Techniques for Property Preservation in the Development of Real-Time Systems

Jewgenij Botaschanjan

Institut für Informatik
der Technischen Universität München

Techniques for Property Preservation in the Development of Real-Time Systems

Jewgenij Botaschanjan

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

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Helmut Seidl

Prüfer der Dissertation:

1. Univ.-Prof. Dr. Dr. h.c. Manfred Broy
2. Univ.-Prof. Dr. Alois Knoll

Die Dissertation wurde am 26.06.2008 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 11.11.2008 angenommen.

Abstract

A significant characteristic of *real-time* systems is the existence of constraints concerning timeliness. Most real-time systems are *reactive*, i.e., they react to events or changes in their environment. Additionally, the tasks enjoined on systems from the reactive domain are often inherently safety-critical. By this, ensuring the correctness of these systems has high significance. *Model-based development* has been recognized as a promising approach for the mastering of the inherent complexity of reactive systems. While it has successfully come into use at the advanced stages like implementation, for its continuous employment in the course of development additional research has to be done. The methodical mastery of real-time requirements and the preservation of system behavior throughout different steps of the model-based development process are the key factors for constructing reactive real-time systems which are conform to their time and functional requirements and meet the real stakeholders' needs.

This thesis examines both the problem of modeling, verification, and validation of real-time requirements as well as the problem of behavioral preservation in the context of model-based computer-aided development of reactive real-time systems.

On the one hand, the time behavior must be expressible within the specification language. On the other hand, the high complexity of reactive systems demands for a separation of concerns and modularization of the specifications along different aspects of functional and time behavior. For the formal modeling of demanded system behavior a denotational framework is presented which is able to capture and interrelate both time and functional requirements. The framework is applicable in the early phases of development where the "unstructured" specification consists of a number of constraints on the system and/or on its environment as well as in the subsequent phases where the system borders are determined and the system's functionality is partitioned and structured, e.g., by components. Using this framework time constraints can be analyzed respective consistency and completeness. Moreover, they can be formally related to and distributed over functional components. By this, the framework establishes the time behavior as an

independent but integrated view on the system under development. In particular, based on this framework a taxonomy of time requirements was created and formalized.

Application logic models belong to the first artifacts which are settled in the solution space. These models serve as a starting point for the developed property-preserving transformation process which results in a deployed system. The process is based on concrete modeling paradigms for application logic, clustered task architecture, and time-triggered deployment platform which are described within a mathematical framework which allows the logical characterization of operational models. For transitions from the application logic to the task architecture and from the task architecture to the deployed system synthesis procedures are presented and their correctness, i.e., property preservation, is formally proved. Finally, the operational models are mapped to the denotational specification formalism from above. By this, they can be verified in regard to the fulfillment of time and functional specifications.

Zusammenfassung

Eine bezeichnende Eigenschaft von *Echtzeitsystemen* ist die Existenz von Pünktlichkeits- und Rechtzeitigkeits-Anforderungen. Die meisten Echtzeitsysteme sind *reaktiv*, d.h., sie reagieren auf externe Ereignisse oder Änderungen in ihrer Umgebung. Die Aufgaben, die diese Systeme erfüllen müssen, sind oft inhärent *sicherheitskritisch*. Daher hat die Sicherstellung der Korrektheit solcher Systeme eine hohe Priorität. *Modellbasierte Entwicklung* ist ein vielversprechender Ansatz um die Komplexität reaktiver Systeme zu beherrschen. Der Einsatz der modellbasierten Methode in den fortgeschrittenen Phasen der Entwicklung, wie Implementierung, erwies sich als fruchtbar. Jedoch für einen kontinuierlichen Einsatz im Laufe des gesamten Entwicklungsprozesses, bedarf es einer Weiterentwicklung dieser Methode. Methodische Beherrschung der Echtzeitanforderungen und die Erhaltung des Systemverhaltens zwischen verschiedenen Phasen eines modellbasierten Entwicklungsprozesses, sind die Schlüsselfaktoren für die Konstruktion reaktiver Echtzeitsysteme, die konform zu ihren zeitlichen und funktionalen Anforderungen sind und die eigentlichen Bedürfnisse aller Interessengruppen erfüllen.

Die vorliegende Arbeit untersucht sowohl das Problem der Modellierung, Validierung und Verifikation von Echtzeitanforderungen als auch das Problem der Eigenschaftserhaltung im Rahmen der modellbasierten rechnergestützten Entwicklung reaktiver Echtzeitsysteme.

Einerseits muss das Zeitverhalten in einer Spezifikationssprache ausdrückbar sein. Andererseits erfordert die hohe Komplexität reaktiver Echtzeitsysteme die Aufteilung und Modularisierung von Spezifikationen entlang verschiedener funktionaler und zeitlicher Aspekte. Für die formale Modellierung des geforderten Systemverhaltens ist ein denotationaler Formalismus präsentiert, welcher in der Lage ist, funktionale und zeitliche Anforderungen zu erfassen und zueinander in Beziehung zu setzen. Der Formalismus ist für die Anwendung in den frühen Entwicklungsphasen gestaltet, in welchen eine unstrukturierte Spezifikation, bestehend aus einer Anzahl von Anforderungen an das System und/oder an seine Umgebung, existiert; sowie für die Anwendung in den nachfolgenden Phasen, in welchen die Systemgrenzen bereits abgesteckt sind

und die Funktionalität des Systems partitioniert und strukturiert ist, beispielsweise durch Komponenten. Mit Hilfe des Formalismus können zeitliche Anforderungen bezüglich Konsistenz und Vollständigkeit analysiert werden. Außerdem können die Anforderungen über eine Komponentenarchitektur verteilt und den einzelnen Komponenten zugewiesen werden. Dadurch wird das Zeitverhalten als eine unabhängige jedoch integrierte Sicht auf das zu entwickelnde System etabliert. Insbesondere wurde mit Hilfe dieses Formalismus eine Taxonomie der Echtzeitanforderungen erstellt und formalisiert.

Modelle der Anwendungslogik gehören zu den ersten Artefakten, die im Lösungsraum angesiedelt sind. Diese Modelle dienen als der Ausgangspunkt für das von uns entwickelte eigenschaftserhaltende Übergangsverfahren, welches in einem integrierten System mündet. Das Verfahren basiert auf den Modellierungsparadigmen für die Anwendungslogik, Task-Architektur und zeit-gesteuerte Zielplattform. Diese wurden innerhalb eines mathematischen Rahmenwerks erstellt, welches die logische Charakterisierung der operationalen Modelle erlaubt. Für die Übergänge von der Anwendungslogik zur Task-Architektur sowie von der Task-Architektur zum integrierten System wurden spezielle Synthese-Verfahren entwickelt und deren Korrektheit im Sinne von Eigenschaftserhaltung formal nachgewiesen. Schließlich wurden operationale Modelle auf den oben vorgestellten denotationalen Formalismus abgebildet. Dadurch können diese auf die Konformität zu funktionalen und Echtzeitanforderungen überprüft werden.

Danksagung

In diesen Zeilen möchte ich die Gelegenheit nutzen und all denjenigen zu danken, die zu erfolgreichen Fertigstellung dieser Arbeit beigetragen haben. An erster Stelle möchte ich meinem Doktorvater, Professor Manfred Broy, dafür danken, dass er mich in seine Forschungsgruppe aufgenommen hat, und mir während der Erstellung dieser Arbeit sowie während der Projektarbeit mit Rat und Tat zur Seite stand. Ferner geht mein Dank an Professor Alois Knoll dafür, dass er das Zweitgutachten übernommen hat.

Das AutoFocus Task Modell, ein wesentlicher Teil vorliegender Arbeit, ist im Rahmen des Verisoft Automotive Projekts entstanden. Ich möchte mich an dieser Stelle bei dem gesamten Verisoft Team an der TU München bedanken für die vielen interessanten Diskussionen und Anregungen.

Mein Dank geht an Iain Bester, Alexander Harhurin, Leonid Kof und Doris Wild dafür, dass sie die beschwerliche Aufgabe des Korrekturlesens meiner Arbeit auf sich genommen haben.

Meiner Frau Diana und meinen Kindern Nikita, Valeria und Arthur, sowie meinen Eltern Larissa und Vladimir Botaschanjan möchte ich für die Unterstützung in jeder Phase dieser Arbeit und für viel Geduld, die sie während dieser Zeit aufbringen mussten, danken.

Contents

1 Introduction

The search for adequate models of programs and systems as well as reasoning about their correctness are important and well-established research areas in computer science. The level of maturity individual formal modeling and verification techniques have achieved makes them applicable in the development of industrial-size systems. This motivates a trend in software engineering which aims at the integration of formal methods into development processes. On the other hand, in practice more and more safety-critical tasks are committed to software-based systems. The size and complexity of these *embedded* systems, e.g., applied in vehicles or aircraft, is growing exponentially [Bro06a, SZ06, Gri03]. One needs to guarantee the correctness of such systems, since a failure while performing a safety-critical task could lead to catastrophic results (e.g., cf. [Lan97]). These circumstances suggest the integration of formal methods and verification techniques into the development process of embedded systems.

Embedded systems are designed for a continuous interaction with their environment. Therefore, they have to react when they observe certain environment events. In other words, the system gets active if it perceives, e.g., through sensors, or by receiving a signal over a transmission link, that a certain event has happened in the environment. The system reacts to this event, using e.g., actuators or sending a response message through a bus line. Thus, this kind of systems is also called *reactive* systems. Reactive systems are constructed for an in principle infinite repetition of their activities. This fact distinguishes them from "classical" programs, which are started, run to completion and terminate. This fact also motivates the need for special formal models of reactive systems. Another particularity of reactive embedded systems is that they monitor and manipulate physical processes. In physical processes, events are stretched over time. Thus, often reactive systems have not only to react correctly, but also to react in time. This class of systems is called *real-time* systems.

In the present thesis we study how the time behavior of reactive real-time systems can be modeled and analyzed throughout their development and how the correctness of this behavior can be ensured. For this purpose, we provide formal models applicable at different development stages and describe methods which provably preserve the time and functional system behavior during the transition from one model to another. More precisely, we define formal models for specifica-

tion of time behavior and for the realization of their (functional) behavior. We instantiate these models for specification, design and implementation paradigms, which are applied during the development of embedded systems. In particular, we formalize time constraints, which are typically used to describe the real-time behavior; we provide formal models of a CASE-tool design paradigm, of a task architecture concept, and of a time-triggered deployment platform. By this, we cover the phases of requirements engineering, design and implementation/deployment in the development process of embedded real-time systems. In every development stage our concern is to ensure the property preservation during the transition to the succeeding one. Thus, we formulate assumptions which, e.g., a task architecture must discharge in order to provably preserve the behavior of the design model. We also provide formal proofs of the property preservation under these assumptions.

Outline. We give a short overview of model-based development, as a promising development paradigm for the integration of formal models and methods in Section 1.1. Then, in Section 1.2 we set the notion of time into the context model-based development processes and discuss how the time behavior can be incorporated as an independent, but yet integrated model-based view on the system. Section 1.3 describes the contributions and the structure of the present thesis. Finally, in Section 1.4 we introduce two case studies, which were used for the evaluation of our results. They serve as running examples throughout this thesis.

1.1 Model-Based Development

The advantages of using models in systems engineering are indisputable. Models enable to build views on and abstractions of systems. In other words, models allow to consider certain aspects like application logic, while fading out others, like package structure, technical platform, or physical distribution of the system components. Another promising perspective provided by models is the possibility of formal analysis. Its prerequisite is the formal foundation of the applied models. The benefits the application of formal analysis brings along are:

- the provability of system correctness,
- improved early detection of inconsistencies, e.g., in the specification and design artifacts, as well as
- the possibility to remove ambiguities from these artifacts, which are often contained in the informal textual or graphical descriptions.

Model-based development is widely recognized as a method of choice for efficient and safe design of computing systems. Development processes referred to as *model-based* provide the ability to represent requirements, specifications, and designs using (graphical) domain-specific languages [SPHP02b]. As discussed above, the assignment of mathematical meaning to the domain-specific languages, i.e., the definition of formal semantics of applied modeling paradigms, allows to verify created models and, provided the semantics is operational, to simulate the resulting behavior for validation purposes. Coupled with increasingly sophisticated code generation technology (researched in the literature [ABG95, HRR91, GEB03, BS01b] and

established in the commercial tools, like TargetLink[1] or Real-Time Workshop[2]) model-based development aims at elevating the level of abstraction at which systems are designed, analyzed, validated, coded, and tested. Further on, formally-founded model-based development permits to structure functional and non-functional descriptions of the system along multiple concerns (or aspects), to integrate them to complete system descriptions, and to establish tracing between the elements of different models and views. Finally, model-based development is also concerned with providing automated support to software engineering. Over the last years, CASE tools, like MATLAB/Simulink[3], Rose RT[4], StateMate[5], or ASCET-SD[6] became established instruments for the design of embedded systems, e.g., in the domains of automotive and avionics.

An important issue, for the formal-foundation of model-based development as well as for the introduction of verification into the model-based development, is the *integration* of different models. *Vertical* integration relates products of different development phases, like implementation and design models, while *horizontal* integration answers the question about the interrelationship of different views and fragments of the system produced during the same development phase, e.g., the dependencies between different subsystems, relations between functional and non-functional requirements, *etc.* The goal of integration is to preserve the properties proved to hold for individual models. For functional properties the vertical integration can be considered as a *refinement relation*: The realization of some functionality refines the corresponding requirement if the behaviors the realization can show are contained in the behaviors permitted by the requirement. The horizontal integration is handled by the notion of *composition*: The correct functioning of a particular subsystem or component often relies on *assumptions* about the behavior of its environment, which consists of further components. Thus, the component, provided that its environment operates according to the assumptions, must give back certain *guarantees* about its own behavior to the environment. The composition expresses, when and how the assumptions and guaranties of different components fit.

We are interested in modeling and analyzing the time behavior on different stages of the model-based development process of reactive real-time systems. Thus, we present models which are able to capture the specified time behavior, and integrate them horizontally and vertically with other models of the system, like logical, or technical design, and implementation/deployment solution. By this, we install time behavior as a separate, but yet integrated view on the system.

In practice, the vision of computer-aided model-based development sketched above, is realized in relatively narrow functional domains and at advanced stages of development. The CASE tools listed in the previous paragraphs incorporate isolated applications of the ideas of the model-based approach, and provide no concepts for integration with artifacts from preceding and subsequent development steps [BBC+03]. In the current thesis we present a model stack consisting of a CASE tool model for logical design, task architecture model for technical design and a programmer's model of a time-triggered distributed deployment platform. Every stack layer is

[1] see http://www.dspace.com (20.06.2008)
[2] see http://www.mathworks.com/products/rtw (20.06.2008)
[3] see http://www.mathworks.com/products/simulink (20.06.2008)
[4] see http://www-01.ibm.com/software/rational (20.06.2008)
[5] see http://modeling.telelogic.com/modeling/products/statemate (20.06.2008)
[6] see http://de.etasgroup.com/products/ascet (20.06.2008)

integrated with others in the sense that conformance conditions are provided, which ensure that the upper layer is vertically integrated with the lower ones. In [BGH+06] we have shown how our task architecture model can be emulated by the aforementioned CASE tool. The transition from the task architecture model to the deployed system was prototypical realized for the concrete time-triggered platform via code generation in the scope of the Verisoft Automotive[7] project. By this, the evidence of feasibility of the envisioned model stack was furnished. Finally, the operational models from the stack can be mapped to the denotational time behavior specification, described in the previous paragraph. By this, we facilitate the model-based development starting with a formal requirements specification up to the deployed system.

The next section contains the description of the reference model-based development process we rely on. The emphasis of the description lies on the notion and role of time in different development phases.

1.2 The Notion of Time in the Model-Based Development

The overall design flow of the model-based development process, which will be discussed next, is sketched in Figure 1.1. The artifacts are mapped to their respective time domains. The figure abstracts from the iterations in the development, e.g., it hides feedback and rollback loops between artifacts away. The process reflects the typical activities and their order during the industrial development of embedded systems [BFG+08, SZ06].

The system specification is by convention subdivided into functional and time behavior requirements. The latter class is a subclass of the so-called performance requirements. The former requirements class serves as an input for the design phase, where the logical and task architectures emerge. Thereby, the task architecture clusters the artifacts of the logical architecture, typically components or classes, by deployable units (tasks). The task architecture, time behavior requirements, as well as the model of the target deployment platform serve as inputs for building the analytical task model (ATM). This model serves as the basis for the schedulability analysis. Finally, the tasks from the task architecture are deployed on the target platform using the results of the schedulability analysis documented by ATM. In the remainder of this section we give a more detailed description of the artifacts from Figure 1.1 and transitions between them in especial consideration of time behavior.

Normally, development projects start with the requirements engineering phase, which aims at creating specification of the system under development. Specifications usually provide a number of constraints on the time behavior of the real-time system under construction. More precisely, they demand a number of functions or features to be realized. These functions are causally ordered by *user cases*, which document the data- and control-dependencies between them. One of the actors in these use-case scenarios is the environment of the system (which can consist of several further systems or human users). The data- and control-flow through the interface to the environment is often constrained respecting its timing. By this, the time constraints refer to the end-to-end behavior visible to the environment of the system. Moreover, the behavior of

[7] see http://www.verisoft.de and http://www4.in.tum.de/~verisoft/automotive (20.06.2008)

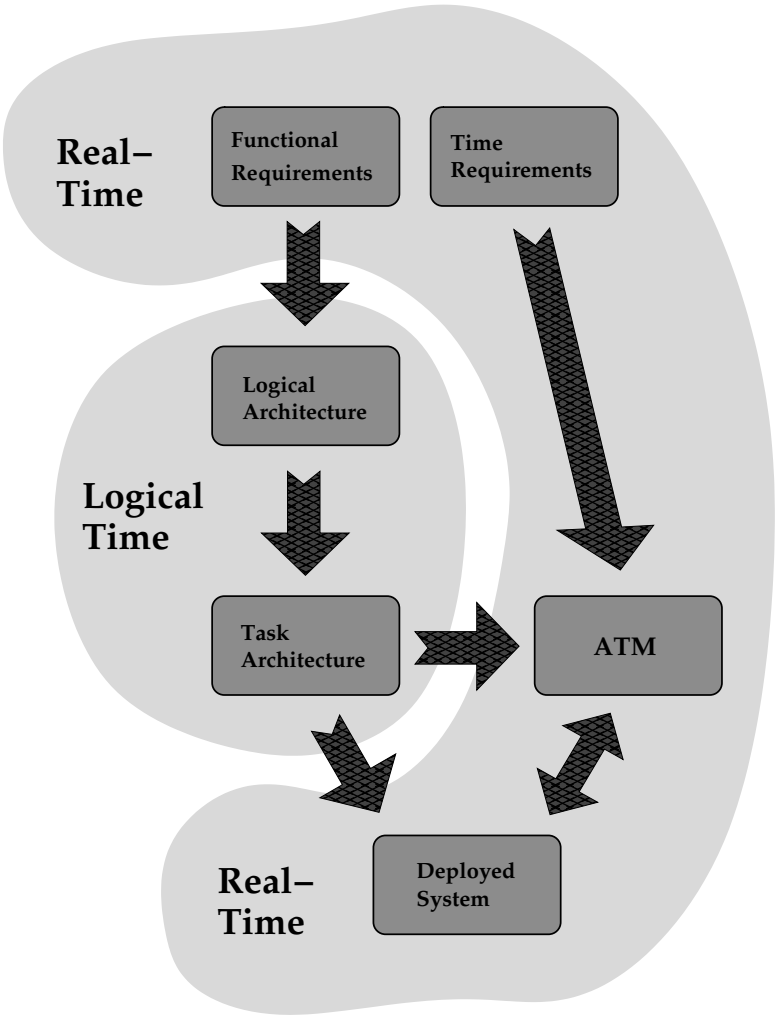


Figure 1.1: *Design Flow & Time Domains*

the environment itself underlies certain time constraints, which are also documented in specifications. In the majority of cases, the time constraints are not exact but rather define certain tolerance bounds in which the actual time behavior of the realized system must lie. In other words, there may exist a refinement relationship between the time behavior of the specification and the corresponding implementation.

A common classification criterion of a time constraint is the observable effect of its miss by the system. In this context, there exists a distinction between *soft real-time* and *hard real-time* constraints. Missing a hard real-time constraint can lead to catastrophic consequences, like damage to equipment, loss of money, or even injury to human lives. Violation of soft real-time constraints merely decreases the performance of the system. In this thesis we will address solely hard real-time constraints. By analogy with functional requirements we can denounce a system which misses some hard real-time constraint as an invalid realization of the specification.

The timing of both specification and implementation is described in the physical time units (typically in the order of magnitude of milli- or microseconds). In other words, the *real-time* axis must be used to model the time behavior of artifacts from these phases. In between, i.e., during the design phase other modeling paradigms come to use. In order to cope with the complexity, they provide modularization concepts to the developers, like components, classes, modules, *etc.* and – either implicitly or explicitly – *models of computation* which compose these modules. Models of computation are defined upon abstractions about the actual behavior of the system. Examples are concurrent execution, global clock, instantaneous computation, zero- or unit-delay communication, atomicity of actions, lossless communication links, and so on. These are extremely useful abstractions, which allow to simplify the description of the application logic, to make it platform independent and by this portable, and, finally, to reduce the state space and facilitate the application of formal analysis techniques. For the same purposes design paradigms are often based upon *logical-time* axes. There, the progress of time is expressed in special artificial units, like ticks, transition steps, coefficients of a global natural-valued clock, or (partial-)ordered events.

With regard to the real-time behavior the combination of the model-based approach with formal verification brings an interesting challenge: The real-time requirements are expressed using the *physical* time. In contrast, as Figure 1.1 suggests the functional requirements are realized in the settings of logical time. At the end of this process the made up models have to be transferred from CASE-tools (via code and wrapper generation as well as schedule synthesis) to a deployment platform. By this, their time behavior is dictated by the physical clocks again and can be validated with respect to the fulfillment of time requirements. By the latter change of the underlying model of computation the overall behavior of the deployed system can deviate from the behavior of the application logic in the execution semantics of a CASE-tool. This surely reduces the utility of the verification effort put on the pure application logic. The aggravating circumstance for the above scenario is the well-known curve [Boe81], which shows the cost of a bug fix in relation to the point in time of its detection. The curve grows exponentially.

In order to improve this situation, several approaches [BJ05, IM07, Dam05] suggest to spread and descriptively assign the time behavior to the design artifacts. This allows to analyze the satisfaction of time requirements already for particular design solutions, under the assumption that the implementations of design artifacts will comply with their assigned time (sub-)requirements.

The authors of [BJ05] propose a necessary condition for the realizability of a timing model: All time requirements should be met under the assumption of maximal parallelism. In fact if every function has a set of dedicated resources which it can use exclusively (e.g., processor, memory, communication link to everyone of its partners, *etc.*) and provided the communication takes no time, it should be possible to find a solution which meets all time requirements. Otherwise, the time specification is contradictory. Obviously, this condition is not sufficient, thus, more elaborate models must be created. The analytical models must perceive the target deployment platform as well as the functional dependencies between the design artifacts. Thereby, the classical scheduling-theory [Jos01, LL73, TBW94] as well as more recent approaches [BJ05, Pop00, AFM+03] make different simplifying assumptions about both the plat-

form and the application logic. However, they do not answer the question, how these assumptions can be discharges by their concrete counterparts?

The collection of assumptions incorporated by different schedulability analysis approaches is called a *task model*. A justification for a task model is the fact that in most cases the target deployment platform consists of already existing technical solutions, like OSes, buses, *etc.*, which can consider different relationships between functions only to some extent. For example, time-triggered OSes, like OSEKtime, can activate their applications on timer events only. The causal relationships between functions cannot be explicitly established in OSEKtime. Thus, correct-by-construction approaches, which provide programmer's models for implementation of reactive systems, like [BBG+08, HHK03, Hon98, STY03, BW95], incorporate a limited set of inter-application relationships and communication pattern. However, if we consult our envisioned development process in Figure 1.1, we see that their starting point is the task architecture or implementation. This brings us back to the problem from above: How can simplifying assumptions of the task model be discharged by the application logic?

One possible solution is to delegate the verification to the task architecture. Approaches, like [BRS04, BGH+06], provide tool support and give methodological guidelines for the transition from the application logic (also called logical architecture or design model) to the task architecture, but do not guarantee property preservation during this procedure. Another approach which goes in the same direction can be found in [BBG+08]. There, the authors have formally proved that the (temporal) properties of any application modeled using a special design paradigm, called AutoFocus Task Model (AFTM), are preserved after the deployment on a distributed time-triggered platform using a special schedule synthesis algorithm. We will learn more about the results presented in that paper in the remainder of this thesis. Another answer to the above question is to ensure the property preservation during both transitions: from application architecture to task architecture and from task architecture to deployed system. Such property-preserving procedure is one of the main contributions of the present work.

In these settings four interesting investigation fields naturally arise:

(1) How can real-time requirements be analyzed respecting their consistency?

(2) How can real-time requirements be mapped to a design model and distributed over its artifacts (components, classes, *etc.*)?

(3) How can a design model be analyzed in respect of fulfillment of time requirements in the view of different time axes?

(4) How can a design model be mapped to a technical platform, without loss of functional or real-time properties of this model?

The present thesis answers these questions, as stated in the following section.

1.3 Contributions and Outline

In the current section we give a brief overview of the material presented in the remainder of this thesis. It is partly based on our contributions in [BBG+08, BGH+06, BKKS05, BJ05, BRS04].

The thesis consists of four major chapters which handle three major topics identified in the previous sections: (1) specification and verification of time behavior, as well as the preservation of time and functional behavior during (2) design and (3) deployment.

