Towards Portable Real-Time Software Components

Dipl.-Ing. Claudiu Ciprian Fărcaș
Department of Computer Sciences
University of Salzburg

A dissertation submitted for the degree of

Doctor technicae (Dr. techn.)

Salzburg, June 2006
Abstract

Real-time systems are nowadays present all around us, in small embedded devices such as household electronics to complex control system in airplanes and power plants. The development of reliable software for such systems is a challenge. Moreover, changing the underlying platform for simple purposes such as a processor upgrade may severely affect the behavior of the real-time software. Working with distributed systems is even more difficult, and transitioning from one system to another is typically impossible.

We address these problems by introducing a development framework for deterministic and portable real-time software. We present a component-oriented language, namely the Timing Definition Language (TDL), which allows for development of time-triggered applications with logical execution times. Along with the language, we introduce a novel concept of transparent distribution in real-time systems. It leverages the developer from the constraints of the platform, allows independent development of real-time components, and enables later component integration regardless of the target system topology, i.e., single-node or distributed. Therefore, TDL provides the software properties of portability, determinism, and compositionality for both functionality and timing behavior.

From a run-time perspective, we provide a performance-optimized algorithm for encapsulating the timing behavior of TDL components into portable embedded code. We introduce the algorithms of a corresponding virtual machine that supports parallel composition with cyclic dependencies between TDL components. Based on that, we present the algorithms for automatic generation of the glue-code that binds the timing with the functionality of TDL components.

We further provide a solution for portability across platforms and network topologies through the introduction of a Platform Abstraction Layer, which implements the mechanism for transparent distribution and mediates the interaction between the user-functionality, the virtual machine, and the underlying platform.

The results of these contributions are substantiated by case studies on various single-node and distributed platforms, which prove the viability of our approach.
Acknowledgements

First, I would like to thank my supervisor, Prof. Dr. Wolfgang Pree, for accepting me as a PhD student and for giving me the opportunity to work within the Software Research Lab on the MoDECS project. The last three years under his guidance were a continuous race to go beyond the state-of-the-art and to invent the future. I am very grateful for his vision, support, and encouragement to find solutions for the most difficult problems. The Timing Definition Language (TDL) would not exist without his continuing efforts to push the boundaries of knowledge.

For his inspiring discussions and support I would like to thank my co-advisor and reviewer, Prof. Dr.-Ing. Christoph Kirsch. His real-time background, theoretical and practical expertise with Giotto, guided me in finding the right abstractions and dealing with challenging questions.

My thanks go also to Dr. Josef Templ for providing many useful ideas, in particular for proposing the TDL and developing the TDL compiler and for his advices on implementing the compiler plugins; and to Michael Holzmann for his help with the implementation of the communication subsystem, fruitful discussions, and hardware assistance. Gerald Stieglbauer was always around with ideas, kind words, or useful practical information. Thanks for everything.

For his continuous support since my arrival in Salzburg, long and interesting discussions (on and off-topic), exchange of ideas, my gratitude goes to Dr. Sebastian Fischmeister. He also encouraged me to strive for the best, express my knowledge through research papers, and focus on the dissertation. Dr. Guido Menkhaus contributed with valuable discussions on research papers, and advices for my teaching experience.

For help with all the administrative burden and practical stuff many thanks to our secretaries Adriana Pratter and Petra Kirschweger.

Life during these years would have been unbearable without the cheerful moments with my friends here in Salzburg, Dan and Monika. Instant messaging and VoIP shortened the distance to my home friends Andrei and Arpiko, and others around the world that always had a smile for me.

For her long-standing and sincere love and understanding, I want to express my warmest thanks to my wonderful wife Emilia. She supported me through the years, cared for me, inspired me, and always encouraged me to go for my dreams, and enhance my social and technical experience.
Finally, my deepest and sincerest thanks go to my parents for their patience, help, and support during the preceding years. They have contributed more than anyone else to make my studies possible and to the successful completion of my work. Even from afar, my family was always present and provided me with love, support and inspiration.

This research was mostly performed within the MoDECS Project (Model-Based development of Distributed Embedded Control Systems) and was supported in part by the FIT-IT Embedded Systems grant 807144 provided by the Austrian government through its Bundesministerium für Verkehr, Innovation und Technologie.
### CONTENTS

3.3.1 Distribution unit ........................................... 61
3.3.2 Transparent distribution concept ............................ 64
3.3.3 Stub modules ................................................. 68
3.4 Operational semantics ........................................... 70
  3.4.1 Single module .............................................. 70
  3.4.2 Multiple independent modules .............................. 78
  3.4.3 Multiple modules with import relationships ............... 79
  3.4.4 Distributed applications .................................. 81
4 The Virtual Machine ............................................. 83
  4.1 E-Code .................................................. 84
    4.1.1 E-Code instruction set .................................. 85
    4.1.2 Encoding TDL semantics into E-Code ...................... 87
  4.2 E-Machine ................................................ 93
    4.2.1 Execution of a TDL module .............................. 96
    4.2.2 Modular composition .................................... 97
    4.2.3 Cyclic dependencies ................................... 99
    4.2.4 Transparent distribution ............................... 103
5 The Platform Abstraction Layer ................................. 105
  5.1 Conceptual overview ....................................... 106
  5.2 TDL Mappings (TDLMp) ..................................... 110
    5.2.1 Module wrappers ....................................... 113
    5.2.2 Module stubs .......................................... 133
    5.2.3 Module dependencies .................................. 135
  5.3 Runtime Resource Management (RTRM) ......................... 136
    5.3.1 Time .................................................. 136
    5.3.2 Tasks ................................................ 138
  5.4 TDL Scheduler ............................................. 139
    5.4.1 Rate Monotonic scheduling ............................. 141
    5.4.2 Earliest Deadline First scheduling .................... 142
  5.5 TDLComm layer for transparent distribution ................ 145
    5.5.1 Packet exchange ....................................... 150
    5.5.2 Clock synchronization .................................. 157
    5.5.3 Communication protocol using hyper-periods ........... 160
    5.5.4 Communication protocol with multiplexing over mode periods 166
    5.5.5 Fault tolerance (TDL-FT) ............................... 170
6 From TDL Models to Executable Code ............................ 175
  6.1 Overview of the TDL tool-chain ............................. 175
  6.2 TDL Compiler ............................................. 178
  6.3 Automatic code generation and configuration ................. 179
Chapter 1

Introduction

Since their invention, computers played ever increasing roles in our lives, and nowadays they are not only present as the classical PC, but in a wide range of devices from digital watches, multimedia devices, to aircraft control systems, communication devices, cars, industrial robots, medical instruments, and power distribution systems. A survey (Halfhill 2000) revealed that around 98% of processors are "embedded" in such systems and many others handling different automation processes. For most embedded systems, the timing of their interactions with the controlled environment is as important as the correctness of their computations. Thus, we also know them under the term of "real-time systems". Consequently, their software becomes "real-time software", and their development methodology yields the "real-time programming" discipline.

1.1 Motivation

In recent years, advances in hardware devices, networks, and related technologies are hardly matched by advances in the construction of real-time software. Moreover, the complexity of real-time control systems is increasing at a faster pace than the traditional development methodologies can handle. Traditional development of software for embedded systems is highly platform specific. The hardware costs are reduced to a minimum whereas high development costs are considered acceptable in case of large quantities of devices being sold. However, with more-powerful processors even in the low cost range, we observe a shift of functionality from hardware to software and in general more ambitious requirements. A luxury car, for example, comprises up to 80 electronic control units interconnected by multiple buses that execute more than a million lines of code.

Using traditional real-time software development methodologies it is hard to perform the migration of the software from one platform to another, even if the platform change is a simple hardware upgrade to a faster processor. One of the main reasons is the different role of time from the perspective of software and control engineering (Sifakis 2001). The control engineering perspective is that processes evolve in
CHAPTER 1. INTRODUCTION

continuous real-time, delays are small, and jitter\(^1\) is negligible. From the perspective of software engineering, a set of tasks needs to be scheduled, and time evolves discontinuously, because time elapses for a task only when it is active and running. Consequently, the timing of the resulting real-time applications is more of an accidental consequence of their implementation. More problems from the conceptual and functional differences between the underlying real-time operating systems, lack of compilers and corresponding run-time environments, and additional testing and verification phases for the behavior of the new system lead to increasing development costs and time-to-market for real-time software.

In order to cope with the increased complexity of the resulting software, a more platform independent high-level programming style becomes mandatory. In case of real-time software, this style applies not only to functional aspects but also to the temporal behavior of the software. Dealing with time, however, is not covered appropriately by any of the existing component models for high-level languages.

Existing programming-language facilities for splitting source code into several files or classes do not suffice for the complexity of real-time applications. Object-oriented design technology can be applied; however, it is just one part of the solution, as there is no way to express the timing behavior or the relationship between objects belonging to different parts such as sensors and tasks. A component-oriented design analog to other technical sciences is crucial but missing in real-time systems; it would allow for decomposition of a complex application into smaller interacting components with well-defined interfaces and behavior.

Extensibility is important for future real-time systems; with increasing hardware performance, there is an obvious desire to run more applications on a system. Current solutions for component-oriented design do not suffice, because in existing real-time systems adding one application to a working system may disturb old functionality, leading to non-deterministic behavior, or may require fundamental changes in the whole system. The developer should to be able to compose applications out of components and to add more applications to one processor without perturbing the timing properties of existing applications and overall behavioral determinism.

In summary, the development of embedded control systems (a subset of the real-time systems domain) encounters difficulties derived from the software evolution, increasing safety and reliability requirements, limited resources, and hardware changes. Solving these problems requires new programming paradigms, better abstractions for the platform and the interaction between the real-time system and the external environment, languages, smart compilers, and additional tools (such as automatic code generators and simulation environments) to help the developer in designing and implementing complex real-time applications.

\(^{1}\)the relative time difference between the logical and physical moments when the interaction occurs
1.2 Portability aspects

Although a widely used term in computer science, *portability* is not a well-defined concept. Depending on additional concepts such as *data*, *source*, and *binary code*, the meaning of the term is totally different. A mere analogy with hardware, "having the same meaning for software as *compatibility* for hardware" (Dahlstrand, 1984), does not give us a solution, as it introduces another ambiguous term "compatibility". The only consensus is that portability is *beneficial* for software. Initial portability concerns about the character-set used (e.g., ASCII vs. EBCDIC) or physical media, which were en-vogue around 1970, no longer apply today. Further limitations of compilers and CPU processing capabilities presented by (Tanenbaum et al., 1978) are mostly solved. Two aspects of software portability are relevant today.

**Software.** The portability of software programs is today an important software design principle, because market share of software is directly proportional with the number and the availability of supported platforms. Over the years, the concept had different interpretations, for example, a software is *portable* if its documentation and design plans are sufficient for a complete reimplementation on a new platform (Wallis, 1982), or the effort of porting it to a new platform is much less than the effort of rewriting it for the new platform (Brown and Hall, 1976). Other less labor-intensive interpretations required the execution of the unmodified software on a new platform after a simple recompilation (Hague and Ford, 1976). An alternate term of *transportable* software was used, in the case when the required software changes could be automated. (Dahlstrand, 1984) introduced the portability as a measurable quantity related with the percentage of source-code lines unchanged when porting the software to a new platform. The usage of high-level languages instead of assembly language or other platform specific languages, was considered as the first step in a design for portability software development methodology. Furthermore, isolation of hardware specific functionality into modules or drivers was accepted as good practice. Mathematical algorithms or other domain specific computations require specialized languages that might appear portable, but have their own problems, such as the floating-point precision of the underlying hardware.

Various approaches appeared to address these problems using *emulators*, *abstraction layers*, or *intermediate languages*. The emulators replicate in software the behavior of the original hardware, abstracting processors or entire computational systems. Operating-system emulators map the original OS functions over the functions of the executing platform’s OS, making the necessary adjustments on parameters and underlying design concepts. Abstraction layers were introduced in the design of portable software, as software bases that abstract from the underlying operating system and provide specific functionality to the upper level software (Weiser et al., 1989).

Intermediate languages are typically low-level languages targeting a fictional hardware, that is the *abstract machine* (Newey et al., 1972; Poole and Waite, 1969), which can be mapped to machine-specific languages (Brown, 1972) and then possibly inter-
CHAPTER 1. INTRODUCTION

preted (Kornerup et al., 1980). The initial idea was a linguistic switchbox (Conway, 1958) for a universal language (Strong et al., 1958), with a corresponding compiler composed of a front-end and a back-end. The language later referred as UNCOL (Steel, 1960, 1961) is obsolete by today’s standards with the advent of arrays and index registers. Later approaches used ”just-in-time” compilation and the similar concept of virtual machines, for example starting with the early Pascal’s P-Code (Nori et al., 1981) and up to modern Java and .NET technologies. Other ideas relate to high-level source-to-source translations (Atkinson et al., 1989) through a general-purpose language as the intermediate language (e.g., the C language (Kernighan and Ritchie, 1978)) or portable linearized parse trees (Ganapathi and Fischer, 1984).

Data. The internal representation of data on various platforms can greatly affect the portability of software. Algorithms using real numbers can provide different results on platforms having different internal floating-point representations. Alternatives using rational numbers, expressed via integers, can also be influenced by the byte ordering in memory, especially when the corresponding data has to be serialized for storage or communication purposes. Prior research for data-definition-languages (Sibley and Taylor, 1973), which describe not only the logical data structures but also their physical representation on a platform, and machine-independent (Atkinson, 1977) intermediate representations for transferring data between different platforms, never had success in practice. The principal reasons are the hardware advancement, which provided the physical means for fast data conversion, and the data storage standards, which mostly rely on the popularity of a set of hardware platforms.

Portability in real-time systems

Existing real-time system are highly platform dependent, making changes to the underlying real-time operating system or hardware upgrades a difficult task. Porting legacy applications to new software platforms requires significant investments in time and effort; thus, a straightforward migration solution with minimal changes in applications is highly desirable.

Distribution adds another dimension to portability (Kopetz, 1997). Although hardware performance is increasing, a faster processor may not solve timing problems. Certain complex applications require a number of processing nodes interconnected via one or more networks, mostly because of wiring problems, sensor localization, or fault-tolerance reasons. Traditional approaches for distributed run-time environments require extensive planning of the network communications prior to actual software development. In the end, such systems impose great restrictions to the design of each node and its corresponding software. New software-technologies are required to enable independent development of components with later integration into a target platform consisting of one or more nodes.
1.3. OBJECTIVES AND SUMMARY OF CONTRIBUTIONS

**Deterministic behavior** is crucial regardless of the portability or distribution aspects, as the behavior of the system is what defines it as appropriate or not for a controlled object. Most programs that pass testing and validation phases provide value determinism, that is, for a given input value, we can always predict the output value. In the case of real-time systems, this level of determinism is not enough. Time determinism is at least as important, meaning that the program must always provide for the same input received at always the same time instant, the same predictable output at always the same predictable time (Burns and Wellings, 2001; Kopetz, 1997). The deterministic behavior of a distributed system is typically orders of magnitude more difficult to prove than a single-node system. Automatic tools for the analysis, testing, and validation of such systems are highly needed. Another requirement in most application is a minimal jitter in reaction with the environment.

**Fault tolerance** is typically non-existent in simple single-node systems, but required in safety-critical distributed systems. Faults cannot be completely avoided in real-world applications, because of factors independent of the computational system, such as environment driven faults. Fault-tolerance is important for detecting the errors and applying the necessary steps to correct, repair, or adapt the behavior of the system before a failure occurs. Fault-tolerant systems break the chain from fault/error to failure and provide increased reliability (Kopetz, 1997). Most existing systems require significant design and development efforts to support a small set of tolerable faults.

### 1.3 Objectives and summary of contributions

A particularly promising approach towards a high-level component model for real-time systems has been laid out in the Giotto project (Henzinger et al., 2003) by introduction of **logical execution time** (LET), which makes the timing an explicit part of the real-time software design. The LET concept abstracts the physical execution time of a computation on a particular platform. Thus, it becomes possible to change the underlying platform and even to distribute components between different nodes without affecting the overall system behavior. Giotto, however, is primarily an abstract mathematical concept and there exist only simple prototype implementations, which show some of the potential of LET: static time-safety checks and platform-independent embedded code (E-Code) executed by virtual machines (Henzinger and Kirsch, 2002).

Starting from the theoretical foundation of Giotto, we address some of the aforementioned problems, by using a new component-oriented high-level language to decouple the timing of an application from its implementation. We aim at providing new concepts for **transparent distribution** of components within a distributed system, to ease the development of complex applications. We intend to speed-up the development cycle through automatic code generation, and to ensure minimal software migration costs by providing portable real-time components and a lightweight, portable, and deterministic run-time environment based on a virtual machine.
CHAPTER 1. INTRODUCTION

The contributions of this dissertation that advance the state of the art of developing real-time systems cover three aspects, which are detailed in the following paragraphs:

1. **component-based modeling** for hard real-time systems,

2. **virtual execution environment** for run-time determinism of component behavior,

3. **platform abstraction layer** for transparent component distribution and code portability among a variety of platforms.

The *Timing Definition Language* (Templ, 2004) extends Giotto with a component model, improved semantics, and full support for distribution. We contributed to the development of TDL and its component model during the MoDECS project\(^2\) at the University of Salzburg. TDL allows deterministic software composition out of individual components, which run in parallel, may exchange information, and switch modes independently. Giotto addresses the platform independence at the timing level (through E-Code) and we go one-step further towards the portability at the integration, distribution, and implementation levels. The virtual-machine concept (Henzinger and Kirsch, 2002) together with the platform abstraction layer lead to a portable runtime environment.

**C1 - Timing Definition Language** This dissertation details the following personal contributions to TDL and its component model:

- We introduce the formal definition of the language with its component model, i.e., the *TDL module*, and the *import* relationship. We discuss the improved semantics of TDL in relationship with its precursor Giotto (Henzinger et al., 2003).

- We introduce the novel concept of transparent distribution in real-time systems with the *stub module* at its foundation. We show that this abstraction enables independent development of TDL modules regardless of their later integration into a single-node or distributed system. Two published papers (Farcas et al., 2005a,b) synthesize these results.

- We define the operational semantics of TDL for single module and for modular composition. We further show that the transparent distribution concept maintains the modular-composition semantics of TDL within distributed systems.

**C2 - Virtual Machine** Starting with the original idea of Giotto regarding a virtual machine for the runtime environment, we adapt it to TDL semantics and introduce a number of improvements:

\(^2\)The Model-Based development of Distributed Embedded Control Systems project was supported by the FIT-IT Embedded Systems grant 807144 provided by the Austrian government through its Bundesministerium für Verkehr, Innovation und Technologie.
1.3. OBJECTIVES AND SUMMARY OF CONTRIBUTIONS

- We define a TDL specific E-Code and present the encapsulation of the timing behavior of a TDL module through E-Code. We further introduce a performance-optimized compilation algorithm (regarding E-Code size and number of preemptions).

- We present the virtual machine of TDL (E-Machine) that maintains at run-time the logical behavior of a TDL module. We introduce a lightweight algorithm of the E-Machine for the execution the E-Code of a TDL module. We then extend the algorithm to support modular composition with regular TDL semantics, i.e., where the import relationships form a directed acyclic graph. Through this algorithm, the E-Machine executes multiple TDL modules in parallel with a minimal run-time overhead (regarding the number of preemptions).

- We show that regular E-Code programs are not composable when there are data dependencies between components. Although TDL solves partially the problem by including import-order information into the binary E-code file, it cannot solve cyclic-dependencies between modules. Other languages relying on E-Code either have no component model (e.g., Giotto) or have similar unsolved problems. Therefore, we introduce several solutions for true E-Code composability, which support arbitrary data dependencies between components and cyclic import relationship (i.e., import order is no longer required), and discuss their advantages and implications on the generation and execution of the E-Code. We further provide an algorithm for the E-Machine supporting modular composition with cyclic dependencies.

- We explain why the stub-modules are the perfect means for transparent distribution, that their inclusion does not affect the operation of the E-Machine, and that they enable the independent development of components. We then show their importance for the late integration of optional components into an existing distributed system.

C3 - Platform Abstraction Layer  We show that the virtual machine itself is not enough to ensure the portability of real-time components. We introduce a Platform Abstraction Layer that abstracts from the underlying platform, its scheduling scheme, and the topology of the system (i.e., single or distributed system). As key ingredients of this abstraction, we present:

- **TDL Mappings** for an automatic portable middleware generation, i.e., the glue-code that binds the timing with the functionality of each TDL module. As consequence, we attain source-code level compatibility between various platforms.

- **Runtime Resource Management** (RTRM) abstraction of the hardware platform, the supported computational unit, clock, and its resource sharing mechanisms.
CHAPTER 1. INTRODUCTION

We achieve a high degree of independence regarding the classical concepts for concurrency support such as tasks, processes, threads, semaphores, locks, mutexes, and their platform specific APIs and implementations.

- **TDL Scheduler**, which abstracts the platform scheduling scheme and provides automatic support for Rate-Monotonic and Earliest-Deadline-First scheduling algorithms. We further introduce a high-performance hybrid version of the EDF algorithm for multi-module run-time composition, which uses compile-time information to improve the scheduling complexity from $O(\sum_{M \in \text{Modules}} ||\text{Tasks}[M]||)$ to $O(||\text{Modules}||)$. This enhancement enables the execution of applications with high CPU requirements and linear complexity network-constrained scheduling for distributed systems.

- **TDLComm**, which abstracts from the topology of the system and provides the mechanisms for transparent distribution of TDL modules. Important aspects are:
  - the relation between the TDLComm layer, E-Machine, and modules;
  - the communication through TDLComm under the control of TDL Scheduler;
  - the task scheduling with constraints from communications scheduling; the problems encountered are outlined in the paper [Menkhaus et al., 2005];
  - the communication protocols and the automatic communication-schedule generation;
  - the support for event-based and time-triggered buses;
  - the support for fault-tolerance at module level and at communication level.

We show that the concept of transparent distribution through the TDLComm layer and the stub-modules is sound and delivers the same deterministic behavior in time and value on distributed platforms [Farcas et al., 2005b]. The Platform Abstraction Layer clearly enables the migration from a "towards platform" development methodology to a "towards application" development methodology.

We finally present two experimental case studies, which corroborate our claims of portability of real-time components over different single-node platforms (in terms of CPU, internal architecture, real-time operating system) and distributed platforms.

1.4 Structure of the dissertation

In this chapter, we presented the context of the dissertation and its motivation. We also presented our objectives and a summary of our contributions that advance the state of the art of developing real-time systems. The rest of the dissertation is organized as follows:

---

3 we note with $||x||$ the cardinality of $x$
1.4. STRUCTURE OF THE DISSERTATION

Chapter 2 starts with a classification of real-time systems using multiple criteria. We present the real-time programming models (i.e., synchronous, scheduled, and timed model) and the advantages each model brings to the supported range of applications. Following, we present an overview of the approaches for portability in the form of micro-kernels and standards for full-featured real-time operating systems. We analyze popular existing systems from the point of view of their capabilities, scheduling mechanisms, and supported range of applications. For distributed real-time systems, we analyze in Section 2.4 the communication technologies that are crucial for the behavior of the overall system and the underlying event-triggered and time-triggered protocols.

Chapter 3 describes our approach for portable real-time software components, namely the Timing Definition Language. TDL is a high-level description language for specifying the explicit timing requirements of a time-triggered application, which may be constructed out of several independently developed components running in parallel and dynamically exchanging information. The first section presents the core concept of the language, that is the Logical Execution Time \cite{Henzinger}. In Section 3.2, we describe the component model of \cite{Templ} with its constituents: the modes of operation, tasks for computation, sensors and actuators for the interaction with the environment, guards for control flow, ports as interfaces between TDL entities, and drivers for port-value information exchange. We introduce in Section 3.3.2 a novel concept of transparent-distribution in real-time systems \cite{Farcas}, which is crucial for the development of platform-independent real-time components. The components can be developed independently and then integrated to form complex applications, regardless of the topology of the final system. We formally define in Section 3.4 the operational semantics of the language by analyzing first the case of a single module, then the modular composition with import relationships between modules, and concluding with the case of distributed modules with dependencies.

Chapter 4 presents our approach for a runtime environment for TDL, based on the initial idea of the Giotto language \cite{Henzinger}: a virtual machine \cite{Henzinger and Kirsch} to handle the timing aspects of an application. We go one-step further from Giotto towards the portability at the integration, distribution, and implementation levels. In the first section of this chapter, we introduce the instruction set of our virtual machine, the so-called E-Code \cite{Henzinger and Kirsch}. We then present the algorithms for specifying the timing behavior of a TDL module through E-Code. We continue in Section 4.2 with the virtual machine itself (E-Machine) and its algorithms for executing the E-Code of one or more TDL modules. In other words, we present the run-time modular composition. We then show that current E-Code specifications are insufficient when there are data dependencies between components, and further provide a solution to the current limitation of TDL’s E-Code with regard to modules with cyclic import dependencies. We complete the chapter with an analysis of the effect of transparent distribution over the operation of the E-Machine.
CHAPTER 1. INTRODUCTION

As significant contribution towards portability, we present in Chapter 5 an abstraction of the target platform, namely the Platform Abstraction Layer. PAL allows us to create a portable middleware in C, which customizes the TDL binding rules (i.e., the mappings of each TDL entity to platform specific constructs), manages the run-time resource of most real-time platforms, and abstracts from the topology of the system. We start in Section 5.1 with a conceptual overview of the PAL, the interaction of the developer with a simplified TDL tool-chain using PAL, and the purpose of the three major parts of PAL: TDL Mappings, Runtime Resource Management, and TDLComm. We detail the mappings of the TDL entities into corresponding C constructs in Section 5.2. We then present the management of their interactions at run-time by abstracting of time and computational units. Section 5.4 presents the runtime scheduling mechanisms of our PAL through the TDL Scheduler and introduces an optimized version of the EDF (Liu and Layland, 1973) algorithm that has less run-time overhead. In Section 5.5, we present our solution for transparent distribution, that is the TDLComm part, which handles the value and time deterministic exchange of information between modules placed on remote nodes of a distributed system. We conclude the chapter with the presentation of an experimental Fault-Tolerance layer for TDL (TDL-FT) targeted at distributed systems.

In Chapter 6, we present the TDL tool-chain as a practical solution for portable real-time software components. It aids the developer with tools to design and model an application, test and simulate its behavior, generate code automatically, compile, and run the resulted application on a variety of platforms regardless of their topology. The first section depicts an overview of the existing TDL tool-chain developed within the MoDECS Project. Section 6.2 details the TDL compiler, its plug-in architecture, and current capabilities. In Section 6.3, we present the automatic code-generation process, the support of platform specific files, and component distribution. Following, we present practical aspects of the implementation of a portable TDL runtime system based on the Platform Abstraction Layer and the Virtual Machine from the previous two chapters. The resulted tool-chain is a complete end-to-end solution for the independent development of real-time components, which can be later integrated in various systems, with arbitrary topologies. The final section presents an evaluation of the overhead of the runtime environment on a real-time application and an analysis of the application behavior in various single-node and distributed systems.

The final chapter of the dissertation presents our conclusions regarding the development of portable real-time software components and the applicability of TDL and its runtime environment within existing real-time systems. The theoretical foundation presented in Chapters 3, 4, and 5 along with the implemented tool-chain from Chapter 6 constitute a sound foundation for future development, substantiated by excellent experimental results that corroborate our claims. We also present several future-research directions based on our work.
Chapter 2

Real-Time Systems

Time is a fundamental factor that defines the behavior of a real-time system (Buttazzo, 2002). In contrast with regular computational systems, the correctness of a real-time system does not depend only on its output values but also on the time when these are provided (Kopetz, 1997). The term real-time itself suggests that the computational system must react within a specific time interval; thus, the computation of a real-time system is time-bounded. The deadline of a computation in a real-time system represents the latest moment when the output values of the computation must be delivered.

As each real-time system has features and capabilities that favor different classes of applications, we begin this chapter with a classification of existing real-time systems through the safety, environment interaction, fault tolerance, topology, and reliability criteria. We continue with an analysis of their programming models synchronous, scheduled, and timed model, along with the advantages each model brings to the supported range of applications. Following, we present an overview of the approaches for portability in the form of microkernels and standards for full-featured real-time operating systems. For distributed real-time systems, in Section 2.4, we analyze the communication technologies that are crucial for the behavior of the overall system.

2.1 Classification

The wide range of existing real-time systems can be categorized following different criteria such as hard or soft real-time, event or time-triggered, fail-safe or fail-operational, single node or distributed systems. Often the real-time systems are encountered under the name of embedded systems, because of the role of the computing system in controlling a physical process and the integration of the two aspects of ”controlling” and ”controlled” into a common system (Burns and Wellings, 2001). Most examples of real-time systems are practically embedded systems and are present almost everywhere, from household electronics to power plants, cars, and airplanes. In the following subsections, we discuss these criteria and the characteristics of the corresponding real-time systems.
CHAPTER 2. REAL-TIME SYSTEMS

Hard vs. soft real-time systems

The difference between the hard and soft real-time system lies in the effects of a missed computation deadline on the environment. Timing constraints are regarded as critical properties of hard real-time system, and typically, the output values of a late computation are no longer useful. Failure to complete the computation prior to a deadline, that is a deadline miss, may lead to catastrophic effects on the controlled environment. Examples of such systems are found in avionics applications, automotive and industrial control systems, robots, and safety-critical systems. All require guarantees about the capability of the real-time system to meet all deadlines and produce correct results. Hence, in most cases the hard real-time systems are validated and certified.

In a soft real-time system, the computation deadlines are not critical but desirable. The results of a late computation are most of the time still useful and the deadline miss leads in general to a degradation of the provided quality of service and system performance. Examples of such applications are media streaming, electronic games, VoIP. Most soft real-time applications have other properties such as throughput, power consumption level, more important than the timing constraints.

Event-triggered vs. time-triggered systems

The interaction model with the controlled environment splits the real-time systems into two categories. The event-triggered systems react to environment initiated external stimuli. The change of some observable state of the environment triggers an appropriate computation of the real-time system. The typical implementation of this notification mechanism relies on interrupts that determine the release of a computational task. Event-driven systems are flexible, scalable, and in general have a low latency, meaning that they can react within a short time-span to changes in the environment. On the other hand, they are less deterministic when multiple changes in the environment trigger an avalanche of interrupts that may be impose a high system load and queuing of the interrupt handlers. More dynamic by nature, an event-based real-time system requires an online scheduler, which uses precious processing resources.

Time-triggered systems may be considered as special kind of event-triggered systems, where the only trigger is time, which means that there is only one clock-based interrupt. In comparison with the event-based systems, the time-triggered systems operate on environment-state information instead of state change events. Thus, the time-triggered systems are program driven, in the sense that the program determines the moments when it samples the state of the environment (often periodically) and the moments when it delivers the results. In comparison with event-based systems, the time-triggered systems introduce latency in their outputs, but they are predictable and may handle deterministically multiple changes in the environment, because their worst-case conditions can be analyzed at design time. Depending on the complexity of the system, a time-triggered system may use static schedules with a low-overhead dispatcher, or an online scheduler.
Fail-safe vs. fail-operational systems

Faults are inherent in any control application and their influence on the behavior of a real-time system groups them into two classes. A fail-safe real-time system brings the controlled object into a non-operational safe state (e.g., gracefully shutdown). Some applications may provide one uncommon but safe state (e.g., all street-traffic lights on red), whereas for others there may be multiple safe states (e.g., multi-phase industrial control systems). In such cases, the algorithms of the real-time system choose the best matching state with the current fault scenario.

A fail-operational system brings the controlled object into a safer but still operational mode, typically with degraded performance. For example, most electronic control units in cars support a so-called "limp-mode" where in the case of an engine failure, the performance of the engine is throttled down just enough to allow the car to reach the nearest repair shop. The control system of an airplane with one of the two engines broken may provide additional power for the working engine to bring the airplane safely to the ground.

It is important to note that these classes of fault-tolerance are related not only to the controlling real-time system but also to the controlled object itself. Not all controlled object have a safe state in any scenario, as may be the case of an airplane in-flight failure of the engines.

Single-node systems vs. distributed systems

Depending on the controlled object, a real-time system may require one or more processing nodes (i.e., a single-node or a distributed system). A single-node real-time system has one processing unit with all the peripherals such as sensors and actuators directly connected to it. Depending on the state of the environment perceived as input information through sensors, the processing unit computes some control laws and provides the results through actuators back to the environment. A distributed system is more complicated; it consists mainly of processing nodes and a communication system. Each node of the distributed system may have its own sensors and actuators or may process data recorded by another node. The communication system may consist of a simple bus or a complicated peer-to-peer interconnection graph.

There are many reasons for using distributed systems in most mature applications: dependability (fault-tolerance), scalability, localization. A distributed system may be more reliable than a single node system as through replication faults on a node may be corrected by the other nodes; thus, maintaining a high degree of dependability of the overall system \cite{Kopetz:1997}. Also a distributed system can be easier extended by adding more processing nodes to solve a computational-intensive job, in comparison with the case of a single node system where a more powerful processor may be too costly, require too much power, or simply unavailable. Performance, cost, and physical properties of the communication lines make localization of the processing node with the attached sensors and actuators a good design principle for real-time systems.
CHAPTER 2. REAL-TIME SYSTEMS

Determinism criteria

Depending on the input values and output values of a computational system, along with the natural progression of time we can categorize the computational systems into four categories: non-deterministic, value deterministic, time deterministic, and time and value deterministic.

We call a computational system non-deterministic with regards to its output values and time, as a system that provides for the same input always different unpredictable output values (e.g., random or pseudo-random) at unpredictable moments of time since the input was provided. Typically, this category includes the software for random number generation.

A computational system provides value determinism when for the same input values, it always provides the same output values; however, the time when it provides the output to the external environment may vary. Most functionality software can be included in this category provided that it works correctly.

Time deterministic systems provide output values always at an exact moment of time since the input was provided, regardless of their correctness. Such computational systems may provide incomplete or partial results; however, the output jitter is minimal.

A fully deterministic system is time and value deterministic, which means that for the same input values provided at always the same time the computational system provides always the same output values at always the same amount of time since the input was provided.

2.2 Programming models

The behavior of the real-time systems is greatly influenced by the underlying programming model used for their construction. The interaction between the system and the environment is governed by two different views over the notion of time.

The environment time represents the continuous time flow observable from the external environment of the computational system (i.e., wall-clock time). The time flow is unidirectional from present towards future and the real number representing its current value grows towards future. Its measurement unit is the standard second or its derivates such as millisecond or microsecond, always the same, regardless of the computational system (see upper part of Figure 2.1). All physical processes evolve in environment time, that is real-time.

The soft time is a discrete time flow of the computational system itself in relationship with the external continuous time flow. The discrete steps of the soft time rely on the number of occurrences of some events such as the pulses of the CPU clock; thus, the number representing its current value is an ever-increasing integer value (e.g., starting with the zero value at boot-time). The measurement unit for the soft time is not limited to the standard second, but can be derived from CPU clock ticks, communication-subsystem data rate, internal counters, and other software time-keeping mechanisms.
2.2. PROGRAMMING MODELS

(Kirsch, 2002) defines the mapping of environment time over the soft time as "the art of embedded programming". He identifies three real-time programming models used by a real-time computational system: synchronous, scheduled, and timed model. The Figure 2.2 illustrates these models. Only at those moments when the computational system interacts with the environment, the soft time becomes real-time.

2.2.1 Synchronous model

The synchronous programming model introduces the abstraction that the computational system performs all computation and communication instantaneously, meaning that the soft time is "zero". In this model, the environment triggers the execution of a synchronous computational process that provides an instantaneous output. As a synchronous system must react to every event from the environment, it works at the speed of the environment. The reaction of the computational system follows the synchrony hypothesis (Berry, 2000) and is atomic (i.e., all events occurring during a reaction are regarded as simultaneously).

The synchronous model was introduced to counteract the non-determinism of the classical sequential programming and concurrent formalism such as Petri-nets, and to aid the development of safety-critical systems in avionics and industrial control. The
abstraction of the model relies on the assumption that the computational system can always keep pace with the environment. However, its implication of an infinitely fast computer is not achievable in practice. Any practical implementation is thus just an approximation, which works when every reaction to an event occurs before the next event. The response time of such an implementation may still be far from “atomic” and present output jitter.

The Figure 2.3 presents this behavior, where two possible responses to the environment event have different execution times; thus, induce different response times and output jitter. The behavior of the system still obeys the synchrony assumption. However, it is difficult to prove that a synchronous program works as expected. A corresponding compiler for synchronous programming language must perform the verification of synchrony, reactivity, and determinism. It would require the maximum event occurrence frequency and the worst-case execution time of the code (derived from static code analysis). For reactivity, the compiler must prove the absence of infinite cycles. Determinism is related to the problem of causality, which is similar to deadlocks in asynchronous languages that happen when a signal or variable depends on itself. The compiler cannot check all such dependencies, as the developer may specify static cyclic dependencies, which still lead at run-time to deterministic programs without cycles.

Figure 2.3: Implementation of the synchronous programming model (Kirsch, 2002)

Existing synchronous languages can be classified under two categories: control-flow and data-flow oriented. The control-flow oriented languages are also imperative languages and are appropriate for control-intensive applications such as communication controllers, real-time process control. As examples, we mention Esterel (Berry, 2000), which has high-level, modular constructs that lead to a real structure of reactive programs based on the semantics of the finite-state Mealy machine; Statecharts (Harel, 1987), which has a graphical formalism and it is not fully synchronous; and ArgoS (Maraninchi, 1991), which simplifies the formalism of Statecharts and provides full synchrony.

The data-flow oriented languages (also known as declarative languages) are appropriate for data-intensive applications such as digital signal processing and steady

\[1\] Jitter - the average time-deviation between the moment when the output is provided, and the moment when it should be provided to the environment.
stream process-control applications. For example, Lustre [Halbwachs et al., 1991] is a declarative language that supports only the data-flow systems that can be implemented as bounded automata-like programs in the sense of Esterel.

There is active research in the area of synchronous languages to unify the two approaches to cope with applications that are more complex. The synchrony assumption of the model makes it particularly interesting for hardware design, as it is identical with the zero-delay model of electronic circuits. Therefore, most synchronous languages provide translation tools, which automatically generate code in C, Ada, or VHDL.

### 2.2.2 Scheduled model

The scheduled model relies on the classical scheduling theory for real-time programming. A real-time program consists of one or more computational jobs (typically named tasks or threads). The execution environment, typically named Real-Time Operating System (RTOS), schedules the computations on the processing unit of the platform. In other words, it determines which computation runs at which time and how long. The performance of the platform (i.e., CPU speed), the utilization level (i.e., the ratio between the required and available CPU time), and the scheduling scheme influence the response time (i.e., soft time) of a scheduled program. Thus, its soft time is no longer an abstract notion equal with zero, but an unpredictable run-time variable with an upper bound, namely the **deadline**, as depicted in the Figure 2.4. To cope with this problem, an off-line schedulability analysis is necessary to guarantee that all computations complete in time. This analysis is based on the computation of the worst-case execution time. In case of hard real-time tasks, meeting the deadlines is critical for the system behavior, whereas soft real-time tasks may miss some deadlines, with a resulting degraded performance.

![Figure 2.4: Implementation of the scheduled programming model (Kirsch, 2002)](image)

The scheduled model does not restrict the implementation language for the real-time program. Hence, a developer may use a classical sequential language, such as C/C++, or a parallel language, that is a real-time programming language, such as Ada, Occam, CSP, and Real-Time Java. For the sequential languages, a RTOS must be present to provide a set of synchronization primitives and handle the inter-program
communication. Parallel programming languages have native support for concurrency and communication, and require a corresponding run-time system, which typically performs the same tasks as a RTOS scheduler. The differences between the two approaches remain in the opportunity for programming errors and the possibility for automated code analysis. From the point of view of the compiler, the hard and soft computational activities are the same, they only need different scheduling schemes. An efficient run-time system handles them differently: it guarantees the deadlines for hard real-time computations and minimizes the average response-time for the soft real-time computations. Classical examples of languages based on the scheduled model are Ada \cite{Taft1995}, which provides an object-oriented core language with many domain-specific extensions available; and Real-Time Java (RTJ). \cite{Carnahan2000} presents the requirements of RTJ extensions, whereas \cite{Bollella2000} comes with a feasible implementation approach.

\subsection{Timed model}

In the timed model, the soft time is always equal with the environment time, such that all computations and communication activities take a fixed logical amount of time, regardless of their physical execution time. Whereas the model specifies that the environment receives the output of the computation at exactly the required response time, the actual implementation of the model has to delay the delivery of the output values in the case when the computation finished earlier.

The model also abstracts the scheduling scheme, as the physical execution pattern of the computation is not relevant provided that it meets its deadline. For any computation in this model, the environment determines the real-time constraints of the computation. The developer works with the assumption that there is enough soft time to perform the computation. The burden of matching the two soft time and real-time remains for the compiler that has to check the time-safety of the computation (i.e., there is enough soft time to complete the computation before its deadline). Thus, the compiler requires additional information about the worst-case execution time of the computation, the platform performance, and the scheduling scheme. At run-time, the execution environment has to verify these checks and schedule the computation according to an equivalent scheduling scheme. As a result of this abstraction, the timed-model ensures value and time determinism and predictability, provided that the platform offers enough resources (see Figure 2.5).

A core concept of this programming model is the \textit{Logical Execution Time} of a computational task. The main idea is that the lifetime of a task has two logical environment-determined moments: a release time and a terminate time. At the release event the task reads its input, whereas at the terminate event it provides its outputs to the environment. Within this time-span, the way the task executes on the target platform is irrelevant for the environment. We detail this concept in Section 3.1.
2.3. Real-Time Environments

There are just a few examples of languages based on this model, most of them derived from Giotto (Henzinger et al., 2003). Giotto is a high-level time-triggered language, which decouples the timing and functionality aspects, and abstracts from the execution platform. As a meta-language, it describes the intended temporal behavior of a system and expects its functionality as being externally implemented in a general-purpose programming language such as C, Oberon, and Java. Thus, it expresses the reactivity of the program in respect to the environment, regardless of the physical implementation aspects and platform details such as CPU scheduling scheme.

XGiotto (Ghosal et al., 2004) introduces an extension of the Giotto development methodology to support event-driven programming. In comparison with other event-oriented languages, the XGiotto approach preserves the benefits of the timed-model with fixed response-time.

Our approach, namely the Timing Definition Language (Templ, 2004) presented in detail in Chapter 3 is also based on the timed model and a derivate of the Giotto development methodology. It introduces new high-level constructs for structuring complex application into components, an improved mode-change protocol, simplified syntax, full support for distribution, and a corresponding tool-chain.

2.3 Real-Time Environments

The behavior of a real-time system and its portability aspects are highly influenced by the underlying real-time environment. Most hard real-time systems employ dedicated software, the so-called real-time operating system (RTOS). The purpose of such a real-time operating system is to provide an optimum allocation of the available physical resources to the software processes composing the real-time application.

A real-time operating system provides in general the following services:

- **Multi-tasking**, to support the execution of multiple computational jobs called *tasks*. This service is present in most operating systems, as a single task system may arguably require a dedicated operating system. Note that preemption is not always required for multi-tasking.
• **preemption**, increases the potential of the multi-tasking system, allowing the RTOS to temporarily-suspend the execution of a task, and execute another task, and improve the system response times. Through preemption, the processor of the system is multiplexed to the computational tasks; thus, the tasks run logically in parallel.

• **priorities or deadlines** or other mechanism for specifying the run-time characteristics of the tasks. Depending on the scheduling scheme of the RTOS, the number of priority levels should be sufficient for the range of supported applications. Deadlines and other run-time information per task are not available in most current RTOS. For systems using priority-based scheduling, a priority inversion may occur when a higher priority task must wait on a lower priority task to unlock a common resource and in turn, the lower priority task is waiting upon a medium priority task (unrelated to the locked resource).

There are two available solutions: the priority inheritance and priority-ceiling protocols, which both need sufficient priority levels. When using the priority inheritance mechanism, a task blocking a higher-priority task inherits its higher priority for the entire duration of the blocked task. In the case of the priority ceiling protocol, a priority is associated with each resource, and the lowest resource priority is higher than the highest task priority. At run-time, the scheduler of the RTOS makes the priority of a task accessing a resource equal to that of the resource, except for the case when the task is waiting on another resource. After the task releases the resource, its priority returns to its original value.

• **synchronization** for user tasks (or threads). Complex applications require mutual exclusion or synchronization between its concurrent parts. The RTOS must provide inter-task communication and synchronization primitives. Locking resources temporarily for exclusive access is commonplace in many applications. Thus, the system must provide the mechanisms for delaying the execution of other tasks contending for a locked resource, until that resource is available.

• **bounded latencies** for the internal operations of the RTOS, mainly task switching, interrupt switching, and dispatching [Heath 2002]. The time spent for creating a new task context or restoring the context of the next executing task represents the dispatching latency of the RTOS. The task switching latency includes the dispatch latency and represents the time the RTOS spent saving the context of a currently executing task and switching to another task from its run queue. The context of a task typically contains the registers of the processor, stack pointer, and other task related information.

Handling interrupts requires several steps: preempting a running task, executing the interrupt handler, and resuming the task or executing another task. Hence, the interrupt latency represents the time elapsed between the execution of the last instruction of the interrupted task and the first instruction of the interrupt
2.3. REAL-TIME ENVIRONMENTS

handler. The interrupt dispatch latency represents the time to switch back from
the last instruction of the interrupt handler to the preempted task or another
task scheduled to run.

There are additional constraints from the target of these real-time operating sys-
tems, namely error recovery, minimal latencies, small memory footprint, and low CPU
overhead. Depending on the capabilities of the target platform and the number of
services provided, the RTOS can be classified into two categories: microkernels and
macrokernels (i.e., full featured). The complex full-featured real-time systems may
also be categorized under monolithic and modular, depending on the level of inclusion
of their services into the kernel of the RTOS.

2.3.1 Microkernels

For real-time systems, the traditional approach of custom designing the applications
on top of the bare hardware cannot cope with increasing complexity of code and pre-
dictability requirements. A conventional RTOS even modular requires several megabytes
of memory and powerful processors, which is not always available because of power,
weight, heat, environment, or cost reasons. One approach followed by RTOS vendors
is to provide a stripped-down or customized version of a conventional RTOS. However,
maintaining more versions for the same RTOS can be expensive and translates into
higher licensing costs to the developer.

Alternatives via real-time microkernels represent a feasible way to support such
low-end embedded systems. Microkernels are a special category of RTOS providing a
reduced set of optimized services to applications running on typical low cost and perfor-
mance hardware. In contrast with general-purpose microkernels such as Mach (Accetta
et al., 1986) and SPIN (Bershad et al., 1995), where performance metrics relate mostly
to throughput, adaptability and flexibility of the operating system, their real-time coun-
terparts have stricter requirements. In addition to a small memory footprint, a typical
real-time microkernel must provide low-overhead system services, through either highly
optimized code or special designed schedulers and OS primitives.

There are several approaches for designing such real-time microkernels, depending
on the supported hardware families and their capabilities. For example, to support
efficiently the process-based computations, the real-time microkernels employ several
techniques to optimize the inter-process communication (Liedtke, 1993). For micro-
kernels supporting threads, the thread management mechanisms are important for the
overall performance of the system. Several approaches touch different aspects of thread-
performance improvements, such as specialized interfaces for scheduler activations (An-
derson et al., 1991) or reduced thread stack through continuation (Draves et al., 1991).
Support or optimizations on virtual memory (Rashid et al., 1988) are not required
for the majority of microkernels as memory is a premium in most embedded system.
In the following, we analyze several microkernels resulted from academic research or
commercial development.
CHAPTER 2. REAL-TIME SYSTEMS

Academic microkernels

Most academic microkernels are experimental by nature; hence, target specific platforms and support a limited set of applications. Some of them provide compatibility with the basic services of the RT-POSIX standard (except for the real-time file system), which make them interesting for research purposes in cross-platform portability of real-time applications. From the large number of implementations available, we focus on most popular/mature approaches and analyze their capabilities.

Emeralds (Zuberi et al., 1999) targets real-time embedded applications with limited resources, for example, micro-controllers operating at less than 50MHz and with tens to hundreds of kilobytes of memory. It implements a hybrid scheduler with three queues using Earliest Deadline First (Liu and Layland, 1973) scheduling for the first two queues containing high frequency tasks, and Rate Monotonic (Liu and Layland, 1973) scheduling (32 bit non-unique priority levels) for the third queue.

The microkernel is entirely reentrant and without internal threads. It is mapped into every user space; hence, its simple call-oriented API has a minimal overhead on the user to kernel-mode transition. Moreover, it implements a context-switching optimization on the semaphoring mechanism relying on the premise that most object-oriented languages require fast synchronization primitives (Ishikawa et al., 1992) such as semaphores (Dijkstra, 1965; Habermann, 1972), binary mutual-exclusive locks (mutextes), and monitors (Hoare, 1974).

EMERALDS uses compile-time source code analysis to introduce additional signaling code into user applications and task swapping in the run-queue to reduce the probability of resource priority inheritance (Sha et al., 1990). For message passing between tasks, it uses state messages (Kopetz and Reisinger, 1993; Kopetz et al., 1989) implemented through global variables.

Spring kernel (Stankovic and Ramamritham, 1989) was designed for distributed systems with globally replicated memory. It implements a dynamic scheduling mechanism for non-critical tasks based on execution time and resource constraints, and a static table-based scheduling mechanism for safety-critical tasks. Thus, it improves fault-tolerance and performance on overload conditions. Using a system description language, the developers can group processes with common goals into "process groups", which may have timing and precedence constraints (M. Teo, 1995). Spring abstracts the reservation, planning, and end-to-end timing support of inter-task communications, and allows real-time synchronous and asynchronous multicast communications in groups.

Harmony (Gentleman, 1988) is a real-time multiprocessor RTOS based on a microkernel architecture and a set of system servers. It provides preemptive priority-based scheduling for tasks (with fixed priorities for the lifetime of a task), dynamic task allocation, interrupt management, inter-task communication through messages, local
exclusion mechanism through interrupt masking, and support for cached/non-cached memory management. For multiprocessor support, it requires homogeneous processors boards.

**RT-Mach** (**Tokuda et al., 1990**) it is a resource based micro-kernel with priority-based preemptive scheduling (**Tokuda and Nakajima, 1991**), and support for quality of service. It correlates an application’s request for a resource with the global real-time requirements and checks whether the requirements still hold, it reserves and allocates the resource for to that application. The implementation requires an additional QoS middleware, and an application awareness of this resource reservation mechanism; thus, limiting the range of portability from other operating systems.

**Commercial microkernels**

Commercial microkernels target two markets for popular embedded devices (small range of platforms such as WindowsCE) or specialized applications (e.g., pSOS). However, there are also customizable versions of full-featured real-time operating systems such as QNX, which can be adapted to application specific needs and still have a large compatibility base. In general, these systems employ conformance classes or additional standards to define the level of compatibility between applications.

**pSOS** (**Thompson, 1990**) is an object-oriented RTOS, where objects can be tasks, memory regions, message queues, and semaphores. For task scheduling, it can use EDH\(^2\) or preemptive priority-driven scheduling with both priority-inheritance and priority-ceiling protocols. In addition, user-tasks may run in supervisory mode. The developer has full control over the interrupt handling, and device drivers may be dynamically loaded. For memory allocation, the RTOS uses the concept of memory regions that are physically contiguous blocks of memory created in response to application calls. The memory regions allocated to tasks may be local or global.

**Windows CE** (**Muench, 2000**) is a derivate of the standard desktop Windows platform targeted for embedded devices. It is a modular RTOS, for 32-bit mobile devices with small memory (at least 4MB). The kernel provides real-time threads with a maximum of 256 priority levels, all running in kernel mode. It has several performance-oriented optimizations for the execution of non-preemptable code: moving of kernel data into physical memory to avoid translation look-aside buffer misses, and partitioning of kernel into small sections. In addition, it allocates all processes, threads, mutexes, events, and semaphores in virtual memory. For portability, it introduces an Equipment Adaptation Layer to isolate the device-dependent code, with authentication support to prevent unauthorized applications from accessing system APIs.

\(^{2}\text{Earliest Deadline First}\)
CHAPTER 2. REAL-TIME SYSTEMS

2.3.2 Full featured Real-Time Operating Systems

The vast majority of complete real-time operating systems center on a small set of standards. This ensures compatibility and portability between similar implementations of the same operating concepts. The standards in general focus a particular industry field such as automotive, avionics, or telecommunications, and offer facilities for the development of characteristic applications for that field. Nevertheless, there are RTOS for special application domains, which provide proprietary interfaces and eventually wrappers for standard conformance (e.g., VxWorks).

Depending on the capabilities of the supported hardware, the real-time operating systems may provide at run-time dynamic loading mechanisms for user functionality, memory protection, soft and hard real-time task facilities, dynamic scheduling. Others, for low powered devices may offer just the basic controls of timers and task contexts. In the following, we present several popular standards and discuss their applicability.

OSEK Standard

OSEK/VDX (OSEK Group, 2005) is an industry standard for an open-ended architecture for distributed control units in vehicles. It describes the operating concepts for a family of multi-tasking capable RTOS targeted at the automotive domain, with tight real-time constraints and high criticality. The driving ideas behind the standard are scalability, portability, configurability, and static allocation of software components. Any application based on this standard belongs to one of its four conformity classes.

- BCC1 - only basic tasks with one activation request per task, one task per priority level (each task has a different priority, out of minimum 8 possible levels)
- BCC2 - in addition to BCC1, tasks may have more activation requests and may share priority levels
- ECC1 - includes BCC1, adds support for extended tasks with waiting capabilities, and requires at least 16 priorities levels
- ECC2 - includes all other conformance classes

The OSEK standard specifies two types of tasks: basic and extended, modeled as three (or four) state automatons. The extended tasks have the option to wait for an event before being scheduled for execution. The Figure 2.6 presents their possible run-time states, and the corresponding run-time control mechanisms.

For portability reasons, the standard supports only ANSI-C based applications and does not specify any I/O interfaces. It also defines an integration language OSEK
2.3. REAL-TIME ENVIRONMENTS

Implementation Language (OIL) to help the developer configure and fine-tune a compliant RTOS implementation to the requirements of the application. In contrast with most other standards, OSEK uses static allocation of software components, which also simplifies the implementation of a compatible RTOS. All tasks and their exact allocation at run-time must be available at compile time; thus, the application with the RTOS can be embedded into the ROM of the electronic controlling unit (ECU). For cost reasons, the standard requires a small memory footprint implementation and low CPU utilization. At run-time, the RTOS uses fixed-priority scheduling for user tasks and interrupt service routines (ISR). Although it defines application modes, an application may decide only at boot time its operating mode (through a reduced set of I/O operations). During run-time, it does not support any mode switches.

