires the same approach for each byte. Thus, we take advantage of CÉU's loop (lines 22 to 42) in order to perform the same procedure for all bytes in `fRaw`. Since the number of loop iterations is equal to the number of bytes in `fRaw`, we deploy a finite loop in this case. It keeps running until its control variable `i` reaches `$fRaw` which is the current frame length. In order to guarantee bounded execution, CÉU's safety concept requires to statically define a maximum limit for loop iterations at compile time. Since `i` might be modified within loop iterations, the loop might never exit otherwise. Once `MAX_FRAME` loop iterations have been performed, the loop exits independent of the value of `i`.

Next, we take advantage of CÉU's **`emit`**-statement in order to transmit a single byte (line 23). **`emit`** triggers the external output event `TX_BYTE`. This causes a handshake with the fieldbus driver's system environment, thereby executing output handler function `ceu_sys_output_TX_BYTE` of the platform interface code. It passes the request for byte transmission to the Linux operating system. Note that `=> fRaw[i]` attaches the corresponding byte value as payload to `TX_BYTE`. In its return value `r`, **`emit`** notifies about whether the handshake has been successful or not. If the transmission request cannot be forwarded to the operating system (line 24), $\texttt{Transmit}_{\texttt{BL}}$ fails. **`escape`** consequently terminates the entire `FrameTransmitter`, thereby returning a proper error code (line 25). On success, we use a **`par`**-block (lines 27 to 41) in order to handle the different communication scenarios with respect to the mirror byte.

The first trail (lines 28 to 31) handles a valid mirror. We await `BYTE` and temporarily store it in `mirror` (line 28). Then, we compare it to `fRaw[i]` which is the byte previously sent (line 29). If they do not match, the mirror's content is erroneous and hence $\texttt{Transmit}_{\texttt{BL}}$ fails. Thus, **`escape`** immediately terminates `FrameTransmitter` (line 30). Else, the concurrent trails rejoin and the next loop iteration starts.

The second (lines 33 to 34) and the third (lines 36 to 37) trail handle content errors similarly. During byte transmission, an end-of-frame character (line 33) is always an invalid mirror and causes $\texttt{Transmit}_{\texttt{BL}}$ to fail. The same applies for a malformed character (line 36). Both cause `FrameTransmitter` to terminate (lines 34 and 37).

The fourth trail (lines 39 to 40) finally checks the mirror time. We await the passage of mirror time t_m (line 39). If none of `BYTE`, `BREAK` and `ERROR` occur within 42 ms, **`await`** `42ms` will return and **`escape`** (line 39) immediately terminates `FrameTransmitter` due to the timing error.

Once all bytes in `fRaw` have been transmitted successfully, the same approach is applied in order to finally transmit the end-of-frame character (line 44 and the following).

Linear Control Flow With respect to our considerations in Chapter 4, the single entry point model turns out to be inappropriate for reactive applications from a software engineering point of view. CÉU takes a fundamentally different event handling approach. Conceptually, each call to **`await`** comprises an entry point for the specified event into the code. On the one hand, **`await`**-statements can be used in sequence such as presented in Listing 5.5, lines 26 and 31. On the other hand, by taking advantage of parallel composition, the same or different events can be awaited concurrently such

as in Listing 5.5, lines 31, 40 and 43. Thus, CÉU's event handling relies on a multiple entry points model which realizes a passive and blocking waiting strategy at multiple code locations in one or several trails. This corresponds to our considered strategies 7 and 15 in Section 4.2.1. Figure 5.5 illustrates the common programming scheme. In the sequential fashion (a), the first BYTE is awaited in line 1 and causes execution of reaction code in line 2. The second one is awaited in line 3 and causes execution of code in line 4 and so on. In the concurrent fashion (b), a single BYTE is awaited in line 2 and 5 at the same time. On occurrence, it causes execution of reaction code in line 3, then in line 6 (see Section 2.3).

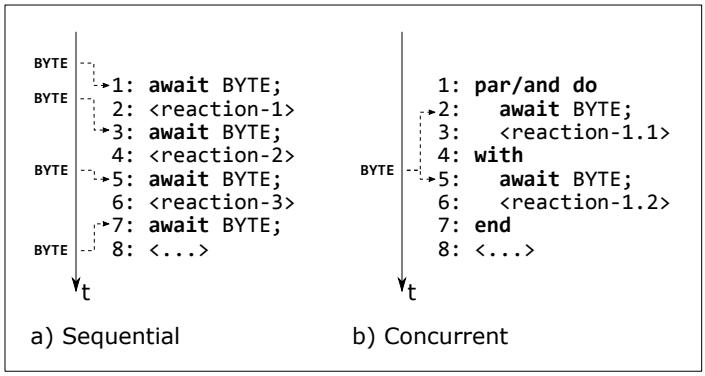

Figure 5.5: Programming scheme of the multiple entry points model in CÉU

Due to the multiple entry points model, each event can be easily distinguished in the code. In particular, it allows to assign individual reaction code to single events and, by taking advantage of CÉU's **loop**, also to groups of events. In `FrameReceiver` (Listing 5.5), for example, we specify a different behavior for the first received valid frame byte than for the remainder. While the first one is stored without checking for an overflow of `fRaw` (line 27), the group of remaining bytes requires the sanity check (line 32). Due to its prior initialization in line 25, it is obvious that `fRaw` is guaranteed to be empty when the first byte arrives in line 26. Note that this obviousness is only provided by the fact that **await** retains the current execution context across reactions. On the one hand, this eliminates any need for manual stack management and global variables. `fRaw` is local to `FrameReceiver` but its content is not discarded across reactions to BYTE. On the other hand, it makes manual state management and complex state machine logic obsolete. When we reach the loop in line 29 for example, it is obvious that the frame start already has been captured successfully. There is no need to encode that piece of information into a state such as `RECEIVING`. The internal state of `FrameReceiver` is determined by the current progress of its control flow. Thus, when considering a certain line of code, for example for debugging purposes, one can easily track the history of previous actions and decisions since they directly manifest in the prior sequence of statements. In CÉU, state is implicit and control flow is explicit.

As it turns out, CÉU's **await**-statement effectively slices control flow into event reactions while retaining its linear nature. Since software developers naturally think in

sequences, we believe that the recovery of a sequential, non-torn control flow is key towards improved comprehensibility.

Decomposition and Encapsulation By taking advantage of CÉU's organisms, we are able to break today's monolithic state machine design into smaller, separated and independent functionalities. `FrameReceiver` and `FrameTransmitter` encapsulate all required data and switching logic locally. This eliminates any code scattering and separates communication concerns. We believe that the provided encapsulation and interfacing capabilities effectively reduce the implementation and cognitive complexity involved. In particular, we encounter the following benefits:

First, although `FrameReceiver` and `FrameTransmitter` share the same input events, they can be implemented and modified without affecting each other – local changes only have local impacts. Thus, they are consistently isolated. This allows parallel programming by different developers and independent testing (see Section 5.4) which simplifies debugging. This is not true for the state machine approach. Adapting a single state or transition potentially changes the overall behavior and hence may lead to side effects across different functionalities which are hard to track.

Second, `FrameReceiver` and `FrameTransmitter` can be implemented once and reused multiple times throughout the code. This avoids code redundancy and allows to change code in one place, recompile and everything is consistently updated. For example, we use the same implementation of `FrameReceiver` and `FrameTransmitter` within `PassiveHandler`, `ReadAccess` and `WriteAccess`. Also, we only need to test an organism's implementation once and can be sure that this particular piece of software behaves correct independent of the location it is called.

Third, local variables are contained and in scope only as needed. For example, the raw frame buffer `fRaw` is only required for actually capturing or transmitting a frame. All subsequent functionalities in the fieldbus driver work on higher-level, parsed frames only. For this reason, `fRaw` locally resides in `FrameReceiver` and `FrameTransmitter` respectively. On organism termination, its corresponding memory is automatically freed from stack and can be reused. The local scope effectively rules out any harmful side-effects across organisms on language-level. For example, it is obvious that the same frame buffer memory can never be used for $\texttt{Receive}_{\texttt{BL}}$ and $\texttt{Transmit}_{\texttt{BL}}$ at the same time.

Fourth, due to their non-scattered language-level interface, the manifestation of `FrameReceiver` and `FrameTransmitter` is naturally delimited against the remaining code base. This makes it simple to grasp the overall architecture and gives a better idea of how the program actually works. Its concise definition of input and output values considerably simplifies to comprehend how the remaining program performs interaction with `FrameReceiver` and `FrameTransmitter` respectively. The ability to automatically provide a return value on completion is a comfortable way to indicate whether a sequence of event reactions in the past has been successful or not. For example, in `FrameReceiver` and `FrameTransmitter` we return proper error codes depending on the communication error scenario. This also eliminates any need for performing side-effects on global memory in

order to transfer data across organisms. Finally, it enables to easily outsource them in a dedicated software module such as *byteProcessing.ceu*.

Fifth, as presented in Listings 5.5 and 5.6, lines 1 to 5, in CÉU it is trivial to document each reactive functionality separately. We can independently specify the behavior of `FrameReceiver` and `FrameTransmitter` solely based on their input and output parameters as well as their return values. This generally leads to concise and readable documentations. In the course of testing, we can use this prose text specification in order to derive and perform proper black-box tests (see Section 5.4). Furthermore, the local language-level interface promotes the deployment of a structured documentation style that may be used for state-of-the-art, automatic source code documentation generation such as provided by the *Doxygen* [Hee16] tool[4].

5.2.2 Composite Functions

Composite functions take advantage of other composite or basic functions in order to implement more complex behavior. With respect to our fieldbus driver architecture in Section 5.1.3, `PassiveHandler`, `ActiveHandler`, `ReadAccess` and `WriteAccess` are composite functions. In this section, we outline the implementation of these organisms and demonstrate how they introduce composability to the reactive domain.

Implementation of $\texttt{Respond}_{\texttt{FL}}$ In Listing 5.7, we outline `PassiveHandler` where each loop iteration (lines 5 to 24) performs one run of $\texttt{Respond}_{\texttt{FL}}$. First, we use `FrameReceiver` in a function-like fashion in order to capture a frame. Therefore, we create an anonymous instance and wait for it running to completion using the **`do`**-instantiation (lines 8 to 10). Thereby, we provide a buffer `fRx` for storing the captured and parsed frame (line 9). On termination, we use its return value `ret` to decide whether the capture has been successful or not (line 11). If not, we take advantage of CÉU's **`continue`** keyword which immediately causes a new loop iteration to start (line 12). By this, the malformed frame is discarded and further processing (lines 15 to 24) skipped. Second, a valid frame is checked for its type (line 15) and subsequently processed (line 16). In case of a read request frame, we just retrieve the corresponding data from the gateway's internal memory. Next, we take the same approach by using `FrameTransmitter` for building and transmitting the response frame (lines 19 to 21). On its instantiation we pass the frame data to be sent via `fTx` in its interface (line 20). Other frame types are handled similar (line 22). Finally, the next loop iteration starts a new run of $\texttt{Respond}_{\texttt{FL}}$.

Implementation of $\texttt{Request}_{\texttt{FL}}$ Organism `ActiveHandler` implements $\texttt{Request}_{\texttt{FL}}$ and takes advantage of `ReadAccess` and `WriteAccess`. `ReadAccess` retrieves the data whereas `WriteAccess` updates the data of another fieldbus participant. Both take a similar implementation approach. For this reason, we only outline `WriteAccess` in Listing 5.8. First, we use `FrameTransmitter` for sending a write request frame (lines 6 to 8). After

[4]Note that Doxygen does not yet support the language CÉU.

```
1  /* @file frameLayer.ceu */
2  class PassiveHandler with
3    <...>
4  do
5    loop do
6      // Step 1: Receive frame.
7      var Frame fRx = <...>;
8      var int ret = do FrameReceiver with
9        this.fRx = &fRx;
10     end;
11     if ret != 0 then
12       continue;
13     end
14     // Step 2: Process frame.
15     if fRx.MESSAGE_READ then
16       <process-message>
17     // Step 3+4: Build and transmit response frame.
18       var Frame fTx = <...>;
19       var int ret = do FrameTransmitter with
20         this.fTx = &fTx;
21       end;
22     <handle-other-frame-types>
23     end
24   end
25 end
```

Listing 5.7: Implementation of organism `PassiveHandler`

successful transmission, the further behavior depends on whether it was a uni- or broadcast request (line 10). In contrast to a unicast, broadcasts are not acknowledged. However, before proceeding, the sender must provide some time for other devices to receive and process the broadcast message. To achieve this, we use **`await`** `250ms` in order to realize a corresponding delay (line 11). After the passage of 250 ms, `WriteAccess` immediately terminates (line 12) – there is no further action required in case of a broadcast. Else, we use `FrameReceiver` to receive the response frame (lines 20 to 22). In the concurrent trail we await 225 ms which is the maximum response time allowed (line 17). If no response is received within that time, the write access fails and hence `WriteAccess` terminates (line 26). Finally, if a response is received in time, we check its type (line 25). If it is not a positive acknowledgment, something went wrong and `WriteAccess` terminates with an error code, too (line 26).

In Listing 5.9, we outline `ActiveHandler` which spawns two concurrent trails. In the first trail (lines 9 to 32) each loop iteration (lines 10 to 31) performs one run of $\texttt{Request}_{\texttt{FL}}$. Therefore, we call the proper access organism depending on the current type of request. In case of a read request frame (lines 14 to 21), we call `ReadAccess` (lines 15 to 18). Once completed, we emit `response` (line 20) in order to notify the caller of `ActiveHandler` about the successfully received response frame. In case of a write request frame (lines 22 to 25), we call `WriteAccess` (lines 23 to 25) instead. Note that in this case we do not emit `response` since a write request is only acknowledged but does not return any data to the caller of `ActiveHandler`.

```
1  /* @file frameLayer.ceu */
2  class WriteAccess with
3    var Frame& fReq;
4  do
5    // Step 1: Transmit write request frame
6    var int ret = do FrameTransmitter with
7      this.fTx = &outer.fReq;
8    end;
9    <...>
10   if <is-broadcast> then
11     await 250ms; // delay
12     escape 0;
13   end
14   // Step 2: Receive response frame
15   var Frame fRes = <...>;
16   par/or do
17     await 225ms;
18     escape -2;
19   with
20     ret = do FrameReceiver with
21       this.fRx = &fRes;
22     end;
23     <...>
24   end
25   if not fRes.ACK_OK then
26     escape -3;
27   end
28   escape 0;
29 end
```

Listing 5.8: Implementation of organism `WriteAccess`

Remember that $\texttt{Request}_{\texttt{FL}}$ is only allowed to execute during token time (see Section 2.1). The fieldbus driver must not send any new request frames once the token active time t_{ta} (800 ms) has been exceeded. On this account, the second trail (lines 32 to 36) is responsible for controlling the time-dependent behavior of the first trail. In contrast to our previous approaches, we must not abort the first trail immediately once 800 ms are exceeded. This would cause `ReadAccess` respectively `WriteAccess` to completely terminate too, thereby stopping to await and process the possibly pending response frame. Instead, we need a different approach that implements a rather soft interruption than a hard abortion. Therefore, we use the flag variable `tx_active` which is shared between both trails. At start-up, `tx_active` is set to **`true`** which indicates that 800 ms have not been exceeded yet. (line 8). In the second trail, we first await 800 ms (line 33) and change `tx_active` to **`false`** once t_{ta} has been elapsed (line 34). Note that we must not yet terminate the second trail since this would cause the first trail to terminate too due to the **`or`**-abortion (line 9). Instead, we await the token passive time t_{tp} (200 ms) (line 35). This gives the fieldbus driver the chance to capture the pending response frame. If both, t_{ta} and t_{tp}, are exceeded, the complete token time t_t is over and we are forced to return to Passive Mode. Thus, the second trail finally causes both trails to rejoin and `ActiveHandler` terminates.

Concurrently, the first trail monitors `tx_active`. Setting `tx_active` to **`false`** does not cause the first trail to terminate immediately. Instead, the current execution instance of `ReadAccess` respectively `WriteAccess` can run to completion unimpeded. Then, in the next iteration, we use `tx_active` to decide whether we are allowed to start a new

```
1  /* @file frameLayer.ceu */
2  class ActiveHandler with
3    var Frame& fReq;
4    var Frame& fRes;
5    event void response;
6    <...>
7  do
8    var bool tx_active = true;
9    par/or do
10     loop do
11       if tx_active <...> then
12         <...>
13         var int ret = -1;
14         if fReq.MESSAGE_READ then
15           ret = do ReadAccess with
16             this.fReq = &outer.fReq;
17             this.fRes = &outer.fRes;
18           end;
19           if ret == 0 then
20             emit this.response;
21           end
22         else/if fReq.MESSAGE_WRITE then
23           ret = do WriteAccess with
24             this.fReq = &outer.fReq;
25           end;
26         end
27         <...>
28       else
29         break;
30       end
31     end
32   with
33     await 500ms;
34     tx_active = false;
35     await 298ms;
36   end
37   <...>
38 end
```

Listing 5.9: Implementation of organism `ActiveHandler`

request-response cycle in our loop (line 11). If not, **`break`** exits the loop (line 29) which terminates the first trail, the **`par`**-block and finally `ActiveHandler`.