Chapter 2 presents a denotational framework for modeling and analyzing time requirements. It provides an interval-based temporal logic, which allows the formalization of time and functional requirements. Using this temporal logic, a taxonomy of time constraints, e.g., response time, inter-arrival time *etc.*, which are typically found in specification and design documents, is formalized. Furthermore, the chapter provides flexible and powerful mechanisms for building timed behavior specifications out of individual time constraints and functional requirements.

In Chapter 3 the notion of refinement of time behavior is introduced. Using refinement, a number of operations on time constraints, which usually have to be performed during the development process, are formalized. In particular, Chapter 3 offers mechanisms for decomposition of time requirements, derivation of further implicit requirements, and distribution of requirements over component networks. Further on, the relationships between time and functional requirements as well as between unstructured collections of time requirements and individual component specifications are written down formally. The relationships describe precisely, e.g., what consequences a refinement of a time requirement will have on the corresponding component specification and vice versa. This allows to establish time behavior as a separate *view* on the system, which is fully integrated with the other views, like the behavioral or architectural view. The changes executed on this view are pursuable to the rest of the system model and vice versa.

The mapping of time requirements to the component architecture described above preserves the descriptive style commonly used in the specifications. However, a substantial class of design paradigms are based on operational semantics. Operational description of system behavior enables automatic or schematic transition from design to implementation. For example, most CASE tools allow code generation out of their models. Chapter 4 presents an automata-based formalism, called *state transition system* (*STS*) [BP99, PP05], for the description of system behavior. By use of STSs we formalize the semantics of a CASE tool called AutoFocus[8] [HSE97]. The tool describes the system model by a set of communicating Mealy machines. Their transition functions produce infinite sequences of states. Different machines are composed using the global clock and parallel execution. The composition constructs the global system step. This step is the measurement unit of the logical time axis of AutoFocus. However, it does not mean much for the real-time behavior of the system, since its real-time length should utmost vary for different transitions, when run on the target hardware. This motivated us to group transition steps between a request and a corresponding response for every AutoFocus machine. Such a unit of system execution is more meaningful for the analysis of time behavior, since it can be associated with the functional and real-time requirements, which relate inputs and outputs of the system. The grouping is achieved by the transformation of AutoFocus models to a novel task architecture paradigm called AutoFocus Task Model (AFTM) [BGH+06, BBG+08]. The further contribution of Chapter 4 is the correctness proof of that transformation under certain assumptions. Finally, the STS formalism is mapped to the denotational framework from Chapter 2.

[8] see http://autofocus.in.tum.de and http://www4.in.tum.de/~af2 (20.06.2008)

This allows conformance checks between time and functional specifications and their respective realizations.

Another purpose of AFTM is to prepare the system for the subsequent deployment. The abstractions incorporated into design paradigms, like AutoFocus or AFTM must be broken down for the implementation. For one-processor deployment targets the execution becomes sequential, or interleaved, the communication takes time *etc.* Thus, it is not obvious at all that properties proved to hold for a design model will still be true for the implementation. This concerns both functional and time behavior. In order to be able to reason about property preservation we need a formal model of the target platform of the system and a relation between the different time axes, described above. In Chapter 5 we describe the property-preserving deployment of systems modeled in AFTM to distributed *time-triggered* platforms, in particular, to a network of nodes connected by a time-triggered bus FlexRay[9] [Fle04] with a time-triggered OS OSEKtime[10] [OSE01b] running on each of them. We provide STS models of FlexRay and OSEKtime and a synthesis procedure, which correctly deploys AFTM models on that target platform. We prove the correctness in a simulation theorem. These results would not be possible for arbitrary application logic models from AutoFocus. AFTM was explicitly designed to support property-preserving deployment.

The summary of the thesis as well as the discussion of possible directions of future work are given in Chapter 6. Finally, Appendix A contains auxiliary definitions, propositions, and proofs used in ordinary Chapters 2 and 4.

1.4 Case Studies

For the illustration of the presented concepts and solutions we use two case studies: Car Periphery Supervision (CPS) System and Emergency Call (eCall) which are described in the following paragraphs.

Case Study 1: Car Periphery Supervision (CPS)

CPS stands for car periphery supervision system and is specified in [KR02]. The study exhibits some typical properties of targeted systems (in this case from the automotive domain), such as real-time requirements, distribution and resource constraints. The tasks enjoined on CPS are in parts safety-critical: the system has to react to critical situations, which may lead to a road accident (crash). By this, the correctness of the CPS system is a high-prioritized task.

CPS is supposed to be the basis for different driver assistance services, such as parking assistance, pre-crash detection, blind spot supervision, lane change assistant, *etc.* The CPS system is sending radar pulses out into the surrounding environment of the car and receives and processes the echoes which are reflected from objects in the environment (cf. Figure 1.2). For this purpose

[9] see http://www.flexray.com (20.06.2008)
[10] see http://www.osek-vdx.org (20.06.2008)

the system can operate up to six sensors at the same time. The sensors are located at different positions in the vehicle. The range of the sensors under consideration is 0 to 7 m. Depending on number, position and relative velocity of the objects, CPS communicates with the airbag system, the belt tensioner, and a human machine interface (HMI). A number of real-time requirements are stated for the CPS system. On the top-level black-box view these are:

(R1) CPS must send the collision data to the airbag ECU (electronic control unit), through the *collision object interface* (*COI*). The data must be delivered *at least* 10 *ms* before a predicted collision.

(R2) A further task of CPS is to activate the belt tensioner via the *belt tensioner interface* (*BTI*) in case of a predicted collision. The activation must take place *at least* 110 *ms* before the predicted time of collision.

These requirements bind the distance between an event issued by the system either to the airbag or to the belt tensioner and the *possible* collision.

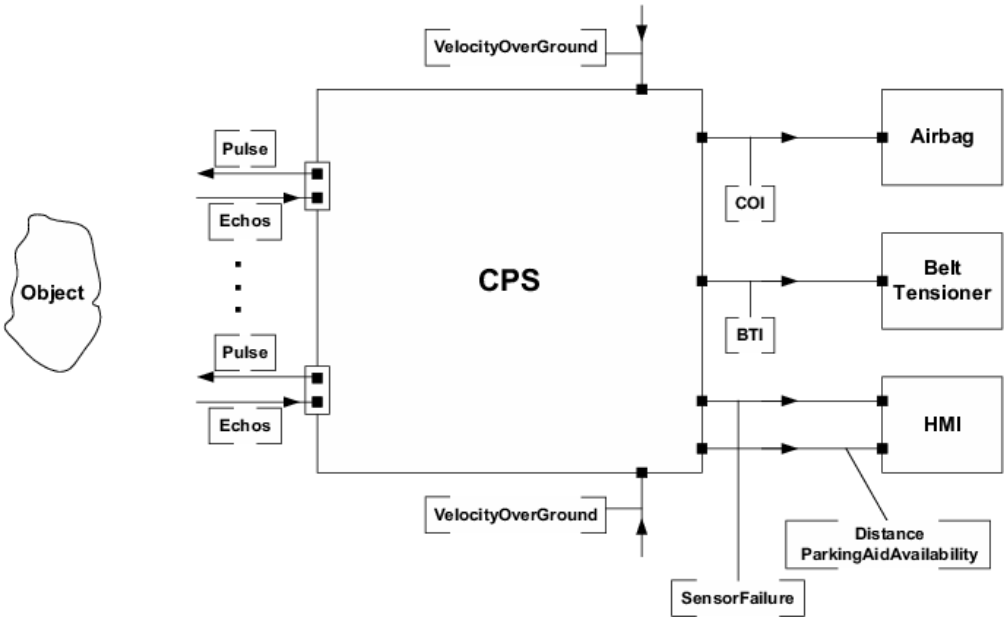


Figure 1.2: *Context of the CPS System from [KR02]*

Case Study 2: Emergency Call (eCall)

According to the proposal by the European Commission [Eur03], an automated emergency call should become mandatory in all new cars as of 2009. The application itself is simple enough to be sketched in a few paragraphs, but it still possesses typical properties of embedded software. By this we mean that it is a safety-critical application distributed over several electronic control units (ECUs), whose correct functionality not only depends on the correctness of the application itself but also on the correctness of a real-time OS and a real-time bus.

We model the eCall as a system consisting of 3 sub-systems, namely: a GPS navigation system, a mobile phone, and the actual emergency call application. External information (e.g. the crash sensor, the GPS signals) is considered to be a part of the environment. According to [Eur03], these components interact as follows: The navigation system sends periodically the vehicle's coordinates to the emergency call application so that it always possesses the latest coordinates. The crash sensor sends periodically the current crash status to the emergency call application. If a crash is detected, the emergency call application initiates the eCall by prompting the mobile phone to establish a connection to the emergency center. As soon as the mobile phone reports an open connection, the application transmits the coordinates to the mobile phone. After the coordinates have been successfully sent, the application orders the mobile phone to close the connection. The emergency call is finished as soon as the connection is successfully closed. If the radio link breaks down during the emergency call, the whole procedure is repeated from the initiation step.

2

Specification of Time Constraints

This chapter provides a formal denotational framework, which allows to specify the time behavior of reactive systems. The framework consists of a first-order quantitative linear temporal logic which is defined over the domain of timed traces. Using this logic time requirements put on the system under development can be written down formally. We use this framework to define a taxonomy of low-level time requirements – primitives often used to describe the time behavior on the level of technical systems. We also show how to compose individual requirements to specifications.

Contents

A SIGNIFICANT AMOUNT OF REACTIVE SYSTEMS has not only to realize the demanded functionality correctly but also to fulfill a number of real-time constraints. The system's reaction conforming to the functional specification, but produced at a wrong time (either too late or too early) can lead to harmful, undesirable results. The aggravating circumstance for a large class of reactive systems is the safety-critical character of their activities. In these settings the correct realization of both their functional and their real-time requirements is of a particular importance.

The real-time constraints are dictated by properties of the environment, which has to be monitored and/or controlled by reactive systems. These properties are firstly based on physical laws, and secondly on protocols and commitments between different control units. The systems have to note changes within the environment in time and react/act within fixed delays.

The time behavior of the environment, as a part of the problem space, is studied, negotiated and captured during the requirements engineering process. The outcomes, i.e., the time constraints, are documented in the system specification. These constraints, available at the early stages of the development process, are important criteria for the acceptance of the system and should be analyzed for correctness as well as used to guide the system design.

On the other hand, the analysis of the actual run-time behavior of embedded systems needs primarily tight estimations of *best case execution time* (*BCET*) and *worst case execution time* (*WCET*). These can be achieved only at the late development phases, when the machine-code of the application is deployed on the target platform. A discovered inconsistency within the demanded time behavior or a miss of a demanded time constraint during the final stages of the development process often lead to an expensive and time-intensive redesign.

This motivates the modeling of the *demanded* time behavior as an explicit view on the system already at the early phases, like specification or design. The justification of design for the fulfillment of time constraints aims at

- reducing the possibility of mismatch with the requirements,
- localizing the cost of final verification/validation,
- localizing/guiding the possible refactoring measures.

In order to achieve the listed goals, the time behavior model should be formally founded and support the notions of *composition* and *refinement*. These are preliminaries for the integration of time behavior with other aspects of the system, like the behavioral and architectural views.

Contributions and Outline. Mathematical preliminaries are covered by Section 2.1. In Section 2.2 we present a formal framework based on the domain of *timed traces*. Timed traces order the messages by the time of their occurrence. Certain occurrence and timing pattern constitute *hierarchical timed events*. The events are hierarchical in the sense that they are built by relating sub-events to each other. The formalism, we use to express relations between (sub-)events is a variant of *metric temporal logic* (*MTL*) [Koy90]. It allows to express causal relationships between events and to constrain their timing. Furthermore, we extend MTL by arbitrary predicates on the domain of timed traces in order to express more elaborate properties/events.

Using the introduced framework we define in Section 2.3 a taxonomy of *low-level* (or *system-level*) time requirements or *constraints*, well-established for the description of real-time behavior on the level of logical design and implementation. Members of this taxonomy like the response-time constraint or the period constraint are also commonly found in specifications. At the end of the section we compose time requirements to *timed behavior specifications* by embedding them into *contexts*. A context describes in which situations a requirement must hold. By this, it allows to isolate the time behavior from its functional embedding and offers a flexible and powerful mechanism for the reasoning about time requirements.

Finally, the CPS case study introduced in Section 1.4 illustrates the presented concepts throughout this chapter and, in particular, in Section 2.4.

2.1 Mathematical Preliminaries

In this section we give basic mathematical definitions that will be used as the foundations of our work. If f is a function then we denote by $\mathsf{dom}(f)$ and $\mathsf{rag}(f)$ the domain and range of f, respectively. We also use a more concise notation $\mathsf{dom}.f$ and $\mathsf{rag}.f$. Given a function f and a subset of its domain $D \subseteq \mathsf{dom}.f$, we denote the *restriction* of the domain of f to D by $f\lceil_D$ which is defined by the formula

$$(\mathsf{dom}.(f\lceil_D) = \mathsf{dom}.f \cap D) \wedge (\forall d \in \mathsf{dom}.(f\lceil_D) : (f\lceil_D).d = f.d).$$

We observe the commutativity of domain restriction: $f\lceil_{D_1}\lceil_{D_2} = f\lceil_{D_2}\lceil_{D_1} = f\lceil_{D_1 \cap D_2}$.

Intervals

Time is modeled by elements from a *time domain* $\mathbb{T}$. We expect $(\mathbb{T}, \leq)$ to be a linear ordered set (cf. Definition A.2). This linear order constitutes the *time-line* within the time domain. In our work, we are interested in infinite non-negative time domains with a least element, such as the discrete domain of naturals ($\mathbb{N}$) and the dense domains of non-negative rationals ($\mathbb{Q}_{\geq 0}$) and non-negative reals ($\mathbb{R}_{\geq 0}$).

We abbreviate $\mathbb{T} \cup \{\infty\}$ by $\mathbb{T}_\infty$ and $\mathbb{T} \setminus \{0\}$ by $\mathbb{T}_+$ and assume the usual properties of ∞, namely that for all $t \in \mathbb{T}$, $t < \infty$ and $t + \infty = \infty + t = \infty - t = \infty$. The *time intervals* over $\mathbb{T}$ are defined as conventional intervals, for example $[t_0, t_1) \stackrel{\text{def}}{=} \{t \in \mathbb{T} \mid t_0 \leq t < t_1\}$. Therefore "degenerate" intervals, like $[a, b]$ with $a > b$ or $[a, a)$ are considered as empty sets. By $[\mathbb{T}] \subseteq \wp(\mathbb{T})$ we denote the set of all right- and/or left-closed/open intervals in the time domain $\mathbb{T}$. We assume for that the numbering systems of interval bounds match with the time domain of the interval they build; e.g., for $t_0, t_1 \in \mathbb{R}_{\geq 0}$, $(t_0, t_1] \in [\mathbb{R}_{\geq 0}]$ and for $t_0, t_1 \in \mathbb{N}$, $[t_0, t_1] \in [\mathbb{N}]$.

The arithmetic operations $+$ and $-$ on intervals are defined by their point-wise application and yield intervals again. Formally, for $\mathbb{I}_1, \mathbb{I}_2 \in [\mathbb{T}_\infty]$ we define binary operations $+ \colon [\mathbb{T}_\infty] \times [\mathbb{T}_\infty] \mapsto [\mathbb{T}_\infty]$ and $- \colon [\mathbb{T}_\infty] \times [\mathbb{T}_\infty] \mapsto [\mathbb{T}_\infty]$ in infix notation as

$$\mathbb{I}_1 + \mathbb{I}_2 \stackrel{\text{def}}{=} \{t_1 + t_2 \mid t_1 \in \mathbb{I}_1 \wedge t_2 \in \mathbb{I}_2\} \qquad \text{and} \qquad \mathbb{I}_1 - \mathbb{I}_2 \stackrel{\text{def}}{=} \{t \mid \exists t_2 \in \mathbb{I}_2 : t_2 + t \in \mathbb{I}_1\}.$$

For $t \in \mathbb{T}$ and $\mathbb{I} \in [\mathbb{T}_\infty]$ we abbreviate $[t,t]+\mathbb{I}$ by $t+\mathbb{I}$ and $[t,t]-\mathbb{I}$, $\mathbb{I}-[t,t]$ by $t-\mathbb{I}$, $\mathbb{I}-t$, respectively. By this, $t+\mathbb{I}$ and $\mathbb{I}-t$ are point-wise shifts of $\mathbb{I}$ by t, and $t-\mathbb{I}$ is a reflection of $\mathbb{I}$ around t. Further on, we define $t \sim \mathbb{I}$ for $\sim \in \{<, \leq, =, \geq, >\}$, $t \in \mathbb{T}$, and $\mathbb{I} \in [\mathbb{T}_\infty]$ as the conjunction of point-wise comparisons, i.e., $t \sim \mathbb{I} \overset{\text{def}}{\Leftrightarrow} \forall t' \in \mathbb{I} : t \sim t'$. By this, in particular, $t = [t,t]$. A comparison with an empty interval yields always true; and the above arithmetical operations applied on an empty set return an empty set again. Finally, the *length* of an interval $\mathbb{I} \in [\mathbb{T}_\infty]$ is denoted by $\#\mathbb{I}$.

Streams

The following definitions of streams and operations on streams are taken from [BS01a]. Let M be an arbitrary not empty set of messages. A *stream* over M is a finite or infinite sequence of elements from M. M^*/M^∞ denotes the set of finite/infinite streams over M, respectively. The set of finite streams also includes an empty stream, that we write as $\langle\rangle$. A non-empty finite stream of length n is denoted by $\langle m_1 \ldots m_n \rangle \in M^*$, an infinite stream is denoted by $\langle m_1 \ldots \rangle \in M^\infty$. The set of all streams is denoted by $M^\omega = M^* \cup M^\infty$.

***Remark* 2.1** (Notational conventions)**.** In the present thesis we borrow the convention from, e.g., [BS01a] and indicate infinite streams by superscript "∞", and the union of finite and infinite streams by ω-superscript. Most works from the formal verification community (e.g., cf. [GTTW02]) use the opposite notation: infinite streams are denoted with ω, the union of finite and infinite streams with ∞. ◦

For a stream $\sigma \in M^\omega$, $\#\sigma$ stands for its length. Further on, $\sigma.n$ with $n \in \mathbb{N}_+$ is the nth element of σ. $\sigma.n$ is a member of M.

The *concatenation* of streams is defined by a binary operator $\frown$: $M^\omega \times M^\omega \mapsto M^\omega$ in infix notation. The concatenation of two stream produces a stream, which contains the messages of the first operand at the beginning followed by the messages of the second operand. Formally, for all $\sigma_1, \sigma_2 \in M^\omega$ and $k \in \mathbb{N}_+$

$$(\sigma_1 \frown \sigma_2).k \overset{\text{def}}{=} \begin{cases} \sigma_1.k & \text{if } 1 \leq k \leq \#\sigma_1, \\ \sigma_2.(k - \#\sigma_1) & \text{if } \#\sigma_1 < k \leq \#\sigma_1 + \#\sigma_2. \end{cases}$$

In accordance with the above definition if $\#\sigma_1 = \infty$ then $\sigma_1 \frown \sigma_2 = \sigma_1$. The length of $\sigma_1 \frown \sigma_2$ is $\#(\sigma_1 \frown \sigma_2) = \#\sigma_1 + \#\sigma_2$.

2.2 Specification of Reactive Real-Time Systems

The notion of streams from the previous section permits to capture the order of message occurrence but lacks of timing information, i.e., no information is available *when* a certain message was observed. Thus, within a stream-based system model it is possible to reason about the causality of messages but impossible to specify or check the timeliness of their occurrence. This motivated us to enrich the stream-based FOCUS formalism from [BS01a] by timed traces

which are presented in Section 2.2.1. Section 2.2.2 contains a number of useful operations on timed traces which are intensely used in the remainder of the current chapter as well as in Chapters 3 and 4. The trace-based semantic domain of reactive real-time systems is presented in Section 2.2.3. Based on this semantic domain composite timed events and a qualitative temporal logic are defined in Sections 2.2.4 and 2.2.5, respectively. By this, reactive real-time systems can be specified by imposing timing and causal constraints on timed events.

2.2.1 Timed Events and Timed Traces

Besides the order in which the messages appear, the time of their occurrence is relevant for the real-time systems. Thus, we define the domain of *timed events* $\mathcal{A} \stackrel{\text{def}}{=} M^* \times \mathbb{T}$. An event $(\langle m_1 \ldots m_n \rangle, t) \in \mathcal{A}$ consists of a finite sequence of events from M, which occurs in time $t \in \mathbb{T}$.

For a pair of timed events $(\sigma_1, t_1), (\sigma_2, t_2) \in \mathcal{A}$ we define the equality and precedence by

$$\begin{aligned} (\sigma_1, t_1) &= (\sigma_2, t_2) \qquad \text{iff } \sigma_1 = \sigma_2 \text{ and } t_1 = t_2, \\ (\sigma_1, t_1) &< (\sigma_2, t_2) \qquad \text{iff } t_1 < t_2, \end{aligned}$$

respectively. By this, the $\leq$-relation defined as

$$(\sigma_1, t_1) \leq (\sigma_2, t_2) \stackrel{\text{def}}{\Leftrightarrow} (\sigma_1, t_1) < (\sigma_2, t_2) \vee (\sigma_1, t_1) = (\sigma_2, t_2)$$

becomes a partial order on $\mathcal{A}$ according to Definition A.1, i.e., it is reflexive, transitive, and antisymmetric.

In order to access one of both components which build the tuple of a timed event, we introduce projections $\lambda \colon \mathcal{A} \mapsto M^*$ and $\phi \colon \mathcal{A} \mapsto \mathbb{T}$ defined as

$$\lambda((\sigma, t)) \stackrel{\text{def}}{=} \sigma \qquad \text{and} \qquad \phi((\sigma, t)) \stackrel{\text{def}}{=} t,$$

respectively.

For the time components of sets of timed events we identify extremal points by $\phi^{glb}, \phi^{lub} \colon \wp(\mathcal{A}) \mapsto \mathbb{T}_\infty$. Intuitively, $\phi^{glb}(A)/\phi^{lub}(A)$ return points in time of the earliest/latest observation in $A \subseteq \mathcal{A}$, respectively. Formally, we define

$$\begin{aligned} \phi^{glb}(A) &\stackrel{\text{def}}{=} \begin{cases} \text{glb}\{\phi(a) \mid a \in A\} & \text{if } A \neq \emptyset, \\ \infty, & \text{o.w.}, \end{cases} \\ \phi^{lub}(A) &\stackrel{\text{def}}{=} \begin{cases} \text{lub}\{\phi(a) \mid a \in A\} & \text{if } A \neq \emptyset, \\ 0, & \text{o.w.}, \end{cases} \end{aligned}$$

where glb/lub stand for greatest lower/least upper bound, and are defined in A.3/A.4, respectively. We say that the interval $[\phi^{glb}(A), \phi^{lub}(A)]$ is the *observation frame*, or *time window* of A. By the above definitions this interval is empty if and only if A is empty too: $[\infty, 0] = \emptyset$. Otherwise, it contours the timing distribution of observed events. We also lift the definition of the ϕ-function for sets of timed events by defining

$$\phi(A) \stackrel{\text{def}}{=} \{\phi(a) \mid a \in A\}.$$

We can now proceed to the definition of *timed traces* also called *timed observations*. Intuitively, a set of timed events is called a timed trace if the observation it contains is complete and unique. For instance, in the time domain of naturals the set $\{(\sigma_1,2),(\sigma_2,3),(\sigma_3,4)\}$ is a timed trace, but neither the incomplete set of observations $\{(\sigma_1,2),(\sigma_3,4)\}$ nor the not-unique set of observations $\{(\sigma_1,2),(\sigma_2,3),(\sigma'_2,3),(\sigma_3,4)\}$ comprise timed traces. This intuition is formally captured by the following definition.

Definition 2.1 (Observation/timed trace)**.** *A* timed trace *or* observation *is a set $\tau \subseteq \mathcal{A}$, for which the following two properties hold*

(1) Order completeness: $\forall a_1, a_2 \in \tau : a_1 \leq a_2 \vee a_2 \leq a_1$
(2) Observation completeness: $\phi(\tau) \in [\mathbb{T}]$, i.e., the observation frame of τ is an interval in $\mathbb{T}$. ◇

The first property states that the order of events in a trace is total. By this, we exclude event sets which contain different messages observed at the same time. The second property demands the trace to be complete within its observation frame. It is mainly motivated by the fact, that observing the *absence* of events can also be a crucial information for a real-time system. An observation $\{(\langle\rangle,t_1),(\langle m\rangle,t_2),(\langle\rangle,t_3)\}$ contains more information than $\{(\langle m\rangle,t_2)\}$. Another reason of a more technical nature is that it allows to subdivide or filter traces without loss of information about their observation frames. For the ϕ-function applied on a timed trace τ holds $\#\phi(\tau) = \#\tau$. Moreover, $(\phi^{glb}(\tau),\phi^{lub}(\tau)) \subseteq \phi(\tau) \subseteq [\phi^{glb}(\tau),\phi^{lub}(\tau)]$. If, e.g., $\phi^{glb}(\tau) \in \phi(\tau)$ and $\phi^{lub}(\tau) \notin \phi(\tau)$, we obtain $\phi(\tau) = [\phi^{glb}(\tau),\phi^{lub}(\tau))$.