OSEKtime [OSEK Group 2001] is an extension of the OSEK standard for distributed time-triggered applications. It aims at modular development of applications with fault-tolerance support and full compatibility with OSEK/VDX. It supports static task allocation, task scheduling with deadline monitoring, clock synchronization in distributed setups, interrupt handling (ISR), local message exchange between tasks, error detection, and fault tolerance.

OSEKtime allows the execution of another OSEK compatible RTOS as a guest operating system. For this purpose it uses the remaining idle time from the time-triggered execution of OSEKtime tasks. Consequently, non-preemptable OSEK tasks may be preempted only by OSEKtime tasks. For dependable applications, the underlying hardware must provide memory protection between the two operating systems. The standard includes support for application modes, with dynamic mode switches at run-time. However, the modes must have the same period and the dispatcher round cannot be changed. Therefore, it is impossible to have multiple applications on the same controlling unit, each switching independently its operational mode (the number of dispatch tables required to capture all possible mode combinations would be extremely large, and the functionality of each application would be severely limited).
APEX Standard

Because of the inefficient resource utilization of the classical federation approach for the development of avionics applications, ARINC developed a set of standards for Integrated Modular Avionics (IMA) applications, which are distributed multiprocessor architectures with shared memory and network communications. One of the standards, APEX (APEX) defines the interfaces of an operating system for IMA, and its implementation is intended to act as a system layer between the application and the underlying operating system. For a portable interaction with the physical hardware, it relies on a standard interface named COEX, sitting on top of a Hardware Interface System (HWIS).

The APEX standard divides the physical memory into partitions and allocates software sub-systems to different partitions. The partitions contain processes that may communicate with each other via buffers, semaphores, or events. Processes have properties such as period, time capacity, base and current priority level, running state. The processes of one partition may use message passing over logical ports and physical channels to communicate across partitions with other processes. A partition is the unit of distribution of the system, and has several properties such as criticality level, period, duration, preemption level. APEX uses a cyclic off-line generated schedule for partitions ensuring temporal isolation between them. Within partitions, it uses fixed-priority scheduling for the contained processes. Practical implementations of APEX compatible applications are already available in the complex systems of Boeing 777.

µITRON Standard

The Japanese µITRON Project introduced a series of standards for real-time kernels since 1984. It covers 8-32bit embedded devices for multimedia equipment, household electronics, PC peripherals, office and communication equipment, and many others. It defines a strict Standard Profile, which maximizes portability without affecting its scalability, for 16/32bit processors with several KB of memory. The profile supports priority-based scheduling with both priority inheritance and priority ceiling protocols (Takada and Sakamura, 1994), semaphores, message queues. However, it does not offers protection between the kernel and application space, all system is one executable image, which goes into the flash ROM of the embedded device. On the Japan’s market, there are more than 50 RTOS based on this standard with support for around 35 processors.

RT-POSIX Standard

The IEEE Portable Operating System Interface for Computer Environments, (POSIX) standard (IEEE Information Technology, 2006) was designed to allow the development of portable applications. Its highly successful real-time version, namely RT-POSIX, has three parts: the first - Base Standards - dedicated to the interfaces to the RTOS services in the C language (POSIX, 2003a); the second - Profiles - dedicated to the
subsets of services for different embedded applications; the third, for bindings to OS services in other programming languages, such as Ada, Fortran (POSIX, 2003b).

The RT-POSIX standard introduces a set of requirements for a compliant RTOS in handling the input/output (I/O) operations, shared memory access, resource management, real-time threads, scheduling of multiple tasks, inter-process communication and high-resolution timers:

- **I/O operations** are highly used in control applications. The controlled object provides information through sensors and receives control through actuators. Both sensors and actuators require I/O operations. Traditional systems use a **synchronous** approach, where a controlling task that issues an I/O function call, waits until its completion. Depending on the type of I/O operation and the physical platform, the RTOS may have the CPU free to execute another task until the I/O operation completes. Another approach, which promises improved performance requires the controlling-task ability to receive notification upon the completion of the I/O operation. Hence, it may continue processing in parallel with the I/O operation (IEEE Information Technology, 2006).

- **Memory** related operations are also frequent in any application. The standard requires the ability to map the common physical-memory space into independent process-specific virtual space. This is important to allow memory protection between processes, such that a failed process does not affect other processes. In addition, to free the main memory the system may move unused parts of an application to secondary storage (i.e., virtual memory).

- **Semaphores** support for resource sharing. The standard enables the concurrent access to common resources such as memory locations of variables storing application states, require locking mechanisms, through semaphores (Silberschatz et al., 2001). There can be binary semaphores, which provide mutual exclusion between running tasks (i.e., mutexes), or multiple value semaphores, which may denote the locking level of a resource or other events.

- **Real-time threads** for parallel processing capabilities within processes. The real-time threads have additional timeline constraints (Liu, 2000); thus, the RTOS scheduler must handle them in a similar manner with other computational tasks. Threads may be scheduled as lightweight processes, or independently within the time-quantum allocated to the parent process.

- **Scheduling** of computational units (i.e., processes, tasks, and threads). This is a crucial requirement implied by the basic requirement of multi-tasking. Common scheduling methods implemented by available RTOS are round robin and priority-based preemptive scheduling. The duration of a round-robin time slot, and the number of priority levels may vary between implementations (Ramamritham and Stancovic, 1994).
• inter-process communication is important for complex applications composed of several processes. The RTOS must provide the mechanisms for a deterministic information exchange between concurrent processes. The standard requires support for mailboxes and queues (POSIX, 2003b).

• timers of high-resolution are important for the behavior of the real-time system. The applications must be able to measure time intervals and plan future actions through the RTOS timer services (Liu, 2000).

• in addition the standard includes support for real-time files, which represents the ability to create and access files with deterministic performance. A precondition for this ability is the existence of a real-time file system implemented by the RTOS.

For implementations on a wide range of systems, the standard defines a set of four profiles (POSIX, 2003b) with increasing number of features and complexity. Thus, each RTOS may implement the part of the standard more adequate for its target application domain, and still be compatible with other RTOS from upper profiles. The four profiles are:

• Minimal Real-Time System profile (PSE51) requires just threading and predefined devices support. Thus, most of the complexity of process handling and file system support are avoided, resulting minimalist RTOS with memory footprints in the range of tens of kilobytes.

• Real-Time Controller profile (PSE52) adds support for a simplified file system for data storage.

• Dedicated Real-Time System profile (PSE53) targets large embedded systems with support for multiple processes (e.g., in avionics applications).

• Multi-purpose Real-Time System profile (PSE54) targets complex systems with a mixture of real-time and non-real-time requirements. This profile includes most POSIX functionality and real-time services.

In the following paragraphs, we present several RT-POSIX compliant real-time operating systems targeted at various application domains. Whereas the core of an application developed for any of them may be easily portable to another RTOS, the device drivers are highly platform dependent and have no real benefit from the POSIX standard.

VxWorks (Wind River Systems, 1993) is a widely used RTOS with almost two thousands APIs and support for a vast set of platforms. As basic services, it provides preemptive priority-based (256 priority levels) and round robin scheduling with deterministic context switching, semaphores and mutual exclusion with inheritance,
private virtual memory for tasks, file system and I/O management. Its networking stack supports full TCP/IP, and UDP sockets and a range of standard services fully customizable. For portability, it provides a Board Support Package that interfaces with the hardware-dependent layer. As separate services, it offers multiprocessing, memory management unit, graphics, and even Java support.

A particularization of the RTOS for the APEX standard, namely VxWorks AE, introduces “protection domains” that correspond to the APEX concept of partitions. A protection domain contains all memory-related resources such as tasks, queues, semaphores. For the temporal domain protection, an optional ARINC-653 compatible scheduler is available. Thus, it provides a two-level scheduling mechanism, which protects tasks from a protection domain from the failures of tasks from other protection domain. Within the protection domain, it uses priority-based preemptive scheduling. This version of VxWorks AE is available only on a small set of platforms.

**Integrity** (Green Hills, 2006) targets embedded systems, which require maximum reliability. Thus, it follows the same design goals of VxWorks AE with partitions. It uses the hardware memory-management unit to isolate the partitions in separate memory spaces. For scheduling, it uses the same two-level scheduling approach, and each task has assigned a fixed CPU budget. For mutual exclusion synchronization, it uses the highest-locker protocol to avoid the priority-inversion problems. As a result of clear separation in memory and CPU time between partitions and their constituent tasks, it provides several fault-tolerance guarantees.

**LynxOS** (Lynxworks, 2006) is a multi-threaded RTOS for complex real-time applications. For task synchronization, the developer may use prioritized FIFO, dynamic deadline monotonic scheduling with 512 priority-levels, or time slicing. The RTOS provides task synchronization primitives, TCP/IP communication with sockets, lightweight service modules, file systems, memory protection (must be supported by hardware) between running tasks, demand memory paging, etc. For interrupt handling, the RTOS uses a kernel thread that can be prioritized and scheduled similar to other threads for a predictable response even under heavy I/O.

**QNX** (Hildebrand, 1992) is a micro-kernel based real-time system, which can be customized for resource-limited devices with minimal functionality or high-performance platforms. It supports processes and threads with 64 priority levels, synchronization through message passing, round robin and dynamic priority-based scheduling, priority inversion protection through priority inheritance mechanism, and fast interrupts handling. It relies on the hardware capabilities of the memory management unit of the platform to run each application in its own memory space. For powerful platforms, it provides GUI capabilities, TCP/IP networking, web-server with CGI, local and networked file-system support, remote management functionality, and SSL encryption.
CHAPTER 2. REAL-TIME SYSTEMS

RTEMS \cite{Acuff and O’Guin 1996} is the acronym for Real-Time Executive for Multiprocessor Systems, a general-purpose real-time system developed initially for the U.S. Army and later released as open-source under GNU General Public License (GPL). Along with POSIX compliance, it also provides specialized interfaces, which enable enhancements and further development of the RTOS \cite{Colin and Puaut 2001}. It supports full UNIX services, and uses free software development tools such as GCC, GDB. Currently it has ports for a wide range of mid-to-high performance platforms. There are more than 50 Board Support Packages for embedded designs based on the supported processors, which operate through a generic driver library \texttt{libchip} for additional portability. In addition, a GNAT compilation system exists for the Ada language that enables cross-platform embedded development.

InTime

The InTime \cite{Tenasys 2005} RTOS adds real-time capabilities to standard Windows desktop platforms through a modified Hardware Abstraction Layer (HAL). Although it provides similar capabilities with a POSIX compliant RTOS, the InTime kernel is not RT-POSIX compliant at the API level. As a successor of iRMX, it relies on an Operating System Encapsulation Mechanism (OSEM) and the Intel x86 hardware memory protection to ensure the separation of the real-time kernel and the Windows kernel into distinct hardware tasks and memory spaces. Thus, the real-time applications run unaffected by the Windows environment, and may even continue running after a windows crash. However, it runs only on Intel Pentium family of processors (no other x86 compatible), and a restart of the Windows part cannot be performed without a full system reboot. Applications running concurrently within the two operating systems may exchange information through NX-extension interface (NTX). For the real-time applications, the InTime RTOS provides full process and threading support, preemptive round-robin and priority-based scheduling, 256 priority levels for real-time processes, message passing through mailboxes, semaphores, alarms, interrupt management. It also supports by default TCP/IP networking for a small set of network adapters. The non-real-time Windows kernel and its applications are executed with the lowest priority, fully preemptable by other real-time jobs.

Linux based

Following the high success and availability of the open-source Linux operating system, there is significant work towards its transformation into a powerful RTOS. There are two general approaches to provide real-time capabilities to the Linux kernel: first, use a lightweight real-time execution environment (micro-kernel) as a based and execute the Linux kernel within one of its threads, and second, modify the Linux kernel itself to enhance its capabilities.
2.4. COMMUNICATION IN DISTRIBUTED SYSTEMS

RT-Linux (FSM Labs 2006) is one of the real-time enhancements available for the Linux kernel. Developed at the University of Mexico, it employs a small execution environment, which executes Linux, and all non-realtime applications as a background thread. It provides alternate interrupt handling mechanisms, which overtake the Linux kernel for real-time activities. All non-realtime interrupts are queued and later deferred to the Linux kernel when there is no real-time activity in progress. In this way, the interaction between the Linux kernel and the real-time part is minimized. There are several drawbacks of this approach: firstly, real-time threads work in Linux kernel address space; thus, any error crashes the Linux OS. Secondly, it is often necessary to rewrite the device drivers for real-time applications (e.g., for real-time networking capabilities). Thirdly, the real-time scheduler is a simple POSIX compliant fixed priority scheduler without any resource management facilities, which are important for low powered and mobile devices. Fourthly, disabling interrupts in device drivers increases the latency of the real-time threads.

RTAI (DIAPM 2006) is developed at DIAPM in Italy starting from the same general idea as RT-Linux. Nevertheless, it introduces several enhancements for soft and hard real-time processing by integrating RTAI tasks, Linux kernel threads and user processes. It also offers a lightweight middleware layer based on the remote-procedure call concept, which can be dynamically extended for distributed systems. An alternate version of RTAI targets a minimalist kernel, namely Adeos, which enables addition of standard open middleware layers, and other COTS software applications with minimal overhead.

Linux/RK (Rajkumar et al. 1998) is a modified version of the Linux kernel for real-time applications. It provides a transparent resource reservation mechanism for user processes. In also introduces advanced APIs for quality of service management through reservation.

2.4 Communication in distributed systems

The behavior of the real-time distributed systems is greatly influenced by their communication infrastructure in addition to the regular constraints of single-node systems. The ability to guarantee the time constraints of the messages exchanged by the node of the system is fundamental to the determinism of the distributed system. The communication infrastructure relies on specific Media Access Control (MAC) protocols to exchange information.

Cost and wiring reasons typically prohibit the one-to-one connections between nodes. Hence, the communication channel is visible as a shared resource among the nodes. In the general case, the applications running on top of the distributed system exchange information through messages, which may be fragmented into one or more data frames for transmission. A data-frame is the indivisible unit of data transmission on the communication medium. As the communication-channels have limited bandwidth and
CHAPTER 2. REAL-TIME SYSTEMS

latency, the messages from one application to another take time (i.e., end-to-end delay) that is orders of magnitude higher than the exchange of similar values on the same processor. The queuing and delivery mechanisms of the communication controllers of the nodes also influence this delay. We analyze two classes of protocols relevant for the following chapters: event-triggered and time-triggered protocols.

2.4.1 Event-triggered protocols

An event-triggered MAC protocols is primarily responsible for resolving contention between two or more nodes that simultaneously intend to transmit a data-frame. There are several methods possible: probabilistic, token, and priority-based. The classical Ethernet technology uses the probabilistic approach with a Carrier Sense Multiple Access - Collision Avoidance (CSMA-CA) scheme. Any node listens to the medium before transmitting and during transmission. When two or more transmitting nodes detect a collision, each node waits a random interval before trying to send the frame again. The token mechanism, involves a special message called token, being passed among the nodes in the system to denote the node that can send at a time. The priority-based mechanism assigns to each node or each message (depending on the protocol) a priority. This priority determines which node may transmit prior to the actual transmission of each data frame. Other nodes intending to send data at the same moment wait until the frame transmission completes. One of the most popular communication event-based protocols is CAN, presented in the next paragraphs.

CAN

The Controller Area Network (CAN) protocol was introduced by [Bosch, 1991] and later ratified as an automotive standard by [ISO Standard 11898, 1993]. The protocol operates on a two-wire bus with a multi-master architecture, and provides variable payloads of up to eight bytes and a maximum data rate of 1Mbit/sec. It defines special properties that are used to provide great flexibility, reliability and determinism. First requires that the signal representing a single bit must be available to all nodes simultaneously. The duration of the transmission of a single bit (the bit-time) at 1Mbits/second is 1us; thus, the propagation delay in the wires limits the bus length to less than 100m (slower data rates increase the bus length). This property can be used to perform clock synchronization between nodes (with time stamping through the CAN processor).

<table>
<thead>
<tr>
<th>Bits exposed to bit-stuffing (34 control bits and 0-8 bytes of data =&gt; 34-98 bits)</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arbitration field</td>
<td>Control field</td>
<td>Data field</td>
<td>CRC field</td>
<td></td>
</tr>
<tr>
<td>11-bit Identifier</td>
<td>4-bit DLC</td>
<td>0-8 bytes</td>
<td>15 bit CRC</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.7: The structure of a CAN frame
2.4. COMMUNICATION IN DISTRIBUTED SYSTEMS

Each CAN frame begins with an 11-bit frame identifier that serves dual purposes as address and priority as depicted in Figure 2.7. For systems with a fixed message sets, the frame identifier can be used to identify the type, source, and the destination of a message within a theoretically possible $2^{11}$ (optionally $2^{29}$ with extended 29 bit identifier) unique messages (less in practice due to control frames).

![Figure 2.8: The arbitration mechanism of CAN](image)

The CAN protocol employs a Collision Sense Multiple Access - Collision Avoidance (CSMA-CA) media arbitration using the frame identifier to ensure that from a set of pending data frames the one with the highest priority is always transmitted first. All nodes actively listen on the bus after sending each bit of the frame identifier, hence, implementing the carrier-sense mechanism. Contention occurs when two or more nodes start transmitting simultaneously, because of waiting the completion of a previous frame transmission. Using a wired-AND operation on the bus, with the logical "0" always overriding the logical "1", the CAN protocol performs a non-destructive bitwise arbitration on the frame identifier. Hence, any node sending a "1" stops upon detecting the concurrently "0" sent by another node, and waits until the transmission is completed. The Figure 2.8 illustrates this collision-avoidance mechanism. When a node does not detect any activity on the bus, it may freely send its frame.

The global and unique priorities of the CAN frames make the classical scheduling algorithms RM (Tindell et al., 1994) and EDF (Tindell et al., 1995) applicable for scheduling the communication within a CAN network, and provide an upper bound to the value of the worst-case response time of a message. However, the bit-stuffing control feature of CAN complicates the estimation of this value. The CAN protocol reserves the sequence of six identical bits (111111 and 000000) for control and error detection. Any data frame containing such bits, would have inserted after its first five identical bits another bit of opposite type. The receiving node would reverse this procedure to retrieve the content of the data frame. The result is that the size of the frame as specified when sending is not the same as the size of the bit stream actually being transmitted on the CAN bus; thus, the estimation of its transmission time depends on the actual content of the data frame.
2.4.2 Time-triggered protocols

The time-triggered protocols employ a Time Division Multiple Access (TDMA) policy for the Medium Access Control (MAC). The nodes connected through a TDMA-based network communicate periodically within the so-called communication-cycles. Each communication period is divided into a number of time slots. Depending on the communication protocol, the time slots may be equal sized or may have different sizes. Each node in the network has pre-allocated a number of time slots for sending its messages. In all other slots, it must not send any message to prevent collisions. Typically, the network remains idle for the duration of a slot when the node assigned to that slot has nothing to send. All TDMA protocols require a common notion of time, which can be achieved either through a common time base or synchronization message from a master node, or a global clock-synchronization algorithm. In addition, all nodes must know at least the start times for their slots, and when the relevant received packets are available. Most systems provide the full communication schedule to each node. The assignment of messages to slots is protocol dependent and may not be statically defined.

For an improved utilization of the communication medium in the case when not all slots are occupied at run-time, the Flexible TDMA protocols allocate a time slot to a node only when the node has messages to send, otherwise another node may take the slot. This introduces additional jitter, and complicates the protocol, but is more flexible and does not require the full schedule on each node (e.g., see FlexRay presented below). By allocating slots for asynchronous messages, most time-triggered protocols can accommodate event-based protocols on top of them. Presented below as examples are TTCAN, TTP/C+CAN, FlexRay.

TTCAN

The standard CAN ([ISO Standard 11898 (1993)] protocol has proven a reliable protocol for automotive industry. However, its powerful bitwise arbitration mechanism has an unwanted drawback. The arbitration guarantees the transmission of a high-priority message without destroying it even when other nodes try to access the bus. Nevertheless, in the case when some other message is already in the process of transmission or another message has a higher priority, the high-priority message is inevitably delayed. Consequently, even the temporal behavior of the message with the highest priority may show a small latency, depending on the network traffic.

The Time-Triggered CAN (TTCAN) protocol [Führer et al., 2000] was introduced as extension of CAN to avoid this problem. It supports time-triggered operations in addition to the priority-based event-triggered message handling. The protocol defines a periodic message exchange between nodes. It relies on a precompiled schedule for the time-triggered communication, namely the system matrix, sized accordingly to the least common multiple of the periods of all messages. All nodes connected to the TTCAN bus have a localized copy of the system matrix, which consists of a number of basic cycles, not-necessarily identical (see Figure 2.9). The number of basic cycles is appli-
2.4. COMMUNICATION IN DISTRIBUTED SYSTEMS

- Basic cycle 1: Message A, Message B, Arbitration Window, free, Message C, Message D
- Basic cycle 5: Message A, Message B, Message F, Message D, Message C, free

Figure 2.9: TTCAN System Matrix and traffic pattern

cation and configuration dependent, and the developer may specify the communication pattern for each node (e.g., every cycle, every second cycle, once per system matrix, etc). The protocol specifies that each basic cycle starts with a reference message for synchronization, and contains a number of time windows of different types: exclusive - for the transmission of statically scheduled messages, arbitration - for the classical CAN communication, and free - reserved for future usage. Within an arbitrating time window the TTCAN protocol uses the standard CANs native non-destructive bit-wise arbitration mechanism to determine the sequence of arbitrary messages from nodes (e.g., debugging information, non-critical events). Two or more arbitrating windows may be merged to allow a larger time interval for the transfer of arbitrary messages (standard CAN event-based mechanism).

Depending on the time source used for the time-triggered communication it defines two operational levels: based on the reference message of a time master (i.e., fault-tolerance through multiple masters) and with a global synchronized time base (i.e., continuous drift correction). The Network Time Unit (NTU) is the base for the time-triggered sending of messages within specific time slots, and is the same for all nodes. It is derived from the CAN bit time (with level 1), or the physical second (with level 2). Every node derives its local time from the period of its internal clock and the duration of the NTU using a local Time Unit Ratio ($NTU = TUR \cdot t_{sys}$). The reference message from the beginning of each basic cycle limits the allowed clock drifts of the nodes attached to the TTCAN network. With the alignment of the messages at NTU intervals, the classical bus arbitration is avoided and the latency time and message

35
delivery jitter becomes predictable. The implicit atomic broadcast property of the protocol provides a membership service at the application level.

For error handling, the protocol has an "all or nothing" policy, meaning that a message is either received correctly by all nodes or a controller that failed to receive the message sends an "error frame" destroying the remainder of the message. For level one operation, the master election is performed initially at the startup of the system, where a potential master node listens on the network traffic and in the absence of another master starts broadcasting the reference message for the beginning of a basic cycle. The priority of the reference message is used to select the masters; thus, the highest priority master defines the cycle time all nodes. When it fails, the master with the highest priority from remaining potential masters takes its place. The whole message exchange still respects the CAN standard, but avoids time-consuming collisions.

**Byeflight**

The Byteflight protocol ([Berwanger et al., 2000](#)) targets automotive applications using a fiber-optic based star topology. The protocol implements a Flexible Time Division Multiple Access (FTDMA) scheme (i.e., the Mini Slotting algorithm) similar with the ARINC 629 Protocol ([Aeronautical Radio Inc., 1991](#)) from the aerospace industry. However, it is not a true time-triggered protocol, as it supports the synchronization of all messages to a common clock, and no global timing synchronization. The protocol relies on periodic clock pulses for constructing a common time base in all nodes connected to the Byteflight network.

The nodes are connected to a central star coupler, and each node has a unique identifier in the network. The network processor implementing the Byteflight protocol ensures that a node gets exclusive access to the bus using a strict ascending identifier sequence within a given cycle, thus avoiding collisions. It also ensures that any identifier is sent only once per network cycle.

For the implementation of the FTDMA scheme, each node in the network has a "slot counter" initialized with zero and then incremented upon synchronization pulses. When the slot counter reaches an identifier value for which a transmission request is present, the corresponding message is transmitted via the bus and all nodes freeze their slot counters until the message transmission completes. Afterward, the nodes continue incrementing their slot counters.

**TTP/C**

The Time-Triggered Protocol (SAE Class C) represents the underlying communication protocol of the Time-Triggered Architecture ([Kopetz and Bauer, 2002](#)). The architecture defines a distributed system of up to 64 computational nodes (more with multiplexing) forming one or more clusters. Each node uses a dedicated network processor to run the TTP/C protocol on two redundant communication channels within a bus or star networking topology (mixtures of star and bus topologies are also possible through star
couplers). The protocol provides the mechanisms for clock synchronization and fault-tolerant deterministic message exchange between applications running on the cluster’s nodes.

The nodes of the cluster exchange information within TDMA rounds (i.e., bus cycles). A TDMA round consists of a set of communication slots, where each slot has an action time (i.e., starting moment of the slot), a transmission phase, and an inter-frame gap, where the protocol algorithms are executed by the network processor of each node. Each node has assigned a slot for communication within each TDMA round. The slot duration for each node depends on the amount of information provided by that node (i.e., proportional with the number and size of messages multiplexed in that slot). The applications running on each node communicate through messages, which are statically scheduled for transmission within the slots of the TDMA rounds. The static allocation of messages to slots and slots to nodes forms the Message Descriptor List (MEDL) that also describes the timing properties of the communication. In other words, it specifies what messages to exchange and when each node has to send or receive messages from other nodes. Every node stores a copy of the global message schedule in its own MEDL prior to the system’s startup. A cluster cycle represents the smallest number of TDMA rounds after which the message schedule becomes periodic; it also determines the length of the MEDL.

![Figure 2.10: The logical blocks of a TTP node](image)

The clock synchronization mechanism [Kopetz and Ochsenreiter 1987] of the TTP/C protocol relies on a fault-tolerant average algorithm with at least four nodes (two and three nodes configurations are possible with degraded performance) to tolerate one Byzantine fault. This mechanism is a specialized version of [Welch and Lynch 1988] for \( t = 1 \) (single fault case). The correctness of the algorithm was formally proven in [Pfeifer et al. 1999]. It exploits the common knowledge within the MEDL, and every node measures the difference between the known expected and the actually observed arrival time of a correct message. Then each node computes the difference between the sender and its local clock, and a correction term for the local clock. The algorithm considers as clock sources only the nodes with accurate clocks (with the SYF flag set in the MEDL). The resulting local notion of time represents the cluster time.
Within its dedicated slot, a node may transmit a single frame containing control information and one or more messages. The messages do not contain the source or the destination nodes, as they can be inferred from the timing of the message and the MEDL. Initially, a network frame was capable of transporting 16 bytes of message data (TTTech, 1999); later, the payload size increased to 240 bytes (TTTech, 2002). An experimental implementation (Schwarz, 2002) using Gigabit Ethernet reached 1476 bytes of useful data per frame (considering the maximum of 1514 bytes of a standard Ethernet frame size).

The fault-strategy of the TTP is the "single-fault" hypothesis, meaning that any single fault is reliably detected and tolerated, up to the loss of an entire node. In addition, the protocol supports the detection of many other higher-level errors. There are three operating scenarios possible: the desired fault-free scenario, a second more probable with error detection and integrated fault-tolerance for all single and multiple faults, and a third "never give up" strategy, where all non-supported faults are handled by the application, which may decide for a clean cluster restart or any other recovery method.

The access to the passive communication medium is guarded on each node by a so-called bus guardian that protects the medium from failures such as babbling-idiot (i.e., faulty nodes that try to overuse the medium). Depending on the network topology, the bus-guardian may be present locally with the TTP processor, or centrally within the star coupler. The interaction between the host processor and the network processor is performed only through a temporal firewall, namely the Communication Network Interface (see Figure 2.10). The CNI contains two shared memory areas: Status/Control and Message areas. The first provides information about the state of the protocol, whereas the second provides the means for sending and receiving messages. The only control information passing through CNI is the global time, as the host processor cannot enforce the transmission time. This design avoids propagation of timing errors from the host to the communication medium.

The Membership Service is a distributed message agreement algorithm that guarantees the sanity of the nodes actively communicating on the bus. Through this algorithm, all faulty nodes lose their membership. It provides reliable error detection of all communication faults within less than two communication rounds, but does not cover application software faults. The membership algorithm was formally proven in (Pfeifer, 2000). The information within each frame is signed with a 24-bit CRC code to ensure its correctness. Before each send operation of a node, a clique avoidance algorithm checks whether the node is a member of the majority clique.

**FlexRay**

FlexRay (Berwanger et al., 2001; Bogenberger et al., 2002) is an emerging high-speed fault-tolerant protocol for control applications. A group of companies including BMW, DaimlerChrysler, Bosch, Philips, and Motorola is actively developing it, and its spec-
2.4. COMMUNICATION IN DISTRIBUTED SYSTEMS

ifications are not yet complete. Apart from time-triggered operations, its focus is flexibility. Thus, it can operate in both active star and passive bus topologies, and can accommodate both static and dynamic parts in its communication rounds. Node may be connected through one or two communication channels (for redundancy), and may transmit the same data on both channels, or different data on both channels, in a given current time slot.

The static part of the communication cycle uses a classical TDMA strategy, whereas for the dynamic part it uses the Byteflight protocol with FTDMA scheme. Three runtime configurations are possible: purely static, purely dynamic, or mixed (static + dynamic). In dynamic configuration, it uses the Byteflight protocol, and a master’s synchronization pulse starts every communication cycle. In the other configurations, it uses a globally synchronized time base for the timing of messages. The time-triggered synchronous messages (critical, no interrupts permitted) are transferred within the static part of a cycle, whereas the asynchronous messages (event-based) are transferred within the dynamic part (interrupts allowed).

Similar with the TTP/C protocol it employs a bus guardian for on each communication channel to avoid the ”babbling idiot” problem, where a faulty node might overuse the communication channel. The bus guardian ensures that, during the static part of the communication round, a node may transmit only within its allocated time.

The global time is available in all FlexRay network; however, only the nodes participating in the static part of the communications cycle determine the global time through the Fault Tolerant Midpoint algorithm. This standard [Welch and Lynch, 1988] averaging clock synchronization algorithm uses the clock values of all or some of the participating nodes, and discards a number $q$ of largest and smallest clock values. It then averages the remaining two extreme clock values (smallest and highest). To tolerate $k$ Byzantine faults, it relies on the hypothesis that at most $k$ clocks are faulty at any synchronization moment, and $n = (3k + 1)$ nodes actively contribute to synchronization.

Unlike the TTP/C protocol, FlexRay does not store the full schedule for the time-triggered static part in each network controller. The cycle is divided into a number of slots of fixed size and each controller and its bus guardians have the information only regarding the slots allocated for their transmission. Because of the fixed slot sizes, the nodes may have more slots per cycle (in contrast with TTP/C) to accommodate higher bandwidth requirements. At startup, the controllers ”learn” the full schedule of the FlexRay cluster, as all messages include the identifier of the sender node. Operational failures may appear when another faulty node incorrectly uses another node’s identifier within the first messages. As there is no membership service, the partitioning of the cycle remains corrupted until the next restart of the network. The fault-hypothesis of the protocol is unknown, and it appears there are no mechanism to handle faults such as slightly out of specification (SOS), asymmetric broadcasts, and the exclusion of a faulty node. A more comprehensive comparison of the TTP/C and FlexRay protocols is available in [Kopetz, 2001].
TTCAN protocol

Node A  Node B  Node C  Node D

Twisted pair, up to 1Mbit/sec.

Basic cycle: From clock master
1-8 bytes of data

Byteflight protocol

Node A  Node B  Node C  Node D

Optical fiber, 10Mbit/sec.

250us From clock master
1-12 bytes of data

TTP/C Protocol

Node A  Node B  Node C  Node D

Twister pair or optical fiber, 2-5-25Mbit/sec.

TDMA round 1-240 bytes of data

FlexRay Protocol

Node A  Node B  Node C  Node D

Twister pair or optical fiber, up to 10Mbit/sec.

Bus, star or mixed topology

12 or 246 bytes

Figure 2.11: Overview of the presented time-triggered protocols
Chapter 3

The Timing Definition Language

The Timing Definition Language was developed within the MoDECS project\(^1\) as a successor of the Giotto language (Henzinger et al., 2003). It addresses several Giotto shortcomings for modeling large-scale and possibly distributed applications. TDL is a high-level description language for specifying the explicit timing requirements of a time-triggered application, which may be constructed out of several independently developed components running in parallel and dynamically exchanging information.

Although, it relies on the same core concept of the Giotto language, namely the Logical Execution Time for computational units, it introduces a component model; improves the syntax and semantics; and provides full support for transparent distribution (Farcas et al., 2005b) of components over a real-time network. As a result, TDL becomes more appropriate for the development of complex applications. A short summary of the differences between the two languages is available in (Farcas et al., 2005b; Templ, 2004). The actual functionality of a TDL application can be developed in any other language available for the target platform (e.g., C, C++, Java), and later linked with the compiled TDL source.

In this chapter, we first present the core concept of the language, the Logical Execution Time. In Section 3.2 we define the component model of TDL and its constituents, which are the mode, task, sensor, actuator, guard, port, driver. We introduce in Section 3.3 a novel concept of ”transparent-distribution” in real-time systems, which is crucial for the development of platform-independent real-time components. The components can be developed independently and later integrated to form complex applications, regardless of the topology of the final system (i.e., single node or distributed system). Within the last section, we formally define the operational semantics of the language by analyzing first the case of a single module, then the modular composition with import relationships between modules, and concluding with the case of distributed modules with dependencies.

---

\(^1\)The Model-Based development of Distributed Embedded Control Systems project (www.modecs.cc) was supported by the FIT-IT Embedded Systems grant 807144 provided by the Austrian government through its Bundesministerium für Verkehr, Innovation und Technologie.
3.1 Logical Execution Time concept

The Logical Execution Time concept was first introduced in the realm of the Giotto Project [Henzinger et al., 2003] at the University of Berkeley, California. This abstraction creates a clear separation from the environment time and the software time in relation with a computational system, and is based on the timed model [Kirsch, 2002] (see Section 2.2.3). In the following we refer to a task as a single-threaded computational unit composed of one or more jobs, without any interaction with any other computational units during its execution. The access to external resources or any other interaction is allowed only at the beginning and at the completion of the task.

The Logical Execution Time abstraction decouples the target behavior of the computational system from its implementation on a particular platform. From the environment’s point of view, the computation encapsulated into a task starts with the release event of that task and continues until the termination event of the task. The upper part of the Figure 3.1, above the time line, illustrates the so-called logical view over the task execution, as logically it does not matter how exactly the task executes within this interval of time. It is only important that the environment provides the input to the task at exactly the release moment, whereas the task provides the results back to the environment at exactly the terminate moment.

However, from the software platform point of view, the task may actually start later than the release event, depending on the scheduling policy used or other software constraints. In a system supporting preemptive multitasking, another task running on the system, or the kernel of the operating system may preempt the task, as depicted in the lower part of the Figure 3.1. Therefore, there could be intervals of time within the terminate-release interval when the task is not actually running. In the case of a fast enough CPU, the task may even complete sooner than the termination event with the stop event. In this case, the system provides the output of the task only at its termination event to the environment.

Using the LET abstraction, we detach from the traditional platform oriented design of an application towards a model-based design. We decouple the timing of an application that reflects its interaction with the external environment from its implementation.
3.1. LOGICAL EXECUTION TIME CONCEPT

on a specific platform. We regard the platform as a combination of hardware and an adequate real-time operation system (RTOS). This is an important step towards platform independence of the application. Changing the platform such as hardware upgrade or newer compiler does not affect the temporal behavior of the application.

From a development perspective, the LET concept has a unit-delay behavior, because tasks exchange information only at LET boundaries and during LET the output ports retain the values of their previous invocations. Although this may appear a disadvantage, it clears by design race conditions, deadlocks, and priority inversions problems. In addition, it provides determinism, composition (Kirsch, 2002), and platform abstraction, which are important for the development of safety-critical systems.

**Determinism**  As a direct consequence of the Logical Execution Time abstraction, provided that the computational system behaves correctly, the interaction between the environment and the computational system has the characteristics of the time and value deterministic systems. From the environment point of view, the computational system provides always the same output values at always the terminate event regardless of its particular implementation on a target platform. We gain immediate benefits in system predictability, with reproducible results, that facilitate later maintenance and debugging.

**Composition**  of real-time software can be easily achieved using the Logical Execution Time abstraction. As the only interaction between the tasks of a computational system and the environment happens at the release and terminate moments, the time and value determinism of the computational system is not influenced by the amount of functionality code, assuming that there are sufficient run-time resources such as CPU, memory. Thus, we can design applications out of individual modules or components with well-defined syntax, semantics and behavior. Moreover, we can decompose complex applications into several simpler components that interact to replicate the initial behavior.

A modular design also brings in another positive property: extensibility typically lacking in traditional real-time systems design. If we have sufficient CPU performance and related run-time resources, then we can extend the existing functionality of a computational system by adding more components.

**Software standardization**  in the context of real-time systems, represents the property of a software development methodology, which allows for independent behavior specification of a software component from its implementation on a particular platform. As the LET abstraction decouples the timing specifications of an application from its implementation, we can use languages based on LET as the first step towards software standardization in real-time systems.

Interesting properties arise, as without a particular platform for an application, we can perform hardware upgrades or complete platform changes without compromising
the behavior of the application. Moreover, we can move software components from
one platform to the other, or within nodes of a distributed system. In addition, we
can construct real-time-component libraries, which we can later use for complex ap-
lications. Although the approach is popular for applications programmed in general
programming languages, it currently remains a challenge in real-time applications.

3.2 TDL component model

The Giotto language, first introduced in the realm of the Giotto Project (Henzinger
et al., 2003), represents the first tangible implementation of the Logical Execution Time
abstraction. It describes the temporal behavior of an application in respect to the en-
vironment and relies on additional languages and tools for the actual implementation
of the functionality of the application. However, the usage of Giotto language is not
adequate for developing large applications, because of the resulting code complexity
and maintenance difficulty. To cope with this severe limitation of Giotto, the Timing
Definition Language inherits from Giotto its basic elements (e.g., task, modes), intro-
duces a new component model, and goes further with new constructs, simplified syntax,
and improved semantic.

The component model of TDL (Templ, 2004) allows the construction of complex
applications out of individual components, which may be developed standalone, and
later integrated into the final design of the application. The TDL components that
form an application may work independently, each addressing a specific part of the
functionality of the application, or may collaborate to implement a complex behav-
ior. In addition, the TDL component model allows for decomposing existing complex
applications into smaller, more manageable parts, each with specific timing and func-
tionality, and provides the means for deterministic component interaction. Developers
can reuse existing components to extend the functionality of an application or create
new applications.

With complex real-time systems, the TDL component model allows the distribution
of parts of an application on several computing nodes. It also allows the opposite action
of consolidating the functionality of several nodes into one or more powerful nodes. In
all these cases, the overall functionality of the system remains unchanged, provided
that the individual processing capabilities of each node are sufficient, and the attached
peripherals, such as sensors and actuators, are adapted successfully to the new system
configuration.

In the following subsections, we present the TDL building blocks: first their informal
description, followed by their formal semantics. For the concrete syntax of TDL, we
refer to (Templ, 2004). We call the language building blocks (i.e., ports, tasks, sensors,
actuators, modes, drivers, types and constants) as TDL elements. The terms TDL
application and TDL program are equivalent.
3.2. TDL COMPONENT MODEL

3.2.1 Modules
The TDL component model relies on the concept of a module. The TDL module encapsulates a complete application or parts of an application. Acting as a unit of composition or decomposition of an application, the TDL module provides separated name-space environment for all contained entities. In this sense, we can regard a TDL module similar with a complete Giotto program.

A TDL module may contain a combination of elements inherited from Giotto such as constants and data types, sensors, actuators, and tasks with input and output ports, modes. For example, it can act just as a container for constants and data types with no actual functionality for an application. The TDL code sequence 3.1 presents the usage of a TDL module construct to group a set of constants and types used for configuring an application. In large applications, a TDL module may partition the set of actuators to indicate a specific functionality, and to prevent accidental access to them by other parts of the application. With increasing processing capabilities of the computational system, it may extend an application with new functionality. Nevertheless, from the developer point of view, all TDL modules that provide the functionality of an application, run independently and logically in parallel.

Listing 3.1: Sample TDL module with constants and types

```tcl
module MyConfiguration {
    const // simple constants used for initialization or configuration
    public IO_PORT = 100;
    public BIT_MASK0 = 0xFC;
    public initzero = 0;
    type // Opaque types in TDL have an external correspondent in a language such as C
    public IOBlockBuffer; // ex. typedef unsigned long IOBlockBuffer;
    public LEDarray; // typedef struct { DWORD hwaddress; WORD val[32]; } LEDarray;
}
```

Formal definition. We define the TDL module \( M \) as the tuple \((C[M], Types[M]_O, Sensors[M], Actuators[M], Tasks[M], Modes[M], Imports[M])\), with \( C[M] \) as the set of constants introduced by the module \( M \), and \( Types[M]_O, Sensors[M], Actuators[M], Tasks[M], Modes[M] \) as the opaque types, sensors, actuators, tasks, respectively modes defined in \( M \). The set \( Imports[M] \) represents the set of modules imported by \( M \). The set \( TDL \) represents the set of all possible TDL modules.

We define the notion of a TDL entity \( e \) from a module \( M \), as a named TDL element defined in the module \( M \). The set \( M_e = C[M] \cup Types[M]_O \cup Sensors[M] \cup Actuators[M] \cup Tasks[M] \cup Modes[M] \) represents the set of entities defined in the module \( M \), whereas the set \( TDL_e \) represents all possible TDL entities from all possible modules. A restricted set \( M_e^f \) of functional entities of a module \( M \) consists of the sensors, tasks and actuators defined in the module: \( M_e^f = Sensors[M] \cup Actuators[M] \cup Tasks[M] \). For example a
module $M$ containing only two tasks $\tau_1$ and $\tau_2$ within a mode $m$ has the entities $M_e = \{\tau_1, \tau_2, m\}$ and $M^1_e = \{\tau_1, \tau_2\}$.

3.2.2 Import relationship

A component model requires support for interactions between components. Therefore, TDL introduces an import/export relationship between modules. We call a module that exports a subset of its TDL entities to another module as a service provider module. Similarly, we call the second module that imports a subset of TDL entities from the service module as a client module. A TDL module can both import one or more modules and still be imported by another one or more modules. However, TDL introduces restrictions in this mechanism, such that one module cannot import itself and the import relationship cannot form cycles. Hence, the import relationship between any TDL modules may only form a directed acyclic graph. This restriction on cyclicity comes from compilation and run-time environment restrictions over the TDL, and it is not a limitation of the LET based programming model. Several solutions for solving these problems are presented in the sections 4.1 and 4.2, whereas their practical implementation remains as future work.

We can use the import relationship to access the constants, types and task output ports from a different module. Sensor values from a service module can only be accessed through tasks from the same module. We regard the actuators as local resources to a module, which cannot be accessed by other modules, to prevent possible conflicts with multiple writers on the same environment output resource. We can only export the set of entities marked as public from a service provider module, and then make available their output values to other entities from a client module, which imports the service module. We regard all other entities from the service module as private; therefore, any reference to them in other modules results in a compilation error. The access to the output values from a service module remains governed by the LET semantics. In a simplified view, we can see the service module as a subcomponent of the client module.

Formal definition. We define the export possibility of a TDL entity $e$ from a module $M$, via a function $\beta_M : M_e \rightarrow M_e$; $\beta_M(e) = \{e \mid e \text{ public}, e \in (\text{Tasks}[M] \cup \text{C}[M] \cup \text{Types}[M]_O)\} \cup \{\emptyset \mid (e \neg \text{public}) \vee (e \in (\text{Modes}[M] \cup \text{Actuators}[M] \cup \text{Sensors}[M]))\}$. The set $\beta_e[M] = \{e \mid \beta_M(e) \neq \emptyset, e \in M_e\}$ represents all exported entities from the module $M$.

An import declaration ($M$, alias) consists of a module $M$ (its name) and an optional alias we can use to refer to $M$. For a module $M_1$, we define an import function $\text{Imports}_{M_1} : \text{TDL} \rightarrow \text{TDL}; \text{Imports}_{M_1}(M_2) = \{M_2 \mid M_1 \text{ imports } M_2\} \cup \{\emptyset \mid M_1 \neg \text{imports } M_2\}; M_1, M_2 \in \text{TDL}$, which expresses the possibility of the module $M_1$ to import other modules. The import relationship for a module $M_1$ is defined on the set of modules $\text{Imports}[M_1]$, where $\forall M_2 \in \text{Imports}[M_1], \exists \text{Imports}_{M_1}(M_2) \notin M_1 \cup \emptyset$, and the module does not import itself nor there are import cycles $\exists M_3, M_4, \ldots, M_n \in \text{TDL}$,
3.2. TDL COMPONENT MODEL

Imports $M_2 (Imports_{M_3} (\cdots Imports_{M_n} (M_1) \cdots)) = M_1$. We define the set of imported entities for a module $M_1$ as $Imports_e[M_1] = \{ e \mid \exists M_2 \in Imports[M_1], e \in \beta_e[M_2] \}$. Any reference in a module $M_1$ to an entity $e \in (M_2 \setminus \beta_e[M_2]) \cup M_3$, $M_2 \in Imports[M_1]$, $M_3 \in TDL \setminus Imports[M_1]$, represents an incorrect reference to another entity, which results in a compilation error. The supplementary restrictions related to the port output values of an imported entity are expressed in the next section.

3.2.3 Constants and data types

To initialize the state and output ports of TDL entities from a module, a developer may use constants in a similar fashion with other programming languages such as C or Java. At compile time, the TDL compiler replaces the constants from a module with their assigned values. Constants can be used for all basic types defined. For opaque types, the developer must provide an appropriate initializer function.

TDL defines two kinds of types for the sensor, actuator, or task ports: basic and opaque. The basic types are the primitive types found in most programming languages: byte, short, int, long, float, double, char, boolean, and a limited size string. The opaque types are used-defined types, for which the developer must provide an external structure or class with the same name as the corresponding TDL type. The bindings between the opaque types referred in the TDL module and their external correspondent is performed through symbolic linking. For an example of constants and types see Listing 3.1.

Formal definition. We define a constant declaration as the tuple $(c, v, attribute)$, where $v$ represents the value $v \in V$ of the constant $c$ and the optional $attribute$ parameter may be equal with $public$ to denote an exportable constant. The set $C[M]$ represents the set of constants defined in the module $M$.

We define an opaque type declaration as the tuple $(\theta, alias, attribute)$, where $\theta \in Types[M]_O$ is the name of the user-defined opaque type with a corresponding external implementation and the $attribute$ optional parameter denotes its exporting option. We use the $Types[M]_O$ notation for the opaque types defined in a module $M$.

3.2.4 Interfacing elements - ports

TDL entities within a TDL module communicate under the LET semantics via ports. From data perspective, a connection from a producer entity to a consumer entity is valid when the output ports of the producer entity and the corresponding input ports of the consumer entity have the same or equivalent types. The ports are also the only interfacing units between different TDL entities from a service to a client modules. In this sense, we can regard the TDL ports as the typical variables from general programming languages. The only possible ports are: input, output and state ports, depending on type of the entity that owns them and their intended purpose. The ports can have a basic type, such as byte, boolean, or an opaque type for which TDL uses symbolic referencing. The developer has to match an opaque type declared into a TDL module.

\[47\]
CHAPTER 3. THE TIMING DEFINITION LANGUAGE

with a corresponding type from the implementation language used for the functionality code. We can use the ports to transfer data between concurrent tasks, or from/to the environment, e.g. via a sensor to a task that computes a control law and responds via an actuator. In all combinations, regardless of their type, the ports retain their values until the entity that owns them updates their values. The set of output ports defined in a TDL module has to be initialized through typed-in values, constants or external initializer functions. This initialization is necessary to prevent the non-determinism regarding their values when the module starts or changes its state.

**Formal definition.** We define a *port* as a data storage entity, e.g. memory space referred through a variable in a general-purpose programming language such as C or Java. Formally, a port $p$ belongs to a TDL entity $e$, and has a set of properties denoted by the functions: $\text{kind}$, $\text{Types}$, $V$, $\psi$, $\beta$.

We define a function $\text{kind}$ for a port $p$, which returns one of the three possible values: *input*, *output*, and *state*. Any port $p$ can be of only one $\text{kind}$. A TDL entity may only read from its *input* ports, and write to its *output* ports, whereas for the *status* ports it may perform both read and write operations. We define the set $P[M] = P_o[M] \cup P_a[M] \cup P_t[M]$ of TDL ports of a module $M$ as the reunion of sets of its sensor ports $P_o[M]$, actuator ports $P_a[M]$, and task ports $P_t[M]$. The set $P_o[M] = P_{\tau_o}[M] \cup P_{\tau_o}[M] \cup P_{\tau_s}[M]$ contains all task input, output, and state ports. We also define its two subsets of input and output ports: $P_i[M] = P_{\tau_i}[M] \cup P_{\tau_o}[M] \cup P_{\tau_s}[M]$, respectively $P_o[M] = P_o[M] \cup P_{\tau_o}[M] \cup P_{\tau_s}[M]$.

For each port $p \in P[M]$ of the module $M$, we define the function $\text{Type}(p)$ as the port type: $\text{Type} : P[M] \rightarrow \text{Types}$, $\text{Type}(p) = \{\text{Types}_p \mid \text{Types}_p \in \text{Types}\}$. The set of possible types $\text{Types} = \text{Types}_B \cup \text{Types}[M]_O \cup \text{Types}[\text{Imports}[M]]_O$, consists of the basic TDL types $\text{Types}_B$, the opaque types $\text{Types}[M]_O$ defined in the module $M$, and the opaque types $\text{Types}[\text{Imports}[M]]_O$ exported from all modules imported by $M$. The *type* function of a port assigns exactly one type for each port of a module.

We define the valuation function $V : P[M] \rightarrow \Upsilon$, $V(p) = v$, $v \in \Upsilon$, which provides the *value* of the data storage entity expressed by the port $p$. The set $\Upsilon$ represents the range of possible values for all ports in $P$. Its subset $\Upsilon_p$ relates to the set of values given by the type $\text{Type}(p)$ of the port $p$. We define the state $S(p,t)$ of a port $p \in P[M]$ at the moment $t \in \mathbb{N}$ as a function: $S : (P[M], \mathbb{N}) \rightarrow \Upsilon$, $S(p,t) = \{V(p) \mid t\}$, in other words, the value $V(p)$ of the port $p$ at the moment $t$.

We further define an initial state for output ports $P_o[M]$ of a module $M$, via the function $\psi : P_o[M] \rightarrow \Upsilon$; $S(p_o, 0) = \psi(p_o)$; $p_o \in P_o[M]$; $P_o[M] \subset P[M]$. The initializer function $\psi(p_o)$ can only refer to a typed-in value, a locally defined constant $c \in C[M]$, an imported constant $c_{\text{Imports}} \in C[\text{Imports}[M]]$, or an externally user-defined function for the ports with opaque types $\text{Type}(p) \in \text{Types}[M]_O \cup \text{Types}[\text{Imports}[M]]_O$. All output ports of a module must be initialized before the execution of the module.

The TDL *import* semantics allow us to refer to ports of TDL entities defined in another module. Therefore, we extend the notion of ports of a module $M_1$ to the
3.2. TDL COMPONENT MODEL

notion of ports available in the module $M_1$, with the public output ports from its imported modules: $P = P[M_1] \cup \{p | \exists M_2 \in \text{Imports}[M_1], \exists \tau \in \beta_e[M_2], p \in P_{ro}[M_2]\}$. The input ports subset $P_i = P_i[M]$ remains unchanged and the output ports subset $P_o$ grows with the imported ports.

3.2.5 Scheduled elements - tasks

The fundamental unit of computation in a TDL program is the task. It has none, one or more input or state ports, but at least one output port and an external implementation. A sample TDL tasks is illustrated in the Figure 3.2. From a functional point of view, a TDL task is a function from its input and state ports to its output and state ports. Following the LET semantics, two tasks can exchange information only via their input and output ports, and only at their release and terminate moments.

We can implement the functionality of a TDL task in a sequential programming language, such as C or Java. However, a TDL task cannot have any internal synchronization points. Hence, a developer cannot use the classical semaphores, monitors, or other synchronization primitives within the functionality code of TDL tasks. The reason for this important restriction is that an internal synchronization point introduces an instance of a platform time into the LET of a task, which can be unpredictable. It also interferes with the platform independence property of TDL as it restricts the applicability of the task functionality to a set of platforms for which the corresponding synchronization primitive is available or can be emulated. In addition, without any internal synchronization mechanisms, the run-time environment can freely schedule the task and avoiding by design any race-conditions, deadlocks, and priority-inversion problems. From a logical point of view, we can regard a TDL task as an atomic computation from its release time until its deadline, or a black-box that does not interfere with other tasks except through its CPU time requirements.

A TDL task has one platform-dependent property: its Worst Case Execution Time (wcet). Depending on the CPU speed of the platform, the particular implementation of the task functionality, the compiler used to generate the executable code, and the scheduling policy of the run-time system, the wcet property of a task along with its LET determines the possibility of its time-safe execution at run-time.
A set of TDL tasks runs *logically in parallel*, whereas a particular implementation of the TDL run-time system may decide to execute them in a non-preemptive sequence, or using a preemptive scheduling mechanism. TDL does not require a preemptable task system, although most run-time platforms support preemptive based scheduling of user-level tasks or threads. However, a TDL task runs until completion, as a forced termination may lead to inconsistent application state. A developer of TDL programs specifies the intended logical behavior, whereas the TDL compiler has to make the necessary time-safety checks and the run-time environment has to ensure its time-safe execution on the target platform.

Formal definition. We regard as TDL task as the tuple \((τ, u[τ], P_i[τ], P_s[τ], P_o[τ], \text{attribute})\), where \(u[τ]\) is an external user-defined function from its set of input and state ports to its output ports: \(u[τ]: (P_i[τ], P_s[τ]) \rightarrow (P_o[τ], P_s[τ])\). Each task \(τ\) introduces its own sets of input, state and output ports \(P_i[τ], P_s[τ], P_o[τ]\) into the port set of the module \(P[M]\). The set \(P[τ] = P_i[τ] \cup P_s[τ] \cup P_o[τ]\) contains all ports of the task \(τ\). Two or more tasks cannot share any of their ports: \(∀τ_1, τ_2 ∈ \text{Tasks}[M], P[τ_1] \cap P[τ_2] = \emptyset\).