Hierarchical Composition As demonstrated above, function-like usage of organisms in Céu allows to compose reactive behavior in a hierarchical fashion. This enables to apply a top-down divide-and-conquer strategy to our reimplementation, thereby generally reducing implementation effort and cognitive complexity likewise. In particular, we encounter the following benefits:

First, organisms reduce the complexity on each level of abstraction by providing a declarative label which allows smooth, compiler-checked calls. For example, for `PassiveHandler` in Listing 5.7 we take advantage of `FrameTransmitter` in order to hide all the details of triggering $\texttt{Transmit}_{\texttt{BL}}$ in the byte layer, performing $\texttt{Transmit}_{\texttt{BL}}$ subsequently and finally passing the captured frame up to the frame layer (lines 19 to 21). This allows to express the request-response scenario on a top level in a very concise and readable way. Reasoning about mutual exclusion between $\texttt{Receive}_{\texttt{BL}}$ and $\texttt{Transmit}_{\texttt{BL}}$, for example, becomes trivial due to the linear control flow in `PassiveHandler`.

Because `FrameReceiver` and `FrameTransmitter` are used in a sequential fashion (lines 8 and 19), CÉU guarantees that they are never alive and hence never react at the same time.

Second, it is easy to define hierarchical dependencies among reactive functionalities. For example, `PassiveHandler` is superior and has control over `FrameReceiver` and `FrameTransmitter`. This implies that if `PassiveHandler` terminates the current instance of `FrameReceiver` or `FrameTransmitter` aborts too. Thus, the termination of a composite functionality is automatically propagated down the hierarchy to any sub-functionalities in use. Note that in the asynchronous approach concurrent state machines are required to manipulate each other in order to synchronize their states accordingly. Also, if `FrameReceiver` and `FrameTransmitter` are respawned in the next loop iteration of `PassiveHandler` (lines 5 to 24) they are automatically reinitialized and restarted. This effectively reduces code verbosity since there is no need for any additional code in order to reinitialize states or perform default transitions. Those operations are automatically performed by CÉU on organism instantiation and termination respectively.

Third, the hierarchical dependency allows to apply a unidirectional coupling between organisms. Only the `PassiveHandler` (the caller) knows `FrameReceiver` and `FrameTransmitter` (the called) and hence can use them. `FrameReceiver` and `FrameTransmitter`, in their turn, do neither know `PassiveHandler` nor about each other. As presented in Section 5.1.3, this enables to divide the fieldbus driver into different layers where each layer serves the layer above it and is served by the layer below it. Each layer deals only with one aspect of the communication, for example byte, frame and data processing. This approach is typically adopted in network protocols in order to facilitate easier communication and better structure [BF14, ch. 13]. Moreover, changes in one layer do not affect other layers since they are independent. They promote modularity and make the software easier to comprehend and maintain. Also, note that code of `ActiveHandler`, `PassiveHandler`, `ReadAccess` and `WriteAccess` is independent of the way $\texttt{Receive}_{\texttt{BL}}$ and $\texttt{Transmit}_{\texttt{BL}}$ are actually performed on the fieldbus. This means it can be easily reused for other applications that rely on a similar request-response discipline. Thus, in CÉU it is simple to separate environment-dependent code into a dedicated environment abstraction layer given by `FrameReceiver` and `FrameTransmitter` in our application.

Fourth, there is no manual synchronization for inter-layer data exchange required anymore. For example, in Section 4.3.1, we elaborate the effort required for programming interaction between the byte and frame layers when performing $\texttt{Respond}_{\texttt{FL}}$ and $\texttt{Request}_{\texttt{FL}}$ respectively. Taking the example of $\texttt{Request}_{\texttt{FL}}$, `WriteAccess` in Listing 5.8 simply calls `FrameTransmitter` (line 6) in order to pass the frame to be sent (line 7) and automatically triggers its transmission at the same time. Calling an organism comprises a handshake on language-level which is intrinsically in sync with the control flow of the caller. This reduces implementation effort and makes code less error-prone.

Temporal Behavior Above organisms implement temporal behavior at different locations. This includes delays as well as timeouts. Due to CÉU's language-level support

for interfacing physical time, their adoption appears to be a simple, straightforward approach.

In Céu, delays are language primitives. By taking advantage of **await** <time-interval> it is trivial to artificially suspend the calling trail until the specified amount of time has passed. In Listing 5.8 line 11, we use that feature in order to postpone the termination of `WriteAccess`. This prevents the calling `ActiveHandler` to instantaneously continue with the next request and hence to overstrain the remaining fieldbus participants. In contrast to the sequential asynchronous approach in Section 4.4, note that we do not have to perform any manual timer management. We even do not have to care about allocating a dedicated timer for that particular delay.

In combination with parallel composition, Céu's delays turn out to be a powerful language feature that allow to easily adopt timeouts too. In particular, we encounter two general types of timeouts: a *hard timeout* immediately aborts the assigned action while a *soft timeout* just notifies about the timeout, thereby giving the chance to properly complete the current operation. Hard timeouts are intrinsic to `FrameReceiver`, `FrameTransmitter`, `ReadAccess` and `WriteAccess` in order to monitor the timing correctness of fieldbus communication. In Céu, they can be easily implemented using the pattern in Listing 5.10. While the first trail executes the intended action (line 2), the second trail monitors the passage of time (line 4). Due to the **or**-abortion (line 1) the first trail is forced to rejoin once the assigned timeout interval has been exceeded. This approach is denoted as "watchdog pattern" by Sant'Anna et al. [SIR12].

```
1 par/or do // The action trail
2   <action>
3 with      // The timing trail
4   await <timeout-interval>;
5 end
```

Listing 5.10: Hard timeout in Céu

Due to our use case in `ActiveHandler`, we extend that pattern by a prior soft timeout in Listing 5.11 which allows the assigned action to proceed until the actual hard timeout exceeds. The flag `softTimeout` (line 1) aids as an inter-trail signal and indicates whether the soft timeout interval has been elapsed. The first trail executes the intended action (lines 3 to 5) which occasionally monitors `softTimeout` (line 4). The second trail first awaits the soft timeout interval (line 7). Once exceeded, it sets `softTimeout` to **true** (line 8). The first trail, in its turn, can take the necessary steps to regularly run to completion. Concurrently, the second trail awaits the hard timeout interval (line 9). Once exceeded, it finally behaves like the watchdog pattern above and terminates the first trail. Consequently, this approach adopts a soft timeout that is finally limited by a hard timeout.

For the sake of completeness, it is also possible to implement unlimited soft timeouts that only provide notification but do never force abortion. Therefore, we replace the

```
1  var bool softTimeout = false;
2  par/or do // The action trail
3    <...>
4    <check-for-softTimeout>
5    <...>
6  with      // The timing trail
7    await <soft-timeout-interval>;
8    softTimeout = true;
9    await <hard-timeout-interval>;
10 end
```

Listing 5.11: Limited soft timeout in CÉU

hard timeout interval with CÉU's **FOREVER**-keyword (line 9). Since **await FOREVER** never returns the second trail will never run to completion and hence never forces the first trail to rejoin. The action trail may also ignore `softTimeout` and run forever.

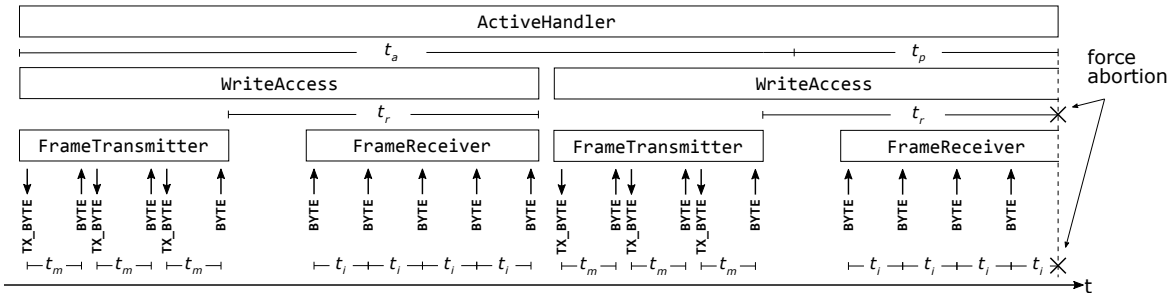

Figure 5.6: Hierarchical timing constraints for fieldbus communication

Furthermore, CÉU's hierarchical usage of organisms expands composability from the functional to the time domain too. This allows to easily specify complex, nested timing dependencies in a structured and readable way. Figure 5.6 illustrates an example on that purpose. Due to the sequential usage of `FrameTransmitter` and `FrameReceiver` in `WriteAccess`, it is obvious that timeout intervals t_m and t_i are mutually exclusive. Also, due to their sequential adoption in `FrameTransmitter` and `FrameReceiver` only one timeout interval – t_m or t_i – can be active at the same time. In contrast, timeout interval t_r is superordinate in `WriteAccess` and exists across several instances of t_i. This creates a hierarchical dependency among t_r and t_i which gives t_r precedence over t_i. For example, if t_r elapses the current t_i must be aborted too. In its turn, t_r is subordinate to the sum of t_a and t_p in `ActiveHandler`. As soon as $t_a + t_p$ elapses, the current t_r and also the current t_i have to be aborted. Note that this abortion is automatically propagated from `ActiveHandler` through `WriteAccess` down to `FrameReceiver`. Moreover, hierarchical timing constraints reduce complexity since, on each abstraction level, we only have to deal with those timing constraints that are relevant for the functionality of the current level. For example, in `FrameReceiver` we only care about t_i whereas in `WriteAccess` we only account for t_r. The coupling between t_i and t_r emerges implicitly by calling `FrameReceiver` within `WriteAccess`. Finally, in CÉU, relations between several timing constraints can be layered

in a hierarchical fashion which promotes abstraction and automatic abortion. This reduces the implementation and cognitive effort.

5.3 Object-Based Design

Object-oriented programming can aid as an effective tool towards good software quality. On this account, Gamma et al. [Gam+95] elaborate a number of world-wide recognized, object-oriented design patterns which aim to provide solutions to common problems that developers encounter when designing software. In traditional object-oriented programming, inheritance and polymorphism play a key role [BF14, ch. 2]. However, those concepts require highly dynamic memory usage which entails the risk of memory overflows. The lack of guaranteed bounded memory makes it generally incompatible with resource-constrained, embedded systems. Additionally, depending on its concrete adoption, it may affect real-time capabilities due to automatic, non-deterministic garbage collection, for example in Java Standard Edition, or lead to memory leaks due to manual memory allocation and deallocation, for example in C++. As a consequence, embedded software typically cannot take advantage of above patterns.

Note that object-oriented programming is generally possible in C, too. Structures and function pointers can be used to tie together data and functions. Nested structures and type casts allow to mimic inheritance and polymorphism. However, all these steps rely on manual implementation from scratch and are not compiler-checked. Schreiner [Sch94] illustrates that this is a very daunting and error-prone task which takes a lot of discipline.

Céu's organisms, in contrast, solely focus on the key idea of encapsulating data and associated functions into a single unit of abstraction – no inheritance or polymorphism. Any dynamic memory usage is deliberately omitted in order to prevent the violation of Céu's fundamental concepts of bounded execution time and memory. By this, Céu supports object-like abstractions on language-level but without the dynamics of an object-oriented language. We refer to this as *object-based* design. Despite these limitations, it appears that organisms are a powerful language feature. Due to Céu's efficient memory layout (see Section 2.3), they introduce object-based programming to the domain of resource-constraint embedded systems. In particular, in the following sections, we will exemplify how Céu's organisms enable to take advantage of the *Command*, *Facade*, *State*, *Observer* and *Chain of Responsibility* design patterns in order to improve the quality of embedded software. For each pattern, we first provide a short introduction. Second, we present the motivation for applying it in our use case. Third, we exemplarily illustrate its adoption in Céu. Fourth, we compare our adoption to the pattern, thereby discussing commonalities and differences. Fifth, we consider the gained software-engineering benefits for our application use case.

Reasoning About Concurrency Taking the example of the Observer pattern, Lee [Lee06] illustrates the difficulties and pitfalls that arise – even with language-level support such as the `synchronized` keyword in Java – when object-oriented programming

is exposed to asynchronous, thread-based concurrency. Data inconsistencies and non-determinism have to be pruned away by the developer in order to make code thread-safe, thereby incidentally inducing deadlock scenarios. Due to the synchronous model of execution, shared-memory access and hence all object-based programming, particularly the below adoption of design patterns, is intrinsically thread-safe in Céu without any additional implementation effort and risk of deadlocks [San+13].

5.3.1 Adoption of Command Pattern

> "Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations." [Gam+95]

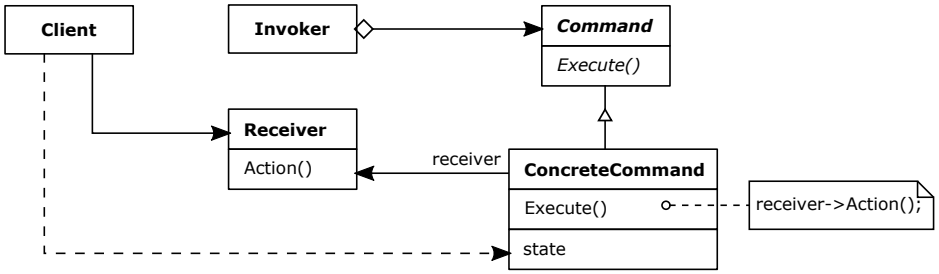

Figure 5.7: Structure of the *Command* pattern [Gam+95]

Figure 5.7 illustrates the structure: "*Command* declares an interface for executing an operation. *ConcreteCommand* [...] defines a binding between a Receiver object and an action [and] implements Execute by invoking the corresponding operation(s) on Receiver. *Client* [...] creates a ConcreteCommand object and sets its receiver. *Invoker* [...] asks the command to carry out the request. *Receiver* [...] knows how to perform the operations associated with carrying out a request. Any class may serve as a Receiver." [Gam+95]

Motivation The fieldbus driver accepts EMS requests by the external input event `EMS_REQUEST`. In general, those requests cannot be answered instantaneously. They require one or more executions of $\texttt{Request}_{\texttt{FL}}$ in order to get physically performed on the fieldbus. Due the fieldbus' medium access policy, $\texttt{Request}_{\texttt{FL}}$ can be only performed in Active Mode (see Section 2.1). Thus, EMS requests have to be enqueued and their actual execution postponed until the next communication time slot. As a consequence, specifying, queuing and executing EMS requests happens at different times. In particular, an EMS request must have a lifetime independent of the original `EMS_REQUEST` event. Also, the group of EMS requests is composed of read and write requests which require different implementations but should be stored in the same queue for processing. This suggests the adoption of the *Command* pattern.

Adoption In Listing 5.12, we declare input event `EMS_REQUEST` for accepting an EMS request from the web server (line 1) and output event `EMS_RESPONSE` for returning the corresponding EMS response (line 2). Their payload data types are user-defined structures which contain all the required information for the associated request and response respectively. `Processor` (lines 4 to 6) actually manages the requests on the fieldbus as exemplified in Section 5.3.2. In order to declare a common interface `IEmsRequest` for all incoming read and write EMS requests, we take advantage of CÉU's interface feature for organisms (lines 8 to 10). Note how line 9 requires all EMS requests to obtain a reference to an instance of `Processor`. Further, we provide two concrete request implementations `EmsReadRequest` (lines 12 to 21) and `EmsWriteRequest` (lines 23 to 25). Lines 13 and 24 apply the interface to them. Essentially, `EmsReadRequest` uses the instance `p` of `Processor` from its interface to perform the request. First, it writes the request frame to `p` (line 16). Second, it awaits the response (line 17). Third, it reads the response from `p` (line 18). Finally, it emits `EMS_RESPONSE`, thereby returning the response to the web server (line 20). `EmsWriteRequest` behaves similar but handles frame content differently.

The actual application code starts in line 27. Here, we first instantiate a pool `requests` which is a container for dynamic organisms of type `IEmsRequest`. Due to the common interface, it allows to store up to 20 instances of `EmsReadRequest` and `EmsWriteRequest` in total. Second, we instantiate a single instance `p` of `Processor` which should be used by all request organisms for interfacing the fieldbus (line 28). Third, we use the infinite loop in lines 29 to 40 in order to continuously accept an incoming EMS request from the web server. Therefore, we await `EMS_REQUEST` (line 30). In case of a read request (line 31), we use CÉU's **`spawn`** in order to create a new `EmsReadRequest` in the pool (line 32). Note how the reference to `p` gets injected into each newly created organism (line 33). In case of a write request (line 35), we take the same approach to create a new `EmsWriteRequest` (line 36). By this, we map each incoming EMS request to a proper request organism for processing and enqueuing it in `requests`.