***Remark* 2.2** (From timed events to timed traces)**.** Provided some $A \in \wp(\mathcal{A})$ fulfills the first property, in order to fulfill the second one, it can be complemented by events of the form $(\langle\rangle,t)$ for all points in time t, for which no events in A exist. For example, a set of timed events $\{(\sigma_1,1),(\sigma_3,3),(\sigma_4,4)\}$ defined over the time domain $\mathbb{N}$ can be complemented to a timed trace by adding the timed event $(\langle\rangle,2)$. ◦

***Remark* 2.3** (Timed traces vs. timed words)**.** The notation of "classical" timed words, as expected, for example, by timed automata [AD94], is more concise. It includes non-empty events only, assuming that at the other points in time (either outside, or inside the observation frame), nothing has happened. We can easily switch from timed traces to timed words, by filtering out all empty events. The transition in the opposite direction is not trivial[1]. By this, model checking approaches for timed automata [Yov98] can be also applied to specifications based on timed traces. ◦

The continuous progress is an essential property of physical time. On the other hand, in the domains with the dense order, like $\mathbb{R}_{\geq 0}$, one can specify a timed trace with infinitely many messages observed within a finite time interval. Such time behavior is called Zeno's paradox. A comparable behavior within a discrete time domain, like $\mathbb{N}$, is an observation of an infinite sequence of messages at some point in time: $(\langle m_1 m_2 \ldots\rangle,t)$. Such an observation was excluded

[1] One could imagine some special events serving as bonding points, without any further semantic meaning, but this would complicate the formalism.

within our settings by the definition of the domain $\mathcal{A}$. In order to exclude it also in the dense case, we provide the following definition.

Definition 2.2 (Non-Zeno timed trace/observation)**.** *An observation according to Definition 2.1 τ over the time domain $\mathbb{T}$ is called* non-Zeno *if and only if for any $t \in \mathbb{T}$ the set*

$$\{a \in \tau \mid \phi(a) < t \wedge \lambda(a) \neq \langle\rangle\} \quad \textit{is finite.} \qquad \diamond$$

By this definition we allow only finitely many non-empty events to happen within any finite interval of time. This is automatically guaranteed if there exists a minimal resolution δ of timed observations, stating that for any pair of non-empty events a_1, a_2 must hold $|\phi(a_1) - \phi(a_2)| > \delta$.

***Remark* 2.4** (Another definition of non-Zenoness)**.** An alternative definition of non-Zenoness which is more appropriate for hybrid systems is given by the *finite variability* condition. This condition demands that the message values change only finitely often in finite intervals. In other words, every finite interval can be completely segmented into finitely many disjoint sub-intervals which are mapped to the same message in each case. Given a trace which is non-Zeno according to Definition 2.2, we can transform it to a non-Zeno trace according to this alternative definition by remapping every point in time in which an empty message is observed to the right-most non-empty message which has happened earlier. If there is no such message, the time point remains mapped to the empty message. The translation in the opposite direction is only possible if we demand that the above partition into sub-intervals yields solely intervals which are left-closed (and, consequently, right-open). $\circ$

The set of all non-Zeno timed traces is denoted by $TiTr \subseteq \wp(\mathcal{A})$. The set of *full* non-Zeno timed observations is denoted by $TiTr^{\kappa}$ and contains all observations, which cover the whole time domain. Formally:

$$\forall \tau \in TiTr : \tau \in TiTr^{\kappa} \overset{\text{def}}{\Leftrightarrow} \phi(\tau) = \mathbb{T}.$$

We can easily show that every single timed event from $\mathcal{A}$ builds a non-Zeno timed trace, i.e., $\mathcal{A} \subseteq TiTr$.

In the rest of this thesis we will consider non-Zeno traces only and thus omit the term "non-Zeno" in the subsequent discussion. In the next section we will lift operations on streams from [BS01a] and Section 2.1 for timed traces.

CPS Case Study (*con'ed*). The CPS system is designed to recognize events which may happen in the future and to react to them within sufficient time budgets. The environment of the system consists of a number of moving objects. Every object at a given measurement time is characterized by its relative velocity v, angle to the x-axis $\theta \in [0, 90]$, and distance to the vehicle d (cf. Figure 2.1). We model this information as a content of a message $m_{obj} \overset{\text{def}}{=} (v, \theta, d)$. Distributed over the real-valued time axis, this information builds a timed observation from the set $TiTr_{obj} \subseteq \wp(M_{obj}^{*} \times \mathbb{R}_{\geq 0})$, where M_{obj} denotes the set of object messages m_{obj}. In accordance with the timed trace semantics, one point in time contains a finite sequence of m_{obj}-messages. This reflects the fact that several objects can be observed at the same time.

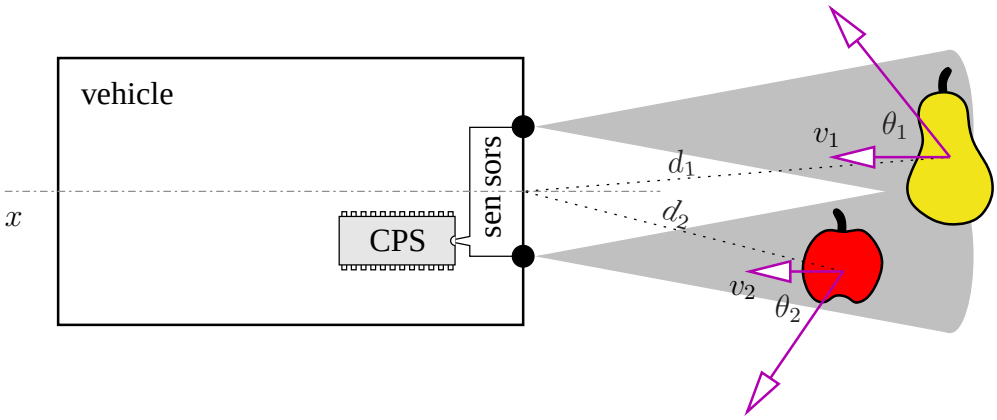


Figure 2.1: *An Illustration of the CPS System in Action*

2.2.2 Operations on Timed Traces

We can identify timed traces from *TiTr* with total functions with a subset of $\mathbb{T}$ as their domains: $\cup_{i\in\mathbb{T}}([0,i] \mapsto M^*) \cup (\mathbb{T} \mapsto M^*)$. They are functions due to the property one and total due to the property two of Definition 2.1. This allows us to identify the event, which has occurred at $t \in \mathbb{T}$ in a trace $\tau \in TiTr$, by a function application $\tau(t)$. To improve the readability of formulas we often write $\tau.t$ instead of $\tau(t)$. $\tau.t$ is defined only if $t \in \phi(\tau)$. The length of τ is denoted by $\#\tau$. It is ∞ for infinite traces.

A timed trace τ' is a *subtrace* of $\tau \in TiTr$, written as $\tau' \stackrel{\text{def}}{=} \tau|_{\mathbb{I}}$ for some interval $\mathbb{I} \in [\mathbb{T}_\infty]$ if and only if

$$\phi(\tau') = (\phi^{glb}(\tau) + \mathbb{I}) \cap \phi(\tau) \qquad \text{and} \qquad \forall t \in \phi(\tau') : \tau'.t = \tau.t.$$

If the relative interval $\mathbb{I}$ lies outside the observation frame of τ (i.e., $\phi(\tau') = \emptyset$), we obtain $\tau|_{\mathbb{I}} = \emptyset$ and by this the empty set is the subtrace of any trace. We also note that for all $n \geq \#\tau$, $\tau|_{[0,n]} = \tau$. Special cases are a *suffix* and a *prefix* of length n ($n \in \mathbb{T}_\infty$) of a timed trace, defined as $\tau\uparrow_n \stackrel{\text{def}}{=} \tau|_{[n,\infty)}$ and $\tau\downarrow_n \stackrel{\text{def}}{=} \tau|_{[0,n]}$, respectively. We also define the *weak* forms of prefix and suffix operators as $\tau\downarrow_{\sqcap n} \stackrel{\text{def}}{=} \tau|_{[0,n)}$ and $\tau\uparrow_{\sqcup n} \stackrel{\text{def}}{=} \tau|_{(n,\infty)}$, respectively.

Besides accessing messages observed at a given point in time, we would also like to obtain the nth non-empty message and to count the overall number of non-empty messages in a trace. This makes especially sense for non-Zeno traces, since there, at every arbitrary but fixed $t \in \mathbb{T}$ only finitely many non-empty events can be observed. For these purposes for all $\tau \in TiTr$, $a \in \mathcal{A}$, and $i \in \mathbb{N}$ we define the functions $len\colon TiTr \mapsto \mathbb{N}_\infty$ and $nth\colon TiTr \times \mathbb{N} \mapsto \mathcal{A}$ as

$$len(\tau) \stackrel{\text{def}}{=} |\tau \setminus (\{\langle\rangle\} \times \mathbb{T})|,$$
$$nth(\tau, i) = a \stackrel{\text{def}}{\Leftrightarrow} a \in \tau \wedge \lambda(a) \neq \langle\rangle \wedge len(\tau\downarrow_{\phi(a)-\phi^{glb}(\tau)}) = i.$$

$nth(\tau, i)$ is only defined for $i \leq len(\tau)$. The parameter of the prefix operator in nth's definition yields a prefix which ends exactly at time $\phi(a)$. In that prefix the number of non-empty messages must be i.

The concatenation of timed traces $\tau_1 \frown \tau_2$ is defined if and only if the time line of the second trace continues the time line of the first one. Formally:

$$\tau_1 \frown \tau_2 \stackrel{\text{def}}{=} \begin{cases} \tau_1 \cup \tau_2 & \text{if } (\phi(\tau_1) \cup \phi(\tau_2)) \in [\mathbb{T}_\infty] \text{ and } \phi(\tau_1) \cap \phi(\tau_2) = \emptyset, \\ \emptyset, & \text{o.w.} \end{cases}$$

It is easy to see that $(\tau\downarrow_n)\frown(\tau\uparrow_{\sqcup n}) = (\tau\downarrow_{\sqcap n})\frown(\tau\uparrow_n) = \tau$.

Notation	Informal Meaning
M^*	set of finite streams over message set M
M^∞	set of infinite streams over message set M
M^ω	set of streams over message set M
$\langle\rangle$	empty stream
$\langle m_1 \dots m_n \rangle$	finite stream consisting, in this order, of messages m_1 through m_n
$\langle m_1 \dots \rangle$	infinite stream
$\sigma.n$	nth element of stream σ
$\#\sigma$	length of stream σ
$\sigma_1 \frown \sigma_2$	concatenation of streams σ_1 and σ_2
$\mathbb{T}$	time domain (typically, $\mathbb{N}$, $\mathbb{Q}_{\geq 0}$, or $\mathbb{R}_{\geq 0}$)
$[\mathbb{T}]$	interval set in time domain $\mathbb{T}$
$\mathcal{A}$	domain of timed events ($M^* \times \mathbb{T}$)
$TiTr$	domain of non-Zeno timed traces/observations ($TiTr \subseteq \wp(\mathcal{A})$)
$TiTr^\kappa$	domain of full non-Zeno timed traces/observations ($TiTr^\kappa \subseteq TiTr$)
$a_1 \leq a_2$	timed event a_2 was observed not earlier than timed event a_1
$\lambda(a)$	contents (a finite message stream) of timed event a
$\phi(a)$	time stamp of timed event a
$\phi(\tau)$	observation frame of timed trace τ
$\phi^{glb}(\tau)/\phi^{lub}(\tau)$	earliest/latest observation time in timed trace τ
$\tau.n$	finite message stream observed at time n in timed trace τ
$\#\tau$	length of timed trace τ
$\tau\vert_{\mathbb{I}}$	subtrace of τ – events observed in τ in time interval $\phi^{glb}(\tau) + \mathbb{I}$
$\tau\downarrow_n/\tau\downarrow_{\sqcap n}$	prefix of τ which ends at/immediately before time $\phi^{glb}(\tau) + n$
$\tau\uparrow_n/\tau\uparrow_{\sqcup n}$	suffix of τ which starts at/immediately after time $\phi^{glb}(\tau) + n$
$len(\tau)$	amount of non-empty messages in timed trace τ
$nth(\tau, i)$	ith non-empty message in timed trace τ
$\tau_1 \frown \tau_2$	concatenation of timed traces τ_1 and τ_2

Table 2.1: *Streams & Timed Traces Notation*

Table 2.1 serves as a quick reference for the sets and operations defined for streams and timed traces in Sections 2.1, 2.2.1, and 2.2.2.

2.2.3 Semantic Domain of Timed Systems

According to the FOCUS framework [BS01a], the system is a set of components connected to each other and to environment by directed and typed channels from the set C. Every channel $c \in C$ has an assigned *type* $ty(c) \in \wp(M^*)$ of messages it transports and a *direction*, in which these messages flow. The channel type is described by the set of all messages which can be sent through the channel. The interaction between components takes place solely through their channels. The direction of a channel designates the sender and the receiver component. A component can be specified either by a network of further components or by a total relation[2] from input to output communication histories. In this context we speak about a *composite* and *elementary* component, respectively. The framework is compositional, i.e., the behavior of a composite component is completely described by the behaviors of its constituting subcomponents combined by a uniform composition operator. We will learn more of these concepts the next chapter in Section 3.2. In the present section we introduce the *semantic domain* of timed systems on which all contributions of this thesis are based. The integral part of that semantic domain is the *timed channel valuation* which is defined next.

Valuations of channels from C by messages from the set M observed using the time domain $\mathbb{T}$ are mappings $C \mapsto TiTr$. For $t \in \mathbb{T}$, channel $c \in C$, and a valuation $\tau\colon C \mapsto TiTr$ (with $TiTr \subseteq \wp(M^* \times \mathbb{T})$), the observation at a given point in time $t \in \mathbb{T}$ is $\tau.t \in \{C \mapsto M^*\}$ and the observation on a particular channel $c \in C$ is $\tau.c \in TiTr$. The observed message sequence on c at t is $\tau.t.c \in M^*$ or, equally, $\tau.c.t \in M^*$.

For further reasoning about reactive real-time systems we need special kinds of channel valuations. These are introduced by the following definition.

Definition 2.3 (Timed channel valuations)**.** *For a channel set C, a set of messages M, and a time domain $\mathbb{T}$, the channel valuation $\tau\colon C \mapsto TiTr$ is called* type-correct *if and only if*

$$\forall c \in C : \forall t \in \phi(\tau.c) : \tau.c.t \in ty(c).$$

τ is called simultaneous *if and only if*

$$\forall c_1, c_2 \in C : \phi(\tau.c_1) = \phi(\tau.c_2).$$

The set of all simultaneous type-correct channel valuations is denoted by $\vec{C} \subseteq (C \mapsto TiTr)$. The set of all full *type-correct channel valuations is denoted by $\vec{C}^\kappa \subseteq (C \mapsto TiTr^\kappa)$. The set of all simultaneous type-correct mappings of channels to singular events from $\mathcal{A}$ is denoted by $\vec{C}^{\mathcal{A}} \subseteq (C \mapsto \mathcal{A})$.* ◇

We observe that by the definition of $TiTr^\kappa$, $\vec{C}^\kappa$ contains simultaneous valuations only and thus, $\vec{C}^\kappa \subseteq \vec{C}$. Moreover, since timed events are a special case of timed traces, i.e., $\mathcal{A} \subseteq TiTr$, the

[2] The relation is total in the sense that it is defined for every input.

definitions of type-correctness and simultaneity are also applicable to $\vec{C}^{\mathcal{A}}$. Simultaneous singular events $a \in \vec{C}^{\mathcal{A}}$ map channels to events which have happened at the same time. Finally, given a subset of C, $D \subseteq C$, by applying the restriction operator to the domain of τ we obtain the valuations of channels from D only, i.e., $\tau\lceil_D \in \vec{D}$.

The time window of a simultaneous valuation τ from $\vec{C}$ is denoted by $\phi(\tau)$ and defined as $\phi(\tau.c)$ of any $c \in C$. Other operations over timed traces introduced in Section 2.2.2 are lifted for simultaneous channel valuations in a straight-forward way by applying them channel-wise (cf. Definition A.5). In the rest of this thesis we will consider simultaneous and type-correct observations only and thus omit the terms "simultaneous" and "type-correct" in the following discussion.

Reactive systems, which are the main concern of presented work, are designed to run infinitely long. Thus, we describe the semantics of a component (or system) with the channel set C over the messages from M and time domain $\mathbb{T}$ by elements from $\wp(\vec{C}^{\kappa})$. The existence of more than one element in the semantic domain for one system indicates its non-determinism.

***Remark* 2.5** (Discrete time vs. dense time)**.** According to the above definition, the semantic domain of reactive systems is parametrized by the time domain. In Section 1.2 we have discussed domains different development phases are usually operating with. However, the problem of choosing the proper time domain has additional aspects: The analytical models of the system on different development stages become easier to verify if their underlying time domain is isomorphic with a set of natural numbers. This reduces the state space of these models a makes the verification techniques developed for untimed systems applicable. Concrete, for our formalism of choice a discrete time domain reduces the concepts presented here to a special variant, called "timed" FOCUS (cf. [BS01a]). The question about the adequacy of discrete-time models for real-time systems and the significance of their analysis results is often answered by different researchers with the argument that discrete-time models are good enough for all practical purposes, provided the unit of time is small enough. Justification for the use of dense-time models for the specification and analysis of real-time systems is given in Chapter 2 of [Alu92]. The main argument is that, so far, there exists no method to figure out the appropriate level of granularity (e.g., to figure out the time unit, which is "small enough"). The author of [Alu92] argues that a method, which finds the appropriate granularity level, should be at least as complex as the direct analysis within the dense-time domain. Another important argument against the discretization is that the granularity level depends on *all* time constraints put on the system. This violates the compositional reasoning about the system: in discrete-time domain a component has to be specified in a way, which respects the speed of other possibly independent parts of the system. Thus, we retain the possibility to choose a dense-time domain, as the central feature of our model. ∘

Table 2.2 summarizes sets and operations for timed channel valuations defined in the current section.

CPS Case Study (*con'ed*). As shown in Figure 1.2, page 10, the CPS obtains object information using $2n$ channels (n input and n output ones), where $n \in \mathbb{N}_+$ is the number of sensors. Additionally, the system sends data to airbag and belt tensioner. In order to keep our CPS model simple, we fade out communication links to the human-machine interface: this part of the system

Notation	Informal Meaning
$\vec{C}$	domain of simultaneous type-correct timed channel valuations ($C \mapsto TiTr$)
$\vec{C}^{\kappa}$	domain of full type-correct timed channel valuations ($\vec{C}^{\kappa} \subseteq \vec{C}$)
$\vec{C}^{\mathcal{A}}$	domain of simultaneous type-correct timed event valuations ($\vec{C} \mapsto \mathcal{A}$)
$\wp(\vec{C}^{\kappa})$	domain of reactive systems
$\tau.c$	timed trace mapped to channel c ($c \in C$ and $\tau \in \vec{C}$)
$\tau.t$	observation at time t ($t \in \mathbb{T}$)
$\tau.c.t/\tau.t.c$	observation (finite message stream) at time t on channel c ($t \in \mathbb{T}$, $c \in C$, $\tau \in \vec{C}$)
All operations on timed traces from Table 2.1 (page 21) apply to timed channel valuations (cf. Definition A.5).	

Table 2.2: *Timed Channel Valuations Notation*

functionality can be considered as non-safety-critical and imposes no hard real-time constraints. By this, the channel set of the CPS system consists of $2n+2$ channels:

$$C_{\mathsf{CPS}} \stackrel{\text{def}}{=} \{c^{(1)}_{\mathsf{Pulse}}, \ldots, c^{(n)}_{\mathsf{Pulse}}, c^{(1)}_{\mathsf{Echos}}, \ldots, c^{(n)}_{\mathsf{Echos}}, c_{\mathsf{BTI}}, c_{\mathsf{COI}}\},$$

where $c^{(i)}_{\mathsf{Pulse}}$ and $c^{(i)}_{\mathsf{Echos}}$ stand for request and response channels to the vehicle's environment, respectively, and c_{COI} and c_{BTI} – for communication links to the airbag and belt tensioner, respectively.

We model all channels except of environment echos as *signals*. A signal is either present (denoted by $\langle\top\rangle$) or not (denoted by an empty sequence $\langle\rangle$). This representation is sufficient to describe time requirements of the CPS system. More formally, the channel types are

$$\begin{aligned}
ty(c^{(i)}_{\mathsf{Pulse}}) &\stackrel{\text{def}}{=} \{\langle\rangle, \langle\top\rangle\} && \text{for all } i \in [1,n],\\
ty(c^{(i)}_{\mathsf{Echos}}) &\stackrel{\text{def}}{=} M^{*}_{obj} && \text{for all } i \in [1,n],\\
ty(c_{\mathsf{BTI}}) &\stackrel{\text{def}}{=} \{\langle\rangle, \langle\top\rangle\},\\
ty(c_{\mathsf{COI}}) &\stackrel{\text{def}}{=} \{\langle\rangle, \langle\top\rangle\}.
\end{aligned}$$

2.2.4 Composite Timed Events and Observations

In the previous section we have defined the system behavior as a set of observations. An observation consists of a sequence of timed events. However, apart from singular events specifications often refer to *composite observations*, like the arrival of certain combination of different signals on different channels within a certain time window. In our specification framework we model composite observations as time intervals in which certain events are observed. Further on, specifications relate composite observations to each other. They demand, for example, that the observation e_A must always be followed by the observation e_B within a fixed delay d. In terms of our specification framework this means that for any trace from the set, which constitutes the

system behavior, must hold: If it contains the observation e_A as a subtrace then it also contains the observation e_B after e_A not earlier than d. In general, there exist thirteen possible relation types (or alignments) between a pair of intervals (cf. [All83]); however, in the present work we are interested in event-based specification. Thus, we generalize the notion of event as a point in time, where a certain composite observation was made within a certain timed trace. By relating these events to each other, hierarchical composite events and observations emerge. In the rest of this section we formalize these ideas. However, let us first illustrate the above considerations by the following small example.

Example 2.1. Consider the triggering condition of an airbag which, roughly speaking, consists of three parts [MA95]:

(1) A frontal collision has occurred.

(2) The angle of impact is within a "window" typically around thirty degrees either side of the vehicle's center line.

(3) The deceleration forces produced are at least equal to those produced when the car collides head-on with an immovable barrier at approximately 25 km/h.

This information is brought together by several sensors and ECUs (electronic control units) which monitor the environment of the vehicle and produce corresponding events. Thus, the fulfillment of above conditions is naturally modeled by an observation of corresponding events on a number of channels (e.g., one channel per sensor or ECU) within a fixed time window. This composite observation can be represented by a composite airbag-trigger-condition event which happens at the same time with the observation of the last constituting sub-event. Consequently, this event should be followed within a certain time interval by an airbag-firing event which can be in turn related to further events. For instance, the distance between the actual collision time and the firing time is crucial to the overall correctness of the airbag system and must be also constrained by the airbag specification. ∘

We model (composite) timed observations by unary predicates $e_1, e_2, \ldots$ over the channel set valuations called *timed observation predicates*. Their domain is the set of mappings to timed traces: $e_i \in \wp(\vec{C})$. The singular observation or timed trace $\tau \in \vec{C}$ as introduced in Definition 2.1 and generalized for channel valuations by Definition 2.3 is a special case of a composite observation predicate: $e = \{\tau\}$. Also, the singular events can be considered as observation predicates, i.e., $a \in \vec{C}^{\mathcal{A}} \subseteq \vec{C}$ is a unary predicate and a mapping to timed traces at the same time. Albeit of these special cases, the main purpose of composite observation predicates is to describe valid periods of system's life.

As discussed above, specifications relate different composite observations to each other. In order to model these relations in our specification framework, we need to answer the following question: Given a finite or infinite timed observation $\tau \in \vec{C}$ and a timed observation predicate $e \subseteq \vec{C}$, *when* (i.e., at what points in time $t_1, t_2, \ldots \in \mathbb{T}$) was e observed in τ? We provide two different modalities on observations to answer this question: $e^{\exists\downarrow}$ and $e^{\exists\uparrow}$. Their formal definitions are:

$$e^{\exists\downarrow} \stackrel{\text{def}}{=} \{(\tau, t) \mid \exists u \in \mathbb{T} : \tau|_{[u,t]} \in e\} \qquad \text{and} \qquad e^{\exists\uparrow} \stackrel{\text{def}}{=} \{(\tau, t) \mid \exists u \in \mathbb{T}_\infty : \tau|_{[t,u)} \in e\},$$

respectively. The $.^{\exists\downarrow}$-interpretation allows to express the fact, that *e was* observed in τ at time t. It is feasible for the formal specification of real-time systems because most of them cannot predict the future, and thus cannot judge if an observation has happened or not before they observe it completely. In our airbag example, the airbag-trigger-condition observation has happened when all its constituting sub-events are observed. The complete observation of infinite traces is, of course, impossible, thus we interpret them as invariants, which always hold along some execution paths. For invariants the starting point is crucial and this is expressed by the second interpretation ($.^{\exists\uparrow}$). For the airbag model it makes sense to assume that if the airbag has deployed, its controller will not change its state anymore; however, the point in time when the airbag was observed in the deployed state for the first time is crucial for the correctness of the airbag system; in fact, the timeliness of the deployment is the matter of life and death for the person sitting in front of it [MA95]. The above modalities constitute the generalized notion of a composite timed event or *timed property A* as timed-trace/point-in-time-pairs, formally, $A \subseteq \vec{C} \times \mathbb{T}$.

Singular events can be satisfied by a trace and a point in time only if they have happened exactly at that time, i.e., for all $A \subseteq \vec{C}^{\mathcal{A}}$ we obtain $A^{\exists\downarrow} = A^{\exists\uparrow} = \{(\tau, \phi(a)) \mid a \in A \wedge \phi(a) \in \phi(\tau)\}$, i.e., e can directly speak about satisfaction of A by some trace τ at $t \in \phi(\tau)$. We can also conclude that if A is satisfied by some (τ, t) then so does its subset $\{a \in A \mid \phi(a) = t\}$ with mappings to events happened at time t.