The *attribute* optional parameter may be *public*, which denotes that the task output ports may be available to other modules that import the module containing this task.

All tasks of a module \(M\) have a positive, non-null and finite *worst-case execution time*: \(∀τ ∈ \text{Tasks}[M], τ_{wcet} = w(τ); w: \text{Tasks}[M] \rightarrow \mathbb{N}^∗\). Considering the execution time \(τ_{ET}\) of an arbitrary instance of the task \(τ\), and its *logical execution time* as required by the application behavior \(τ_{LET}\), the Equation (3.1) provides a generic condition for the time-safe execution of the module on the target platform.

\[
∀τ ∈ \text{Tasks}[M], \quad τ_{ET} \leq τ_{wcet} \leq τ_{LET}
\] (3.1)

To preserve the determinism of a module \(M\), we call a *design error* for the module \(M\) the case when \(∃τ ∈ \text{Tasks}[M], τ_{wcet} > τ_{LET}\), which means that the task is required to complete faster than it is possible. We call an *overrun* for the module \(M\) the case when \(∃τ ∈ \text{Tasks}[M], τ_{ET} > τ_{wcet}\), which means that the task executed longer than expected. All TDL tasks must have a finite execution time.

### 3.2.6 Reactive elements - sensors and actuators

A TDL program reacts to changes in the environment via *sensors* and *actuators*. The sensors provide input to the TDL program by querying some properties of the environment and returning corresponding values. A sensor has an output port and an external getter function, which returns a value compatible with the sensor’s type. However, depending on the sensor type, the returned value may encompass multiple status bits, or other multiplexed data. For example, a sensor may provide the speed and direction of a vehicle, or its acceleration over three axes, the inside and outside temperatures. The actuators provide the output of the computational system to the external environment. For example, heating the interior of a vehicle, turning the wheels, or simply setting an
I/O pin. An actuator has a pre-initialized input port and an external setter function implemented in a general-purpose language such as C or Java. In a typical application, to provide the results of a task to the environment we connect actuators to the task’s output ports.

The mode declaration defines the exact moments in time when the run-time system performs the interaction with the environment. At those moments, the run-time system calls the appropriate sensor getter driver that updates the sensor port. Similarly, when an actuator must be updated, the run-time system calls its setter driver with the value of the actuator port as argument. According to the timed model semantics, the only environment to TDL program interactions are through sensors and actuators, and the actual functional code of these reactive elements executes in zero logical time. This ideal concept for executing the sensor and actuator functionality simplifies the design and comprehensibility of TDL programs. However, in practice the run-time system has to take into account the reading time of the sensors and the writing time of the actuators. A typical implementation for reading a sensor would be reading a memory location or a CPU register, which would come close to the maximum performance of the target platform.

In contrast with Giotto, TDL sensors can be used as inputs to actuators. This feature implies synchronous computation (see Section 2.2.1) similar with Esterel (Berry, 2000), which can be applied in the case when the overhead of the sensor getter and actuator setter functions is negligible. Typical applications could be signaling various changes in sensors states, such as a LED indicating the push of a button.

Formal definition. Syntactically, the declaration of a TDL sensor consists of the tuple \((\sigma, \text{Type}(\sigma), u[\sigma], \text{attribute})\), where \(\text{Type}(\sigma)\) represents the type of the sensor \(\sigma\) and the optional \text{attribute} parameter denotes its exporting possibility. Similarly, an actuator declaration consists of the tuple \((\alpha, \text{Type}(\alpha), \psi[\alpha], u[\alpha])\).

Operationally, we regard the reactive TDL element sensor \(\sigma\) and actuator \(\alpha\) as interaction ports with the environment, that are the tuple \((u[\sigma], p_\sigma)\), respectively \((u[\alpha], p_\alpha)\). The external user-defined functions: sensor getter \(u[\sigma]: \text{Env} \rightarrow \Upsilon_\sigma\), respectively actuator setter \(u[\alpha]: \Upsilon_\alpha \rightarrow \text{Env}\), provide the interaction between the computational system and the external environment \(\text{Env}\). For a module \(M\) the following relations exists: \(\forall \sigma \in \text{Sensors}[M], \ p_\sigma \in P_\sigma[M] \subset P_\sigma[M], \ \text{Type}(\sigma) = \text{Type}(p_\sigma), \ V(p_\sigma) \in \Upsilon_\sigma\), and \(\forall \alpha \in \text{Actuators}[M], \ p_\alpha \in P_\alpha[M] \subset P_\alpha[M], \ \text{Type}(\alpha) = \text{Type}(p_\alpha), \ V(p_\alpha) \in \Upsilon_\alpha\). In addition all actuator output ports must be initialized \(\forall \alpha \in \text{Actuators}[M], \ \exists \psi(\alpha)\).

For actuators with basic types, the initializer function \(\psi(\alpha)\) can be replaced with a constant \(c \in \text{Imports}[M]\). The sensor getter function translates a state change of the environment into a value change in the corresponding sensor output port. The actuator setter function translates the value from its input port into a specific state or a state change of the environment. We use the notations \(\text{Sensors}[M]\) and \(\text{Actuators}[M]\) for the set of sensors, respectively actuators of the module \(M\).
3.2.7 Synchronous elements - drivers

The original Giotto drivers are user-defined explicit functional entities, which perform the actual interfacing operation between different Giotto computational units, by copying the values of the producer output ports to the consumer input ports, provided that the ports have the same types or a type conversion can be made (see Figure 3.3). Giotto drivers are atomic, single-threaded pieces of code (Henzinger et al., 2003). The drivers execute very fast in comparison with other Giotto activities, so that the assumption of zero logical time holds true. Therefore, we can regard them as synchronous elements (Kirsch, 2002) in the sense of synchronous programming languages, which means that they satisfy the synchrony assumption (Halbwachs, 1993) of executing before the state of the environment changes significantly to influence the behavior of the control system.

![Figure 3.3: Implicit driver connecting two TDL tasks](image)

TDL simplifies the Giotto drivers by making them implicit instead of explicit. From the TDL developer point of view, the drivers are transparent, as there is no need to define drivers to interface different parts of TDL programs. A simpler and stricter syntax relieves the developer from the burden of specifying and then implementing the drivers. The TDL compiler provides automatic syntactic type checking for the interconnection of various TDL entities, whereas its plugins generate automatically glue-code that implements the semantic-correct port value copying mechanism. Hence, the Giotto concept drivers are no longer relevant at development time but at run-time.

**Example.** Consider the TDL module from Listing 3.2, where we define a set of three sensors, which read the position of a high-pressure valve, and a filter task, which calculates the correct position of the valve, discarding possible errors from a faulty sensor. We also define an actuator, which takes as input the output of the filter task.

```
Listing 3.2: Ports and drivers example

module sensor_filter {
  const max = 10; // maximum accepted deviation
  sensor
    int s1 uses getSensor1; // Port s1_out, sensor getter driver(getSensor1)
    int s2 uses getSensor2; // Port s2_out, sensor getter driver(getSensor2)
    int s3 uses getSensor3; // Port s3_out, sensor getter driver(getSensor3)
  actuator
    int a1 uses setActuator1; // Port a1_in, actuator setter driver(getSensor1)
```

52
3.2. TDL COMPONENT MODEL

```
3.2. TDL COMPONENT MODEL

```
CHAPTER 3. THE TIMING DEFINITION LANGUAGE

We define in the Equation (3.2), the port mapping function \( \lambda \), which maps an output port \( p_o \) from the set of output ports \( P_o \subset P \), to one port \( p_i \), with the same type, from the set of input ports \( P_i \subset P \). The set \( P_o \) consists of all output and status ports of all producer entities available to the module \( M \) (i.e., tasks and sensors) \( P_o = P_o[\tau] \cup P_o[\alpha] \cup P_o[\sigma] \). The set \( P_i \) consists of all input ports of all consumer entities (i.e., tasks or actuators) \( P_i = P_i[\tau] \cup P_i[\alpha] \cup P_i, \forall \tau \in \text{Tasks}[M], \alpha \in \text{Actuators}[M], \sigma \in \text{Sensors}[M] \). The function \( \lambda \) practically provides syntactical type checking for all possible interactions between different TDL entities in the module \( M \) and imported entities from other modules.

\[ \lambda: P_o \rightarrow P_i; \quad \lambda(p_o) = \{p_i \mid \text{Type}(p_i) = \text{Type}(p_o)\} \cup \{\emptyset \mid \text{Type}(p_i) \neq \text{Type}(p_o)\}; \quad p_i \in P_i; \quad p_o \in P_o \]  

We formalize in the Equation (3.3), a defined data dependency between two TDL entities \( e_1 \) and \( e_2 \) (i.e., tasks, sensors, and actuators) via the function \( \gamma \), which encodes all semantically correct relationships between the entities defined in a TDL module \( M \) or its imported modules. The set \( M_{\text{Imports}} \) refers to all TDL entities available to the module \( M \). The function \( \gamma \) captures the intended functional behavior of the module \( M \) as a restriction over all possible port mappings defined by the function \( \lambda \) in the module \( M \). The status ports of a task are available only to the task that defines them.

\[ \gamma: (M_{\text{Imports}}, M_{\text{Imports}}) \rightarrow (P_o, P_i); \quad P_i, P_o \subset P; \]

\[ M_{\text{Imports}} = \text{Actuators}[M] \cup \text{Sensors}[M] \cup \text{Tasks}[M] \cup \text{Imports}[M] \]

\[ \gamma(e_1, e_2) = \{(p_o, p_i) \mid \exists p_i = \lambda(p_o), \ p_o \in P_o[e_1] \setminus P_o[e_1], \ p_i \in P_i[e_2] \setminus P_i[e_2]\} \cup \{(p_o, p_i) \mid e_1 = e_2, \ \exists p_i = \lambda(p_o), \ p_i, p_o \in P_i[e_1]\}; \]  

(3.3)

In the Equation (3.4), we introduce a restriction \( \Lambda \) on the function \( \lambda \) over the interactions defined in the TDL module \( M \), via the function \( \gamma \). Practically, we can see the results of the function \( \Lambda \) as a semantically correct wiring standard from output ports to input ports.

\[ \Lambda: P_o \rightarrow P_i; \quad \Lambda(p_o) = \{p_i \mid \exists(p_o, p_i) = \gamma(e_1, e_2); \ e_1, e_2 \in M_e, \ p_o \in P_o[e_1], \ p_i \in P_i[e_2]\} \]  

(3.4)

We define in the Equation (3.5), a value assignment function \( \Delta \), which returns either \( \emptyset \) or a possible value-copy assignment between two ports \( p_i \) and \( p_o \), defined as in the previous equation.

\[ \Delta: (P_i, P_o) \rightarrow A; \quad \Delta(p_i, p_o) = \{V(p_i) \leftarrow V(p_o) \mid \exists p_i = \Lambda(p_o)\} \cup \{\emptyset \mid \exists p_i = \Lambda(p_o)\}; \quad p_i \in P_i; \quad p_o \in P_o; \quad A = \Delta[P] \]  

(3.5)
3.2. TDL COMPONENT MODEL

We define in the Equation (3.6), an implicit TDL driver $d$ as the tuple $(f[d], P_i[d], P_o[d])$, where the function $f[d]$ copies the values of a set of driver input ports $P_i[d]$ to a set of driver output ports $P_o[d]$ and makes the necessary type conversions. The driver input ports set $P_i[d]$ consists of a subset of the output ports $P_i[d] \subset P_{\text{producer}} \cap (P_o[\tau] \cup P_s[\tau] \cup P_\alpha), \forall \tau \in \text{Tasks}[M], \alpha \in \text{Sensors}[M]$, of one producer entity that has a defined functional relationship with another consumer entity, which provides the driver output ports set $P_o[d]$ consisting of a subset of its input ports (i.e., ports of tasks or actuators) $P_o[d] \subset P_{\text{consumer}} \cap (P_i[\tau] \cup P_s[\tau] \cup P_\alpha), \forall \tau \in \text{Tasks}[M], \alpha \in \text{Actuators}[M]$. Both sets $P_i[d]$ and $P_o[d]$ have the same number of elements and their elements have exactly one correspondent in the other set: $||P_i[d]|| = ||P_o[d]||$. In this sense, we regard the driver function as a restriction of the assignment function $\Delta$ over the sets $P_i[d]$ and $P_o[d]$.

$$f[d]: (P_i[d], P_o[d]) \rightarrow A; \quad f[d](p_i, p_o) = \{\Delta(p_o, p_i) \mid \forall p_o = \Lambda(p_i)\};$$

$$p_i \in P_i[d], \quad p_o \in P_o[d]; \quad P_i[d] \subset P_o[d]; \quad P_o[d] \subset P_i; \quad P_i, P_o \subset P \quad (3.6)$$

The implicit TDL drivers for sensors execute the actual sensor getter functions to update a subset of $P_\tau$. For the actuators there are two actions to perform: actuator port updates on a subset of $P_\alpha$, and then the execution of the actual actuator setter functions. For a description of the functionality of the implicit TDL drivers see Section 5.2.1.

All drivers execute synchronous in logical zero time. As the number of entities in a TDL module and their relationships are finite, we can define a finite number of drivers to perform all value transfer operations required by the LET semantics of a TDL module.

Assuming that the number sensors and actuators of the module $M$ is $||\text{Sensors}[M]||$, respectively $||\text{Actuators}[M]||$, the number of tasks is $||\text{Tasks}[M]||$, with the four main events from the lifetime of their instances release, terminate, start, and stop, the Equation (3.7) defines the maximum number $||D[M]||$ of implicit drivers required to capture all intercommunication actions within the TDL module $M$.

$$||D[M]|| = ||\text{Sensors}[M]|| + ||\text{Actuators}[M]|| +$$

$$+ ||\text{Modes}[M]|| \cdot (||\text{Actuators}[M]|| + 4 \cdot ||\text{Tasks}[M]||);$$

$$||\text{Sensors}[M]|| \geq 0; \quad ||\text{Actuators}[M]|| \geq 0; \quad ||\text{Tasks}[M]|| > 0 \quad (3.7)$$

The number of drivers is typically lower, as the start and stop events are not relevant to the environment, and even zero in the case when a TDL module contains only constants or types declarations. Nevertheless, this number shows that the coding effort with TDL is significantly lower than with Giotto, because with TDL developer simply does not have to worry about them, whereas with Giotto it has to implement each of them correctly.
3.2.8 Application state - modes

A TDL *mode* represents one state of a TDL module, with the possibility of dynamically changing between them according to the intended functionality. Each *mode* of a TDL module has a fixed execution period $\Pi_m$, measured in standard time units such as $\mu$s, ms. A TDL mode encapsulates a fixed set of periodically executed activities. The activities can be *task invocations*, *actuator updates*, or *mode switches*, each with a fixed frequency in relation with the mode period.

In contrast with Giotto, in TDL there are no mode ports. Explicit ports belong only to tasks, whereas the sensors and actuators have implicit output, respectively input ports. During mode changes output ports of the entities from the target mode can be initialized from the old mode.

**Task invocations.** A task invocation represents the execution of a task instance within a mode. The task model of TDL implies that the LET of a task invocation is equal with its period, and the frequency of the task within a mode denotes the number of corresponding task invocations in that mode. Thus, the period of a TDL task invocation in relationship with the period of a mode containing the task is a finite natural number $\pi_T = \Pi_m / \omega_T$. At run-time, we release each new instance of a task at the terminate event of its previous instance. We regard the task invocations as *scheduled elements* (Kirsch, 2002).

The input ports of a task invocation may be task output or sensor ports. In addition, we can use the TDL status ports to transmit private state information to successive invocations of the same task. The binding of a sensor output port to a task input port having the same type, within a mode $m$, denotes the execution of the user-defined sensor getter function with the period of the task invocation. As more than one task can use a sensor as their input, we cannot define a strict period property for sensors.

![Figure 3.4: The interaction of TDL task instances](image-url)

The LET semantics restricts the interaction between different task instances, or task instances and actuators. The output port values of a task instance are available to other entities only at the end of the LET of that task instance, following the so-called *unit delay* behavior (Henzinger et al., 2003). The Figure 3.4 illustrates this behavior, as the first instance of task A communicates with the second instance of task B. Although this
communication mechanism may seem as a waste of time, the availability of the output values only at the end of LET brings us determinism. In short, we trade latency for predictability.

**Actuator updates.** During a mode period, an actuator can be updated with a type-matching output port of a task instance or a sensor. Syntactically, within a mode there can be only one update operation per actuator, which means that there cannot be more tasks updating the same actuator even when user-defined guards provide mutually exclusive access to the actuator setter.

The actuators updates have their own update period defined in relationship with the period of the mode, which encompasses them: \( \pi_\alpha = \Pi_m / \omega_\alpha \). TDL regards the actuator updates as instantaneous, meaning that the execution of the corresponding actuator setter function takes zero logical time. When the period of an actuator update is smaller than the period of the task instance that provides the output values, e.g. it is updated twice as fast, the actuator obtains the previous values of the task instance.

**Mode switches.** A mode may contain a set of mode switch activities \( \text{ModeSWs}[m] \), each with its own frequency \( \omega_\eta \) in relation with the mode period \( \Pi_m \). We consider the state change of a module via a mode switch as a synchronous operation, that is, the interval of time between the end of a mode period and the beginning of the new mode is equal with zero logical time. In practice we assume this time interval is extremely small in comparison with the system clock resolution and the task execution times.

The TDL mode concept is semantic improvement over the Giotto mode concept. A Giotto application can theoretically switch the application mode at any given time; however, a well-timed application \( (\text{Kirsch, 2002}) \) requires that in the target mode the interrupted tasks are present and therefore may continue their execution. TDL applications are well-timed by design as the only moments when a mode switch can occur within a TDL application are the moments when the switch to the target mode does not occur during any of the LETs of all tasks from the source mode. Therefore, TDL allows only harmonic mode switches. We introduce this restriction over the relaxed Giotto semantics to prevent a run-time hazard where in the target mode a high CPU utilization would prevent the time-safe execution of the remainder of a interrupted task from the source mode.

In addition, Giotto requires that only one of the mode-switch conditions evaluates to TRUE. TDL improves this semantic, as the runtime environment evaluates the mode-switch conditions in the order in which they appear in the source code of the module. This sequentiality is the only exception to the general parallel programming model of TDL. There are several benefits from this approach, which comes closer to the well-known sequential programming model of general-purpose programming languages, such as level-based evaluation of sensors/task outputs and adequate mode changes into states with different performance levels, or hierarchical mode changes depending on combinations of values on task output ports.
Furthermore, when a mode switch condition evaluates to TRUE, the execution of a Giotto program continues in the target mode as close to the end of the target mode as possible for a synchronous mode switch. This behavior implies that the target mode is aligned to the logical time, which means that every mode cycle starts at an integer multiple of its period, regardless of the previous number of mode switches. Consequently, during a mode switch the Giotto runtime environment delays the execution of user tasks, until the logical time is an integer multiple of the mode period or the GCD of its activities. In the case of TDL with harmonic mode switches, the mode switches are instantaneous and a new cycle of the target mode starts immediately after the evaluation of the mode switch condition. As a consequence, after a long interval of time, a mode may start at any logical time which is an integer multiple of the GCD of all mode periods of the module.

As there are no mode ports in TDL the developer may specify mode switch assignments between ports of the entities from the old mode and ports of the entities in the target mode. The mode switch drivers update at run-time the values of the ports specified by these assignments.

Listing 3.3: Non-harmonic mode-switch

```golang
module NH {
  task nh1 { ... }
  task nh2 { ... }

  start mode m1 [period = 6ms] {
    task [freq = 2] nh1;
    task [freq = 3] nh2;
    mode [freq = 6] if condition1 then m2;
  }
  mode m2 [period = 10ms] { ... }
}
```

Listing 3.4: Harmonic mode-switch

```golang
module H {
  task h3 { ... }
  task h4 { ... }

  start mode hs1 [period = 8ms] {
    task [freq = 8] h3;
    task [freq = 4] h4;
    mode [freq = 2] if condition2 then hs2;
  }
  mode hs2 [period = 100ms] { ... }
}
```

Examples  Consider the module NH from Listing 3.3 which contains two tasks: nh1 and nh2, with periods of 3ms, respectively 2ms. The mode switch frequency of six may imply a mode switch at any 1ms time instant, however, the only safe mode switch in TDL terms may happen only at the end of the mode period. In all other cases, the LETs of both tasks are broken. In the second example containing the module H (see Listing 3.4), we evaluate the mode switch condition twice per mode period. In both cases, the two tasks h3 and h4 have completed their LET at those moments; therefore, we can perform safe mode switches.

Greatest Common Divider
3.2. TDL COMPONENT MODEL

We regard implicitly the mode changes inside of modules as local. A TDL module may switch its mode of execution independent of the mode in which other modules currently execute. For example, in a GPS navigation application composed of the modules: GPS, routing, navigation, and display, the GPS module may switch its modes from init into acquiring or tracking modes, depending only in its input signal quality and independent of the state of the other modules.

In complex applications, there is a need to change the state of parts of the application, which may be implemented by several modules. TDL provides means for dynamical state changes within each application component through the import relationship. In such cases, we can implement partial or global mode changes within the application by introducing one or more switch-control modules. A switch-control module contains a task, which validates the global or partial mode-change condition. This task has as inputs the sensors or task output ports that denote the required mode switch. We import the control module and its task in all modules that have to perform the state transition. In each of these modules, we add a guard for a mode-switch, based on the task output from the control module. When the control task changes its output, all guards relying on this task propagate a mode switch in their corresponding modules. Although the import semantics make the control task output port available at the same time to all client modules, their individual mode change happens only when their guards watching the control task are executed and evaluate to TRUE (i.e., their current running modes can safely switch into new modes without breaking any running task’s LET).

Formal definition. A task invocation adds two properties to the task \( \tau \): its period \( \pi_\tau \) within a mode, and a set of port mappings for its input and output ports. The period of the task invocation depends on the period of its encompassing mode: \( \pi_\tau = \Pi_m/\omega_\tau \). The set of port mappings for a task \( \tau \in \text{Tasks}[m] \), is a subset of the \( \Lambda \) domain, for which a driver \( f[d] \) is defined. In different modes \( m_1 \) and \( m_2 \) the period of a task invocation \( \pi_\tau \), and the port mappings for its input and output ports may be different.

We call a set of tasks as harmonic within a mode \( m \), if the following condition holds true: \( \forall \tau_i, \tau_j \in \text{Tasks}[m], \pi_{\tau_i} > \pi_{\tau_j}, \pi_{\tau_i} = k \cdot \pi_{\tau_j}, k \in \mathbb{N}^* \). In other words, we can express the period of any task of the mode as an integer multiple of another task in the mode (excepting the task with the smallest period). We regard a mode switch \( \eta \) as harmonic when it does not break the LET of any task \( \tau \in \text{Tasks}[m] \). Therefore, the period \( \pi_\eta \) of a harmonic mode switch \( \eta \) is bound to a set of values: \( \pi_\eta = k \cdot LCM(\pi_{\text{Tasks}[m]}), \pi_\eta \leq \Pi_m, k \in \mathbb{N}^* \). Note that the existence of harmonic mode switches within a mode is not conditioned by a harmonic task set (e.g., \( \pi_m = 12, \pi_{\tau_1} = 6, \pi_{\tau_2} = 2, \pi_\eta = 6 \)).

We can regard the frequency \( \omega_\tau \) of a task \( \tau \subset \text{Tasks}[m] \), as a divisor of the mode period \( \Pi_m \), which specifies that we invoke the task \( \tau \) a number of \( \omega_\tau \) times per mode period. Considering the mode \( m \), its set of tasks \( \text{Tasks}[m] \in \text{Tasks}[M] \), actuators

\[3\]GPS - Global Positioning System
Actuators\([m]\) ∈ Actuators\([M]\), and mode-switches ModeSWs\([m]\), the Equation (3.8) provides the relationship between the mode period \(\Pi_m\) and their periods \(\pi_\tau, \pi_\alpha, \pi_\eta\).

\[
\begin{align*}
\forall \tau \in \text{Tasks}[m], \quad & \Pi_m = \pi_\tau \cdot \omega_\tau \\
\forall \alpha \in \text{Actuators}[m], \quad & \Pi_m = \pi_\alpha \cdot \omega_\alpha \\
\forall \eta \in \text{ModeSWs}[m], \quad & \Pi_m = \pi_\chi \cdot \omega_\eta
\end{align*}
\]  
\[
\Rightarrow \Pi_m = k \cdot \text{LCM}(\pi_\tau, \pi_\alpha, \pi_\eta) ; k \in \mathbb{N}^* \quad (3.8)
\]

For given task, actuator and mode-switch periods, the Equation (3.8) shows that the period of an encompassing TDL mode cannot have arbitrary values, the smallest mode period being the least-common-multiple (LCM) of the periods of its activities.

We refer to the declaration of a TDL mode \(m\) within the module \(M\) as the tuple \((m, \Pi_m, \text{Tasks}[m], \text{Actuators}[m], \text{ModeSWs}[m])\), where \(\Pi_m\) represents the mode period (in microseconds), and \(\text{Tasks}[m] \subset \text{Tasks}[M]\) represents the set of tasks invoked in the mode \(m\). The set \(\text{Actuators}[m] \subset \text{Actuators}[M]\) represents the set of actuators updated by tasks within the mode \(m\). The set \(\text{ModeSWs}[m]\) represents the defined mode switches from the mode \(m\). The set \(\text{ModeSWs}[M]\) represents all mode switches defined in all modes of the module \(M\). The set of drivers \(d[m] \subset d[M]\) defines the functional bindings between the TDL entities within the mode \(m\). TDL restricts a module to have only one start mode \(m_s\), which denotes the initial state of the module at startup.

We define the synchronous operation of a mode switch \(\eta\) within the mode \(m\), as a state transition of the module \(M\) from the mode \(m\) to another mode \(m'\). The mode switch \(\eta\) has a period \(\pi_\eta = \Pi_m/\omega_\eta\), and refers to the target mode \(m' \subset \text{Modes}[M]\) of the module \(M\). The well-timed property of TDL modules is enforced via harmonic mode switches, that is \(\forall m \in \text{Modes}[M], \forall \tau \in \text{Tasks}[m], \forall \eta \in \text{ModeSWs}[m], \omega_\tau/\omega_\eta \in \mathbb{N}^*\). Optionally, we may refer a set of drivers \(d[\eta]\) to initialize the output ports of the target mode \(m'\).

### 3.2.9 Flow control - guards

TDL guards allow conditional execution of task instances, actuator setters, or mode switches. A guard is a Boolean function that takes as arguments sensor or task output ports, and returns one of the two possible values: TRUE or FALSE. During the program execution, if the guard function evaluates to TRUE, the run-time system performs the corresponding task or actuator update. Moreover, we can use a guard to change at run-time the current application state by switching between the available application modes. Such dynamical behavior enables at run-time a flexible response to changes in the environment.

As an improvement on the mode switching semantic of Giotto, we evaluate the switching guards in a new target mode only after at least one task is released (i.e., not at the switch time). This apparent restriction, takes into account the zero time abstraction for mode switches, and prevents the dangerous cyclical mode switches without any useful computation in between.
Formal definition. We define the abstract notion of TDL guard $g$, operating over an arbitrary number of output ports $p_{o1}, p_{o2}, \ldots, p_{on}$, via an external user-defined Boolean function: $u[g] : S[P_o[g], t] \rightarrow \mathbb{B}; t \in \mathbb{N}$. The function $u[g]$ performs a set of mathematical operations on the set of output ports $[P_o[g]]$, and returns one of the two possible Boolean values TRUE or FALSE, depending on its internal implementation and the values of the ports in the set at the moment $t$ of evaluation. We note with $\text{Guards}[m]$ the set of guards from a mode $m$.

In the example from Listing 3.2, the external function $\text{bigdev}$ checks whether the deviation between successive sensor readings is higher than the predefined constant value $\text{max}$. In such case, it returns the value TRUE, which triggers a mode switch into a faster analysis mode. In this new state, the $\text{filter}$ task processes the sensor values twice as fast as in the initial mode of the module to compensate for a transitory state of the high-pressure valve, such as sudden pressure loss. The corresponding function $\text{smalldev}$, triggers a return into the initial state of the module, as soon as the successive sensor readings are within the predefined safety limits. The dynamic behavior of the module is still deterministic in the sense that we allow state changes only at the end of the mode periods. The mode switch frequency is one in both modes; therefore, we evaluate the guard conditions every 10ms.

3.3 Component distribution

Complex real-time computational systems can be composed of several processing nodes, each with its own array of sensors and actuators, and connected via a real-time communication system. Such systems, typically execute large and sophisticated controlling applications. The traditional development cycle of the applications themselves is a bottom-up approach, which starts with the exact configuration of the system in mind: from low level details such as network frame size, the allocation of frames to nodes, and a fixed communication pattern. Following is the description of each frame’s content, and only then, the developers start programming the applications. The result is a rigid system with high maintenance costs, and difficult debugging. Most changes in the network infrastructure or the processing power of individual nodes lead to changes in the overall application design or complete reimplementations of parts of it. A good step towards keeping the costs of testing, debugging, and maintenance to a reasonably low level is to use a component model for developing complex distributed applications.

3.3.1 Distribution unit

The original idea of handling distributed systems with Giotto was to distribute tasks on the processing nodes. However, this idea cannot cope with complex applications where not only processing power but also I/O resources are localized, and the number of tasks in a system is high. Without a clear separation of the logical parts of an application to corresponding nodes, even a theoretically functional system is unmanageable.
Starting with the component model of TDL based on *modules*, we can decompose complex applications into individual logical and functional parts. Therefore, we consider the TDL module as the *unit of distribution*. We can only distribute full modules and not individual tasks. For simple applications requiring just pure processing power, we can still make use of the Giotto initial approach by wrapping a task within a module with a corresponding mode.

TDL is platform independent and its import relationship does not specify where exactly each module is located in a system, or the architecture of the system. Therefore, using the TDL module as a logical unit, which may be distributed, we can design complex application without a specific target platform in mind. This feature allows independent development of modules (even by third parties) that can be integrated to compose a sophisticated application. Only in the integration phase of the modules into an application, we specify the target system architecture, such as a single node system or a distributed system.

![Figure 3.5: Multimeter application](image)

**Example.** We consider a simple embedded application for electrical measurements and model it using TDL. We construct first a TDL module for voltage measurements, which consists of two sensors (i.e., the measurement probe and the operating mode switch), and one task, which computes the voltage difference between the two contacts of the probe. Depending on the operating mode of the voltmeter, we define three TDL modes, as depicted in the Figure 3.5. The first state of the module upon startup is the *calibration* mode, where it compares the voltage difference at the probe tips with the current voltage range and retains an internal scaling ratio. Depending on the operating-mode switch position, the module switches into an AC or DC measurement modes, each with a different period. We call the voltage measurement task in each mode with different frequencies to compensate for transitory states where voltage peaks from electrical noise could influence the output values of the task. In the case when the measured voltage exceeds a predefined limit, we switch the mode of the module into the initial calibration mode. The TDL source code for the example is available in Listing 3.5.
In the following Listing 3.6, we add to our application its main module Multimeter, which imports the MeasureVoltage module and two other modules for configuration and
amperage measurement. The LEDArray type exported by the MyConfiguration module allows us to create a platform specific LED display, which we regard logically as an actuator. For the sake of simplicity, we have just one task, which encodes the voltage and amperage measurements passed at its input into a special buffer for the LED display. We use the import relationship semantics to access the Voltage task output from the remote MeasureVoltage module, without influence from its current mode of operation.

Listing 3.6: Sample TDL module importing remote resources

```tcl
module Multimeter {
    import MyConfiguration; // contains LEDArray type
    import MeasureVoltage as V1; // contains Voltage task
    import MeasureAmperage as A1; // contains Amps task

    actuator
        LEDArray display uses updateDisplay; // external C function

    task measurements {
        input int volts, amps;
        output LEDArray disp;
        uses measure(volts, amps, disp); // external C function
    }

    start mode m1 [period = 200ms] {
        task
            [freq=1] measure(V1.Voltage.o, A1.Amps.o); // we refer to remote tasks Voltage, Amps
        actuator
            [freq=20] display := measure.disp; // refresh display at 100Hz
    }
}
```

The two modules Multimeter and MeasureVoltage can be physically located on two different CPUs, more adequate for their purposes (e.g., one a general-purpose CPU and the other one a specialized micro-controller with integrated ADC\(^4\)). The TDL compiler and the corresponding runtime system must ensure that their logical interaction and overall behavior remains the same.

3.3.2 Transparent distribution concept

Traditional development of real-time applications treats the functionality distribution as the most important part of the application design. As hardware progresses at a faster pace than software development methodologies, we shift from the ”develop an application to work on top of existing distributed system” paradigm to ”for a given

\(^4\)ADC - Analog to Digital Converter
application use the right configuration of the system”. By splitting an application into
TDL modules and distributing them smarter, we make better use of available CPUs
and networking resources in a given distributed system.

We regard the distribution of the TDL modules composing an application in a
multi-node system as transparent distribution if the behavior of the application remains
identical regardless of the placement of the modules on nodes provided that there are
enough runtime resources available. Alternatively, the behavior of the system should
remain unchanged on a powerful single node system. This property can only be achieved
when the actual distribution of TDL modules does not interfere with their import
relationship, and their timing and functional properties.

Therefore, for a successful transparent distribution of the TDL modules of an ap-
plication, we use the following assumptions:

- the sensors and actuators are connected to the node that hosts the module that
  uses them
- each node has enough CPU performance to execute the modules assigned to it
- the time-triggered networking subsystem that connects the nodes has sufficient
  bandwidth and low latency to allow a time-safe exchange of values between the
  nodes that host modules with a import/export relationship.

With transparent distribution, the logical timing of all modules is always preserved,
only the physical timing, which is not observable from the outside, may be changed.
Secondly, for the developer of a TDL module, it does not matter where the module itself
and any imported modules are executed. The TDL tool chain and runtime system frees
the developer from the burden of explicitly specifying the communication requirements
of modules. Note that in both aspects transparency applies not only to the functional
but also to the temporal behavior of an application.

The advantage of transparent distribution for a developer is that the TDL modules
can be specified without having the execution on a potentially distributed platform in
mind. The mapping of modules to computation nodes is defined separately. Neverthe-
less, the functional and temporal behavior of a system is exactly the same no matter
where a component is executed.

The only place where distribution is visible is for the system integrator, who must
specify the module-to-node assignment by means of a configuration file, based on the
available processing nodes with their peripherals (e.g., directly connected sensors and
actuators), and network resources. This file is used as input for the TDL tool chain.
Figure 3.6 shows an example of a set of four TDL modules distributed across three
nodes.

Therefore, the transparent distribution mechanism has to abstract the communi-
cation and create a transparent interface to the remote modules in an import/export
relationship. The whole mechanism has to be also completely transparent to the de-
veloper of individual TDL modules.
It should be noted that Giotto considers individual tasks as the units of distribution. TDL, however, uses the higher-level construct of a module, which encapsulates a set of tasks, as the unit of distribution and thereby deviates from Giotto.

**Example.** To illustrate transparent distribution of TDL modules, we start with a subset of Figure 3.6. Let us consider modules M1 and M2, which are located on two different nodes. For the sake of simplicity, we assume that each module has a single mode of operation, which invokes a single task.\textit{task1} runs within module M1 and \textit{task2} runs within module M2 using as input the output of \textit{task1}. In this case, following the TDL semantics, module M2 has to import module M1, and \textit{task2} must have as input the output port of \textit{task1}. The arrow between the two tasks from the modules M1 and M2 in Figure 3.7 expresses this relationship.

For this example, we further assume that \textit{task2} runs twice as often as \textit{task1}, meaning that the LET of \textit{task1} is twice as large as the LET of \textit{task2}. Remember that the LET concept specifies that no matter when the task runs within its LET, the task gets its inputs at the beginning of LET and provides its outputs to other tasks or actuators only at the end of its LET. As a first step, Figure 3.8 shows a sample execution of the two tasks on a single node.

After \textit{task1} finishes its physical execution, the TDL run-time system buffers its output internally and provides it to \textit{task2} at the end of LET1. \textit{task2} reads its input at the beginning of the LET2, but the TDL run-time system schedules it for execution.
3.3. COMPONENT DISTRIBUTION

Figure 3.8: The execution of the two tasks from Figure 3.7 later. According to LET semantics, the first instance of task1 communicates its outputs to the third instance of task2 at the end of LET1, as the vertical arrow indicates.

Copying values from one location of memory to another takes close to zero time on a single node. In a distributed setting, however, there is a delay because communication takes much longer and typically only one node can send at a time. Figure 3.9 shows a sample communication pattern between the two tasks on different nodes. To implement this exchange of information between the two tasks, we need to add an auxiliary communication layer on both nodes that we call TDLComm. Its purpose is to send and receive messages at the right times.

Figure 3.9: The communication between the two tasks from Figure 3.7

To achieve our goal of transparent distribution, after task1 finishes, the system copies the internal output value to the TDLComm layer on node1 (comm1) that buffers it. Afterwards, comm1 sends the value in a packet at the time specified in the bus schedule, whereas the TDLComm layer from node2 (comm2) has to receive the packet and buffer it. We assume that a dedicated network controller performs the network operations in parallel with the execution of the tasks by the main CPU, which is the case in most systems. On node2, at the LET-end instant of task1, when the value
CHAPTER 3. THE TIMING DEFINITION LANGUAGE

should logically arrive, the system provides the value on the TDLComm layer. Clients of task1, such as task2, then use this value without making any difference between importing it locally or remotely.

The bus-schedule generation tool (detailed in Section 5.5) determines automatically the communication pattern and the maximum network load for a given set of modules and network properties. The resulting bus schedule is a statically defined table that specifies which node sends which package at which time. The table defines all network activities within one communication period (also named bus period), which is the least common multiple of all activity periods involved. A feasible schedule for the network activity guarantees the deterministic behavior of the application in the distributed setup. When the tool cannot find such a feasible schedule that maintains the properties of the single node system, the TDL compiler does not compile the application. It warns the system integrator that the current distribution of modules is not appropriate (strong coupling between components or insufficient network bandwidth).

We use a TDMA\(^5\) (Farcas et al., 2005b) approach to guarantee the timing of messages, meaning that any node is allowed to send messages in statically defined slots only. Furthermore, we implement the Producer-Consumer (i.e., Push) model, meaning that the tasks generating information, the producers, trigger the sending of a message. The consumers do not send any requests to the producers, as for example in the Client-Server model. In the previous example, the Push model avoids to resend messages without any value being changed.

3.3.3 Stub modules

The stub module concept is one of the possible ways of implementing the transparent distribution requirements. Previous attempts for distributing real-time functionality with Giotto were limited to trivial examples, and hard-coded interactions between functional parts, far from the transparency we aim at.

Given a set of modules bound by an import/export relationship, we regard the stub module of the service provider module, as an image of this module, with the same timing properties and public ports, but without its functionality. The service provider module executes actual tasks, which take non-negligible time, whereas the stub module does not have any task (i.e., only their public output ports); therefore, the stub has a minimal influence on the performance of the system. We can now split the problem of transparent distribution into two separate problems.

Local interaction. When the service provider module and the client module reside on two different nodes in a distributed system, we add on the node containing the client module the stub of the service module. Therefore, we substitute the interaction between the client and service modules with the interaction between the client and

\(^5\)Time Division Multiple Access
3.3. COMPONENT DISTRIBUTION

stub modules. As the stub has the same timing properties of the service module, we maintain the LET semantics on the node.

**Stub state synchronization.** As the service provider module and its stub module reside on two different nodes, we have to synchronize their states. This synchronization implies that their public output ports must have the same values at the same time instants. Moreover, both modules have to be in the same mode.

![Figure 3.10: Usage of module stubs to transparently distribute an application](image)

Consider the distributed real-time application depicted in the Figure 3.6, consisting of four modules, with their import relationships indicated by arrows. The module M1 from node 1 provides information from some of its public task output ports to the input ports of some tasks from the modules M2 and M4. Using the stub module concept, we create for the module M1 a stub M1-Stub, which resides on node 2 and node 3. We have now on the node 3 a direct, local interaction between the M1-Stub and M4, as if M1 was also running on node 3. However, the module M1 from node 1 has to push information to the node 3 to synchronize its state with M1-Stub. Similarly, we create on node 3 the stub M3-Stub of the module M3. The Figure 3.10 depicts the final configuration of modules on the three-node system.

Using transparent distribution, we solve an additional problem of too much functionality for one CPU. For example, assuming we have a distributed system consisting of two hardware-identical nodes, with a maximum accepted CPU utilization level of 70%, the first node containing two modules M1 using 20% of the CPU, and M2 using 30% of the CPU, and the second node with one module M3 using 30% of the CPU, we want to extend the application with another module M4 that requires 45% CPU time and lots of data from module M1. We cannot simply add the M4 module to the first node as it would increase its CPU load to 95%. We could however move the module M2 to the second node and put M4 in its place. The resulted system would have a 65% CPU utilization for the first node and 60% CPU utilization for the second node, well within our safety requirements.
3.4 Operational semantics

As TDL is a successor of Giotto, we present its operational semantics by following a similar approach (Henzinger et al., 2003; Horowitz, 2003). In this way a reader familiar with Giotto may easier spot the differences between the two languages and the improved TDL semantics. We analyze the TDL semantics incrementally, first with the case of a single module, followed by the case of multiple independent modules, and finalizing with the case of multiple modules with import relationships. Although in the case of a single module, a reader familiar with Giotto may believe the semantics of the module are the same as with a corresponding Giotto program, the mode switches and the implicit drivers of TDL change the so-called "Giotto micro-steps" (Horowitz, 2003) significantly.

3.4.1 Single module

The execution trace of a TDL module $M$ consists of an infinite sequence of configurations: $C_0, C_1, C_2, \ldots, C_n$. We define a configuration $C$ of a TDL module as the tuple $(t, m, t_m, Tasks[m], V[P])$, where $t$ represents the logical time at which the configuration $C$ is active. The mode $m$ represents the mode in which the module $M$ runs in this configuration; $t_m$ represents the time interval since the beginning of the period of the mode. The set $Tasks[m] \subset Tasks[m]$ represents the set of active tasks at the moment $t$, and the valuation function $V[P]$ provides the current values of the set of ports $P$ of the module $M$. In contrast with Giotto semantics, the time interval $t_m$ does not depend on other previous mode periods, and relates strictly to the current mode of operation $m$ of the configuration. The configuration $C_0$ represents the initial configuration of the module, $C_0 = (0, m_s, 0, \emptyset, \psi[P])$, where $m_s$ represents the start mode of the module $M$ and the set of ports $P$ has the initial values defined by the function $\psi$.

Every configuration contains a set of activities, software or environment related. The software activities consist of task invocations or completions, guard evaluations, and mode switches. The environment related activities consist of sensor readings and actuator updates. A TDL module evolves in time-triggered fashion from an initial configuration $C_0$ to any other configuration. The transition from one configuration to another takes a non-null discrete time interval.

We perform the following steps to proceed from an arbitrary configuration $C_{i-1}$ to its successive configuration $C_i$:

1. **Task completions** - We first perform the update of output and state ports for the set of completed tasks: $Tasks_{C_i} = \{\tau \mid \tau \in Tasks[m], t_m = k \cdot \Pi_m / \omega_\tau, \ k \in \mathbb{N}^\ast\}$. Consider a task $\tau \in Tasks_{C_i}$, and one of its output or state ports $p \in P_o[\tau] \cup P_s[\tau]$, we define its valuation $V_{C_i}(p) = u(V_{C_{i-1}}(P_o[\tau]), V_{C_{i-1}}(P_s[\tau]))$. For all other task ports the valuation function remains the same: $\forall p \in P_o[M] \setminus (P_o[\tau] \cup P_s[\tau]), \forall \tau \in P_s[\tau]$.
3.4. OPERATIONAL SEMANTICS

Tasks[M] \ Tasks_{C_{t_1}}. V_{C_t}(p) = V_{C_{t-1}}. We then remove the completed tasks from the set of active tasks of the configuration: Tasks_{a}[m] ← Tasks_{a}[m] \ Tasks_{C_{t}}.

2. Actuator updates - We evaluate the guards of the actuators updated in the mode $m$, execute the drivers to update their input ports, and then execute their setter functions to communicate the new values to the environment. Consider for the mode $m$ the set of actuators $\text{Actuators}[m] \in \text{Actuators}[M]$ updated $t_m = k \cdot \Pi_m / \omega_a$, $k \in \mathbb{N}^*$, the set of actuator guards $\text{Guards}_{a}[m] \in \text{Guards}[m]$, and the update drivers $d_a \in \text{Drivers}[m]$. We define for the input port $p_a$ of an actuator $\alpha \in \text{Actuators}[m]$ the valuation $V_{C_t}(p_a) = \{V_{C_{t-1}}(p_a) | \exists g_a \in \text{Guards}_{a}[m], g_a(\cdot) = \text{FALSE}\} \cup \{d_a(V_{C_t}(p)) | \exists p \in P_o[\tau] \cup P_o[M], \tau \in \text{Tasks}[M], \exists \Lambda(p) = p_a, (\exists g_a \in \text{Guards}_{a}[m]), g_a(\cdot) = \text{TRUE}\} \lor (\exists g_a \in \text{Guards}_{a}[m])\}$. For all other actuators $\alpha \in \text{Actuators}[M] \setminus \text{Actuators}[m]$, we retain the previous values of their input ports through the valuation $V_{C_t}(p_a) = V_{C_{t-1}}(p_a)$.

3. Sensor readings - We query the state of the environment through a sequence of sensor readings. For a sensor $\sigma \in \sigma[m]$ used as input port for an entity in the mode $m$, we define the valuation of its output port $V_{C_t}(p_{\sigma}) = u(\sigma)$, where $u(\sigma)$ represents the user-defined sensor getter function. All other sensors output ports retain their previous values: $\forall \sigma \in \text{Sensors}[M] \setminus \text{Sensors}[m], V_{C_t}(p_{\sigma}) = V_{C_{t-1}}(p_{\sigma})$.

4. Mode switches - If the mode time $t_m > 0$ and one of the guards for the mode switches $\text{ModeSWs}[m]$ defined in the mode $m$ evaluates to $\text{TRUE}$, we define the next operational mode $m'$ of the module, otherwise the module retains its current mode of operation $m$. In contrast with Giotto, which requires that only one guard may evaluate to $\text{TRUE}$ at any given moment, we evaluate the guards in the order in which they are defined in the TDL source of the module $M$, until one guard function returns $\text{TRUE}$ or there are no more guards to evaluate. Therefore, even if more guards may eventually evaluate to $\text{TRUE}$, we perform a deterministic mode switch as instructed by the developer through the sequence of mode switch conditions. TDL enforces harmonic mode switches; therefore, all requested mode switches must not break any active tasks LET: $\forall g_{\eta} \in \text{Guards}_{\eta}[m], t_m = k \cdot \Pi_m / \omega_{\eta}, \text{Tasks}_{\eta}[m] = \emptyset, k \in \mathbb{N}^*$, otherwise the TDL compiler issues an error message.

For the ordered list of guards $\text{Guards}_{\eta}[m] \subset \text{Guards}[m]$, depending on Boolean result of the evaluation of the guard function $u(g_{\eta})$, where $g_{\eta} \in \text{Guards}_{\eta}[m], t_m = k \cdot \Pi_m / \omega_{\eta}, k \in \mathbb{N}^*$, we have two possible execution paths:

$u(g_{\eta}) = \text{TRUE}$ - We execute the mode switch drivers and perform the mode switch into mode $m'$.

(a) Mode switch drivers - Consider a driver $d_{\eta}$ from the set of mode switch drivers $\text{Drivers}[\eta] \in \text{Drivers}[m]$ of the mode $m$, we define for its output port $p_o$ the valuation $V_{C_t}(p_o) = d(p_o)$, where $p_o$ is an input port from the
target mode \( m' \) and \( p_i \) is a task or sensor output port from the current mode \( m \). In contrast with Giotto, there are no active tasks, and all their values are already settled; therefore, all other ports retain their current values.

(b) New mode - We define the new configuration \( C_i \) through the tuple \((t, m', 0, \emptyset, V_{C_i})\) with the updated ports valuation. As an important semantic difference to Giotto, TDL mode switches start the target mode always at zero target mode time \( t_{m'} \), and with an empty set of active tasks.

\[
u(g_\eta) = \text{FALSE}\]

- We remain in the mode \( m \) and with the same mode time \( t_m \).

Note that as a difference to Giotto, there are no mode ports in TDL that require updates.

5. Task releases - We perform the new release cycle of the tasks of the mode \( m \), which have LET starting at the moment \( t_m \). In the case when we did not change the mode at the previous step, we have to release new task instances from the same task set. When at previous step we changed the mode of the module into \( m' \), the set of tasks to release is a subset of the set of tasks of the mode \( \text{Tasks}[m] \) that have the release time 0. For all tasks \( \tau \in \text{Tasks}_{\eta_i} \), \( \text{Tasks}_{\eta_i} = \{ \tau \mid \tau \in \text{Tasks}_a[m], t_m = k \cdot \Pi_m/\omega_\tau, k \in \mathbb{N}^+ \} \), possibly conditioned by the guards \( g_\tau \in \text{Guards}_\tau[m] \), we have two possible outcomes. If the guard condition of a task evaluate to \text{FALSE}, we skip the update of the task input ports and do not release a new instance of that task. Alternatively, if the guard condition evaluates to \text{TRUE} or there is no guard for the task \( \tau \in \text{Tasks}_{\eta_i} \), we perform the following two sub-steps:

(a) Update task input ports - For all task input ports \( p_i \in P_\tau \), we define the valuation \( V_{C_i}(p_i) = \{ d(V_{C_i}(p)) \mid \exists p \in P_\rho[M], \exists \Lambda(p) = p_i \} \), where \( d(V_{C_i}(p)) \) represents the release driver of the task \( \tau \) operating with the value of the port \( p \) of another entity at the configuration \( C_i \).

(b) Release task instance - We release the task \( \tau \) by adding it to the set of active tasks of the configuration \( C_i \), that is \( \text{Tasks}_a[m] \leftarrow \text{Tasks}_a[m] \cup \tau \), when the module is in the mode \( m \), respectively \( \text{Tasks}_a[m'] \leftarrow \text{Tasks}_a[m'] \cup \tau \), when the module changed into the new mode \( m' \).

6. Advance Logical Time - We define the minimum time interval for logical activities in the mode \( m \) as \( \delta_{min} = \text{GCD}(\Pi_m/\omega_a) \), where \( a \) represents an activity in the mode \( m \), such as task invocation, actuator update, or mode switch. We have a similar computation when in the current configuration \( C_i \) the module changed state into the module \( m' \). If the mode time has reached the end of the mode period \( t_m = \Pi_m \), we reset it \( t_m \leftarrow 0 \). Otherwise, consider the moment \( t'_m > t_m, t'_m = k \cdot \delta_{min}, k \in \mathbb{N}^* \), where there is an activity to perform, that is \( \exists \tau \in \text{Tasks}[m], t'_m = \).
3.4. OPERATIONAL SEMANTICS

\[ k' \cdot \Pi_m/\omega_p \lor (\exists \alpha \in \text{Actuators}[m], t_{m} = k' \cdot \Pi_m/\omega_p) \lor (\exists \eta \in \text{ModeSWs}[m], t_{m} = k' \cdot \Pi_m/\omega_p). \]

We compute the minimum time interval \( \delta = t'_{m} - t_{m} \), such that \( t_{m}' = k' \cdot \Pi_m/\omega_p \lor (\exists \alpha \in \text{Actuators}[m], t_{m}' = k' \cdot \Pi_m/\omega_p) \lor (\exists \eta \in \text{ModeSWs}[m], t_{m}' = k' \cdot \Pi_m/\omega_p) \). The minimum time interval \( \delta \) between the configuration \( c_{i} \) and \( c_{i+1} \) is computed as follows. We perform the next invocation of the module \( M \) that reaches the new configuration \( c_{i+1} \) at time \( t'_{m} \). We update the logical time \( t \leftarrow t + \delta \). In the time interval \( \delta \) the scheduler of the TDL runtime system may execute the set of active tasks.

Example

We take the TDL module from listing 3.7. It has four tasks (noted from A to D), and two operational modes \( m_1 \) and \( m_2 \), with the periods of 800ms, respectively 400ms. The module also contains two byte-sensors \( s_1, s_2 \), one float-sensor \( s_3 \), along with a byte-actuator \( a_1 \), and two float-actuators \( a_2, a_3 \), initialized with the constant zero. In the mode \( m_1 \), the sensor \( s_1 \) provides the input value to taskA every 200ms (four times per \( m_1 \) period). Every 400ms taskB reads its input values from the output of sensor \( s_2 \) and port \( o \) of taskA. We update the actuator \( a_1 \) with the output port \( o \) of taskB on each mode period. In the second mode \( m_2 \), the sensor \( s_1 \) provides the input value to another task taskD every 100ms. The two tasks can share the common resource \( s_1 \) because they are in different modes and the resource is read-only. Every period of \( m_2 \), we pass the output values of sensor \( s_3 \) and port \( o_2 \) of taskD to a new invocation of taskC. In the mode \( m_2 \), taskB has a higher invocation rate (i.e., every 200ms). Twice per mode \( m_2 \) period, we invoke taskB with input values from sensor \( s_2 \) and port \( o_1 \) of taskD. We assume that the float actuators require a settle time; therefore, we update them with twice the frequency of the task that provides their input. We update all actuators every 200ms: \( a_1 \) with the output of taskB, \( a_2 \) and \( a_3 \) with the output \( o_1 \), respectively \( o_2 \) of taskC. The module can switch from the mode \( m_1 \) to \( m_2 \) and back every 400ms, depending on the evaluation result of the corresponding guard conditions.