Comparison In our adoption, `IEmsRequest` corresponds to the common Command interface while `EmsReadRequest` and `EmsWriteRequest` provide two ConcreteCommand implementations. The actual command execution is delegated to `Processor` (line 16) which aids as the *Receiver*. Since there is only one fieldbus physically available, the same *Receiver* is used for all commands. The top-level fieldbus driver is the *Client* which creates new commands during runtime (lines 32 and 36). In contrast to the pattern, the CÉU adoption does not require an explicit *Invoker*. This is due to the fact that in CÉU (dynamic) organisms allow to separately encapsulate data and control flow for each individual ConcreteCommand. While commands in the pattern are rather passive entities that require to get invoked by an execution context, commands in the CÉU adoption are active entities that concurrently execute their code themselves. Thus, in our approach, commands can be best compared to the notion of jobs which tie together a set of data and an assigned working progress that advances concurrently during runtime. Although `EmsReadRequest` and `EmsWriteRequest` immediately start running

```
1  input _data_request EMS_REQUEST;
2  output _data_response EMS_RESPONSE;
3  // Receiver
4  class Processor with
5    var Frame& fReq; var Frame& fRes; event void response; <...>
6  do <...> end
7  // Command
8  interface IEmsRequest with
9    var Processor& p; <...>
10 end
11 // ConcreteCommand A
12 class EmsReadRequest with
13   interface IEmsRequest;
14 do
15   <build-read-request-frame>
16   p.fReq = <read-request-frame>;  // delegate to Receiver
17   await this.p.response;
18   <read-response-frame> = this.p.fRes;
19   <process-response>
20   emit EMS_RESPONSE => <response-data>;
21 end
22 // ConcreteCommand B
23 class EmsWriteRequest with
24   interface IEmsRequest;
25 do <...> end
26 // Client
27 pool IEmsRequest[20] requests;
28 var Processor p with <...> end;
29 loop do
30   <...> = await EMS_REQUEST;
31   if <is-read-request> then
32     <...> = spawn EmsReadRequest in requests with // create and invoke
33       this.p = &p; // inject Receiver
34     end;
35   else/if <is-write-request> then
36     <...> = spawn EmsWriteRequest in requests with // create and invoke
37       this.p = &p; // inject Receiver
38     end;
39   end
40 end
```

Listing 5.12: Adoption of the *Command* pattern

after instantiation, the injected `Processor` internally takes care of postponing the actual command execution until the next instance of Active Mode (see Section 5.3.3). This becomes apparent due to the fact that `EmsReadRequest` respectively `EmsWriteRequest` have to wait for the corresponding reply (line 17).

Benefit In our adoption, new types of requests can be easily added without changing existing code. It only requires to provide additional organism classes that implement `IEmsRequest`. For example, the fieldbus provides additional *meta requests* which allow to retrieve information about which services are actually available in the concrete heating appliance setup. They also follow a request-response frame discipline – this means they can use `Processor` likewise – but require completely different data handling and interpretation. We can easily add a new organism, for example `EmsMetaRequest`, which encapsulates the required knowledge and store it in the same pool.

`IEmsRequest` decouples the organism that invokes the operation (the top-level fieldbus driver) from the one that knows how to perform it (`Processor`). This allows the top-level fieldbus driver to accept and enqueue any type of existing or potential future EMS requests in the same way. This is only possible because the top-level fieldbus driver only needs to know how to issue the request and not how it will be carried out. By taking advantage of dependency injection (lines 32 and 36), it is also possible to inject different *Receivers* for different fieldbus systems. For example, the gateway must generally serve different bus systems apart from EMS such as Controller Area Network (CAN), Meter-Bus (M-Bus), Modbus, etc. Our approach allows to inject a reference to a bus-specific `Processor` implementation for each individual incoming request. This enables to distinguish between different gateway product lines at compile time (statically) or at runtime if multiple buses must be served simultaneously (dynamically).

Finally, `EmsReadRequest` and `EmsWriteRequest` are first-class organisms that can be manipulated and extended like any other organism. For example, they can be extended by an internal event mechanism which emits events on each bus access or communication error for logging or debugging purposes (see Section 5.3.4).

5.3.2 Adoption of Facade Pattern

> "Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use." [Gam+95]

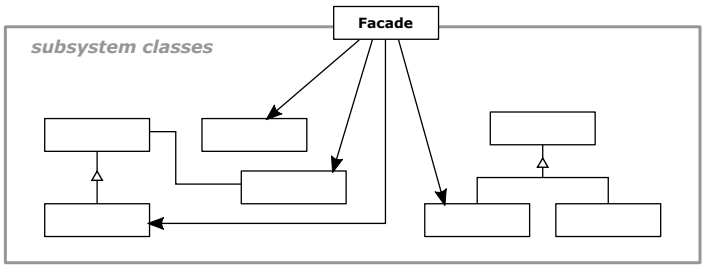

Figure 5.8: Structure of the *Facade* pattern [Gam+95]

Figure 5.8 illustrates the structure: "*Facade* [...] knows which subsystem classes are responsible for a request [and] delegates client requests to appropriate subsystem objects. *Subsystem classes* [...] implement subsystem functionality [and] handle work assigned by the Facade object. [They] have no knowledge of the facade; that is, they keep no references to it." [Gam+95]

Motivation In our fieldbus driver architecture the static layer is responsible for actually interfacing the fieldbus. Thus, organisms `EmsReadRequest` and `EmsWriteRequest` (see Section 5.3.1) from the dynamic layer must use the static layer in order to fulfill their task. However, as presented in Sections 5.1.3 and 5.2, we structured the static layer into several smaller organisms: `FrameReceiver`, `FrameTransmitter`, `ReadAccess`, `WriteAccess`,

`PassiveHandler` and `ActiveHandler`. On the one hand, this helps reduce complexity and makes the static subsystem more reusable and easier to customize. On the other hand, it becomes harder to use for the clients `EmsReadRequest` and `EmsWriteRequest` since they have to deal with a comparatively large number of organisms with specialized interfaces. Consequently, we aim for a single, simplified interface to the static subsystem which minimizes communication and dependencies between dynamic and static layer. This suggests the adoption of the structural *Facade* pattern.

Adoption In Listing 5.13, we implement `Processor` (lines 2 to 21) which acts as the Facade for the static subsystem. As illustrated in Section 5.3.1, it is injected into and used by the clients `EmsReadRequest` and `EmsWriteRequest` likewise. `Processor` knows which organisms of the static fieldbus driver layer are actually responsible for request processing during runtime and how to access them. For the dynamic layer, it provides a simple fieldbus communication interface (lines 4 to 7) based on a frame transmit (line 4) and receive (line 5) buffer as well as event `response` (line 6). The latter notifies when a new response in `fRes` is ready to be read out. In its implementation `Processor` essentially delegates the requests from `EmsReadRequest` and `EmsWriteRequest` to the appropriate organism during runtime. Usually, EMS requests are realized by `ActiveHandler` in Active Mode. That is, the fieldbus driver explicitly requests the data from the heating appliance. Therefore, `Processor` creates an instance `ah` of `ActiveHandler` (lines 14 to 18) and forwards the request frame by passing a reference to `fReq` (line 15). The response frame is returned similarly (line 16). This way, `EmsReadRequest`, for example, can send a request frame `fReq` to `ah` trough `Processor` and await the response `fRes` using `response`. Apart from that, some data is regularly published by broadcasts without explicit request. Broadcast messages are processed by `PassiveHandler` in Passive Mode. Consequently, EMS requests can be implicitly fulfilled by `PassiveHandler` too. Therefore, `Processor` creates an instance `ph` of `PassiveHandler` (lines 10 to 12) and passes a reference to `fRes` (line 11). This allows `ph` to forward incoming broadcast messages to instances of `EmsReadRequest` and `EmsWriteRequest` respectively.

Comparison `Processor` corresponds to the Facade while `PassiveHandler` and `ActiveHandler` are two subsystem classes. `EmsReadRequest` and `EmsWriteRequest` are the clients that use the Facade. The adoption of the Facade pattern in Céu is straightforward. There are no notable differences.

Benefit `Processor` provides a single, simplified interface to the static fieldbus driver subsystem which hides all the complexity of instantiating and interacting with the contained set of organisms. Also, it comprises an entry point to the static fieldbus driver layer making the dynamic and the static layer solely communicate through `Processor`. This reduces the number of organisms `EmsReadRequest` and `EmsWriteRequest` have to deal with and hence makes the static layer easier to use. For example, they do not have to care about whether the fieldbus driver currently runs in Passive or

```
// Facade
class Processor with
  // simple interface
  var Frame& fReq;
  var Frame& fRes;
  event void response;
  <...>
do
  loop do
    var PassiveHandler ph with
      this.fRes = &outer.fRes;
    end;
    <...>
    var ActiveHandler ah with
      this.fReq = &outer.fReq;
      this.fRes = &outer.fRes;
      <...>
    end;
    <...>
  end
end
// Subsystem class A
class PassiveHandler with
  var Frame& fRes;
do <...> end
// Subsystem class B
class ActiveHandler with
  var Frame& fReq; var Frame& fRes; <...>
do <...> end
```

Listing 5.13: Adoption of the *Facade* pattern

Active Mode. Furthermore, `Processor` decouples `PassiveHandler` and `ActiveHandler` from `EmsReadRequest` and `EmsWriteRequest`. As a consequence, organisms of the static subsystem can be modified or even exchanged without affecting `EmsReadRequest` and `EmsWriteRequest`, thereby promoting independence and portability. For example, we can easily implement new organism classes that realize another fieldbus communication protocol such as CAN. In this case, `EmsReadRequest` and `EmsWriteRequest` only require adaptation with respect to payload handling. Also, it allows to implement the dynamic and static fieldbus driver layers independently and in parallel by several developers. Nonetheless, it is still possible to access static subsystem organisms directly. For example, we can implement a concurrent logging organism which continuously runs `FrameReceiver` in a loop, thereby capturing all received frames for debugging purposes.

5.3.3 Adoption of State Pattern

> "Allow an object to alter its behavior when its internal state changes. The object will appear to change its class." [Gam+95]

Figure 5.9 illustrates the structure: "*Context* [...] defines the interface of interest to clients [and] maintains an instance of a ConcreteState subclass that defines the current state. *State* [...] defines an interface for encapsulating the behavior associated with a particular state of the Context. Each [*ConcreteState*] subclass implements a behavior associated with a state of the Context." [Gam+95]

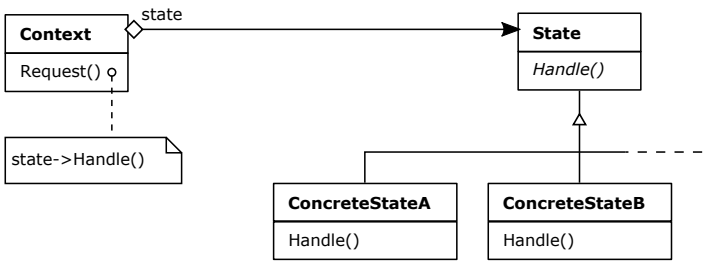

Figure 5.9: Structure of the *State* pattern [Gam+95]

Motivation As presented in Section 5.3.2, `Processor` acts as the actual software interface to the fieldbus. However, it reacts to incoming characters differently depending on its current state respectively operation mode. Thus, the behavior of `Processor` has to change at runtime depending on that state. Due to the drawbacks discussed in Section 4.2, we particularly aim to avoid any state machine-like implementation that relies on large, conditional code blocks for selecting the proper actions to take. This suggests the adoption of the *State* pattern.

Adoption In Listing 5.14, we implement `Processor` (lines 2 to 16) as the Context that has to change its behavior. `PassiveHandler` (lines 18 to 28) implements the behavior in Passive Mode while `ActiveHandler` (lines 30 to 41) realizes the behavior in Active Mode. The loop in `Processor` (lines 4 to 15) is used to toggle between the different behaviors during runtime. By default, `Processor` is in Passive Mode. Therefore, it creates an instance `ph` of `PassiveHandler` (line 5) and awaits its termination (line 7). In its internal loop (lines 20 to 27), `PassiveHandler` repeatedly receives a frame (line 22) and processes it subsequently. If the frame is of type *Token* (line 24), `Processor` must switch to Active Mode (see Section 2.1). Therefore, `PassiveHandler` exits its loop using **`break`** (line 25) and terminates. This causes **`await`** `ph` (line 7) to return. `Processor` proceeds, creates an instance `ah` of `ActiveHandler` (line 10) and awaits its termination (line 12), thereby switching to Active Mode. In its internal loop (lines 33 to 39), `ActiveHandler` performs EMS requests (line 35) assigned from `EmsReadRequest` and `EmsWriteRequest` respectively. If the communication time slot expired or there are no more requests to send (line 34) it exits its loop (line 37) and terminates. This causes **`await`** `ah` (line 12) to return and the current loop iteration in `Processor` to complete. The next iteration automatically creates a new instance of `PassiveHandler` (line 5), thereby returning to Passive Mode.

Comparison `Processor` corresponds to the Context while `PassiveHandler` and `ActiveHandler` represent two ConreteStates. In the reactive domain, the functionality that must differ depending on the current state is how the external input events, for example `BYTE`, `BREAK` and `ERROR`, are processed. In Céu, those events have global scope and hence can be awaited and processed by any organism. Thus, we do not have to implement a common State interface for `PassiveHandler` and `ActiveHandler` since it is intrinsically

```
// Context
class Processor with <...>
do
  loop do
    var PassiveHandler ph with <...> end;
    par/or do
      await ph;
    <...>
    // State switch: Passive to Active
    var ActiveHandler ah with <...> end;
    par/or do
      await ah;
    <...>
    // State switch: Active to Passive
  end
end
// ConcreteState A
class PassiveHandler with <...>
do
  loop do
    <...>
    <...> = do FrameReceiver with this.fRx = &fRx; end;
    <...>
    else/if fRx.TOKEN then
      break;
    <...>
  end
end
// ConcreteState B
class ActiveHandler with <...>
do
  <...>
  loop do
    if <enough-time> and <more-to-send> then
      <perform-next-ems-request>
    else
      break;
    end
  end
  <...>
end
```

Listing 5.14: Adoption of the *State* pattern

given by language design. During runtime, we just need to exchange the organisms that contain the event handling code presented in Section 5.2.

Furthermore, the State pattern does not specify who – Context or ConcreteState – defines the state transitions. Essentially, a transition requires two decisions: First, the circumstances in which the current state is left. Second, the successor state. Gamma et al. [Gam+95] generally consider transitions in the ConcreteState subclasses more flexible and appropriate. However, they also note that this introduces implementation dependencies between those subclasses. In our adoption, we separate both decisions. `PassiveHandler` and `ActiveHandler` terminate themselves when their time is over. This is reasonable since they have the required knowledge. On the one hand, `PassiveHandler` knows the type of the current frame and that a token indicates the end of Passive Mode. On the other hand, `ActiveHandler` knows how much communication time is left and whether there are still more requests to send. `Processor`, in its turn, knows that `ActiveHandler` must succeed `PassiveHandler` and vice versa. As a consequence, our

ConcreteStates determine under which circumstances a state must be left while the Context determines the successor state. This separation of responsibilities avoids unnecessary dependencies across organisms and hence allows to keep the required knowledge local and encapsulated. `PassiveHandler` and `ActiveHandler` do not know `Processor` nor do they know each other. This allows to easily add new transitions without changing the existing ConcreteStates.

Finally, we believe that organisms in Céu better support the intuitive notion of "behavior". In [Gam+95], the user of the Context actually executes the behavior by calling the corresponding state function. Thus, behavior as such is only a one-step action. In contrast, organisms really "live" – so to speak – and execute their behavioral code themselves. Thus, they really have an independent existence over a longer period of time.

Benefit `PassiveHandler` and `ActiveHandler` partition the state-specific behavior and encapsulate it locally in dedicated organisms. This allows to easily add new states and transitions by implementing new organism classes. Since the transition logic does not reside in a monolithic **`if`** or `switch` statement but is structured into state-specific code that runs in a linear control flow, we consider it easier to comprehend. In particular, note how the loop in `Processor` supports the intuitive notion on how Passive and Active Mode alternate during runtime.

5.3.4 Adoption of Observer Pattern

> "Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically." [Gam+95]

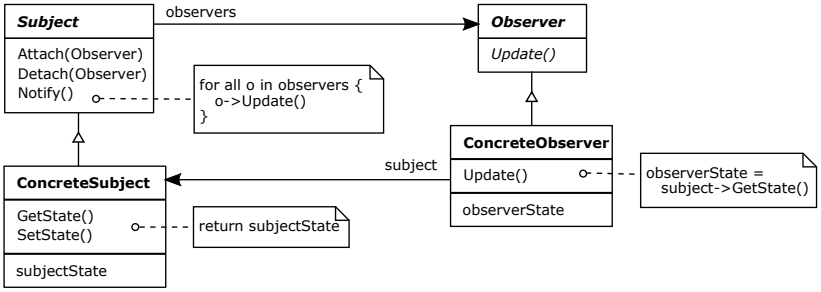

Figure 5.10: Structure of the *Observer* pattern [Gam+95]

Figure 5.10 illustrates the structure: "*Subject* knows its observers. Any number of Observer objects may observe a subject. [It] provides an interface for attaching and detaching Observer objects. *Observer* defines an updating interface for objects that should be notified of changes in a subject. *ConcreteSubject* stores state of interest to ConcreteObserver objects [and] sends a notification to its observers when its state changes. *ConcreteObserver* maintains a reference to a ConcreteSubject object. [It] stores

state that should stay consistent with the subject's [and] implements the Observer updating interface to keep its state consistent with the subject's." [Gam+95]

Motivation As presented in Section 5.3.1, all `EmsReadRequest` and `EmsWriteRequest` organisms in pool `requests` use the same injected instance of `Processor` for accessing the fieldbus, thereby using the interface described in Section 5.3.2. In particular, they send request frames to `Processor` for transmission. Once `Processor` changes its state due to the reception of a response frame, the request organisms require notification in order to synchronize their internal receive buffer with the one of `Processor`. That means that we have two aspects of abstraction – the dynamic requests and the static fieldbus interface – that are encapsulated in separate organisms. An arbitrary number of request organisms, which is not known at development time, depends on a single `Processor`. In order to achieve independency and design for reuse, we do not want these organisms tightly coupled. However, they should behave as tough they are. This suggests the adoption of the *Observer* pattern.