Besides the both mappings from observations to event presented above, we also need to reason about the whole prefixes or suffixes of the system behavior. For these purposes we define a further pair of modalities:

$$e^{\downarrow} \stackrel{\text{def}}{=} \{(\tau, t) \mid \tau\downarrow_t \in e\} \qquad \text{and} \qquad e^{\uparrow} \stackrel{\text{def}}{=} \{(\tau, t) \mid \tau\uparrow_t \in e\},$$

A timed trace τ satisfies $e^{\downarrow}/e^{\uparrow}$ at time t only if its prefix/suffix is contained in e, respectively.

We are now able to map observations to events. However, we are still unable to construct observations out of events, i.e., to relate (composite) events to each other. In order to be able to do this, we define a metric temporal logic over $\vec{C}$ in the next section. The formulas of this logic have timed events and observation predicates as atomic propositions and are semantically defined over the domain of sets of valuations themselves.

CPS Case Study (*con'ed*). For our CPS case study we define the observation predicate $e_{PC} \subseteq TiTr_{obj}$ which contains observations which begin with an environment-object-observation list and end at the time of the earliest potential crash under the assumption that the velocity and angle of the objects will not change over time:

$$e_{PC} \stackrel{\text{def}}{=} \{\tau \mid crash(\tau.(\min \phi(\tau))) = (\max \phi(\tau))\},$$

where $crash\colon M^*_{obj} \mapsto \mathbb{R}_{\geq 0}$ is defined as

$$crash(\langle\rangle) \stackrel{\text{def}}{=} \infty,$$

$$crash(\langle m^{(1)}_{obj} \ldots m^{(n)}_{obj}\rangle) \stackrel{\text{def}}{=} \min_{1 \leq i \leq n} \begin{cases} \frac{m^{(i)}_{obj}.d}{m^{(i)}_{obj}.v \cdot \cos m^{(i)}_{obj}.\theta} & \text{if } m^{(i)}_{obj}.\theta < 90 \text{ and } m^{(i)}_{obj}.v \neq 0, \\ \infty, & \text{o.w.} \end{cases}$$

The *crash*-function expresses that at every given point in time the CPS system is interested in the object with the most urgent collision prognosis.

By $e_{PC}^{\exists\uparrow}$ we obtain the set of all predicted collisions and can relate them to the activations of airbag and belt tensioner. $e_{PC}^{\exists\downarrow}$ consists of environment-object-observation events which occur during different lifelines of a vehicle. $e_{PC}^{\downarrow}$ is satisfiable only at time of the first predicted collision, and $e_{PC}^{\uparrow}$ is satisfiable only for the points in time, in which no collision is predicted, or for the points in time within finite traces where the crash was predicted correctly.

2.2.5 Metric Temporal Logic for Timed Traces

Temporal logic is the formalism commonly used to express relations between events. While pure temporal logic, like LTL [Pnu77], is intended to capture causality relationships, its various extensions enrich it by additional timing constraints. These logics are referred to as *quantitative* temporal logics. [BMN00] gives an overview of temporal logics specially tailored for real-time system specification. Moreover, [BMN00] provides a taxonomy of classification criteria for quantitative temporal logics. Among these criteria are:

order of temporal logic This is the order of the classical logic on which the temporal logic is constructed. Typical examples are propositional and first-order logics.

temporal domain structure There exists a distinction between *linear* and *branching* temporal logics. Intuitively, linear temporal logics (e.g., LTL) are defined over sets of system executions, while branching temporal logics (e.g., CTL [CES83]) – over trees, which describe for every system state the sets of possible predecessor and successor states.

fundamental entity of logic The basic way to characterize a temporal logic is whether it uses points or intervals to model time. Point-based temporal logics express relationships among events in terms of points, and define intervals as a connected points set. Time durations are expressed by using quantification over time.

Interval temporal logics are more expressive, since they are capable of describing observations during time intervals, and a single time instant is represented by a singular interval $[d,d]$. Interval-based logics usually present specific operators to express the relationships between intervals (e.g., the thirteen interval alignments from [All83]), and/or operators for combining intervals (e.g., the chop operator $\varphi \mathbin{C} \psi$, which partitions intervals in two subintervals fulfilling φ and ψ, respectively), or operators to specify the interval boundaries on the basis of the truth of predicates (cf. previous section).

metric for time The presence of a metric for time distinguishes the quantitative temporal logics (e.g., MTL [Koy90]) from the *qualitative* ones (e.g., LTL, CTL). In a qualitative temporal logic only predecessor/successor relationships between events can be expressed. On the other hand, quantitative temporal logics can also express distances between events or durations of observations.

time implicit/explicit Time in temporal logics can be defined in an *implicit* or *explicit* manner. In implicit models of time, the meaning of formulas depends on the evaluation time, and

this remains implicit in the formula (e.g., LTL, CTL, MTL). In explicit time models, time is represented by a special variable (e.g., RTL [JMS88]).

Here, we present a variation of the *metric temporal logic (MTL)* [Koy90] defined over the domain of timed valuations introduced in Section 2.2.3. According to the classification criteria described above, our semantics of MTL yields a quantitative linear first-order temporal logic which is interval based and time implicit.

Syntax and Semantics of MTL

In the previous section we introduced two different interpretations for the satisfaction of timed events by observations, which are convenient for real-time specifications: A timed event happens during an observation at a certain point in time, and an event represents the occurrence of a composite observation, which has happened/will happen at a certain point in time during another observation. Both of these interpretations and transitions between them, have to be captured by a formalism designed for the description of timed specifications.

The syntax of MTL written in extended Backus-Naur Form (BNF) is:

$$\varphi ::= \mathrm{tt} \mid A \mid e^{\exists\downarrow} \mid e^{\exists\uparrow} \mid e^{\downarrow} \mid e^{\uparrow} \mid \varphi \wedge \varphi \mid \neg\varphi \mid \varphi \,\mathrm{U}_{\mathbb{I}}\, \varphi \mid \varphi \,\mathrm{S}_{\mathbb{I}}\, \varphi$$

Here, A and e stand for symbolic names which are mapped to the respective semantic meanings $A \subseteq \vec{C}^{\mathcal{A}}$ and $e \subseteq \vec{C}$. We map MTL formulas to their semantic meanings using the predicate $(\tau, t) \models \varphi$. It is inductively defined for a trace $\tau \in \vec{C}$, a point in time $t \in \mathbb{T}$, an observation predicate $e \subseteq \vec{C}$, a singular event $A \subseteq \vec{C}^{\mathcal{A}}$, and formulas $\varphi, \varphi_1, \varphi_2$, as following:

- $(\tau, t) \models \mathrm{tt}$ iff $t \in \phi(\tau)$,
- $(\tau, t) \models A$ iff $t \in \phi(\tau)$ and $\tau.t \in A$,
- $(\tau, t) \models e^{\downarrow}$ iff $\tau{\downarrow_t} \in e$,
- $(\tau, t) \models e^{\uparrow}$ iff $\tau{\uparrow_t} \in e$,
- $(\tau, t) \models e^{\exists\downarrow}$ iff $\exists u \in \mathbb{T} : \tau|_{[u,t]} \in e$,
- $(\tau, t) \models e^{\exists\uparrow}$ iff $\exists u \in \mathbb{T}_\infty : \tau|_{[t,u)} \in e$,
- $(\tau, t) \models \varphi_1 \wedge \varphi_2$ iff $(\tau, t) \models \varphi_1$ and $(\tau, t) \models \varphi_2$,
- $(\tau, t) \models \neg\varphi$ (or, equally $(\tau, t) \not\models \varphi$) iff not $(\tau, t) \models \varphi$,
- $(\tau, t) \models \varphi_1 \,\mathrm{U}_{\mathbb{I}}\, \varphi_2$ iff exists $t' \in \phi(\tau)$, s.t. $(\tau, t') \models \varphi_2$ and $t' - t \in \mathbb{I}'$ and $\forall t'' \in t + \mathbb{I}' : t'' < t' \Rightarrow (\tau, t'') \models \varphi_1$, where $\mathbb{I}' = \mathbb{I} \setminus \{\infty\}$[3],
- $(\tau, t) \models \varphi_1 \,\mathrm{S}_{\mathbb{I}}\, \varphi_2$ iff exists $t' \in \phi(\tau)$, s.t. $(\tau, t') \models \varphi_2$ and $t - t' \in \mathbb{I}'$ and $\forall t'' \in t - \mathbb{I}' : t' < t'' \Rightarrow (\tau, t'') \models \varphi_1$, where $\mathbb{I}' = \mathbb{I} \setminus \{\infty\}$[3],

The following explanations omit previously discussed $.^{\exists\uparrow}$, $.^{\exists\downarrow}$, $.^{\uparrow}$, $.^{\downarrow}$, and A modalities, cf. Section 2.2.4 for details.

[3] By this trick we omit additional case distinctions for interval parameters of the form $[n, \infty]$ or $(n, \infty]$.

We abbreviate $(\tau,t) \not\models \mathrm{tt}$ by $(\tau,t) \models \mathrm{ff}$. Please note that for any t holds: $(\{C \mapsto \langle\rangle\},t) \models \mathrm{ff}$. The semantics of a formula φ is a set of timed observation and point in time pairs. Thus, an observation $\tau \in \vec{C}$ satisfies an MTL formula at a given point in time $t \in \mathbb{T}$ if and only if (τ,t) is the member of the corresponding set. We say that φ was observed or has happened in τ at t.[4] The conjunction and negation on formulas can be seen as intersection and subtraction from subset of $\wp(\vec{C} \times \mathbb{T})$ in the set-theoretical sense, respectively.

$$
\begin{array}{rcl@{\qquad}rcl}
\mathrm{ff} & \stackrel{\mathrm{def}}{\equiv} & \neg\mathrm{tt} & \Diamond_{\mathbb{I}}\varphi & \stackrel{\mathrm{def}}{\equiv} & \mathrm{tt}\,\mathrm{U}_{\mathbb{I}}\,\varphi \\
\varphi_1 \vee \varphi_2 & \stackrel{\mathrm{def}}{\equiv} & \neg(\neg\varphi_1 \wedge \neg\varphi_2) & \Box_{\mathbb{I}}\varphi & \stackrel{\mathrm{def}}{\equiv} & \neg\Diamond_{\mathbb{I}}\neg\varphi \\
\varphi_1 \rightarrow \varphi_2 & \stackrel{\mathrm{def}}{\equiv} & \neg\varphi_1 \vee \varphi_2 & \Diamond\varphi & \stackrel{\mathrm{def}}{\equiv} & \Diamond_{[0,\infty)}\varphi \\
\varphi_1 \leftrightarrow \varphi_2 & \stackrel{\mathrm{def}}{\equiv} & (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1) & \Box\varphi & \stackrel{\mathrm{def}}{\equiv} & \Box_{[0,\infty)}\varphi \\
\varphi_1\,\mathrm{U}\,\varphi_2 & \stackrel{\mathrm{def}}{\equiv} & \varphi_1\,\mathrm{U}_{[0,\infty)}\,\varphi_2 & \blacklozenge_{\mathbb{I}}\varphi & \stackrel{\mathrm{def}}{\equiv} & \mathrm{tt}\,\mathrm{S}_{\mathbb{I}}\,\varphi \\
\varphi_1\,\mathrm{S}\,\varphi_2 & \stackrel{\mathrm{def}}{\equiv} & \varphi_1\,\mathrm{S}_{[0,\infty)}\,\varphi_2 & \blacksquare_{\mathbb{I}}\varphi & \stackrel{\mathrm{def}}{\equiv} & \neg\blacklozenge_{\mathbb{I}}\neg\varphi \\
 & & & \blacklozenge\varphi & \stackrel{\mathrm{def}}{\equiv} & \blacklozenge_{[0,\infty)}\varphi \\
 & & & \blacksquare\varphi & \stackrel{\mathrm{def}}{\equiv} & \blacksquare_{[0,\infty)}\varphi
\end{array}
$$

Table 2.3: *Derived MTL Modalities*

A timed trace satisfies a formula with the timed until-modality ($\mathrm{U}_{(.)}$) if and only if it fulfills the left sub-formula starting from $\mathrm{glb}\,\mathbb{I}$ up to some point in time t, which still lies within $\mathbb{I}$; and at this point, the right sub-formula is satisfied. The past can be referenced by the since-modality ($\mathrm{S}_{(.)}$) which is defined analogously to until, but from the current point in a timed trace backward.

Given a formula φ and a trace τ, $\tau \models \varphi$ if and only if $(\tau,\phi^{glb}(\tau)) \models \varphi$. φ is called *satisfiable* if there exists a timed trace $\tau \in \vec{C}$ such that $\tau \models \varphi$. φ is called *valid* (or *tautology*) if it is satisfiable by all timed observations from $\vec{C}$ (denoted by $\models \varphi$). Finally, we define the equivalence relation on MTL formulas as

$$(\varphi \equiv \psi) \stackrel{\mathrm{def}}{\Leftrightarrow} (\forall \tau \in \vec{C}, t \in \mathbb{T} : (\tau,t) \models \varphi \Leftrightarrow (\tau,t) \models \psi).$$

Further useful derived abbreviations are listed in Table 2.3. Among others we define untimed versions of until and since modalities, timed and untimed "finally φ" (also called "eventually φ") and "globally φ", denoted by $\Diamond$ and $\Box$, respectively, as well as their past counterparts: the "once"-operator $\blacklozenge$ and the past-version of globally-operator $\blacksquare$. Figure 2.2 gives a graphical illustration of some more involved MTL modalities.

***Remark* 2.6** (Alternative semantics for MTL)**.** Our definitions of the until and since modalities from above deviate from their semantics typically found in the literature (cf. [AH92, BMN00] for comprehensive surveys on real-time and quantitative logics). In approaches listed there, the first formula has to be satisfied for all points in time starting from the current one, not from the earliest one within $\mathbb{I}$. However, we can easily represent these definitions using our semantics by

$$\varphi\,\mathrm{U}'_{\mathbb{I}}\,\psi \stackrel{\mathrm{def}}{\equiv} (\Box_{[0,\mathrm{glb}\,\mathbb{I}]\setminus\mathbb{I}}\varphi) \wedge (\varphi\,\mathrm{U}_{\mathbb{I}}\,\psi) \qquad \text{and} \qquad \varphi\,\mathrm{S}'_{\mathbb{I}}\,\psi \stackrel{\mathrm{def}}{\equiv} (\blacksquare_{[0,\mathrm{glb}\,\mathbb{I}]\setminus\mathbb{I}}\varphi) \wedge (\varphi\,\mathrm{S}_{\mathbb{I}}\,\psi).$$

[4] Although for the $.^{\exists\uparrow}/.^{\uparrow}$-modalities the description "will be observed", or "will happen" is more appropriate.

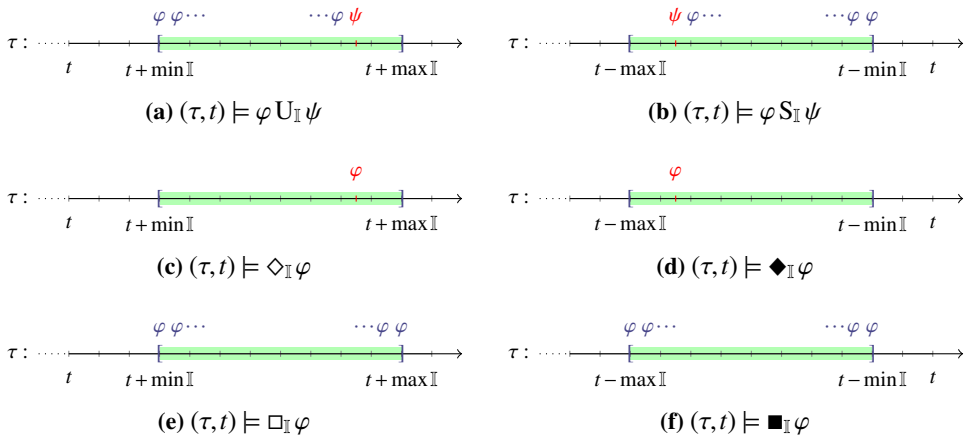


Figure 2.2: *An Illustration of Selected MTL Modalities*

The transition in the opposite direction is also possible. The reason for this choice of semantics is its effect on the definition of $\Box_{(.)}$- and $\blacksquare_{(.)}$-modalities: they allow us to express *distances* between events more naturally. For example, $\varphi \rightarrow \Box_{(0,b]} \neg\varphi$ would always yield "false" if $\Box_I$ has been defined using U'_I. In our semantics this is not the case. ◦

$$\Diamond_I(\varphi \vee \psi) \equiv \Diamond_I \varphi \vee \Diamond_I \psi \qquad (* \ \Diamond_I \text{ distributive over } \vee \ *) \tag{2.1a}$$

$$\Box_I(\varphi \wedge \psi) \equiv \Box_I \varphi \wedge \Box_I \psi \qquad (* \ \Box_I \text{ distributive over } \wedge \ *) \tag{2.1b}$$

$$\Diamond_I \varphi \vee \Diamond_J \varphi \equiv \Diamond_{I \cup J} \varphi \quad \text{if } (I \cup J) \in [\mathbb{T}_\infty] \qquad (* \text{ interval arithmetic for } \Diamond_I \ *) \tag{2.1c}$$

$$\Box_I \varphi \wedge \Box_J \varphi \equiv \Box_{I \cup J} \varphi \quad \text{if } (I \cup J) \in [\mathbb{T}_\infty] \qquad (* \text{ interval arithmetic for } \Box_I \ *) \tag{2.1d}$$

$$\neg \Box_I \varphi \equiv \Diamond_I \neg\varphi \qquad (* \text{ duality of } \Diamond_I \text{ and } \Box_I \ *) \tag{2.1e}$$

$$\neg(\varphi \, U_I \, \psi) \equiv (\neg\psi \, U_I \, (\neg\varphi \wedge \neg\psi)) \vee \Box_I \neg\psi \qquad (* \text{ negation of } U_I \ *) \tag{2.1f}$$

$$(e_1 \cap e_2)^\updownarrow \equiv e_1^\updownarrow \wedge e_2^\updownarrow \qquad (* \ \updownarrow \in \{\downarrow, \uparrow\} \text{ distributive over } \wedge \ *) \tag{2.1g}$$

$$(\neg e)^\updownarrow \equiv \neg(e^\updownarrow) \qquad (* \ \updownarrow \in \{\downarrow, \uparrow\} \text{ commutative with } \neg \ *) \tag{2.1h}$$

Table 2.4: *Useful MTL Equivalences*

A variety of well-known results about the properties of different temporal logic operators can be transferred into the presented semantic domain. Table 2.4 summarizes the most important ones. Their correctness is proved in Proposition A.1.

2.3 A Formal Model for Time Requirements

Specifications define use cases of the system and constrain their time behavior. The *high-level* time requirements define the observable time behavior of certain functions without referencing any implementation details. In order to ensure their fulfillment by the implementation, the high-level requirements have to be broken down to the level of technical systems. While a big variety of different high-level time requirements are imaginable, the technical systems operate with a limited set of of established concepts for time behavior, like latency, WCET, response time, *etc.* In this context we speak about *low-level* requirements. The current section assigns a formal meaning to these concepts and shows how they can be composed to consistent specifications.

Formally, a time requirement r is a (composite) timed event, modeled by an MTL formula, over some semantic domain $\vec{C}$, i.e., $r \subseteq \vec{C} \times \mathbb{T}$. By this, r incorporates and relates both functional and time constraints. We discuss this circumstance in the subsequent paragraphs.

Time vs. Causality

According to [Fle83]:

> "*Causality* denotes a necessary relationship between one event (called cause) and another event (called effect) which is the direct consequence (result) of the first."

The causality relationships between events often build the foundation for the time behavior of a system. In fact, a statement that A_2 must follow A_1 within some time window implies that A_1 is eventually followed by A_2. By this, the specification of time behavior can be seen as the strengthening of causality constraints. This fact also explains the existence of numerous extensions of temporal logic by different notions of real-time (cf. [EMSS92] for a survey on this topic).

On the other hand, constraints on distances between events, without any further considerations of their relative order, are clearly orthogonal to the causality. This brings us to the question whether the approach chosen here, namely the embedding of time constraints into temporal operators, is really feasible? Can a system specification be subdivided into pure temporal and timed parts, whose conjunction yields the original specification again? Surely these questions can be answered in favor of the alternative approach for certain subclasses of temporal formulas, e.g., for

$$\Box \bigwedge_i (\varphi_1^{(i)} \boldsymbol{X}_{\mathbb{I}}^{(i)} \varphi_2^{(i)}),$$

where for all i, $\varphi_1^{(i)}$ and $\varphi_2^{(i)}$ contain no timed modalities and $\boldsymbol{X}_{\mathbb{I}}^{(i)} \in \{\mathrm{U}_{\mathbb{I}}, \mathrm{S}_{\mathbb{I}}\}$. Here, by use of the commutativity of $\wedge$ and the distributivity of $\Box$ and $\wedge$ (cf. Property (2.1b)) we could partition every conjunct into timed and causal parts and group them to pure causal and pure time properties. However, in general case such a partition is not possible, due to the non-distributivity of the temporal modalities "until" and "since". Intuitively, the outer modalities in the formula define the *context*, in which the inner sub-formulas should be true. Bringing (parts of) these sub-formulas

out of the context, in the most cases, leads to an unjustified strengthening of the specification. Let us illustrate the above considerations by the following example.

Example 2.2 (Causal context for time constraints)**.** We define a distance constraint between events specified by formulas φ_1 and φ_2 by

$$d(\varphi_1, \varphi_2, \mathbb{I}) \stackrel{\text{def}}{\equiv} \Box(\varphi_1 \rightarrow (\Diamond_{\mathbb{I}} \varphi_2 \vee \blacklozenge_{\mathbb{I}} \varphi_2)) \wedge \Box(\varphi_2 \rightarrow (\Diamond_{\mathbb{I}} \varphi_1 \vee \blacklozenge_{\mathbb{I}} \varphi_1)).$$

Now consider the following requirement: "In the mode m1 the system must react to any request q by issuing the response p within $\mathbb{I}$ units of time." Let e_{m1}, e_q and e_p characterize the observations of the mode change, request and response, respectively. We also need the notion of switch to some other mode[5]: e_{m2}. Then, using the MTL framework, introduced so far, we can formalize the above requirement as:

$$r \stackrel{\text{def}}{\equiv} \Box\Big(e_{\text{m1}}^{\exists\downarrow} \rightarrow ((e_q^{\exists\downarrow} \rightarrow \Diamond_{\mathbb{I}} e_p^{\exists\downarrow}) \,\mathrm{U}\, e_{\text{m2}}^{\exists\downarrow})\Big).$$

Now compare this formalization with the conjunction of its untimed version and the distance event:

$$r' \stackrel{\text{def}}{\equiv} \Box\Big(e_{\text{m1}}^{\exists\downarrow} \rightarrow ((e_q^{\exists\downarrow} \rightarrow \Diamond\, e_p^{\exists\downarrow}) \,\mathrm{U}\, e_{\text{m2}}^{\exists\downarrow})\Big) \wedge d(e_q^{\exists\downarrow}, e_p^{\exists\downarrow}, \mathbb{I}).$$

We can show that $\models r' \rightarrow r$, but the implication in the opposite direction does not hold. It is common sense not to over-constrain specifications. Thus, we consider the r-constraint to be more preferable. ∘

After this rationale of our design choices, we can now proceed to the formal definition of low-level time requirements. They are described in Section 2.3.3. However, before this we define a number of important occurrence patterns in Section 2.3.1 and important time patterns in Section 2.3.2, which serve as the basic building blocks of the time constraints. At the end of the current section (in 2.3.4) we combine time requirements to time specifications. For this purpose we introduce functional *contexts* of requirements, A context defines, when associated requirements must hold. For instance, using contexts it is possible to state that a certain requirement is an invariant of the system (must always hold), or that it must hold in certain operational modes only. By this, we can isolate time requirements from their functional embedding.

2.3.1 Important Occurrence Patterns

Besides the causality which is the main concern of classical temporal logics, a further important class of (untimed) relations between events is the relative number of their occurrences. Informally, these relations express that within any observation of an arbitrary but finite length the number of observed instances of event A_1 and the number of observed instances of event A_2 stand in a certain relation to each other. An especially important relationship form the aforementioned class is the *one-to-one correspondence* between events. It is described and formalized in the rest

[5] Alternatively we could model the operating mode as an interval event, i.e., an event which evaluates to true in every point in time in which the system is in the mode m1.

of this section and will be used as a building block in the formalization of time requirements in Section 2.3.3.

When a specification imposes a certain constraint on the occurrence of a pair of events, a question naturally arises, whether this constraint must be fulfilled for every individual event pair or not. For example, the requirement "a request A_1 must be always followed by the response A_2 within the time window $[min, max]$" does not answer the question, whether there can exist further occurrences of A_2 either within the specified time window or even outside. However, in most cases, spontaneous or redundant responses are *not* a desired feature. This information, which often remains implicit in informal requirements documents, has to be made explicit in formal models, since it permits to deduce more elaborate statements concerning the consistency and completeness of the specification model (cf. Sections 3.1.2 and 3.1.3 in the next chapter).