Listing 3.7: Four tasks example

```plaintext
module M {
    sensor
        byte s1 uses getSensor1; // out port s1, driver d[s1]
        byte s2 uses getSensor2; // out port s2, driver d[s2]
        float s3 uses getSensor3; // out port s3, driver d[s3]

    actuator
        int a1 := 0 uses setActuator1; // port a1, driver d[a1]
        float a2 := 0 uses setActuator2; // port a2, driver d[a2]
        float a3 := 0 uses setActuator3; // port a3, driver d[a3]

    task taskA {
        input byte i; // port taskA.i
        output byte o := 0; // port taskA.o
        uses uA(i,o);
    }
}
```
CHAPTER 3. THE TIMING DEFINITION LANGUAGE

\{
\}

\textbf{task} taskB { 
  \textbf{input} byte i1, i2; // port taskB.i1, taskB.i2
  \textbf{state} s; // port taskB.s
  \textbf{output} int o := 0; // port taskB.o
  \textbf{uses} uB(i1, i2, s, o);
}\}

\textbf{task} taskC { 
  \textbf{input} float f1, f2; // port taskC.f1, taskC.f2
  \textbf{output} float o1 := 0; float o2 := 0; // port taskC.o1, taskC.o2
  \textbf{uses} uC(f1, f2, o1, o2);
}\}

\textbf{task} taskD { 
  \textbf{input} byte i; // port taskD.i
  \textbf{state} s; // port taskD.s
  \textbf{output} byte o1 := 0; float o2 := 0; // port taskD.o1, taskD.o2
  \textbf{uses} uD(i, s, o);
}\}

\textbf{mode} m1 \textbf{[period = 800ms]} { 
  \textbf{task} \textbf{[freq = 4]} taskA(s1); // driver release d[taskA'], terminate d[taskA']
  \textbf{actuator} \textbf{[freq = 1]} a1 := taskB.o; // driver d[a1]
  \textbf{mode} \textbf{[freq = 2]} if toswitch(taskB.o) then m2; // d[\eta_{m1 \rightarrow m2}] = \emptyset
}\}

\textbf{mode} m2 \textbf{[period = 400ms]} { 
  \textbf{task} \textbf{[freq = 1]} taskC(s3, taskD.o2);
  \textbf{[freq = 2]} taskB(s2, taskA.o);
  \textbf{[freq = 4]} taskD(s1);
  \textbf{actuator} \textbf{[freq = 2]} a1 := taskB.o; // driver d[a1]
  \textbf{[freq = 2]} a2 := taskC.o1; // driver d[a2]
  \textbf{[freq = 2]} a3 := taskC.o2; // driver d[a3]
  \textbf{mode} \textbf{[freq = 1]} if toswitchback(taskB.o) then m1; // d[\eta_{m2 \rightarrow m1}] = \emptyset
}\}
We assume that in this example the guard conditions for the mode switches (i.e., toswitch and toswitchback) always evaluate to TRUE, resulting alternate mode switches from mode m1 to mode m2 and vice-versa. The module starts in the mode m1, from the configuration $c_0 = (0, m1, 0, \emptyset, \emptyset)$. There are no initializer functions, as all output ports of actuators and task are initialized with constant values.

$\begin{align*}
    c_0 & : (0, m1, 0, \emptyset, \emptyset) & \rightarrow & (0, m1, 0, \{\text{taskA, taskB}\}, \cdot) \\
    c_1 & : (200, m1, 200, \{\text{taskB}\}, \cdot) & \rightarrow & (200, m1, 200, \{\text{taskA, taskB}\}, \cdot) \\
    c_2 & : (400, m1, 400, \emptyset, \cdot) & \rightarrow & (400, m2, 0, \{\text{taskB, taskC, taskD}\}, \cdot) \\
    c_3 & : (500, m2, 100, \{\text{taskB, taskC}\}, \cdot) & \rightarrow & (500, m2, 100, \{\text{taskB, taskC, taskD}\}, \cdot) \\
    c_4 & : (600, m2, 200, \{\text{taskC}\}, \cdot) & \rightarrow & (600, m2, 200, \{\text{taskB, taskC, taskD}\}, \cdot) \\
    c_5 & : (700, m2, 300, \{\text{taskB, taskC}\}, \cdot) & \rightarrow & (700, m2, 300, \{\text{taskB, taskC, taskD}\}, \cdot) \\
    c_6 & : (800, m2, 400, \emptyset, \cdot) & \rightarrow & (800, m1, 0, \{\text{taskA, taskB}\}, \cdot) \\
\end{align*}$

Figure 3.11: Configuration trace for the execution of the TDL module from Listing 3.7

For the module M, consider its configuration trace $c_0, c_1, c_2, c_3, \ldots, c_6, \ldots$, from the Figure 3.11. We correlate it with the execution trace from Figure 3.12 using the logical time $t$. After initialization, at time $t = 0\text{ms}$, we update the configuration $c_0$, and execute the drivers for the sensor getters of s1 and s2. We then update the set of active tasks with new task instances for taskA and taskB. The constant zero from the TDL source code represents the initial value for the input port i2 of taskB. In the mode m1 the minimum time interval between activities is $\delta_{min} = 200\text{ms}$. We compute the time interval until the next activity $\delta$ as being equal with $\delta_{min}$ (the task set is harmonic).

Therefore, we proceed to the next configuration $c_1$, where at the logical time $t = 200\text{ms}$, the first instance of taskA is completed. We update its output ports with the results of its computation, and then read the sensor s1. We add a new instance of taskA to the set of active tasks, and recalculate $\delta = 200\text{ms}$ until the next configuration.

At logical time of 400ms, we reach the configuration $c_2$, where both tasks of mode m1 are completed and we update their output ports. Afterward, we evaluate the mode-switch condition toswitch, which returns the Boolean value of TRUE. In the middle of the mode m1, we perform a mode switch into the mode m2. In contrast with Giotto, where a mode switch jumps as close to the end of the target mode as possible, the TDL semantics require that we start the mode m2 from the beginning. Therefore, we do not perform the rest of the activities of the mode m1, and do not update the actuator a1. Instead we change the mode of the configuration, and start the execution of the activities from the mode m2 with $t_{m2} = 0\text{ms}$. We update the input ports of the tasks with the values of the sensor s1, s2, s3 output ports, and the initial values of taskD.o1, taskD.o2, taskC.o1, taskC.o2 output ports. Following we change the set of active tasks with new instances of taskB, taskC, and taskD. We compute the minimum time interval

75
between the activities in the mode \( m2 \) as \( \delta_{\text{min}} = 100\text{ms} \). The task set of mode \( m2 \) is also harmonic, therefore \( \delta = \delta_{\text{min}} \). The mode switch in the configuration \( C_2 \) is logically instantaneous.

After \( \delta = 100\text{ms} \), we reach the configuration \( C_3 \), where the first instance of \( \text{taskD} \) is completed. We update its state and output ports with the results of its calculations, and add a new instance of it to the set of active tasks. We proceed then to the next configuration after \( \delta = 100\text{ms} \).

At \( t = 600\text{ms} \) the module is in the configuration \( C_4 \), and the second instance of \( \text{taskD} \), and the first instance of \( \text{taskB} \) completed their computations. We update their output and state ports, and then update the actuator \( a1 \) with the latest value of the port \( \text{taskD} . o1 \), and the actuators \( a2, a3 \) with the initial values of the ports \( \text{taskC} . o1 \), respectively \( \text{taskC} . o2 \). We read the state of the environment via the sensors \( s1, s2 \), and update the input port of the third instance of \( \text{taskD} \) with the output port of \( s1 \). Then we update the input ports of the second instance of \( \text{taskB} \) with the output port of \( s2 \) and the result of the calculation from the output port \( o1 \) of the second instance of \( \text{taskD} \). We add these new task instances to the set of active tasks of the configuration and recalculate \( \delta = 100\text{ms} \).

We reach the configuration \( C_5 \) at mode time \( t_{m2} = 300\text{ms} \), and logical time \( t = 700\text{ms} \). At this moment, the third instance of \( \text{taskD} \) completed and we update its output and state ports. We add the fourth instance of \( \text{taskD} \) to the set of active tasks and proceed to the next configuration \( C_6 \) after \( \delta = 100\text{ms} \).

We complete the cycle of the mode \( m2 \), at logical time \( t = 800\text{ms} \). All tasks of the mode are completed and we update their output and state ports. Following we update the actuator \( a1 \) with the value of the port \( \text{taskB} . o \), and the actuators \( a2, a3 \) with the values from the ports \( o1 \), respectively \( o2 \) of \( \text{taskC} \). We evaluate the mode-switch condition \( \text{towithswitchback} \) and perform a mode switch back into the mode \( m1 \). We update the mode time \( t_{m1} = 0\text{ms} \), and change the mode of the configuration. We then start a new cycle of the mode \( m1 \) with new sensor readings on \( s1 \) and \( s2 \).

The module evolves through a deterministic set of configurations. In the previously presented example, the module alternates between the two modes resulting a simple periodic execution trace. However, in the general case, the module may evolve in a completely aperiodic way following one path from the possible execution graph, which depends on the Boolean result of the mode switch conditions.
Figure 3.12: Execution flow for the TDL module from Listing 3.7
3.4.2 Multiple independent modules

Consider a TDL application consisting of a set of modules, each encapsulating a part of the functionality of the application. The modules are independent in the sense that they do not import other modules, nor exchange information. In this case, we can consider each module independently, and handle them using the operational semantics of a single module. It may even be possible to execute the modules on independent nodes of a multi-node system. The runtime environment may consist in this case of multiple identical execution environments, which run independently one of the other. However, in this way we may loose the ”global picture” over the application behavior.

An alternate way of looking at this application is to consider the modules together and define the operational semantics of a group of modules, which may even belong to different applications. With this approach, we define in addition to the execution trace of a module consisting of successive configurations $C_0[M], C_1[M], \ldots, C_{nM}[M]$, the execution trace of the group of modules as a whole, through the set of configurations $\hat{C}_0, \hat{C}_1, \ldots, \hat{C}_n$.

We define the set $\text{Modules}[N]$ as the set of modules on a node $N$. We define a group configuration $\hat{C}$ as the tuple $(t, \text{Modes}[N], t_m[N], \text{Tasks}_a[N], V[P[N]])$, where $t$ represents the group logical time at which the configuration $\hat{C}$ is active, the set of modes $\text{Modes}[N]$ denotes the mode in which each module $M \in \text{Modules}[N]$ runs in this configuration at this logical time, $t_m[N]$ represents the time interval since the beginning of the period of each mode $m$ from each module $M$, the set of active tasks from all modules at the moment $t$, and the valuation function $V[P[N]]$ provides the current values of the sets of ports $P[N]$ of all modules running on the node. In the initial configuration $\hat{C}_0 = (0, m_s[N], 0, \emptyset, \psi[P[N]])$ all output ports are initialized, the global logical time is zero, and the modules begin with their start mode $m_s$.

The execution trace of a group of modules, represents the evolution of the group from a configuration $\hat{C}_{i-1}$ to a new configuration $\hat{C}_i$, following a similar number of steps as in the case of a single module. We consider each module independently and perform each step for all modules, then proceed to the next step. The configuration of each module $C_i[M]$ is updated after performing all six steps for all modules. The sixth step of advancing logical time is important for the evolution of the group as each module may have different timing requirements. However, we have only one group logical time $t$ that keeps track of the execution trace of the group. We compute the minimum time interval per module $\delta[M] = t'_m[M] - t_m[M]$, such that $\#t'_m[M] = k' \cdot \delta_{\text{min}}[M], t_m[M] < t''_m[M] < t'_m[M], k'' \in \mathbb{N}^*$ with the same properties as $t'_m[M]$. We have to update the configuration of the module $M$ after $t'_m[M]$, which means that $t[M] = t + t'_m[M]$. As there can be modules which require a sooner update of their configuration, we compute the minimum time interval of the group $\delta$ as the minimum of the time intervals of the modules $\delta[M]$, that is $\delta = \min(\delta[M]), \forall M \in \text{Modules}[N]$. We update the logical time of the group with $t \leftarrow t + \delta$. In the time interval $\delta$ between the configuration $\hat{C}_i$ and
the scheduler of the TDL runtime system may execute the set of active tasks from all modules. After the time interval $\delta$ elapses, we update the configuration of at least one module, and update the mode time $t_m$ of the modules that have not changed their internal configuration at this moment, and then proceed to a new group configuration.

### 3.4.3 Multiple modules with import relationships

In the case when the modules composing an application have import relationships and exchange values of output ports, the only option we have is to treat the modules as a group. The global logical time becomes extremely important, as all modules must run "in-sync".

Similarly as in the case of independent modules, we define the execution trace of the group as a sequence of group configurations $\hat{C} = (t, Modes[N, t_m[N], Tasks_a[N, V[P[N]]]$). However, the steps we have to perform to get from an initial configuration $\hat{C}_0 = (0, m_s[N], 0, \emptyset, \psi[P[N]])$, to an arbitrary configuration $\hat{C}_i$ are different. Considering the export function $\beta_M$ and the set of imported modules $\text{Imports}_M$ for a module $M$ of the group, we update the group configuration through the following steps:

1. **Task completions** - We first perform the update of output and state ports for the set of completed tasks from all modules: $\text{Tasks}_{\hat{C}_i} = \{ \tau \mid \tau \in \text{Tasks}_a[N], t_m[M] = k \cdot \Pi_m/\omega_\tau, m \in Modes[N], m \in m[M], k \in \mathbb{N}^* \}$. Note that $t_m[M]$ is relative to the beginning of the mode $m$ in which the module $M$ runs at the current group configuration. The group configuration may be updated with a faster rate than the configuration of the module $M$. Consider a task $\tau \in \text{Tasks}_{\hat{C}_i}$, and one of its output or state ports $p \in P_o[\tau] \cup P_s[\tau]$, we define its valuation $V_{\hat{C}_i}(p) = u(V_{\hat{C}_{i-1}}(P_o[\tau]), V_{\hat{C}_{i-1}}(P_s[\tau]))$. Here we abstract the LET of the task, as the valuation function is updated synchronously, and the value of the input ports of the task is unchanged since the release of the task at a previous group configuration $\hat{C}_{i-x}$. For all other task ports the valuation function remains the same: $\forall p \in P_{\tau}[\text{Modules}[N]] \setminus (P_o[\tau] \cup P_s[\tau]), \forall \tau \in \text{Tasks}_\text{Modules}[N] \setminus \text{Tasks}_{\hat{C}_i}, V_{\hat{C}_i}(p) = V_{\hat{C}_{i-1}}(p)$. We remove the completed tasks of all modules from the set of active tasks of the group configuration: $\text{Tasks}_a[N] \leftarrow \text{Tasks}_a[N] \setminus \text{Tasks}_{\hat{C}_i}$. The task output ports of each module of the group have now stable values, which may be exported to other modules or used internally by other tasks of the same module. It is important to perform this step for all modules, instead of executing all steps in sequence for all modules, as it avoids race conditions on reading the task output values between the modules that may import the same module. We discuss this problem in detail in the Section 4.2.3.

2. **Actuator updates** - We evaluate the guards of the actuators updated in each mode of each module whether they return a TRUE value. In such cases, we execute the drivers to update their input ports and then we execute the drivers of their setter functions. As the guards and the actuator update drivers may refer to task
CHAPTER 3. THE TIMING DEFINITION LANGUAGE

output ports from other modules imported by this module it is important that
the tasks from all modules have the correct output values. For each module,
the set of actuators updated in the mode \( m \in \text{Modes}[N] \) is \( \text{Actuators}[m] = \{ \alpha \mid t_m[M] = k \cdot \Pi_m/\omega_\alpha, \ m \in \text{Modes}[M], \ k \in \mathbb{N}^* \} \), with its corresponding guards and
actuator update and setter drivers. We define for the input port \( p_\alpha \) of an actuator
\( \alpha \in \text{Actuators}[m] \) the valuation \( V_{\widehat{C_\alpha}}(p_\alpha) = \{ V_{\widehat{C_{\alpha-1}}}(p_\alpha) \mid \exists g_\alpha \in \text{Guards}_\alpha[m], \ g_\alpha(\cdot) = \text{FALSE} \} \cup \{ d_\alpha(V_{\widehat{C_\alpha}}(p)) \mid \exists p \in P_o[\tau] \cup P_a[M] \cup P_\alpha[\text{Imports}[M]], \ \tau \in \text{Tasks}[M] \cup \\ \text{Tasks}[\text{Imports}[M]], \ \exists \Lambda(p) = p_\alpha, (\exists g_\alpha \in \text{Guards}_\alpha[m], \ g_\alpha(\cdot) = \text{TRUE}) \lor (\exists g_\alpha \in \text{Guards}_\alpha[m]) \} \). For all other actuators \( \alpha \in \text{Actuators}[M] \setminus \text{Actuators}[m] \), we retain the previous values of their input ports through the valuation \( V_{\widehat{C_\alpha}}(p_\alpha) = V_{\widehat{C_{\alpha-1}}}(p_\alpha) \). On a theoretical system, there is no jitter in the update of a subset of the actuators of all modules of the node, as all happen at the same group logical time \( t \), although each module may be in a different mode with a different relative mode time \( t_m \).

3. Sensor readings - At this step we query the state of the environment through
the sensors used as input ports for the entities available in the current mode of
each modules. We define the valuation of a sensor port from a module \( M \) at the current group configuration as \( V_{\widehat{C_\alpha}}(p_\alpha) = \{ u(\sigma) \mid \exists p \in P_1[\tau] \cup P_a[M] \cup P_\alpha[\text{Imports}[M]], \ \tau \in \text{Tasks}[M] \cup \text{Tasks}[M'], \ M \in \text{Imports}[M'], \ \exists \Lambda(p_\alpha) = p \} \), where \( u(\sigma) \) represents
the user-defined sensor getter function. All other sensors output ports from all
modules retain their previous values.

4. Mode switches - Although the modules are bound by import relationships and run ”in-sync” within the group, they may change their operational mode independent of other modules. If the mode time \( t_m > 0 \) and one of the guards for the harmonic mode switches \( \text{ModeSWs}[m] \) defined in the mode \( m \) of a module \( M \) evaluates to \( \text{TRUE} \), we define the next operational mode \( m' \) of the module, otherwise the module retains its current mode of operation \( m \). For the ordered list of guards \( \text{Guards}_\alpha[m] \subset \text{Guards}[m] \), depending on Boolean result of the evaluation of the guard function \( u(g_\eta) \), where \( g_\eta \in \text{Guards}_\eta[m], t_m = k \cdot \Pi_m/\omega_\eta, \ k \in \mathbb{N}^* \), we have two possible execution paths:

\[ u(g_\eta) = \text{TRUE} \ - \text{We execute the mode switch drivers and perform the mode}
\ - \text{switch into mode } m'. \]

(a) Mode switch drivers - We define for a driver \( \eta \in \text{Drivers}[m] \) the valuation \( V_{\widehat{C_\alpha}}(p_\alpha) = d(p_i) \), where \( p_\alpha \) is an input port from the target mode \( m' \)
and \( p_i \) is a task or sensor output port from the current mode \( m \).

(b) New mode - In the new group configuration \( \widehat{C_\eta} \), we update the module of
the modules as \( \text{Mode}[N] \leftarrow \text{Mode}[N] \setminus \{ m \} \cup \{ m' \} \), the mode time of
the module \( t_m \leftarrow 0 \), and the valuation \( V_{\widehat{C_\alpha}} \).

\[ u(g_\eta) = \text{FALSE} \ - \text{We remain in the mode } m \text{ and with the same mode time } t_m. \]
3.4. OPERATIONAL SEMANTICS

Note that a guard function from a module may use values of task output ports or sensors from other imported modules. Therefore, we can perform coordinated mode switches in multiple modules. Typical applications of this property are global or partial mode switches of the entire group of modules.

5. **Task releases** - For the modules in which the tasks have completed their period, we may perform a new task release cycle. For all tasks $\tau \in \text{Tasks}[\text{Modules}[N]]$, $\text{Tasks}_0 = \text{Tasks}_1 \cup \{\tau \mid \tau \in \text{Tasks}_a[m], t_m = k \cdot \Pi_m/\omega_r, m \in \text{Modes}[N], k \in \mathbb{N}^+\}$, if the guards $g_\tau \in \text{Guards}_r[m]$ evaluate to FALSE, we skip the update of the respective task’s input ports and do not release a new instance of that task. Alternatively, if the guard condition evaluates to TRUE or there is no guard, we perform the following two sub-steps:

(a) **Update task input ports** - For all task input ports $p_i \in P_i[\tau]$, we define the valuation $V_0(p_i) = \{d(V_0(p)) \mid \exists p \in P_o[M] \cup P_o[Imports[M]], \exists \Lambda(p) = p_i\}$, where $d(V_0(p))$ represents the release driver of the task $\tau$ operating with the value of the port $p$ of another entity from the module $M$ or its imported modules.

(b) **Release task instance** - We release the task $\tau$ by adding it to the set of active tasks of the configuration $\hat{C}_i$, that is $\text{Tasks}_a[N] \leftarrow \text{Tasks}_a[N] \cup \tau$.

6. **Advance Logical Time** - This step is identical with the sixth step performed for the independent modules. We calculate the minimum time interval $\delta_{\min}[M]$ for updating the configuration of each module, then their minimum $\delta = \min(\delta_{\min}[M])$, $\forall M \in \text{Modules}[N]$. We pass the set of active tasks to the scheduler of the runtime system, and sleep until $t = t + \delta$. We update the mode time of each module by subtracting the elapsed time $t_m[M] \leftarrow t_m[M] - \delta$, and proceed to a new group configuration.

In essence, for a set of modules running in the same TDL environment bound by import relationships, we perform each step for all modules, and then proceed to the next step. In this way, provided we maintain the semantics of the import relationships, the modules remain synchronized and produce deterministic results.

3.4.4 Distributed applications

The case of distributed applications is a particular case of the applications with independent modules or modules bound by import relationships. The difference consists of the practical implementation, which relies on a communication mechanism between the nodes of a distributed system to exchange the values between the modules on remote nodes.

Through the mechanism of transparent distribution and the concept of stub modules, TDL retains the operational semantics at the node level, with the stub module...
replacing the service provider module in the relationship with a client module. The remaining part is the synchronized execution of the service provider module and its stubs from remote nodes. Although this part does not change the operational semantics of the language, a practical implementation may fail to achieve the desired results and a deterministic behavior if it does not obey the following rules:

1. Time synchronization - When the group evolves from one configuration to another, effectively changing the group logical time $t$, all nodes must have the same notion of time $t$. In practice, this behavior may be impossible to obtain, as platform limitations may introduce clock jitter and other inaccuracies. The requirement in this sense is that the difference between an "official" value of the group logical time $t$ and any node’s internal value $t'[N]$ is orders of magnitude smaller than the GCD of all activities from all modes of all modules in the group.

2. Value synchronization - Along with the evolution in time of the group, its set of valuations must be consistent among all nodes running the group. In practice, this global valuation may require broadcasts of the updates on task or sensor output ports, at each group configuration.

Provided the two rules are always valid in a distributed system setup, and the system provides sufficient runtime resources, a group of modules may run deterministically on the system.
Chapter 4

The Virtual Machine

The Timing Definition Language described in the previous chapter represents the first step towards platform independent real-time components. However, we need to take the platform-independent timing specification of the application, formalize it and then compile it into machine-usable form. Afterward, we need to add its functionality code and a run-time environment for the target platform. For these additional steps, we rely on the initial idea presented in the Giotto: a virtual machine to handle the timing aspects of an application [Henzinger et al. 2003]. Giotto addresses the platform independence at the timing level (through E-Code) and we go one step further towards the portability at the integration, distribution, and implementation levels. We make the virtual-machine implementation and the bindings of module timing and functionality code portable by designing and implementing a Platform Abstraction Layer (detailed in Chapter 5) for abstracting from the execution platform, its underlying RTOS, internal resource management mechanisms, and scheduling scheme. The virtual machine together with the Platform Abstraction Layer lead to a portable real-time environment.

In this chapter, we first present the instruction set of our virtual machine, the so-called E-Code [Henzinger and Kirsch 2002]. We then introduce the algorithms for specifying the timing behavior of a TDL module through E-Code. We continue in Section 4.2 with the virtual machine itself (E-Machine) and its algorithms for executing the E-Code of one or more TDL modules. In other words, we present the run-time modular composition. We then show that regular E-Code is not composable, meaning that just the E-Code program is insufficient when there are data dependencies between components. Hence, we provide a solution to the current limitation of TDL’s E-Code with regard to modules with cyclic import dependencies. We complete this chapter with an analysis of the effect of transparent distribution over the operation of the E-Machine.
CHAPTER 4. THE VIRTUAL MACHINE

4.1 E-Code

The E-Code concept (Henzinger and Kirsch, 2002), a novel way of encapsulating the real-time behavior of an application, was first introduced as a compilation target for the Giotto language (Henzinger et al., 2003). Naturally, we reuse this flexible concept for encapsulating the timing information of TDL. Instead of a high-level description related to functional items (i.e., tasks) and their relationships, we have a near machine-level description of the activities expressing the LET semantics of TDL.

In this section, we extend the E-Code instruction set proposed by (Henzinger and Kirsch, 2002) to TDL with its semantics. The instruction set is platform independent and is interpreted at run-time by a virtual machine. We consider the E-Code as reactive code (Kirsch, 2002), which we execute in logical zero time, meaning that the run-time environment executes E-Code instructions in negligible time compared with tasks execution times. We then present an algorithm for encoding the timing behavior of a TDL module into a sequence of E-Code instructions. The algorithm produces optimized E-Code (for size and preemptions) and can handle both kinds of modules: normal and stub modules.

We recall from the previous chapter, the LET model on the fundamental scheduled element of TDL: the task as described in Section 3.1 and illustrated in Figure 3.1. The lifetime of a task instance has four main events; from the logical view, we have the release and terminate events. From the soft time point of view, we have the start and stop events, which are highly platform and application dependent.

We interact with the environment via the reactive elements (i.e., sensors and actuators). For example, the LET semantics of a task, which uses a sensor for its input, requires that we execute the code of the sensor getter function at the release event of the task. Similarly, we execute the code of the actuator setters at the terminate event of the task that provide its output value. We assume at this point that the run-time environment handles the correct execution of the task between these two events. In addition, we have to deal with the TDL guard functions, which may decide at run-time whether we release a task, update an actuator, or switch the state of a module (or of a subset of modules composing an application).

Extending this analysis over a full TDL mode period, we obtain a set of actions related only to environment time and a subset of the ports and drivers of the module containing this mode. The implicit drivers of TDL handle the port value-copying operations from the output ports of a set of TDL entities to the input ports of another set of TDL entities. For multi-modal applications, we have in addition mode switch actions, which determine the application state at run-time. To increase the portability of the TDL model of an application, instead of a typical table planning our run-time actions, we compile the model into a sequence of embedded instructions (i.e., E-Code), which encode the previously identified actions.
4.1. E-CODE

4.1.1 E-Code instruction set

We define a TDL E-Code instruction as the pair \((c, [args])\), where \(c\) represents a command in the set \(C\), and \([args]\) represents a set of arguments to the command \(c\). We define the following command set \(C = \{\text{CALL}, \text{RELEASE}, \text{FUTURE}, \text{IF}, \text{SWITCH}, \text{JUMP}, \text{RETURN}\}\).

The E-Code instruction set represents the set of all commands with their possible arguments. We use the notation: \(c(\text{arg}1, \text{arg}2, \ldots, \text{arg}_n)\) to denote the E-Code instruction with the command \(c\) and the set of arguments \([\text{args}] = (\text{arg}_1, \text{arg}_2, \ldots, \text{arg}_n)\). All E-Code instructions are synchronous, meaning that the E-Machine executes them in zero logical time.

A block of E-Code instructions represents a sequence of E-Code instructions (i.e., commands with their arguments), each instruction having a fixed offset from the beginning of the block. An E-Code program represents a sequence of E-Code blocks. Each E-Code instruction has a corresponding address \(a\) equal with its offset from the beginning of the E-Code program. The size of an E-Code program represents its number of E-Code instructions.

We can express the behavior of a TDL mode \(m\) with one or more E-Code blocks (i.e., a small E-Code program) depending on its constituents and their interactions. We define as E-Code of a TDL module \(M\) the reunion of all E-Code blocks for all modes of the module: \(E[M] = \bigcup_{m\in\text{Modes}[M]} E[m]\). Each E-Code block of each of its modes has a positive start address equal with its offset within the E-Code of the module.

The E-Code instructions have the following semantics:

- **CALL\((d)\)** instruction has the functional purpose of encapsulating the interaction between different TDL entities via drivers. Its argument represents a driver \(d \in D[M]\), which performs the actual port copying operation required by LET semantics.

- **RELEASE\((\tau, \text{deadline})\)** instruction performs the release action for a new instance of the task \(\tau\). A release action prepares the task instance \(\tau\) for execution; however, it does not imply an immediate dispatch operation. The scheduler of the run-time system maintains one or more queues of released task instances, and depending on a specific scheduling policy decides at run-time which task instance runs at a given moment on the CPU.

- **FUTURE\((a, t)\)** instructions plans a later execution of the E-Code block starting at the address \(a\), after the environment time \(t\) has elapsed. The address \(a\) belongs to the address space \(A = [1 \ldots ||E[M]||]\) of the E-Code program which contains this instruction. The time argument \(t\) represents a relative offset expressed in microseconds, from the moment when the E-Machine executes this instruction, until it executes the instructions beginning at address \(a\). In contrast with Giotto, the E-Code of TDL does not require a trigger binding and a trigger queue (Henzinger and Kirsch, 2002).
• \textbf{IF}(g, a_{\text{true}}, a_{\text{false}})\ instruction\ performs\ the\ evaluation\ of\ \textit{guard}~g.\ The\ two\ arguments\ \(a_{\text{true}}, a_{\text{false}}\)\ represent\ the\ addresses\ of\ two\ E-Code\ blocks\ for\ the\ two\ possible\ Boolean\ outcomes\ of\ the\ \textit{guard}\ function\ \(u[g]\).\ At\ run-time,\ when\ the\ guard\ condition\ evaluates\ to\ \textsc{true},\ the\ run-time\ environment\ that\ executes\ the\ current\ E-Code\ block,\ continues\ the\ execution\ starting\ from\ the\ \(a_{\text{true}}\ address.\ Otherwise,\ the\ next\ E-Code\ block\ to\ be\ executed\ starts\ at\ the\ address\ denoted\ by\ \(a_{\text{false}}\ address\ argument.\}

• \textbf{SWITCH}(\eta)\ instruction\ triggers\ a\ change\ in\ the\ state\ of\ the\ current\ module.\ The\ run-time\ environment\ continues\ the\ execution\ with\ the\ E-Code\ of\ the\ mode\ \(\eta.\ We\ regard\ the\ instruction\ as\ synchronous,\ with\ instant\ mode\ switch\ capability.\ However,\ the\ context\ in\ which\ this\ instruction\ appears\ has\ to\ conform\ to\ the\ TDL\ restriction\ regarding\ harmonic\ mode\ switches.\ The\ TDL\ compiler\ has\ to\ provide\ this\ instruction\ in\ the\ E-Code\ block\ of\ a\ module\ only\ in\ places\ where\ a\ mode\ switch\ can\ safely\ occur,\ and\ the\ run-time\ environment\ has\ to\ check\ and\ ensure\ its\ correct\ behavior\ at\ run-time.\}

Giotto\ performs\ mode\ switches\ through\ \textbf{JUMP}\ instructions,\ which\ hide\ the\ logical\ control\ flow\ of\ a\ Giotto\ program\ and\ the\ current\ execution\ mode.\ TDL\ introduces\ the\ \textbf{SWITCH}\ instruction\ because\ it\ provides\ additional\ support\ for\ distribution,\ by\ clearly\ marking\ the\ moment\ when\ a\ mode\ switch\ occurs.\ In\ this\ way,\ we\ can\ also\ inform\ other\ modules\ about\ the\ new\ state\ of\ the\ module.\}

The\ \textbf{SWITCH}\ instruction\ may\ also\ be\ used\ to\ switch\ into\ the\ same\ mode\ if\ we\ model\ the\ TDL\ mode\ as\ an\ automaton.\ In\ this\ way,\ the\ beginning\ of\ each\ mode\ is\ clearly\ defined\ and\ module\ state\ can\ easily\ be\ asserted\ by\ a\ distributed\ runtime\ environment.\}

• \textbf{JUMP}(a)\ instruction\ instructs\ the\ run-time\ environment\ to\ continue\ the\ execution\ of\ the\ E-Code\ instructions\ with\ the\ next\ address\ a.\ Although\ the\ instruction\ behaves\ similar\ with\ the\ classical\ "go\ to"\ instruction\ from\ the\ general\ purpose\ languages,\ a\ typical\ usage\ of\ this\ instruction\ is\ to\ implement\ loops,\ e.g.\ for\ mode\ periods.\}

• \textbf{RETURN}\ instruction\ terminates\ a\ block\ of\ E-Code.\ To\ continue\ the\ execution\ of\ its\ E-Code\ instructions,\ a\ TDL\ module\ has\ to\ provide\ a\ pair\ of\ future\ address,\ and\ time\ offset\ via\ a\ \textbf{FUTURE}\ instruction\ before\ the\ \textbf{RETURN}\ instruction.\ At\ run-time,\ except\ for\ the\ initialization\ phase,\ in\ the\ case\ when\ we\ have\ no\ future\ E-Code\ block\ planned,\ we\ stop\ the\ execution\ of\ the\ current\ module.\}

The\ E-Code\ instructions\ have\ well\ defined\ syntax\ and\ semantics;\ however,\ they\ are\ not\ strictly\ limited\ to\ the\ current\ semantics\ of\ TDL.\ The\ E-Code\ instructions\ provide\ increased\ flexibility\ in\ expressing\ arbitrary\ timing\ behavior\ over\ a\ classic\ table,\ and\ with\ their\ looping\ abilities\ reduce\ the\ overall\ length\ of\ the\ sequence\ of\ actions\ to\ be\ performed\ for\ each\ module.
4.1.2 Encoding TDL semantics into E-Code

We present one possible way of encoding the TDL semantics of a module M using E-Code instructions. We analyze both kinds of TDL modules: normal modules that contain tasks, sensors, actuators and have a defined functionality, and stub modules that represent an image (proxy) of a normal module located on a remote node of a distributed system. The E-Code of a stub module $M_{stub}$ is a subset of the E-Code of its original module $M$.

The algorithm presented in the Listing 4.1 produces E-Code for a given module $M$, using the formal abstractions of the TDL elements previously presented. The resulting E-Code program is optimized for size and minimal number of preemptions. For simplicity, we use the function $\text{emit}(c, [args])$ which produces the E-Code instruction $(c, [args])$, and increments the address counter $a$.

Listing 4.1: Algorithm for compiling a TDL module into E-Code

```plaintext
a ← 0 // initialization of the address counter
∀po ∈ P_o[M] : emit(CALL(d[ψ(po)])) // initialization of output ports
emit(RETURN) // module initialization is complete

∀m ∈ Modes[M]
start_m ← a // start address of mode m
Activities ← ∅ // set of actions indexed by time
∀τ ∈ Tasks[m] :
  Activities[t · (Π_m/ω_τ)] ← Activities[t · (Π_m/ω_τ)] ∪ τ^t // add release event of task τ
  Activities[(t+1) · (Π_m/ω_τ)] ← Activities[(t+1) · (Π_m/ω_τ)] ∪ τ^t // terminate event of τ
∀α ∈ Actuators[m] :
  Activities[t · (Π_m/ω_α)] ← Activities[t · (Π_m/ω_α)] ∪ α // add actuator update event
∀η ∈ ModeSWs[m] :
  Activities[t · (Π_m/ω_η)] ←Activities[t · (Π_m/ω_η)] ∪ η // add mode switch event

if (Activities = ∅)
  emit(RETURN) // no activities in the mode, therefore we stop the module
else
  for t ∈ Activities // at each time instant t perform the following actions
    ∀τ^t ∈ Activities(t) : emit(CALL(d[τ^t])) // terminate driver for task τ
    if (not STUB(M)) // the module M is not a stub
      // ACTUATORS
      ∀α ∈ Activities(t) :
        if (∃g[α]) // guards for actuator updates
          emit(IF(g[α], a + 1, a + 2))
        end if
      emit(CALL(d[α^t])) // update actuator driver
      emit(CALL(d[α])) // set actuator driver
```

87
CHAPTER 4. THE VIRTUAL MACHINE

// SENSORS
∀σ ∈ Sensors[M] : emit(CALL(d[σ | ∃pi = Λ(σ), pi ∈ Pτi[m] ∪ PModeSws[m]]))

// MODE SWITCHES
∀η ∈ Activities(t) : emit(IF(g[η], a + 1, a + 3)) ; emit(CALL(d[η])) ; emit(SWITCH(η))

// TASKS
∀τ′ ∈ Activities{t} :
    if (∃g[τ]) // guards for task releases
        emit(IF(g[τ], a + 1, a + 3))
    end if
    emit(CALL(d[τ'])) // update task input ports
    emit(RELEASE(τ)) // release task τ
end if

if (∃' ∈ Activities, t' > t) then
    δ ← t' - t // time interval until next moment t'
    emit(FUTURE(δ, a + 2))
    emit(RETURN)
else // t = Πm
    emit(JUMP(startm) / SWITCH(m)) // loop to the beginning of the mode m
end if
end for // loop to the next t in the list of activities Activities

We first generate the E-Code for the initialization of module M. We emit a CALL instruction for each entity whose output ports are initialized with external functions. In contrast with Giotto-generated E-Code that at this point would perform an immediate JUMP to the first address of the start mode, we terminate the sequence with a RETURN instruction. This approach allows us to initialize multiple modules that may run in parallel on a processing node. After the initialization of all modules is completed, we can start each module in its own mode. Thus, we obtain a synchronized startup of an application composed of multiple modules.

For each mode m of the module M we retain the starting address startm of its E-Code sequence. We later reuse this information to perform mode switches into this mode or to loop back to the beginning of the mode’s period. We create a set of activities Activities[], which defines the semantics of the TDL module M.

The subset Activities{t} represents the set of activities to perform at time t. For each task τ in the task set Tasks[m] of the mode m, we add the release marker τr to the activities set Activities{t} related to its release events (i.e., when t is a multiple of its period πτ). We also add the termination marker τt to the activities set Activities{t} where the task τ should logically terminate (i.e., beginning of a new task period). We continue by adding in the set of activities Activities{t} the updates of all actuators α which have t multiple of their periods πα. Assuming the mode switches defined within the mode m are harmonic, we add them to the set Activities{t} at the moments where
they could occur (i.e., when \( t \) is multiple of their period \( \pi \eta \)). As a result of these steps, the set Activities\([ ]\) contains a description of the logical behavior of the mode \( m \).

Using the information encoded in the set Activities\([ ]\), we create a timeline ordered from 0 to \( \Pi_m \) out of the moments \( t \) defined in the set, that is \( t \in [0, \Pi_m], \exists \text{Activities}\{t\} \). We call the moment \( t_2 \) as a successor of the moment \( t_1 \) if \( t_1 < t_2 \) then \#t, \( t_1 < t < t_2 \); in other words, \( t_1 \) is right before \( t_2 \) in the timeline. In this sense, \( t_1 \) is in the past of \( t_2 \), whereas \( t_2 \) is in the future of \( t_1 \). For each moment \( t \) in the timeline, starting from 0, we emit the E-Code sequence that encodes the actions we have to perform at that time instant. Therefore, we process first the task termination markers by CALL-ing their appropriate drivers. A driver in the form \( d[\tau] \) updates the task output port set \( P_o[\tau] \) with the results of the calculation \( u[\tau] \), meaning that it makes the results available to other TDL entities of the module \( M \). For a stub module there is no actual functionality \( u[\tau] \) to be executed nor actuators, sensors, or guards; therefore, we skip generating their corresponding E-Code.

For all other "normal" modules, we continue by issuing the CALL instructions for the set of actuators of the mode \( m \) updated at the moment \( t \). The drivers in the form \( d[\alpha] \) perform the value update on the actuator input ports, with the values from the output ports of the driving tasks. Afterward, the drivers execute the actuator setter functions, which provide the reaction of the computational system to the environment. Following the update of the actuators, we issue the CALL instructions for the drivers of the sensor getter functions. We update the sensor output ports which are required as inputs into new task instances or guard conditions in the mode \( m \). For each of the mode-switches defined in the mode, we emit a sequence of three E-Code instructions: an IF with the guard function \( u[g] \), a CALL to the optional driver \( d[\eta] \) which performs the update of output ports in the target mode \( m' \), and the actual SWITCH instruction to switch in the new mode \( m' \). A TRUE evaluation of the guard condition implies the execution of the CALL instruction and then the mode switch, whereas a FALSE result skips the two following instructions and continues with other instructions in the E-Code of the mode \( m \). At this point it is safe to proceed with task releases; therefore, we optionally emit an IF instruction for the guard of each task, followed by the CALL to the release driver \( d[\tau^r] \), which copies the output ports of other tasks or sensors to the input ports of the task \( \tau \), and the actual RELEASE instruction, which releases the task \( \tau \) to the scheduler of the runtime system for execution.

If we have a successor of the time instance \( t \) on the timeline, we compute the time interval \( \delta \) between \( t \) and its successor. We issue a FUTURE instruction planning the execution of the following instructions after the time \( \delta \) is elapsed at run-time, and then complete the block of E-Code instructions with a RETURN command. In the case when we have no successor for \( t \) (i.e., we are at the end of the mode \( t = \Pi_m \)), after all tasks are terminated and we completed all other activities, we loop back to the beginning of the E-Code block of the mode \( m \) via a JUMP or SWITCH instruction. The usage of a SWITCH instruction to complete the mode is useful in distributed systems, or in systems with cyclic dependencies between modules (see Sections 4.2.3 and 5.5). This
final instruction bounds the E-Code sequence of the mode \( m \). The algorithm produces a finite sequence of E-Code instructions for any mode \( m \) of the module \( M \), therefore the E-Code program of the module \( E[M] \) is also finite.

**Example.** We compile the TDL module from Listing 4.2 which consists of two sensors \( s1 \) and \( s2 \), two tasks taskA and taskB, and one actuator \( a1 \). The module defines two modes: \( m1 \) with the period of 20ms, and \( m2 \) with the period of 100ms. The first mode \( m1 \) contains the two tasks, with their relative frequencies \( \omega_{\text{taskA}} = 1 \) and \( \omega_{\text{taskB}} = 2 \), whereas the second mode \( m2 \) contains no task (i.e., stop mode). The mode switch \( \eta \) from mode \( m1 \) to mode \( m2 \) using the guard toswitch has a frequency \( \omega_\eta = 1 \) in relation with the period \( \Pi_{m1} \) of the mode \( m1 \).

Listing 4.2: Sample TDL source for compilation: module \( M \) with two tasks

```plaintext
module M {
  sensor
    byte s1 uses getSensor1; // port s1.out, driver d[s1]
    byte s2 uses getSensor2; // port s2.out, driver d[s2]
  actuator byte a1 := 0 uses setActuator; // port a1.out, driver d[a1], init driver d[a1\psi]

  task taskA [wcet = 1ms] { // annotation of task’s wcet
    input byte i; // port taskA.IN.i
    output byte o := 0; // port taskA.o.out, init driver d[taskA\psi]
    uses uA(i,o);
  }

  task taskB [wcet = 1ms] { // annotation of task’s wcet
    input byte i1; byte i2; // port taskB.I1.in, taskB.I2.in
    output int o := 0; // port taskB.o.out, init driver d[taskB\psi]
    state byte s := 0; // port taskB.s
    uses uB(i1, i2, s, o);
  }

  start mode m1 [period = 20ms] {
    task
      [freq = 1] taskA(s1); // driver release d[taskA'], terminate d[taskA']
      [freq = 2] taskB(s2, taskA.o); // driver release d[taskB'], terminate d[taskB']
    actuator
      [freq = 2] a1 := taskB.o; // driver d[a1]
    mode
      [freq = 1] if toswitch(taskA.o) then m2; // d[\eta] = \emptyset
  }

  mode m2 [period = 100ms] { // dummy mode
  }
}
```
We construct first its initialization E-Code with just the CALL to the actuator setter driver and a RETURN instruction at address 0 (the output ports are already initialized with constants). We continue by creating a timeline of the activities to perform in the mode \( m_1 \), that are task releases, task terminations, actuator updates, and mode switches. The periods of the tasks in the mode \( m_1 \) are: \( \pi_{\text{task}_A} = \Pi_{m_1}/\omega_{\text{task}_A} = 20\text{ms}/1 = 20\text{ms} \), respectively \( \pi_{\text{task}_B} = \Pi_{m_1}/\omega_{\text{task}_B} = 20\text{ms}/2 = 10\text{ms} \). For \( \text{task}_A \) we add to the set Activities\[ ] the release activity at the beginning of the mode: Activities\{0ms\} = \{\text{task}_A^r\}, and the terminate activity at the end of the mode: Activities\{20ms\} = \{\text{task}_A^t\}.

The second task has twice the frequency of the first task; therefore, we add its release marker at 0ms: Activities\{0ms\} = \{\text{task}_A^r, \text{task}_B^r\}, release and terminate markers at 10ms Activities\{10ms\} = \{\text{task}_B^r, \text{task}_B^t\}, and terminate marker at 20ms Activities\{20ms\} = \{\text{task}_A^t, \text{task}_B^t\}. We continue by adding the actuator updates, which have a period of \( \pi_{a_1} = \Pi_{m_1}/\omega_{a_1} = 20\text{ms}/2 = 20\text{ms} \), to the sets Activities\{10ms\} = \{\text{task}_B^r, \text{task}_B^t, a_1\}, and Activities\{20ms\} = \{\text{task}_A^t, \text{task}_B^t, a_1\}. We finally add the mode switch \( \eta \), with the period of 20ms, to the end of the mode Activities\{20ms\} = \{\text{task}_A^t, \text{task}_B^t, a_1, \eta\}. The Figure 4.1 illustrates the resulted timeline with the time instants of 0ms, 10ms, and 20ms.

According to the timeline that captures the semantics of the mode \( m_1 \), we start to generate the E-Code of the mode, from the moment of 0ms and address 2. As there are no task terminations, we emit the CALL instructions for the sensor getter drivers of \( s_1 \) and \( s_2 \) to update their output ports \( s_{1\text{out}} \) and \( s_{2\text{out}} \). We then issue the release driver calls to update the task \( A \), and \( B \) input ports with the values of the sensor output ports. We add the task release instructions, compute \( \delta = 10\text{ms} - 0\text{ms} \), and issue the FUTURE instruction planning the execution after 10ms of the next block starting at address 10 (see Figure 4.2).

We continue with the E-Code block for the moment of 10ms, where task \( B \) terminates, and we update and set the actuator \( a_1 \) with its output port. We also have to read the sensor \( s_2 \) and to release a new instance of \( B \) using the last sensor reading. We then plan the execution of the block starting at address 18 after another 10ms. The last E-Code block of mode \( m_1 \), contains the termination drivers of both tasks, the actuator
Initialization

00: CALL(d[a1u])
01: RETURN

Start mode m1 - 0ms
02: CALL(d[s1])
03: CALL(d[s2])
04: CALL(d[taskA])
05: RELEASE(taskA)
06: CALL(d[taskB])
07: RELEASE(taskB)
08: FUTURE(10ms, 10)
09: RETURN

Block for 10ms
10: CALL(d[taskB'])
11: CALL(d[a1u])
12: CALL(d[a1])
13: CALL(d[s2])
14: CALL(d[taskB'])
15: RELEASE(taskB)
16: FUTURE(10ms, 18)
17: RETURN
18: CALL(d[taskA'])
19: CALL(d[taskB'])
20: CALL(d[a1u])
21: CALL(d[a1])
22: IF(toswitch(taskA.o), 23, 25)
23: SWITCH(m2) - change mode
24: SWITCH(m2) - change mode
25: JUMP(2)

Block for 20ms
26: RETURN - stop

Figure 4.2: E-Code of the TDL module M from Listing 4.2

update and setter drivers, and the mode switch. We issue the IF instruction to trigger the guard invocation that conditions the mode switch. As there is no activity in the second mode of the module, we issue a single instruction at address 26: RETURN. The resulted E-Code of the module M has 27 instructions.

The E-Code is an abstract and compact way of expressing the TDL semantics of a module. By assigning an integer number to each driver, guard and task, and a binary opcode to each command from the E-Code instruction set, we obtain a binary representation of the E-Code program for a TDL module. Such binary E-Code still remains a platform independent representation of the original TDL module, provided that we use the same convention for encapsulating E-Code commands and arguments (i.e., byte ordering for numbers).

At run-time, we use a virtual machine that executes E-Code assuming that there are sufficient run-time resources (e.g., CPU and memory) to meet the real-time behavior requirements expressed by the E-Code and the accompanying functionality code.

The TDL compiler performs a time-safety checking with the ucet of tasks on that platform and verifies our assumption about the performance of the platform. The time-safety check provides the guarantees that the time and value determinism of the TDL module holds at run-time; in other words, there exists at least one mapping from the environment time to the software time, which maintains the LET semantics and the behavior specifications of the TDL module.
4.2 E-Machine

We retain the Giotto naming regarding the virtual machine that interprets the E-Code of a module, namely the Embedded machine or in short E-Machine; however, we adapt it to the semantics of TDL. In this section, we introduce the new algorithms with reduced run-time overhead for executing the E-Code of a TDL module. We further provide, the algorithms for run-time modular composition of TDL applications consisting of modules with import relationships. We finally present a solution to the problem of TDL modules with cyclic import dependencies.

We first define the computation of the E-Machine for a block of E-Code instructions as synchronous, which means that it takes logical zero-time. A practical implementation of such E-Machine has to measure the time spent with this computation and consider it for schedulability analysis.

Secondly, we require that the E-Machine has the potential to preempt the execution of TDL tasks and to execute the E-Code instructions at the logical times defined in the corresponding TDL module. However, this does not imply a truly preemptive run-time system, as in practice, the E-Machine could be implemented in an interrupt handler, or as part of the kernel of the operating system (Kirsch et al., 2005).

For a TDL module $M$, we define its E-Code program configuration at a time $t$ as $(E[M], i_p, f_a, f_t, Tasks_a[t], t)$, where $i_p$ represents an instruction pointer, $f_a$ and $f_t$ represent a possible future address, respectively relative time offset, and the set $Tasks_a[t]$ denotes the active tasks at the moment $t$. The instruction pointer $i_p$ denotes the address from which the E-Machine executes the next E-Code instruction of the module $M$. The future address $f_a$, represents an address in the address space defined by the E-Code of the module $E[M]$.

In contrast with Giotto, the E-Code generated for a TDL module following the algorithm in Listing 4.1 does not require at run-time a trigger queue, as the pair $(f_a, f_t)$ encoding the semantic of a single time-trigger suffices. This improvement leads to less run-time overhead in comparison with a Giotto program implementing a similar behavior.

We call a task $\tau \in Tasks_a$ as active at a moment $t$, if the E-Machine released the task before the time $t$, and the task did not complete until $t$. A task $\tau \in Tasks_a$ may be active at a moment $t$ but not actually running, because it could have been preempted by some other task. A task $\tau$ that terminates at a moment $t$ is automatically removed from the set of active tasks $Tasks_a[t]$.

Algorithm. In the algorithm from Listing 4.3 we present a way of executing the E-Code instructions of a TDL module, assuming that is was correctly generated with the algorithm from Listing 4.1. We also assume that the function $\text{FetchInstruction}$, returns the next E-Code instruction $e$ from the address $i_p$, and then increments the address $i_p$. An overflow of the address $i_p > a[M]$ is equivalent with an invalid address $i_p = \bot$. 

93
Listing 4.3: Algorithm for executing an E-Code instruction block of a TDL module

// t is the current logical time
// ip contains the beginning address of the current E-Code block

\[ f_t = \infty \] // reset future time
\[ f_a = \perp \] // reset future address

while (ip ≠ \perp) // until an invalid address is encountered = termination
  \[ e = \text{FetchInstruction}(E[M], ip) \] // returns the E-Code instruction of E[M] at address ip
  // and increments ip
if (e = CALL(d))
  \[ f[d] \] // executes the driver d functionality code
else if (e = RELEASE(τ))
  Tasks_a[t] ← Tasks_a[t] \cup τ // add the task τ to the active tasks set Tasks_a[t]
else if (e = FUTURE(a′, t′))
  \[ f_a ← a' \] // store the planned E-Code address a'
  \[ f_t ← t' \] // store the future relative logical time t'
else if (e = IF(g, atrue, afalse))
  if (u[g] = TRUE) // evaluate the guard condition u[g]
    \[ ip ← atrue \] // jump to the address atrue
  else
    \[ ip ← afalse \] // jump to the address afalse
  end if
else if (e = SWITCH(η))
  if (Tasks_a[t] = ∅) // only harmonic mode switches
    \[ ip ← \text{start}_{m'} \] // where start_{m'} is the start address of target mode m' of η
  end if
else if (e = JUMP(a'))
  \[ ip ← a' \] // unconditional jump to the address a
else if (e = RETURN)
  \[ ip ← \perp \] // terminate the execution of this E-Code block
end if

end while

return f_a, f_t // end of the algorithm

We interpret each E-Code instruction, until we encounter the termination mark \perp, either from an invalid address resulted from an overflow of ip or from the RETURN instruction. For a driver CALL, we execute its implementation. A RELEASE instruction adds the task τ from its argument to the set of active tasks Tasks_a[t] at the current logical time t. Depending on the Boolean result of the user implemented function u[g], a guard function g triggers an immediate change of the instruction pointer ip to one of the addresses atrue or afalse. A FUTURE instruction plans the execution of another E-Code block starting at the address f_a, at the moment t'' = t + f_t. For a mode switch η, we update the address ip with the starting address of the target mode m'. The only mode switches allowed are harmonic mode switches, which in the case of E-Code for a TDL module are equivalent with having no active tasks, meaning that we do not break any task’s LET. For loops we use the JUMP instruction, and update the address ip
with a new address \( a' \). Although we could use this instruction to express also mode changes, or the execution of other arbitrary E-Code instructions from other modes, the algorithm from Listing 4.1 prevents such usage that could alter the TDL semantics of the module’s E-Code and its reflection of a correct behavior. In addition, as the E-Code generated from a TDL module is finite for all its modes, we have a guaranteed termination of the interpreting loop with a \texttt{RETURN} instruction.

Figure 4.3: E-Machine interaction with E-Code, drivers, guards, and tasks

\textbf{Run-time information.} The interaction between E-Machine and the components of a TDL module such as drivers, guards, E-Code, and tasks is depicted in the Figure 4.3. We use dotted lines for data interactions or relationships and solid lines for control interactions. The E-Code provides the information for the timing interval \( f_t \), the sequence of drivers to execute, and the set of tasks to release. The drivers called by the E-Machine, perform the updates on the port values, execute the sensor getter and actuator setter functions, and pass the parameters to task implementations. The guards condition the execution flow of the module. The logical time always reflects the environment time and the deterministic interaction with the environment.

The algorithm from Listing 4.3 represents the core of the E-Machine, which performs the actual interpretation of E-Code instructions, but does not cover at this point any aspects related to the TDL component model. Therefore, to implement the semantics of TDL, the E-Machine retains for each module a set of run-time parameters \((i_p, f_a, f_t)\).
4.2.1 Execution of a TDL module

We use the algorithm from Listing 4.4 to execute a module on a platform. We perform first the initialization phase of the module until we reach a RETURN instruction. Afterward, we enter the main loop of the E-Machine, which we execute until we encounter a critical error or a user-defined flag shutdown becomes TRUE.