Adoption In Listing 5.15, we implement `Processor` (lines 2 to 8) which promotes the internal event `response` (line 3) in order to notify about the successful reception of a response frame. By taking advantage of CÉU's **`await`**-statement, `EmsReadRequest` (lines 10 to 16) and `EmsWriteRequest` (lines 18 to 24) observe the injected instance `p` of `Processor` for an incoming response (lines 13 and 21). On receipt, `p` triggers `response` (line 6), thereby waking up all observing organisms. Subsequently, `EmsReadRequest` and `EmsWriteRequest` process the response (lines 14 and 22), thereby reading out `p`'s receive buffer.

```
1  // ConcreteSubject
2  class Processor with
3    event void response; <...>
4  do
5    <...>
6    emit response; // notify
7    <...>
8  end
9  // ConcreteObserver A
10 class EmsReadRequest with <...>
11 do
12   <...>
13   await p.response; // observe p
14   <process-response>
15   <...>
16 end
17 // ConcreteObserver B
18 class EmsWriteRequest with <...>
19 do
20   <...>
21   await p.response; // observe p
22   <process-response>
23   <...>
24 end
```

Listing 5.15: Adoption of the *Observer* pattern

Comparison `Processor` corresponds to the ConcreteSubject while `EmsReadRequest` and `EmsWriteRequest` provide two implementations for a ConcreteObserver. In the pattern, subjects and observers have to implement a common Subject and Observer interface respectively. The former enables a group of objects – the subjects – to attach and detach Observer objects. The latter, in its turn, enables a group of objects – the observers – to get notified about changes in a Subject. Both interfaces are not required in CÉU. This is due to the fact that "observing", means awaiting a certain event triggered by another organism, is a first-class feature in CÉU and hence intrinsically supported by any organism. Thus, there is no need for interfacing and generalization.

In CÉU, we naturally have loose coupling between subject and observer. Due to CÉU's built-in internal event mechanism, observers do not have to register. Consequently, the subject does not know anything about its observers. It does not have to maintain a list of its observers and does not make any assumptions, for example it does not force a common interface. This allows any arbitrary organism to observe the subject. Furthermore, the subject is not forced to call subject-foreign code. In [Gam+95], the update code of an observer is actually executed by the subject by traversing through its list of registered observers. In CÉU, control is inverted. The subject does not talk to each observer explicitly. Instead, the linkage between observers and the subject is implicitly created by CÉU– observers really "listen". Moreover, there is no need for un-registration which is a major risk for memory leaks [DF10] in conventional observer implementations. The latter is avoided in CÉU by language design.

Benefit Our adoption allows to vary `Processor` and request organisms independently. Due to the loose coupling, the Subject `Processor` and the ConcreteObservers `EmsReadRequest` and `EmsWriteRequest` can belong to different abstraction layers – static and dynamic – in our fieldbus driver architecture (see Section 5.1.3). Although the lower-level `Processor` communicates with the higher-level request organisms, the fieldbus driver's layering is still intact. Note that a tight coupling would require the involved organisms to belong to both layers, thereby violating the layering, or to forcibly reside in one layer – static or dynamic – , thereby compromising the layering abstraction. Furthermore, it provides a natural way for implementing broadcasts since we do not need to specify the receiver of an internal event in CÉU. `response` is automatically sent to all request organisms that observe the injected instance `p` of `Processor`. Also, this allows to easily add new observing organisms, for example for logging or debugging purposes. Finally, received frames, that do not require further processing because `requests` is currently empty, are automatically ignored since no organism is observing `Processor`.

5.3.5 Adoption of Chain of Responsibility Pattern

> "Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it." [Gam+95]

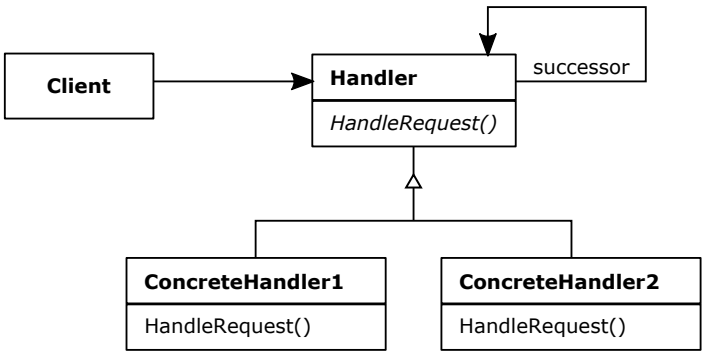

Figure 5.11: Structure of the *Chain of Responsibility* pattern [Gam+95]

Figure 5.11 illustrates the structure: "*Handler* [...] defines an interface for handling requests [and optionally] implements the successor link. *ConcreteHandler* [...] handles requests it is responsible for [and] can access its successor. If the ConcreteHandler can handle the request, it does so; otherwise it forwards the request to its successor. *Client* initiates the request to a ConcreteHandler object on the chain." [Gam+95]

Motivation As presented in Sections 5.3.2 and 5.3.3, `ActiveHandler` is used by `Processor` for explicitly realizing EMS requests on the fieldbus. If a response frame has been received, `ActiveHandler` triggers its corresponding processing. However, response frames are not handled by the caller of `ActiveHandler`, which is `Processor`, but by one of the organisms in pool `requests`. Which specific instance of `EmsReadRequest` or `EmsWriteRequest` is actually responsible for processing a certain frame it not known a priori at development time. Thus, the organism that ultimately handles the response frame is not known explicitly to the organism that initiates the processing request – each response frame has an implicit receiver. The set of organisms that can handle the frame is spawned dynamically during runtime (see Section 5.3.1). Also, organisms in `requests` can outlive several instances of `ActiveHandler`, for example if a certain `EmsReadRequest` requires the transmission of multiple request frames that do not fit in a single communication time slice. Thus, the concrete instance of `ActiveHandler` changes during runtime. Consequently, we aim to decouple sender `ActiveHandler` and receivers `EmsReadRequest` and `EmsWriteRequest` respectively by giving multiple organisms a chance to handle a response frame. This suggests the adoption of the *Chain of Responsibility* pattern.

Adoption In Listing 5.16, `ActiveHandler` (lines 2 to 8) triggers event `response` (line 6) through its interface (line 3) in order to request the processing of the currently received response frame. `Processor` (lines 10 to 26) creates an instance `ah` of `ActiveHandler` (line 15) and awaits its termination (line 17). While `ah` is running, `Processor` uses a loop (lines 19 to 22) in order to concurrently wait for every `response` triggered by `ah` (line 20), thereby adopting the Observer pattern from Section 5.3.4. On occurrence, `Processor` forwards the event by emitting `response` through its own interface (line 21). `EmsReadRequest` (lines 28 to 34), for example, waits for its injected instance `p` of `Processor` to trigger `response`

(line 31) and finally processes the received response frame (line 32). Note that the last step takes advantage of the Observer pattern too.

```
1  // Client
2  class ActiveHandler with
3    event void response; <...>
4  do
5    <...>
6    emit response;
7    <...>
8  end
9  // ConcreteHandler A
10 class Processor with
11   event void response; <...>
12 do
13   loop do
14     <...>
15     var ActiveHandler ah with <...> end;
16     par/or do
17       await ah;
18     with
19       loop do
20         await ah.response;
21         emit response;
22       end
23     <...>
24     end
25   end
26 end
27 // ConcreteHandler B
28 class EmsReadRequest with <...>
29 do
30   <...>
31   await p.response;
32   <process-response>
33   <...>
34 end
```

Listing 5.16: Adoption of the *Chain of Responsibility* pattern

Comparison `ActiveHandler` corresponds to the Client while `Processor` and `EmsReadRequest` implement two ConcreteHandlers. In Céu, the Chain of Responsibility is a straightforward approach which relies on two nested adoptions of the Observer pattern. `EmsReadRequest` observes `Processor` while `Processor`, in its turn, observes `ActiveHandler`. Thus, for the same reason as described in Section 5.3.4, a common Handler interface is not required. Also note that ConcreteHandlers does not know their successor. Using internal event `response`, `Processor` performs a delegation by broadcast. This allows, in principle, to implement a branched chain where multiple receivers for the same request are thinkable. For example, `Processor` might forward a response frame for processing to `EmsReadRequest` and a concurrent organism for logging purposes.

Benefit By taking advantage of Céu's internal events, we can easily implement a delegation chain across an arbitrary number of organisms. This allows to completely decouple the initial event sender from the final receiver. `ActiveHandler` does not

know anything about the existence of `EmsReadRequest` and `EmsWriteRequest`. The same is true vice versa. This allows to independently create new instances of `EmsReadRequest`, `EmsWriteRequest` and `ActiveHandler` during runtime – they are implicitly linked by the single, static instance `p` of `Processor`. Also, organisms in the chain do not have to know about the chain's structure. This generally increases flexibility in assigning responsibilities to organisms. Note that this approach can be effectively applied to exception handling too. `ActiveHandler`, for example, might encounter an exception while performing fieldbus interaction and throw a corresponding exception event to `Processor`. `Processor`, in its turn, might not be aware of the exception's consequences and hence delegate it up the chain to the currently active request organism. The latter, finally, might know that this specific kind of exception causes the request to completely fail – without any need for retry – and hence terminates immediately.

5.4 Testing Capabilities

> "Software is tested to establish its 'quality, performance or reliability'." [OK13, p. 443]

In contrast to static validation techniques such as code reviews or theorem proving, software testing is a dynamic approach where code is actually executed. Usually, this involves unit, integration and system tests. They all compile, link, build and run code chunks on specified test data and check the outcome against the expectations. Following a bottom-up strategy, basic functionalities are first unit-tested and then, for integration testing, composed into subsystems. The tested subsystems are finally brought together to perform system tests. [OK13, ch. 15] With respect to our gateway system, we assign these test classes as follows (see Figure 5.12):

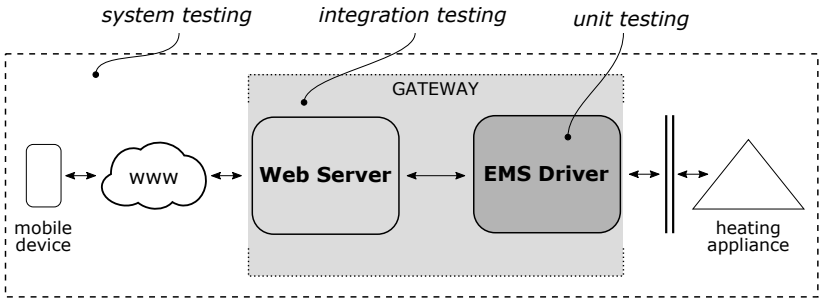

Figure 5.12: Test classes for the gateway application

Unit Testing deals with testing the software entities of the fieldbus driver. For example, we check whether our organism implementations correctly react to BYTE, BREAK and ERROR.

Integration Testing deals with testing the interaction between the two subsystems web server and fieldbus driver. For example, we check if a certain EMS request

submitted by the web server is acknowledged with the correct EMS response by the fieldbus driver within a specified period of time.

System Testing deals with testing the entire system chain including the mobile device, the Internet, the web server, the fieldbus driver, the fieldbus and the heating appliance. For example, if a user configures a certain room temperature using its smart phone we check whether the value set actually reaches the heating appliance.

During the course of development, testing should be started as soon as possible. The sooner a defect is detected, the lower are the costs to fix it. Thus, early unit testing is an important step towards reducing effort for rectification. [OK13, ch. 15]

In this chapter, we consider Céu's support for dynamic testing. Therefore, we first present its ability to simulate Céu code in the language itself. Then, we demonstrate how this language-feature can be used to implement and run unit tests.

5.4.1 Program Simulation

Based on a concrete, minimal example, Sant'Anna et al. [SIR12] fundamentally demonstrate Céu's ability to simulate programs in the language itself. The concept of simulation is centered around the notion of explicit asynchronous execution. Céu's `async`-block (asynchronous block) locally relaxes the rigorous synchronous model and allows execution under a different scheduling policy. In particular, asynchronous blocks interleave execution with the synchronous part of the Céu program as follows:

1. They start respectively resume whenever the synchronous side is idle.
2. They suspend after each loop iteration.
3. They suspend on every `emit`.
4. They execute automatically and run to completion unless (2) and (3) apply.

As a consequence, asynchronous blocks never execute with real parallelism with the synchronous side, thereby preserving determinism in the program.

> "Asynchronous blocks are allowed to emit input events and also events that represent the passage of wall-clock time towards the synchronous side of the program. This way, it is easy to simulate and test the execution of programs with total control and accuracy with regard to the order of input events – all is done with the same language and inside the programs themselves." [SIR12, p. 14]

Based on this, program simulation in Céu generally relies on the simple pattern presented in Listing 5.17. Simulation requires two concurrent trails. In the first trail (line 2), we run the Céu code which should be simulated. In the second trail (lines 4 to 7), we simulate the environment given by the sequence of input events. In particular, `async` (lines 4 to 6) aids as the source for the external input stimuli that drive the code in the first trail.

On simulation start, the code trail executes first. This runs the code under simulation

```
1 par do // Trail 1: Code to simulate
2   <...> // run the code under simulation
3 with   // Trail 2: Environment
4   async do
5     <event-sequence> // emit the external input stimuli
6   end
7   escape 0;
8 end
```

Listing 5.17: Fundamental program simulation in CÉU

which, right after start-up, suspends waiting for an input event. Thus, the environment trail executes subsequently. This causes **async** to emit the first input event to the code and immediately suspend afterwards. Control returns to the code trail which performs the corresponding event reaction and finally suspends again. The environment trail continues triggering the next input event and so on. This alternating execution proceeds until the entire event sequence in **async** has been processed, thereby indicating the end of simulation. Note that the **par**-block in line 1 deliberately abstains any abortion mechanism. This is to ensure that the entire input event sequence is completely performed even if the code under simulation terminates previously. In case it never terminates, for example due to an infinite loop, **escape** (line 7) finally forces the simulation to end at the latest. By this, it is possible to simulate terminating and non-terminating CÉU code likewise.

5.4.2 Unit Testing

As presented in Sections 5.2 and 5.3, the software design in CÉU is centered around organisms for hierarchical structuring and modularization. For this reason, it seems reasonable to consider an organism as a Unit Under Test (UUT) when testing reactive CÉU code.

Testability of Organisms Our deployment of organisms in Sections 5.2 and 5.3 shows that they may have different relationships among each other. In particular, we generally distinguish between four different classes of organisms to test (see Figure 5.13):

First, if an organism does not rely on any other organism in its interface or execution body it is completely independent (Class 1). Second, dependent organisms may instantiate required organisms in their execution body (Class 2) or take advantage of existing instances. Third, the latter may be injected via the organism's interface (Class 3) or referenced from CÉU's top-level global interface (Class 4). The global interface **interface** `Global` **with** `<...>` **end** allows to explicitly declare a set of variables and organisms with global scope and application lifetime. Figure 5.13 shows how our fieldbus driver organisms map to those classes. It appears, that its class has a considerable impact on an organism's testability.

Organisms of classes 1 and 2 are generally easy to test since they are self-contained. Either they are completely isolated (Class 1) or they self-sufficiently manage instantiation, initialization and interfacing of dependent organisms (Class 2). Accordingly,

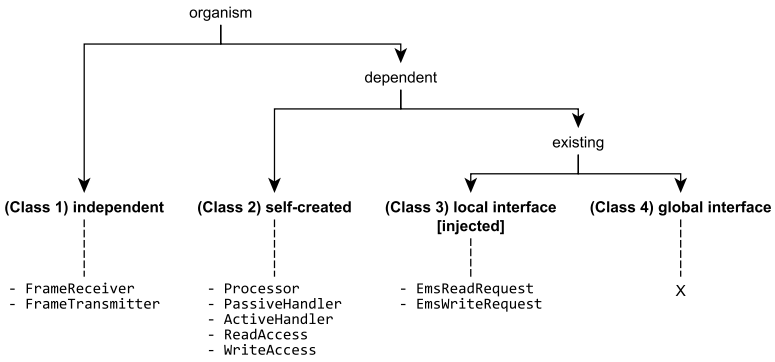

Figure 5.13: Classification of organisms

testing organisms of Class 2 requires testing the dependent organisms before, thereby deploying a bottom-up strategy. In contrast, organisms of classes 3 and 4 comprise external dependencies and hence decrease testability. In particular, Class 3 organisms additionally require to create, initialize and inject the instances of the dependent organisms for each test case. Note that this considerably increases test effort since the behavior of the organism under test depends on the initial state of the injected organisms. At least, due to the dependency injection, it becomes obvious on interface-level which organisms are actually involved. In contrast, implementing organisms of Class 4 is like using global variables in order to affect the internal behavior of a function in C. Only reviewing the organism's execution body reveals which organism instances are actually involved and must be provided. Due to a line up of software engineering disadvantages [SK13], i.a. high testing complexity, this approach is generally discouraged. For this reason, we suggest to favor organisms of classes 1 and 2, use Class 3 organisms as rarely as possible and entirely avoid organisms of Class 4 in order to gain good testability.