In order to be able to reason about the observation numbers of events, we denote the set of all atomic events happened in the trace τ during time interval $\mathbb{I} \in [\mathbb{T}_\infty]$ at which the formula φ is satisfied by $\mathsf{sat}(\varphi, \tau, \mathbb{I})$. It is formally defined as:

$$\mathsf{sat}(\varphi, \tau, \mathbb{I}) \stackrel{\text{def}}{=} \{(\tau.t, t) \mid t \in (\mathbb{I} \cap \phi(\tau)) \wedge (\tau, t) \models \varphi\}.$$

Please note that the interval parameter in the above definition defines the *absolute* time window. Then, $|\mathsf{sat}(\varphi, \tau, \mathbb{I})|$ is the *observation number* of φ in τ during $\mathbb{I}$. Given two events A_1 and A_2 we are able to describe observation predicates which satisfy some predicate on A_1's and A_2's occurrences $p \colon \mathbb{N}_\infty \times \mathbb{N}_\infty \mapsto \mathbb{B}$ by

$$e(A_1, A_2) \stackrel{\text{def}}{=} \{\tau \mid p(|\mathsf{sat}(A_1, \tau, \phi(\tau))|, |\mathsf{sat}(A_2, \tau, \phi(\tau))|)\}.$$

Using the above preliminaries, we define two predicates, whose conjunction will forbid the occurrence number of one event to exceed or under-run the occurrence number of the other at any time. Formally, for φ_1, φ_2 and $d \in \mathbb{T}_\infty$, we define

$$creq(\varphi_1, \varphi_2) \stackrel{\text{def}}{\equiv} \blacksquare\{\tau \mid |\mathsf{sat}(\varphi_1, \tau, \phi(\tau))| \geq |\mathsf{sat}(\varphi_2, \tau, \phi(\tau))|\}^{\downarrow}, \tag{2.2a}$$

$$cres(\varphi_1, \varphi_2, d) \stackrel{\text{def}}{\equiv} \blacksquare \Diamond_{[d,d]}\{\tau \mid |\mathsf{sat}(\varphi_1, \tau, \phi(\tau) - d)| \leq |\mathsf{sat}(\varphi_2, \tau, \phi(\tau))|\}^{\downarrow}. \tag{2.2b}$$

The predicate $creq(A_1, A_2)$ permits only runs, where for any prefix the number of occurrences of A_1 ("requests") exceeds the number of occurrences of A_2 ("responses"), i.e., it disallows spontaneous or multiple responses. The second predicate *cres* demands that any time an instance of A_1 should be followed by an instance of A_2 within at most d time units, i.e., it forbids unanswered requests. The constraint $creq(A_1, A_2) \wedge cres(A_1, A_2, d)$ yields a one-to-one correspondence between event pairs. This is proved by the next proposition.

Lemma 2.1 (One-to-one correspondence of events)**.** *Given a pair of events $A_1, A_2 \in \wp(\vec{C} \times \mathbb{T})$ and an event distance $d \in \mathbb{T}_\infty$. Then, provided*

$$(\tau, t) \models creq(A_1, A_2) \wedge cres(A_1, A_2, d)$$

holds for some $\tau \in \vec{C}$ and $t \in \phi(\tau)$,

(1) for every point in time $t_1 \leq t$ in which $(\tau, t_1) \models A_1$ there exists a point in time t_2 such that $t_1 \leq t_2 \leq t_1 + d$ and $(\tau, t_2) \models A_2$, and
(2) for every $t_2 \leq t + d$ where $(\tau, t_2) \models A_2$ there exists $t_1 \geq t_2 - d$ in which $(\tau, t_1) \models A_1$.

Proof. Suppose there exists some point in time t_1, where $(\tau, t_1) \models A_1$ and there is *no* $t_2 \in [t_1, t_1 + d]$, such that $(\tau, t_2) \models A_2$, then the number of instances of A_2 does not change for any point in this interval and we obtain

$$
\begin{aligned}
& (\tau, t_1) \models creq(A_1, A_2) \wedge cres(A_1, A_2, d) \\
\Rightarrow \quad & (* \text{ definitions of } creq \text{ and } cres, \text{ above assumption } *) \\
& |\mathsf{sat}(A_1, \tau, [0, t_1])| \geq |\mathsf{sat}(A_2, \tau, [0, t_1])| \wedge |\mathsf{sat}(A_1, \tau, [0, t_1])| \leq |\mathsf{sat}(A_2, \tau, [0, t_1])| \\
\Leftrightarrow & \\
& |\mathsf{sat}(A_1, \tau, [0, t_1])| = |\mathsf{sat}(A_2, \tau, [0, t_1])|
\end{aligned}
$$

Now we must consider two cases: $t_1 = \phi^{glb}(\tau)$ and $t_1 > \phi^{glb}(\tau)$. In the first case we immediately obtain a contradiction, since the number of A_1 instances is at least one and, according to the assumption made at the beginning, the number of A_2 instances has to be zero. In the second case, we can consider the subtrace $\tau|_{[0,t_1)}$. The following holds for it:

$$|\mathsf{sat}(A_1, \tau, [0, t_1))| < |\mathsf{sat}(A_2, \tau, [0, t_1))|.$$

And this contradicts the fact that any prefix of τ satisfies $creq(A_1, A_2)$.

The opposite direction for some instance of A_2 goes analogously. Thus, due to the monotony of timed traces, both properties hold for τ and t. □

The nature of a one-to-one correspondence has the transitivity as one of its consequences. This property is used in the next chapter for the forward and backward propagation of time constraints.

Lemma 2.2 (Transitivity of *creq* and *cres*)**.** *For any $A_1, A_2, A_3 \in \wp(\vec{C} \times \mathbb{T})$ and $d_1, d_2 \in \mathbb{T}$*

$$(creq(A_1, A_2) \wedge creq(A_2, A_3)) \Rightarrow creq(A_1, A_3), \tag{2.3a}$$
$$(cres(A_1, A_2, d_1) \wedge cres(A_2, A_3, d_2)) \Rightarrow cres(A_1, A_3, d_1 + d_2). \tag{2.3b}$$

Proof. The transitivity of *creq* (Property (2.3a)) follows immediately from the transitivity of $\leq$. For *cres* we obtain

$$
\begin{aligned}
& (\tau, t) \models cres(A_1, A_2, d_1) \wedge cres(A_2, A_3, d_2) \\
\Leftrightarrow \quad & (* \text{ definition of } cres *) \\
& \forall t_1, t_2 \in [0, t] \cap \phi(\tau) : \quad |\mathsf{sat}(A_1, \tau, [0, t_1 - d_1])| \leq |\mathsf{sat}(A_2, \tau, [0, t_1])| \\
& \qquad\qquad\qquad\qquad \wedge |\mathsf{sat}(A_2, \tau, [0, t_2 - d_2])| \leq |\mathsf{sat}(A_3, \tau, [0, t_2])|
\end{aligned}
$$

We fix $t_1 = \max\{\phi^{glb}(\tau), t_2 - d_2\}$. Then, due to the definition of the suffix-operator "$\uparrow_{(.)}$" we have

$$\tau\uparrow_{t_1} = \tau\uparrow_{t_2 - d_2}$$

and thus we obtain Property (2.3a) by the following

$$
\begin{aligned}
&\Rightarrow \\
&\quad \forall t_2 \in [0,t] \cap \phi(\tau): \quad |\mathsf{sat}(A_1,\tau,[0,t_2-d_1-d_2])| \leq |\mathsf{sat}(A_2,\tau,[0,t_2-d_2])| \\
&\qquad\qquad\qquad\qquad \wedge |\mathsf{sat}(A_2,\tau,[0,t_2-d_2])| \leq |\mathsf{sat}(A_3,\tau,[0,t_2])| \\
&\Leftrightarrow \\
&\quad \forall t_2 \in [0,t] \cap \phi(\tau): |\mathsf{sat}(A_1,\tau,[0,t_2-d_1-d_2])| \leq |\mathsf{sat}(A_3,\tau,[0,t_2])| \\
&\Leftrightarrow \quad (* \text{ definition of } \mathit{cres} *) \\
&\quad (\tau,t) \in \mathit{cres}(A_1,A_3,d_1+d_2)
\end{aligned}
$$

□

2.3.2 Important Time Patterns

The building blocks of time requirements, which are presented in the next section, are causal relations between events parametrized by relative time intervals. The general scheme for these relations is: Provided the event A_1 has happened at time t then the event A_2 must happen/must not happen within $t+\mathbb{I}/t-\mathbb{I}$.[6] The four cases resulting from this scheme are formalized below. In particular, in our settings of timed traces we present the notions of the *bounded response* and the *minimal separation*. These patterns were originally identified in [MP95] as important time properties and cover the both variants for the forwards-directed case "$t+\mathbb{I}$" contained in the scheme. Here, we present slightly more general versions of these properties. We also introduce the respective inverse patterns which correspond to the $t-\mathbb{I}$-case – the *bounded request* and the *minimal displacement*. Finally, due to the asymmetry between A_1 and A_2 in the scheme from above, for the formalization of requirements which do not demand a particular order of events we also will require the combination of bounded response and request – the *bounded distance*.

- *Bounded response*: A_1 should be followed by A_2 not earlier than $\mathrm{glb}\,\mathbb{I}$ and no later than $\mathrm{lub}\,\mathbb{I}$ time units

$$br(A_1,A_2,\mathbb{I}) \stackrel{\text{def}}{\equiv} A_1 \rightarrow \Diamond_{\mathbb{I}} A_2. \tag{2.4a}$$

- The symmetrical pattern is the *bounded request*, stating that a response is always preceded by a request within a certain time-span

$$bq(A_1,A_2,\mathbb{I}) \stackrel{\text{def}}{\equiv} A_2 \rightarrow \blacklozenge_{\mathbb{I}} A_1. \tag{2.4b}$$

- *Bounded distance* demands that A_2 must occur at some distance from A_1 without stating anything about their order

$$bd(A_1,A_2,\mathbb{I}) \stackrel{\text{def}}{\equiv} br(A_1,A_2,\mathbb{I}) \vee bq(A_2,A_1,\mathbb{I}). \tag{2.4c}$$

  We can simplify the definition of bd to $A_1 \rightarrow (\Diamond_{\mathbb{I}} A_2 \vee \blacklozenge_{\mathbb{I}} A_2)$.

[6] Though, for the case $t-\mathbb{I}$ the expressions "must have happened/must not have happened" are more suitable.

- *(Weak) minimal separation*: No A_2 can occur earlier than d time units after an occurrence of A_1

$$ms^{\omega}(A_1, A_2, d) \stackrel{\text{def}}{\equiv} \begin{cases} A_1 \rightarrow \Box_{(0,d)} \neg A_2 & \text{if } d > 0, \\ \text{tt}, & \text{o.w.}, \end{cases} \tag{2.4d}$$

$$ms(A_1, A_2, d) \stackrel{\text{def}}{\equiv} A_1 \rightarrow \Box_{[0,d)} \neg A_2. \tag{2.4e}$$

  The weak version of constraint allows A_2 to occur simultaneous with A_1. This is especially useful, when different occurrences of the same event have to be related. Contrary, $ms(A, A, d)$ yields "false" for any d.

- We call the converse pattern of the minimal separation, the *minimal displacement* and define it as

$$md(A_1, A_2, d) \stackrel{\text{def}}{\equiv} (\blacksquare_{[0,d]} \neg A_1) \rightarrow A_2. \tag{2.4f}$$

  The property holds if A_2 occurs as soon as no A_1 was observed d time units long.

2.3.3 Low-Level Time Constraints

The current section contains description and formalization of low-level time constraints. Time constraints are an integral part of requirements documents of reactive real-time systems. For a better classification and integration with other requirement types, we sort time constraints presented in this section into the taxonomy of requirements from [Gli07]. In that work the author compares and unifies the variety of different definitions of the term "non-functional requirement"; and it is a general consensus that time constraints belong to this requirements class. The taxonomy of [Gli07] extended by time constraints and relationships between them is visualized in Figure 2.3. It contains the following hierarchy of requirement types:

- On the first level, requirements are partitioned according to their respective purpose into *project*, *system*, and *process* requirements. The taxonomy of [Gli07] and the present thesis primary concentrate on the requirements put on the system.
- On the next level, system requirements are partitioned into functional requirements, *attributes*, and *constraints*. Constraints are requirements that constrain the solution space beyond what is necessary for meeting the given functional and performance requirements. Attributes gather requirements which are related either to the system as a whole (for instance, maintainability requirements) or to individual functional requirements (for instance, performance requirements).
- On the third level, attributes are partitioned into *performance requirements* and (further) *specific quality requirements*. Examples for specific quality requirements are usability, reliability, security, maintainability, *etc.* The performance requirements build the anchorage point of our taxonomy of the low-level time requirements, which are typically used to describe the time behavior of technical systems. It encompasses *jitter*, *deadline*, *period*, *offset*, *synchrony*, and *timeouts*. In the rest of this section every requirement type is given along with its formalization in our MTL framework.

The dashed arrows in Figure 2.3 denote the relationships between different requirement types. An arrow from concept "A" to concept "B" expresses that B is *based* on A. This relationship between functional and time requirements has already been discussed at the beginning of this section. It can be generalized for all performance requirements. The concept "jitter" which is described next is inherent to all other types of time requirements.

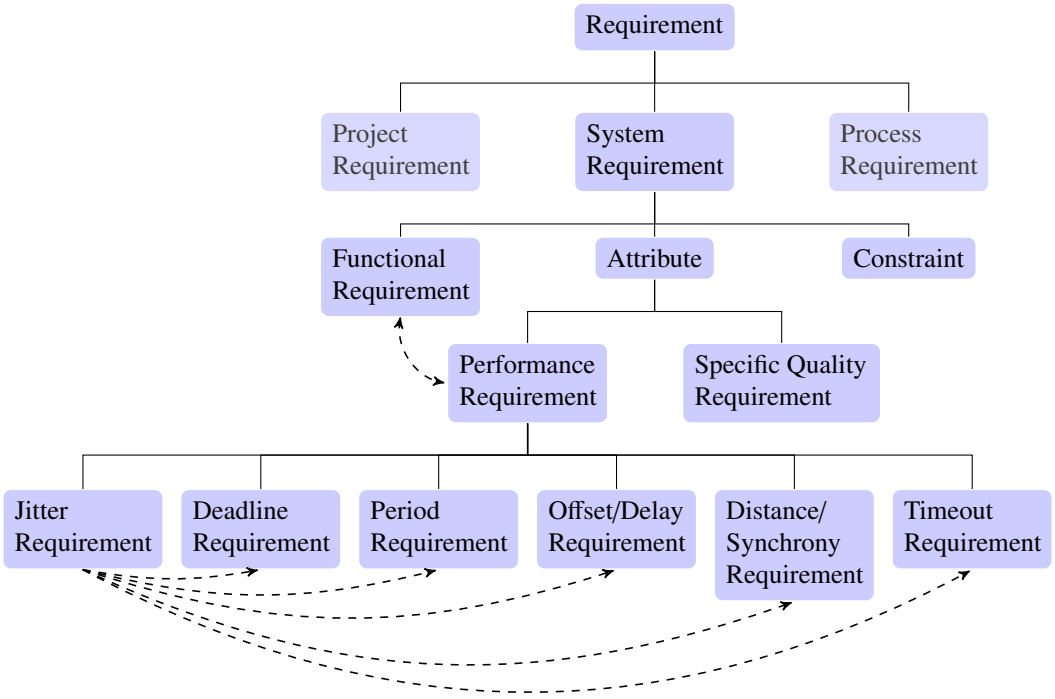


Figure 2.3: *A Taxonomy of Requirements*

Jitter

Jitter is an inherent part of any (hard) real-time behavior. Due to the non-determinism of hardware and input-dependent behavior of software, it is, in most cases, impossible to work with exact execution times for the operations of the system. Hardware solutions, like pipelines and caches, are designed to increase the average throughput of processors, but make the estimation of exact timing of operations difficult [FHL+01]. Depending on the values of the current inputs and the operation mode, the control flow within a software system may differ and thus require different amount of computing time. Finally, in the distributed systems, priority- or event-based communication protocols, like the one established in the CAN-buses, increase the level of uncertainty about the arrival time of events, since their latencies cannot be guaranteed. By the combination of the above factors and due to the fine resolution of time requirements, which are often expressed in hundreds or tens of microseconds, a real-time embedded system must operate within certain tolerance bounds for the actions of its environment.

We model jitter using the interval indexes of the modal operators introduced above. An interval $\mathbb{I} \in [\mathbb{T}_\infty]$ expresses, that an event must happen in at least $\mathrm{glb}\,\mathbb{I}$ and at most $\mathrm{lub}\,\mathbb{I}$ units of time, relative to the actual position in the trace. It does not state anything about occurrences of this event outside of $\mathbb{I}$, nor anything about the exact number of occurrences during $\mathbb{I}$. Please note that this approach also does not exclude the exact time case, which can be modeled by use of the singular intervals: $[n,n]$.

Deadline

Deadline or *response time* is a time span, in which a system has to react to a certain (external) stimulus. Thus, it can be specified using the bounded response formula introduced above:

$$r(A_1,A_2,\mathbb{I}) \stackrel{\text{def}}{\equiv} br(A_1,A_2,\mathbb{I}). \tag{2.5a}$$

With $\Box\, r(A_1,A_2,\mathbb{I})$ we obtain a timed version of the "leads-to" operator, known from many specification approaches, like UNITY [Cha88], TLA [Lam94], *etc.*

The r-requirement guarantees that an input event A_1 is followed by an output event A_2 within a certain time interval. However, this requirement does not prohibit the system to react (possibly spontaneous) with the same output *without* the preceding input event A_1. Quite often it is feasible to prohibit this possibility, i.e., to state that every A_2 is preceded by A_1:

$$r'(A_1,A_2,\mathbb{I}) \stackrel{\text{def}}{\equiv} br(A_1,A_2,\mathbb{I}) \wedge bq(A_1,A_2,\mathbb{I}).$$

If we look at the above definitions carefully, we notice that neither r, nor r' can guarantee, that there exists exactly one request for every response and vice versa. Consider, for example, a pair of events from A_2, which lie within an interval $\mathbb{I}$ relative to some event from A_1. A trace including this situation satisfies both above formulas. One possibility to prohibit this, is to demand that the minimal separation between any pair of A_1 or A_2 events exceeds $\mathbb{I}$, i.e., the jitter lies under the period of both event types. However, such a constraint is not always justifiable. For the general case we define the following constraint.

$$\begin{aligned} r_u(A_1,A_2,\mathbb{I}) \stackrel{\text{def}}{\equiv}\quad & br(A_1,A_2,\mathbb{I}) \wedge bq(A_1,A_2,\mathbb{I}) \\ & \wedge creq(A_1,A_2) \wedge cres(A_1,A_2,\mathrm{lub}\,\mathbb{I}). \end{aligned} \tag{2.5b}$$

The inverse property for response time is the *request time*, which states that if a response has happened, it must have been preceded by request. Formally, for a request event A_1, a response event A_2, and an interval $\mathbb{I}$ we define request time constraint q as:

$$q(A_1,A_2,\mathbb{I}) \stackrel{\text{def}}{\equiv} bq(A_1,A_2,\mathbb{I}). \tag{2.6}$$

Period

A period or *inter-arrival* time of events constraints times of their repetitious occurrence. The arrival pattern allows to designate events as *periodic* and *sporadic* ones. The periodic events

occur within an interval $\mathbb{I} \in [\mathbb{T}_+]$, i.e., the time span between two subsequent instances of an event is not zero and finite. The sporadic events define only the lower bound of the period: $\mathbb{I} \in [\mathbb{T}_\infty \setminus \{0\}]$, which must be also greater than zero. Both constraint types are formalized by the following definitions.

$$i_p(A, \mathbb{I}) \stackrel{\text{def}}{\equiv} ms^\omega(A, A, \text{glb}\,\mathbb{I}) \wedge br(A, A, \mathbb{I}), \tag{2.7a}$$

$$i_s(A, \mathbb{I}) \stackrel{\text{def}}{\equiv} ms^\omega(A, A, \text{glb}\,\mathbb{I}). \tag{2.7b}$$

i_p models periodic and i_s sporadic events. Once a periodic event has happened, it must reoccur within the specified time window. This is not demanded for sporadic events. By this, we permit sporadic events to occur finite number of times within infinite timed observations. If this property is not desired one could also use the i_p-constraint to specify sporadic events.

Another common scheme used to specify periodicity enjoins for an event A a period of $p \in \mathbb{T}$ and a jitter of $j \in \mathbb{T}$, with $2j < p$. The interpretation is that A must/will occur once in every interval $[k \cdot p - j, k \cdot p + j]$, for all $k \in \mathbb{N}$. This form of inter-arrival time requirement for some involved event A can be described as

$$i_p(A, p, j) \stackrel{\text{def}}{\equiv} \blacksquare\{\tau \mid |\text{sat}(A, \tau, \phi(\tau))| \in [\max\{0, \left\lfloor \frac{\#\tau - 2j}{p} \right\rfloor\}, \left\lceil \frac{\#\tau + 2j}{p} \right\rceil]\}^\downarrow. \tag{2.7c}$$

Informally, the definition of $i_p(A, p, j)$ is satisfied by timed traces in which at any time the actual number of observed A-events deviates from their demanded occurrence number by at most the specified jitter.

The constraints from above do not state anything about the occurrence of the first instance of the event A. In fact, they evaluate to true for any A-free trace. As for sporadic events, in general, there is no need to demand that A must eventually happen. For instance, a crash-event does not have to happen during every lifeline of a vehicle. As for periodic events, it is not only feasible to make a statement that A must happen ($\Diamond A$), but also to specify the time of its first occurrence. This will be the topic of the next paragraph devoted to *offsets* of events.

Offset and Delay

The *offset* or *delay* of an event A specifies the time of its first occurrence. This can be either an absolute point in time or a distance from some other event, like an operation-mode change. In the second case offset relates some event with the first occurrence of A after that event. The offsets become crucial for several causally related events, since their *relative* offsets influence the delays in the data and control flow and by this the top-level time behavior, like end-to-end response times. Formally, we model offsets by providing the following constraint:

$$o(A_0, A, l) \stackrel{\text{def}}{\equiv} ms(A_0, A, l). \tag{2.8a}$$

By this, the offset constraint of $l \in \mathbb{T}$ on an event A relative to an event A_0 disallows A to occur in τ within any time interval $t + [0, l)$, where $(\tau, t) \models A_0$, if $0 < l$ and evaluates to true, otherwise. For the absolute point-in-time constraints we set $A_0 = \{A \in \vec{C}^{\mathcal{A}} \mid \phi(A) = 0\}$; whereas, this solution

works for observations which start by 0 only, i.e., for any $\tau \in \vec{C}$ with $0 \notin \phi(\tau)$, $o(A_0, A, l)$ will always yield "true". Since timed traces which belong to our semantic domain are full, the above solution for the absolute-offset constraints is sufficient.

The o-constraint does not state anything about the actual point in time, when the first event observation should happen. In order to be able to require that the first A should happen exactly at time $t_0 + l$ we provide the following constraint.

$$o_e(A_0, A, l) \stackrel{\text{def}}{\equiv} ms(A_0, A, l) \wedge br(A_0, A, [l, l]). \tag{2.8b}$$

Next, we discuss three different approaches to consider offset of an event A within a formal timed specification. First of all, we can state the offset explicitly as an individual part of specification using the o-constraint from above, provided this information is available. The remaining two interpretations of offset assume that there is no explicit information about the first occurrence time points of A available.

An obvious treatment of the missing offset is to allow arbitrary finite offsets. For a given timed trace $\tau \in \vec{C}$ and the earliest point in time $t \in \phi(\tau)$ such that $(\tau, t) \models A$ we build an equivalence class

$$PR(\tau, t, A) \stackrel{\text{def}}{=} \{\tau' \frown \tau \mid \tau' \in \vec{C} \wedge \#\tau' < \infty \wedge \forall t' < t : (\tau' \frown \tau, t') \not\models A\}$$

and take τ as its representative member[7]. Then, τ fulfills a period constraint $i(A, \mathbb{I})$ if and only if any member of its equivalence class does so. Further on, τ with $(\tau, t) \models A_1 \vee A_2$ for the smallest $t \in \phi(\tau)$ fulfills a response time requirement $r(A_1, A_2, \mathbb{I})$ and a request time requirement $q(A_1, A_2, \mathbb{J})$ if and only if any member of the equivalence class $PR(\tau, t, A_1 \vee A_2)$ does so. This is explained by the fact that the period, response and request time constraints introduced above are left-prefix monotone.

Finally, for a given period constraint $i_p(A, \mathbb{I})$, a further feasible interpretation restricts the prefix length of a member of the equivalence class $PR(\tau, t, A)$ by the maximal allowed separation, i.e., for all $(\tau' \frown \tau) \in PR(\tau, t, A)$ holds $\#((\tau' \frown \tau)\downarrow_{\sqcap t}) < \mathrm{lub}\,\mathbb{I}$. Intuitively, this means that valid observations must fulfill the period constraint at any time. In other words, if we wait long enough (for at most $\mathrm{lub}\,\mathbb{I}$ units of time) we must observe at least one instance of A. Formally, we can model this by adding the constraint

$$r(\{\tau \in \vec{C}\}^{\uparrow}, A, [0, \mathrm{glb}\,\mathbb{I}] \cup \mathbb{I})$$

to the specification.

Distance and Synchrony

The response time requirements constrain not only the distance between events, but also the order of their occurrence. However, this is not always required. There exist events which have to occur simultaneously (within a certain tolerance bound $\mathbb{I}$) without any prescribed order. For instance, warning lights must flash nearly simultaneously, but the order of their flashes does not matter. For this purpose we define the *distance* constraint, which states that if A_1 was observed at

[7] Since $\tau = \emptyset \frown \tau$, holds $\tau \in PR(\tau, t, A)$.

some point in time, there must exist an A_2-instance either before or after that time. The same is true for any observation of A_2. The distance between event pairs is bounded by $\mathbb{I}$. We redefine the d-formula introduced in the example at the beginning of Section 2.3 using the bounded distance predicate from Section 2.3.2 and obtain the *sc*-requirement.

$$sc(A_1, A_2, \mathbb{I}) \stackrel{\text{def}}{\equiv} bd(A_1, A_2, \mathbb{I}) \wedge bd(A_2, A_1, \mathbb{I}). \tag{2.9}$$

Timeout

The above requirement types describe the normal-case behavior. In other words, the behavior of the system provided its environment fulfills certain assumptions. However, the environments of embedded systems are not always reliable. Thus, operation in exceptional situations should be a part of system specification. For time requirements these exceptions are often expressed by use of *timeouts*. A timeout of $d \in \mathbb{T}$ means that a certain condition was not fulfilled for a time d. We formalize this intuition by

$$tmo(A_1, A_2, A_3, d) \stackrel{\text{def}}{\equiv} bq(A_1, A_3, [d, d]) \wedge md(A_2, A_3, d). \tag{2.10}$$

The event A_2 typically represents the expected reaction of the environment to event A_1, If the system was waiting for A_2 for at least d units of time up to now, then A_3 is the exception handling of the system for this case.