Listing 4.4: Algorithm for executing a TDL module through E-Code

// initialization phase
execute E[M] // run the E−Code block of the module M

\[ f_a \leftarrow \text{start}_m, \]
\[ f_t \leftarrow 0 \]
\[ t \leftarrow 0 \] // logical time \( t \) starts at 0

while (\neg \text{shutdown}) // main loop
   execute E[M] // use the Algorithm 4.3 for the block starting at \( f_a \) and update \( f_t \)
   invoke_scheduler(Tasks_{\text{a}}[M]) // pass to scheduler the list of active tasks
   sleep(f_t) // free CPU for user tasks until next E−action
   t \leftarrow t + f_t // update logical time
end while

// end of the algorithm

Let’s revisit the example from Listing 4.2, which we previously compiled into the E-Code from Figure 4.2. We first execute its initialization code, using the algorithm from Listing 4.3. Afterward, we lookup the start mode of the module \( M \), and pick its starting address of two, and simulate a FUTURE instruction at time 0. We initialize the logical time \( t \) and the elapsed time \( f_t \), and enter the main loop of the processing algorithm. As we have already planned the start of mode \( m1 \) at time 0, we rerun the E-Code interpreter for the first E-Code block of the mode \( m1 \). Here after getting the latest sensor values, we release the two tasks with their input ports updated with the sensor data. We plan to execute the next E-Code block from address 10, after 10ms and complete the interpreter algorithm. Back in the main loop, we pass to the scheduler the set of active tasks (\( A \) and \( B \)) and wait for 10ms until we have to execute the next E-Code block.

After the 10ms have elapsed, we update the logical time: \( t=10\text{ms} \). At this point, we expect that task \( B \) completed its execution and start again the E-Code interpreter for the E-Code block at the future address \( f_a = 10 \). We execute the task termination driver to update its output ports with the result of its functionality code calculations, and then call the drivers of the actuator to provide the response to the environment. As the execution time of the E-Code interpreter and the code in the main loop is orders of magnitude smaller than the execution time of the tasks, the time difference between the sensor readings and the actuator update is very close to the expected 10ms, with a minimal jitter. We perform a new reading on the sensor \( s2 \), and we update the input port of task \( B \) with its value. After we release a new instance of \( B \), we plan to
execute the E-Code block starting at address 18 after another 10ms, and complete the
equation of the E-Code interpreter. In the main loop, we invoke the system scheduler
to continue the execution of task A and the newly released task B, and then sleep 10ms.
At the moment of 20ms, we restart the interpreter for the E-Code block at address \( f_a = 18 \). We run the termination drivers of both tasks, and the actuator setter function.
Afterward, we evaluate the guard condition to switch based on the value of the output
port taskA.o. If the Boolean result of the guard condition is TRUE, we execute the
empty mode switch driver, and perform a mode switch to the starting address of mode
\( m_2 \). In mode \( m_2 \) the RETURN instruction terminates the interpretation of the E-Code
of the module \( M \), and the main loop waits forever \( t_f = \infty \). On the other hand, when
the guard condition evaluates to FALSE, we skip the two instructions and perform the
unconditional JUMP instruction to the beginning of the mode \( m_1 \), starting a new cycle.

The two algorithms from Listings 4.3 and 4.4 implement a basic E-Machine, capable
of executing a TDL module expressed in E-Code. The number of E-Code instructions
in the executed module, constitutes a run-time overhead, which in practice has to be
considered for correct scheduling of user-tasks. Nevertheless, the overhead grows only
linear with the number of E-Code instructions to execute at each logical time instance.

### 4.2.2 Modular composition

To execute one or more modules on a platform, we use the algorithm from Listing 4.5.
We first have to solve the dependencies between modules to cope with possible im-
port relationships. As TDL allows only the imports that conform to a directed acyclic
graph, we sort the modules according to their depth in the import tree from producer to consumer. Therefore, at any given logical time \( t \), the E-Code of a producer
module runs before the E-Code of any module that consumes one or more of its task
output ports. We define the function \( s \) that assigns to each producer module \( M \) an
index greater than any of its consumer modules: \( s: \text{Modules}[N] \rightarrow \mathbb{N}, s(M_1) = \{i \mid \nexists M_2 \in \text{Imports}[M_1], s(M_2) < i\} \), where \( \text{Modules}[N] \) represents
the set of modules on a node \( N \). Using the function \( s \) we order the producer-consumer set
\( \text{Modules}[N] \), where the independent modules can have any arbitrary index.

We first perform an initialization phase for all modules until we reach their RETURN
instruction. In this phase, no tasks from any module are released, only the output
ports are initialized by their corresponding initializer functions. Afterward, we enter
the main loop of the E-Machine, which we execute until we encounter a critical error
or a user-defined flag shutdown becomes TRUE.

Listing 4.5: Algorithm for modular composition through E-Code

```plaintext
// initialization phase
for i ← 0 to ∥\text{Modules}[N]∥
  // use the Algorithm 4.3 to initialize all modules
  execute E[\text{Modules},[N]] // run the E-Code block of the module \( \text{Modules},[N] \)
end for
```
CHAPTER 4. THE VIRTUAL MACHINE

// lookup start mode of each module and prepare modules for execution
for i ← 0 to ||Modules||
  // initialize fa of each module with corresponding start mode address
  f[ Modules[I] ] ← 0 // simulate a future instruction at time 0
end for

// initialize logical time
δ_min ← 0 // initialization of time offset until first E-action

while (∼shutdown) // main logical loop when E-Machine runs standalone
  // main block of the E-Machine
  for i ← 0 to ||Modules||
    if ( f[ Modules[I] ] = δ_min ) // we have to execute the E-Code of this module
      execute E[ Modules[I] ] // use the Algorithm 4.3 for the block starting at fa
      // and update ft, fa of the current module
    else // we don’t have to run the E-Code of this module yet
      δ_min ← f[ Modules[I] ] − δ_min // update elapsed time
    end if
  end for

  δ_min = ∞ // time offset until next E-action from a module
  for i ← 0 to ||Modules||
    if ( f[ Modules[I] ] < δ_min ) // get the minimum future time from all modules
      δ_min ← f[ Modules[I] ]
    end if
  end for

// main block of the E-Machine ends here

// at this point we abstract the control of the scheduler and other runtime components
// and assume that the E-Machine operates standalone
pass_to_scheduler(Tasks[ M[I] ]) // the list of active tasks from all modules

// we abstract here the scheduling of tasks and the logical time update
sleep(δ_min) // free CPU for user tasks until next E-action
  t ← t + δ_min // update logical time
end while
// end of the algorithm

As a difference to the algorithm in the Listing 4.4 where the logical time t was tightly correlated with the time of the module, in the algorithm from Listing 4.5 we have to deal with multiple parallel FUTURE instructions that set arbitrary points in time to continue their execution. One solution would be to store an absolute logical time
4.2. E-MACHINE

for each module. However, this solution is impractical for memory and performance reasons. A better alternative implemented by the aforementioned algorithm updates the relative time \( f_t \) of each module in relation with the unique logical time \( t \).

After we complete the initialization phase of all modules, we start processing the modules in the order of their import relationships. At the moment \( t = 0 \), we execute for all runnable modules their E-Code and update their future time and address parameters \( (f_t, f_a) \). We then compute the minimum waiting interval \( \delta_{\text{min}} \) until we have to execute the E-Code of one or more modules. This interval is given by the smallest argument \( f_t \) of a \texttt{FUTURE} instruction from all modules. At this point, we assume that the E-Machine operates standalone in parallel with the TDL Scheduler. We detail the interaction between the E-Machine, TDL Scheduler, and the other components of the TDL runtime environment in Section 5.4.

We pass to the TDL Scheduler the set of active tasks from all modules \( \text{Tasks}_{\text{a}}[M[N]] \), and sleep \( \delta_{\text{min}} \) time units (until \( t + \delta_{\text{min}} \)). At this moment, we begin a new cycle and run the E-Code of the modules that have the \( f_t \) equal with the waiting interval \( \delta_{\text{min}} \) we had in the previous cycle. All other modules have their \( f_t \) parameter updated with the elapsed time \( \delta_{\text{min}} \). The complexity of the algorithm is in \( O(\|\text{Modules}[N]\|) \) range, where \( \|\text{Modules}[N]\| \) represents the number of modules on the node \( N \).

4.2.3 Cyclic dependencies

Contrary to previous statements (Henzinger and Kirsch, 2002) a pure E-Code program is not compositional when there are data dependencies between components (with or without cyclic dependencies). The problem exists for all languages based on E-Code starting with Giotto, although Giotto does not have a component model and there cannot be external data dependencies between tasks. Consequently, in the current version, TDL does not allow cyclic dependencies between modules. The main reason is that at run-time, the order in which the E-Machine executes the E-Code of all modules influences the results of the tasks that exchange values across module boundaries.

As an example consider three modules \( M_1, M_2, \) and \( M_3 \), each with one mode and one task. Both modules \( M_1 \) and \( M_3 \) import the module \( M_2 \). If we run the modules in a sequence \( M_1-M_2-M_3 \), the modules \( M_1 \) and \( M_3 \) would operate with different values, because the E-Code of \( M_2 \) calls the task termination drivers after \( M_1 \) and before \( M_3 \).

Furthermore, consider that the modules \( M_1 \) and \( M_3 \) have a cyclic dependency. The task from \( M_1 \) provides as output the double of its input value, whereas the task from \( M_3 \) provides as output a value three times bigger than its input value. The output of the task from \( M_1 \) goes to the input of the task from \( M_3 \) and vice versa. Both tasks have their outputs initialized with the value of one. Following the algorithm from Listing 4.5, we initialize first the modules, then proceed with the \( M_1 \). We release the its task, then continue with the module \( M_2 \), and finally release the task of \( M_3 \). After the end of their period, we execute the E-Code of the \( M_1 \), update the visible task output port of the task with the value of two, and release a new task instance with the input value from
the task of M3, which yet is unchanged (i.e., one). We then execute the E-Code of the M2 and M3, update the visible task output port of the M3’s task with the value of three and release a new instance with the input value from the M1’s task (i.e., two). The second instances of both tasks have different input values than assumed: the task of M1 started with an input value of one instead of the expected value of three.

For TDL the problem does not reside in the LET semantics or the TDL component model, but in the E-Code instruction set, and the way we generate and execute the E-Code of multiple modules. Without cyclical dependencies as is the case of the current TDL implementation, a simple sorting of the list of modules suffices to avoid this problem. With cyclical dependencies, it is impossible to sort the modules according to their dependencies and we have to extend the semantics of the E-Code. We have to decouple the termination of a task instance from the release of a new task instance. We then execute the E-Code for updating the visible task output ports from all modules and afterward the remaining E-Code to update actuator ports, perform sensor readings, and release the new task instances. This two-step approach may have several implementation options, which we describe in the following paragraphs along with their advantages or disadvantages.

**Future+Return.** We change the algorithm from Listing 4.1 as depicted in the Listing 4.6. We insert two instructions immediately after the CALL instructions for the task termination drivers: a FUTURE to the current logical time \( t \) and a RETURN. Therefore, for a module \( M \) the E-Machine executes the termination drivers of the tasks completed at moment \( t \), then terminates the execution of the module \( M \) and starts processing the E-Code of another module \( M' \). As the future time \( f_t \) argument of the inserted FUTURE instruction from module \( M \) is equal with the current time \( t \), the E-Machine appends the module \( M \) to the list of modules to process. After it processed all modules that have logical activities at the current moment \( t \), the E-Machine executes again the module \( M \) from the next instruction following the inserted RETURN instruction; therefore, it starts a new release cycle of the module \( M \).

Listing 4.6: Updated algorithm for compiling a TDL module into E-Code with Future+Return solution for cyclical dependencies

```plaintext
... if (Activities = ∅)
   emit(RETURN) // no activities in the mode, therefore we stop the module
else
   for t ∈ Activities // at each time instant t perform the following actions
      ∀τ ∈ Activities[t] : emit(CALL(d[τt])) // terminate driver for task τ
      // split E-Code processing into two phases
      emit(FUTURE(t, a + 2))
      emit(RETURN)
      // E-Code processing continues from here after all modules have completed their
      // update of visible task output ports
```
4.2. E-MACHINE

if (not STUB(M)) // the module M is not a stub
    \( \forall \alpha \in \text{Activities}\{t\} : \)

[...]

The corresponding change of the E-Code processing algorithm from Listing 4.5 is available in the Listing 4.7. We execute first all modules that have a set of activities at the current moment \( t \) and update the logical time for the other modules. We then reprocess only the modules that have the additional future instructions. These modules update the relative time offset \( f_t \) with the value of zero, denoting the current time \( t \). The rest of the algorithm from Listing 4.5 remains unchanged and afterward we continue with the calculation of the waiting period \( \delta_{\text{min}} \).

Listing 4.7: Updated algorithm for executing one or more TDL modules using E-Code with Future+Return solution for cyclical dependencies

[...]
while (\( \neg \) shutdown) // main loop
    for \( i \leftarrow 0 \) to \( \|\text{Modules}[N]\| \)
        if (\( f_t[\text{Modules}[N]] = \delta_{\text{min}} \)) // we have to execute the E-Code of this module
            \( i_p[\text{Modules}[N]] \leftarrow f_a[\text{Modules}[N]] \) // set entry point for E-Code block
            execute \( E[\text{Modules}[N]] \) // use the Algorithm 4.3 for the block starting at \( f_a \)
        else // we don’t have to run the E-Code of this module yet
            \( f_t[\text{Modules}[N]] \leftarrow f_t[\text{Modules}[N]] - \delta_{\text{min}} \) // update elapsed time
        end if
    end for
    for \( i \leftarrow 0 \) to \( \|\text{Modules}[N]\| \)
        if (\( f_t[\text{Modules}[N]] = 0 \)) // we have to continue the E-Code of this module
            \( i_p[\text{Modules}[N]] \leftarrow f_a[\text{Modules}[N]] \) // set entry point for E-Code block
            execute \( E[\text{Modules}[N]] \) // use the Algorithm 4.3 for the block starting at \( f_a \)
        end if
    end for
[...]

This solution requires small changes in the algorithms for generating and executing the E-Code of TDL modules. The drawback is that the resulting E-Code grows with two instructions for each moment within a mode period where one or more tasks terminate. This enlargement of the E-Code may impose problems on systems with limited memory.

**LET Delimiter.** A second solution is related to the **Future+Return** solution, and introduces an explicit delimiter for the ending of a LET period and the beginning of a new LET period. Instead of inserting two instructions, we insert just a new **LET** instruction, which has no functional purpose other than instructing the E-Machine about the completion of the task termination drivers. As an optimization, to retain the same implementation of the processing algorithm from Listing 4.7, we set the relative time offset \( f_t \) with the value of zero and the future address \( f_a \) to the address of the
next instruction following the LET instruction. The result is a smaller E-Code and the same simplicity in executing E-Code through the algorithm from Listing 4.4. The drawback consists in the extended instruction set and the additional decoding phase of this instruction on the algorithm from Listing 4.3.

**Alternate Call instruction for tasks.** We can further reduce the E-Code by using an alternate CALL instruction for the task termination drivers. A new instruction such as \texttt{TERMINATE}(d) has the same semantic of the \texttt{CALL}(d) instruction but applies only to task termination drivers. The resulted change of the E-Code generating algorithm from Listing 4.1 is available in the Listing 4.8.

Listing 4.8: Updated algorithm for compiling a TDL module into E-Code with alternate call for tasks solution for cyclical dependencies

```plaintext
if (Activities = ∅) emit(RETURN) // no activities in the mode, therefore we stop the module else for t ∈ Activities // at each time instant t perform the following actions ∀τ ∈ Activities\{t\} : emit(TERMINATE(d[τ])) // terminate driver for task τ if (not STUB(M)) // the module M is not a stub ∀α ∈ Activities\{t\} :
```

For this solution, we use a simplified version of the algorithm from Listing 4.3 to execute only the task termination drivers as depicted in the Listing 4.9. As soon as we encounter an instruction different from \texttt{TERMINATE}, we store the address of this instruction along with the current time offset and stop processing the current module.

Listing 4.9: Additional algorithm for executing task termination drivers for cyclical dependencies

```plaintext
while (i_p ≠ ⊥) // until an invalid address is encountered = termination e = FetchInstruction(E[M], i_p) // returns the E−Code instruction of E[M] at address i_p // and increments i_p if (e = TERMINATE(d)) f[d] // executes the terminate driver d else f_t = 0 // reset future time f_a ← i_p // store continuation address break; end while;
```

We update the algorithm from Listing 4.5 as depicted in the Listing 4.10 to execute the E-Code of multiple modules. We execute first the algorithm from Listing 4.9 for all modules that have logical activities at the current moment \(t\), and then we update the logical time of the other modules. Afterward, we execute the algorithm 4.3 for the
remainder part of the E-Code block containing the previously executed termination
drivers of each module.

Listing 4.10: Updated algorithm for executing one or more TDL modules using E-Code
with alternate task call solution for cyclical dependencies

```
[...]
while (~ shutdown) // main loop
  for i ← 0 to ∥Modules∥
    if (ft[Modules,i] = δ_min) // we have to execute the E-Code of this module
      ip[Modules,i] ← fa[Modules,i] // set entry point for E-Code block
      execute E[Modules,i] // Algorithm 4.9 for task termination drivers starting at fa
    else // we don’t have to run the E-Code of this module yet
      ft[Modules,i] ← ft[Modules,i] − δ_min // update elapsed time
    end if
  end for
  for i ← 0 to ∥Modules∥
    if (ft[Modules,i] = 0) // we have to continue the E-Code of this module
      ip[Modules,i] ← fa[Modules,i] // set entry point for E-Code block
      execute E[Modules,i] // Algorithm 4.3 the block after last termination driver
    end if
  end for
[...]
```

The size of the resulting E-Code program remains identical with the one provided
by the original algorithm from Listing 4.1. As a small drawback, the decoding phase
becomes more complicated because of the additional instruction. Therefore, we use the
executing algorithm from Listing 4.10.

**Additional parameter to the Call instruction.** The solution is an optimization
of the previous solution with an alternate CALL for task termination drivers, but instead
of a new instruction, we use an additional parameter for the CALL instruction. The
extended decoding phase in this case is limited to only the CALL instruction. We can
then run twice the same E-Code processing algorithm for the task-termination phase
of each module and then for their task-release phase.

We implemented this solution as it implies minor changes to the E-Machine and
allows backward compatibility with older implementations that cannot handle cycles.
Also the additional parameter to the CALL instruction does not affect other tools that
rely on E-Code such as the Bus Schedule Generator Tool and the TDL Decoder. They
simply ignore the parameter and work as before.

4.2.4 Transparent distribution

The concept of transparent distribution in real-time systems introduced with TDL, has
a minimal impact on the design and the mode of operation of the E-Machine. The
import relationships that form the dependency graph between modules and the mechanism of transparent distribution that makes possible the distribution of modules on different nodes is hidden within the Platform Abstraction Layer - TDLComm presented in the next chapter. The Bus Schedule Generator Tool computes the necessary communication pattern for the deterministic exchange of information through TDLComm. Therefore, at the logical level we have an uniform view over an application consisting of multiple modules.

From the point of view of the E-Machine, all modules regardless of their dependencies, such as provider or consumer, have similar runtime properties. All modules, including the stub modules have E-Code, which is the fundamental runtime element for the E-Machine. The difference that the stub modules do not release any tasks is of no concern to the overall execution flow of the E-Machine. Thus, the E-Machine treats all modules in the same way. The stub modules have the same timing requirements as the service provider modules they replace on remote nodes; thus, they issue the same sequence of FUTURE + RETURN instructions that affect their corresponding \( f_t \) and \( f_a \) run-time parameters.

The E-Machine and the TDLComm layer work in parallel using the same logical time. Consequently, all modules, regardless of their physical placement in a distributed setup run ”in-sync” as if they were all running on the same node.
Chapter 5

The Platform Abstraction Layer

We presented in Chapter 3 the Timing Definition Language (TDL) as a portable way of explicitly specifying the timing constraints of real-time software. By using the E-Code to encapsulate the TDL semantics as presented in the Chapter 4, we solve only half of the platform-independence problem. The bindings between the user-code expressed in a general programming language (e.g., C) and the E-Code, along with the run-time environment remain to be solved. Readers familiar with Giotto may argue that its runtime environment and middleware have been ported on a number of platforms; however, they are rather incompatible and are tightly integrated with specific features of each platform.

The addition of the component model with parallel execution of multiple modules and the transparent distribution concept make the porting of the TDL semantics to new platforms a rather difficult task. Each TDL building block, such as port, task, sensor getter, and actuator setter, may have a completely different implementation on each platform. Moreover, the computational unit of a platform and the CPU multiplexing mechanism, that is the scheduling scheme, are different on most platforms.

As a solution to the aforementioned problem, we present in this chapter an abstraction of each TDL building block and the CPU multiplexing mechanism (implemented differently on each platform), and introduce a Platform Abstraction Layer (PAL). PAL allows us to create a portable real-time environment, which customizes the TDL binding rules (i.e., the mappings of each TDL entity to platform specific constructs), the management of the run-time resources for most real-time platforms, and the topology of the system. The portable runtime environment of TDL interacts and controls through an automatically generated middleware the user-defined functionality for tasks, sensors, and actuators; hence, implementing the TDL semantics on the platform. As a result, PAL enables source-code level portability for the TDL runtime environment and middleware on different platforms. The only non-portable elements remain the sensor getters and actuator setters, which deal with specific IO resources part of/connected to the hardware platform.
CHAPTER 5. THE PLATFORM ABSTRACTION LAYER

We begin this chapter with a conceptual overview of the PAL, the interaction of the developer with a simplified TDL tool-chain using PAL, and the purpose of the three major parts of PAL: TDL Mappings (TDLMp), Runtime Resource Management (RTRM), and TDLComm. Section 5.2 details the mappings of the TDL entities into corresponding C constructs. We then describe the RTRM part for the management of time and computational units. Following, we present the TDL Scheduler as a critical component of the TDL runtime environment and introduce a high-performance hybrid version of the EDF scheduling algorithm. In Section 5.5, we detail our solution for transparent distribution, namely the TDLComm part, which handles the value and time deterministic exchange of information between modules placed on remote nodes of a distributed system. We conclude this chapter with the presentation of an experimental Fault-Tolerance layer for TDL (TDL-FT) targeted at distributed systems.

5.1 Conceptual overview

The Platform Abstraction Layer consists of a set of coding rules, a small set of platform independent API calls presented to the upper layers (the TDL runtime environment) and a mechanism for deterministic distribution of TDL modules in multi-node systems. We provide in this section a conceptual overview on the role of PAL for a portable and deterministic TDL runtime environment and middleware.

The PAL abstracts from the target platform comprising the hardware and the real-time operating system. The C language (Kernighan and Ritchie, 1978) is the de facto standard for implementing embedded software; thus, we use it to implement our PAL. Figure 5.1 depicts a simplified TDL tool-chain for compiling TDL modules and the associated functionality code on single-node systems. Nevertheless, through the plug-in architecture of the TDL compiler, we can use a similar approach for distributed systems (as described in Section 5.5) or other combination of implementation language and RTOS.

The role of PAL within the TDL tool-chain. In the lower part of the Figure 5.1 depicts the extension of the TDL compiler with an ANSI-C plugin. The compiler parses each TDL module and checks whether the module contains errors. For an error-free module it generates the corresponding binary E-Code file, and then it invokes the ANSI-C plugin.

The TDL Mappings (TDLMp) part of PAL contains the coding rules for the generation of a middleware which binds the functionality code of a module with its timing specifications. We apply these rules in two phases: first, at compile time, when the ANSI-C plugin generates automatically the middleware (i.e., glue-code), and second, at run-time, when the TDL runtime environment interacts with the middleware and the functionality code of the tasks, sensors, actuators, and guards.

In the first phase, compilation of TDL modules via the abstract syntax tree passed by the TDL compiler, the plugin generates a corresponding module wrapper for each
TDL module through TDLMp. We present the details of the module wrappers and its constituents in the next section, and ignore them at this point. For simplicity, we consider a module wrapper as a corresponding ANSI-C glue-code file containing the mappings of each TDL entity into C language.

The top-right part of the Figure 5.1 shows the compilation process of the portable TDL runtime environment, with the functionality code of the modules composing the TDL application, and with the generated module wrappers (middleware). Other platform specific libraries or drivers may be added at this step. TDLMp abstracts in this case the interaction between these runtime components in the resulting real-time application.

In the second phase, we transfer the executable real-time application to the target platform and execute it. The real-time executable contains the TDL runtime environment, the middleware, the user-defined functionality (for tasks, guards, sensors and actuators), and platform specific functionality (such as drivers or RTOS libraries). In the startup phase of the application depicted in Figure 5.2, the PAL acts as mediator between the Initializer of the TDL runtime environment and the underlying RTOS. Thus, it abstracts the low-level initialization functions of the system (the top grayed box in the figure). Afterward, it performs the loading and initialization of the TDL module-wrappers. It then provides the access of the E-Machine to the E-Code and the drivers provided by the module wrappers. After the initialization phase is completed, the TDL Scheduler takes control of the runtime environment and interacts through PAL with the underlying RTOS.

During the execution of a real-time TDL application, PAL acts as a bridge between the underlying RTOS, the components of the TDL runtime environment, and the logical modules of the application. Each logical module consists of the generated middleware and the user-defined C functionality code. The Runtime Resource
Management (RTRM) part of PAL from the bottom of the Figure 5.3 abstracts the computational units of the platform (such as processes or threads) and the platform scheduling scheme. Hence, it mediates the interaction of the TDL Scheduler with the underlying RTOS.

The third part of PAL, namely the TDLComm layer, is available only in distributed applications. In such cases, it provides the mechanisms for transparent distribution of TDL modules and deterministic exchange of information between the nodes of the distributed system. Through TDLMp, TDLComm synchronizes the states of a service provider module with its stubs (previously presented in Section 3.3.2), and updates the corresponding ports values. Through RTRM, it interacts with the RTOS to communicate the information between the nodes of the distributed system and with the TDL Scheduler to maintain the time-synchronized state of the E-Machines of all nodes.

The design of PAL. To implement a PAL that covers a variety of platforms, we group the platforms into two classes, by exploiting the similarities between their underlying operational concepts. The classes cover all existing real-time platforms:

- **time-triggered** such as TTA [Kopetz and Bauer, 2002] and OSEKtime [OSEK Group, 2001]. The existing time-triggered platforms are based on static schedules for tasks, generated off-line (typically at compile time). Such task schedule is
5.1. CONCEPTUAL OVERVIEW

Figure 5.3: The layered architecture of the TDL runtime environment

generally expressed as a table. For different application states most platforms support additional task scheduling tables (sometimes restricted to have the same size), which can be interchanged at run-time. Therefore, for a TDL module running on a similar platform, we would require a number of scheduling tables equal with the number of modes of the module. At run-time depending on the mode of operation of the module, we would exchange the scheduling tables at the time of a mode-switch. However, for multiple TDL modules running in parallel, the number of scheduling tables grows to the product of the number of phases per module (detailed in Section 5.5.4), because they have to capture all possible states of parallel modules. In most time-triggered platforms this remains the only solution, because of their simple clock-driven dispatcher that is unable to mix task from multiple scheduling tables at the same time. Consequently, in the general case of a TDL application, the static schedules of these platforms limit or inhibit the potential of TDL mode switches (Pletzer, 2005).

- event-triggered, such as OSEK (OSEK Group, 2005), RT-POSIX (POSIX, 2003b). Most of the event-based RTOS use priority-driven scheduling. On OSEK compatible platforms, the priorities are statically defined at compile time and cannot be changed during run-time. Other platforms such as RT-POSIX allow the developer to set both at compile time and at run-time arbitrary priorities to threads and processes. In general, these platforms rely on a low-overhead on-line scheduler, which dispatches the tasks according to their priorities.

In our experience, an event-based RTOS offers the appropriate flexibility for implementing a runtime system for TDL, because we can map the LET concept over
CHAPTER 5. THE PLATFORM ABSTRACTION LAYER

standard RTOS priorities. Therefore, we further analyze the differences between the
event-based platforms for the time management functionality and the computational
units available:

• native tasks, for example on OSEK and micro-kernels;

• processes, for example on RT-POSIX and InTime (Tenasys 2005);

• threads, for example on RT-POSIX and InTime.

The computational unit available on a platform greatly influences the RTRM. For
example, on systems supporting both processes and threads, the inter-process commu-
nication mechanisms have a different influence on the system performance than the
intra-process communication mechanisms available when using threads. We detail in
the Section 5.4 the effect of this analysis on the scheduling mechanisms for TDL tasks.

Aside from the scheduling scheme available on the target platform, the synchro-
nization primitives (e.g., semaphores, locks, and mutexes) used to exchange state infor-
mation between different components of the real-time application greatly influence the
determinism and performance of the application. These platform specifics also influence
TDLMp, as for example the notion of TDL task may map to a thread or process.

In the case of distributed systems, from the physical communication point of view, we
have two possible options for exchanging information on the communication bus: CPU
assisted or via a dedicated network processor. In the case when the main CPU performs
the actual communication by serializing the bytes of information, the functional tasks
may have less CPU time at their disposal to compute. The possible jitter and non-
determinism at task level introduced by this approach is typically unacceptable in the
case of hard real-time systems. Therefore, in our case we rely on a dedicated network
processor, such as a CAN controller, which interacts with the TDLCComm layer for
exchanging the data between the nodes.

In the following sections, we detail each part of PAL and present their interactions
with the TDL runtime environment, the TDL modules, and the underlying RTOS.

5.2 TDL Mappings (TDLMp)

As previously mentioned, on single node systems, the C-based PAL is composed of two
parts. In this section we describe the TDL Mappings part, whereas the next section cov-
ers the Runtime Resource Management (RTRM). The third part, the TDLCComm layer
that completes the PAL in the case of distributed systems is presented in Section 5.5.

The TDLMp part of PAL represents a set of macros, naming and coding conventions,
along with automatic generated code, which enable a translation of a set of TDL entities
to their correspondent in C language. The resulted C code is often called middleware or glue-code.

Through the set of TDL Mappings, the ANSI-C plugin generates for each TDL module a C module-wrapper, which defines the equivalent of TDL ports, the implicit drivers, the computing entities corresponding to TDL tasks, the guards, and the E-Code. In addition, the TDLMp abstracts not only the specific C constructs of each TDL entity, but also the binding of the TDL entities with their corresponding functionality code (also implemented in C).

To clarify the role and functionality of PAL, we use mostly in this chapter the following application consisting of two modules M1 and M2. The first module contains two tasks with the same period of 10ms: inc, which increments its output values with 10 in the range of 50-200, and dec, which decrements its output values in the same range. When reaching their extreme values the tasks reset their output ports to their base values. The tasks provide the input values to two actuators a1 and a2.

The module has two operational modes with a period of 150ms, f11 and f12. In the first mode f11, the two tasks are invoked 15 times per period, resulting a LET of 10ms per task. The actuators are updated at the same rate as the tasks. The first mode defines a mode switch condition to the second mode, which is evaluated once per mode period (i.e., every 150ms). In the mode f12 the dec task has twice its frequency from the mode f11 (i.e., a period of 5ms). The sensor s provides the input for the guard that supervises the mode switches between the two modes. As external hardware, a simple push-button may be connected to the provide this functionality. The Figure 5.4 depicts a visual representation of the two modules.

The module M2 imports the module M1 and has only one mode that contains the task sum. The task has as inputs the output ports of the two tasks from the module M1. Its implementation sums the output values of the inc and dec tasks and produces the input value for the actuator a.

Figure 5.4: Visual representation of the modules M1 and M2
Listing 5.1: Module M1

```c
module M1 {
  public const c1 = 50; c2 = 200;
  refPeriod = 150ms;

  sensor
  int s uses getS;

  actuator
  int a1 := c1 uses setA1;
  int a2 := c2 uses setA2;

  public task inc [wcet=1ms] {
    output int o := c1;
    uses incImpl(o);
  }

  public task dec [1ms] {
    output int o := c2;
    uses declmpl(o);
  }

  start mode f11 [period=refPeriod] {
    task
      [15] inc(); // 10ms period
      [15] dec(); // 10ms period
    actuator
      [15] a1 := inc.o;
      [15] a2 := dec.o;
    mode
      [15] if switch2f12(s, inc.o) then m2;
  }

  mode f12 [period=refPeriod] {
    task
      [15] inc(); // 10ms period
      [30] dec(); // 5ms period
    actuator
      [15] a1 := inc.o; // updated every 10ms
      [30] a2 := dec.o; // twice as fast as a1
    mode
      [1] if switch2f11(s, inc.o) then m1;
  }
}
```

Listing 5.2: Module M2

```c
module M2 {
  import M1;
  // get access to M1 and
  // all its public entities

  actuator
  int a := M1.c2 uses setA;

  public task sum [wcet=1ms] {
    input
      int i1;
      int i2;
    output
      int o := M1.c2;
    uses sum Impl(i1, i2, o);
  }

  // use public constant from M1
  // as reference period
  start mode main [period=M1.refPeriod] {
    task
      [15] sum(M1.inc.o, M1.dec.o); // 10ms
    actuator
      [15] a := sum.o; // update every 10ms
  }
}
```
5.2.1 Module wrappers

The fundamental runtime component of a TDL application is the module-wrapper. Each TDL module has one corresponding ANSI-C module wrapper. It acts as a logical container of a group of related tasks, modes, guards, sensors and actuators. The module-wrapper file contains the glue-code, which binds the TDL specifications of a module with its C functionality.

The name space defined by a group of files compiled together by a standard C compiler is global; therefore, naming conventions apply to all elements of generated C code that make a module-wrapper (such as ports, tasks definitions, etc). In the Figure 5.3, we see the top-level logical modules containing the module wrapper (C glue-code) and the accompanying C functionality. In addition, for distributed systems there can be an additional number of stub module wrappers (presented later in this section).

For the sample module M1 from Listing 5.1, the corresponding module-wrapper file contains a number of sections automatically generated by the ANSI-C plugin (see Listing 5.3). Each section may contain other wrappers for the TDL entities of the module. The algorithm for the generation of the module wrapper is presented later in this section.

The first section of the wrapper module is the includes section, which refers to the platform-independent core of the TDL runtime environment, the PAL definitions, a dependency solver file, and the user defined functionality for tasks, sensor, actuators, and guard functions. Following is a section of port definitions, and the tasks section containing task wrappers. The modes table comes next, along with the C-encoded block of E-Code. The driver wrappers and guard wrappers, along with the module descriptor finalize the module wrapper.

Listing 5.3: Overview of module wrapper M1

```c
#include "emachine.h" // TDL Runtime core
#include "system_dependent.h" // PAL implementation
#include "modules.h" // Dependency solver file
#include "M1_TDL.h" // Definitions for the module wrapper
#include "M1.h" // User functionality

/** Ports definitions *********************************************************
 static TDL_Byte _TDL_M1_Port0 = 50; // M1.a1
 [...] TDL_Byte _TDL_M1_Port4_VAL = 50; // actual value of output o1
```
CHAPTER 5. THE PLATFORM ABSTRACTION LAYER

/** Tasks listing ******************************************/  
// table of platform specific constructs for task wrappers (e.g. thread ids)  
_TDL_TASK_WRAPPER_DATA _TDL_TASK_WRAPPERS_DATA_M1[_TDL_TASKS_COUNT_M1];

TDL_Byte _TDL_ACTIVE_TASKS_M1[1]; // bit vector for task status

_TDL_TASK_WRAPPER(M1, dec) // Wrapper for the actual M1_dec functionality
{  
  _TDL_TASK_WRAPPER_HEADER(M1, 0);  
  M1_decImpl(&_TDL_M1_Port3_VAL);  
  _TDL_TASK_WRAPPER FOOTER(M1, 0); // Signal completion
}

_TDL_TASK_WRAPPER(M1, inc) // Wrapper for the actual M1_inc functionality
{ [...]

// table of tasks properties
const _TDL_Task _TDL_TASKS_M1[_TDL_TASKS_COUNT_M1] = {
  TDL_TASK_PROPERTY(M1, dec, 1000),
  TDL_TASK_PROPERTY(M1, inc, 1000)
};

/** Modes ******************************************/  
const _TDL_Mode _TDL_MODES_M1[_TDL_MODES_COUNT_M1] = {
  TDL_MODE_PROPERTY(M1, m2, 21, 10000, 3),
  TDL_MODE_PROPERTY(M1, m1, 4, 10000, 2)
};

/** Ecode block ******************************************/  
const _TDL_ECode _TDL_ECODES_M1[_TDL_ECODES_COUNT_M1] = {
  // initialization
  CALL(6), // #0 actuator init: setA1(a1)
  CALL(7), // #1 actuator init: setA2(a2)
  CALL(8), // #2 actuator init: getS(s)
  RETURN(), // #3 return

  // Start Mode: m1[0] starting at: #4 period: 10000
  CALL(9), // #4 call 9 task input update: inc
  [...]

  // Mode: m2[1] starting at: #21 period: 10000
  CALL(14), // #21 call 14 task input update: inc
  RELEASE(1,3,10000), // #22 release 1 : incImpl [1] until 10000 wcet = 1000
  [...]

5.2. TDL MAPPINGS (TDLMP)

```c
SWITCH(1), // #44 jump 21 next cycle: m2
```

```c
/* Drivers ***************************************************************************/
_TDL_DRIVERS_PROLOG(M1)

_TDL_DRIVER_BEGIN(M1, 0) //start task dec
    TDL_RELEASE_TASK(M1, dec, 0); // special macro for OSEK or UNIX/POSIX
_TDL_DRIVER_END

[…] TDL_DRIVERS_EPILOG

/* Guards ***************************************************************************/
_TDL_GUARDS_PROLOG(M1)

_TDL_GUARD_BEGIN(M1, 0)
    return M1_switch2m2(_TDL_M1_Port2, _TDL_M1_Port4);
_TDL_GUARD_END

[…] TDL_GUARDS_EPILOG

/* Module descriptor ***************************************************************************/
TDL_MODULE(M1);
// end module M1
```

Listing 5.4: Macro expansion of module descriptor

```c
#define TDL_MODULE(m) const _TDL_Module _TDL_MODULE_##m = {
    (_TDL_DRIVERS_LOOKUP *) _TDL_DRIVERS_##m, _TDL_DRIVERS_COUNT_##m,
    (_TDL_GUARDS_LOOKUP *) _TDL_GUARDS_##m, _TDL_GUARDS_COUNT_##m,
```

Most automatically generated C constructs are encoded through macros, which can be easier adapted to platform specifics. For example, on platforms with sufficient memory resources the developer may include debugging/profiling information in the form of explicit names for TDL tasks and modes. The macro of the module descriptor from Listing 5.4 provides automatically named references to all translated TDL entities such as tasks, drivers, guards, modes, and E-Code. This method of using macros relies on the C preprocessor to expand the macros at compile-time. It also enables changes into the core of the TDL runtime system without requiring recompilation of the TDL sources and generation of new module wrappers.
CHAPTER 5. THE PLATFORM ABSTRACTION LAYER

\begin{verbatim}
(.TDL_ECode *) _TDL_ECODES_##m, \
_TDL_ECODES_COUNT_##m, \
(.TDL_Task *) _TDL_TASKS_##m, \
_TDL_TASKS_COUNT_##m, \
(TDL_Byte *) _TDL_ACTIVE_TASKS_##m, \
(.TDL_TASK_WRAPPER_DATA *) _TDL_TASK_WRAPPERS_DATA_##m, \
(.TDL_Mode *) _TDL_MODES_##m, _TDL_MODES_COUNT_##m, \
m##.INIT, #m);
\end{verbatim}

Formally, we define a module wrapper as the tuple \((M, \hat{\text{Ports}}[M], \hat{\text{Types}}[M], \hat{\text{Tasks}}[M], \hat{\text{Drivers}}[M], \hat{\text{Guards}}[M])\). The set \(\hat{\text{Ports}}[M]\) represents the set of mapped ports of the entities from module \(M\), forming a concrete expression of \(P[M]\). The set of opaque types \(\hat{\text{Types}}[M]\) has a direct correspondent in C structures. The set \(\hat{\text{Tasks}}[M]\) represents the set of task wrappers we define for the set of task \(\text{Tasks}[M]\) of the module \(M\). For the implicit drivers of the module, we define the set \(\hat{\text{Drivers}}[M]\). The TDL guards have their correspondent in the set \(\hat{\text{Guards}}[M]\).

5.2.1.1 Types

For the TDL entities of a module, we define a set of binding rules for the basic types of TDL to ANSI-C types, such as \textit{int}, \textit{float}; for the opaque types, we expect a user-implemented structure with the same name as the corresponding opaque type from the TDL source code.

PAL isolates the opaque types in the global C-language namespace via naming conventions, which limit the scope of each type to the module that defined it. Thus, the developer has to provide in the header file of the functionality code, the definitions of the opaque types referred in the corresponding TDL module, prefixed with the name of the module (see the following table).

<table>
<thead>
<tr>
<th>TDL type</th>
<th>C name</th>
<th>Default ANSI-C type</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>TDL_Byte</td>
<td>unsigned char</td>
</tr>
<tr>
<td>boolean</td>
<td>TDL_Boolean</td>
<td>unsigned char</td>
</tr>
<tr>
<td>char</td>
<td>TDL_Char</td>
<td>unsigned char</td>
</tr>
<tr>
<td>short</td>
<td>TDL_Short</td>
<td>short int</td>
</tr>
<tr>
<td>int</td>
<td>TDL_Int</td>
<td>int</td>
</tr>
<tr>
<td>long</td>
<td>TDL_Long</td>
<td>long long</td>
</tr>
<tr>
<td>float</td>
<td>TDL_Float</td>
<td>float</td>
</tr>
<tr>
<td>double</td>
<td>TDL_Double</td>
<td>double</td>
</tr>
<tr>
<td>string</td>
<td>TDL_String</td>
<td>unsigned char[24]</td>
</tr>
<tr>
<td>opaque</td>
<td>M_opaque</td>
<td>struct ... M_opaque</td>
</tr>
</tbody>
</table>

Table 5.1: Summary of TDL types mapping to C language
The drivers perform the value-copy operations only between ports of compatible types. For the execution of task implementations, we use \textit{pass-by-value} semantic for input ports and \textit{pass-by-reference} semantic for output ports. As in the C-language the arrays are always passed by reference, we have to enclose them in structures. In this way, we have compatible value-copy operations and correct pass-by-value semantics.

The table 5.1 summarizes the TDL type and their correspondent in C. The default binding rules for the C correspondents are fully customizable to accommodate various machine-word lengths, ranging from 8-bit micro-controllers to 64-bit processors.

Formally, \(\forall \theta \in \text{Types}[M]_O, \exists \tilde{\theta} \in \text{Types}[M]_O, \theta \leftrightarrow \tilde{\theta}, \) and \(\forall N \in \text{TDL}, \not\exists \theta' \in \text{Types}[N]_O, \theta = \theta', \tilde{\theta} = \tilde{\theta}', \) where \(M\) and \(N\) are TDL modules.

### 5.2.1.2 Ports

Dynamic memory allocation at run-time has a negative effect on the performance and determinism of real-time systems; hence, most real-time systems (e.g. the Ravenscar profile \cite{Burns_etal_2003}) prohibit dynamic task attributes and dynamic memory allocation.

Following a similar approach, we statically define and allocate the set of ports of a module, using the aforementioned binding rules for their types. In addition to task ports, the sensors and actuators introduce implicitly output, respectively input ports. The task output ports are duplicated: one set represents the \textit{internal} values provided as an immediate result of the task execution, whereas the other set represents the values \textit{visible} by all other entities according to the LET semantics. Only at the end of the LET of each task, we update the corresponding visible port values from the internal values, through the task termination drivers. For the data flow between ports and TDL entities see Figure 5.5.

To prevent the naming collisions in the C namespace, we may use a convention such as the fully qualified name of the port, e.g. \(M.A.o\) for the output port \(o\) of the task \(A\) from module \(M\), or better an encoded name, e.g. \(\text{Port}_M.2\). The only restriction we have is that the port names from all modules must be unique in the system, because only at the integration phase we know which modules run on the same node and their interactions. For example, in the Listing 5.5, we present the ports of the module \(M1\) from Listing 5.1.

```
// Ports definitions
static TDL_B_ _TDL_M1_Port0 = 50; // M1_a1
static TDL_B_ _TDL_M1_Port1 = 204; // M1_a2
static TDL_B_ _TDL_M1_Port2; // M1_s
TDL_B_ _TDL_M1_Port3 = 204; // M1_dec_o public
TDL_B_ _TDL_M1_Port3_VAL = 204; // internal value of output M1_dec_o
TDL_B_ _TDL_M1_Port4 = 50; // M1_inc_o public
TDL_B_ _TDL_M1_Port4_VAL = 50; // internal value of output M1_inc_o
```

Listing 5.5: Ports of module M1
We enforce a local scope for the private ports of the module M1, such as the sensor and the two actuators, through the use of the static keyword. On the other hand, the public ports are declared as external in the header file of the module wrapper. Thus, we can access them directly from any other module compiled along with the module M1. This approach of declaring the ports in the body of the module wrapper and exporting them from its associated header solves the dependency problem between modules and allows for cyclical dependencies in their import relationships.

The runtime environment controls the access to the values of the ports declared in a module wrapper through the generated drivers. As the ports from our example have constant initializers, we initialize them statically with the corresponding values. For ports of opaque types, the types are already defined in the header file of the user functionality code; thus, we can directly prefix the ports with the corresponding type. Their initialization in this case may be subject to an external initializer function.

Formally, we use the port mapping function $\hat{\text{Ports}}$ from Equation (5.1), which defines the mapping of a port from a TDL module to its correspondent port of the module wrapper. There is one exception; an output port has two correspondent ports for its internal and visible values in the wrapper module.

$$
\hat{\text{Ports}}: P[M] \to \hat{\text{Ports}}[M], \quad \hat{\text{Ports}}(p) = \{ p' | p \in P[M] \setminus P_{\tau_o}[M], \ \theta(p) = \theta(p') \} \cup \\
\{ (p'_i, p'_v) | p \in P_{\tau_o}[M], \ \theta(p) = \theta(p') \} \tag{5.1}
$$

### 5.2.1.3 Drivers

The drivers represent the only way of manipulating the port values available to the entities of a module. There are two exceptions: task status ports and internal output ports are private to a task and can be directly updated by the owning task (see Figure 5.5). In addition, only the E-Machine can execute the drivers, ensuring the correct behavior under the LET semantics expressed in E-Code.

The C code generated by the ANSI-C plugin uses macros to express two possible implementations: through a switch mechanism or using a table lookup mechanism. The set of drivers is encompassed between a prolog and an epilog sections. The prolog denotes the actual C switch statement or the beginning of the set of driver functions, whereas the epilog finalizes the corresponding C construct used in the prolog. As the set of driver is finite, we relate in E-Code to each driver through its number. In the Listing 5.6 we present the general structure of a driver wrapper generated automatically by the ANSI-C plug-in.

Listing 5.6: PAL - Driver wrapper

```c
// beginning of C wrapper
.TDL_DRIVER_BEGIN(M,i) // module M and an index i of the driver in the set of drivers
d // the actual driver (port value assignments)
.TDL_DRIVER_END
```

118
5.2. TDL MAPPINGS (TDLMP)

Between the prolog and epilog of the driver set of a module, there can be a finite number of drivers. The driver itself may consist of port-value assignment constructs, sensor getter or actuator setter calls, and release-task constructs. In the Listing 5.7, we present as example a subset of the drivers of the module M2.

Listing 5.7: Drivers of module M2 from Listing 5.2

```
//Drivers
.TDL_DRIVERS_PROLOG(M2)

.TDL_DRIVER_BEGIN(M2, 0)  // driver to release the sum task
    TDL_RELEASE_TASK(M2, sum, 0);  // special macro to identify task
.TDL_DRIVER_END

[..]

.TDL_DRIVER_BEGIN(M2, 3)  // driver to set the actuator a of M2
    M2_setA(_TDL_M2_Port0);  // call the setter function directly
.TDL_DRIVER_END

.TDL_DRIVER_BEGIN(M2, 4)  // prepare release of sum task
    _TDL_M2_Port1 = _TDL_M1_Port4;  // copy output ports values
    _TDL_M2_Port2 = _TDL_M1_Port3;  // of tasks inc and dec from M1
.TDL_DRIVER_END

.TDL_DRIVER_BEGIN(M2, 5)  // driver to update the actuator a
    _TDL_M2_Port0 = _TDL_M2_Port3;  // copy output value of sum task
.TDL_DRIVER_END

.TDL_DRIVERS_EPILOG
```

The Table 5.2 summarizes the functionality of TDL drivers. The actual representation of the drivers in the wrapper module is described in the algorithm from Listing 5.17.

We define statically the set of drivers, which perform the value copy operations between ports, according to the defined interactions such as sensor to task, task to task, and task to actuator. For each interaction, we have a separate driver. A sensor-getter driver $d[\sigma]$ updates the implicit output port $p_\sigma$ of the sensor $\sigma$ with the result of the sensor-getter function $u[\sigma]$. Similarly, an actuator-setter driver $d[\alpha]$ executes the implementation of the actuator-setter function $u[\alpha]$ with the current value of the implicit input port of the actuator $p_\alpha$. The actuator-update driver $d[\alpha^u]$, copies the value of a task output port $p_{\tau,\alpha}$ or sensor output port $p_\sigma$ into the implicit input port $p_\alpha$ of the corresponding actuator $\alpha$.

For tasks we define the drivers that maintain their LET (i.e., release and terminate drivers). The task release driver $d[\tau^r]$ performs the update of the task input ports $P_i[\tau]$
CHAPTER 5. THE PLATFORM ABSTRACTION LAYER

<table>
<thead>
<tr>
<th>Driver target</th>
<th>Type</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>sensor</td>
<td>sensor getter</td>
<td>$p_\sigma \leftarrow u[\sigma]$</td>
</tr>
<tr>
<td>actuator</td>
<td>actuator update</td>
<td>$p_\alpha \leftarrow (p_\tau \land p_\sigma)$</td>
</tr>
<tr>
<td></td>
<td>actuator setter</td>
<td>$u<a href="p_%5Calpha">\alpha</a>$</td>
</tr>
<tr>
<td>task</td>
<td>release</td>
<td>$P_i[\tau] \leftarrow P'; P' \in P$</td>
</tr>
<tr>
<td></td>
<td>terminate</td>
<td>$P_o[\tau] \leftarrow P_o[\tau]$</td>
</tr>
<tr>
<td></td>
<td>start</td>
<td>$u<a href="P%5B%5Ctau%5D">\tau</a>$</td>
</tr>
<tr>
<td></td>
<td>stop</td>
<td>$TDLComm &amp; TDL runtime$</td>
</tr>
</tbody>
</table>

Table 5.2: Summary of TDL drivers functionality mapping to C language

with the values of the output ports $P'$ passed as parameters to the corresponding task invocation. The terminate driver $d[\tau_t]$ updates the visible task output ports $P_o[\tau]$ with the result of the task computation stored in the internal copy of task output ports $P_o[\tau]$.

In addition, we define helper drivers for the physical events of task start, and stop. The task-start driver executes the functionality of the task $u[\tau]$ with parameters the set of task ports $P[\tau]$, with pass-by-value for input ports $P_i[\tau]$, respectively pass-by-reference for output and status ports $P_o[\tau]$, $P_s[\tau]$. We use the task-stop drivers for interacting with the $TDLComm$ layer for distribution and E-Machine or $TDL Scheduler$ for resource management.

Figure 5.5: Dataflow between TDL entities and ports

**Algorithm.** In the Listing 5.8 we present an algorithm for generating the implicit TDL drivers. We start by creating a driver for each sensor-getter $u[\sigma]$ and actuator-setter $u[\alpha]$ function. These drivers are used for interacting with the environment, regardless of the operating mode of the containing module. We continue with the
drivers for the actuator updates. Actuators may be updated with different output ports from tasks or sensors, depending on the mode of operation of the module and the user-defined guard conditions. Therefore, we define multiple actuator updates drivers for each actuator. The actuator update action is only valid when the output port of the entity providing the updated values to the actuator has the same type as the actuator and is reachable from the namespace of the actuator.

Listing 5.8: Algorithm for generating the implicit TDL drivers

Drivers[M] ← ∅

// sensor getters
∀σ ∈ Sensors[M]
    d[σ] ← {pσ ← u[σ]}
    Drivers[M] ← Drivers[M] ∪ d[σ]

// actuator setters
∀α ∈ Actuators[M]
    d[α] ← {u[α] ← po[α]}
    Drivers[M] ← Drivers[M] ∪ d[α]

// actuator updates
k ← 0 // there are more updates from different modes or guard combinations
∀σ ∈ Sensors[M]
    ∀α ∈ Actuators[M]
        if (∃γ(σ, α)) then // we have an actuator updated with sensor values
            d[α] ← {pα ← pσ}
            Drivers[M] ← Drivers[M] ∪ d[α] // copy values between sensor and actuator ports
            k ← k + 1
        end if
    ∀τ ∈ Tasks[M]
        if (∃γ(τ, α) ∧ ∃Δ(α, τ)) then // we have an actuator updated with task output
            d[α] ← {pα ← po[τ]}
            Drivers[M] ← Drivers[M] ∪ d[α] // copy values between task output and actuator ports
            k ← k + 1
        end if

// task drivers
∀τ ∈ Tasks[M]
    // termination driver
d[τ] ← ∅
    ∀p ∈ po[τ] // update task output ports (visible from internal)
        d[τ] ← d[τ] ∪ {Ports(p)_o ← Ports(p)_i}
    Drivers[M] ← Drivers[M] ∪ d[τ]

// start driver – release functionality through PAL
Drivers[M] ← Drivers[M] ∪ {RELEASE(τ)} // start driver
// stop driver — TDLComm and E-Machine interaction
\[ d_{TDLComm}^\tau \leftarrow \emptyset \]
\[ \forall p \in p_o[\tau] \]
\[ d_{TDLComm}^\tau \leftarrow d_{TDLComm}^\tau \cup \{TDLComm(\text{Ports}(p_i))\} \] // access internal output ports
\[ \text{Drivers}[M] \leftarrow \text{Drivers}[M] \cup d_{TDLComm}^\tau \]

// release drivers depending on the input/output assignments from each mode
\[ \forall m \in \text{Modes}[M] \]
\[ d[\tau_m^\tau] \leftarrow \emptyset \] // release driver from mode \( m \)
\[ \forall \sigma \in \text{Sensors}[M] \] // check all sensor to task relations
\[ \text{if} \left( \exists p_i[\tau] \land \exists \gamma(\sigma, p_i[\tau]) \right) \] // task input port updated with sensor values
\[ d[\tau_m^\tau] \leftarrow d[\tau_m^\tau] \cup \{p_i[\tau] \leftarrow p_o[\sigma]\} \] // copy values between sensor and actuator ports
\[ \text{end if} \]
\[ \forall \tau' \in \text{Tasks}[M] \] // check all task to task relations
\[ \text{if} \left( \exists \gamma(\tau', \tau_m) \land \exists \Delta(\tau, \tau_m') \right) \] // we have the task updated with another task’s output
\[ d[\tau_m^\tau] \leftarrow d[\tau_m^\tau] \cup \{p_o[\tau] \leftarrow p_o[\tau']\} \]
\[ \text{end if} \]
\[ \text{Drivers}[M] \leftarrow \text{Drivers}[M] \cup d[\tau_m^\tau] \]
// end algorithm

We then create the task drivers, starting with the termination drivers, which update
the values of the visible output ports from the values of the interval output ports. A

task has read-only access to its input ports during its LET (pass by value mechanism).
It has full access over its status and internal copy of the output ports (pass by reference
mechanism).