Unit Test Case Implementation It appears, that implementing unit test cases is a straightforward approach in Céu. Therefore, we take advantage of its built-in program simulation capabilities (see Section 5.4.1). In Listing 5.18, we exemplarily demonstrate how a simple *black-box test* case for our `FrameTransmitter` (see Section 5.2.1, Listing 5.6) – the UUT – may look like in Céu. Our aim is to check whether `FrameTransmitter` correctly behaves in case of timing errors during frame transmission. For this, we provide a predefined frame to send and simulate the incoming mirror bytes as well as the passage of physical time in between them, thereby deliberately enforcing a mirror timeout. Afterwards, we check the return value of `FrameTransmitter` which is expected to indicate the timing error. In general, each test case requires the following steps:

1. Include the UUT code and, if necessary, dependent code.
2. Set up everything required to run the UUT.
3. Instantiate, initialize and start the UUT.
4. Perform the simulation.

5. Check the assertions.

```
1  /* @file tests.ceu */
2  // Step 1: Include the UUT code and, if necessary, dependent code
3  #include "byteProcessing.ceu"
4  // Step 2: Set up everything required to run the UUT
5  var int ret = 1;
6  var Frame fTx = Frame.MESSAGE_READ(<...>); // e.g. 0x01 0x02 0x03
7  // Step 3: Instantiate, initialize and start the UUT
8  var FrameTransmitter tx with
9    this.fTx = &fTx;
10 end;
11 // Step 4: Perform the simulation
12 par do
13   ret = await tx;
14 with
15   async do
16     emit 25ms;
17     emit BYTE => 0x01;
18     emit 18ms;
19     emit BYTE => 0x02;
20     emit 50ms; // timing error: mirror time exceeded!
21     emit BYTE => 0x03;
22     emit 32ms;
23     emit BREAK;
24   end
25 // Step 5: Check the asserts
26   _assert(ret == -2);
27   escape 0;
28 end
```

Listing 5.18: An exemplary black-box test case for `FrameTransmitter`.

According to this, we implement our test case in a separated source file *tests.ceu* and first include all the required byte layer code (line 3). Second, we set up `ret` (line 5) to store the return value and `fTx` (line 6) to hold the frame that should be transmitted. Note that in case of Class 3 and 4 organisms at this point the dependent organisms must be set up too. Third, we create and start `tx` (lines 8 to 10), thereby passing the predefined frame (line 9). Fourth, we simulate the transmission (lines 12 to 28) based on the specified input event sequence (lines 16 to 23). In the code trail we just await the termination of `tx` in order to retrieve its return value (line 13). In the environment trail, we emit the input stimuli. When we run the test case, `tx` transmits the first byte of `fTx` and awaits the corresponding mirror. **`async`** takes control and emits 25 ms (line 16). `tx` processes the passage of time which has no effect since it is less than the mirror timeout of 42 ms. Then, the first mirror byte is emitted (line 17) which causes `tx` to transmit the next frame byte. Again, we simulate the passage of time and the corresponding mirror (lines 18 and 19). In line 20, **`async`** emits 50 ms which causes a mirror timeout to occur. Consequently, `tx` immediately terminates, thereby providing a proper return value. The remaining events are emitted too but are never processed since `tx` has already run to completion. Fifth, after termination of **`async`**, we use assertions to automatically check whether `tx` worked correctly or not (line 26). A mirror timeout should be indicated by a value of `-2`. Therefore, we expect that `ret==-2` holds (line 26). If not, the test case fails and a corresponding warning is thrown.

5.4.3 Discussion

In reactive embedded systems, code execution is entirely linked to the occurrence of events. Both, sporadic and recurring events are eventually triggered by dedicated hardware. Whether it is the serial communication device for interfacing the fieldbus or a hardware timer that measures the passage of physical time and cyclically activates OSEK tasks. Thus, executing a certain reactive functionality for testing purpose usually requires to either deploy and run the code on the actual target platform or execute it in a hard- or software emulator. Note that in those Hardware-In-the-Loop (HIL) respectively Software-In-the-Loop (SIL) tests the system environment must be simulated too. For this reason, dynamic testing of reactive embedded code is generally a complex and tedious task.

> "Traditionally, unit testing involves the development of a 'harness' to provide an environment where the subset of code under test can be exposed to the desired parameters in order for the tester to ensure that it behaves as specified." [OK13, p. 449]

In Céu, it is possible to escape from that rigorous fixture dependency due to the fact that events are a language-feature and can be triggered in the language itself. In particular, testing in Céu is completely independent from any external hard- or software tool. It does not require any dedicated testing platform nor any additional knowledge about how to configure and run test cases. This makes it possible to easily test code on the feature-rich development system with full control over the sequence of input events before deploying it on the constrained target platform [SIR12]. In the early development phase, this also allows uncomplicated, rapid testing of different solutions.

Due to Céu's synchronous-reactive model of execution, program behavior solely depends on the order of input events. Their exact timings are irrelevant to the application outcome [SIR12, p. 14]. Since **async** allows to deterministically specify the order of environment events, several runs of the same test case always produce the same output. Note that this guaranteed reproducibility particularly expands to temporal behavior as well as concurrency. Remember that in the asynchronous model concurrent behavior is inherently non-deterministic and hence not reproducible. This makes testing and debugging a challenge since related failures only occasionally appear in the field [Lee05; Lu+08].

Listing 5.18 makes apparent that test cases can be easily separated into dedicated source files, thereby leaving the original production code untouched. Sant'Anna et al. [SIR12] indicate that this allows to develop a test framework which retrieves the UUT code as well as the related test event sequences from separated files and automatically integrates and runs multiple test cases according to above approach. In combination with reproducibility, this makes implementing regression testing a straightforward task in Céu.

Furthermore, Céu's simulation approach enables *black-box* and *white-box* testing likewise. For black-box testing, also known as functional testing, we only consider an organism's public interface (including its return value) and test it against the

specification. Listing 5.18 is an example for this. For white-box testing, also known as coverage testing, we additionally take an organism's execution body into account. Taking the example of `FrameTransmitter`, we can derive particular sequences of input events, that cover all possible classes of communication scenarios such as successful transmission, unexpected end-of-frame character, malformed mirror, mirror timeout and so on. By this, we can ensure that each trail in `FrameTransmitter` is executed once at least. Note that the latter requires to deliberately insert errors into the test cases in order to check the correctness of error handling. In Céu, error injection appears to be a trivial task. In Listing 5.18, for instance, we easily simulate a timing error by emitting a time interval greater than the mirror timeout interval (line 20).

5.5 Important Points to Consider

During our work, we encounter a number of important aspects to consider when deploying the synchronous paradigm as provided by Céu. Some of them might influence fundamental design decisions in the early development phase.

1. *No physical concurrency*: In Céu, **`par`**-blocks allow to compose an application as a set of concurrent, cooperating trails. Remember that every Céu program compiles to single-threaded, sequential code. Thus, concurrency in Céu is only logical. This means that synchronous code cannot benefit from performance provided by multi-core or distributed hardware architectures. However, the majority of embedded systems still relies on a single processor [EE15] which lacks physical concurrency anyway. Consequently, for most of today's embedded projects Céu is not expected to induce any drawback with respect to performance compared to existing solutions. Quite the contrary, the deployment of Céu makes concurrency on single-core architectures deterministic and reproducible.

2. *Synchronization is still required*: Even if the entire business logic is reactive and relies on synchronous code, reasoning about synchronization is still required to some extend. This is due to the fact that in an embedded system event sources – hard- or software – are inherently asynchronous. Thus, events may be asynchronously triggered by any concurrent context of execution, for example a task/thread or an interrupt service routine. In order to make event processing synchronous and deterministic, Céu demands serialization of the incoming events by entering the single input event queue which is shared between those asynchronous units of execution. As a consequence, the platform interface code and the operating system, if any, have to make access to the queue thread-safe. However, note that in Céu's synchronous approach, dealing with mutual exclusion and deadlocks is limited to the event-based interface between the asynchronous environment and the synchronous code. By this, asynchronous concurrency issues are extracted from the large and complex business logic (the Céu program) and concentrated in a comparatively small and easy to manage part of the entire application (that

part of the platform interface code that deals with enqueuing). This significantly reduces synchronization efforts and makes the code less error-prone.

3. *Concurrent behavior depends on the lexical order of trails*: In Céu, concurrent trails execute in the order they appear in the code and are allowed to perform side effects on shared variables (see Section 2.3). As a consequence, reordering trails may change the program behavior. Consider the following example code:

```
1 var u8 x = 5;
2 par/or // Trail A (read access on x)
3   var u8 y = x;
4 with   // Trail B (write access on x)
5   x = 2;
6 end
```

Trail `A` executes before `B` in the same reaction. Thus, `y` is always set to `5` while `x` is updated to `2`. If the order of lines 3 and 5 is inverted, `y` is always set to `2` instead. Thus, the outcome depends on the lexical order of `A` and `B`. Due to this fact, Céu is *not truly synchronous* according to the synchronous hypothesis (see Section 2.2). Note that the FIFO-like execution policy applies to organisms too. Static organisms and dynamic organisms of the same pool execute in their order of instantiation. Dynamic organisms of different pools execute in the order in which the respective pools are declared in the code. In general, this makes Céu code less robust towards changes and requires careful consideration whenever organisms are deployed and composed. For example, if we move a pool declaration inside the code we have to consider its location relative to other pool declarations since this might change the order in which the entire set of dynamic organisms is scheduled. If the application's business logic relies on that order, it might not work as expected anymore.

In contrast, Esterel, for instance, is considered truly synchronous [Ben+03]. It distinguishes between variables and signals. Variables are local to trails and hence cannot be used in a read-write fashion among them. This prevents shared memory concurrency. Signals are provided for inter-trail communication and have global scope. Thus, they may be set or unset by several concurrent trails in the same reaction. However, Esterel ensures that in each reaction, a signal can be either absent or present but not both. Therefore, the Esterel compiler performs a causality analysis which detects whether signal operations of concurrent trails may interfere at run time. If this is the case, it tries to rearrange the concurrent statements so that the signal's value is well-established before any read access is performed. If this is not possible, the code is rejected otherwise [Ber00]. Consequently, the program behavior is independent of the order in which trails are scheduled for signal processing. This makes the code not only deterministic but also robust towards changes – a significant benefit of Esterel.

However, while Céu's implementation seems kind of arbitrary, it is a very simple and pragmatic approach which statically assigns priorities to trails, allows thread-safe shared memory access and ensures deterministic, reproducible execution for

concurrent code too. Due to its consistent establishment throughout the entire language design, it provides a transparent semantic that matches the common, sequential execution style of software. Thus, we believe that this is a reasonable approach for software development.

4. *Dealing with blocking and computation-intensive tasks*: Blocking and long-lasting tasks do not belong to the reactive domain. Accordingly, the synchronous hypothesis generally prohibits their deployment in Céu. Instead, they must be delegated to the asynchronous environment. For dealing with computation-intensive tasks, however, Céu provides two workarounds which allow to stay in the language:

   First, the low-level approach distributes long-lasting computations across multiple reactions using the **`async`**-block which simulates asynchronous execution (see Section 5.4). The core idea is to let long-running loops only execute a single iteration in each reaction, thereby keeping a single reaction short. However, this scheduling is too simple for real-life applications exposed to real-time requirements. In particular, remember that code in an **`async`**-block only runs if the synchronous side is idle. Thus, it is not guaranteed to run at all. For this reason, we believe that this approach is more suitable for testing purpose as presented in Section 5.4.

   Second, Céu provides an **`async/thread`**-block which concurrently executes the enclosed lines of code in a real, dedicated thread scheduled by the environment. The block rejoins as soon as the corresponding thread runs to completion. By this, the computation-intensive task is performed independently in the asynchronous environment and only interferes on an event level with the synchronous code. This approach seems better for real-life applications but shifts real-time problems to the scheduler of the environment.

5. *Limited real-time capability*: Céu is not designed for meeting hard real-time deadlines [San+13]. Due to its purely event-driven execution scheme (see Section 2.3) – each event triggers a reaction – it is generally impossible to predict the frequency of incoming events and the system may not be able to keep up with them. While Céu's input queue (see Section 5.1.2) prevents the loss of events, it is still possible that event occurrences accumulate so that reactions to (time-critical) events may be delayed and hence cannot complete in time.

   Purely sample-driven synchronous languages, such as Esterel, make it easier to reason about real-time behavior. The minimum inter-arrival time between any two consecutive events is a fixed system design parameter given by the sample rate of the clock that cyclically triggers the reactions. Furthermore, remind that, in the sample-driven approach, multiple events (changes in the environment) can be processed in a single reaction (see Section 2.2). This prevents the program to be flooded by event occurrences. However, events can still get lost if there is more than one occurrence of the same event between two clock ticks.

   Finally, Céu's synchronous language design generally establishes a solid basis for real-time applications in which the required memory and execution time

for each reaction is bounded on language level. Stringent timing-constraints are principally achievable but they require additional knowledge and guarantees from the program's soft- and hardware environment which are out of scope of Céu. For example, if the minimum inter-arrival of events is known at development time, a WCET analysis can be applied in order to meet hard real-time requirements.

6. *No debugging support for* Céu *code*: To our knowledge, there is no tool available that supports state-of-the-art debugging Céu code, for example in terms of setting break points, inspecting memory locations or providing stack traces. This appears to be a clear drawback compared to full-featured debugging support for C which demands, for example, to fall back to extensive usage of the `printf` output. The generated state machine in C comprises `#line` annotations that enable traceability of the original Céu code. That is, we can cross-check which line of C code has been generated in response to which line of Céu code. However, this approach seems to be very low-level and tedious since it resembles debugging of C code by inspecting the generated assembler code.

6 Quantitative Evaluation of Synchronous Programming

In this chapter, we substantiate our investigations from Chapters 4 and 5 in quantitative terms. First, we provide a static code analysis that extracts different quantitative performance indicators from both, the asynchronous and the synchronous, implementations. This measurement allows an objective comparison of selected software quality aspects. Second, we perform a user study that subjectively evaluates both approaches by comparatively quantifying different software quality attributes.

6.1 Code Analysis

In our static code analysis we measure the separation of concerns, the scattering of interfaces and the code size. We focus on the byte and frame layers since they contain the actual control functionality and hence where fully reimplemented.

6.1.1 Separation of Concerns

With respect to Sections 4.2.3 and 5.2.1, we measure the separation of concerns and the scattering of code for the implementation of $\texttt{Receive}_{\texttt{BL}}$ and $\texttt{Transmit}_{\texttt{BL}}$. Therefore, we assign each involved line of code to its related communication concern, thereby distinguishing between sole $\texttt{Receive}_{\texttt{BL}}$, sole $\texttt{Transmit}_{\texttt{BL}}$, both and other. "Other" contributes functionality to the frame layer. The results are presented in Table 6.1. While reading the table from left to right reveals the separation of concerns, reading from top to bottom indicates the code scattering.

In the existing C implementation, $\texttt{Receive}_{\texttt{BL}}$ and $\texttt{Transmit}_{\texttt{BL}}$ manifest in 41 + 77 + 48 = 166 Lines of Code (LOC) in total. While the 89 LOC of $\texttt{Receive}_{\texttt{BL}}$ are scattered across one interrupt service routine and two functions, the 125 LOC of $\texttt{Transmit}_{\texttt{BL}}$ reside in two interrupt service routines and three auxiliary functions. However, the majority of both functionalities, 61 percent (102 LOC), is merged in `RxChar` whereby $\texttt{Receive}_{\texttt{BL}}$ and $\texttt{Transmit}_{\texttt{BL}}$ have nearly the same proportion, 49 and 51 percent respectively. This should make apparent that this approach is clearly centered around the `RxChar` monolith which significantly contributes to both functionalities likewise. This becomes even more apparent if we consider their lexical arrangement within `RxChar` in Figure 6.1. Both functionalities are intertwined and hence cannot be usefully extracted and separated.

In contrast, in the CÉU reimplementation, $\texttt{Receive}_{\texttt{BL}}$ and $\texttt{Transmit}_{\texttt{BL}}$ manifest in 72 LOC in total. CÉU's organisms eliminate any code scattering. While the 27 LOC

Language	Function	Communication Concern				Σ
		$\texttt{Receive}_{\texttt{BL}}$	$\texttt{Transmit}_{\texttt{BL}}$	both	other	
C	`RxChar`	34	33	35	108	210
	`TxChar`		15			15
	`sendByte`		7			7
	`sendBreak`		6			6
	`blReset`			13		13
	`readFrame`	7				7
	`writeFrame`		16			16
	Σ	41	77	48	108	274
CÉU	`FrameReceiver`	27				27
	`FrameTransmitter`		45			45
	Σ	27	45			72

Table 6.1: Quantitative distribution of communication concerns (in lines of code)

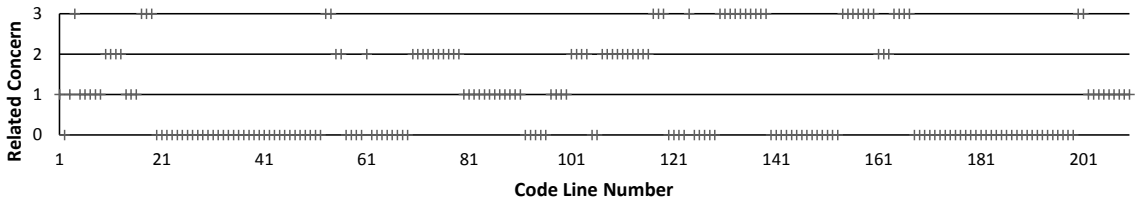

Figure 6.1: Mapping between routine `RxChar` code line and communication concern in the C implementation: $\texttt{Transmit}_{\texttt{BL}}$ (3), $\texttt{Receive}_{\texttt{BL}}$ (2), both (1), other (0).

of $\texttt{Receive}_{\texttt{BL}}$ are entirely encapsulated in `FrameReceiver`, the 45 LOC of $\texttt{Transmit}_{\texttt{BL}}$ are completely abstracted by `FrameTransmitter`. By this, both communication concerns are perfectly separated and locally contained in a single abstraction entity. Both organisms are completely isolated and do not share any code line or resource.