Worst/Best Case Execution Time and Latency

Time constraints do not always reference to events explicitly. For example, the *worst* or *best case execution time* (*WCET*/*BCET*) of a function specify distance between its activation and suspension. The activation can be modeled as an event coming from the environment. For suspension there are two feasible options: it is either an environment event (e.g., an external interrupt), which preempts the execution by activating the OS dispatcher or it is a (composite) event initiated by the function/task itself. In the latter case the task causes an (internal) interrupt, which requests the scheduler to suspend it. Modeling of such events demands for a detailed model of the technical infrastructure of the system.

Another important example is the *latency* of some signal on the bus, which describes the transport duration of this signal through a communication line, in other words, the distance between the writing of a value into the send buffer on the producer side and its availability for the receiver side. Again, modeling availability of a message must rely on a sophisticated model of the bus system.

Without an underlying model of the technical infrastructure, there is neither a possibility to model BCET/WCET or latency formally, nor a possibility to relate them to other requirements, like response time, offset, *etc.* For example, the relation between a WCET constraint for some function and a response time requirement, which must be fulfilled by this function, is not obvious, since in general the communication points of this function with its environment do not

have to coincide with the time points of its activation and suspension. Without any further assumptions about technical infrastructure, the minimal response time can exceed WCET or the maximal response time can be less than BCET.

Analogous problems emerge by modeling the latency. A number of technical factors, like higher-prioritized events combined with event-triggered dispatching, time-triggered scheduling, context switch times, or polling intervals, induce additional delays into the data- and control-flow. Thus, not only the upper but also the lower bound of the actual delay may lie over the maximal latency. One may think that the lower bound of a latency constraint must be smaller than the minimal specified delay, but this also does not have to be true. A constraint which requires an offset $\mathbb{I}$ is fulfilled by a system with the actual offset of $\mathbb{J}$, provided $\mathbb{J} \subseteq \mathbb{I}$. Thus, the minimal latency can be greater than the demanded minimal offset, but has to be smaller then the actual one. In the next chapter we will discuss and formally define when the actual time behavior satisfies the specified one (cf. Section 3.1.1).

In general, technical models of OS and bus systems are not available/considered on the level of logical or functional specification, hence there is no possibility to describe constraints, like BCET/WCET or latency, directly within the formal specification. Further on, these constraints reference to implementation details, i.e., their actual impact on the system can only be concluded from the specifics of the technical realization. Thus, the constraints of such kind restrict directly the solution space (the realization) for a system, rather than its problem space (the specification). Chapter 5 demonstrates how the assumptions about the execution time of tasks and the communication latencies can be incorporated into a simulation proof between a task architecture and a technical platform.

Provided there exist formal models of OS and bus system, latency becomes response time of the bus system and BCET/WCET lower and upper bounds for the time span between release and suspension of some technical component minus the preemption times in between.

2.3.4 Timed Behavior Specification

The above requirement types constitute the low-level time behavior specification of reactive systems. This specification together with further functional requirements has to be fulfilled by the design and implementation of the system. Example 2.2 at the beginning of Section 2.3 has demonstrated that the functional and time specifications are closely interrelated: The functional requirements formulate conditions under which the time requirements must be fulfilled and vice versa. For example, the operation mode can trigger a specific response time requirement and a timeout can change the operation mode. The combination of functional and time requirements of the system p is captured by the *timed behavior specification* $Spec_p^{\mathbb{T},M} \subseteq \vec{C}^{\kappa}$. It is defined by use of timed events in the time domain $\mathbb{T}$ over the message set M as a subset of allowed full valuations of the system channels C. In the future, we will omit the superscripts and the subscript where the time domain, the message set, and the system identifier are clear from the context or irrelevant. We recall that the timed event or *time property* is defined as a subset of $\vec{C} \times \mathbb{T}$. Thus, we can formally relate these both concepts, i.e., define when a specification fulfills a property.

Coming back to Example 2.2, it is not feasible to assume that the different time requirements defined so far, will be valid during the whole lifeline of the system. Merely, under certain conditions, there will be phases, in which some time requirements have to be fulfilled, and others – not. Thus, for a given time requirement r we define the *context* $\mathcal{C}$ in which it must be fulfilled. Contexts contain infinite as well as finite parts of specifications. The intuitive interpretation of a time requirement/context pair for a given specification is, that the specification fulfills the requirement if and only if the requirement holds at every point in time which lies within the context. The following definition puts this idea in concrete terms.

Definition 2.4 (Context and satisfaction of time requirements)**.** *For a given specification* $Spec \subseteq \vec{C}^{\kappa}$*, a time requirement* $r \subseteq \vec{C} \times \mathbb{T}$*, and a* context $\mathcal{C} \subseteq \vec{C}^{\kappa} \times [\mathbb{T}_{\infty}]$*, the specification* $Spec$ fulfills *the requirement* r *in the context* $\mathcal{C}$*, denoted by* $Spec \models_{\mathcal{C}} r$ *if and only if*

$$\forall \tau \in Spec, \mathbb{I} \in [\mathbb{T}_{\infty}] : (\tau, \mathbb{I}) \in \mathcal{C} \Rightarrow (\tau, \mathrm{glb}\,\mathbb{I}) \models \Box_{[0,\#\mathbb{I})}\, r.$$

We call $\mathcal{C}$ non-trivial *context for* $Spec$ *if and only if* $\exists (\tau, \mathbb{I}) \in \mathcal{C} : \tau \in Spec \wedge \mathbb{I} \neq \emptyset$. ◇

The context $\mathcal{C}(Spec) \stackrel{\text{def}}{=} \{(\tau, \phi(\tau)) \mid \tau \in Spec\}$ contains the whole specification $Spec$ and any property fulfilled by $Spec$ in $\mathcal{C}(Spec)$ is the invariant of this specification. In an empty context every property becomes a tautology, i.e., it holds $\models_{\emptyset} r$ for any r. Please also note that the second parameter of the context defines the fulfillment scope of some requirement r in *absolute* values, while the parameters of timed operators in r define the scope relative to the current evaluation point.

Finally, we note that from Property (2.1d) in Table 2.4 (page 30) follows: the set of all satisfiable properties in some context $\mathcal{C}$ does not change if we replace $(\tau, \mathbb{I}_1) \in \mathcal{C}$ and $(\tau, \mathbb{I}_2) \in \mathcal{C}$ with $\mathbb{I}_1 \cap \mathbb{I}_2 \neq \emptyset$ by $(\tau, \mathbb{I}_1 \cup \mathbb{I}_2)$. The converse replacement is also possible. By this, we obtain an equivalence relation on contexts.

Context Construction

Technically, a context of some requirement r can be derived from three further MTL properties ψ_1, ψ_2, and ψ_3, whereas ψ_1 marks the beginning and ψ_2 the end points of the intervals in $\mathcal{C}$; ψ_3 describes the properties of the interval in-between. Formally:

$$\mathcal{C}(\psi_1, \psi_2, \psi_3) \stackrel{\text{def}}{=} \{(\tau, \mathbb{I}) \mid \quad (\tau, \mathrm{glb}\,\mathbb{I}) \models \psi_1 \wedge (\mathrm{lub}\,\mathbb{I} < \infty \Rightarrow (\tau, \mathrm{lub}\,\mathbb{I}) \models \psi_2) \\ \wedge (\tau, \mathrm{glb}\,\mathbb{I}) \models \Box_{(0,\#\mathbb{I})}\, \psi_3\}.$$

Then, r put in its context yields the property $\psi_1 \rightarrow (r\,\mathrm{U}\,\psi_2 \vee \Box(r \wedge \psi_3))$. For instance, provided $\models \psi_3 \leftrightarrow \neg\psi_2$ holds, the term on the right-hand side of the implication is known under the name *weak until* or *unless*. It means that r holds for as long as ψ_2 does not hold, even forever if necessary. Five context types which are commonly used in requirements specifications are described in [DAC98]. Table 2.5 demonstrates that they all can be formalized using the scheme from above. Every line of the table contains an informal description along with the parameter values for the corresponding $\mathcal{C}(\psi_1, \psi_2, \psi_3)$-context. By changing the interval bounds, in which ψ_3 must hold, e.g., to $[0, \#\mathbb{I})$, $(0, \#\mathbb{I}]$, or $[0, \#\mathbb{I}]$, we obtain further useful context types.

	Parameter		
Context Type	ψ_1	ψ_2	ψ_3
"globally"	tt	ff	tt
"before A"	tt	A	ff
"after A"	A	ff	tt
"between A_1 and A_2"	A_1	A_2	ff
"after A_1 until A_2"	A_1	A_2	$\neg A_2$

Table 2.5: $\mathcal{C}(\psi_1,\psi_2,\psi_3)$*-Formalization of Context Types from [DAC98]*

Context Combination

Within the settings of our formal trace-based framework, specifications are build by a number of requirement/context pairs. Thus, the scope of a particular requirement is bounded by its context. In order to be able to reason about consistency and completeness of specifications we need to identify situations where the individual requirements can potentially interfere. These are exactly all points in time in which their contexts intersect. In these time points the conjunction of requirements must hold. In inconsistent specifications this may result in unsatisfiable formulas. Finally, this conjunction can also be used as the premise in the derivation process of further implicit requirements. In the rest of this section we introduce the notion of *context combination* which answers the question, given two requirement/context pairs $(r_1,\mathcal{C}_1)$ and $(r_2,\mathcal{C}_2)$, in which context holds their composition $r_1 \wedge r_2$?

Definition 2.5 (Combination of contexts)**.** *Given a specification* $Spec \subseteq \vec{C}^\kappa$*, and a pair of contexts* $\mathcal{C}_1,\mathcal{C}_2 \subseteq \vec{C}^\kappa \times [\mathbb{T}_\infty]$*, The* combined *context of* $\mathcal{C}_1$ *and* $\mathcal{C}_2$ *is denoted by* $\mathcal{C}_1 \oslash \mathcal{C}_2$ *and defined as*

$$\mathcal{C}_1 \oslash \mathcal{C}_2 \stackrel{\text{def}}{=} \{(\tau,\mathbb{I}_1 \cap \mathbb{I}_2) \mid (\tau,\mathbb{I}_1) \in \mathcal{C}_1 \wedge (\tau,\mathbb{I}_2) \in \mathcal{C}_2\}. \qquad \diamond$$

The context combination is associative, commutative and idempotent. The next proposition demonstrates the correctness of the above definition: it states that if a specification satisfies two requirements within their respective contexts than it also satisfies the conjunction of these requirements within the combined context.

Proposition 2.1 (Composition of requirement/context pairs)**.** *Given a specification* $Spec \subseteq \vec{C}^\kappa$*, two requirements* $r_1,r_2 \subseteq \vec{C} \times \mathbb{T}$*, and a pair of contexts* $\mathcal{C}_1,\mathcal{C}_2 \subseteq \vec{C}^\kappa \times [\mathbb{T}_\infty]$*, then*

$$((Spec \models_{\mathcal{C}_1} r_1) \wedge (Spec \models_{\mathcal{C}_2} r_2)) \Rightarrow Spec \models_{\mathcal{C}_1 \oslash \mathcal{C}_2} r_1 \wedge r_2.$$

Proof. For a trace/time point pair (τ,t) such that

$$\tau \in Spec, \qquad (\tau,t) \models_{\mathcal{C}_1} r_1, \qquad \text{and} \qquad (\tau,t) \models_{\mathcal{C}_2} r_2$$

holds, there must exist intervals $\mathbb{I}_1,\mathbb{I}_2$ such that

$$(\tau,\mathbb{I}_1) \in \mathcal{C}_1, \qquad (\tau,\mathbb{I}_2) \in \mathcal{C}_2, \qquad \text{and} \qquad t \in \mathbb{I}_1 \cap \mathbb{I}_2.$$

Finally by the definition of the context combination: $(\tau,\mathbb{I}_1 \cap \mathbb{I}_2) \in \mathcal{C}_1 \oslash \mathcal{C}_2$. □

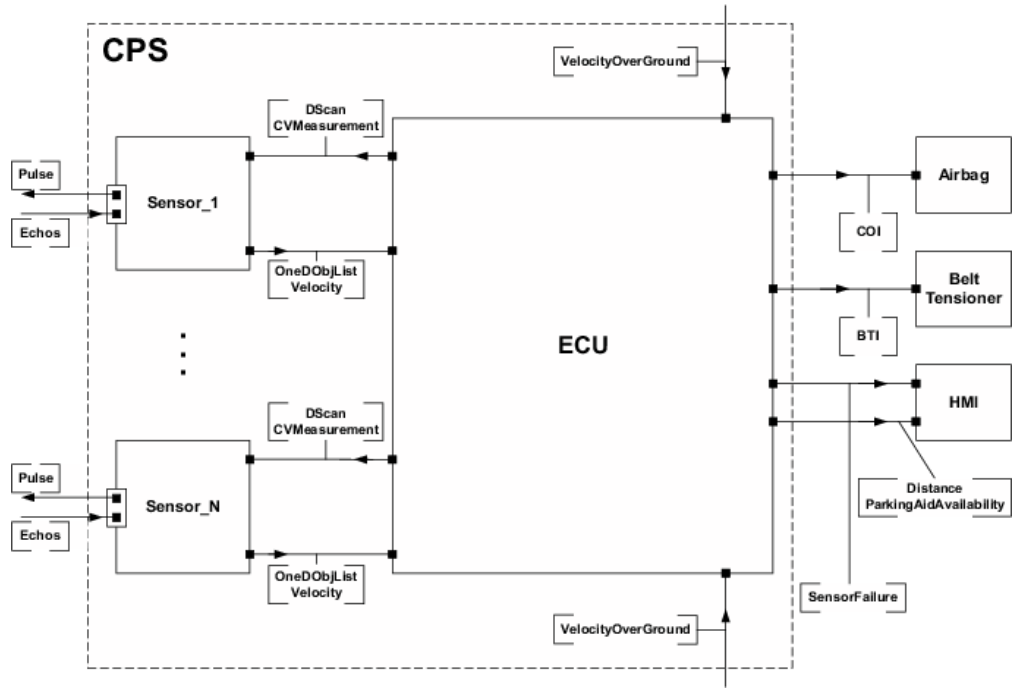


Figure 2.4: *Structure of the CPS System from [KR02]*

2.4 CPS Case Study (*con'ed*)

The specification of the CPS includes a number of different constraint types presented above. We have already met two of them in Section 1.4. They clearly belong to response time constraints. These and further requirements are formalized in the current section using our trace-based specification framework. We denote events for the activation of the airbag and belt tensioner through the COI and BTI interface, respectively, by A_{COI} and A_{BTI}, respectively. Now we can state both time requirements from the list above formally, as

$$r_{\mathrm{R1}} \stackrel{\text{def}}{\equiv} r(A_{COI}, e_{PC}^{\exists\uparrow}, [10, \infty)) \qquad \text{and} \qquad r_{\mathrm{R2}} \stackrel{\text{def}}{\equiv} r(A_{BTI}, e_{PC}^{\exists\uparrow}, [110, \infty)).$$

We immediately see that these constraints are insufficient: A criterion is missing, which states when a situation becomes critical. The intervals of the response time requirements are unbounded on the right side and allow the CPS system to activate airbag and belt tensioner, as soon as it registers an approaching object. On the other hand, since the sense of sight of the sensors is finite (bounded to 7 m in our case), there indeed exists an upper bound for the response time requirements from above: It is the maximal measurement distance $d_{s\_max} = 7$ m divided by the minimal measurable positive relative velocity v_{s_min}, minus a minimal latency ϵ_{l_min} between

the discovery of an object by the sensors and issuing of the COI and BTI signals by the CPS. Thus, we can reformulate the requirements from above, as

$$r_{\mathrm{R1}} \stackrel{\text{def}}{\equiv} r(A_{COI}, e_{PC}^{\exists\uparrow}, \left[10, \frac{d_{s\_max}}{v_{s\_min}} - \epsilon_{l\_min}\right)) \quad \text{and} \quad r_{\mathrm{R2}} \stackrel{\text{def}}{\equiv} r(A_{BTI}, e_{PC}^{\exists\uparrow}, \left[110, \frac{d_{s\_max}}{v_{s\_min}} - \epsilon_{l\_min}\right)).$$

This upper bound for the system response is further refined by additional requirements introduced below.

Next, we zoom to a more detailed view on the CPS system. It is depicted in Figure 2.4. The system consists of one central electronic processing unit (ECU) and up to six sensors connected to it via designated direct wires. The sensors can operate in two different modes: DMode and CVMode. Additionally, DMode has two sub-modes with different supervision range. It can amount either to 2 or 7 m. In the CVMode the system checks for approaching objects in the range from 1.41 to 0.69 m (see below). The modes and transitions between them are depicted in Figure 2.5. They are triggered by the ECU, which can issue DScan and CVMeasurement commands. The operating modes reflect the criticality of the situation in the environment of the vehicle. DMode is the normal-case behavior of the system. When an object approaches too fast and/or enters a critical region, the systems switches to CVMode. In the CVMode (i.e., as the response to CVMeasurement command) sensors produce a series of measurements. If no crash has occurred, the system switches back to DMode. Immediately after switching back one short range scan (2 m) is performed in order to ensure that another objects closely following the originally tracked one are detected quickly. Afterwards, the system proceeds with the normal 7-m-distance scans.

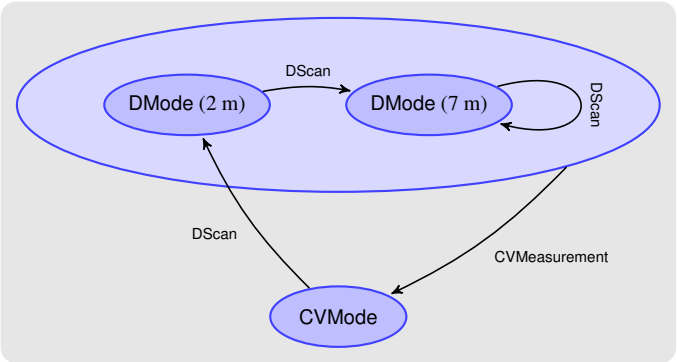


Figure 2.5: *Operational Mode Transition Diagram of CPS*

The operational modes build the contexts of different time requirements put on the CPS system. DMode and CVMode are triggered by events from

$$\mathcal{A}_{\mathsf{Mode}} \stackrel{\text{def}}{=} \{\mathsf{DScan(2)}, \mathsf{DScan(7)}, \mathsf{CVMeasurement}\} \times \mathbb{R}_{\geq 0}.$$

For example, in a given run starting from $(\langle \mathsf{DScan(2)} \rangle, t_1)$-message and until the first $(\langle \mathsf{CVMeasurement} \rangle, t_2)$-message $(t_1 < t_2)$ the system remains in DMode mode. $\mathcal{A}_{\mathsf{Mode}}$ is partitioned into two (disjoint) subsets

$$A_{\mathsf{DScan}} \stackrel{\text{def}}{=} \{\mathsf{DScan(2)}, \mathsf{DScan(7)}\} \times \mathbb{R}_{\geq 0} \quad \text{and} \quad A_{\mathsf{CVMeasurement}} \stackrel{\text{def}}{=} \{\mathsf{CVMeasurement}\} \times \mathbb{R}_{\geq 0},$$

which contain switches to DMode and CVMode, respectively.

We construct contexts $\mathcal{C}_{\text{DMode}}$ and $\mathcal{C}_{\text{CVMode}}$ using the scheme presented at the end of Section 2.3.3 (on page 42), i.e.,

$$\begin{aligned}\mathcal{C}_{\text{DMode}} \stackrel{\text{def}}{=} \{(\tau,\mathbb{I}) \mid\ & (\tau,\text{glb}\,\mathbb{I}) \models A_{\text{DScan(2)}} \wedge (\text{lub}\,\mathbb{I} < \infty \Rightarrow (\tau,\text{lub}\,\mathbb{I}) \models A_{\text{CVMeasurement}}) \\ & \wedge (\tau,\text{glb}\,\mathbb{I}) \models \Box_{(0,\#\mathbb{I})}\neg A_{\text{CVMeasurement}}\},\\ \mathcal{C}_{\text{CVMode}} \stackrel{\text{def}}{=} \{(\tau,\mathbb{I}) \mid\ & (\tau,\text{glb}\,\mathbb{I}) \models A_{\text{CVMeasurement}} \wedge (\text{lub}\,\mathbb{I} < \infty \Rightarrow (\tau,\text{lub}\,\mathbb{I}) \models A_{\text{DScan}}) \\ & \wedge (\tau,\text{glb}\,\mathbb{I}) \models \Box_{(0,\#\mathbb{I})}\neg A_{\text{DScan}}\},\end{aligned}$$

whereas $A_{\text{DScan(2)}}$ contains 2-m-distance scans only. By this, the pure functional requirements on modes and mode changes illustrated in Figure 2.5 are incorporated into contexts of time requirements which must hold in these modes. These requirements grouped by their respective embedding mode context are described and formalized next.

DMode Requirements

Following requirements have to hold in DMode.

(R3) As the response on the DScan-command, the sensor sends information about at most eight objects observed in the environment denoted as the OneDObjList-response in Figure 2.4. The response time on the DScan-command must lie between 3 and 4.5 ms.

(R4) In the normal case, the ECU is operating sensors in DMode mode by sending out DScan commands *every* 15 *ms* to all sensors.

Both formal requirements presented below must hold in the context $\mathcal{C}_{\text{DScan}}$. The formal model for the first requirement is the following response-time constraint

$$r_{\text{R3}} \stackrel{\text{def}}{\equiv} r_u(A_{\text{DScan}}, A_{obj}, [3, 4.5]),$$

where $A_{obj} \stackrel{\text{def}}{=} M_{obj}^{[0,8]} \times \mathbb{R}_{\geq 0}$ is the set of timed observations of at most eight objects, i.e., $M_{obj}^{[0,8]} \stackrel{\text{def}}{=} \{\sigma \in M_{obj}^{*} \mid \#\sigma \leq 8\}$.

The requirement R4 is formalized using a periodic inter-arrival time constraint:

$$r_{\text{R4}} \stackrel{\text{def}}{\equiv} i_p(A_{\text{DScan}}, [15, 15]).$$

CVMode Requirements

The Velocity-message on the data-flow link from sensor to ECU depicted in Figure 2.4 refers to the CVMode-behavior. In the CVMode the sensor starts to deliver a time series of velocity measurements, with the maximal length of ten. A series lasts as long as an object steadily approaches the vehicle and is observed within specific distance ranges. The CPS specification subdivides the distance of 1.41 m before the vehicle in 9 disjoint, decreasing segments (distance ranges) of equal length. Every measurement in the series is associated with a distance range

in which an object should be located. If no object is found in this area a zero is sent to ECU and the measurement sequence terminates. Following two additional constraints supplement the specification of the time behavior of the CPS system in the CVMode.

(R5) When an object on collision course needs *less than* 30 *ms* before it will undercut the 1.41-m distance, ECU must start to operate sensors in CVMode.

(R6) The measurement sequence in the CVMode is terminated by 0. It is terminated either after nine "real" velocity measurements, or if no object enters the next distance range within 25 ms.

The below formalization of requirements assumes $\mathcal{C}_{\mathsf{CVMode}}$ as context. The first requirement is formalized as follows: Using the same scheme as for the definition of the e_{PC}-observation, we define the observation $e_{1.41\mathrm{m}}$ which captures observations starting with an object-measurement event $A_{obj} \in M^*_{obj} \times \mathbb{R}_{\geq 0}$ and ending at time of predicted passing through of object's registered in A_{obj} the 1.41-m mark. Formally,

$$e_{1.41\mathrm{m}} \stackrel{\text{def}}{=} \{\tau \mid pre\_crash(\tau.(\min \phi(\tau))) = (\max \phi(\tau))\},$$

where $pre\_crash\colon M^*_{obj} \mapsto \mathbb{R}_{\geq 0}$ is defined as

$$pre\_crash(\langle\rangle) \stackrel{\text{def}}{=} \infty,$$

$$pre\_crash(\langle m^{(1)}_{obj} \ldots m^{(n)}_{obj}\rangle) \stackrel{\text{def}}{=} \min_{1\leq i\leq n} \begin{cases} \frac{m^{(i)}_{obj}.d - 1.41}{m^{(i)}_{obj}.v \cdot \cos m^{(i)}_{obj}.\theta} & \text{if } m^{(i)}_{obj}.\theta < 90,\ m^{(i)}_{obj}.v \neq 0, \text{ and} \\ & m^{(i)}_{obj}.d \geq 1.41 \\ \infty, & \text{o.w.} \end{cases}$$

Then,

$$r_{\mathrm{R5}} \stackrel{\text{def}}{\equiv} r_u(A_{\mathsf{CVMeasurement}}, e^{\exists\uparrow}_{1.41\mathrm{m}}, (0, 30)).$$

For the formalization of the second requirement form the above list define the set of velocity measurement events $A_{vel} \stackrel{\text{def}}{=} \mathbb{Q}^1_+ \times \mathbb{R}_{\geq 0}$, where $\mathbb{Q}^1_+$ stands for a set of positive rational-number sequences of length one: $\mathbb{Q}^1_+ \stackrel{\text{def}}{=} \{\langle q\rangle \mid q \in \mathbb{Q}_+\}$. Further on, let $A_{vel}(0) \stackrel{\text{def}}{=} \{0\}^1 \times \mathbb{R}_{\geq 0}$ be the terminating 0-velocity event. Then, the second time requirement in the CVMode can be modeled using a timeout constraint:

$$r_{\mathrm{R6}} \stackrel{\text{def}}{\equiv} tmo(A_{vel}, A_{vel}, A_{vel}(0), 25).$$

2.5 Related Work

The present chapter contributed to several topics of software engineering and formal methods research, namely formal models for real-time and reactive behavior, formalization of requirements specifications, as well as cataloging of real-time requirements. The obtained results are compared with the existing works grouped by the respective topic below.