For the release mechanism of PAL (see Section 5.3) we use the start driver of a

task. It performs the logical task-release functionality regardless of the physical task
model of the platform.

The stop driver of a task executes at the completion time of the task. Its purpose
is to communicate to the TDLComm layer and E-Machine the results of the task
computation from the internal output ports. Note that the values from the internal
output ports are not available to other TDL entities until the end of the LET, when
the termination driver updates the visible output ports (see Section 5.5).

Each task has a release driver for each mode in which it is invoked. In each mode, the
release driver of a task instance updates the values of the input ports of the instance
with the corresponding values of the output ports of the entities that are passed as
parameters. When the input parameters of a task instance refer to output ports from
other tasks, the latest visible output ports values are used for initializing the input
ports.

5.2.1.4 Guards

The ANSI-C plugin generates for each TDL guard a corresponding guard wrapper using
the same macro mechanisms previously described for drivers. Thus, the developer can
enable one of the two possible implementations for the invocation of the user-defined
5.2. TDL MAPPINGS (TDLMP)

implementations of the TDL guards. One implementation has a wrapper-function for each user guard function and a table that maps the guard numbers to their wrapper functions. The runtime system performs a lookup in this table of function pointers and then calls the respective function. For a small number of guards, an alternative implementation uses a switch for the user guard functions, resulting a shorter and faster executable code.

Listing 5.9: TDL Guard

```plaintext
module TurboPump {
    sensor byte pressure uses getPressure;
    [...]
    start mode normal [period = 10ms] {
        [...]
        mode [1] if High(pressure) then degraded;
    }

    mode degraded [period = 50ms] {
        [...]
    }
}
```

Listing 5.10: C Implementation

```c
#include "TurboPump.h"
#define MAX_PRESSURE 200

// [...] sensor

TDL_Boolean TurboPump_High(TDL_Byte s) {
    if (s >= MAX_PRESSURE)
        return(TRUE);
    else
        return(FALSE);
}
```

The guard used for the mode switch in the Listing 5.9 has a corresponding user-defined C implementation with the same number, type, and order of arguments as exemplified in the Listing 5.10. For an actual implementation of a guard wrapper we refer to the Listing 5.11.

Listing 5.11: Guard wrappers of module M1 from Listing 5.1

```c
// Guards
.TDL_GUARDS_PROLOG(M1)

    .TDL_GUARD_BEGIN(M1, 0)
    // Call to actual implementation with visible port parameters
    return M1_switch2f12(.TDL_M1_Port2, .TDL_M1_Port4);
    .TDL_GUARD_END

    .TDL_GUARD_BEGIN(M1, 1)
    // Call to actual implementation with visible port parameters
    return M1_switch2f11(.TDL_M1_Port2, .TDL_M1_Port4);
    .TDL_GUARD_END

.TDL_GUARDS_EPILOG
```

123
For the Boolean return values of TRUE and FALSE, we use the same type mapping as for Boolean ports of other TDL entities. As the TDL specifications do not provide any mean for type checking of the port types and the actual user provided functions, we rely on late binding via the ANSI-C compiler.

### 5.2.1.5 Sensors and actuators

The TDL sensors and actuators cannot have a generic implementation on any platform because of the different hardware they must support. On the other hand, we can generalize their API. Therefore, we abstract the sensors as simple C functions without parameters that return values of a specific TDL type (see Listing 5.13). The type mapping of a TDL sensor port must match with the return type of the corresponding C function (see Listing 5.12). The actuators are simple void C functions with one parameter only. The parameter has the same type as the type of the actuator port specified in the TDL module.

#### Listing 5.12: TDL Sensor/Actuator
```
module TurboPump {

sensor
int s1pressure uses getS1Pressure;

// module name prefixes impl. name

actuator
byte a := 0 uses setAct1;

[...] // tasks, modes ...
}
```

#### Listing 5.13: C Implementation
```
#include "TurboPump.h"

TDL_Int TurboPump_getS1Pressure()
{
    return inport(IO_Port); // read some I/O
}

void TurboPump_setAct1(TDL_BByte o)
{
    outport(IO_Port, o); // write to some I/O
}
```

We abstract the interaction between the TDL sensors/actuators and the TDL tasks via the TDL drivers mechanism and the generated E-Code. To avoid naming conflicts with other sensors or actuators from other modules possibly located on the same node, we prefix the names of the implementations of each sensor and actuator with the name of the module containing them (see Listing 5.13).

### 5.2.1.6 Tasks

The TDL task mapping presents a problem, as the nature of a task on an arbitrary platform is unclear. Although the semantics of TDL do not specifically require a preemptive runtime system, running the functionality of TDL tasks in a sequence by using a non-preemptive scheduler may not be possible for most applications. Thus, we assume that the target platform supports the preemption of computational jobs. We classify the event-based platforms into three categories depending on the underlying RTOS support for tasks: with native tasks, based on threads, and based on processes.
5.2. TDL MAPPINGS (TDLMP)

Native platform tasks. The first category covers most closed embedded systems, where the small memory size, low CPU frequency, and limited power consumption requirements impose great limitations on the type and amount of functionality provided. In such systems, a minimal RTOS with the user-applications are encapsulated into an executable (firmware) stored into a flash memory or ROM and never changed during run-time. The developer can update the system functionality only with specialized hardware or software tools.

For example, the OSEK standard \cite{OSEK Group 2005} can be used to develop such systems. An OSEK compliant RTOS has a native notion of tasks, which comes close to the TDL notion of task except that the OSEK tasks have no parameters. In addition, the developer has to specify into a configuration file the runtime properties of OSEK tasks such as stack size, priority, and scheduling options.

The second and third categories cover embedded systems with sufficient storage and processing power for a full-featured RTOS. Platforms conforming to the RT-POSIX standard \cite{POSIX 2003b} support processes and threads, some with hardware memory protection between processes. Other platforms, such as InTime \cite{Tenasys 2005} (successor of iRMX), which adds real-time support to standard Windows platforms, also support processes and threads but with a completely different run-time semantics.

Processes for tasks. Using processes for separating tasks of an application is widely used in practice for client-server applications, networking and disk IO. Typical implementations offer virtual memory support, protected memory space for processes, dynamic inter-process communication through messages, pipes, and signals. However, this technology is hardly applicable in the context of embedded systems because of the poor real-time performance of inter-process communication APIs available, the RTOS scheduling restrictions for processes, the lack of control over other processes, etc. Most real-time systems implement threads as an alternate mechanism for lightweight processes.

Threads for tasks. Using threads represents a promising approach for implementing TDL tasks on such systems, as it allows us to bind a user level function to a dedicated execution thread that has direct access to the process memory resources, and can easily manipulate it. This is probably the largest category as it can cover all modern RTOS such as RT-Linux, QNX, InTime, etc.

On the platforms with real-time threading support, there are several options available in running concurrently a set of TDL tasks. A first option is to create a thread on each task invocation. However, in this case the RTOS wastes many CPU cycles for creating and deleting the threads; therefore increasing the overall system load.

A better option is to create one thread for each task during the initialization phase of the runtime environment. To maintain the TDL semantics, we keep the threads corresponding to inactive TDL tasks in a suspended or blocked state, which means
that they wait for the release event from the E-Machine. With this solution, we have
the advantage of knowing exactly the thread that implements a TDL task, which is
useful for debugging or run-time analysis. The minor drawback is that we potentially
waste memory by having an unused set of threads for some infrequent tasks.

For systems with a high number of tasks or limited memory space, another option is
to create a pool of threads and then at run-time select one free thread and execute one
task in its environment. We allocate initially a small number of threads (e.g. 10-20)
and in the case when the application requires more threads we allocate dynamically
additional threads during the idle time, avoiding the typical performance penalty. An
alternative is to preallocate a number of threads corresponding to the maximum number
of parallel tasks. In both cases, we trade memory space for CPU cycles when selecting
a physical thread for a TDL task.

For performance reasons, our PAL implementation focuses on platforms with native
support for TDL tasks and platforms with threads using either one-to-one mapping of
TDL tasks to threads or thread multiplexing from a thread pool.

**Task wrapper.** To abstract from the different physical task model implemented by
an arbitrary platform, we define at least one task wrapper for each user-defined function
that implements the functionality of a TDL task. We require more task wrappers for a
task on systems where the priorities of the task wrappers are assigned statically at com-
pile time and there are no mechanisms for implementing dynamic-priority scheduling
at run-time. In such cases, as we have to use static-priority scheduling and the period
of a TDL task may be different among modes, we implement dynamic mode-switches
via multiple task wrappers. A detailed description of this mechanism is presented in
the next section.

A task wrapper has a declaration and a body. The body of the task wrapper contains
a header and a footer, which encompass a call to the user-defined functionality \( u[\tau] \) of
the TDL task. The PAL task-declarator, the header, and footer are platform dependent,
but the same for all tasks running on that platform.

From the implementation point of view, we describe for each supported platform
category, a set of standard C macro definitions for the task-declarator, its header and
footer. At compile time, the C preprocessor expands the macros into corresponding C
instructions available on that platform.

**Listing 5.14: PAL - Task Wrapper**

```c
TDL_TASK_WRAPPER(M, \tau)  // beginning of a task wrapper \( \hat{\tau} \) for task \( \tau \)
{
    // Header \( \hat{h}_\tau \) implementing a waiting loop or signaling mechanisms
    _TDL_TASK_WRAPPER_HEADER(M, \tau)
    Invoke(u[\tau])  // execute user functionality code for task \( \tau \)
    // Footer \( \hat{f}_\tau \) completing the loop and/or signaling
    _TDL_TASK_WRAPPER FOOTER(M, \tau)
}
```

126
For example, in the case of OSEK compatible platforms, the task wrapper declarator matches exactly the OSEK task declaration: \texttt{TASK(Task wrapper name)}. RT-POSIX has a completely different expansion of the same constructs as depicted in Listing \ref{lst:taskwrapper5.15}. The header implements a waiting loop, whereas the footer marks the task completion and completes the loop.

\begin{quote}
\textbf{Listing 5.15: Macro expansion of task wrapper for RT-POSIX}
\end{quote}
\begin{verbatim}
// RT−POSIX compliant task wrapper constructs
#define TDL_TASK_WRAPPER(m,task) 
   void TDL_Launcher_##m_##task(void)

#define _TDL_TASK_WRAPPER_HEADER(m,taskid) 
   for(;;) { _TDL_WaitSignal(_TDL_MODULEID_##m,taskid);

#define _TDL_TASK_WRAPPER FOOTER(m,taskid) 
   _TDL_ACTIVE_TASKS_##m[taskid] >> 3] &= ~(1 << (taskid & 0x07)); } 
\end{verbatim}

The task wrapper abstraction has the benefit of making transparent for the developer the way the intended functionality actually runs on the target platform. The macros implementing the task wrapper can be further customized to separate the tasks of each module into dedicated memory spaces for increased reliability. Thus, they can partition the application code to trap and isolate faulty task implementations. They are also a path for portability by allowing the execution of the functionality code on a different platform through a simple recompilation (with some platform specific parameters).

\begin{quote}
\textbf{Listing 5.16: Task wrappers of module M1 from Listing 5.1}
\end{quote}
\begin{verbatim}
TDL_TASK_WRAPPER(M1, dec) // Wrapper for the actual M1_dec functionality
{
   _TDL_TASK_WRAPPER_HEADER(M1, 0); // Release mechanism
   M1_decImpl(&TDL_M1_Port3_VAL); // Call implementation with internal ports
   _TDL_TASK_WRAPPER FOOTER(M1, 0); // Signal completion
}

TDL_TASK_WRAPPER(M1, inc) // Wrapper for the actual M1_inc functionality
{
   _TDL_TASK_WRAPPER_HEADER(M1, 1); // Release mechanism
   M1_incImpl(&TDL_M1_Port4_VAL); // Call implementation with internal ports
   _TDL_TASK_WRAPPER FOOTER(M1, 1); // Signal completion
}

const _TDL_Task _TDL_TASKS_M1[.TDL_TASKS_COUNT_M1] = {
   TDL_TASK_PROPERTY(M1, dec, 1000),
   TDL_TASK_PROPERTY(M1, inc, 1000) }; // module_name, task_name , wcet
\end{verbatim}
CHAPTER 5. THE PLATFORM ABSTRACTION LAYER

In addition to the task wrappers, we create a task-description table containing the properties of the task such as parent module, task name, and its wcet. As example in the Listing 5.16 we present the task wrappers of module M1.

Formally, we define the task wrapper \( \hat{\tau} \) of a task \( \tau \) as the tuple \((\hat{\tau}, \tau, \hat{d}_\tau, \hat{h}_\tau, \hat{f}_\tau)\), where \( \hat{d}_\tau \) represents a PAL task-declarator, \( \hat{h}_\tau \) represents its header and \( \hat{f}_\tau \) its footer.

5.2.1.7 Modes

The TDL modes have no real correspondent in the ANSI-C world as they describe groups of periodically executed activities. A typical implementation of a similar concept consists mostly of a control loop containing a set of computations and a waiting mechanism, such as sleep or delay. In our case, the E-Code already contains the timing and behavioral properties of the TDL modes, meaning that each mode has a corresponding E-Code sequence that specifies the actions to perform in that mode and their exact timing. Thus, we define a mode description table to retain the additional mode properties such as mode name, E-Code entry point, and period.

For systems where dynamic scheduling is possible (see Section 5.4), we have to minimize the run-time scheduling overhead to increase the run-time performance. Therefore, we take advantage of the periodic nature of a TDL mode and its static properties (number of tasks, actuators, data dependencies between tasks, sensors and actuators). We construct a static dispatch table for each mode of a module by using the EDF algorithm. Nevertheless, the same approach can be used for any other algorithm implemented by the TDL Scheduler (see Section 5.4). For distributed systems the dispatch tables already contain the constrains on the task schedule from the communication schedule (see Section 5.5).

A dispatch table for a mode \( m \) is a set of pairs \((\tau, \text{deadline})\), where \( \tau \in \text{Tasks}[m] \) is a task invoked in the mode \( m \) and deadline represents a relative time \( t_\tau \) in relation with the mode period \( t_\tau < \Pi_m \), such that the task is dispatched according to the EDF algorithm at a moment \( t < t_\tau \). Note that the task is not required to complete at the moment \( t_\tau \) (for details see the Section 5.4).

5.2.1.8 E-Code

The E-Code contains the compiled TDL source code that describes the interaction between the external environment and the software system. The TDL compiler generates by default a binary E-Code file for each module. This file contains several sections that describe the ports, tasks, and modes along with their properties and a sequence of E-Code instructions. As most embedded systems do not provide real-time means for accessing a file at run-time and the E-Code is a crucial part of a real-time application, we perform a C based encapsulation of the E-Code into the module wrapper.

We define two encoding mechanisms depending on the allocation of E-Code instructions: fixed or flexible. We first assign an opcode to each E-Code instruction as
5.2. TDL MAPPINGS (TDLMP)

depicted in Table 5.3. When using the fixed allocation option, we use three parameters to accommodate the longest E-Code instructions, e.g. IF, and we pad the other instructions with a reserved value for unused parameters. For clarity we define a macro for each instruction that maps the macro name and its parameters to the actual opcode of the instruction and its three parameters. Using these macros, we write the E-Code instruction sequence as an array in the module wrapper file. The advantage of using this approach is that we can benefit from platform specific optimizations on decoding of E-Code instructions at run-time. A fixed size E-Code instruction can be aligned to machine specific words or double words and processed faster. As an alternate mechanism, the flexible allocation encodes each E-Code instruction into a reduced set of bytes, without any padding of its parameters. The runtime system decodes the instruction and then fetches its parameters. This approach fits best the systems that have limited memory.

<table>
<thead>
<tr>
<th>E-Code instr.</th>
<th>Opcode</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOP</td>
<td>0</td>
<td>no parameters</td>
</tr>
<tr>
<td>FUTURE</td>
<td>1</td>
<td>l, address $f_a$, time $f_t$</td>
</tr>
<tr>
<td>CALL</td>
<td>2</td>
<td>number of driver $d$</td>
</tr>
<tr>
<td>RELEASE</td>
<td>3</td>
<td>number of task $\tau$, number of driver $d[\hat{\tau}]$, deadline</td>
</tr>
<tr>
<td>IF</td>
<td>4</td>
<td>number of guard $g$, address $a_{true}$, else address $a_{false}$</td>
</tr>
<tr>
<td>JUMP</td>
<td>5</td>
<td>target address $a'$</td>
</tr>
<tr>
<td>RETURN</td>
<td>6</td>
<td>no parameters</td>
</tr>
<tr>
<td>SWITCH</td>
<td>7</td>
<td>number of mode $m'$</td>
</tr>
</tbody>
</table>

Table 5.3: Summary of E-Code instructions

The Table 5.3 presents the default opcodes of the E-Code instructions. In contrast with the binary form of the E-Code, where the opcodes have strict semantics, in the encapsulated E-Code the opcodes of the E-Code instructions can be changed without influencing the semantics of the E-Code instruction. The reason is that the E-Code instructions are explicit through C macros, and we can assign them arbitrary underlying binary correspondents. Therefore, by changing the opcodes we can favor certain compiler optimizations, e.g. -O2 or -Os for GCC, that may be used to generate faster or smaller code for the target platform. One typical optimization is to change the opcodes to match the frequency of the instructions within an E-Code program, and then optimize the order in which the E-Code interpreter decodes each instruction.

Using the opcodes from the Table 5.4 and decoding the instructions in the descending order of their opcodes, we reduce the number of comparisons performed by the CPU of the target platform when interpreting the E-Code. When using hashes or lookup-tables for decoding the E-Code instructions, the opcodes have no influence on the decoding speed.
CHAPTER 5. THE PLATFORM ABSTRACTION LAYER

<table>
<thead>
<tr>
<th>E-Code instr.</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL</td>
<td>7</td>
</tr>
<tr>
<td>RELEASE</td>
<td>6</td>
</tr>
<tr>
<td>FUTURE</td>
<td>5</td>
</tr>
<tr>
<td>IF</td>
<td>4</td>
</tr>
<tr>
<td>JUMP</td>
<td>3</td>
</tr>
<tr>
<td>RETURN</td>
<td>2</td>
</tr>
<tr>
<td>SWITCH</td>
<td>1</td>
</tr>
<tr>
<td>NOP</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.4: Frequency based optimization of E-Code opcodes

5.2.1.9 Code generation

We previously described the principal elements of a module wrapper. We introduce in the Listing 5.17 an algorithm that summarizes these elements and generates the module wrapper for a TDL module $M$. We assume that the user-provided functionality code for a module contains the correct opaque type definitions, prototypes for the implementation of each function, the corresponding set of includes for external functionality, compiler flags and any other auxiliary code required for an error-free C compilation. In the presented algorithm we use the function emit to generate the actual C code. For simplicity, we skip the optimizations for drivers, guards and E-Code.

Listing 5.17: Algorithm for generating a module wrapper

```c
// include user defined header file of the module
emit(includes)

// create port mappings
∀p ∈ P_{τi}[M] ∪ P_{σ}[M] ∪ P_{α}[M]P_{o}[M]
  emit(θ(p) Ports(p)) // task input or status port, sensor and actuator ports
∀p ∈ P_{τo}[M]
  emit(θ(p) Ports(p)_i) // internal values
  emit(θ(p) Ports(p)_v) // visible values

// create table of tasks
TT ← ∅ // task table
r[M] ← ∅ // flag for denoting a running task
∀τ ∈ Tasks[M]
  TT ← TT ∪ {τ, d_τ, wcet(τ), r_τ}
emit(TT)

// create task wrappers
∀τ ∈ Tasks[M]
  emit(TaskDeclarator(τ)) // Task wrapper for τ
```
5.2. TDL MAPPINGS (TDLMP)

\[\text{emit}\left(\{\hat{h}_\tau; \text{Invoke}(u[\tau]) ; \hat{f}_\tau\}\right)\] // task wrapper body

// create table of modes
\[MT \leftarrow \emptyset\] // modes table
if \(m = m_s\) then // place the start mode first
\[MT \leftarrow MT \cup \{m,\ \text{start}_m,\ \Pi_m\}\]
end if
\[\forall m \in \text{Modes}[M] \setminus m_s\]
\[MT \leftarrow MT \cup \{m,\ \text{start}_m,\ \Pi_m\}\]
\[\text{emit}(MT)\]

// add dispatch tables
\[DT[M] \leftarrow \emptyset\]
\[\forall m \in \text{Modes}[M]\]
\[DT[M] \leftarrow DT[M] \cup DT_m\]
\[\text{emit}(DT[M])\]

// execute drivers
\[\text{execute(Algorithm from Listing 5.8)}\]

// generate driver wrappers
\[k \leftarrow 0\]
\[\text{emit}\left[\text{switch}(d)\right]\]
\[\forall \sigma \in \text{Sensors}[M] \] // sensor getters
\[\text{emit}(\text{Header $\hat{\text{Drivers}}_k(d[\sigma])$ Footer})\]
\[k \leftarrow k + 1\]
\[\forall \alpha \in \text{Actuators}[M] \] // actuator setters
\[\text{emit}(\text{Header $\hat{\text{Drivers}}_k(d[\alpha^*])$ Footer})\]
\[k \leftarrow k + 1\]
\[\forall \alpha \in \text{Actuators}[M] \] // actuator updates
\[\forall \sigma \in \text{Sensors}[M]\]
if \((\exists \gamma(\sigma, \alpha))\) then // we have an actuator updated with sensor values
\[\text{emit}(\text{Header $\hat{\text{Drivers}}_k(d[\alpha_k])$ Footer})\] // copy sensor values
\[k \leftarrow k + 1\]
end if
\[\forall \tau \in \text{Tasks}[M]\]
if \((\exists \gamma(\tau, \alpha) \land \exists \Delta(\alpha, \tau_o))\) then // we have an actuator updated with task output
\[\text{emit}(\text{Header $\hat{\text{Drivers}}_k(d[\alpha_k])$ Footer})\] // copy task output values
\[k \leftarrow k + 1\]
end if
\[\forall \tau \in \text{Tasks}[M] \] // task drivers
\[\text{emit}(\text{Header $\hat{\text{Drivers}}_k(d[\tau^*])$ Footer})\]
\[k \leftarrow k + 1\]
CHAPTER 5. THE PLATFORM ABSTRACTION LAYER

\[
\text{emit}(\text{Header } \overset{\text{Drivers}}{k}(\text{RELEASE}(\tau))) \text{ Footer}) \hspace{1em} \text{// start driver}
\]
\[
k \leftarrow k + 1
\]

\[
\text{emit}(\text{Header } \overset{\text{Drivers}}{k}(d[\tau^{TDLComm}])) \text{ Footer}) \hspace{1em} \text{// stop driver}
\]
\[
k \leftarrow k + 1
\]

\[
\forall m \in \text{Modes}[M] \hspace{1em} \text{// release driver}
\]
\[
\text{emit}(\text{Header } \overset{\text{Drivers}}{k}(d[\tau^r_m])) \text{ Footer})
\]
\[
k \leftarrow k + 1
\]

\[
\hspace{1em} \text{// encapsulate E−Code for closed systems}
\]
\[
\text{emit}(E[M])
\]

\[
\hspace{1em} \text{// generate guard wrappers}
\]
\[
\text{emit}(\text{switch}(g))
\]
\[
k \leftarrow 0
\]
\[
\forall m \in \text{Modes}[M]
\]
\[
\forall g \in \text{Guards}[m]
\]
\[
\hspace{1em} \text{emit}(\text{case } k : \text{return } u[g])
\]
\[
k \leftarrow k + 1
\]

\[
\hspace{1em} \text{// finalize module wrapper}
\]
\[
\overset{\text{Modules}}{M}[M] \leftarrow (M, TT[M], DT[M], MT[M], E[M], \overset{\text{Tasks}}{M}, \overset{\text{Drivers}}{M}, \overset{\text{Guards}}{M}, M_{\text{INIT}})
\]

We first generate the includes section, containing the references to the core of the TDL runtime system, the PAL definitions, the dependency solver file, and the user functionality. We then generate the ports section, where we statically allocate the ports of the entities defined in the current module. Following, we generate the tasks sections, in which we first declare the tasks table, with their properties, and then the task wrappers. We add the modes section, with the table containing their properties and the helper dispatch tables (their run-time purpose is described in the next section). For closed systems, where the run-time loading of the E-Code is not possible, we translate the E-Code into a C array and include it into the generated module wrapper. In the drivers section, we create the drivers and emit their implementation. We add the guards section, with the guard wrappers containing references to the user-defined guard functions.

As an addition to the TDL semantics, in the case when the sensors or actuators used within a module require some special initialization, we provide a standard method to run such initialization before the execution of the corresponding module. This extension to the TDL semantics is available as a standard C function without parameters defined along with the functionality code of that module and using a special naming convention void \( M_{\text{INIT}}() \). The PAL provides implicitly an empty initializer function for each module.
Finally, we add the module descriptor containing references to all previously generated C structures. The module descriptor is the only entity visible from a module, aside from the public ports. Through it, the E-Machine can access all relevant module information such as tasks and modes tables, the E-Code and drivers, the task wrappers and the user-defined guards.

<table>
<thead>
<tr>
<th>TDL elements</th>
<th>correspondent in C glue-code</th>
</tr>
</thead>
<tbody>
<tr>
<td>module</td>
<td>glue-code file</td>
</tr>
<tr>
<td>import</td>
<td>symbolic references</td>
</tr>
<tr>
<td>ports</td>
<td>variables</td>
</tr>
<tr>
<td>sensors</td>
<td>sensor getters</td>
</tr>
<tr>
<td>actuators</td>
<td>actuator setters</td>
</tr>
<tr>
<td>tasks</td>
<td>task wrappers</td>
</tr>
<tr>
<td>guards</td>
<td>Boolean functions</td>
</tr>
<tr>
<td>modes</td>
<td>- (contained in E-Code)</td>
</tr>
</tbody>
</table>

Table 5.5: Summary of TDL mappings to C language

The table 5.5 summarizes the mappings of TDL elements to corresponding C language elements. Through the glue-code, we bundle at compile time the timing specifications of a TDL module, expressed in E-Code, with the TDL mappings of its entities and the user-defined functionality.

5.2.2 Module stubs

The TDL import relationship allows data transfers between entities of different modules, regardless of their physical placement. In distributed systems, a service provider module and its clients (there can be more than one module importing a service provider module) may be placed on different nodes. The transparent distribution concept of TDL defines in this case a stub of the service-provider module on each node that does not contain it but contains one of its clients.

Similar with a module wrapper, the PAL defines for each stub module a corresponding stub-module wrapper. A stub module has a wrapper derived from the service-provider module it replaces on the node containing the client module. However, without tasks to execute there are no task-wrappers, there are also no guards because the other entities of the service provider module, such as sensors and actuators, are not available either. The wrapper of the stub module contains only the set of public output ports, the modes table, and a subset of the drivers and the E-Code of the service-provider module.

To avoid costly value-copying operations between modules, the drivers of a client module contain symbolic references to ports from the service-provider module. We rely for this operation on the global C namespace available when compiling the service-provider and client modules together (this mechanism works when the client and the
service-provider modules reside on the same node).

When the client module and the service-provider module are on two different nodes, we compile the stub together with the client module. The stub-module wrapper provides identical-named ports with the service-provider module. Thus, the symbolic references from the client-module wrapper point to the ports from the stub-module wrapper. There are no naming conflicts between a service-provider module and any of its stubs as they are never compiler together on any node (the dependency solver file handles this problem).

As a result of this approach, the port references of the client modules always point to the ports of the imported service-provider module regardless of the final distribution of modules in a distributed system. However, we have to maintain synchronized through the TDLComm layer the ports of all stubs with the corresponding service-provider module.

For the stub-module wrappers, we generate the E-Code using the algorithm presented in the Listing 4.1 from Section 4.1.2 and the STUB property of the module M. As a result, the E-Code of a stub module contains only the driver calls that update the public output ports (their visible set), along with the timing information of the service provider module. The E-Code instruction set of a stub module is reduced to the CALL, FUTURE, and RETURN instructions. Accordingly, the set of drivers contains only the task termination drivers, because the stub contains no functional entity and all other drivers from the service provider module would have no correspondent. The stub-module wrapper does not contain any reference to user-defined functionality such as tasks, or guards. The ANSI-C plugin generates automatically the stub-module wrappers and includes them in the compiler-configuration file (i.e., makefile) of the corresponding nodes. The algorithm from Listing 5.18 is a reduced version of the algorithm generating the module wrapper for a service-provider module.

Listing 5.18: Algorithm for generating a stub-module wrapper

```plaintext
emit(includes) // include user defined header file of the module
∀p ∈ P_o[M], ∃β(p) // create port mappings for public output ports
   emit(θ(p) Ports(p)_i) // internal values from TDLComm
   emit(θ(p) Ports(p)_v) // visible values to client modules

MT ← ∅ // create table of modes without dispatch tables
if (m = m_s) then // place the start mode first
   MT ← MT ∪ {m, start_m, Π_m}
end if
∀m ∈ Modes[M] \ m_s
   MT ← MT ∪ {m, start_m, Π_m}
emit(MT)

// create drivers
execute(Algorithm from Listing 5.8)
```

134
5.2. TDL Mappings (TDLMP)

\( k \leftarrow 0 \) // generate driver wrappers
\( \text{emit}(\text{switch}(d)) \)
\( \forall \tau \in \text{Tasks}[\text{M}] \) // task drivers
\( \text{emit}(\text{Header Drivers}_k(d[\tau])) \text{Footer}) \)
\( k \leftarrow k + 1 \)

// encapsulate E–Code for closed systems
\( \text{emit}(E[\text{M}]) \)
// finalize module wrapper
\( \text{Modules}[\text{M}] \leftarrow (\text{M}, MT[\text{M}], E[\text{M}], Drivers[\text{M}]) \)

From the E-Machine point of view, as the E-Code of a stub module wrapper does not refer any tasks, or any data structure different from a normal module, we simply treat it as any other module. Without any computational tasks, it also has no influence on the run-time scheduling of tasks on the client node; therefore, its inclusion on remote nodes has a negligible run-time overhead.

### 5.2.3 Module Dependencies

The import relationship introduced by TDL creates dependencies between modules. At PAL level we recognize them as references in the wrapper code of a module to ports from other modules that may even reside on a different node.

We address this problem at the TDL compilation phase, where the ANSI-C plugin performs a final dependency analysis after all modules are compiled. At this point, the TDL compiler acknowledged their import-export relationships and the plugin analyzes the placement of the modules on nodes.

In the case of a single node system, the modules benefit from the global C namespace and the plugin generates a list of module wrappers with references to their module descriptors. Through the list of module descriptors, the E-Machine may load, initialize, and execute the contents of the module wrappers.

In the case of a distributed system, the plugin generates a list of module wrappers for each node. It substitutes in the list the modules not available on the current node with their stubs. In addition, it generates the structure of TDL packets describing the communication requirements of this node via the TDLComm layer. We detail this procedure in Sections 5.5.3 and 5.5.4.

The plugin also generates for each node a function to load the modules, which provides at run-time the references to the compiled module wrappers. Finally, it places the dependency information of the compiled set of TDL modules, along with the communication requirements (in the case of distributed systems) in modules.c dependency solver file. This file provides at run-time the information available only at the compilation time of the C source files (i.e., the structure of the system and which module wrapper files were compiled together on each node).
CHAPTER 5. THE PLATFORM ABSTRACTION LAYER

5.3 Runtime Resource Management (RTRM)

The RTRM part of PAL mediates the interaction between the TDL runtime environment (i.e., E-Machine, TDL Scheduler, and TDLComm) and the underlying RTOS. To execute a set of TDL modules on a target platform, along with their representation into a corresponding language such as C, we require logically time and task management functionality from the underlying platform. As TDL is strictly a time-triggered language, for the platforms where the hardware sensors and actuators use interrupt handlers, we require that either the interrupt overhead is negligible or the time resolution of the platform is high enough so that we can measure the amount of time spent servicing the interrupts and consider it for scheduling decisions.

5.3.1 Time

Regarding the time management functionality, we define two types of time-related actions: sampling and planning. The sampling action provides us with as accurate as possible timestamps from the hardware, which we use to measure the amount of time used for task computations, E-Code processing, interrupt handlers, and scheduling. In practice, all platforms provide at least one function for this purpose such as `getTime`. However, the clock resolution can vary between hardware platforms and even between the different RTOS that may be available for a specific platform. Thus, we use two generic functions to access the high-resolution hardware timer of the platform and the RTOS internal clock (i.e., system ticks). We map the two functions to the actual platform specific functions using the configuration file of the runtime environment. The RTOS related function allows us to map the logical time of TDL to platform time, whereas the high-resolution timer related function measures the exact task execution times.

The planning action provides us with the means of activating specific sequences of code after a predefined amount of time. Such functionality exists in all RTOS; however, its design may be different. For example, OSEK compatible platforms use the notion of alarms that can be used to activate a task or other functionality, either in a single-shot mode (after a specified amount of time has expired) or in a periodic way. Other systems support the notion of sleep with different clock resolutions from milliseconds to nanoseconds. A typical implementation of such functionality relies on either a software counter or a hardware timer, which decrements periodically its time register and issues an interrupt when the register reaches zero. In a similar way with the sampling function mapping, we use a generic function to plan the execution of the E-Machine or the TDL scheduler after specific amounts of time (i.e., until logical or physical actions have to be performed on the task set of the application) and execute the user functionality for a predetermined amount of time.

Figure 5.6 presents the basic usage of sampling and planning actions. First, the TDL Scheduler queries the platform time through PAL. Assuming the platform time
5.3. RUNTIME RESOURCE MANAGEMENT (RTRM)

maps to a logical time, meaning that it is time to perform a set of logical actions such as

task releases or actuator updates, the TDL Scheduler invokes the E-Machine to execute

an E-Code block of a module. The E-Machine interacts through the PAL-TDLmp with

the E-Code of the module, its drivers, guards, and ports. It then releases a set of tasks
to the RTOS and completes its execution. Following, the TDL Scheduler decides which
task to execute from the list of active tasks (including the newly released ones), plans
when to take the next scheduling decision, dispatches the task, and sleeps until the
planned moment.

From the point of view of the upper control layers (i.e., the TDL Scheduler and the
E-Machine), the sampling and planning actions are both in environment time (e.g., micro-
seconds), whereas their actual implementation on a platform may use a corresponding
conversion function from platform time (e.g., CPU clock ticks, network messages,
internal counters). The sampling operation may provide timestamps with an accuracy
higher than the basic time unit of TDL (i.e., the microsecond), which may be used
to provide more accurate debugging information. On the other hand, the planning
operation must provide a discrete mapping of the logical time to system time. The

Figure 5.6: Usage of PAL for time sampling and planning
system time is generally expressed as a number of system clock ticks. Thus, the logical
time will not progress linearly, but in steps of microseconds (hundreds or thousands of
microseconds) depending on the time resolution of the system ticks.

Although TDL is platform independent, not all combinations of applications and
platforms can be supported because of limitations in the underlying clock resolution.
In the case, when at least one TDL module has a time-resolution requirement higher
than the platform sampling and planning capabilities, the compiler, plugins, or the
runtime system inform the developer of the insufficient time resources of the platform.
The developer is also informed when the platform clock resolution although higher than
the requirements of all modules cannot be mapped to the logical environment time, e.g.
when the platform performs the clock updates with every 250us and the modules have
logical actions at 300us intervals. It must be possible to express the logical time as an
integer multiple of the platform time.

5.3.2 Tasks

The task management functions are more complex, as they depend on the TDL task
mapping mechanism we use on a platform. We previously defined the task wrappers
to encapsulate the user functionality with possible parameters into a portable struc-
ture. We also defined the possible physical task representations at the RTOS level:
native, threads, and processes. For platforms without native support of TDL tasks
such as RT-POSIX, we create at load-time (i.e., when the runtime environment loads
and initializes each module) the corresponding physical elements such as threads and
processes to handle each task wrapper. On platforms supporting both approaches, we
prefer the usage of threads to benefit from the common memory space that provides
better performance than the classical inter-process communication primitives of the
platform.

From the logical point of view, we require the release task functionality, which
activates a task wrapper and prepares it for execution (i.e., the task enters into a ready
state). As the E-Machine has to perform the release operation, we define a generic
Release_Task macro, which we map on the corresponding C function available on the
target platform. For thread or process-based platforms, we implement a waiting loop
in each task wrapper using their customizable header and footer. The release action in
this case is a simple message or semaphore post, which marks as ready the task wrapper
for the corresponding released task.

From the physical point of view, we may either leave the RTOS scheduler to schedule
the active tasks or implement our own scheduling scheme on top of the RTOS scheduler.
In the second case, we require two actions for manipulating the set of active tasks:
dispatch and preempt. TDL prohibits the forced termination of a task, when it exceeds
its wcet, because it may leave the system in an inconsistent state. However, using
a special function of the TDL runtime environment we can maintain a task into a
preempted state for debugging purposes.
5.4 TDL Scheduler

The TDL Scheduler represents the core controlling entity of the TDL runtime environment. Although not a true part of PAL, we present it here, as its role and interaction with the RTRM layer and the underlying RTOS are crucial for the run-time behavior of a TDL application. It represents the mediator between the logical time and the soft time, the component that "knows" both time lines and ensures that all operations follow their "right" course.

Depending on the progression of the logical time and the scheduling algorithm, it decides when to execute the functionality of the E-Machine, the TDLComm layer, or the user-tasks. The logical actions of the E-Machine or the TDLComm layer always have precedence in relation with the dispatching operations of the user-level tasks. Also as a prerequisite, we require that on any platform the functionality of the TDL runtime environment (including the E-Machine, TDL Scheduler, and TDLComm) always has the highest priority among all platform tasks, as depicted in the Figure 5.7.

In the Listing 5.19 we present the core algorithm of the TDL Scheduler. It implements a control loop, which operates with two time offsets: one for the invocation of the E-Machine, and the other for the scheduling operations of the user tasks. In the case of distributed systems, the TDL Scheduler executes first the TDLComm layer to update the internal output ports of the stub modules. We detail the operation of the TDL-Comm layer in the Section 5.5. Afterward, the TDL Scheduler determines whether any of the available modules has a logical activity to perform and invokes the E-Machine in such case. At this point, we refer to the main block of the algorithm of the E-Machine from Listing 4.5, which determines the module to execute and updates the $\delta_{\text{min}}$ waiting
interval (the E-Machine and the TDL Scheduler do not work in parallel, but instead the TDL Scheduler calls the E-Machine when appropriate). Following the execution of the E-Machine, the TDL Scheduler selects one task to dispatch from the list of active tasks, using a specific algorithm (e.g. RM, EDF). It also records the time δS until the next preempt/dispatch operation and determines the minimum wait interval between logical actions and rescheduling operations. After the dispatched task completes or the waiting interval expires, the TDL Scheduler resumes its computations and updates the logical time and the waiting intervals for the E-Machine and rescheduling.

Listing 5.19: Core algorithm of the TDL Scheduler

```cpp
// initialize E−Machine waiting interval
δE ← 0
// initialize TDL Scheduler waiting interval
δS ← 0

while (true)

if (∃ TDLComm) // if we have a distributed system
    invoke(TDLComm) // execute the TDLComm functionality

// perform the logical actions
if (δE = 0) // execute E−Machine for all modules
    invoke(E−Machine) // that have to react at this time
    δE ← δmin // update from E−Machine δmin
    δS ← 0 // force rescheduling

end if

if (δS = 0) // we have to perform rescheduling
    // pick one active task to schedule using EDF or other algorithm
    (τ, δS) ← select_to_schedule(Tasksa[Modulesa[N]]) // τ will be dispatched next
    // δS represents the time till the task deadline or next preemption

end if

// check whether we have to perform logical actions before rescheduling
if (δE ≤ δS) // ex. when updating actuators faster than task invocations
    dispatch(τ) // prepare τ for execution
    wait_completion(δE) // wait till task ends or at most δE
    t ← t + min(δE, elapsed) // update logical time t
    δE ← 0 // δE passed, thus no more waiting for E−Machine
else // we have to perform rescheduling before other logical action
    dispatch(τ) // prepare τ for execution
    wait_completion(δS) // wait till task ends or at most δE
    t ← t + min(δS, elapsed) // update logical time t
    δE ← δE − δS // update elapsed time for E−Machine
    δS ← 0 // rescheduling in the next cycle
end if

end while
```
5.4. TDL SCHEDULER

We currently support three optimal algorithms for TDL task scheduling: Rate Monotonic (RM) \cite{Liu1973} and Deadline Monotonic (DM) \cite{Leung1982} for fixed-priority scheduling, and Earliest Deadline First (EDF) \cite{Liu1973} for dynamic-priority scheduling. Note that on a single processor system, the deadline of each task is equal to the end of its LET (i.e., the period of the task). However, on distributed systems we use a TDMA\textsuperscript{1} protocol for message scheduling \cite{Farcas2005a,Farcas2005b} and the deadline of a sender task is the sending time of the corresponding message; receiver tasks are not affected by distribution, because they interact with the stub modules, which obtain the messages before the end of LET of a sender task. We present distribution details related to TDLComm in Section 5.5.

5.4.1 Rate Monotonic scheduling

As most RTOS are based on fixed-priority scheduling, in the case when the schedulability analysis for a set of TDL modules holds for RM or DM algorithms, we can simply assign the task wrappers the corresponding priorities, release the TDL tasks from the E-Machine by activating their task wrappers, and let the RTOS scheduler perform the actual scheduling decisions. However, TDL allows us to define complex functionality with mode-switches, where TDL tasks can have different frequencies and we have to ensure for their functionality code a priority inverse proportional with their period (RM) or deadline (DM).

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure5.8.png}
\caption{Example of RM scheduling of three TDL tasks}
\end{figure}

On platforms such as OSEK, which do not allow changes of the platform task priorities at run-time, implementing TDL mode-switches may seem impossible. Nevertheless, let’s not forget that a TDL task is not equal with a platform task (or thread/process).

\footnote{\textsuperscript{1}Time Division Multiple Access}
A TDL task is mapped to one or more task wrappers. Therefore, for all TDL tasks with different frequencies within modes, we create a corresponding number of identical task wrappers with adequate priorities and naming derived from the mode name. Using this set of task wrappers for each TDL task, although we cannot change at run-time the priorities of the task wrappers, we can perform TDL mode-switches by activating the right task wrapper for a task within the desired mode. In such cases, the mapping of TDL tasks to task wrappers, which represent the actual platform tasks, is not 1:1, but 1:$m'$, where $m'$ is the number of modes in which a TDL task has different periods.

For all other systems, where it is possible to change the priority of a platform task at run-time, one task wrapper for each task is enough, because at mode switches we can set the priority of the task wrapper according to the requirements of the new mode.

For large systems, with a high number of modules and tasks, the number of priorities required for a correct RM or DM scheduling may exceed the number of priorities available on the platform. Fortunately, prior research [Lehoczky and Sha, 1986] proved that for the Rate Monotonic algorithm and deadlines equal to periods, using just 256 priority levels is almost as good as unlimited priorities. An example of RM scheduling of three tasks is presented in Figure 5.8. The code and memory overhead of the task wrappers is minimal, as the actual functionality code exists in only one place and the task wrappers introduce just a few more bytes for the functionality call and the corresponding entry in physical task list of the RTOS.

### 5.4.2 Earliest Deadline First scheduling

The Earliest Deadline First algorithm allows better processor utilization than RM but it is not directly supported as a native scheduling scheme in most RTOS, because they do not support explicit timing constraints at the task level. Furthermore, there are several misconceptions that unfairly favor RM [Buttazzo, 2003]. For EDF or any other dynamic-priority scheduling scheme, we use the TDL Scheduler in addition to the E-Machine. Its purpose is to manipulate the active task set and to assign the CPU to one active task by using the dispatch and preempt operations.

We first define three priority levels: low, medium and high. Using just these three levels, we can enforce on most systems an EDF scheduling policy on top of the classical scheduling policy provided by the priority-based RTOS. The basic mechanism is to dynamically change at run-time the priorities of each task wrapper according to the intended preemption or dispatch operation we want the RTOS to perform. We assume that the RTOS scheduler always executes higher-priority functionality by preempting lower-priority functionality.

Therefore, we initially assign to all task wrappers the low priority. At each logical moment, the E-Machine benefitting from its highest priority preempts all tasks, executes the E-Code, plans its next invocation after the $\delta_{\text{min}}$ time as described in the algorithm from Listing 4.3, and sleeps until that moment. Following the E-Machine, the TDL scheduler adds the released tasks into its list of active tasks ordered by deadlines,
and picks the task that has the closest deadline. It changes the medium priority of a previously executing task wrapper into the low priority, and the priority of the selected task wrapper from low to medium priority.

Then, the TDL scheduler plans its next invocation at the minimum from the remainder of the selected task wcet and the next release time, and sleeps until that moment. At this point, all task wrappers except the selected task have a low priority. Therefore, the RTOS scheduler dispatches the selected task with the medium priority on the CPU. By changing the priorities from low to medium, the TDL scheduler enforces a dispatch operation, whereas through a medium to low change it maintains the other tasks into a preempted state.

This mechanism is applicable to all RTOS that allow the dynamic changes of the priorities of platform tasks at run-time. There are exceptions, such as OSEK, where such changes are not possible. However, a non-portable approach exists: depending on particular implementations of the OSEK standard, when all task wrappers have the same low priority, the task chaining functionality may be used to enforce a dynamic scheduling policy.

**Hybrid EDF scheduling**

To support parallel composition of modules and to execute them on the same node, we have to allocate a small fraction of the CPU-time to each module of the node. Traditionally we would solve this problem via a time-sharing mechanism, or CPU partitioning. With this approach, we would allocate for each module a percentage of a previously computed time quantum equal with the load the module generates on the CPU. However, this mechanism introduces a high context-switch overhead on the running system, because of the infinitesimal time-quantum required to implement this mechanism.

We adopt a different approach by using EDF to schedule the tasks from all modules. Conceptually the modules are simply logical constructs; therefore, from a scheduling perspective we can treat equally all tasks from all modules. The only property that matters for EDF scheduling is the deadline of the task, regardless of the parent module. The overhead of EDF scheduling on traditional systems is higher than classic static priority based scheduling (with RM). In the case of TDL, as the application is strictly time-triggered, and we know at compile time all task and mode periods, we can benefit from this prior known information to reduce the run-time scheduling overhead.

We introduce in Listing 5.20 a lightweight algorithm for implementing EDF scheduling of multiple modules on the same node. We use the precompiled dispatch tables of each mode of a module, and therefore have to perform EDF scheduling only among a single task per module having the closest deadline.

We note with $t$ the current absolute logical time and with $t_m$ the absolute logical time when the mode $m$ started its current period. The dispatch table $DT_m$ of a mode $m$ contains a set of entries, each entry consisting of a relative deadline since the beginning of the mode and a task. On single node systems, the deadlines are simply multiples
of the task periods (last entries have the form \(\{\pi_m, \tau\}\), denoting the deadline equal with the mode period). For each module \(M\), we have an associated dispatch table index \(DT^i\), which points to a task entry in the dispatch table \(DT_m\) that has a deadline closest to the current logical time. Each time the E-Machine performs a mode switch in a module, or starts a new cycle of a mode, it resets the index \(DT^i\). We use \(t_m\) to convert from the deadlines relative to the beginning of the mode to the absolute deadlines required by the EDF algorithm.

Listing 5.20: Hybrid EDF using precompiled dispatch tables

```c
// logical time is t
Preempt(\tau_{old}) // suspend the execution of the previously executing task \tau_{old} \in Tasks[M[N]]

\delta \leftarrow \infty // retains closest deadline of a task from all modules
\forall M \in Modules[N] // process all modules of the node N
    while (DT^i < \|DT_m\| and t - t_m \geq DT_m[DT^i].deadline) // skip past entries
        DT^i \leftarrow DT^i + 1
    end while

i \leftarrow DT^i // seek the first active task
while (i < \|DT_m\|)
    if (\delta > DT_m[i].deadline - t_m and DT_m[i].\tau \in Tasks_a[M])
        \delta \leftarrow DT_m[i].deadline - t_m
        \tau_{new} \leftarrow DT_m[i].\tau
        break // exit from the while loop
    else
        i \leftarrow i + 1
    end if
end while
// \tau_{new} is the task with the closest deadline from all modules
Dispatch(\tau_{new}) // start/resume the execution of task \tau_{new}
```

We invoke the TDL Scheduler using this algorithm at the time intervals specified by \(\delta\). In simple cases the waiting intervals between successive invocations of the E-Machine coincide with the rescheduling intervals; however, in the general case the invoke times of the E-Machine and the TDL Scheduler differ. For example, in the case when an actuator has a frequency higher than the task that provides its input, the E-Machine would be invoked more often without the need for rescheduling (system level preemption of user-level tasks). In other cases with multiple modules, the scheduler may be invoked more often than the E-Machine because of the larger and more complex task set.

We first preempt a previously running task, and then assume the waiting interval until the next deadline is infinite. We process all modules and skip the entries in the dispatch table of the currently executing mode that have the deadlines less than the current logical time relative to the beginning of the mode. We then select as the next dispatch-able task the first task that is active in the dispatch table, as it would
have the closest deadline. We cannot increment the dispatch index at this point as it could be that more tasks have the same deadline but have not been released yet (the alternative of keeping track of both released and running tasks requires twice the amount of memory than the simple set of active tasks $\text{Tasks}_a[M]$). We update $\delta$ with the current closest deadline and the supposed next task to dispatch $\tau_{\text{new}}$. After we iterate through all modules, we obtain the smallest dispatch interval until we have to invoke the scheduler again.

The complexity of the algorithm is $O(||\text{Modules}[N]||)$, where $||\text{Modules}[N]||$ represents the number of modules on the node. There is no need for any run-queue as the mode dispatch tables already contain the tasks in the right order; thus, we only need for each module an index pointing to the task to dispatch from the current mode’s dispatch table.

### 5.5 TDLComm layer for transparent distribution

The TDLComm layer is available only in distributed systems, where it abstracts the mechanisms for deterministic exchange of information between the nodes, and acts as foundation for transparent distribution of TDL modules (recall from Section 3.3.2). As a third part of PAL, it provides to the developer of TDL modules an abstraction over the final integration of the modules and the physical placement of modules in the system.

From the platform point of view, the TDLComm layer provides an abstraction over the following services: packet exchange used for the transmission of data packets, clock synchronization, communication protocol, and fault-tolerance. We introduce two communication protocols that are targeted to different classes of applications, and discuss each service in the following subsections.

**Conceptual overview.** For the smallest influence on the overall performance of the distributed system, the TDLComm functionality relies at run-time on pre-computed communication scheduling. Thus, we extend the TDL compiler with a communication-schedule generator plugin that provides the required scheduling information. As depicted in Figure 5.9 [5.9] it analyzes the TDL modules with their dependencies and determines the required message exchange between service provider and client modules.

As input to the communication-schedule generator tool we provide the description of the distributed platform and the placement of modules on nodes. The platform description contains the available number of nodes, the topology of the network connecting the nodes (e.g. several buses), and the properties of the communication channels (e.g. bus rate, minimum and maximum packet sizes).

An example input file for the tool is available in Listing 5.21 [5.21]. Here the two nodes are named master and slave, and they hold three modules, the first node with module M1 and the second node with the modules M2 and M3. The properties of the CAN bus that interconnects the two nodes are presented at the bottom of the configuration file.
CHAPTER 5. THE PLATFORM ABSTRACTION LAYER

Figure 5.9: TDL tool-chain for distribution

Listing 5.21: Distributed system configuration file

```
# network configuration:
tdl.bus.nodes = 2 # tdl.bus.nodes = nofNodes
tdl.bus.nodes.0 = MASTER # tdl.bus.nodes.i = host[:port]
tdl.bus.nodes.1 = SLAVE

# modules to nodes assignments
tdl.bus.modules = 3 # tdl.bus.modules = nofModules
tdl.bus.modules.0 = M1:0 # tdl.bus.modules.i = moduleName:nodeID
tdl.bus.modules.1 = M2:1 #
tdl.bus.modules.2 = M3:1

# bus properties
tdl.bus.envelopeBits=44 # tdl.bus.envelopeBits=nofEnvelopeBits
tdl.bus.gapBits=3  # tdl.bus.gapBits=nofGapBits
tdl.bus.minPacketSize=0 # tdl.bus.minPacketSize=nofBytes
tdl.bus.maxPacketSize=8  # tdl.bus.maxPacketSize=nofBytes
tdl.bus.rateHz=100000 # tdl.bus.rateHz=busRate[Hz]
tdl.bus.syncFrameSize=1 # tdl.bus.syncFrameSize=nofBytes
tdl.bus.syncFramePos=end # tdl.bus.syncFramePos=start|end
tdl.bus.clock=1000 # tdl.bus.clock=clock resolution in us
```
5.5. TDLCOMM LAYER FOR TRANSPARENT DISTRIBUTION

The tool analyzes the set of TDL modules, and determines their remote dependencies, by following the import relationships and references to task output ports from modules located on remote nodes. It determines the set of messages required for exchanging the information between producer tasks and consumer entities such as tasks, actuators, or guards. In the case when it finds a feasible communication schedule that satisfies the communication requirements implied by the set of TDL modules and the distributed platform, it generates the schedule covering a network communication cycle of a proprietary TDMA protocol. Theoretically, provided that the relevant characteristics of a networking protocol are available to the tool, our approach for communication scheduling may be used on top of any existing network protocol. We considered for implementation only the CAN and RT-Ethernet networks.