Finally, it is worth to mention that in the C implementation `RxChar` significantly contributes to frame layer concerns (denoted as "other"). This is caused by the fact that only the byte layer knows if a frame transmission has been successful or not and the exact point in time where the transmission actually completed. Both pieces of information are required by the frame layer in order to account for possible retransmissions and the measurement of the response timeout interval t_r. In order to reduce the burden of passing this information up to the frame layer, in the C approach the according retransmission and timer management code is directly interlaced into `RxChar` although it conceptually belongs to the frame layer. As Table 6.1 reveals, in CÉU this is not necessary anymore. Due to the synchronous model of execution the termination of `FrameTransmitter` intrinsically comprises the exact point in time of transmission completion. In addition,

its return value indicates success or failure. Thus, the byte layer implementation in Céu is free of foreign concerns.

6.1.2 Interface Scattering

With respect to Sections 4.2.3 and 5.2.2, we measure the scattering of the byte and frame layer interfaces. Therefore, we count the points of interaction between the layers. That is, all the different code locations in the byte layer that access the frame layer interface and vice versa. By "access" we mean a call to one of the interface functions or a read or write operation on shared variables. The less points of interaction the lower the scattering. Low scattering promotes layer independency and makes interaction easier to comprehend. The results are presented in Table 6.2.

in C		in Céu	
accesses frame to byte layer			
`blReset`	9	`FrameReceiver`	3
`readFrame`	1	`FrameTransmitter`	3
`writeFrame` / `startTransmission`	17		
Σ	27		6
accesses byte to frame layer			
`flReset`	6		
`notify`	3		
Σ	9		0

Table 6.2: Number of interaction points between byte and frame layer

In the existing C implementation, the frame layer interfaces the byte layer by calling the functions `blReset`, `startTransmission`, `readFrame` and `writeFrame`. Since calls to `writeFrame` and `startTransmission` always appear pairwise we treat them as a single call, thereby favoring the implementation in C to some extend. Table 6.2 reveals that this leads to 27 points of interaction. The byte layer, in its turn, interfaces the frame layer by calling `flReset` and performing write operations on flag variable `notify`. This leads to 9 points of interaction which is one-third of the frame layer. This imbalance seems reasonable since the frame layer is conceptually superior to the byte layer and uses its functionalities. In total, byte and frame layer interaction is scattered across 36 code locations.

In contrast, in the Céu reimplementation, the frame layer interfaces the byte layer solely through calls to `FrameReceiver` and `FrameTransmitter` respectively. Note that, due to their function-like usage, the byte layer functionalities never need to explicitly call frame layer code. Thus, there are no points of interaction in the byte layer code. Reconsidering our architectural design in Section 5.1.3 it seems reasonable that there are only a few points of interaction in the frame layer (see Table 6.2). `FrameReceiver` is called once in `PassiveHandler`, `ReadAccess` and `WriteAccess` which makes 3 accesses in total. The same applies to `FrameTransmitter`. Thus, there are 6 points of interaction in

total which is only one-sixth compared to the existing C approach. This should make apparent that in the Céu implementation interactions between byte and frame layer are considerably easier to localize and comprehend.

6.1.3 Code Size

We compare the code size of the byte and frame layer implementations. For this, we use our reimplementation in Céu– not counting the platform interface code – as a baseline and extract, in a best effort, all the code from the existing C implementation that covers the same functionality. Since concerns in C are not that clearly separated as in Céu, this is actually a challenging task and does not allow a 100 percent accuracy. Nonetheless, we believe that our results at least provide an impression about how programming effort compares in C and Céu. The results are presented in Table 6.3.

	C	Céu	Céu vs. C
byte layer	166	72	-56.63%
frame layer	660	302	-54.24%
Σ	826	374	-54.72%

Table 6.3: Code size of byte and frame layer implementation (in LOC)

In the existing C implementation, the byte layer requires 166 LOC while the frame layer manifests in 660 LOC. This leads to 826 LOC in total. In contrast, in the Céu reimplementation, the byte and frame layer only cover 72 LOC and 302 LOC respectively. This comprises a code reduction of about 56 respectively 54 percent. In total, 374 lines of Céu code compare to 826 lines of C code which is an overall reduction of more than 50 percent.

Céu eliminates any effort for manual stack, state and timer management as well as for manual synchronization. For this reason, it seems reasonable that the reimplementation is less verbose. In order to make this plausible, we examine the implementation effort in C that is solely required for manual state and timer management in the byte layer. Therefore, we count each code line that deals with manipulating and selecting the current state or with interfacing timers. The results are presented in Table 6.4. It reveals, that 55 LOC deal with state management and 15 LOC interface the time domain. Thus, in total, 70 LOC of the entire byte layer implementation (166 LOC) comprise effort for manual state and timer management. Since both is taken care in Céu, this alone makes a code reduction of 42 percent. Note that in this investigation we have not yet considered effort for control and handling of events, stack management and so on.

Finally, it appears, that Céu's language-level support effectively allows developers to focus on the business logic, thereby producing concise and readable code. However, reducing implementation effort can be generally achieved by any code generation tool such as Matlab/Simulink or SCADE for example. We believe that the primary benefit of Céu does not rely in reducing code verbosity but in re-enabling fundamental software engineering principles.

Function	Management state	Management timer	Σ
`RxChar`	39	9	48
`TxChar`	11	4	15
`sendByte`	1		1
`sendBreak`	1		1
`blReset`	3	2	5
`readFrame`			0
`writeFrame`			0
Σ	55	15	70

Table 6.4: Quantitative distribution of manual state and timer management in the byte layer in C (in LOC)

6.2 User Study

In our user study we intended to comparatively measure the software quality of the synchronous-reactive approach versus the asynchronous-sequential implementation. The aim was to validate our considerations in Chapter 5 by the experiences of other software developers.

6.2.1 Design

Hypothesis Reactive software that relies on the synchronous-reactive paradigm (as deployed in Chapter 5) is of higher quality – this means easier to program, comprehend and maintain – than a corresponding implementation based on conventional, sequential programming and asynchronous execution (as exemplified in Chapter 4).

Population The user study targeted 20 undergraduate students of "Computer Science for Engineers" in the fourth semester. We consider them to be the next generation embedded software developers which might be potential users for synchronous languages. During their studies, they have already gained extended theoretical and practical experience in the programming language C via various courses such as System Programming Concepts, Microprocessor Engineering, Operating Systems and an embedded-oriented Software Engineering Project. However, they do not know anything about Céu nor synchronous programming in general.

Metrics For the quantitative evaluation of software quality we considered four quality attributes. (1) *Comprehensibility* measures the mental effort to understand a given piece of code. (2) *Changeability* measures the effort for modifying existing source code, for example due to changes in system specification or for maintenance purpose. (3) *Time need* measures the time effort required for implementing a certain piece of functionality. (4) *Overall impression* measures the tendency to favor the corresponding

programming language. Our measurement relied on a survey. Each student had to fill out the comparative questionnaire below:

	C			Céu	
Comprehensibility	2	1	0	1	2
Changeability	2	1	0	1	2
Time Need	2	1	0	1	2
Overall Impression	2	1	0	1	2

The students had to compare C and Céu with respect to each quality attribute. They could chose between considerably better (2), tends to be better (1) and no difference (0).

6.2.2 Procedure

The study was performed at the University of Applied Sciences in Gießen (Germany) in summer semester 2016 and winter semester 2016/17. The students attended the lecture Introduction to Embedded Systems. First, in order to allow a fair comparison, we used two consecutive lectures (180 minutes in total) to teach Céu's fundamental language concepts. This included the synchronous model of execution, external and internal events and the synchronous control statements. We deliberately abstained from any high-level abstraction concepts such as organisms. We believed that the available time was insufficient for teaching them in required detail. Thus, the study focused on the qualities of a non-torn, linear control flow and the specification of temporal behavior. Second, the students got a homework assignment which was composed of three exercises exemplified below. They required to read and write C and Céu code with comparable complexity. Each exercise addressed a subset of the investigated software quality attributes (see Table 6.5). Third, after completion, the students filled out the questionnaire. Fourth, the students submitted their work as well as the questionnaires and we evaluated the results.

	1a	1b	2	3a	3b
Comprehensibility	x	x	x	x	
Changeability				x	
Time Need				x	x
Overall Impression	x	x	x	x	x

Table 6.5: Mapping of investigated software quality aspect to exercise

Exercises We could not use the original fieldbus driver code base for comparison. However, our aim was to compare general concepts rather than their concrete manifestation. Therefore, the exercises were based on a fictive, simple, embedded system which mimicked the characteristics elaborated in Chapter 3. The fictive system was connected to a computer mouse and keyboard for user interaction. Its behavior fully

relied on continuous event- and time-based user interaction. Mouse and keyboard provided the following input events: (1) `LEFT` and (2) `RIGHT` mouse click as well as button (3) `DOWN` and (4) `UP`. A payload was associated to button events indicating the American Standard Code for Information Interchange (ASCII) of the pressed and released button respectively. In addition, the system provided a red and a green Light-Emitting Diode (LED) which could be switched on and off depending on user input. Above events mimicked fieldbus events while the ASCII payload corresponded to the byte value that was received in case of `BYTE`. The exercises increased in their difficulty level. To force the students to examine both code chunks, their switching behavior varied in nuances resulting in different results. The exercises as well as the questionnaire can be referred to in Appendix A.

Exercise 1: Comprehension of Event-Controlled Flow Exercise 1 addressed the comprehension of program control flow across reactions. Switching behavior was solely determined by events. Reactive code was given in C and Céu. Code was comparable in complexity. However, its switching logic varied in nuances.

In 1a), code executed the functions `ledOn` and `ledOff` in order to change the state of the green and red LED. Given an event sequence, the students had to determine the state of the red and green LED – on or off – after the last reaction. The students had to comprehend in which order `ledOn` and `ledOff` were executed across events. Their execution depended on the current software state and the event.

In 1b), the value of integer variable `ret` was modified. Given an event sequence with associated ASCII payload, the students had to determine the value of `ret` after the last reaction. The students had to track accesses to `ret` across several events. Access depended on the current software state, the event, the payload and the payload that had been processed in the past. Considering a (global) variable's state across multiple events is a common problem for developers. Another challenge was given by button handling. Events `DOWN` and `UP` of a certain button were logically linked but did not have to be consecutive. Thus, some kind of bookkeeping was required in order to remember the current button state.

Exercise 2: Comprehension of Time-Controlled Flow Exercise 2 extended reactivity to the time domain. Switching behavior was determined by events and the passage of physical time. Reactive code was given in C and Céu. Code was comparable in complexity. However, its switching logic varied in nuances. Code executed the function `printf` in order to output a sequence of letters. Given an event and time sequence, the students had to determine the letter sequence which was generated as output. The students had to comprehend how the passage of time changed the systems internal state. This is especially difficult if nested timing behavior must be taken into account such as discussed in Section 4.4. Also, temporal behavior in the C implementation was not deterministic. If both timers elapsed at the same time, for example after 10 milliseconds, C code did not define the order in which interrupt service routines executed. The actual order depended on the hardware configuration and was out of

scope of the programming language. This led to different possible output sequences for the C implementation. Due to its synchronous execution semantics, Céu code, in contrast, did not comprise this problem. Students should have encountered this difference.

Exercise 3: Implementation of Synchronous Code Exercise 3 required to modify and write synchronous code.

In 3a), a prose text specification on how mouse clicks should change the value of integer variable `ret` over time was given. A corresponding implementation was given in C and Céu too. In addition, a second specification defined a set of changes that had to be applied in comparison to the former. The students had to modify the given implementations according to the change set. They had to establish the cognitive link between the switching scenario defined in prose text and its manifestation in the code. In order to identify the correct code locations that required modification, they had to comprehend which set of software-internal states was actually affected and how to correctly change their transitions. While the C implementation required to globally extend the state machine and apply changes to several service routines, Céu code only required an additional trail in its parallel block. Students should have encountered this difference.

In 3b), a prose text specification defined how mouse clicks should increment a counter depending on button presses and releases. The result was outputted using the `printf` function. The students had to implement the given specification in C and Céu from scratch. They had to understand how to properly apply the synchronous language concepts to program an application.

6.2.3 Analysis and Conclusion

In total, 20 questionnaires had been filled out. For our analysis, we first counted the votes and calculated their relative frequencies. In order to allow the calculation of an average vote for each quality attribute, we mapped the votes cast to numerical values as follows:

Vote	Value
Considerably better in C	-2
Tends to be better in C	-1
No difference	0
Tends to be better in Céu	+1
Considerably better in Céu	+2

Subsequently, we determined the average vote based on their relative frequencies. The more positive the average value, the more did the student favor Céu with respect to the corresponding software quality attribute. Negative values, in their turn, indicated a favor for C. The results are presented in Figure 6.2 and Table 6.6.

It appears, that the results are entirely positive for Céu. The average vote reveals that, in all software quality attributes, Céu outperforms C. In particular, this is

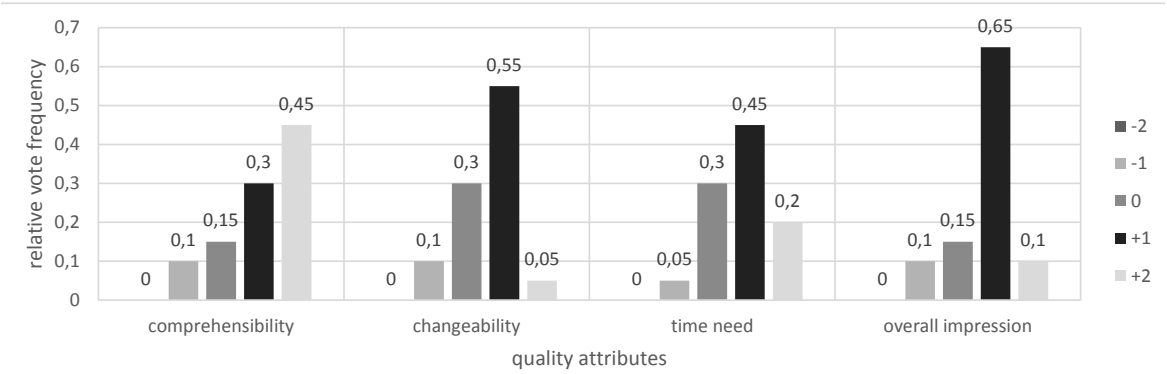

Figure 6.2: Survey results: relative vote frequency
(based on 20 questionnaires)

Quality Attribute	Average Vote
Comprehensibility	+1.10
Changeability	+0.55
Time Need	+0.80
Overall Impression	+0.75

Table 6.6: Survey results: average vote
(based on 20 questionnaires)

true for comprehensibility. The performed exercises tackle only very small and simple applications. Real-life automata, such as for fieldbus communication, are far more complex and difficult to understand. However, even in this small example application 75% of the students agree that Céu's sequential control flow is easier to understand; 45% even considerably. Votes for time need and overall impression are convincing too. With a tendency for Céu, changeability seems to be comparable to C.

Furthermore, we investigated all votes in total. Every questionnaire provided four votes, one for each quality attribute. This led to $4 \cdot 20 = 80$ votes in total. It appears that 7 votes are less, 18 votes are equal and 54 votes are greater than zero. This means that, in direct comparison, 8.75% of all votes favor C, 22.50% have not noticed any difference, but the majority of 67.50% favors the synchronous approach in Céu.

Also, remember that students had been familiar with C in reading and writing for at least one year. They were used to the low-level automata approach and asynchronous execution from the context of microcontroller programming. Actually, they were not even aware that there was an alternative to the asynchronous model. Nonetheless, 180 minutes appeared to be sufficient to comprehend the synchronous execution policy and read and write simple Céu code.

Finally, we are aware that the number of survey participants is far to small to be representative. However, we believe that the results support our argumentation

and indicate at least a trend towards an improved software quality provided by the synchronous-reactive programming approach.

7 Conclusion

In this chapter, we draw conclusions from this research and outline some possible issues for future work.

7.1 Summary

In this thesis, we presented a case study of a real-world industrial smart device that investigated the feasibility and suitability of synchronous programming for simplifying software engineering and improving software quality of resource-constrained embedded applications that are exposed to transformational and reactive concerns at the same time.

We examined the engineering challenges and quality issues of reactive concerns in an existing production code. An embedded operating system extends C's conventional sequential tool set by support for event handling, concurrency and temporal behavior. It appears that the provided technologies encourage a number of shortcomings and typically make developing sound and transparent code a challenging task. In particular, the logic required to make the application actually work is scattered and mainly resides in the execution environment rather than in the application code. The induced non-locality requires to navigate back and forth between multiple places in source and configuration files in order to see and comprehend everything the code actually does. Further, it leads to several engineering challenges such as a torn and convoluted control flow, the loss of function-oriented decomposition and hierarchical composability as well as explicit synchronization and manual timer management. Those issues make application code generally hard to program, comprehend and maintain. The final software solution is fragile and enhancing its quality is difficult to achieve due to the technologies in use.