Specification Formalisms for Reactivity and Real-Time

Over the years, a variety of formalisms for the specification of reactive and real-time systems were proposed (cf. [Car02, Jos01] for surveys on this topic). Among them temporal logic-based approaches play a prominent role. This is explained by the fact that a number of important causal properties (cf. [MP95, MP92]) can be concisely formulated and efficiently verified (cf. [CGP99]) using temporal logic. In the last years, temporal logic has found wide acceptance for specifying and verifying real-life hardware and software systems.

The point-based quantitative temporal logic MTL [Koy90] was extended by durations in [LH95]. However, that extension does not allow to relate intervals to each other, its propose is rather to express that some property holds for a certain time. This makes the logic from [LH95] applicable for the specification of hybrid systems, but does not facilitates the event-based specification which is commonly used to reason about the behavior of reactive real-time systems without analog parts.

Our notion of intervals corresponds to the ideas incorporated in Real-Time Logic (RTL) [JMS88]. In RTL, using a special notation, the beginning and the end of non-instantaneous observations can be explicitly referenced in the formulas. "The main problem with RTL is the fact that absolute system time is referenced, with a low level of abstraction, leading to very complex formulas required to describe the system" [BMN00].

The duration, or interval concept of [LH95] is similar to the one of Duration Calculus (DC) [ZCA91]. DC is a requirements specification language that allows concise expression of properties about duration of signals and a calculus to reason about these durations. DC allows the expression of safety properties only, and there is hardly any program verification theory based on it [LH95]. Also, as argued above the concept of duration used in DC is not appropriate to reason about the systems from the semantic domain we are dealing with.

Further important class of formal specification approaches build the automata-based ones. In the context of requirements engineering it is rather questionable if an operational, implementation-close formalism is an adequate means to document requirements. However, analytical models based on automata are an effective instrument for the analysis and the validation (by simulation) of specifications. The automata-based approaches which are conceptually close to the results presented here are the models of timed automata [AD94] and timed I/O automata [KLSV06]. The relation to these models and our approach resembles the relation between sequence charts and finite automata (e.g., cf. [Krü00]): They provide complementary views on the system and are appropriate to reasoning about the system behavior at different stages of development.

Formal Foundation in Requirements Engineering

The definition of formal semantics for requirement specifications is not new. This idea goes back to the *4-variable model* from [PM95] that proposes to specify a system in a black-box manner by means of mathematical functions. These functions describe the relations between variables of the considered system and its environment. The functions have real-valued domains and ranges and, thus, the approach is capable to specify the real-time behavior. In contrast to

our work, this approach offers neither a method to specify single requirements modularly nor to refine, or decompose them[8] and therefore it is not able to support the scalability necessary for the state-of-the-art systems and cannot be integrated into a model-based development process.

A reference model for requirements and specifications called *functional documentation model* presented in [GGJZ00] is based on the Church's higher order logic. Analogously to the 4-variable model, it subdivides the requirements specification into four types of *phenomena*. They can be either system or environment controlled and either visible or invisible to the system/environment. The deficits of the 4-variable-model approach also apply here: the authors do not provide the concepts of modularization and refinement.

The SCR* method and tool set [HBGL95] can be seen as a variant of the 4-variable model. It proposes to model the system specification as an automaton which reads and writes variables of the four types identified in [PM95]. The deficits of SCR* are its a-priori discretized system step and a rather coarse-grained modularity concept: Every subsystem has to be completely described by three tables: condition table, event table, and mode-transition table. In [Hei95] SCR* was extended by the notion of real-time. This was done in an ad-hoc manner by associating every event with a time stamp (similar to the timed-word model of timed automata). No concepts for the reasoning about durations or non-instantaneous composite observations were provided.

A number of approaches which extract (formal) models out of specifications mainly deal with functional requirements. Approaches, like KAOS [vL03] or Albert II [HD98], address the time requirements as annotations or as a temporal logic formulas, operating in terms of logical system steps. We have already discussed in Section 2.2.3 that such an a-priori discretization cannot be justified offhand in the early requirements engineering phases. In contrast, the presented approach allows the system engineer to capture and model the timing information explicitly as well as to relate it to design and implementation artifacts, like components, channels, schedules, tasks, *etc.*

Classification of Real-Time Requirements

Over the years, different approaches to classification of system properties were proposed in the literature. They can be roughly subdivided into three categories:

(1) Verification-oriented classifications distinguish properties according to the different proof methodologies, which are needed to reason about their fulfillment. An important example is liveness/safety distinction presented first in [Lam77].

(2) Another category build classifications which are guided by the capabilities of the applied specification formalism. For example, [MP95] presents a taxonomy of properties which is based on the syntactic structure of LTL and MTL.

(3) The last category comprises approaches which gather and order patterns frequently occurring in practice. This is also the origin of the taxonomy form Section 2.3.3. Thus, we compare our contribution with other approaches from this category below.

[8] Refinement and decomposition for our framework are introduced in the next chapter.

[KC05] presents a repository of specification patterns for real-time requirements. The pattern template is borrowed from [DAC98], where a system of specification patterns for temporal (untimed) system properties is presented. A pattern from [KC05] includes amongst others

- the context of the property (one of five from Table 2.5, page 44),
- formalizations of the property in three different quantitative temporal logics, and
- its textual description in structured English.

All three temporal logics used in [KC05] are propositional and, thus, properties which can be documented by that approach are restricted to the subset expressible using propositional interval- and state-oriented temporal logics. By this, a number of important requirement types, like time-out, synchrony, offset, or recurrence specified by a period/jitter pair are missing in the catalog of [KC05]. A further unclear aspect of that approach is the co-action of several property specifications, or, more generally, the relation between patterns and other system artifacts, like components, classes, *etc.*

In [LvL02] the KAOS approach is extended by a taxonomy of goal patterns for the *goal-oriented* specification. Goal-oriented requirements engineering refers to the use of *goals* during this phase of development. Goals are objectives the system under development must achieve. Goals are formulated in terms of statements which may refer to functional or non-functional properties and range from high-level concerns to lower-level ones. The goal patterns presented in [LvL02] are clearly too low-level: they correspond rather to our building blocks presented in Sections 2.3.1 and 2.3.2. The approach also does not support embedding of requirements into different contexts.

2.6 Summary

We have presented a mathematical framework for specification and analysis of time requirements of reactive systems. The framework provides a possibility to precisely describe the desired time behavior and formally integrate it within a formal specification. We have based our treatments on a rigorous system model that captures the communication flow observed on the system border through channels over time. The model can be parametrized by different discrete or continuous time domains. This system model is adequate for the black-box specification of all kinds of reactive (real-time) systems.

Within the system model we have introduced the notion of timed events and observations as the central modeling primitives which are used to reason about reactive real-time systems. These primitives served as a basis for the definition of the semantics of an MTL derivative – a quantitative first-order interval-based linear temporal logic. Using this logic different causal and timing relationships between timed observations were formalized.

The above contributions coalesced into a taxonomy of low-level time requirements. It allows to work with the primitives of timed behavior specification used on the level of technical systems. These primitives, in particular, end-to-end latencies (response times) and periods, also often

appear in specification documents. Hence, the members of this taxonomy can be used as building blocks for the creation of formal system specifications at different stages of development.

The process of formalization using predefined specification patterns is also facilitated by the notion of timed behavior specification introduced in the current chapter. The specification embeds its time requirements into associated contexts. This allows to specify real-time behavior during distinguished periods of system execution and to relate time behavior with further system requirements. In particular we have shown how contexts can be constructed using further MTL constraints and how a common context of two requirement/context-pairs, i.e., the context in which both requirements must hold, can be calculated.

This treatment of time behavior prepares the ground for the investigation of the methodical usage of time requirements and specifications in the subsequent chapters of this thesis. In particular, it allows us to study refinement notions for time requirements, to formalize consistency and completeness conditions on collections of requirements, and to guide the transition from "unstructured", global black-box view on the system to individual component specifications.

3 Analysis of Time Constraints

In the previous chapter we have defined the notions of time requirements and timed behavior specification using a formal trace-based framework. In the current chapter, on the basis of this framework, we provide a number of formally founded analysis and transformation methods which aim at ensuring the correctness and increasing the quality of specification documents and other artifacts created during the system development. These methods comprise the refinement of time behavior, formal consistency and completeness checks, and correct distribution of time requirements over component networks. Finally, we introduce the notion of structured timed behavior specification, and formally relate the behavioral and structural views on the system.

Contents

AT THE MINIMUM, writing formal specifications helps engineers to clearly define their problems, goals, and solutions. The clarity that this activity produces is a benefit in itself. However, the ultimate purpose of formal models is the application of formal analysis techniques which promises even greater evidence of the correctness of specification documents as well as of the whole system. Generally, the correctness refers to the consistency and completeness of the specification itself and to the conformance of system's design and implementation to its specification. These both types of correctness are the central concerns of the current chapter. The general analysis and transformation methods for time behavior presented here are instantiated for the members of the time-requirements taxonomy from the last chapter.

Contributions and Outline. During the development time specifications are involved into different quality and conformance assurance activities. For these purposes Section 3.1 offers a number of methods and mechanisms. In particular, it deals

- with formulation of consistency constraints between time requirements,
- with distribution of time requirements over different design and implementation artifacts,
- with derivation of new time requirements from the set of existing ones, as well as
- with the refinement of time requirements and specifications.

All these operations are formally founded and designed to facilitate the construction of reactive systems with the time behavior which satisfies the specification and meets the actual stakeholders' needs.

Section 3.2 treats the transition from unstructured to structured specifications. The universe of timed traces defines the semantic domain of our denotational system model. While the unstructured specifications can be seen as a conjunction of constraints or predicates over this universe, the structured specifications consist of a number of interconnected components which make assumptions about the timed traces they obtain as inputs and guarantee certain timed traces as the corresponding outputs. Structured specifications impose an asymmetry in the treatment of input and output traces. For the formal description of structured specifications we use the FOCUS formalism [BS01a] carried over to our trace-based domain. The relationship between unstructured and structured specifications is defined formally at the end of Section 3.2. This relationship permits the independent, but yet tightly integrated and, by this, property preserving development of both structured component specification and (unstructured) timed behavior specification.

3.1 Quality and Conformance Assurance During Development

In the previous chapter we have defined the notion of individual time constraints and embedded them into functional contexts of formal requirements specifications. The requirements specification and, in particular, time constraints are involved into different activities carried out during the development process. These activities are mostly (1) the quality assurance measures, which analyze the specifications regarding their consistency and completeness, as well as (2) the conformance checks of artifacts created in the subsequent development phases, e.g., does the implementation satisfies the specification. Further important class of activities aims at deriving

requirements for particular subsystems or components. Thereby, the task of the system engineer is to formulate requirements on the individual components of the system in a way which guarantees the conformance of the whole system to the original requirements specification.

The current section delivers the formal foundation for the aforementioned activities. Section 3.1.1 introduces refinement as a notion of conformance in the development process. In particular, for our time constraints from previous section we specify, what changes in the time behavior are valid refinement steps. Section 3.1.2 deals with the two main quality attributes of specifications: consistency and completeness. Thereby, the emphasis lies on the derivation of further constrains from a set of existing ones. Finally, Section 3.1.3 describes methods for the decomposition of time constraints. By this, we are able to distribute time constrains over system components. In that section we also investigate when different time requirements influence each other. In other words, we decompose requirement specification into change-insensitive parts. This is of especial interest for the bounding of verification and validation efforts – in the case of a change in the specification only dependent requirements have to be reconsidered.

3.1.1 Refinement

The notion of *refinement* captures formally the dependency between different artifacts originating from a top-down development process. The main idea behind refinement is the consideration of the system development as a restriction of the set of possible satisfactory solutions (permitted by the specification) until only one solution (implementation) is left. From a more bottom-up perspective, refinement is the relation in which a valid design or implementation must stand to the specification of the system, i.e.,– their formal acceptance criterion. The next definition introduces abstraction/refinement relation for specifications, individual requirements, and contexts.

Definition 3.1 (Refinement of time properties and specifications)**.** *Given two specifications* $Spec_0, Spec_1 \subseteq \vec{C}^\kappa$*, then* $Spec_1$ refines $Spec_0$ *or* $Spec_0$ abstracts $Spec_1$*, denoted by* $Spec_1 \rightsquigarrow Spec_0$ *if and only if* $Spec_1 \subseteq Spec_0$*.*

Given two time requirements $r_0, r_1 \subseteq \vec{C} \times \mathbb{T}$*, then* r_1 refines r_0 *or* r_0 abstracts r_1*, denoted by* $r_1 \rightsquigarrow r_0$ *if and only if* $r_1 \subseteq r_0$*.*

Given contexts $\mathcal{C}_0, \mathcal{C}_1 \subseteq \vec{C}^\kappa \times [\mathbb{T}_\infty]$*, then* $\mathcal{C}_1$ refines $\mathcal{C}_0$ *or* $\mathcal{C}_0$ abstracts $\mathcal{C}_1$*, denoted by* $\mathcal{C}_1 \preceq \mathcal{C}_0$ *if and only if*

$$\forall (\tau, \mathbb{I}) \in \mathcal{C}_1 : \forall t \in \mathbb{I} : \exists (\tau, \mathbb{J}) \in \mathcal{C}_0 : t \in \mathbb{J}. \qquad \diamond$$

The refinement notions of specifications and time requirements inherit reflexivity, transitivity, and antisymmetry directly from set inclusion. By refinement of contexts we decrease the length of time spans and/or the amount of observations in which some time requirement must hold. Next, we prove that the fulfillment of a requirement/context-pair is monotone respective specification refinement.

Corollary 3.1 (Property preservation during specification refinement)**.** *Given two specifications* $Spec_0, Spec_1 \subseteq \vec{C}^\kappa$*, a requirement* $r \subseteq \vec{C} \times \mathbb{T}$*, and a context* $\mathcal{C} \subseteq \vec{C}^\kappa \times [\mathbb{T}_\infty]$*. If* $Spec_0 \models_{\mathcal{C}} r$ *and* $Spec_1 \rightsquigarrow Spec_0$*, then*

$$Spec_1 \models_{\mathcal{C}} r.$$

Proof. Follows directly from the subset relation between $Spec_1$ and $Spec_0$. □

On the level of MTL formulas the refinement relation between a pair of requirements corresponds to an implication. Formally, given a pair of time requirements r_0 and r_1 which are formulated using the MTL formulas φ_0 and φ_1, respectively, then $r_1 \rightsquigarrow r_0$ if and only if $\models \varphi_1 \rightarrow \varphi_0$, i.e., if the MTL formula $\varphi_1 \rightarrow \varphi_0$ is a tautology. Thus, in order to improve readability we will abbreviate the above relation on MTL formulas by $\varphi_1 \rightsquigarrow \varphi_0$.

Refinement of Time

As usual, our emphasis lies on the refinement of time behavior. It can be restricted by changing the interval parameters of timed modalities. In fact, this is an important class of refinement relations in the development of real-time systems: In practice, it is mostly impossible to enforce the time behavior in a top-down manner – the time behavior of the implementation will almost surely deviate from the specified one. Thus, it is important to determine exactly when the actual time behavior satisfies the time specification. The following rules define when a change in time parameters is a sound refinement step in *all* possible contexts. These rules can be used for bottom-up conformance checks during development.

Lemma 3.1 (Interval refinement in MTL)**.** *Given a pair of formulas φ_1, φ_0 such that $\varphi_1 \rightsquigarrow \varphi_0$ and two intervals $\mathbb{I}, \mathbb{J} \in [\mathbb{T}_\infty]$ such that $\mathbb{I} \subseteq \mathbb{J}$, then the following relations hold*

$$\begin{aligned} \boldsymbol{X}_{\mathbb{I}}\varphi_0 &\leftsquigarrow \boldsymbol{X}_{\mathbb{J}}\varphi_1, \quad \textit{where } \boldsymbol{X} \in \{\Box, \blacksquare\}, \\ \boldsymbol{Y}_{\mathbb{I}}\varphi_1 &\rightsquigarrow \boldsymbol{Y}_{\mathbb{J}}\varphi_0, \quad \textit{where } \boldsymbol{Y} \in \{\Diamond, \blacklozenge\}. \end{aligned}$$

Proof. Here, we prove the lemma for the future modalities only. Proofs for the past modalities go analogously.

From the predicate calculus we know that $\models \psi_1 \vee \psi_2 \leftarrow \psi_1$ and $\models \psi_1 \wedge \psi_2 \rightarrow \psi_1$ and by this $\psi_1 \vee \psi_2 \leftsquigarrow \psi_1$ and $\psi_1 \wedge \psi_2 \rightsquigarrow \psi_1$. Further on, let $\mathbb{I}_1, \mathbb{I}_2 \in [\mathbb{T}_\infty]$ be intervals such that $\mathbb{I}_1 \cup \mathbb{I}_2 = \mathbb{J} \setminus \mathbb{I}$. Then, using Property (2.1d) from Table 2.4 (page 30) we obtain

$$\Box_{\mathbb{J}}\varphi_1 \equiv \Box_{\mathbb{I}_1}\varphi_1 \wedge \Box_{\mathbb{I}}\varphi_1 \wedge \Box_{\mathbb{I}_2}\varphi_1.$$

Due to Property (2.1c) from the same table, we have

$$\Diamond_{\mathbb{I}_1}\varphi_0 \vee \Diamond_{\mathbb{I}}\varphi_0 \vee \Diamond_{\mathbb{I}_2}\varphi_0 \equiv \Diamond_{\mathbb{J}}\varphi_0.$$

Next we need to show that $\Box_{\mathbb{I}}\varphi_1 \rightsquigarrow \Box_{\mathbb{I}}\varphi_0$ and $\Diamond_{\mathbb{J}}\varphi_0 \rightsquigarrow \Diamond_{\mathbb{J}}\varphi_1$ hold. The semantics of the formula $\Box_{\mathbb{I}}\varphi_1$ is the set

$$\{(\tau, t) \mid \forall t' \in \phi(\tau) \cap (t + \mathbb{I}) : (\tau, t') \models \varphi_1\}.$$

Traces with time points which satisfy $\Diamond_{\mathbb{J}}\varphi_0$ are gathered by the set

$$\{(\tau, t) \mid \exists t' \in \phi(\tau) \cap (t + \mathbb{J}) : (\tau, t') \models \varphi_0\}.$$

We obtain set inclusion in both cases and due to the transitivity of the refinement relation we are done. □

Based on Lemma 3.1 we define refinement relations between different constraints introduced in Section 2.3. This allows us to justify formally when relaxing or tightening the time budgets of real-time requirements (i.e., their time-interval parameters) results in a refinement step. Tables 3.1 and 3.2 summarize the results proved by the next propositions. Every table line contains a pair of constraints of the same type which stand in the abstraction/refinement relation to each other provided the corresponding condition shown in the last column is fulfilled.

Nr.	Abstraction	Refinement	Condition
1	$cres(\varphi_1,\varphi_2,d_1)$	$cres(\varphi_1,\varphi_2,d_2)$	$d_1 \geq d_2$
2	$br(\varphi_1,\varphi_2,\mathbb{I}_1)$	$br(\varphi_1,\varphi_2,\mathbb{I}_2)$	$\mathbb{I}_1 \supseteq \mathbb{I}_2$
3	$bq(\varphi_1,\varphi_2,\mathbb{I}_1)$	$bq(\varphi_1,\varphi_2,\mathbb{I}_2)$	$\mathbb{I}_1 \supseteq \mathbb{I}_2$
4	$bd(\varphi_1,\varphi_2,\mathbb{I}_1)$	$bd(\varphi_1,\varphi_2,\mathbb{I}_2)$	$\mathbb{I}_1 \supseteq \mathbb{I}_2$
5	$ms^{\omega}(\varphi_1,\varphi_2,t_1)$	$ms^{\omega}(\varphi_1,\varphi_2,t_2)$	$t_1 \leq t_2$
6	$ms(\varphi_1,\varphi_2,t_1)$	$ms(\varphi_1,\varphi_2,t_2)$	$t_1 \leq t_2$
7	$md(\varphi_1,\varphi_2,t_1)$	$md(\varphi_1,\varphi_2,t_2)$	$t_1 \geq t_2$

Table 3.1: *Interval Refinement Rules for Building Blocks*

Lemma 3.2 (Interval refinement of building blocks)**.** *Given two formulas φ_1,φ_2, two intervals $\mathbb{I}_1,\mathbb{I}_2 \in [\mathbb{T}_\infty]$ and $d_i \in \mathbb{T}, t_i \in \mathbb{T}_\infty$ with $i \in \{1,2\}$, for every line of Table 3.1 the expression in the middle column is a refinement of the expression in the left column provided the condition in the right column holds.*

Proof. Table rows 2-4 and 6 follow directly from Lemma 3.1. The remaining three properties are shown next.

$d_1 \geq d_2 \Rightarrow (cres(\varphi_1,\varphi_2,d_2) \rightsquigarrow cres(\varphi_1,\varphi_2,d_1))$: First we observe that $cres(\varphi,\varphi,d)$ is true for any φ and $d \in \mathbb{T}_\infty$. Then, we can rewrite $cres(\varphi_1,\varphi_2,d_2)$ as $cres(\varphi_1,\varphi_2,d_2) \wedge cres(\varphi_2,\varphi_2,d_1-d_2)$ and apply Lemma 2.2.

$t_1 \leq t_2 \Rightarrow (ms^{\omega}(\varphi_1,\varphi_2,t_2) \rightsquigarrow ms^{\omega}(\varphi_1,\varphi_2,t_1))$: In the case $0 < t_1$ the first part of Lemma 3.1 applies. Otherwise, this case follows from the fact that tt is the most abstract formula in MTL.

$t_1 \geq t_2 \Rightarrow (md(\varphi_1,\varphi_2,t_2) \rightsquigarrow md(\varphi_1,\varphi_2,t_1))$: Due to our closed-world semantics for MTL formulas ψ_1, ψ_2 holds $(\psi_1 \rightsquigarrow \psi_2) \Leftrightarrow ((\neg\psi_1) \leftsquigarrow (\neg\psi_2))$. Thus, the property follows from the first part of Lemma 3.1. □

Proposition 3.2 (Interval refinement of time requirements)**.** *Given two formulas φ_1,φ_2, two intervals $\mathbb{I}_1,\mathbb{I}_2 \in [\mathbb{T}_\infty]$, and $p, j_i, l_i \in \mathbb{T}$ with $2j_i < p$ for all $i \in \{1,2\}$, for every line of Table 3.2 the expression in the middle column is a refinement of the expression in the left column provided the condition in the right column holds.*

Proof. The table rows 1-7 and 9-10 are direct consequences of Lemma 3.2. The remaining property is

$$j_1 \geq j_2 \Rightarrow (i_p(\varphi_1,p,j_2) \rightsquigarrow i_p(\varphi_1,p,j_1)).$$

Nr.	Abstraction	Refinement	Condition
1	$r(\varphi_1, \varphi_2, \mathbb{I}_1)$	$r(\varphi_1, \varphi_2, \mathbb{I}_2)$	$\mathbb{I}_1 \supseteq \mathbb{I}_2$
2	$r_u(\varphi_1, \varphi_2, \mathbb{I}_1)$	$r_u(\varphi_1, \varphi_2, \mathbb{I}_2)$	$\mathbb{I}_1 \supseteq \mathbb{I}_2$
3	$q(\varphi_1, \varphi_2, \mathbb{I}_1)$	$q(\varphi_1, \varphi_2, \mathbb{I}_2)$	$\mathbb{I}_1 \supseteq \mathbb{I}_2$
4	$i_p(\varphi_1, \mathbb{I}_1)$	$i_p(\varphi_1, \mathbb{I}_2)$	$\mathbb{I}_1 \supseteq \mathbb{I}_2$
5	$i_s(\varphi_1, \mathbb{I}_1)$	$i_s(\varphi_1, \mathbb{I}_2)$	$\text{glb}\,\mathbb{I}_1 \leq \text{glb}\,\mathbb{I}_2$
6	$i_p(\varphi_1, p, j_1)$	$i_p(\varphi_1, p, j_2)$	$j_1 \geq j_2$
7	$o(\varphi_1, \varphi_2, l_1)$	$o(\varphi_1, \varphi_2, l_2)$	$l_1 \leq l_2$
8	$o_e(\varphi_1, \varphi_2, l_1)$	$o_e(\varphi_1, \varphi_2, l_2)$	$l_1 = l_2$
9	$sc(\varphi_1, \varphi_2, \mathbb{I}_1)$	$sc(\varphi_1, \varphi_2, \mathbb{I}_2)$	$\mathbb{I}_1 \supseteq \mathbb{I}_2$
10	$tmo(\varphi_1, \varphi_2, \varphi_3, d_1)$	$tmo(\varphi_1, \varphi_2, \varphi_3, d_2)$	$d_1 \geq d_2$

Table 3.2: *Interval Refinement Rules for Time Constraints*

Let the premise of the above implication hold and $(\tau, t) \models i_p(\varphi_1, p, j_2)$ for some $\tau \in \vec{C}$ and $t \in \mathbb{T}$. Then, holds

$$
\begin{aligned}
& (\tau, t) \models i_p(\varphi_1, p, j_2) \\
\Leftrightarrow \quad & (* \text{ definition of } i_p \; *) \\
& (\tau, t) \models \blacksquare\{\overline{\tau} \mid |\text{sat}(\varphi_1, \overline{\tau}, \phi(\overline{\tau}))| \in [\max\{0, \left\lfloor \tfrac{\#\overline{\tau} - 2j_2}{p} \right\rfloor\}, \left\lceil \tfrac{\#\overline{\tau} + 2j_2}{p} \right\rceil]\}^{\downarrow} \\
\Leftrightarrow \quad & (* \text{ definition of } \blacksquare, \text{ semantics of } \blacksquare_{(.)}, .^{\downarrow} \; *) \\
& \forall t' \leq t : \tau{\downarrow_{t'}} \in \{\overline{\tau} \mid |\text{sat}(\varphi_1, \overline{\tau}, \phi(\overline{\tau}))| \in [\max\{0, \left\lfloor \tfrac{\#\overline{\tau} - 2j_2}{p} \right\rfloor\}, \left\lceil \tfrac{\#\overline{\tau} + 2j_2}{p} \right\rceil]\}
\end{aligned}
$$

Further on, due to the premise hold

$$
\left\lfloor \frac{\#\tau - 2j_2}{p} \right\rfloor \leq \left\lfloor \frac{\#\tau - 2j_1}{p} \right\rfloor \qquad \text{and} \qquad \left\lceil \frac{\#\tau + 2j_2}{p} \right\rceil \geq \left\lceil \frac{\#\tau + 2j_1}{p} \right\rceil,
$$

and thus,

$$
[\max\{0, \left\lfloor \frac{\#\tau - 2j_2}{p} \right\rfloor\}, \left\lceil \frac{\#\tau + 2j_2}{p} \right\rceil] \subseteq [\max\{0, \left\lfloor \frac{\#\tau - 2j_1}{p} \right\rfloor\}, \left\lceil \frac{\#\tau + 2j_1}{p} \right\rceil].
$$

By this, the set-comprehension condition of the $i_p(\varphi_1, p, j_1)$-constraint is weaker and we can conclude that $(\tau, t) \models i_p(\varphi_1, p, j_1)$. □

During refinement of contexts the number of times a requirement must be fulfilled decreases. By this, the number of different specifications which fulfill a given requirement in some context is increased by context refinement. The relation between the refinement of a time requirement and its fulfillment by a specification is reciprocal: the specification fulfills a refined requirement if it fulfills a more abstract one, i.e., the number of specifications which fulfill a requirement is greater then the number of specifications fulfilling one of its refinements. These relationships between refinements of specifications, contexts, and time requirements are captured formally by the following proposition.