According to the TDMA protocol, any node is allowed to send messages in statically defined slots only. We rely on a mechanism for global clock synchronization over the network, mechanism implemented within the TDLCComm layer. The data exchange model implemented by the scheduling tool adheres to the Producer-Consumer model. The nodes that generate information (the producers), trigger the sending of information over the network. Contrary to the classical Client-Server model, in the Producer-Consumer model the consumers (the nodes that need the information) do not send any requests to the producers. Hence, we reduce the bandwidth requirements of the distributed system and remove the additional constraints on task scheduling from the request messages of the Client-Server model.

The ANSI-C plugin receives the generated table-driven communication schedule (Far-cas 2005b), and includes it through PAL along with the dependency information for each node of the system. From a logical point of view, using the previously computed scheduling information the TDLCComm layer performs at run-time three steps: the encapsulation of port values from service provider modules into packets, the transmission of the packets over the communication medium, and the extraction of stub-port values from the packets received on the client node (see Figure 5.10). As the encapsulation and the extraction of port values to and from packets depend on the communication protocol, we describe them in the corresponding protocol sections. TDLMp facilitates the encapsulation and the extraction steps by abstracting from the actual type and implementation of the module-wrapper ports. Hence, the TDLCComm layer can synchronize the run-time state of a service provider module with all of its stubs without exposing this procedure to the upper runtime components such as the E-Machine.

For its time-triggered transmission and reception of packets, the TDLCComm layer relies on the TDL Scheduler to invoke its functionality at moments of time defined by the pre-computed communication schedule. On the other hand, it provides the clock-synchronization service on each node of the system, and introduces constraints on the scheduling of the tasks that exchange values over the network. Thus, the close cooperation between the TDLCComm layer and the TDL Scheduler is crucial for a successful implementation of the transparent distribution concept of TDL.
Example. Figure 5.11 presents the operational mode of the TDLComm layer. The two modules placed on two nodes of a distributed system exchange information through the output ports of Task 1. The figure omits the clock synchronization service and presents the runtime systems of both nodes with the same view of the logical time.

From a logical point of view over Node 1, the TDL runtime environment executes Task 1 and updates its visible output ports through the termination drivers at the end of its LET. On the physical CPU level, the communication schedule used by the TDLComm layer imposes scheduling restrictions on Task 1 as it has to complete until the moment defined as transmission time of the internal port values. Thus, the TDL Scheduler dispatches and preempts Task 1, so that the task completes at 9 ms. At this point, the TDL Scheduler invokes the functionality of the TDLComm layer, which reads their values either through the stop driver of the task or directly through references to the internal ports of the task. According to the encoding mechanisms defined by the communication protocol, it encodes the internal port values into datagrams and then packets. It then sends the packet through the communication channel to the second node.

On Node 2, the stub of Module 1 runs in parallel with the Module 2. Around 10 ms, the physical layer of the second node receives the packet sent from the first node. The communication schedule specifies a completed receive action at 11 ms; thus, the TDL Scheduler of Node 2 invokes the TDLComm layer of this node to retrieve the packet, and decode it.

The TDLComm layer through TDLMp places the extracted port values into the internal ports of the Task 1 stub. At the end of the LET of Task 1 stub, the E-Machine of the second node executes the termination drivers of the stub task, and updates the visible port values of the stub from the internal ports. Following, the release driver of Task 2 uses the visible port values as if Task 1 would run on the same node. Hence, the TDLComm layer successfully implements the transparent distribution concept of TDL.
5.5. TDLCOMM LAYER FOR TRANSPARENT DISTRIBUTION

Figure 5.11: Data flow between modules through TDLComm
5.5.1 Packet exchange

TDLComm layer abstracts the physical exchange of information between the nodes of a distributed system. Its packet exchange service represents an abstraction over the low level networking operations. The upper services for clock synchronization, fault-tolerance, and logical data exchange via a specific communication protocol build upon the abstractions introduced with this service.

We analyze the distributed systems from the physical communication point of view and identify two kinds of hardware nodes, depending on the role of the main processor (which executes the real-time TDL application) in the communication process: CPU assisted, and with dedicated communication processor (e.g. CAN or Ethernet controller).

The first kind of hardware platform uses the main CPU to perform the communication. A piece of software, usually called communication/network driver performs the serialization of the bytes of information and the low-level input/output operations. This method is typically used in non real-time systems, where the main CPU is sufficiently fast to perform the communication without interfering too much with the user functionality. In other systems, the cost reasons or the limited resources (e.g. power, available I/O pins) prohibit the addition of a dedicated network processor. Regardless of the reasons for the lack of a dedicated network controller, the additional overhead of the communication on the main CPU may prevent the deterministic execution of tasks in hard real-time systems. In our case, the LET concept of TDL hides the underlying execution flow of the tasks in the system, which may introduce uncertainty in the communication patterns on the physical channel.

Thus, with our TDLComm layer we focus on the second kind of platforms, where a dedicated network processor offloads the communication job from the main CPU.

Messages, Datagrams, and Packets

A message represents a data exchange between the ports of a pair of TDL entities from two modules located on different nodes in a distributed system. It corresponds to a value exchange operation between sets of ports, discarding the output ports of the producer entity that are not used by a consumer entity. It has a fixed size equal with the sum of the sizes of the producer port types and two constraints derived from the availability of the corresponding port values and the latest allowable receive moment (i.e., the end of the LET of the producer task). If the message has an associated tag, then its size increases with the size of the tag. A message belongs to a particular task instance; thus, it is not periodic. The number of messages depends on the periods of the producer tasks, the number and period of mode switches, and possibly on the number and periods of the consumer entities. Several optimizations available are detailed in (Farcas, 2006). However, they do not change the basic concepts and the mode of operation of the TDLComm layer.

A datagram represents a collection of messages exchanged at the same time instant.
5.5. TDLCOMM LAYER FOR TRANSPARENT DISTRIBUTION

It contains one or more messages; thus, it refers indirectly to one or more task entities that provide output values from the same or different modules of a node. A datagram has a fixed size equal with the sum of the sizes of each of its message constituents. During the generation of the communication schedule, the scheduling algorithm may grow or shrink the size of a datagram by adding or removing messages; however, once a feasible communication schedule is found and generated, the allocation of messages to datagrams remains fixed.

We refer to a packet as the unit of information to send on the communication channel. A packet has a minimum and maximum size derived from the physical properties of the communication channel and the low-level data-exchange protocol. Any packet may contain one datagram only, but more packets may refer to the same datagram. Thus, we can consider a packet as a physical instance of a datagram on the communication channel. Note that in contrast with a datagram, a packet contains actual port-value information, whereas a datagram acts just as a logical container. The order and timing of the packets is fixed within a communication round and expressed statically in the communication schedule. In addition to the actual message data, a packet may contain control information such as timestamps, mode vectors, or other runtime information relevant to the communication protocol.

![Diagram of packet structure](image)

**Figure 5.12: Port values, messages, datagram, packet and frame**

The packets have two important designators: a time and a direction. The direction specifies the type of operation the TDLComm layer has to perform with the packet. It also specifies the relevance of the time designator in relation with the logical time and the direction of the packet. Thus, a time designator for a sending operation reflects the logical time when the TDLComm layer of a node containing a producer module has to send the packet. On the other hand, for the receiving operation it reflects the logical time when the packet was already received by the network processor of the node containing the client module, and stored in the processor’s local buffers or the main memory. At this moment the TDLComm layer can safely read the corresponding data structures, to fetch the information contained in the received packet. Following, the packet has an index and one or more references to datagrams. Depending on the protocol, one of the datagrams referred by a packet is actually assigned to that packet.

The Figure 5.12 presents the encapsulation of messages into a datagram, which is sent within a packet along with some control information. The packet accompanied by
CHAPTER 5. THE PLATFORM ABSTRACTION LAYER

a header and a trailer (typically containing CRC information) forms a frame. The frame represents the physical unit of information exchanged on the communication channel.

The TDLComm layer performs two operations for the exchange of packets: sending and receiving packets. The two operations are highly hardware and RTOS dependent; thus, we have to provide their implementation for each platform we use in the distributed setup. The API calls of TDLComm to implement are: `TDLComm_Send_Packet` and `TDLComm_Receive_Packet`. In the following paragraphs, we present the design considerations for the implementations of the two functions.

Sending packets

Depending on the number of steps required for sending the data over the communication channel, we identified two methods for sending packets: direct transfer and prefetch + transfer.

In the first case, where the network processor has direct memory access to the same memory space as the main processor, we can perform the encapsulation of the packet data into a single step. Conceptually, it is sufficient that the TDLComm layer passes to the network processor the address of the information block to send, its length, and a send command at the time when it has to send a packet. The network processor performs a direct memory access cycle (using some DMA line), copies the packet data into its internal memory, and begins the transmission. Typically, this operation has a negligible effect on the performance of the main CPU; although, on some systems, the DMA cycle for the memory access may introduce additional memory latency for the main CPU.

The second case is common with field-buses, or priority-based buses (e.g. CAN), where the network processor has one or more internal memory buffers dedicated for the transmission of packets. Some systems even allow the setup of several packets in advance on different buffers, and the network processor sends them in sequence according to some predefined rules. Under these circumstances, the TDLComm layer performs two steps before the data actually goes through the communication channel. In the first step, as soon as the values of the TDL output ports to send in the next packet are available, it loads them encapsulated into a packet in the memory buffers of the network processor. Afterward, when the time to send the packet according to the protocol specifications comes, it instructs the network processor which buffer contains the information to send, its length and the send command. At this point, the network processor already has the data in its buffers and performs immediately the communication on the physical channel, without any interference on the main memory/CPU.

We support both methods for sending packets and furthermore present two approaches for encapsulating the port values into packets. The approaches differ in the mechanism of accessing the port values, the complexity of the glue-code, and the separation of concerns within the layers of the resulting real-time application.
The first approach relies on the stop drivers of the producer tasks. In this case, after the completion of the tasks, the stop-drivers copy the relevant internal port values to the TDLComm layer as presented in Figure 5.13. On the client-node side, the terminate driver of the stub of the service-provider module performs the port-value copy operation from the TDLComm layer into its visible ports at the end of the LET of the producer task. The release driver of the client task reads the value from the visible output ports of the stub as if the producer task was running on the same node. With this approach, there are multiple drawbacks from the additional meta-information required at the stop-drivers and stub task-termination drivers level about the TDLComm data-structures, operational mode of the module, and logical time. Thus, the module wrappers of the service provider and client modules containing the release, start and stop-drivers would no longer be identical with a the non-distributed setup.

A better approach gives the TDLComm layer direct access to the relevant internal ports at the moments defined by the TDL Scheduler and the communication schedule (see Figure 5.14). Thus, TDLComm reads directly the internal output ports of the producer task, encapsulates them into a packet and sends the packet to the other nodes. On the client-module side, the corresponding TDLComm layer invoked by the TDL Scheduler at the receiving time extracts the port values from the received packet and stores them directly into the internal output ports of the stub module. Hence, the stop-drivers are no longer needed, the stub task-termination drivers are simply identical with its service-provider module, and the module wrappers remain the same regardless of the system architecture. Note that under any circumstances the internal output
port values are available only to the TDLComm layer before the end of the LET of the corresponding task. All the user tasks in the system can access only the visible output port values, which retain their previous values until the termination event of the task (when the termination driver updates them from the values of internal output ports).

![Diagram](image)

**Figure 5.14: TDLComm initiated encapsulation of port values**

The overhead of copying the data from the main memory to the network processor memory may be negligible, but on slow systems it has to be evaluated and considered as networking overhead when performing the time-safety checking of the distributed system. We present the mechanism of identifying the ports for the encapsulation of their values into packets in the corresponding protocol sections.

In our PAL implementation, we pursue the second approach, where the TDL Scheduler may easily consider the possible delays and perform better scheduling decisions. This approach also improves the separation of concerns regarding port value availability and supports our claims of distribution that is transparent even at the level of drivers.

**Receiving**

Most hardware platforms support one of the following methods for transferring a received packet to the upper software layers. When the network processor has direct access to the main memory, it may alone store the information into a dedicated space without assistance from the main CPU. Thus, it is generally sufficient that the TDL Scheduler invokes the TDLComm layer at the moment defined in the communication schedule as the receiving time of the packet. At this moment, the packet is already in the main memory and the TDLComm layer extracts from it the port values of a service provider module and stores them in the internal output ports of the corresponding stub module.

Alternatively, when the network processor has internal buffers storing the received information, two steps are required. First, the TDLComm layer invoked at the receiving
moment transfers the content of the receiving buffers of the network processor into the main memory. Afterward, it decodes the content of the packet into the target stub-module port values. The overhead of the data transfer from the internal memory of the network processor to the main memory is typically low, but on highly loaded or slow systems it has to be accounted for when performing the time safety analysis of the modules on the distributed system.

Depending on the communication protocol used, a received packet may contain the output port values of several modules from the same node; thus, the TDLComm layer has to distribute this information to several stub modules. At compile time, the ANSI-C plugin already places only the sections of the communication schedule relevant to each node; hence, at the moment when the TDL Scheduler invokes the TDLComm layer, the received packet always is relevant to the receiving node. All other received packets using the broadcast mechanism of the communication channel may be safely ignored. Nevertheless, an event-based communication bus may perturb the execution flow of user tasks, by issuing interrupts to the main CPU as a notification of the receive event. This represents the information push mode of operation (Deline, 1999).

The algorithm for the generation of the communication schedule rules out by design the problem that may appear when more packets arrive before the TDLComm layer is invoked to read the content of the network processor’s memory. Thus, TDLComm layer can operate without interrupts from the network processor, because the communication schedule clearly specifies when a packet is expected to arrive at any given node. This is the information pull mode of operation (Deline, 1999). In the case when the interrupts from the network processor cannot be disabled or are required to transfer the content of its internal volatile memory to the main memory, the implementation of this functionality must have the smallest run-time overhead possible and its wcet and packet arrival rate must be considered in the scheduling analysis for user tasks.

Listing 5.22: Activation of the TDLComm layer from TDL Scheduler

```c
// logical time is t
// k is the index of the packet to send
int tcomm = t mod πsync; // compute current time within the network cycle

if (Packets[k].when == tcomm) // it's time to process this packet
    if (Packets[k].send == true) // we have to send this packet
        _TDLComm_Send_Packet(k) // encapsulate and send packet
    else
        _TDLComm_Receive_Packet(k) // decode packet k
end if

// advance to the next packet
k = k + 1

if (k ≥ ||Packets||) // we are at the end of the set of packets
    k = 1 // start a new network cycle
end if
else
```

155
CHAPTER 5. THE PLATFORM ABSTRACTION LAYER

// there is no packet to process at this time
\[\delta_S \leftarrow \text{Packets}[k].\text{when} - t_{comm} \quad \text{// schedule next operation}\]
end if
// end of TDLComm actions
// continue with E−Machine operations

The TDL Scheduler invokes the TDLComm layer following the algorithm from Listing 5.22. At the current logical time \( t \), we expect to process the packet \( k \). We first verify that the time to process this packet matches the current network time. In this case the logical time must be equal with an integer number of network cycle times plus an offset equal with the packet time. Note that the packet time is always bounded by the network cycle time. Secondly, we identify the type of operation to perform on this packet, and invoke the appropriate packet encapsulation/extraction methodology. Afterward, we advance the index \( k \) to the next packet.

In the case when the TDLComm layer has no packet to process at the current time, which may happen depending on the communication protocol used, we simply set the TDL scheduler waiting time \( \delta_S \) to the moment of the next packet send/receive operation. Recall that in the case of sending, the packet time designator points to the moment in time when to send the packet, whereas for receiving packets it indicates the moment when the packet was already received at the network processor level.

Formal definition. A message \( m \) is denoted by the tuple \((p_p, p_c, t^*, t^d)\), where \( p_p \) represents the port of the producer entity (source) and \( p_c \) the port of the consumer entity (destination). The time \( t^* \) represents the moment when the message is released by the producer entity and it becomes available for sending. The time \( t^d \) represents the moment when the content of the message is required by the consumer entity, which means that it is the latest time the message can be received on the remote node to be successfully read by the consumer.

We note with \( D \) the set of messages composing a datagram, \( D \subset \text{Messages} \). Thus, a packet \( pk \) has the form \((pk, V_m, t, D)\), where \( t \) is a logical time instant of the data exchange for the datagram \( D \). The vector \( V_m \) represents the state of the modules on the node sending the packet (i.e., the mode in which each module runs at that moment). A feasible communication schedule follows the rule: \( \forall pk \in \text{Packets}, \forall m \in D_{pk}, \forall t^* \leq t \leq t^d \).

We define the send operation as the tuple \((S, pk, t)\), where \( pk \) represents a data packet, \( t \) represents the logical time at the moment of sending. The sending function \( S: PK \rightarrow (\hat{PK}, N^*) \) operates on the set of packets \( PK \) and provides the duration of the transmission \( S(pk) = \{(\hat{p}k, t_S) | t_S = a_S + b_S \cdot \|pk\|; a_S, b_S \in N^*\} \) of \( \hat{p}k \) from the point of view of the main processor, that is the CPU time it takes to copy its content to the internal buffers of the network processor. On systems where the network processor has direct memory access, the values of \( a_S \) and \( b_S \) are generally small, so that the effect of the sending operation on the logical time is negligible: \( t \approx t + t_S \).
5.5. TDLCOMM LAYER FOR TRANSPARENT DISTRIBUTION

Similarly, we define the receive operation as the tuple $(R, pk, t)$, where $pk$ represents a received data packet, and $t$ represents the logical time when the packet is received. The receiving function $R: \vec{PK} \rightarrow (PK, \mathbb{N}^*)$ operates on the set of physical stream of bits composing packets $\vec{PK}$, and provides the receiving duration $R(pk) = \{(pk, t_R) \mid t_R = a_R + b_R \cdot \|pk\|; \ a_R, b_R \in \mathbb{N}^*\}$ from the point of view of the main processor, that is the CPU time it takes to copy the content of the packet from the internal buffers of the network controller to the main memory.

We abstract the communication in a simplified model ignoring fault-tolerance as the function $SR: \vec{PK} \rightarrow (\vec{PK}, \mathbb{N}^*)$. It operates on a sent packet and returns the same packet on the receiver side along with the communication time: $SR(pk) = \{(pk, t_SR) \mid t_SR = a_SR + b_SR \cdot \|pk\|; \ a_SR, b_SR \in \mathbb{N}^*\}$.

Thus, a communication cycle can be abstracted as $S \circ SR \circ R$, with a duration $t = (a_S + a_R + a_SR) + (b_S + b_R + b_SR) \cdot \|pk\|$.

5.5.2 Clock synchronization

The TDLComm layer relies on time-triggered communication for the information exchange between nodes. To implement a TDMA communication scheme, we require a synchronized time base across the distributed system. With a common notion of a "global time", each node may access the communication channel without interfering with the communications of other nodes. This collision-free mechanisms relies on dedicated communication slots for every communication.

Our approach for clock synchronization is inspired by previous work on time-triggered communication and in special TTCAN (Führer et al., 2000) and TTA (Kopetz and Bauer, 2002), with a master-slave mechanism derived from the TTP/A fireworks protocol (Kopetz, 1995). The algorithms are part of the Cristian class of algorithms for clock synchronization: a master node sends periodically a synchronization frame and the other nodes use its timing information to adjust their internal clocks.

In the initialization phase of the TDL runtime environment, upon the initialization of the system hardware, the TDLCComm layer initializes the network processor and starts listening for traffic. In case it detects an active master it takes the latest global logical time from the synchronization frame and initializes the logical time of the node. Otherwise, after a user-defined time elapsed without detecting any master, the TDLCComm layer establishes the node as a master node, initializes the global logical time, and starts broadcasting regularly its values.

On all nodes we assume that the logical time evolves in a discreet fashion, meaning that there is a direct mapping between an integer number of system clock ticks and an integer amount of microseconds (i.e., the base time unit of TDL). As a platform restriction, we assume that we can express the synchronization period $\pi_{sync}$ as an integer number $n_{sync}$ of system clock ticks or the so-called macro-ticks $t_{Mt}$ in TTA terminology.
\[ \pi_{\text{sync}} = n_{\text{sync}} \cdot t_{Mt} \]

Each macro-tick is composed of a fixed number of micro-ticks (i.e., internal timer clock ticks). Most systems set a specific ratio between a number of micro-ticks and a macro-tick at boot-time. We note with \( t_{Mt} \) the duration of a macro-tick and with \( t_{\mu t} \) the duration of a micro-tick (CPU internal clock). The ratio \( k \) defines the clock scaling factor of the RTOS.

\[ t_{Mt} = k \cdot t_{\mu t} \]

The master encapsulates its global logical time \( t \) into the synchronization frame and sends it to the clients. Each client receives the message after an individual \( t_{\text{trans}} \) time. The sampling operation of RTRM allows us to query the state of both micro and macro-ticks of the system, thus, obtain an accurate view over the system’s current time. For example, assuming the system clock updates every 500ms, the logical time within the upper layers updates also every 500ms. However, when the TDLCComm layer receives the synchronization frame it may determine through RTRM the exact time of the reception, with a resolution higher than the system clock (e.g. 492.18ms since the last reception).

For an initial value of the transmission time \( t_{\text{trans}} \) the TDLComm layer uses either an automatic pre-computed value or a user-defined value. The user may define a transmission time when the communication medium has special properties, such as the bit-delay time of CAN, which can be used to determine with a high precision the exact transmission time. The transmission time may also be set to zero on high bit-rate communication channels that may exchange the frames faster than a micro-tick, or on systems with dedicated clock lines. Otherwise, from the bit-rate of the communication channel, the length of a physical frame, the inter-frame gap the ANSI-C plugin may compute an estimation of the transmission delay. Benefiting from the broadcast mechanism of the communication channel, the TDLCComm layer later improves this estimation by comparing it with the estimated transmission times of other nodes. In this way, the estimation of the transmission time comes closer to the real value, which is influenced by a variety of physical factors unknown at compile time, such as distance between nodes, existence of active networking elements such as switches or gateways, queuing delays at the network processor level, etc.

We compute the deviation \( d_i \) of each client node \( i \) between its internal logical time \( t_i \) and the global logical time \( t \), considering the transmission time \( t_{\text{trans}} \).

\[ d_i = t_i - (t + t_{\text{trans}}) \]

To synchronize the clocks of the clients with the master we have to either slow-down or speed-up the delivery of clock pulses to the macro-ticks counter of the system. Hence, we increase the ratio between the nominal number of micro ticks for every macro tick to artificially-slowdown the system clock, and vice-versa we decrease the ratio to speed
it up. To compensate for a clock deviation at the client level of $d_i$, during the following synchronization period (i.e., the time until the next synchronization frame is expected at the client side), we change the number of micro-ticks that make a macro-tick. This procedure is highly platform dependent and typically requires in-depth knowledge about the underlying hardware platform, available clock sources, and their capabilities and accuracy.

Consider that the initial ratio between the macro-ticks and the micro-ticks in the client system is: $t_{Mt_i}^0 = k^0_i \cdot t_{\mu t_i}$. We assume that at the network round $j$, we have a clock deviation on the client $i$ of $d_i^j$ microseconds. In the next period $\pi_{sync}^{j+1}$, we have to compensate a nominal number of micro-ticks $c_{\mu t_i}$, which means that we distribute the deviation $d_i^j$ over the entire following synchronization period of $n_{sync_i}$ macro-ticks of the client. Fractional parts of $c_{\mu t_i}$ are passed to the next synchronization period. Note that the deviation may be positive or negative.

$$c_{\mu t_i} = \frac{d_i^j}{n_{sync_i}}$$

A positive deviation indicates that the client clock runs faster than the master clock, and we have to increase the number of micro-ticks per macro-tick. Similarly, a negative deviation indicates that we have to speed-up the client’s clock and use less micro-ticks for a macro tick as seen in Figure 5.15. Hence, in the next period $\pi_{sync}^{j+1}$, the duration of a macro-tick is:

$$t_{Mt_i}^{j+1} = (k^j_i + c_{\mu t_i}) \cdot t_{\mu t_i}$$

In the figure we can also see that the new synchronization period for the client starts always aligned at least at micro-tick level (typically at macro-tick level), although the synchronization frame may have been received in between micro or macro ticks.
CHAPTER 5. THE PLATFORM ABSTRACTION LAYER

This synchronization mechanism updates every synchronization period the ratio between the micro-ticks and macro-ticks. As result, the system clock may appear to go slightly faster or slower than a reference clock. This behavior is unavoidable as the source of the micro-ticks (i.e., the hardware oscillator) cannot be influenced through software on most platforms. On the other hand, this mechanism can easily compensate different oscillator periods in "identical" hardware nodes that operate at different temperatures, where typical assumptions of "identical" behavior lead to critical failures. It also avoids clock jumps that could severely affect the operation of the TDL Scheduler and the E-Machine.

Special applications may require external synchronization from additional hardware, which may not be as accurate as the internal clock of the computational system. In this case, when a computational node already completed a full synchronization period, its TDLComm layer maintains the local value of the logical time unchanged until the next synchronization cycle begins. Thus, the upper layers (E-Machine and TDL Scheduler) remain idle from the end of the local view over the synchronization period and until the external hardware signals the beginning of a new period. This behavior is unavoidable because of the impossibility to synchronize the accurate clock of the node with signals from a less accurate external clock (which may even depend on mechanical devices such as a crankshaft). On such platforms it is possible to enable a special mode of operation for the E-Machine, which allows the execution of the actuator-update drivers before the end of the LET of the corresponding tasks. This feature apparently violates the LET concept, but in reality it is just a special case of tasks with deadlines less then their period. Their LET is thus reduced with a user-specified amount to update the actuators at well-defined moments since the synchronization signal of the external hardware. From an external point of view there may be a small gap between the ending and the beginning of the periods depending on the synchronization period and the mode periods.

The clock-synchronization mechanism of the TDLComm layer abstracts the differences between the hardware capabilities of the nodes and the communication infrastructure. The TDL Scheduler always operates through RTRM with local values of the logical time, which are periodically synchronized to the global notion of logical time in the cluster. The E-Machine works under the control of the TDL Scheduler; thus, the reactivity of the node is clearly driven by the global logical time, and the distribution appears transparent at the application level.

5.5.3 Communication protocol using hyper-periods

In this section, we present the protocol in relation with the TDLComm layer along with the data structures and algorithms for encapsulating and extracting the port-value information to and from the exchanged data packets. For a complete analysis of the protocol and the methodology for generating the corresponding communication schedule we refer to [Farcas, 2006].
As a general condition for the applicability of the protocol, the relation from Equation (5.2) must hold. The $mspGCD_P$ represents the GCD of the mode-switch periods and mode periods of the producer modules (the modules which are imported). The task set for the LCM, is the set of tasks of the producer modules that have remote clients. The communication cycle, also known as bus-period, is in this case equal with the LCM of all producer tasks.

$$mspGCD_P = GCD(\pi_m, \pi_\eta), \forall m \in \text{Modes}[M], \forall \pi_\eta \in \text{ModeSWs}[M]$$

$$\text{Tasks}_P = \{\tau \mid \tau \in \text{Tasks}[M], \exists \Lambda(\tau,e), e \in N_e, M \in \text{Imports}[N]\}$$

$$mspGCD_P = k \cdot LCM(\tau), \forall \tau \in \text{Tasks}_P \quad (5.2)$$

This class of applications covers many existing systems, where modules have the same period for all their modes (the behavior is not changing radically between different modes), or do not perform mode switches (most existing time-triggered distributed systems). In all other cases, the protocol presented in the next section may be a viable solution for modules with arbitrary mode periods and mode switches.

**Scheduling strategy variants**

The protocol operates according to the producer-consumer model. The task that produces the output values, which have to be sent to the remote consumer entity, dictates the scheduling restrictions of the messages it generates.

The communication-schedule generator tool tries to provide a schedule where the messages are sent as close as possible to the end of the LET of the producer task, so that the task has more time to finish its computation, and the TDL Scheduler has more flexibility in scheduling the tasks to complete before the messages must be sent.

The scheduling algorithm accepts constraints from the physical properties of the communication channels connecting the nodes of the distributed system. Thus, it may include gaps or control packets (e.g., for synchronization) in the schedule and may skip unused messages to save bandwidth. For example, in Figure 5.16 the output of the first producer-task instance is not used by the consumer task and may be safely ignored from the communication point of view. The algorithm also aligns the sending time according to the clock resolution on the computing nodes.

The resulted communication schedule has typically a number of entries equal with the LCM of the periods of all producer entities from all modules. Optimizations on the communication-scheduling algorithm may change this number. From the schedule, the ANSI-C plugin extracts the parts relevant to each node and adds them to the middleware targeted to that node.

There are two versions of this protocol depending on the number of scheduling tables generated, the number of messages, and the control information passed within a communication cycle. We detail them in the following paragraphs, along with their advantages and drawbacks.
CHAPTER 5. THE PLATFORM ABSTRACTION LAYER

Single communication-scheduling table. This version of the protocol considers all instances of a producer task from all the modes it is present in a module. Thus, it considers all possible messages it may generate regardless of the mode in which it actually executes at run-time. Recall that the working condition for this protocol guarantees a synchronized startup of the task in relationship with the network cycle.

As a result, the number of messages is always at least as big as the number of task instances that produce useful information; therefore, regardless of the actual running mode, a task has always a reserved communication slot with its clients. On the other hand, this approach overuses the capacity of the communication channel, wasting bandwidth with unnecessary information. Nevertheless, from the point of view of TDLCComm, the number of messages, their grouping into datagrams and packets is irrelevant. The same procedure of encapsulation and extraction, sending and receiving applies.

The Figure 5.17 presents this approach, where task1 may be running in one of the modes modeA or modeB with the relative frequencies of two respectively three. The task generates three messages in the first mode, and two in the second mode. In the resulted communication schedule, the last messages have similar constraints; thus, we obtain four messages.
At run-time, depending on the current mode of execution of the module, the task task1 may produce two or three messages, although we have allocated four messages for it on the communication channel. In this case, the last message in the LET of the task gets the latest task-output value, whereas the other messages from the same LET get the old value from the previous LET. When the module is in modeA, the message m1 has the same content as the message m4 from a previous communication cycle and the message m3 has the same content as the message m2. If the module would contain another modeC, where task1 would not be invoked, then all four messages would have the same content as the last message m4 from a previous modeA or modeB.

When a producer module changes its operating mode, the TDLComm layer of the respective node sends the new mode of the module. Hence, the TDLComm layer of a receiving node notices the new producer mode through comparison with the current mode of the stub module. Consequently, it updates the mode of the stub module. With this approach, the stub module and the producer module are always synchronized and their termination drivers run at the same time. Hence, the behavior of the overall system is the same as if the producer module would run on the same node with the consumer module.

There are two possible optimizations here: send the mode information along with each message (control bytes), thus increasing the size of a message, or send the mode information in the control part of a packet when mode changes occur. The first approach has a fixed allocation and may be preferred for deterministic systems as it denotes with each message the current state of the module. The second approach may be feasible for systems where the channel bandwidth is insufficient for sending the tag along with each message. The drawback in this case is that a fault on receiving the mode-change packet may lead to an inconsistent state between the producer module and one or more of its stubs. This behavior may be avoided using the fault-tolerance layer with an additional CPU overhead. The two approaches trade communication channel bandwidth with CPU time.

A communication-scheduling table per parallel mode combination. This version of the protocol exploits the exclusivity property of the modes of a module. The algorithm for computing the communication schedule generates a scheduling-table for each configuration of parallel modes from all modules. This approach may be feasible for applications with a small number of modes that exchange large amounts of information, as the communication is optimized for each particular mode of operation. On applications with large number of modes or with a high number of modules per node, the resulting number of scheduling-tables may be impractical. Optimizations on the number of frames according to the consumers of the messages may increase the number of tables.

When using this protocol, the scheduling tables of each parallel mode combination begin with a control frame per node containing the actual mode of operation of each module on that node. Afterward, there are frames for only the messages from the cor-
responding modes of all modules. At run-time, each node’s TDLComm layer switches the communication-scheduling tables upon learning the state of all modules on the network. The working condition for the protocol guarantees that the mode changes may happen only at the end of a communication period. Thus, it is safe to operate with the table of a previous mode until all control frames are received, and then switch to the table that defines the communication in the current state of the application (depending on the active mode of each module). The encoding and extraction mechanisms of this protocol remain the same, the only difference being in the handling of control frames. Practical implementations of TDLComm may detect the protocol version by analyzing the number of scheduling tables provided on each node.

**Encapsulation and extraction of the messages to and from packets**

To implement the requirements of transparent distribution, on the service-provider-module side, the TDLComm layer fetches at predetermined moments in time the values of the producer tasks and then sends them over the communication channel. On the consumer-module side, the TDLComm layer of the consumer node fetches at the predefined receive-moments the data from the network processor. It then extracts the relevant port values and stores them in the internal output ports of the stub of the service-provider module.

The encapsulation and the extraction of port values to and from packets rely on the intermediary messages and datagrams concepts and their appropriate run-time data-structures. In the Figure 5.18, we present a basic diagram illustrating the relations between packets, datagrams, messages, and ports, and their corresponding properties (encapsulated into appropriate data structures such as tables).

![Figure 5.18: Ports to/from packets direct mappings using hyper-periods](image)

The packets designators (i.e., its direction and time) are the main entry points for the algorithms of encapsulation/extraction. In the list of packets from Figure 5.18, the first packet denotes a synchronization packet sent at time 0 (relative to the communication cycle). The second is a packet expected to be received at time RxTime. It points to an entry in the datagrams table. The datagrams specify the number of messages and links to them. The messages contain references to the relevant ports of the service provider task or its stub (on the client node), and the size of the ports. Thus, we can...
follow the links from the packets, through datagrams, and then messages, up to the
target ports. The two algorithms from Listing 5.23 and 5.24 present the encapsulation,
respectively extraction of port-value information to and from packets.

The TDLComm layer maintains an index \( k \) of the current packet \( pk \) to send or
receive. When the time designator of the current packet is equal with the current
logical time in relation with the current network cycle, the TDLComm layer executes
the encapsulation or the extraction algorithms depending on the direction of the packet.
It first retrieves the datagram information of the current packet. When sending, the
datagram provides the links to the messages to send, whereas on receiving the datagram
provides the path to the internal output ports of a stub module. The algorithms iterate
through all messages of a datagram and either add the bytes of the output port values
to a new packet, or update them from the received packet. The encoding and decoding
mechanisms are symmetrical; thus, they abstract from the platform specific in-memory
byte ordering of the port values.

For a successful synchronization of a producer module with its stub, we have to
transmit also the active mode of the producer module. The ANSI-C plugin defines for
each node and each module two \textit{masks} for storing, respectively decoding the mode of

<table>
<thead>
<tr>
<th>Listing 5.23: Packet encapsulation</th>
<th>Listing 5.24: Port-value extraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>// k is the index of the packet to send</td>
<td></td>
</tr>
<tr>
<td>// first construct a new packet</td>
<td></td>
</tr>
<tr>
<td>( pk \leftarrow \emptyset )</td>
<td></td>
</tr>
<tr>
<td>// get a reference to datagram</td>
<td></td>
</tr>
<tr>
<td>( datagram \leftarrow \text{Packets}[k] \rightarrow \text{datagram} )</td>
<td></td>
</tr>
<tr>
<td>// initialize packed data offset</td>
<td></td>
</tr>
<tr>
<td>( o \leftarrow 1 )</td>
<td></td>
</tr>
<tr>
<td>// process all messages</td>
<td></td>
</tr>
<tr>
<td>\textbf{for} ( i \leftarrow 1 ) \textbf{to} ( \text{datagram.msgs} )</td>
<td></td>
</tr>
<tr>
<td>// fetch the ( i )-th message</td>
<td></td>
</tr>
<tr>
<td>( msg \leftarrow \text{datagram} \rightarrow \text{messages}[i] )</td>
<td></td>
</tr>
<tr>
<td>// get the source port of the message</td>
<td></td>
</tr>
<tr>
<td>( p \leftarrow \text{msg} \rightarrow \text{port} )</td>
<td></td>
</tr>
<tr>
<td>// serialize port value</td>
<td></td>
</tr>
<tr>
<td>\textbf{for} ( j \leftarrow 0 ) \textbf{to} ( \text{msg.size} )</td>
<td></td>
</tr>
<tr>
<td>// retrieve byte ( j ) of ( V(p) )</td>
<td></td>
</tr>
<tr>
<td>( b \leftarrow \text{b}_j[V(p)] )</td>
<td></td>
</tr>
<tr>
<td>// add a byte of port value to ( pk )</td>
<td></td>
</tr>
<tr>
<td>( pk \leftarrow pk \cup { b } ) / ( \text{b}_o[pk] \leftarrow b )</td>
<td></td>
</tr>
<tr>
<td>// increment packed data offset</td>
<td></td>
</tr>
<tr>
<td>( o \leftarrow o + 1 )</td>
<td></td>
</tr>
<tr>
<td>\textbf{end for}</td>
<td></td>
</tr>
<tr>
<td>\textbf{end for}</td>
<td></td>
</tr>
<tr>
<td>// ( pk ) contains latest port values</td>
<td></td>
</tr>
<tr>
<td>// ( k ) is the index of the expected packet</td>
<td></td>
</tr>
<tr>
<td>// ( pk ) is the actual received packet</td>
<td></td>
</tr>
<tr>
<td>// get a reference to datagram</td>
<td></td>
</tr>
<tr>
<td>( datagram \leftarrow \text{Packets}[k] \rightarrow \text{datagram} )</td>
<td></td>
</tr>
<tr>
<td>// initialize packed data offset</td>
<td></td>
</tr>
<tr>
<td>( o \leftarrow 1 )</td>
<td></td>
</tr>
<tr>
<td>// process all messages</td>
<td></td>
</tr>
<tr>
<td>\textbf{for} ( i \leftarrow 1 ) \textbf{to} ( \text{datagram.msgs} )</td>
<td></td>
</tr>
<tr>
<td>// fetch the ( i )-th message</td>
<td></td>
</tr>
<tr>
<td>( msg \leftarrow \text{datagram} \rightarrow \text{messages}[i] )</td>
<td></td>
</tr>
<tr>
<td>// get the destination port of the message</td>
<td></td>
</tr>
<tr>
<td>( p \leftarrow \text{msg} \rightarrow \text{port} )</td>
<td></td>
</tr>
<tr>
<td>// update internal port value</td>
<td></td>
</tr>
<tr>
<td>\textbf{for} ( j \leftarrow 1 ) \textbf{to} ( \text{msg.size} )</td>
<td></td>
</tr>
<tr>
<td>// extract byte ( o ) of ( pk )</td>
<td></td>
</tr>
<tr>
<td>( b \leftarrow \text{b}_o[pk] )</td>
<td></td>
</tr>
<tr>
<td>// store it into port byte ( j )</td>
<td></td>
</tr>
<tr>
<td>( \text{b}_j[p] \leftarrow b )</td>
<td></td>
</tr>
<tr>
<td>// increment packed data offset</td>
<td></td>
</tr>
<tr>
<td>( o \leftarrow o + 1 )</td>
<td></td>
</tr>
<tr>
<td>\textbf{end for}</td>
<td></td>
</tr>
<tr>
<td>\textbf{end for}</td>
<td></td>
</tr>
<tr>
<td>// internal ports contain updated values</td>
<td></td>
</tr>
</tbody>
</table>
the module into/from control bytes. This scheme is highly flexible and accommodates different number of modules per node and corresponding number of modes per module, without enforcing a system-wide encoding mechanism. The drawback is that the decoding masks must be available on all nodes containing stubs of a producer module.

5.5.4 Communication protocol with multiplexing over mode periods

In this section, we detail the integration within the TDLComm layer of an innovative protocol (Farcas et al., 2006) that dynamically multiplexes the messages over a static schedule. For the complete analysis of the protocol and the methodology for generating the corresponding communication schedule, we refer to the same work of (Farcas, 2006).

This protocol takes a different approach for dealing with the problem of mode switches within the modules of an application. It considers the communication period as the smallest interval where mode switches cannot occur, that is the GCD of the mode-switch periods of all producer modules $\text{mspGCD}_P$. The resulting communication period equally divides the period of the mode of a producer module into a fixed number of phases. The phases of a mode are mutually exclusive, and any producer module may change its mode only at phase boundaries.

Scheduling strategy

The communication-scheduling algorithm maps the messages of a phase into one or more communication windows such that the communication windows can be reused in all phases of a module. It then constructs frames within the communication windows to encapsulate the messages of each phase of a mode. As the phases are exclusive, the messages from different phases are also exclusive; hence, they may reuse the frames in the schedule. Nevertheless, the frames must have the adequate size to accommodate the largest message allocated to them, and their timing constraints must also match the allocated message with the tightest constraints.

Afterward, the algorithm determines the frames and binds each message to exactly one frame. During the execution of a module, the phase of its current mode determines the message to be sent from the subset of messages bound to a frame.

In the example from Figure 5.19, the tasks of a module produce four messages within the mode period. The first two messages are larger with a size of four, respectively three bytes, whereas the last two messages have only one byte. The mode period is equally divided into three phases, where the messages have release and deadline constraints that translate into corresponding communication windows. Mapping them into one communication window results in a frame with a data capacity of four bytes to be sent between 20 and 40ms. The timing constraints of the frame come from the minimum message deadline and the maximum message release. The allocation of the messages to the frame is fixed within each network cycle.
Encapsulation and extraction of the messages to and from packets

The communication scheduling algorithm identifies the messages from producer tasks per each phase of a mode, and then it associates a message with the phase at the end of the producer’s task LET. As the phase and the mode in which a message is produced change at run-time, we associate a tag to each message. The tag encapsulates the mode and the phase of each message. Note that this tag is a reduced version of the tag presented in (Farcas, 2006), because it does not contain task instance information which can be inferred from the target port of the message. The size of the tag depends on the number of modes in each module and the number of phases per mode. Hence, the message size is increased with the corresponding tag size (typically one more byte).

\[ \text{tag} = (M, m, \phi_m), \quad m \in \text{Modes}[M], \quad \phi_m \in \left[ 0, \frac{\pi_m}{\Pi_C} \right), \quad \Pi_C = \text{GCD}(msgGCD) \]

Similarly with the previous protocol, the TDLComm layer performs the encapsulation and the extraction of port values to and from packets under the control of the TDL Scheduler. It relies on the intermediary messages and pseudo-datagrams concepts and their appropriate run-time data-structures. The pseudo-datagrams contain the actual tag that identifies the message within the phase as depicted in Figure 5.20.

For the encapsulation of the port values into packets, we present an extension of the algorithm of the previous protocol, which uses the message tag information to identify and operate on the correct port memory locations. The algorithm from Listing 5.25 computes first the phase of the current mode of a module and creates a corresponding tag from the module, mode, and phase. It then starts building a new packet by matching the set of possible messages with the previously identified tag. As there may be more than one message with the same tag, it packs the tag and the content

Figure 5.19: Dynamic mapping of messages to frames
CHAPTER 5. THE PLATFORM ABSTRACTION LAYER

Figure 5.20: Ports to/from packets dynamic mappings

of the corresponding ports into the packet. For the case where multiple messages from different modules and phases are merged into a larger frame, we have to repeat the algorithm for each module.

Listing 5.25: Dynamic packet encapsulation

```c
// k is the index of the packet to send
// M is a module
// m ∈ Modes[M] is the current mode of M
// t_m is the time when the current mode period started
// t is the current time
// Π_C is the communication period GCD(mspGCD_P)

pk ← ∅ // first construct a new packet
φ_m = (t − t_m) mod Π_C // compute phase
tag ← (M, m, φ_m) // build tag tuple
o ← size(tag) + 1 // initialize packed data offset

∀msg ∈ Packets[k] → tag // process all messages
pk_tag ← tag // add message tag
p ← msg → port // get the source port of the message

for j ← 0 to msg.size // serialize port value
    b ← b_j[V(p)] // retrieve byte j of V(p)
    // add a byte of port value to pk
    pk ← pk ∪ {b} // b_o[pk] ← b
    o ← o + 1 // increment packed data offset
end for
// pk contains latest port values
```

The extraction of port values from the packets relies on the tag information from the packet. The algorithm from Listing 5.26 first identifies the tag of the received packet, and decodes the state of the producer module, its mode, and current phase.
We assume that a corresponding stub module is present on the node that receives this packet (other nodes ignore the content of this packet). At the moment when the node receives the packet, the producer may have changed the mode at the beginning of the communication cycle. Thus, the mode $m'$ of the stub module may be different from the producer mode $m$. In this case, we change the stub mode and update its runtime information (i.e., mode start time $t'_{m}$ and phase $\phi'_{m}$). Otherwise, we check whether the stub and the producer are in the same phase. When there are multiple messages per phase, the stub and the producer may already be in the same phase. Alternatively, we update the stub mode phase $\phi'_{m}$ to keep the producer and its stub synchronized. At this point, we can proceed to the extraction of the content of the packet and to store it into the corresponding port locations based on the tag information (see Figure 5.20).

Listing 5.26: Dynamic extraction of port values from packet

```c
// Prerequisites:
// k is the index of the expected packet
// pk is the actual received packet
// t is the current time
// $\Pi_{C}$ is the communication period GCD($mspGCD_P$)

o ← size(tag) + 1 // initialize packed data offset (tag size is known)
tag ← pk // fetch tag from packet

(M, m, $\phi_{m}$) ← tag // decode tag
// $M_{stub}$ is a stub module for $M$
// $m' \in Modes[M_{stub}]$ is the current mode of $M_{stub}$
// $t'_{m}$ is the time when the current stub mode period started

$\phi'_{m} = (t - t'_{m}) \mod \Pi_{C}$ // compute phase of stub

// synchronize timing of producer and stub
if ($m' \neq m$) // producer has changed mode during this phase
    $t'_{m} ← \lfloor t/\Pi_{C} \rfloor \cdot \Pi_{C}$ // update mode start time
    $m' ← m$ // update mode of stub
    $\phi'_{m} ← \phi_{m}$ // update mode phase
else
    if ($\phi'_{m} \neq \phi_{m}$) // producer is in different phase
        $\phi'_{m} ← \phi_{m}$ // update mode phase
end if
end if

// extract messages
$\forall msg \in Packets[k] \rightarrow tag$
$p ← msg \rightarrow port$ // get the source port of the message

// update internal port value
for $j ← 1$ to $msg.size
```
b ← b[pk]  // extract byte o of pk
b[wp] ← b // store it into port byte j
o ← o + 1 // increment packed data offset
end for
// internal ports contain updated values and stub is in the right mode

This protocol brings flexibility in static communication scheduling, by allowing dy-
namic mapping of messages over the same frames. Nevertheless, it still provides deter-
ministic communication patterns and a solid foundation for transparent distribution.

In addition, this protocol may improve the clock synchronization mechanism of TDL-
Comm and its reliance on master clocks. Provided that each node has along with the
list of packets it has to process, also the timing of the packets of other nodes, we can use
an averaging Berkley class algorithm [Welch and Lynch, 1988]. Each node listens to all
packets, and processes the content of only the relevant packets for its producer or stub
modules. From all other packets it extracts the timing information regarding the trans-
mission time and the reception time of the packet from other nodes. Thus, it detects the
deviation of its clock from all other nodes and through averaging it computes the ad-
justments for its clock. This mechanism may be extended to consider only the packets
from node with accurate clocks (i.e., the masters of the current clock-synchronization
protocol). The implementation of this new clock-synchronization protocol remains as
future work.

5.5.5 Fault tolerance (TDL-FT)

In safety-critical applications, such as X-by-wire (where X maybe any aerospace or
automotive concepts such as fly, drive, brake, meaning that there are no mechanical
connections between the control instruments such as steering wheel, brake pedal
and the control actuators of the plane/car), requirements for high reliability demand
fault-tolerance and redundancy. For example, safety-critical aerospace functions (at
Level A of DO-178B) are generally required to have failure rates less than $10^{-9}$ per
hour (Rushby, 2003). Most embedded systems become extremely complicated with
synchronization, voting, and redundancy management. In this section, we present an
optional layer of PAL, which simplifies the addition of fault-tolerance to TDL applica-
tions.

The fault-tolerance layer of our PAL implementation relies on the following fault-
hypothesis:

- software-design faults are excluded through extensive checking and simulations of
  the models, the underlying runtime environment, and platform specific software
  (e.g. RTOS, I/O drivers).

- on any node, the failure of a module does not affect other modules. This re-
  quirement supports a composable design in which individual applications are
5.5. TDL COMMUNICATION LAYER FOR TRANSPARENT DISTRIBUTION

unaffected by the choice of the other applications with which they are integrated.

- in distributed systems, the communication channels are replicated, and a failure in communication translates to a missing or corrupt message on one communication channel. This requirement supports the design of complex distributed systems without the burden of adding application level fault-tolerance.

Time-triggered operation provides efficiency, determinism, and partitioning, but typically at the price of flexibility. Our second TDMA protocol removes this limitation, allows for independent, group-wide, or global mode-switches within the modules of a distributed application, and provides improved support for fault-tolerance.

Fault containment with TDL is at module level. The practical implementation of a run-time system for TDL must provide separation between the modules concurrently running on a node. Whereas the reactivity is already separated through the semantics of the language, the runtime system must enforce the necessary mechanisms to isolate the tasks from different modules to contain faults in their implementation.

At module level, our TDL Scheduler already implements support for handling timing violations and suspending the execution of a module. Having the highest priority, the TDL Scheduler preempts any user-task which may exceed its estimated wcet and it either throws an exception and stops the execution of the module or continues the task execution when there is still time left without interfering with the operation of other modules (e.g., in the idle time).

Furthermore, there is additional redundancy in the sequence of E-Codes generated for each mode of a module. The algorithm for compiling the semantics of a TDL module into E-Code provides a clear path for execution, which can be easily checked for consistency. For example, a RELEASE instruction is always occurring after a corresponding CALL instruction for the release driver of the corresponding task. The arguments of the RELEASE instruction also contain redundant information such as task, start driver id, and deadline. Following, a FUTURE instruction there is always a RETURN instruction. These run-time hints further improve the reliability of the E-Machine operation.

Depending on specific platform support, the PAL may provide support for hardware isolation (through the Memory Management Unit) between user tasks, so that they run in different address spaces. The macros for the task wrappers already provide the basic mechanisms for this partitioning of task code in memory. Thus, any software error within their implementation does not affect the operation of any other task or the TDL runtime environment.

For distributed systems, the number of possible faults is highly increased, with timing, spatial proximity, and slightly out of specification (SOS) faults. TDL does not address these faults directly, because they must be supported through the physical infrastructure (e.g., fiber optics for communication lines) and the topology of the distributed system (e.g., star instead of bus).
To tolerate at most $a$ arbitrary faults such as nodes or communication channel failures, the clock-synchronization algorithm currently implemented by TDLComm requires at least $3a + 1$ nodes connected with $2a + 1$ disjoint communication channels. Alternatively, there can be $a + 1$ disjoint broadcast channels between them and $a + 1$ communication cycles (Lynch, 1996). There are currently no mechanisms in place to achieve self-checking or fail-silence behaviors. The averaging clock-synchronization mechanism described with the second communication protocol may be used to improve this fault-tolerant behavior.

**Module replication** is the underlying mechanism for fault-tolerance in distributed systems with TDL. A service provider module may have a number of replica modules located on other nodes. It is also possible to have replica modules on the same node to validate the correctness of the calculations on the sensor values. This approach filters software errors from transient electro-magnetic interferences on good I/O input data and prevents their propagation into the distributed system (i.e., node-level containment).

The TDL-FT layer acts on two levels: local and distributed. On the local level, provided there is a service-provider module and a number of replicas performing the same calculations, the TDL-FT layer provides the arbitration and value consolidation mechanisms on the output values required by other modules on the same node or on a remote node. The termination drivers of the producer task invoke the TDL-FT layer to retrieve the fault-tolerant value of the internal ports of all replicas of the same module from the same node. The TDL-FT layer may also use other values from replicas situated on different nodes. On the distributed level, the TDL-FT layer works on the consolidation of port values from module replicas on different node onto the output port values of a stub module on the consumer node.

TDL provides fault-tolerance concerning the *exact agreement* on port values from module replicas located on different nodes of the distributed system. This agreement is better suited for safety-critical applications than the averaging agreement, which may provide incorrect information from the "slightly-different" readings of different hardware sensors.

There are two possible mechanisms for creating the additional data structures for replication. One approach may use large buffers within the TDLComm layer for the additional fault-tolerance information and store all values from replicated modules within them. This mechanism requires additional information at the drivers level about the packets from replicated nodes and the buffers in which to store their content. Furthermore, at TDLComm level it requires the logic to determine which replica values target which stub port at run-time. Expressing this logic in a consistent manner may be difficult for an arbitrary number of replica modules.

We choose a better approach to create replica ports on the nodes containing the stub modules. Hence, we reuse the same TDLComm mechanisms for the encapsulation/extraction of messages to/from packets and we store the duplicated port values.
into additional FT-ports. The TDLMp provides the means for duplication of ports on
the client node similarly with the ports of the stub of the producer module. Practically,
we can consider that we operate with the ports of another copy of the stub module
(there is no need to consider another copy of the E-Code of a stub module). In this case,
the ANSI-C plugin generates the FT-ports on the consumer node when it detects that
there are multiple producer module replicas on different nodes. The TDLComm layer
simply transports the port values as with any other module ports. When the values of
all replicas have reached the client node (after the last packet from a replica module was
received and decoded successfully), the TDLComm layer invokes the TDL-FT layer to
consolidate the FT-port values corresponding to one producer module into the internal
port values of its stub. The stub’s termination drivers perform then the final port-value
copy operation that makes the consolidated values available to the client module.

Currently, there are no mechanisms at the language level for specifying the fault
behavior at the application level. Experimental support for the number of working
replicas may be used in the future to enable the restarting of modules (e.g. switch to
the start mode) or other application specific behavior.
Chapter 6

From TDL Models to Executable Code

We previously presented the Timing Definition Language for the development of real-time components and the fundamental concepts for component portability, namely the transparent distribution, virtual machine, and Platform Abstraction Layer. Nevertheless, a full development solution requires an adequate tool-chain. The developer requires tools to design and model an application, simulation environments for its behavior, testing facilities, code generators, compilers and runtime environments for the designed application.

In this chapter, we present first an overview of the existing TDL tool-chain developed by the MoDECS Project \footnote{http://www.modecs.cc} team. We continue detailing the TDL compiler, its plug-in architecture, and current capabilities. In Section \ref{sec:codegen}, we describe the automatic code generation process, the support of platform specific files, and component distribution. Following, we present practical aspects of the implementation of a portable TDL runtime system based on the Platform Abstraction Layer and the E-Machine presented in the previous two chapters. Further technical details are available in \cite{Farcas2005a}.

The resulting tool-chain is a complete end-to-end solution for the independent development of real-time components, which can be later integrated in various systems with arbitrary topologies (i.e., single-node or distributed systems).