In comparison, we presented and examined a synchronous reimplementation of the reactive part in Céu. In this context, we proposed a domain-oriented software architecture that allows a seamless reconciliation of reactive, synchronous and transformational, asynchronous code. Also, we provided some general considerations, guidelines and best practices for effectively deploying Céu's language constructs, for example in order to apply a stepwise refinement strategy to the architectural design of synchronous code. Based on several qualitative discussions, we demonstrated that Céu's domain-specific language support overcomes the engineering challenges and quality issues imposed by callbacks and inversion of control. In particular, in Céu the intelligence required to make the application work solely manifests on language level and hence is shifted from the execution environment back into the application's business logic. By this, the

developer has full control over how concurrent execution entities interact and how the passage of physical time influences the system behavior. This makes the program completely independent from the operating system and regains locality. Céu's synchronous control statements and abstraction entities effectively recover a linear control flow, function-oriented decomposition and hierarchical structuring. Its inherent, continuous synchronization eliminates any burden of error-prone explicit synchronization. Also, the adoption of time as a physical quantity in the language itself makes specifying and understanding temporal behavior a comparably easy task.

Additionally, Céu's efficient abstraction capabilities allow to deploy well-established object-oriented software design patterns – which are known to improve software quality – without violating embedded constraints. Furthermore, it enables to specify and run deterministic, reproducible unit tests, covering event processing, concurrency and temporal behavior, in the language itself, not depending on any external tool. This allows to escape from the rigorous dependency on a suitable hard- and software fixture in order to test reactive embedded code.

Our quantitative code analysis confirms that the synchronous approach enables a clear separation of different reactive functionalities while keeping their interfaces local. Further, it indicates a reduction in code size of more than 50 percent. The reduced code verbosity allows developers to focus on business logic and to express the examined use case in a concise and readable way.

In a user study we evaluated software quality aspects of the synchronous paradigm based on practical experiences from embedded software developers. According to the participants reactive behavior is easier to program and comprehend in Céu compared to the conventional callback workaround.

To sum it up, our deployment and evaluation shows the feasibility and suitability of synchronous programming in resource-constrained, real-world industrial embedded applications. By using synchronous programming, we were able to recover fundamental software engineering principles while, at the same time, fulfill the strong resource limitations – a combination that is known to be hard to achieve.

7.2 Future Work

The research presented in this thesis points out some possible directions for future work.

- The development of a general test framework is a desirable task in order to simplify and automate regression testing. Test cases could be specified in a lightweight scripting language that automatically generates and runs the required Céu code for each test case according to the pattern presented in Section 5.4.2. In this context, a state-of-the-art debugging tool for Céu code would be generally of great help in order to inspect failed test cases for instance.

- Our quantitative evaluation does not compare the executables of the existing productive code and the synchronous reimplementation with respect to resource

consumption such as memory footprint and processing time. This was not possible since the reimplementation is deliberately based on a different hard- and software platform. Thus, it might be interesting to port the platform interface code to the OSEK operating system in order to compile and run the synchronous reimplementation on the existing microcontroller platform. This would also enable the deployment on existing devices.

- Our user study could be extended in two ways: First, it could be made more realistic and representative by increasing the number of participants and the scale or style of the exercises. For example, we could imagine to have two developer groups working for an extended period of time – several weeks – on the same reactive embedded application. The first group uses C for implementation while the second group uses Céu. Second, it should additionally take the synchronous abstraction capabilities into account.

- In our work, we presented a general approach for implementing soft and hard timeouts in Céu (see Section 5.2.2). Also, Sant'Anna et al. [SIR12] consider a "sampling" and "watchdog" pattern. For this reason, we are confident that, if Céu is deployed in more use cases, there might be a chance to identify and extract a catalog of synchronous design patterns that generally solve a certain class of problems in the reactive domain – similar to their object-oriented counterparts.

- In order to find its way into successful industrial use, economic aspects of synchronous programming remain to be investigated. Examples are:
  - The learning curve for synchronous programming. How much effort is required for language trainings (for students or employees)?
  - The degree of maturity of the language and its tool chain. How likely are changes to the formal syntax or semantic? How stable is the compiler implementation?
  - The development tool support. Is there any Integrated Development Environment (IDE) available that provides state-of-the-art programming support?[1]

Finally, we believe that our work generally suggests a practicable way of improving embedded software quality in reactive industrial applications.

[1] A very first prototype for the Eclipse IDE has already been developed successfully [Lan16].

A User Study Exercises and Questionnaire

For all exercises we assume that the program code is executed on the following system:

An embedded system is connected to a *computer mouse* and a *computer keyboard*. Its behavior is entirely event- and time-triggered. The following external input events can be received and processed:
(1) `LEFT` - The left mouse key has been clicked.
(2) `RIGHT` - The right mouse key has been clicked.
(3) `DOWN` - A keyboard key has been pressed.
(4) `UP` - A keyboard key has been released.
The events `DOWN` and `UP` additionally provide the ASCII code of the corresponding key. In *C*, the ASCII code can be read from the registers `REG_KEY_DOWN` and `REG_KEY_UP` respectively. In Céu, it is provided as the payload of the respective event.

---

1. Exercise

   (a) The following programs in *C* and Céu use a slightly different approach in order to switch a green and a red LED on and off. At program start, both LEDs are off.

```c
enum State {INIT, LEFT};
State state = INIT;

ISR(LEFT) {
  if(state == INIT) {
    ledOn(GREEN);
    ledOff(RED);
    state = LEFT;
  } else if(state == LEFT) {
    ledOff(GREEN);
    state = INIT;
  }
}

ISR(RIGHT) {
  static u8 red_on = 0;
  if(state == LEFT) {
    if(red_on) {
      ledOff(RED);
      red_on = 0;
    } else {
      ledOn(RED);
      red_on = 1;
    }
  }
}
```

```
input void LEFT;
input void RIGHT;

loop do
  await LEFT;
  _ledOn(GREEN);

  par/or do
    await LEFT;
    _ledOff(GREEN);
    _ledOff(RED);
  with
    loop do
      await RIGHT;
      _ledOn(RED);
      await RIGHT;
      _ledOff(RED);
    end
  end
end
```

   For each of the following event sequences, determine the state of the green (G) and the red (R) LED after the last event has been processed!

		C		Céu	
	Event Sequence	G	R	G	R
1	`LEFT`, `LEFT`, `RIGHT`, `RIGHT`	on/off	on/off	on/off	on/off
2	`LEFT`, `RIGHT`, `RIGHT`, `RIGHT`, `LEFT`	on/off	on/off	on/off	on/off
3	`RIGHT`, `LEFT`, `RIGHT`, `RIGHT`, `RIGHT`	on/off	on/off	on/off	on/off
4	`RIGHT`, `LEFT`, `RIGHT`, `LEFT`, `RIGHT`	on/off	on/off	on/off	on/off
5	`LEFT`, `RIGHT`, `LEFT`, `LEFT`, `RIGHT`	on/off	on/off	on/off	on/off

(b) The following implementations are given in *C* and Céu.

```c
enum State {INIT, KDOWN, KUP, RIGHT};
State state = INIT;
int ret = 0;
uint8_t kd;

ISR(RIGHT) {
  if(state == KDOWN) {
    state = RIGHT;
  } else if(state == KUP) {
    state = INIT;
  }
}

ISR(DOWN) {
  if(state == INIT) {
    kd = REG_KEY_DOWN;
    state = KDOWN;
  }
}

ISR(UP) {
  uint8_t ku = REG_KEY_UP;
  if(state == KDOWN) {
    if(ku == kd) {
      state = KUP;
    } else {
      ret = ret + 1;
    }
  } else if(state == RIGHT) {
    if(ku == kd) {
      state = INIT;
    } else {
      ret = ret + 1;
    }
  }
}
```

```
input void RIGHT;
input u8 DOWN;
input u8 UP;

loop do
  var u8 ret = 0;
  var u8 kd = await DOWN;
  par/and do
    loop do
      var u8 ku = await UP;
      if(ku == kd) then
        break;
      else
        ret = ret + 1;
      end
    end
  with
    await RIGHT;
  end
end
```

For each of the following event sequences, determine the value of `ret` after the last event has been processed!

	Event Sequence	*C*	Céu
1	`DOWN(0x20)`, `RIGHT`, `RIGHT`, `UP(0x20)`		
2	`DOWN(0x25)`, `DOWN(0x20)`, `UP(0x20)`, `RIGHT`, `UP(0x25)`		
3	`DOWN(0x20)`, `DOWN(0x36)`, `UP(0x36)`, `UP(0x20)`, `DOWN(0x36)`		
4	`DOWN(0x20)`, `DOWN(0x36)`, `DOWN(0x15)`, `UP(0x36)`, `UP(0x15)`, `RIGHT`		
5	`RIGHT`, `DOWN(0x20)`, `RIGHT`, `UP(0x20)`, `DOWN(0x25)`, `RIGHT`		

2. Exercise
   The following implementations are given in *C* und CÉU.

```
enum State {INIT, RIGHT};
State state = INIT;

ISR(LEFT) {
  if(state == RIGHT) {
    stopTimer(1);
    stopTimer(2);
    printf("C");
    printf("F");
    state = INIT;
  }
}

ISR(RIGHT) {
  if(state == INIT) {
    startTimer(1, 10000);
    startTimer(2, 2000);
    printf("A");
    state = RIGHT;
  }
}

ISR(TIMER1) {
  if(state == RIGHT) {
    stopTimer(1);
    stopTimer(2);
    printf("B");
    printf("F");
    state = INIT;
  }
}

ISR(TIMER2) {
  if(state == RIGHT) {
    stopTimer(2);
    printf("D");
    startTimer(2, 2000);
  }
}
```

```
input void LEFT;
input void RIGHT;

loop do
  await RIGHT;
  _printf("A");
  par/or do
    await 10s;
    _printf("B");
  with
    await LEFT;
    _printf("C");
  with
    loop do
      await 2s;
      _printf("D");
    end
    _printf("E");
  end
  _printf("F");
end
```

   For each of the following event sequences, check whether the behavior of the given implementations is equivalent! Therefore, determine the corresponding `printf()`-output generated by the *C* and the CÉU code!

   i) `10s`, `RIGHT`, `RIGHT`, `1s`, `LEFT`, `5s`, `LEFT`

   ii) `LEFT`, `3s`, `RIGHT`, `8s`, `RIGHT`, `2s`, `LEFT`

   iii) `RIGHT`, `7s`, `RIGHT`, `1s`, `LEFT`, `RIGHT`, `3s`,`LEFT`

   iv) `LEFT`, `10s`, `LEFT`, `LEFT`, `4s`, `LEFT`, `1s`

   v) `RIGHT`, `12s`, `RIGHT`, `LEFT`, `10s`

3. Exercise

(a) The following specification is given:

> After system start, the program should terminate (1) as soon as any keyboard key is pressed or (2) first the left, then the right mouse key is clicked. In case (1), the program should return `0`, else `1`.

Above specification has been implemented in *C* and Céu below.

```c
enum State {INIT, LEFT, TERM};
State state = INIT;
int ret;

int main(void) {
  while(state != TERM);
  return ret;
}

ISR(LEFT) {
  if(state == INIT) {
    state = LEFT;
  }
}

ISR(RIGHT) {
  if(state == LEFT) {
    ret = 1;
    state = TERM;
  }
}

ISR(KEY_DOWN) {
  ret = 0;
  state = TERM;
}
```

```
input void LEFT;
input void RIGHT;
input u8 DOWN;

var int ret = 0;

par/or do
  await DOWN;
  ret = 0;
with
  await LEFT;
  await RIGHT;
  ret = 1;
end

escape ret;
```

Now, the specification has been modified as follows:

> In case (2), the order of left and right mouse clicks is now irrelevant. That is, if the left mouse key is clicked followed by the right, then the program will terminate. The same applies if the right mouse key is clicked first. If the program terminates due to a left mouse click, it should return `1`. If it is terminated by a right mouse click it should return `2` instead. If it is terminated due to a keyboard key is should return `0`.

Determine the required adaptations that must be applied to both implementations in order to account for above changes!

(b) The following specification is given:

> As soon as a keyboard key x is pressed, the left and right mouse clicks should be counted independently from each other. As soon as x has been released, `printf()` should output the ASCII code of x as well as the counter values. Subsequently, the procedure restarts.

Implement above specification in *C* **and** in Céu! You are encouraged to use the previous code examples as templates.

The submission of this page is **anonymous**!

Compare the implementations in *C* and CÉU with respect to the following aspects:

	C considerably better	*C* tends to be better	no difference	CÉU tends to be better	CÉU considerably better
Readability/Comprehensibility	2	1	0	1	2
Changeability/Extendibility	2	1	0	1	2
Time Need	2	1	0	1	2
Overall Impression	2	1	0	1	2

Bibliography

[Ady+02] Atul Adya et al. “Cooperative Task Management Without Manual Stack Management: or, Event-driven Programming is Not the Opposite of Threaded Programming”. In: *Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*. ATEC ’02. Berkeley, CA, USA: USENIX Association, 2002, pp. 289–302. ISBN: 1-880446-00-6.

[And+15] Brian Anderson et al. *Experience Report: Developing the Servo Web Browser Engine using Rust*. 2015. URL: https://arxiv.org/abs/1505.07383 (visited on 06/28/2017).

[AP93] Charles Andre and Marie-Agnes Peraldi. “Synchronous programming: introduction and application to industrial process control”. In: *1993 CompEuro Proceedings*. IEEE, 1993, pp. 461–470. DOI: 10.1109/CMPEUR.1993.289839.

[Bai+13] Engineer Bainomugisha et al. “A Survey on Reactive Programming”. In: *ACM Comput. Surv.* 45.4 (2013), 52:1–52:34. ISSN: 0360-0300. DOI: 10.1145/2501654.2501666.

[BJ13] Karim Barkati and Pierre Jouvelot. “Synchronous Programming in Audio Processing: A Lookup Table Oscillator Case Study”. In: *ACM Comput. Surv.* 46.2 (2013), 24:1–24:35. ISSN: 0360-0300. DOI: 10.1145/2543581.2543591.

[BM06] Michael Barr and Anthony J. Massa. *Programming embedded systems: With C and GNU development tools*. 2nd ed. Sebastopol, Calif.: O’Reilly, 2006. ISBN: 0596553285.

[BB91] Albert Benveniste and Gérard Berry. “The synchronous approach to reactive and real-time systems”. In: *Proceedings of the IEEE* 79.9 (1991), pp. 1270–1282. ISSN: 0018-9219. DOI: 10.1109/5.97297.

[Ben+03] Albert Benveniste et al. “The synchronous languages 12 years later”. In: *Proc. IEEE*. 2003, pp. 64–83.

[Ber00] Gérard Berry. *The Esterel v5 Language Primer Version v5_91*. Sophia-Antipolis, France, 2000. URL: https://cseweb.ucsd.edu/classes/wi17/cse237A-a/handouts/Esterelv5Primer.pdf (visited on 06/28/2017).

[BG92] Gérard Berry and Georges Gonthier. “The Esterel synchronous programming language: design, semantics, implementation”. In: *Science of Computer Programming* 19.2 (1992), pp. 87–152. ISSN: 0167-6423. DOI: 10.1016/0167-6423(92)90005-V.

[BMM11] Nicolas Berthier, Florence Maraninchi, and Laurent Mounier. “Synchronous Programming of Device Drivers for Global Resource Control in Embedded Operating Systems”. In: *Proc. SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*. LCTES ’11. New York, NY, USA: ACM, 2011, pp. 81–90. ISBN: 978-1-4503-0555-6. DOI: 10.1145/1967677.1967689.

[Bos17] Bosch Thermotechnik GmbH. *Logamatic web KM300*. 2017. URL: https://webservices.buderus.at/download/pdf/file/8737803811.pdf (visited on 06/09/2017).

[BF14] Pierre Bourque and Richard E. Fairley, eds. *Guide to the Software Engineering Body of Knowledge (SWEBOK (R)): Version 3.0*. IEEE Computer Society Press, 2014. ISBN: 978-0-7695-5166-1.

[BS91] Frederic Boussinot and Robert de Simone. “The ESTEREL language”. In: *Proceedings of the IEEE* 79.9 (1991), pp. 1293–1304. ISSN: 00189219. DOI: 10.1109/5.97299.

[BW90] Alan Burns and Andy J. Wellings. *Real-time Systems and Their Programming Languages*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc, 1990. ISBN: 0-201-17529-0.

[CRT07] Paul Caspi, Pascal Raymond, and Stavros Tripakis. “Synchronous Programming”. In: *Handbook of Real-Time And Embedded Systems*. Chapman & Hall, 2007.

[CHP71] Pierre Jacques Courtois, F. Heymans, and David Lorge Parnas. “Concurrent Control with "Readers" and "Writers"”. In: *Commun. ACM* 14.10 (1971), pp. 667–668. ISSN: 0001-0782. DOI: 10.1145/362759.362813.

[DF10] Dino Distefano and Ivana Filipović. “Memory Leaks Detection in Java by Bi-abductive Inference”. In: *Fundamental Approaches to Software Engineering: 13th International Conference, FASE 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*. Ed. by David S. Rosenblum and Gabriele Taentzer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 278–292. ISBN: 978-3-642-12029-9. DOI: 10.1007/978-3-642-12029-9_20.

[Dun+06] Adam Dunkels et al. “Protothreads: Simplifying Event-driven Programming of Memory-constrained Embedded Systems”. In: *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*. SenSys ’06. New York, NY, USA: ACM, 2006, pp. 29–42. ISBN: 1-59593-343-3. DOI: 10.1145/1182807.1182811.