6.1 Overview of the TDL tool-chain

The Timing Definition Language comes with a complete tool-chain, which helps the developer in designing, compiling and executing a real-time application. Most of the tool-chain is implemented in Java, making it highly portable. However, the run-time system is available both in Java for simulation purposes and in C for embedded systems. Figure \ref{fig:toolchain} presents a simplified version of our TDL tool-chain.
The tool-chain consists of the following functional components: a TDL compiler, a visual editor fully integrated with the Matlab/Simulink environment, and a corresponding run-time environment. The TDL compiler has a plug-in architecture, which allows its extension with platform specific plug-ins or other tools, e.g., bus schedule generator. The figure also shows a number of files that are used within the development process. In the following, we present the tool-chain by analyzing the development steps of a real-time application.

**Design** We start with the design phase of a real-time application. We can model the application by typing the corresponding TDL source files for the modules composing the application or we can use a more visual approach through the Visual Editor tool (Stieglbauer and Pree, 2004) through the Visual Editor tool (Werner, 2005), which is integrated with the Simulink environment from Matlab. In the Simulink library we have a special TDL block, which opens the Visual Editor and allows us to create a new TDL module from scratch or open an existing file. For a new module, the visual tool assists us by providing three subeditors for TDL modes, mode communications, and mode transitions.

The mode subeditor allows us to describe each mode of the module, create the basic elements such as tasks, sensors, actuators, and define the data-flow within each mode. We also specify here the mode periods, the LET properties of each task in relationship with the defined mode periods, actuator update frequencies, and guard conditions.
6.1. OVERVIEW OF THE TDL TOOL-CHAIN

The mode-communications subeditor, enables the design of complex applications, where state information is required between different modes of operation of an application. It allows us to specify the assignments for task output ports that are used when the module switches modes.

Using the mode-transition subeditor, we define the transitions of the module between modes and corresponding guards. For guard conditions, we can specify simple Boolean operations on the output ports of the previously defined tasks or sensors.

As a result of using the visual editor, we obtain the TDL source files for our application with one file per module. The visual editor has a plug-in architecture similar with the TDL compiler. With additional plug-ins it may create specific configuration files for our embedded platform.

Following the design of the application, we require its functional implementation. The Matlab/Simulink environment can be used to generate C code out of a Simulink model (Naderlinger, 2005). Alternatively, we can implement the functionality of our tasks, sensor getters, actuator setters, and guards using any other adequate tool.

Compilation For each TDL module of the application, the compiler processes the TDL source code and generates an abstract syntax tree (AST) representation of the module as intermediate format as well as the E-Code, which describes the timing constraints of the module in a close to machine-level format. The plug-in architecture of the TDL compiler allows its extension with any number of tools that rely on the AST.

For our embedded platform, we extend the compiler with the ANSI-C plugin. It implements the code-generation part for the middleware that binds the timing specifications of the previously designed application with its C functionality. The middleware (glue-code) adheres to the specifications of TDLMp.

For distributed systems, a bus schedule generator tool plugs in the TDL compiler to create the communication schedule. It relies on a configuration file that contains a list of computing nodes of the distributed platform, the assignment of TDL modules to computing nodes, and the physical properties of the communication infrastructure. As a result it provides the communication schedule that specifies ”which node sends what information at which time and to whom”.

We compile and link the generated middleware with the C functionality code obtained in the design phase, and with the TDL runtime environment. The C compiler produces an executable file, which contains a bundle of our application and the runtime environment. This file may now be transferred and executed on the target platform.

Execution At run-time, the TDL application bundled with the TDL runtime environment coexists in the main memory of the target platform. The Platform Abstraction Layer mediates the interactions between the E-Machine and the user functionality, and between the TDL Scheduler and the underlying RTOS of the platform. The functionality code of the application runs under the control of the E-Machine and the TDL Scheduler, which operate according to the algorithms presented in Chapters 4 and 5.
For distributed systems, the TDLComm layer handles the mechanisms for data exchange between the computational nodes. The E-Machine supervises the logical behavior of the application and its interaction with the environment. The TDL Scheduler performs the preemption and dispatching operations of the user tasks, and the mapping of platform time to logical time. It also controls the communication operations of the TDLComm layer as described in the previous chapter.

6.2 TDL Compiler

The compiler developed for the current version of TDL supports all the features of the language and adheres to the specifications of (Templ, 2004). The compiler was developed in Java using the Coco/R compiler generator (Mössenböck, 2005). Hence, it represents a platform-independent part of the development path of real-time components.

It provides the core functionality of parsing and analyzing the TDL modules. For a given module, it checks whether the module adheres to the syntax and semantics of the language, and provides feedback to the developer regarding the errors encountered. In the case, when no errors are detected, the TDL compiler produces for each module a corresponding E-Code file that encapsulates the timing and behavioral specification of that module.

Plug-in architecture

The TDL compiler implements a "plug-in architecture", meaning that it deliberately leaves platform details and other enhancements to additional components. Regarding the concepts of plug-ins versus traditional components, although they perform similar tasks of supplying additional functionality, they also have several differences. Mainly the plug-ins target a specific application, whereas the components are function-specific and may target various applications from any applicable domain. Secondly, the host application already provides basic functionality without plug-ins, which is not the case with a simple component container. In the general case, depending on the underlying platform capabilities, the plug-ins may be loaded at run-time, providing dynamic extension of the functionality of the host application.

The plug-in architecture of the TDL compiler delegates platform-code generation, communication scheduling, and other similar aspects to specialized plug-ins. The plug-ins are loaded dynamically at run-time and have access to the compiled modules through either AST or the generated E-Code files. The plugins extend the Platform class and must implement the following methods: emitCode, open, and close.

The TDL compiler invokes the open method of a plugin, before it starts processing the TDL modules. The plugin uses this method to initialize internal data structures, detect the current compiler configuration, and possibly the existence of other plug-ins. It may also process at this step additional command-line parameters, which configure its behavior on the set of TDL modules currently prepared for compilation.
The method `emitCode` is invoked by the TDL compiler upon the successful compilation of a TDL module. The plugin implementing this method may arbitrarily extend the functionality of the compiler for the compiled module, but typically generates additional code for a specific platform. The generated code is platform dependent; thus, the functionality of the plugin clearly does not belong to the core TDL compiler. The plugin has access at this step to all properties of the TDL module through AST, the generated E-Code, and the data structures defined with the `open` method.

The `close` method finalizes the compilation process. The TDL compiler invokes this method from all loaded plugins as soon as it completed successfully the compilation of all modules. Through this method, a plugin may perform the final steps to complete the generation of platform specific code, as at this point all modules with all their properties are known to be compiled. The analysis of the dependencies between modules through their import relationships may be performed at this step.

In the previous chapters, we discussed the functionality of two core-plugins: the ANSI-C plugin responsible for the generation of the C middleware and the bus-schedule generator plugin responsible for the computation of a feasible communication schedule on distributed systems. We focus in the next sections on their practical aspects and their specific interaction with the TDL compiler.

### 6.3 Automatic code generation and configuration

The ANSI-C plugin targets the platforms supporting a C development suite (most real-time platforms have at least a C compiler and required runtime libraries). The TDL compiler invokes this plugin through its `-ansic` command-line argument.

The ANSI-C plugin implements an automatic C-code generator for the middleware that integrates the timing and behavioral specifications of a TDL module with its functionality expressed in the C language. The plugin generates code according to the TDL Mappings conventions. Thus, using the same conventions, we can bundle into a real-time application the TDL runtime environment, the middleware, and the functionality code.

Within its `open` method the plugin checks for the availability of a set of optional arguments. The arguments configure the behavior of the plugin and control the generation of platform specific files or code optimizations. The supported arguments are:

- `-bus bus_property_file` instructs the plugin to analyze the distributed system configuration from the file provided as argument;
- `-osek` instructs the plugin to generate additional OSEK platform specific files - the OIL configuration files;
- `-edf` instructs the plugin to provide the EDF dispatch tables for each mode of the modules. The TDL Scheduler uses at run-time this information with a hybrid EDF scheduler for improved performance.
CHAPTER 6. FROM TDL MODELS TO EXECUTABLE CODE

After it successfully compiled a TDL module, the TDL compiler invokes the `emitCode` method of the ANSI-C plugin. The plugin generates for the current module a C module-wrapper file and its corresponding header file. For the structure of the module-wrapper file we refer to the Section 5.2.1. The header file of a module wrapper contains the declaration of the public output ports provided by the corresponding module, the number of tasks, modes, ports, drivers, guards, and the length of the E-Code.

In addition, the plugin provides at this step a template header file containing the prototypes of all user tasks. Hence, the development of the user functionality is eased with the automatic naming and parameterization of the functionality code specified in the corresponding TDL modules. This feature avoids confusions between the TDL ports of a task and the parameters of the function that implements it.

At the end of the compilation process, the `close` method of the ANSI-C plugin performs the automatic generation of several additional files. Their purpose is to automate the development process, and ease the integration of the generated files with the user-provided functional code. Two kinds of files are generated by the current version of the ANSI-C plugin: makefiles and OIL files, described in the following paragraphs.

**Makefiles.** For an automatic build process, the ANSI-C plugin generates automatically a configuration file for the popular ”make” tool. Each node of a system (single or distributed) has a corresponding `makefile`, which contains a definition of the node’s name, and the list of module-related files (module wrapper + functionality code) to compile on that node. The developer may import these makefiles from a top-level makefile describing the compilation options for the target platform, and automate the building process for the code of all nodes.

**Platform specifics.** Currently for the OSEK platform, the ANSI-C plugin enables the automatic generation of the *OSEK Implementation Language* (OIL) files. For each module, it generates an OIL file containing the description of the user-tasks defined in that module. In addition, it generates an OIL file for each node of the system, which includes the OIL files of the modules on that node, describes the configuration of the node, and defines the TDL runtime environment specific tasks (e.g. TDL Scheduler). As a result, the full runtime configuration of a node is specified and the OSEK build process is highly automated.

**System initialization** is important in real-time systems, as in general the hardware of the sensors, the actuators, and in distributed systems also the network must be initialized. The PAL introduces a separation of concerns about system specific initialization and module specific initialization. Thus, for each module the ANSI-C plugin provides a dedicated initialization function, which may be used to initialize the hardware specific to that module. The system-wide initialization prepares the rest of the hardware or specific RTOS functionality common to all modules running on that.
platform. Similarly, through PAL the shutdown procedure may clean the allocated resources and turn off the used hardware.

**Distribution**

The TDL tool-chain allows for transparent distribution of modules. The bus-schedule generator plugin and the ANSI-C plugin enable the compile-time automatic setup of the distribution. The TDLComm part of PAL handles the run-time aspects of the distribution.

As detailed in the previous chapter, the bus-schedule generator tool produces a table-driven communication schedule. The ANSI-C plugin includes this schedule in the middleware of each node of the distributed system. It also prepares the TDLComm layer for compilation on each node. The dependencies between modules and the list of packets to exchange between nodes are already solved at this point and included in the `modules.c` file. Depending on the communication protocol used, the list of packets has different formats, but still a well-defined length. Thus, at run-time the TDLComm layer follows the bus schedule and performs transparently the data exchange between the modules involved in remote import relationships.

**Fault Tolerance** aspects are included in the middleware of each node containing replica modules. The ANSI-C plugin generates the duplicated ports and the additional data structures required for the port-value consolidation. The TDLComm layer handles at run-time through its TDL-FT part the actual replication and then the consolidation of output port values. This mechanism is automatic and transparent to the application. The developer only has to specify the number of replica modules and their placement, whereas the TDL tool-chain handles the fault-tolerance from design to the execution levels.

### 6.4 Portable TDL Runtime Environment

The TDL runtime environment is composed of four logical parts: an Initializer, a Virtual Machine, a Scheduler, and an Exception Handler. The main design goals of the runtime environment were portability, flexibility in configuration, low memory footprint, low CPU utilization, and efficient allocation of resources to user-tasks. For portability, we implemented the Platform Abstraction Layer completely, whereas for increased flexibility we added a number of platform specific configuration options that fine-tune the runtime environment for maximum performance. The code of the runtime environment is split into sections, which are compiled and enabled when specific functionality is required, e.g. exception handling, E-Code safety checks, placement of constant data into program memory on Harvard architectures, etc. Instruction ordering and loops were carefully analyzed to enable maximum optimization of the code.
CHAPTER 6. FROM TDL MODELS TO EXECUTABLE CODE

from the C compiler. For the scheduling of user-tasks, the hybrid EDF algorithm was implemented.

The base algorithms for the logical parts of the runtime environment have already been detailed in the previous chapters. Therefore, we analyze just particular aspects of their implementation useful for the understanding of the overall behavior of the runtime system.

Initializer

The Initializer of the TDL runtime environment has to perform the initialization of the hardware, setup all modules from all applications, and the start of the TDL Scheduler. In the following paragraphs, we detail each phase of the initialization process, as a reflection of the current implementation.

Platform initialization  The initialization of the basic hardware such as memory registers, IO ports, system clocks, etc is highly platform dependent. We must implement it every time we port the TDL-RTS to a new platform. We support currently a number of five platforms identified through the following C macros at compile time:

- **OSEK** - OSEK/VDX compliant RTOS, e.g. Wind River OSEKWorks on MPC555
- **INTIME** - InTime version 2.21-2.23, e.g. PC with WindowsXP+InTime
- **POSIX** - RT-POSIX, e.g. PC with RT-Linux,
- **AVR** - Atmel AVR 8 bit microcontrollers, e.g. AVR Butterfly development board
- **RENESAS** - Renesas M32C/85 microcontrollers, e.g. Flexray cluster

The `system_dependent.c` file contains a dedicated section for each platform where we define the `TDL_System_Init` function that performs this platform specific initialization. An optional initializer-function from each module may extend this initialization.

In the case of a distributed system, the ANSI-C plugin defines in the `modules.c` dependency solver file the number of packets that have to be exchanged by the current node with other nodes. This definition of `TDL_PACKETS_COUNT` reflects the requirement for communication and is used as a trigger for the Initializer to call a platform-dependent network-initializer function. This function performs the hardware setup of the network controller, the allocation of receive and transmit buffers, and starts the clock synchronization service.

Module setup.  After setting up the hardware, the Initializer performs the module setup. We divide this phase in three steps: loading, initializing, and preparing for startup.

In the current version, we only support startup-time module loading. During the execution of one or more applications, currently there is no possibility to load another
module or application dynamically, because the modules, their glue-code, E-Code and dependencies are compiled together with the TDL runtime libraries forming a single real-time application. Note that this real-time application may actually contain several independent applications, each composed of one or more dependent modules. We choose this approach because most existing hard real-time systems do not support dynamic loading of real-time modules and furthermore, the inter-process or inter-module communication and synchronization options are very limited. Moreover, deep embedded devices have usually the whole RTOS + application code written in a flash memory, without any support for a file-system or run-time code updates.

The ANSI-C plugin generates the corresponding loading function for each node (we regard a non-distributed system as a single node system) in the module.c file. As we perform one-pass compilation of the TDL modules, we do not know in advance how many modules we have on a node and all their properties. The additional header files ModuleName.h provide the necessary information about each module. The dependency header file modules.h provides their assigned IDs. The plugin uses the IDs of the modules to generate the module-loading function.

We add the possibility to execute an initializer function for the dedicated sensors or actuators of module, which may require some additional hardware/software initialization not covered by the TDL program. This option allows us to divide and cleanup the system initialization, and associate specific hardware with the module that uses it. In the case where such initialization is not required for a particular module, the module has to provide a dummy initialization routine in the form of a void C function with the name of the module suffixed by _INIT and taking no parameters.

Following the execution of the initializer functions of all modules, we have to prepare the environment for the execution of the functionality tasks of each module. The platforms with native support of TDL tasks do not require this step, whereas for the others with threading support we must create a thread for each TDL task of each module. We can perform this action also when the actual execution of a functionality code assigned to the TDL tasks takes place; however, we would loose useful CPU time by just creating and deleting threads.

Therefore, for the platforms with threading support, we must define in the platform specific section of the system_dependent.h file the macro _TDL_THREADING. The Initializer creates the threads for all tasks from all modules when we define this macro. The implementation of the function that actually creates these threads depends on the underlying RTOS and its API. We provide this function _TDL_Create_Exec_Threads for the supported platforms in the system_dependent.c file.

The threads created for each TDL task use the task wrappers defined accordingly to PAL. After their creation, the threads do not start the functionality code assigned to the corresponding TDL task until the TDL Scheduler enables them. On multi-threaded systems, after the completion of a functionality-code instance, the thread blocks on a semaphore until a new release action via the corresponding semaphore post.

We prepare the modules for execution by initializing first the TDL task input ports
as specified in the TDL program of each module. The Initializer calls the E-Machine to execute the first E-Code instruction block of each module, terminated by a RETURN instruction. The E-Machine calls the corresponding TDL initialization drivers of each module. The ANSI-C plugin helps with the starting phase, as it provides the table of modes with the starting mode first. Thus, there is no need to scan the loaded modules for their starting mode. In case when such mode exists, we simply pick the first mode of the module and retain the starting address of its E-Code sequence into the E-Code Future Instruction Pointer of each module. This mechanism is equivalent with a future event at time zero and takes place of a permanent instruction pointer per module.

When all modules and related data structures have been loaded and initialized correctly, the Initializer starts the Scheduler. Depending on the platform, this operation may involve the creation of a high-priority task/thread and setup of a clock system.

**E-Machine**

The E-Machine is a virtual machine executing E-Code instructions generated by the TDL compiler for each module. It is able to handle multiple modules at a time under the TDL assumption that E-Code is executed in logical “zero time”.

For each loaded module, we retain the future instruction pointer as the starting address of the next E-Code block. Temporarily, when we process an individual module we also retain the current address in the E-Code of that module. We also keep track of the current mode of the module and the time until the next sequence of E-Codes must be executed. For convenience, the future time is available both as relative to the current time and as an absolute value, with a configuration option to toggle the usage of one or the other. The TDL runtime environment starts at logical time zero. As we have to scan often these properties of the modules, we store them separately from the module construct to avoid an additional data indirection at run-time.

The E-Code instructions are composed of one opcode and up to three parameters. In the current version, we support eight E-Code instructions with optimized opcodes described in the emachine.h file (the opcodes are ordered by the frequency of the corresponding instructions in an E-Code block). The execution of the E-Code instructions is assumed to be ”zero” logical time. As this is not the case in the physical world, the TDL Scheduler measures the time of the E-Machine execution and adjust the time interval between successive E-Machine invocations.

**Scheduling Tasks through PAL**

The TDL Scheduler is the actual bridge between the TDL semantics and the underlying platform (RTOS+HW). Its purpose is to run the E-Machine for each module at the right time as defined by the TDL semantics and then to execute the released tasks according to a specific scheduling policy (e.g., Rate Monotonic or Earliest Deadline First). For a distributed setup, it furthermore has to coordinate the exchange of data between the nodes via the TDLComm layer.
Abstracting from various platforms, the TDL Scheduler is a high priority task/thread that controls the execution of the E-Machine for each module and the scheduling of the released tasks. The Scheduler sleeps during the execution of the TDL tasks and avoids unnecessary context switches by preempting the tasks only when it has to execute the E-Machine or reschedule them. On distributed systems, it also preempts the user tasks to invoke the TDLComm layer. Therefore, it allows a better CPU utilization as its sleep interval is smaller than the period of the fastest task from all applications, yet still bigger than the typical interval of a time-slicing approach.

From the TDL Scheduler point of view, the logical partitioning of the intended functionality into modules and applications is not relevant. All modules from all applications have to equally-share the CPU and memory of the platform. However, this view over modules does not mean that an application requiring 75% of the CPU when running standalone will struggle to complete its tasks with only 50% of the CPU when running in pair with another application. For example, when using RM scheduling, a task with a period of 30ms from an application is treated equally with another task from a different application that has the same period of 30ms. In the case of dynamic scheduling with EDF, the tasks are scheduled depending on their deadlines regardless their parent module.

On systems similar with OSEK that provide fixed priorities for the scheduling policy, such as RM, provided that there are enough priority levels available the TDL Scheduler can be simplified: it is enough to release the tasks as the RTOS performs their dispatching as needed. However, this mechanism requires additional configuration options in some platform specific file (in the case of OSEK: an .OIL file per module with the right priorities for each task). However, multiple modules with independent mode changes cannot be supported, because of the inherent rigidity of the fixed priority scheduling mechanism.

Furthermore, not all modules have E-Code that should be run at all time instants when the TDL Scheduler takes control. For example, in the case of two modules with the first module having a task with a period of 1ms and the other having a task with a period of 100ms, when using EDF, the scheduler will call the E-Machine for the first module each ms, but only every 100ms for the second module. This approach maintains the TDL semantics and allows increased flexibility in scheduling and in the allocation of available resources to the tasks that actually need them.

Depending on the priority (deadline) of a TDL task released by the E-Machine, and the priorities of other preempted or released tasks, the TDL Scheduler may decide to actually start (dispatch) or delay the task execution, until all other higher priority threads have completed. After completing the execution of a task, the Scheduler selects the next task to dispatch, based on the priority of the remaining preempted or released tasks.

The timing aspects are highly platform dependent; therefore, we use alarms or sleeping functions depending on the target platform. The clock resolution of the platform may be an impediment when trying to maintain jitter-free operations and the logical
"zero" time execution of the E-Code. Thus, we measure as accurately as possible the amount of time spent by the E-Machine and by the TDL Scheduler itself and to subtract it from the amount of time allocated for the execution of the tasks. Not all platforms allow a clean access to an accurate time source and typically the system time has a resolution in the range of hundreds of microseconds. In such cases, we use hardware dependent assembly functions to get time samples with up to nanoseconds resolution.

In the particular cases of MPC555 systems with OSEK, we can use as more accurate time source the internal hardware timer or the clock of the CAN controller. For PC based systems running InTime, RT-Linux, QNX, or other derivate RTOS, there are a number of clock sources available; however, not all can be used for our purposes. The most accurate time sampling on these systems comes from the Pentium TSC register (nanoseconds resolution); however, accessing it may be a problem when Ring-0 privileges prevent its usage. Moreover, this register only provides get-time functionality that cannot be used to schedule future actions. The additional functionality is available through the Advanced Programmable Interrupt Controller (APIC) implemented in the latest PCs. Such controller may be programmed to issue an interrupt after a specified amount of time has elapsed. We use this feature to further improve the clock resolution of the InTime RTOS and compensate for the RTOS scheduler jitter, by adjusting the number of hardware clock pulses per system clock tick.

Exception handler

The TDL Exception Handler provides by default only the error status messages and optionally terminates the execution of the current running applications. We preferred this default behavior for a development environment where any error can be easily spotted and corrected. However, for a production environment the default exception handler can be overwritten to correct at run-time some faults or perform additional tasks. An application or group of applications can provide their own exception handler by defining _TDL_USER_ERROR_HANDLER and implementing a similar function. The recognized error messages are defined in the emachine.h file (see Appendix A.1).

6.5 Measurements and discussion

In this section, we present an evaluation of our approach implementation. We show that TDL greatly simplifies the development process, its runtime system based on virtual machine achieves excellent performance with a low memory and CPU overhead, and the Platform Abstraction Layer greatly helps in porting the real-time components and the runtime system to a variety of platforms, single node or distributed. We take a simple application as a case study to avoid the complex hardware requirements of a real-world application.
6.5. MEASUREMENTS AND DISCUSSION

6.5.1 Portable real-time components from design to executable

The complexity of the runtime environment and the generated middleware is important for the future support of other platforms and the maintainability of the system. Hence, we analyze it first and then we continue with the efficiency of the generated code and the runtime environment on a set of platforms. We show that the overhead of PAL is minimal, and that the TDL runtime environment has a small footprint making it adequate even for simple 8-bit microcontrollers with less than 256 bytes of SRAM.

**Code complexity.** We divide the code of our approach into two classes of code: off-line executing code and on-line code. The off-line code consists of the code of the whole TDL tool-chain, including visual editor, TDL compiler and its plugins. For our work, the relevant parts are the ANSI-C plugin and the on-line code.

The ANSI-C plugin implementing the automatic code generation of module wrappers and related middleware has around 2740 lines of code (LOC). For comparison, all other platform plugins have around 1360 LOC. This numbers illustrate the complexity of the portable code generation and the number of features provided by the ANSI-C plugin, including platform specific support (makefiles, OSEK OIL files), distribution, and support for hybrid EDF scheduling.

The on-line code described in Section 6.4 consists of three parts: a platform-independent core, a platform-specific set of functions and configuration options, and the code of the application.

The platform-independent core contains the Initializer, E-Machine, and TDL Scheduler and has around 1020 LOC. Its associated definitions, types and data-structure definitions, configuration options contribute with another 480 LOC. The implementation of PAL for the five supported platforms accounts for 970 LOC, plus around 400 LOC of data structures, types, macros and configuration.

The PAL greatly helps in porting the runtime environment to arbitrary platforms. For example, the code for a minimalist AVR platform is around 60 LOC of which 25 LOC handle hardware specific initialization and the rest implement two functions for delay and dispatch. For an OSEK compatible platform there are around 150 LOC within three functions for initialization, time sampling, and planning through alarms. A more complex system using InTime has around 250 LOC for the set of functions related to initialization, shutdown, threading, release, dispatch, sampling, planning.

Because of the clear separation of the platform-independent core and the platform specific functionality through PAL, supporting a new platform is a breeze. For example, the RENESAS platform was supported within minutes starting from the base configuration of the AVR platform. Nevertheless, more-complex platforms may require in-depth knowledge about their internal functionality for a performance oriented timing.

To test the effectiveness of PAL on various single-node platforms, we use the application presented in Section 5.2 consisting of the modules M1 and M2, with the TDL runtime environment into one real-time executable for each platform. The implemen-
CHAPTER 6. FROM TDL MODELS TO EXECUTABLE CODE

...tation of the first module has around 200 LOC, whereas its generated wrapper code has around 235 LOC. The second module has an implementation of around 90 LOC, plus 115 LOC for its glue-code.

Run-time overhead. Traditional real-time systems avoid several abstraction layers to reduce the code size of the application, and to avoid the induced performance penalty. Our implementation of a portable TDL runtime environment relies on a platform-independent core presented in the previous chapter, and the platform specific functionality abstracted through PAL. We compiled with size optimizations the modules M1 and M2, with the TDL runtime environment into one real-time executable for the following single-node platforms:

- AVR - Atmel AVR ATmega169, 8-bit microcontroller, 1-8MHz, 16KB Flash, 1KB SRAM, 32 general-purpose registers. For this platform, we implemented a microkernel that supports currently non-preemptive scheduling. As compiler, we used the AVR port of GNU gcc.

- RENESAS - Renesas M32C/85, 32-bit microcontroller, 24MHz, 2MB Flash, 2MB RAM. The platform operates with a proprietary RTOS similar but not compatible with OSEKtime. As compiler, we used the corresponding port of GNU gcc.

- OSEK - Kanis OAK_EMUF, PowerPC MPC555 32-bit microcontroller with Floating-Point Unit, 40MHz, 448KB Flash, 26KB Fast RAM, 4MB RAM, 32 general purpose registers for integer operations, 32 floating-point registers for single or double precision operations. The platform operates with an OSEK compatible RTOS produced by WindRiver, namely OSEKWorks. The tool-chain accompanying this RTOS contains the DIAB C-compiler.

- Linux - Laptop system, Intel Pentium-M, 1.8GHz, 1GB RAM. The compilation measurements were performed using GNU gcc compiler under Cygwin; nevertheless, they are similar on RT-Linux and other derivates.

- InTime - Desktop system, Intel Xeon 3.0GHz, 1GB RAM. The platform uses WindowsXP as guest OS on top of the InTime RTOS version 2.23. The Microsoft Visual C/C++ 6.0 compiler was used for measurements.

As we can see from Table 6.1 the functionality of the runtime environment is highly customizable to include support for exception handling, cyclic dependencies between modules, table lookups for drivers and guards, hybrid EDF scheduling, and additional debugging information. On deep-embedded platforms, such as AVR, the resulting application code (including the TDL runtime environment) takes less than 2.5KB of Flash memory and uses less than 256 bytes of RAM. We can also see that in the general case, the footprint of the same application depends on the platform architecture (CISC/RISC, number of registers, instruction set), compiler optimizations,
6.5. MEASUREMENTS AND DISCUSSION

<table>
<thead>
<tr>
<th>Compile option</th>
<th>AVR</th>
<th>OSEK</th>
<th>Linux</th>
</tr>
</thead>
<tbody>
<tr>
<td>minimal</td>
<td>2148</td>
<td>103</td>
<td>26008*</td>
</tr>
<tr>
<td>+ Exceptions</td>
<td>238</td>
<td>0</td>
<td>1344</td>
</tr>
<tr>
<td>+ Cyclic dep.</td>
<td>160</td>
<td>0</td>
<td>208</td>
</tr>
<tr>
<td>+ Lookups</td>
<td>-2**</td>
<td>54</td>
<td>448</td>
</tr>
<tr>
<td>+ Hybrid EDF</td>
<td>204</td>
<td>30</td>
<td>1024</td>
</tr>
<tr>
<td>+ Except.+ Cycl.</td>
<td>430</td>
<td>0</td>
<td>1552</td>
</tr>
<tr>
<td>Normal (Full)</td>
<td>2842</td>
<td>187</td>
<td>29032</td>
</tr>
<tr>
<td>+ Debugging</td>
<td>2890</td>
<td>235</td>
<td>29048</td>
</tr>
</tbody>
</table>

* - OSEKWorks libraries account for around 20KB
** - compiler optimizations reduced the code with two bytes

Table 6.1: Code and heap size (bytes) for the example from Section 5.2

and additional RTOS libraries. Nevertheless, the TDL runtime environment has an almost negligible influence on the application code size.

Run-time behavior. We execute the resulted real-time application on the Kanis board, using the OSEKworks RTOS, and RM scheduling scheme. The Figure 6.2 presents the behavior of the two modules M1 and M2. The left-most part displays the output of the actuators from the two modules, in the case when M1 runs in the f11 mode and the two tasks have the same frequency but complementary outputs. The first two lines illustrate the outputs of the inc and dec connected actuators, whereas the third continuous line represents their sum from the second module. In the right side of the figure, we see the behavior of the application in the case when the module M1 runs in the mode f12. The dec task has twice the frequency of the inc task. Their sum is in this case a falling ramp signal. The offset between the rising edges of the sum and dec comes from the unit-delay of the TDL semantics.

Compiling and executing the application on any of the aforementioned platforms yields the same results in value and time when analyzed at the same time scale (milliseconds range). On a higher accuracy time scale (microseconds or hundreds of nanoseconds), the differences between the platforms in terms of CPU speed and I/O performance become visible, but do not influence the real-time requirements of the application, meaning that the actuators are still updated every 5ms.

6.5.2 Same deterministic behavior with transparent distribution

We intend to use the same application as in the previous case study in a distributed system. Thus, we first define the topology of the system: the number of nodes, the number and type of network connections between nodes, and the characteristics of the network. For our experiments, we take two similar Kanis boards with the same
characteristics as previously mentioned. We connect the boards through a CAN bus and decide for a bus rate of 100Kbit/s. The TDLComm layer implements a time-triggered protocol on top of CAN with global clock synchronization, that achieves similar performance with the standard TTCAN.

Following, we perform the integration of the modules in the previously defined distributed system. We specify the assignments of modules to nodes, and update their sensors and actuators accordingly. To determine the correctness of our implementation of the transparent distribution concept, we assign the modules M1 and M2 to the first board, and only the module M2 to the second board. We expect to obtain the same results from both modules M2 on the two nodes.

We recompile the application using the TDL compiler, and the bus-schedule generator plugin. The input for the plugin is the platform description file containing the topology of our distributed system. The tool detects two messages of two bytes each from the inc and dec tasks to the sum task located on the second node. Using the first protocol, the tool computes a network cycle of 10ms and generates the schedule consisting of two frames: a synchronization frame at time 0ms, and a data frame for the output values of the two tasks at time 8ms (the second node expects the frame at time 9.08ms).

We use the ANSI-C plugin to generate the stub of the module M1 for the second node and include the generated communication schedule. In the resulted middleware, the TDLComm layer is enabled to perform the information exchange between the two nodes.

Using a C compiler and the generated makefiles, we compile the functionality code of the two modules, the TDL runtime environment (including TDLComm), and the generated middleware (module wrappers, dependency files, OIL files) for each node. We obtain two real-time executables that we transfer on the two boards.

The left part of the Figure 6.3 presents the underlying communication mechanism of the TDLComm layer for transparent distribution. The TDLComm layer drives the
6.5. MEASUREMENTS AND DISCUSSION

Figure 6.3: Run-time behavior of the modules M1 and M2 in a distributed system CAN bus in a time-triggered fashion, resulting a similar behavior with a standard TTCAN bus. The two sum tasks on the two nodes run synchronized to the same global clock. The resulted output is the same on both nodes. The right part of the figure depicts the behavior of the system upon a mode switch. Both modules M2 receive the updated task outputs from module M1 at the same time. Thus, the mechanism of transparent distribution correctly abstracts from the topology of the platform, and the application can simply be placed either on one node as in the first case study, or on a distributed system without changing its timing behavior.

These two case studies show that with TDL real-time components can be developed independently and later integrated on a variety of platforms without influencing their time or value determinism.
Chapter 7

Outlook and Conclusions

In this dissertation, we presented a methodology for development of portable real-time software components. Its fundamental elements are the component-oriented language TDL, the concept of transparent distribution, the runtime environment based on a virtual machine, and the Platform Abstraction Layer. The PAL abstracts the communication requirements between the modules involved into an import relationship and physically running on different nodes of a distributed system, the platform specific scheduling scheme, and mappings of TDL entities to platform constructs. As a result, the development process may be easily automated and components may be designed and implemented independently one of another. Their timing and value behavior remains unchanged at integration, regardless of the target platform and the final system topology as long as the target platform offers sufficient resources.

However, along with all these advantages in comparison with classical approaches for development of real-time systems and current methodologies for distributing real-time computations, there may be also apparent disadvantages. In this chapter, we show that the disadvantages are minor and may be easily overcome.

One drawback may be the lack of support for aperiodic computation and communication. TDL is a purely time-triggered design language; therefore, its lack of implicit support for aperiodic tasks reflects in its execution environment and the mechanisms for transparent distribution. However, it is trivial to provide explicit support for such computation and additional communication with a server approach. The developer adds in his design a server task, which handles the aperiodic computations. The communication of this task with other similar tasks in the system results in one or more reserved communication slots, which may be used for aperiodic communication without interfering with the operation of other TDL tasks in the system. A similar approach is already present in many existing commercial time-triggered buses.

For distributed systems, both communication protocols presented in Section 5.5 help to transparently distribute different classes of applications. The communication schedule, the middleware, and the required data structures are automatically generated, simplifying the development of distributed applications.
CHAPTER 7. OUTLOOK AND CONCLUSIONS

Here, a first drawback may appear to be the proprietary TDMA protocols used in the communication. However, the protocols themselves do not make special assumptions regarding the underlying communication channel. Thus, they can operate on top of existing event-based protocols (such as CAN, Ethernet) and they introduce additional benefits in the simplification of the communication arbitration via their time-triggered operational mode. The underlying protocols act in this case as simple transport mechanisms. The time-triggered busses such as TTP/C and Flexray may be supported through the second protocol from Section 5.5.4, with adequate configuration files for the communication-schedule generator tool. The only remaining drawback is that some protocol specific features may no longer be available.

A second drawback may appear with the automatic generation of the communication schedule, which depends on the configuration of the underlying communication channel, the TDL modules of the distributed application, and their communication requirements. Changing any of these factors may change the resulting communication schedule. In this case, all nodes must update their view of the communication schedule. On a closer look at this apparent problem, we see that in practice this is not the case. There are typically two scenarios available:

- In experimental setups, the final configuration of the distributed system is not known, and several integration options are tested before settling to the final design. In this case, changing the functionality and the communication schedule is normal; thus, the automatic schedule generation is not a drawback but a significant advantage, because it simplifies and accelerates the development and testing of the distributed system. The integrator may easily change the configuration and the placement of the modules on node, whereas all the integration steps are automated.

- In production-ready setups, the final configuration of the distributed system is known. Apparently adding components may require the update of all nodes of the system. However, in practice there are only two possibilities: when the additional components are new and untested, the integrator implicitly starts a new experimental setup. In the case when the components are already developed and tested, the integrator simply starts with a full configuration. The integrator compiles and places the executable code (including module stubs) on each node, with the automatically generated communication schedule of the whole system. When later the integrator removes the physical nodes with the optional components, the stub modules of the optional components from the remote nodes (still in the executable code running on those nodes) simply operate with their initial values. Thus, there is no need to reconfigure the communication or update all nodes on the addition of known components. For example, in the case of a car, the integrator may easily design the car with all features such as ESP, ABS, EBD, and then in the cost-effective versions remove the ECUs providing services such as ESP and EBD, but retain the critical ones such as ABS.
The fault-tolerance level provided by the TDLComm layer may be insufficient. The TDLComm layer currently provides minimal fault-tolerance. On the other hand, it is highly flexible and can easily be adapted to support the underlying FT services of protocols such as TTP/C. In addition, these changes are not visible explicitly at the application level; thus, easily migrating applications from one topology, bus and protocol to another completely different system. The automatic schedule generator may introduce also new levels of fault-tolerance currently not present in most available commercial solutions. For example, more identical modules may reside on the same node and perform the same computations. In-memory faults from radiation or power failures may be easily filtered at the node level and then correlated with replicas from remote nodes.

Future work

The existing TDL development methodology can be further extended in several directions. Most require implementing additional capabilities to existing TDL software platform.

- **support for arbitrary task patterns** would be a solution for the ”1-to-10 problem”. In control systems, there is a strong desire to provide an environment response as close as possible to the query of its state through sensors. However, complex control laws require significant computations, which typically cannot be solved in the desired time. Hence, an alternative is to use a two-phase computation. In the first 10% of the control-cycle period, a pre-computation phase operates with the current sensor values and previously computed data to provide the control values for the actuators. In the remaining 90% of the control-cycle period, a post-calculation phase prepares data for the next cycle. With current TDL, the LET of the tasks is equal with their period; thus, actuators updated within the LET get old task values. The extension of the TDL task model would allow for the specification of tasks with LET smaller then their period, or micro-tasks (Menkhaus et al., 2005). This approach would allow the implementation of periodic sequences of tasks invocations and actuator updates. An advantage of this approach is that the real-time environment remains unchanged and compatible with older models.

- **implementation of a real-time microkernel** directly operating with task deadlines instead of classical priority levels. The existing PAL already makes the steps in this direction by providing support for arbitrary scheduling mechanism (TDL Scheduler supports by default RM, EDF and hybrid EDF), and task abstractions. This microkernel would improve the run-time performance of the supported platforms.

- **modeling of drivers** is important as most of the development time in embedded systems is allocated to interacting with the controlled hardware. The TDL abstraction of ”zero” logical time for the execution of the drivers applies only to
CHAPTER 7. OUTLOOK AND CONCLUSIONS

simple memory-to-memory transfer operations that typically are not as CPU expensive as port I/O operations. Sensor getters and actuator setters may actually take significant CPU time to alter the semantics of TDL at run-time. Thus, TDL would have to include support for the worst-case execution time of the sensors and actuators functionality and its tool-chain would have to consider them for scheduling analysis.

In the end, we consider that the presented work constitutes a solid foundation for future work, and that its extension may push the boundaries of known science further ahead.
Appendix A

Appendix

This chapter contains the technical details of our implementation. We provide the list of exceptions currently handled by the TDL runtime environment, a short description of the process of porting TDL to other platforms, the list of files and their purpose.

A.1 Exceptions

The current supported exceptions along with the context they are generated is presented below.

<table>
<thead>
<tr>
<th>Exception Name</th>
<th>Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDL_INITIALIZATION_FAILURE</td>
<td>the Initializer or one of its component failed</td>
</tr>
<tr>
<td>TDL_STARTUP_MODE_NOT_FOUND</td>
<td>there is no start mode in all modules</td>
</tr>
<tr>
<td>TDL_PORT_TYPE_MISMATCH</td>
<td>the ports have different types (TDLComm)</td>
</tr>
<tr>
<td>TDL_PORT_ACCESS_VIOLATION</td>
<td>the ports are accessed inappropriately (TDLComm)</td>
</tr>
<tr>
<td>TDL_CANNOT_CREATE_THREAD</td>
<td>threading initialization mechanism failed</td>
</tr>
<tr>
<td>TDL_CANNOT_DETACH_THREAD</td>
<td>threading cleanup mechanism mailed</td>
</tr>
<tr>
<td>TDL_MULTIPLE_FUTURE</td>
<td>multiple future instructions encountered in the same block</td>
</tr>
<tr>
<td>TDL_JUMP_OUT_OF_RANGE</td>
<td>instruction pointer is out of max E-Code count</td>
</tr>
<tr>
<td>TDL_UNKNOWN_OPCODE</td>
<td>invalid E-Code instruction</td>
</tr>
<tr>
<td>TDL_INVALID_DRIVER_NUMBER</td>
<td>the corresponding driver call does not exist</td>
</tr>
<tr>
<td>TDL_INVALID_GUARD_NUMBER</td>
<td>the corresponding guard does not exist</td>
</tr>
<tr>
<td>TDL_CANNOT_SET_THREAD_SCHED</td>
<td>the Scheduler thread failed</td>
</tr>
</tbody>
</table>
A.2 Supporting new platforms

In this section we briefly describe the steps required for porting the TDL runtime environment to a new platform.

The core functionality for a platform can be summarized by a small set of functions.

1. identify platform support for tasks. For non-native tasks define _TDL_THREADING to enable the creation of threads for user-tasks.

2. implement _TDL_System_Create_Exec_Threads when using threads for user-tasks to create the RTOS specific real-time threads. The TDL Scheduler requires a thread too.

3. implement _TDL_Release_Task to enable a user task for execution. Most systems using threads support the use of semaphores for signaling to a task wrapper waiting in a continuous loop.

4. for thread-based systems implement _TDL_Wait_Signal for the release functionality using a semaphore.

5. implement _TDL_Dispatch_Task to assign a task to the CPU for execution. When using threads their priority may be changed to enforce a dispatch from the RTOS scheduler.

6. implement _TDL_Preempt_Task to prevent a task from running. In general, lowering thread priority may be used as preemption mechanism.

7. implement _TDL_Sched_Wait to allow the TDL Scheduler to sleep until the latest dispatched tasks completes or the timeout expires.

8. implement _TDL_GetSystemTime to read the platform clock with the highest possible accuracy.

9. implement _TDL_System_Init and _TDL_System_Shutdown to initialize the necessary hardware and enable a safe startup and shutdown of the system.
A.3. FILES SUMMARY

The runtime system may be configured through the following macros:

- define the `TDL_ECOUNTER_TYPE`, `TDL_ECODE_ARG_TYPE`, `TDL_RELTIME_TYPE`, `TDL_ABSTIME_TYPE` to the corresponding data types available on the platform (e.g. char, int)

- for systems using threads use the `_TDL_TASK_WRAPPER_DATA` data structure to encapsulate their handles or other required runtime information

- define the `_TDL_ECODE_TIMESCALER(x)` to the required scaling factor for the platform time (e.g. x, x / 1000 to slow it down, etc).

- for debugging purposes define the size of the task/modes/module names through `_TDL_TASK_NAMES_CHARS`

- define the macros `TDL_TASK_WRAPPER(module, task)`, `_TDL_TASK_WRAPPER_HEADER(module, taskid)`, `_TDL_TASK_WRAPPER_FOOTER(module, taskid)` for the task wrappers

- map the release task macro `TDL_RELEASE_TASK(module, task, taskid)` to the actual implementation

- map the task termination macro `_TDL_End_Task()` to the appropriate system function (e.g. exit)

- define the nature of the internal tasks (E-Machine, Scheduler) through the macro `_TDL_INTERNAL_TASK(x)` into platform specific code (e.h. void x())

- assign the macro `_TDL_INVOKE_INTERNAL_TASK(x)` to the platform specific function to activate an internal task

A.3 Files summary

The runtime environment of TDL consists of the following files:

`emachine.c` The file contains the platform independent core that via the PAL API initializes the runtime environment, loads, initializes, and executes the applications and their modules handling all E-Code, scheduling, and networking control operations.

`emachine.h` The file contains the macro definitions of the E-Code instructions, all platform independent TDL internal types (E-Code blocks, tasks, modes, modules), and recognized error codes.
APPENDIX A. APPENDIX

system_dependent.c The file contains the implementation of the PAL API on the supported platforms, with each platform in its own section delimited by the platform-name macro definition. The functions available for each platform perform the hardware initialization, task/thread manipulation, clock services, and so on.

system_dependent.h The file defines the platform specific macros used to enable internal handling of TDL tasks in regard to the platform specific support, task wrapper definition, and additional references to required files (includes).

tdl_types.h The file contains the platform specific definitions of basic TDL types (boolean, int, float, etc). All changes to the actual types to be used on a specific platform should be made in this file.

tdl_comm.c For the distributed systems, the file implements the TDLComm specific network initialization, clock synchronization service, and basic packet sending/receiving operations.

tdl_comm.h The header file defines the TDLComm API for sending/receiving packets, network initialization, and system specific parameters.

The TDL runtime environment requires the following generated files to correctly load and process the modules of one or more applications:

modules.c The file represents a platform independent dependency solver file, which maps the symbolic references of the modules to actual positions in the table of loaded modules for each node. In addition, it includes the support for distribution via the _TDL_Packets structure that contains the bus schedule for each node. In the case of a non-distributed system the bus schedule is not available and the file contains only one section for the default node single.

modules.h The file includes the generated ModuleName_TDL.h files that contain the symbolic references of the compiled modules for each node. In addition, for the distributed systems, it exports to the TDL-Comm layer the networking data structures such as datagrams and packets.

Per node generated files (non-distributed or isolated systems are treated as single node systems):

NodeName.mak The file represents a node specific Makefile that must be included in a platform-specific Makefile to compile the real-time application for that node.

NodeName.OIL Only for the OSEK platform, this file represents the configuration of the RTOS for a given node.
Per module files:

**ModuleName.c** The file contains the user specified functionality code for a module. The data types used in the file must match the ones defined in the tdl_types.h file.

**ModuleName.h** The file contains the function prototypes of the user functionality code, along with additional data types that may be defined by the user.

Per module generated files:

**ModuleName_TDL.c** The file contains the module-wrapper code that interfaces the module functionality code with the rest of the system, the E-Code, the TDL tasks wrappers, modes, driver and guards functions, and TDL port definitions.

**ModuleName_TDL.h** The file contains the definition of some symbolic references used in the corresponding C file, along with prototypes of the TDL task wrapper functions.

**ModuleName_STUB.c** For the distributed systems, the file represents the stub of a module running on a remote node. It has a very similar structure with the regular ModuleName_TDL.c file, without task wrappers, guards, and with changed drivers and ports.

**ModuleName_STUB.h** For the distributed systems, this stub header file contains the similar information as the corresponding ModuleName.h file of the target remote module.
Bibliography


APEX. ARINC 653: Avionics application software standard interface (draft 15). avail. from ARINC Incorporated, June 1996. 26


Carnahan and Ruark. Requirements for real-time extensions for the Java platform, 1999.


E. W. Dijkstra. Cooperating sequential processes. EWD 123, Technical University, Eindhoven, the Netherlands, 1965. 22


IEEE Information Technology. Portable Operating System Interface (POSIX), 2006. 26 27


BIBLIOGRAPHY


210


# List of Figures

2.1 Environment time versus Soft time ........................................... 15
2.2 The embedded programming models \cite{Kirsch2002} ......................... 15
2.3 Implementation of the synchronous programming model \cite{Kirsch2002} . 16
2.4 Implementation of the scheduled programming model \cite{Kirsch2002} . 17
2.5 Implementation of the timed programming model \cite{Kirsch2002} ....... 17
2.6 The OSEK/VDX task model .................................................... 25
2.7 The structure of a CAN frame .................................................. 32
2.8 The arbitration mechanism of CAN .......................................... 33
2.9 TTCAN System Matrix and traffic pattern .................................. 35
2.10 The logical blocks of a TTP node ............................................ 37
2.11 Overview of the presented time-triggered protocols ..................... 40

3.1 The Logical Execution Time concept ........................................ 42
3.2 Visual representation of a TDL task ........................................ 49
3.3 Implicit driver connecting two TDL tasks ................................ 52
3.4 The interaction of TDL task instances ..................................... 56
3.5 Multimeter application ......................................................... 62
3.6 TDL modules in a distributed system of three nodes ..................... 66
3.7 Communication between the tasks of modules M1 and M2 from Figure 3.6 66
3.8 The execution of the two tasks from Figure 3.7 .......................... 67
3.9 The communication between the two tasks from Figure 3.7 . . . . . 67
3.10 Usage of module stubs to transparently distribute an application ...... 69
3.11 Configuration trace for the execution of the TDL module from Listing 3.7 75
3.12 Execution flow for the TDL module from Listing 3.7 ................... 77

4.1 Timeline for the activities of the TDL module M from Listing 4.2 .... 91
4.2 E-Code of the TDL module M from Listing 4.2 .......................... 92
4.3 E-Machine interaction with E-Code, drivers, guards, and tasks ....... 92

5.1 Simplified TDL tool-chain using PAL ...................................... 107
5.2 The role of PAL at the startup of a TDL application .................... 108
5.3 The layered architecture of the TDL runtime environment ................ 109
5.4 Visual representation of the modules M1 and M2 ....................... 111
**LIST OF FIGURES**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.5</td>
<td>Dataflow between TDL entities and ports</td>
<td>120</td>
</tr>
<tr>
<td>5.6</td>
<td>Usage of PAL for time sampling and planning</td>
<td>137</td>
</tr>
<tr>
<td>5.7</td>
<td>TDL task scheduling through PAL</td>
<td>139</td>
</tr>
<tr>
<td>5.8</td>
<td>Example of RM scheduling of three TDL tasks</td>
<td>141</td>
</tr>
<tr>
<td>5.9</td>
<td>TDL tool-chain for distribution</td>
<td>146</td>
</tr>
<tr>
<td>5.10</td>
<td>TDLComm services</td>
<td>148</td>
</tr>
<tr>
<td>5.11</td>
<td>Data flow between modules through TDLComm</td>
<td>149</td>
</tr>
<tr>
<td>5.12</td>
<td>Port values, messages, datagram, packet and frame</td>
<td>151</td>
</tr>
<tr>
<td>5.13</td>
<td>Stop-drivers initiated encapsulation of port values</td>
<td>153</td>
</tr>
<tr>
<td>5.14</td>
<td>TDLComm initiated encapsulation of port values</td>
<td>153</td>
</tr>
<tr>
<td>5.15</td>
<td>Clock-synchronization through TDLComm</td>
<td>159</td>
</tr>
<tr>
<td>5.16</td>
<td>Example of communication scheduling skipping unused messages</td>
<td>162</td>
</tr>
<tr>
<td>5.17</td>
<td>Example of messages when using a single communication-scheduling table</td>
<td>162</td>
</tr>
<tr>
<td>5.18</td>
<td>Ports to/from packets direct mappings using hyper-periods</td>
<td>163</td>
</tr>
<tr>
<td>5.19</td>
<td>Dynamic mapping of messages to frames</td>
<td>167</td>
</tr>
<tr>
<td>5.20</td>
<td>Ports to/from packets dynamic mappings</td>
<td>168</td>
</tr>
<tr>
<td>6.1</td>
<td>The TDL Tool-chain</td>
<td>176</td>
</tr>
<tr>
<td>6.2</td>
<td>Run-time behavior of the modules M1 and M2</td>
<td>190</td>
</tr>
<tr>
<td>6.3</td>
<td>Run-time behavior of the modules M1 and M2 in a distributed system</td>
<td>191</td>
</tr>
</tbody>
</table>
Listings

3.1 Sample TDL module with constants and types .................. 45
3.2 Ports and drivers example ........................................ 52
3.3 Non-harmonic mode-switch ......................................... 58
3.4 Harmonic mode-switch .............................................. 58
3.5 Sample TDL module exporting a task output port ............... 63
3.6 Sample TDL module importing remote resources ................. 64
3.7 Four tasks example .................................................. 73
4.1 Algorithm for compiling a TDL module into E-Code ............ 87
4.2 Sample TDL source for compilation: module $M$ with two tasks 90
4.3 Algorithm for executing an E-Code instruction block of a TDL module 91
4.4 Algorithm for executing a TDL module through E-Code .......... 96
4.5 Algorithm for modular composition through E-Code ............ 97
4.6 Updated algorithm for compiling a TDL module into E-Code with Future+Return solution for cyclical dependencies .......... 100
4.7 Updated algorithm for executing one or more TDL modules using E-Code with Future+Return solution for cyclical dependencies 101
4.8 Updated algorithm for compiling a TDL module into E-Code with alternate call for tasks solution for cyclical dependencies 102
4.9 Additional algorithm for executing task termination drivers for cyclical dependencies ........................................... 102
4.10 Updated algorithm for executing one or more TDL modules using E-Code with alternate task call solution for cyclical dependencies 103
5.1 Module $M_1$ ......................................................... 112
5.2 Module $M_2$ ......................................................... 112
5.3 Overview of module wrapper $M_1$ ................................. 113
5.4 Macro expansion of module descriptor ............................. 115
5.5 Ports of module $M_1$ ............................................... 117
5.6 PAL - Driver wrapper ............................................... 118
5.7 Drivers of module $M_2$ from Listing 5.2 ......................... 119
5.8 Algorithm for generating the implicit TDL drivers .............. 121
5.9 TDL Guard ......................................................... 123
5.10 C Implementation .................................................. 123
LISTINGS

5.11 Guard wrappers of module M1 from Listing 5.1 .................................. 123
5.12 TDL Sensor/Actuator ................................................................. 124
5.13 C Implementation ................................................................. 124
5.14 PAL - Task Wrapper ................................................................. 126
5.15 Macro expansion of task wrapper for RT-POSIX ......................... 127
5.16 Task wrappers of module M1 from Listing 5.1 .......................... 127
5.17 Algorithm for generating a module wrapper .............................. 130
5.18 Algorithm for generating a stub-module wrapper ....................... 131
5.19 Core algorithm of the TDL Scheduler ...................................... 140
5.20 Hybrid EDF using precompiled dispatch tables ......................... 141
5.21 Distributed system configuration file ....................................... 146
5.22 Activation of the TDLComm layer from TDL Scheduler ............. 155
5.23 Packet encapsulation ............................................................. 165
5.24 Port-value extraction ............................................................. 165
5.25 Dynamic packet encapsulation ............................................... 168
5.26 Dynamic extraction of port values from packet ....................... 169