[EJ09] Christof Ebert and Capers Jones. "Embedded Software: Facts, Figures, and Future". In: *Computer* 42.4 (2009), pp. 42–52. ISSN: 0018-9162. DOI: 10.1109/MC.2009.118.

[EE15] Embedded Systems Design magazine and Embedded Systems Conference. *2014 Embedded Market Study: Then, Now: What's Next?* 2015. URL: http://cms.edn.com/ContentEETimes/Documents/Embedded.com/MarketStudy/2014-embedded-market-study-then-now-whats-next.pdf (visited on 09/26/2016).

[Est14] Esterel Technologies. *SCADE Suite: Control Software Design*. 2014. URL: http://www.esterel-technologies.com/products/scade-suite/ (visited on 06/08/2017).

[Gam+95] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc, 1995. ISBN: 0-201-63361-2.

[Gay+03] David Gay et al. "The nesC Language: A Holistic Approach to Networked Embedded Systems". In: *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. PLDI '03. New York, NY, USA: ACM, 2003, pp. 1–11. ISBN: 1-58113-662-5. DOI: 10.1145/781131.781133.

[Hal+91] Nicholas Halbwachs et al. "The synchronous data flow programming language Lustre". In: *Proceedings of the IEEE* 79.9 (1991), pp. 1305–1320. ISSN: 00189219. DOI: 10.1109/5.97300.

[Hal93] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems*. Boston, MA: Springer US, 1993. ISBN: 978-1-4419-5133-5. DOI: 10.1007/978-1-4757-2231-4.

[Hal05] Nicolas Halbwachs. "A synchronous language at work: the story of Lustre". In: *Proc. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design*. MEMOCODE '05. 2005, pp. 3–11. DOI: 10.1109/MEMCOD.2005.1487884.

[HP85] D. Harel and A. Pnueli. "On the Development of Reactive Systems". In: *Logics and Models of Concurrent Systems*. Ed. by Krzysztof R. Apt. Berlin, Heidelberg: Springer Berlin Heidelberg, 1985, pp. 477–498. ISBN: 978-3-642-82453-1. DOI: 10.1007/978-3-642-82453-1_17.

[Hee16] Dimitri van Heesch. *Doxygen: Source code documentation generator tool*. 2016. URL: www.doxygen.org (visited on 05/28/2017).

[Hoa+87] C. A. R. Hoare et al. "Laws of Programming". In: *Commun. ACM* 30.8 (1987), pp. 672–686. ISSN: 0001-0782. DOI: 10.1145/27651.27653.

[ISO05] ISO. *Road vehicles – Open interface for embedded automotive applications – Part 3: OSEK/VDX Operating System (OS)*. 2005.

[JPV95] Lalita J. Jagadeesan, Carlos Puchol, and James E. Von Olnhausen. "A formal approach to reactive systems software: a telecommunications application in ESTEREL". In: *Proc. Workshop on Industrial-Strength Formal Specification Techniques*. 1995, pp. 132–145. DOI: 10.1109/WIFT.1995.515485.

[Kas07] Oliver Kasten. "A State-Based Programming Model for Wireless Sensor Networks". Phd. Zurich, Switzerland: ETH Zurich, 2007. URL: https://www.vs.inf.ethz.ch/publ/papers/kasten-astate-2007.pdf (visited on 11/10/2016).

[KR78] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1978. ISBN: 0131101633.

[Lan16] Jonathan Lange. "Integration domänenspezifischer Sprachen in Entwicklungsumgebungen: Konzepte und Realisierung". Bachelor Thesis. Gießen, Germany: University of Applied Sciences, 2016.

[Le +91] Paul Le Guernic et al. "Programming real-time applications with SIGNAL". In: *Proceedings of the IEEE* 79.9 (1991), pp. 1321–1336. ISSN: 00189219. DOI: 10.1109/5.97301.

[Lee05] Edward A. Lee. "Absolutely Positively on Time: What Would It Take?" In: *Computer* 7 (2005), pp. 85–87. ISSN: 0018-9162. DOI: 10.1109/MC.2005.211.

[Lee06] Edward A. Lee. "The Problem with Threads". In: *Computer* 39.5 (2006), pp. 33–42. ISSN: 0018-9162. DOI: 10.1109/MC.2006.180. (Visited on 02/10/2016).

[LT09] P. Liggesmeyer and M. Trapp. "Trends in Embedded Software Engineering". In: *IEEE Software* 26.3 (2009), pp. 19–25. ISSN: 0740-7459. DOI: 10.1109/MS.2009.80.

[Lu+08] Shan Lu et al. "Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics". In: *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XIII. New York, NY, USA: ACM, 2008, pp. 329–339. ISBN: 978-1-59593-958-6. DOI: 10.1145/1346281.1346323.

[MO12] Ingo Maier and Martin Odersky. *Deprecating the Observer Pattern with Scala.React*. 2012. URL: https://infoscience.epfl.ch/record/176887/files/DeprecatingObservers2012.pdf (visited on 11/09/2016).

[Mar11] Peter Marwedel. *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems*. Dordrecht: Springer Netherlands, 2011. ISBN: 978-94-007-0256-1. DOI: 10.1007/978-94-007-0257-8.

[MH15] Christian Motika and Reinhard von Hanxleden. "Light-weight Synchronous Java (SJL): An approach for programming deterministic reactive systems with Java". In: *Computing* 97.3 (2015), pp. 281–307. ISSN: 1436-5057. DOI: 10.1007/s00607-014-0416-7.

[MS92] Gary Murakami and Ravi Sethi. "Terminal Call Processing in Esterel". In: *Proc. IFIP 92 World Computer Congress*. Madrid, Spain, 1992.

[NM12] Mouaaz Nahas and Adi Maait. "Choosing Appropriate Programming Language to Implement Software for Real-Time Resource-Constrained Embedded Systems". In: *Embedded Systems - Theory and Design Methodology*. Ed. by Kiyofumi Tanaka. InTech, 2012. ISBN: 978-953-51-0167-3. DOI: 10.5772/38167.

[Neu93] John von Neumann. "First draft of a report on the EDVAC". In: *IEEE Annals of the History of Computing* 15.4 (1993), pp. 27–75. ISSN: 1058-6180. DOI: 10.1109/85.238389.

[OK13] Robert Oshana and Mark Kraeling. *Software Engineering for Embedded Systems: Methods, Practical Techniques, and Applications*. 1st. Newton, MA, USA: Newnes, 2013. ISBN: 978-0-12-415917-4.

[Poi+98] Axel Poigné et al. "The Synchronous Approach to Designing Reactive Systems". In: *Formal Methods in System Design* 12.2 (1998), pp. 163–187. ISSN: 1572-8102. DOI: 10.1023/A:1008697810328.

[Rus] Rust Community. *The Rust Programming Language: Documentation*. URL: https://doc.rust-lang.org/book/ (visited on 04/19/2017).

[Sak12] Dan Saks. "Unexpected trends". In: *Embedded Systems Design* 25.4 (2012), pp. 31–34.

[SK13] Hemaiyer Sankaranarayanan and Prasad A. Kulkarni. "Source-to-Source Refactoring and Elimination of Global Variables in C Programs". In: *Journal of Software Engineering and Applications* 06.05 (2013), pp. 264–273. ISSN: 1945-3116. DOI: 10.4236/jsea.2013.65033.

[San09] Francisco Sant'Anna. "A Synchronous Reactive Language based on Implicit Invocation". Master Thesis. Rio de Janeiro: Pontificia Universidade Catolica Do Rio De Janeiro, 2009. URL: http://www.ceu-lang.org/chico/luagravity_msc.pdf (visited on 04/26/2016).

[San13] Francisco Sant'Anna. "Safe System-level Concurrency on Resource-Constrained Nodes with Céu". PhD. Rio de Janeiro: Pontificia Universidade Catolica Do Rio De Janeiro, 2013. URL: http://www.ceu-lang.org/chico/ceu_phd.pdf (visited on 08/19/2016).

[SIR15] Francisco Sant'Anna, Roberto Ierusalimschy, and Noemi Rodriguez. "Structured synchronous reactive programming with Céu". In: *The 14th International Conference on Modularity*. Ed. by Robert B. France, Sudipto Ghosh, and Gary T. Leavens. 2015, pp. 29–40. ISBN: 978-1-4503-3249-1. DOI: 10.1145/2724525.2724571.

[SIR12] Francisco Sant'Anna, Roberto Ierusalimschy, and Noemi de La Roque Rodriguez. *Céu: Embedded, Safe, and Reactive Programming*. 2012. URL: ftp://ftp.inf.puc-rio.br/pub/docs/techreports/12_12_santanna.pdf (visited on 05/05/2017).

[San+13] Francisco Sant'Anna et al. "Safe system-level concurrency on resource-constrained nodes". In: *The 11th ACM Conference on Embedded Networked Sensor Systems*. Ed. by Chiara Petrioli, Landon Cox, and Kamin Whitehouse. 2013, pp. 1–14. ISBN: 978-1-4503-2027-6. DOI: 10.1145/2517351.2517360.

[San+16] Rodrigo C.M. Santos et al. "CÉU-MEDIA: Local Inter-Media Synchronization Using CÉU". In: *Proc. 22Nd Brazilian Symposium on Multimedia and the Web*. Webmedia '16. New York, NY, USA: ACM, 2016, pp. 143–150. ISBN: 978-1-4503-4512-5. DOI: 10.1145/2976796.2976856.

[Sch94] Axel Tobias Schreiner. *Object-oriented Programming with ANSI C*. München: Hanser, 1994. ISBN: 3-446-17426-5.

[SG01] D. Simon and A. Girault. "Synchronous programming of automatic control applications using ORCCAD and ESTEREL". In: *Decision and Control, 2001. Proceedings of the 40th IEEE Conference on*. Vol. 4. 2001, 3290–3295 vol.4. DOI: 10.1109/.2001.980329.

[STP05] Robert de Simone, Jean-Pierre Talpin, and Dumitru Potop-Butucaru. "The Synchronous Hypothesis and Synchronous Languages". In: *Embedded Systems Handbook*. Ed. by Richard Zurawski. Vol. 6. Industrial Information Technology. CRC Press, 2005, pp. 8-1–8-23. ISBN: 978-0-8493-2824-4. DOI: 10.1201/9781420038163.ch8.

[SL05] Herb Sutter and James Larus. "Software and the Concurrency Revolution". In: *Queue* 3.7 (2005), pp. 54–62. ISSN: 1542-7730. DOI: 10.1145/1095408.1095421.

[Tan12] Kiyofumi Tanaka, ed. *Embedded Systems - Theory and Design Methodology*. InTech, 2012. ISBN: 978-953-51-0167-3. DOI: 10.5772/2339.

[Ter16] Matthias Terber. "Domänenorientierte Softwarearchitektur mit Céu und Rust am Beispiel eines Heizungsgateways zur Fernüberwachung und Fernparametrisierung". In: *Internet der Dinge: Echtzeit 2016*. Ed. by Wolfgang A. Halang and Herwig Unger. Berlin, Heidelberg: Springer, 2016, pp. 117–126. ISBN: 978-3-662-53443-4. DOI: 10.1007/978-3-662-53443-4_13.

[Ter17] Matthias Terber. "Function-Oriented Decomposition for Reactive Embedded Software". In: *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 2017, pp. 288–295. DOI: 10.1109/SEAA.2017.42.

[TV10] Stefan Tilkov and Steve Vinoski. "Node.js: Using JavaScript to Build High-Performance Network Programs". In: *IEEE Internet Computing* 14.6 (2010), pp. 80–83. ISSN: 1089-7801. DOI: 10.1109/MIC.2010.145.

[Tur12] Jim Turley. “So this is progress”. In: *Embedded Systems Design* 25.3 (2012), pp. 22–24.

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich diese Dissertation selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel genutzt habe. Alle wörtlich oder inhaltlich übernommenen Stellen habe ich als solche gekennzeichnet. Ich versichere außerdem, dass ich die beigefügte Dissertation nur in diesem und keinem anderen Promotionsverfahren eingereicht habe und, dass diesem Promotionsverfahren keine endgültig gescheiterten Promotionsverfahren vorausgegangen sind.

18. November 2018, Lollar

Datum, Ort

Unterschrift

Aachener Informatik-Berichte

This list contains all technical reports published during the past three years. A complete list of reports dating back to 1987 is available from:

http://aib.informatik.rwth-aachen.de/

To obtain copies please consult the above URL or send your request to:

Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen,
Email: biblio@informatik.rwth-aachen.de

2015-01 *	Fachgruppe Informatik: Annual Report 2015
2015-02	Dominik Franke: Testing Life Cycle-related Properties of Mobile Applications
2015-05	Florian Frohn, Jürgen Giesl, Jera Hensel, Cornelius Aschermann, and Thomas Ströder: Inferring Lower Bounds for Runtime Complexity
2015-06	Thomas Ströder and Wolfgang Thomas (Editors): Proceedings of the Young Researchers' Conference "Frontiers of Formal Methods"
2015-07	Hilal Diab: Experimental Validation and Mathematical Analysis of Cooperative Vehicles in a Platoon
2015-08	Mathias Pelka, Jó Agila Bitsch, Horst Hellbrück, and Klaus Wehrle (Editors): Proceedings of the 1st KuVS Expert Talk on Localization
2015-09	Xin Chen: Reachability Analysis of Non-Linear Hybrid Systems Using Taylor Models
2015-11	Stefan Wüller, Marián Kühnel, and Ulrike Meyer: Information Hiding in the Public RSA Modulus
2015-12	Christoph Matheja, Christina Jansen, and Thomas Noll: Tree-like Grammars and Separation Logic
2015-13	Andreas Polzer: Ansatz zur variantenreichen und modellbasierten Entwicklung von eingebetteten Systemen unter Berücksichtigung regelungs- und softwaretechnischer Anforderungen
2015-14	Niloofar Safiran and Uwe Naumann: Symbolic vs. Algorithmic Differentiation of GSL Integration Routines
2016-01 *	Fachgruppe Informatik: Annual Report 2016
2016-02	Ibtissem Ben Makhlouf: Comparative Evaluation and Improvement of Computational Approaches to Reachability Analysis of Linear Hybrid Systems
2016-03	Florian Frohn, Matthias Naaf, Jera Hensel, Marc Brockschmidt, and Jürgen Giesl: Lower Runtime Bounds for Integer Programs
2016-04	Jera Hensel, Jürgen Giesl, Florian Frohn, and Thomas Ströder: Proving Termination of Programs with Bitvector Arithmetic by Symbolic Execution
2016-05	Mathias Pelka, Grigori Goronzy, Jó Agila Bitsch, Horst Hellbrück, and Klaus Wehrle (Editors): Proceedings of the 2nd KuVS Expert Talk on Localization

2016-06	Martin Henze, René Hummen, Roman Matzutt, Klaus Wehrle: The SensorCloud Protocol: Securely Outsourcing Sensor Data to the Cloud
2016-07	Sebastian Biallas : Verification of Programmable Logic Controller Code using Model Checking and Static Analysis
2016-08	Klaus Leppkes, Johannes Lotz, and Uwe Naumann: Derivative Code by Overloading in C++ (dco/c++): Introduction and Summary of Features
2016-09	Thomas Ströder, Jürgen Giesl, Marc Brockschmidt, Florian Frohn, Carsten Fuhs, Jera Hensel, Peter Schneider-Kamp, and Cornelius Aschermann: Automatically Proving Termination and Memory Safety for Programs with Pointer Arithmetic
2016-10	Stefan Wüller, Ulrike Meyer, and Susanne Wetzel: Towards Privacy-Preserving Multi-Party Bartering
2017-01 *	Fachgruppe Informatik: Annual Report 2017
2017-02	Florian Frohn and Jürgen Giesl: Analyzing Runtime Complexity via Innermost Runtime Complexity
2017-04	Florian Frohn and Jürgen Giesl: Complexity Analysis for Java with AProVE
2017-05	Matthias Naaf, Florian Frohn, Marc Brockschmidt, Carsten Fuhs, and Jürgen Giesl: Complexity Analysis for Term Rewriting by Integer Transition Systems
2017-06	Oliver Kautz, Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe: CD2Alloy: A Translation of Class Diagrams to Alloy
2017-07	Klaus Leppkes, Johannes Lotz, Uwe Naumann, and Jacques du Toit: Meta Adjoint Programming in C++
2017-08	Thomas Gerlitz: Incremental Integration and Static Analysis of Model-Based Automotive Software Artifacts
2017-09	Muhammad Hamad Alizai, Jan Beutel, Jó Ágila Bitsch, Olaf Landsiedel, Luca Mottola, Przemyslaw Pawelczak, Klaus Wehrle, and Kasim Sinan Yildirim: Proc. IDEA League Doctoral School on Transiently Powered Computing
2018-01 *	Fachgruppe Informatik: Annual Report 2018
2018-02	Jens Deussen, Viktor Mosenkis, and Uwe Naumann: Ansatz zur variantenreichen und modellbasierten Entwicklung von eingebetteten Systemen unter Berücksichtigung regelungs- und softwaretechnischer Anforderungen
2018-03	Igor Kalkov: A Real-time Capable, Open-Source-based Platform for Off-the-Shelf Embedded Devices
2018-04	Andreas Ganser: Operation-Based Model Recommenders

* These reports are only available as a printed version.
Please contact biblio@informatik.rwth-aachen.de to obtain copies.