Proposition 3.3 (Property preservation during property/context refinement)**.** *Given a specification* $Spec \subseteq \vec{C}^{\kappa}$*, two requirements* $r_0, r_1 \subseteq \vec{C} \times \mathbb{T}$ *such that* $r_1 \rightsquigarrow r_0$ *and a pair of contexts* $\mathcal{C}_0, \mathcal{C}_1 \subseteq \vec{C}^{\kappa} \times [\mathbb{T}_\infty]$ *such that* $\mathcal{C}_1 \preceq \mathcal{C}_0$*. Then,*

$$Spec \models_{\mathcal{C}_0} r_1 \Rightarrow Spec \models_{\mathcal{C}_1} r_0.$$

Proof. Due to the refinement relation between $\mathcal{C}_0$ and $\mathcal{C}_1$ for every $(\tau, \mathbb{I}) \in \mathcal{C}_1$ there is a $(\tau, \mathbb{J}) \in \mathcal{C}_0$ such that $\mathbb{I} \subseteq \mathbb{J}$. Thus, by applying Lemma 3.1 to the definition of the $\models_{(.)}$-relation we are done. □

The above proposition provides us conditions under which we can modify given time requirements or derive further time requirements so that they remain satisfied by the specification. Proposition 3.2 gives us a mechanism to perform such transformations and derivations on our low-level time requirements. A number of useful transformation and derivation steps are described in Sections 3.1.2 and 3.1.3.

We are now able to state formally, when a requirement/context pair refines another one respecting a given specification.

Corollary 3.4 (Refinement invariance re context composition)**.** *Given four requirements* $r_1, r_1', r_2, r_2' \subseteq \vec{C} \times \mathbb{T}$*, and four contexts,* $\mathcal{C}_1, \mathcal{C}_1', \mathcal{C}_2, \mathcal{C}_2' \subseteq \vec{C}^{\kappa} \times [\mathbb{T}_\infty]$*. Then, provided* $r_i' \rightsquigarrow r_i$ *and* $\mathcal{C}_i' \preceq \mathcal{C}_i$ *for all* $i \in \{1, 2\}$*, the following holds for any specification* $Spec \subseteq \vec{C}^{\kappa}$

$$((Spec \models_{\mathcal{C}_1} r_1') \wedge (Spec \models_{\mathcal{C}_2} r_2')) \Rightarrow Spec \models_{\mathcal{C}_1' \otimes \mathcal{C}_2'} r_1 \wedge r_2.$$

Proof. By Proposition 2.1 we obtain

$$((Spec \models_{\mathcal{C}_1} r_1') \wedge (Spec \models_{\mathcal{C}_2} r_2')) \Rightarrow Spec \models_{\mathcal{C}_1 \otimes \mathcal{C}_2} r_1' \wedge r_2'.$$

Because $(\mathcal{C}_1' \otimes \mathcal{C}_2') \preceq (\mathcal{C}_1 \otimes \mathcal{C}_2)$ and $(r_1' \wedge r_2') \rightsquigarrow (r_1 \wedge r_2)$ we can apply Proposition 3.3. □

***Remark* 3.1** (Refinement for Operational Semantics)**.** In Chapters 4 and 5 we are respectively dealing with operational design and implementation models. For the formalism applied there, the notion of refinement coincides with the *simulation relation* [Mil71] (cf. Definition A.6). The same also applies to the temporal logic – if a specification $Spec_A$, which simulates a specification $Spec_B$, satisfies a property, then so does $Spec_B$ (cf. the proof in [CGP99]). ∘

3.1.2 Consistency and Completeness

Consistency and completeness are primary quality attributes that specifications must exhibit. In our settings the consistency can be represented as the existence of at least one element from $\wp(\vec{C}^{\kappa})$, which satisfies all requirements in their respective contexts. Formally, a given set $R = \{(r_1, \mathcal{C}_1), (r_2, \mathcal{C}_2), \dots\}$ defined over the channels from C can be characterized by the set of fulfilling specifications:

$$Spec(R) \stackrel{\text{def}}{=} \{Spec \subseteq \vec{C}^{\kappa} \mid \forall (r, \mathcal{C}) \in R : Spec \models_{\mathcal{C}} r\}.$$

An obvious way to define R's consistency is to demand that $Spec(R) \neq \emptyset$. However, $Spec(R)$ can contain specifications which trivially satisfy some requirements by disjointness to their respective contexts; in other words, for any $(r_1, \mathcal{C}_1)$, $(r_2, \mathcal{C}_2)$ a specification $Spec \in Spec(R)$ could contain solely traces which are either in $\mathcal{C}_1$ or in $\mathcal{C}_2$ but not in both contexts. In order to rule out this case, the consistency definition can be strengthened to $ConsSpec(R) \neq \emptyset$, where

$$ConsSpec(R) \stackrel{\text{def}}{=} Spec(R) \cap \{Spec \subseteq \vec{C}^{\kappa} \mid \forall R' \subseteq R : \mathbb{O}_{(r,\mathcal{C})\in R'}\mathcal{C} \text{ is non-trivial for } Spec\}.$$

The completeness is not an obvious issue: It is primary defined from the customer's point of view, i.e., the specification is complete if it addresses all features desired by the customer [Poh07]. According to this definition the completeness can only be validated, not formally formulated or verified. Other reasonable interpretation of the term "completeness" is the absence of *implicit* requirements. A set of constraints can imply further constraints, which may be of importance for subsequent development steps, e.g., design and implementation. Of course, there may be infinitely many possible constraints and the question to which extent the requirements have to be made explicit is of methodological nature. An answer lies beyond the scope of the present work, but we will provide a mathematical framework for the formal derivation of some interesting types of constraints from existing specifications, with emphasis lying on time behavior.

In general, if we prove for a set of requirements R that it implies some further requirement r we can add $(r, \mathbb{O}_{(\bar{r},\mathcal{C})\in R}\mathcal{C})$ to the set R, without changing the corresponding set of satisfying specifications.

Sporadic Events

If no information about periodicity of an event is provided by the specification, the event can be seen as sporadic. The only information, needed for its formal specification, is the minimal separation between consecutive event instances. This separation must be greater than zero, since, otherwise, there is no possibility of distinguishing between the particular instances. Our assumption here is that the resolution chosen for the time scale is sufficient to order the instances of any event A totally in every observation. In fact for a dense-time domain, like non-negative reals, one should either exclude the possibility of multiple event observations, or explicitly define a new composite event, which captures a sequence of messages from A, in order to deal with arriving request vectors, which result in series of responses. Our semantics explicitly facilitates the latter solution, which we have already met in the formal model of the CPS study (cf. Section 2.4).

For a given specification $Spec \subseteq \vec{C}^{\kappa}$ we define the minimal inter-arrival time of some (sporadic) event A as the length of the shortest distance between A's occurrences. Formally,

$$d_A \stackrel{\text{def}}{=} \min\{t_2 - t_1 \mid t_1 < t_2 \wedge \tau \in Spec \wedge (\tau, t_1) \models A \wedge (\tau, t_2) \models A\}.$$

Then, it holds per construction for an arbitrary context $\mathcal{C}$

$$Spec \models_{\mathcal{C}} i_s(A, [d_A, \infty)).$$

The same procedure can be lifted for the sets of specifications by calculating d_A over *all* specifications in the set. Then, by introducing $(i_s(A,[d_A,\infty)),\mathcal{C})$ into the set of requirements does not change the set of specifications they describe. Moreover, provided $\mathcal{C}$ combined with the other contexts from requirements set R is non-trivial for all $Spec \in Spec(R)$, the consistency of the specification is preserved, i.e., $ConsSpec(R) = ConsSpec(R \cup \{(i_s(A,[d_A,\infty)),\mathcal{C})\})$.

Period Propagation

The periodic behavior of the system formulated as an i_p- or i_s-requirement on its outputs along with a response time requirement can be propagated backwards to the input behavior, which has to be guaranteed by the environment. This is also possible the other way around: for a given frequency of an input event and a response time, to derive the expected period of the output. However, in both cases an additional condition is needed, which states that the system cannot produce outputs without corresponding inputs or ignore some inputs. The one-to-one correspondence between inputs and outputs, as introduced in Section 2.3.1, is a feasible requirement for the most systems, which often stays implicit in the specifications. Thus, in the following we will work with r_u-constraints (cf. Section 2.3.2). The following proposition states the formal relationships between events, as discussed above.

Proposition 3.5 (Period propagation)**.** *Given events* $A_1, A_2 \in \wp(\vec{C} \times \mathbb{T})$ *and a pair of intervals* $\mathbb{I}_1, \mathbb{I}_2 \in [\mathbb{T}_\infty]$ *with* $\operatorname{glb}\mathbb{I}_1 \geq \#\mathbb{I}_2$*, the following two properties hold*

$$i_s(A_1,\mathbb{I}_1) \wedge r_u(A_1,A_2,\mathbb{I}_2) \Rightarrow i_s(A_2,\mathbb{I}_3), \tag{3.1a}$$

$$i_s(A_2,\mathbb{I}_1) \wedge r_u(A_1,A_2,\mathbb{I}_2) \Rightarrow i_s(A_1,\mathbb{I}_3), \tag{3.1b}$$

where $\mathbb{I}_3 = [\operatorname{glb}\mathbb{I}_1 - \#\mathbb{I}_2, \operatorname{lub}\mathbb{I}_1 + \#\mathbb{I}_2]$.

Proof. Here, we prove (3.1a); Property (3.1b) is symmetrical and its proof goes analogously. Given two (infinite) sequences $a_1, a_2, \ldots, b_1, b_2, \ldots \in \mathbb{T}$ such that for any i

- a_i/b_i is the point in time of the ith occurrence of A_1/A_2, respectively,
- according to the premise of the first implication, $a_{i+1} - a_i \in \mathbb{I}_1$ and $b_i - a_i \in \mathbb{I}_2$.

We show that the relation between any pair of successive occurrences of A_2 belongs to $\mathbb{I}_3$, i.e., $b_{i+1} - b_i \in \mathbb{I}_3$ for any i. Let $n_i = \operatorname{glb}\mathbb{I}_i$ and $x_i = \operatorname{lub}\mathbb{I}_i$ for all $i \in \{1,2,3\}$, then both points from the above result in the following inequality system

$$\begin{array}{ll}
 & \left\{\begin{array}{lcccl} n_1 & \leq & a_{i+1} - a_i & \leq & x_1 \\ n_2 & \leq & b_i - a_i & \leq & x_2 \\ n_2 & \leq & b_{i+1} - a_{i+1} & \leq & x_2 \end{array}\right. \\
\Rightarrow & \quad (* \text{ subtract the second line from the third } *) \\
 & \left\{\begin{array}{lcccl} n_1 & \leq & a_{i+1} - a_i & \leq & x_1 \\ n_2 - x_2 & \leq & b_{i+1} - a_{i+1} - (b_i - a_i) & \leq & x_2 - n_2 \end{array}\right. \\
\Rightarrow & \quad (* \text{ add the first line to the second } *) \\
 & n_2 - x_2 + n_1 \leq b_{i+1} - b_i \leq x_2 - n_2 + x_1 \\
\Leftrightarrow & \\
 & n_3 \leq b_{i+1} - b_i \leq x_3
\end{array}$$

□

We note that in the case of exact time constraints for response times, the input and output periods remain the same. Moreover, if the response time is bigger than the period, the order of responses can change, i.e., in this case the sequence $b_1, b_2, \ldots$ is not monotonically increasing. As shown by the next corollary, the same results can be obtained for periodic events.

Corollary 3.6 (Period propagation 2)**.** *Given events* $A_1, A_2 \in \wp(\vec{C} \times \mathbb{T})$ *and a pair of intervals* $\mathbb{I}_1, \mathbb{I}_2 \in [\mathbb{T}_\infty]$ *with* $\mathrm{glb}\,\mathbb{I}_1 \geq \#\mathbb{I}_2$*, the following two properties hold*

$$i_p(A_1, \mathbb{I}_1) \wedge r_u(A_1, A_2, \mathbb{I}_2) \Rightarrow i_p(A_2, \mathbb{I}_3), \tag{3.2a}$$

$$i_p(A_2, \mathbb{I}_1) \wedge r_u(A_1, A_2, \mathbb{I}_2) \Rightarrow i_p(A_1, \mathbb{I}_3), \tag{3.2b}$$

where $\mathbb{I}_3 = [\mathrm{glb}\,\mathbb{I}_1 - \#\mathbb{I}_2, \mathrm{lub}\,\mathbb{I}_1 + \#\mathbb{I}_2]$.

Proof. We show the first property, the proof for the second one goes analogously. According to definitions of i_s and i_p it holds

$$i_p(A_1, \mathbb{I}_1) \equiv i_s(A_1, \mathbb{I}_1) \wedge br(A_1, A_1, \mathbb{I}_1) \qquad \text{and} \qquad i_p(A_2, \mathbb{I}_3) \equiv i_s(A_2, \mathbb{I}_3) \wedge br(A_2, A_2, \mathbb{I}_3).$$

Moreover, since $\mathbb{I}_1 \subseteq \mathbb{I}_3$ we can apply Lemma 3.2:

$$br(A_1, A_1, \mathbb{I}_1) \Rightarrow br(A_1, A_1, \mathbb{I}_3).$$

Then, we prove Property (3.2a) using Proposition 3.5:

$$\begin{aligned}
& i_p(A_1, \mathbb{I}_1) \wedge r_u(A_1, A_2, \mathbb{I}_2) \Rightarrow i_p(A_2, \mathbb{I}_3) \\
\Leftarrow \quad & (* \text{ definitions of } i_s,\ i_p, \text{ predicate calculus } *) \\
& (i_s(A_1, \mathbb{I}_1) \wedge r_u(A_1, A_2, \mathbb{I}_2) \Rightarrow i_s(A_2, \mathbb{I}_3)) \wedge (br(A_1, A_1, \mathbb{I}_1) \Rightarrow br(A_1, A_1, \mathbb{I}_3)) \\
\Leftrightarrow \quad & (* \text{ Lemma 3.2, Proposition 3.5 } *) \\
& \text{true}
\end{aligned}$$

□

A direct consequence of the both propositions from above is that the type of the period constraint (periodic/sporadic) is preserved by the forward and backward propagation of input/output-event periods by response-time requirements.

CPS Case Study (*con'ed*). In the previous chapter, in Section 2.4, we have met period and response time requirements put on a sensor of the CPS system in the DMode:

(R3) The response time on the DScan-command must lie within $[3, 4.5]$ ms, and

(R4) the DScan-command is sent out every 15 ms.

Using the propagation proposition introduced above, we can deduce the period of the sensor responses to the ECU, and add the following requirement to the CPS specification:

$$r_{\mathrm{R7}} \stackrel{\mathrm{def}}{\equiv} i_p(A_{obj}, [13.5, 16.5]).$$

r_{R7} is embedded into the context $\mathcal{C}_{\mathsf{DScan}}$.

3.1.3 Decomposition

Time requirements are formulated for the visible behavior of the whole system. The user is interested, for example, that a request to the system and its reaction to this request lie within a certain time span. However, often, the answer on a request is the result of an interplay of several (possibly distributed) components, the system is built of. Thus, the time constraint has somehow to be broken down and distributed between the constituents of the system. Thereby, the original constraint should not be violated.

Another topic of this section is decomposition of specifications in change-insensitive parts. This type of decomposition is useful for bordering of verification and validation effort in a case of a change in requirements.

Partitioning Response Time

Given a response requirement $r(A_1, A_2, \mathbb{I})$, a system architecture, which can be seen as a directed graph with components as nodes and channels as directed edges, we want to answer the question, what is a valid partition of r over the component network.

For a response time requirement $r(A_s, A_e, \mathbb{I})$, a compatible *response chain* of length n $(1 < n)$ is a conjunction of response requirements $rc = \bigwedge_{i\in[1,n]} r_i$ such that for any pair r_i and r_{i+1} $(i < n)$ holds:

$$r_i \equiv r(A_{i-1}, A_i, \mathbb{I}_i) \qquad \text{and} \qquad r_{i+1} \equiv r(A_i, A_{i+1}, \mathbb{I}_{i+1}),$$

with

$$A_0 = A_s, \qquad A_n = A_e, \qquad \text{and} \qquad \mathbb{I} = \sum_{i\in[1,n]} \mathbb{I}_i.$$

By $RC_n(A_s, A_e, \mathbb{I})$ we denote the set of all possible chains of length n for $r(A_s, A_e, \mathbb{I})$. For $n = 1$ we define $RC_1(A_s, A_e, \mathbb{I}) \stackrel{\text{def}}{=} \{r(A_s, A_e, \mathbb{I})\}$.

Proposition 3.7 (Correctness of response time partition)**.** *For a response requirement $r(A_s, A_e, \mathbb{I})$, a context $\mathcal{C}$, $n \in \mathbb{N}_+$, and any response chain $rc \in RC_n(A_s, A_e, \mathbb{I})$, holds for any $(\tau, t) \in \vec{C} \times \mathbb{T}$*

$$(\tau, t) \models_{\mathcal{C}} \bigwedge_{r\in rc} r \Rightarrow (\tau, t) \models_{\mathcal{C}} r(A_s, A_e, \mathbb{I}).$$

Proof. The case $n = 1$ is trivial. For $1 < n$ holds:

$$(\tau, t) \models_{\mathcal{C}} \bigwedge_{r\in rc} r \Leftrightarrow (\tau, t) \models_{\mathcal{C}} r(A_s, A_2, \mathbb{I}_1) \wedge \left(\bigwedge_{1<i<n} r(A_i, A_{i+1}, \mathbb{I}_i) \right) \wedge r(A_n, A_e, \mathbb{I}_n).$$

Then, the statement of the proposition follows immediately from the transitivity of response time requirements. □

***Remark* 3.2** (Combining refinement and decomposition)**.** Combined with the notion of refinement from Section 3.1.1 we can weaken the definition of response time chains and allow $\sum_{i\in[1,n]} \mathbb{I}_i$ to be a subset of $\mathbb{I}$. Then, every response time chain would refine the original response time requirement. ◦

For contexts of a response time chain of length n must hold $\mathbin{⍟}_{i\in[1,n]} \mathcal{C}_i = \mathcal{C}$. This can be achieved by defining for every $r_k = r(A_{k-1}, A_k, \mathbb{I}_k)$ $(k \in [1,n])$

$$\mathcal{C}_k \stackrel{\text{def}}{=} \{(\tau, \mathbb{I}') \mid (\tau, \mathbb{I}) \in \mathcal{C} \wedge \mathbb{I}' = \mathbb{I} + \sum_{i\in[1,k)} \mathbb{I}_i\}.$$

Then, the sets of satisfying specifications which contains $(r(A_1, A_2, \mathbb{I}), \mathcal{C})$ does not change if we add the requirements from the response time chain to it.

Independent Requirements

Changes in requirement specification at different stages of development are a common challenge in today's practice of system development. The reasons for changes are manifold [Poh07]: First of all, the stakeholders can change their mind concerning the problem that the system has to solve. Second, the detected inconsistencies in the specification, either as the result of its analysis or during subsequent development phases. A common source of the latter inconsistencies are implicit technical constraints, which we have already discussed in Section 2.3.2 in the paragraphs about BCET/WCET and latency on page 41, – in general, they cannot be fully considered during the analysis phase.

While changing a particular requirement (or adding a new one), in order to omit an inconsistent specification its consistency with the rest of requirements must be checked (again). In order to bound the effort in this case, we introduce the notion of *independent* time constraints, whose satisfaction by specifications is independent from each other.

We can identify two reasons of independence: the requirements must hold in disjoint contexts and/or they constrain disjoint sets of channels. In order to identify which parts of a specification are constrained by some time requirement we define the *constraint scope* $scop(r, \mathcal{C}) \subseteq C$ of a time requirement r. Informally, a channel belongs to a scope of r if and only if r constraints the set of possible valuations on this channel. Formally,

$$\begin{aligned} c \in scop(r, \mathcal{C}) \stackrel{\text{def}}{\Leftrightarrow} \quad & \{\tau \mid \tau \models_{\mathcal{C}} r\} = \emptyset \\ & \vee \exists \tau : \{\tau'\lceil_{\{c\}} \mid (\tau'\lceil_{C\setminus\{c\}} = \tau\lceil_{C\setminus\{c\}}) \wedge (\tau' \models_{\mathcal{C}} r)\} \subsetneq \overrightarrow{\{c\}}. \end{aligned}$$

Please note that by this definition, as expected, $scop(\mathrm{tt}, \mathcal{C}) = \emptyset$ and $scop(\mathrm{ff}, \mathcal{C}) = C$. It is important to note, what kind of channels are declared out of scope by the *scop*-predicate: Compared with syntactic filtering, tautologies, like $P(\tau.c) \vee \neg P(\tau.c)$, do not come into $scop(r, \mathcal{C})$. In the case of implications, e.g., $P(\tau.c_1) \rightarrow Q(\tau.c_2)$, both c_1 and c_2 are considered to be constrained by r. Now we can define the independence formally.

Definition 3.2 (Independence of time requirements)**.** *Given two requirement/context pairs* $(r_1, \mathcal{C}_1), (r_2, \mathcal{C}_2) \subseteq (\vec{C} \times \mathbb{T}) \times (\vec{C}^\kappa \times [\mathbb{T}_\infty])$*. Then, we say that* $(r_1, \mathcal{C}_1)$ *is* independent (satisfiable) *from* $(r_2, \mathcal{C}_2)$ *if and only if*

$$scop(r_1, \mathcal{C}_1) \cap scop(r_2, \mathcal{C}_2) = \emptyset \qquad or \qquad \mathcal{C}_1 \oslash \mathcal{C}_2 = \emptyset. \qquad \diamond$$

The notion of independence is symmetric. Using it we can ensure that, e.g., the procedure of introducing a new constraint/context pair $(r, \mathcal{C})$ into a set of specifications described by constraints $R = \{(r_1, \mathcal{C}_1), (r_2, \mathcal{C}_2), \dots\}$ must check the consistency between $(r, \mathcal{C})$ and a subset of R, which contains dependent constraints only. Then, provided both $(r, \mathcal{C})$ and R are satisfiable, the set $R' = R \cup \{(r, \mathcal{C})\}$ is satisfiable too. In other words, R' is consistent and thus, there exists a system, which fulfills all constraints from R'. Such procedure can surely bound the overall effort. The above considerations are formally underpinned by the following proposition.

Proposition 3.8 (Independence of satisfiability)**.** *Given two independent satisfiable requirement/context pairs* $(r_1, \mathcal{C}_1), (r_2, \mathcal{C}_2) \subseteq (\vec{C} \times \mathbb{T}) \times (\vec{C}^\kappa \times [\mathbb{T}_\infty])$*, then*

$$\exists Spec \subseteq \vec{C}^\kappa : (Spec \neq \emptyset) \wedge (Spec \models_{\mathcal{C}_1} r_1) \wedge (Spec \models_{\mathcal{C}_2} r_2).$$

Proof. We construct *Spec* as the intersection of the sets of traces satisfying r_1/r_2:

$$Spec \stackrel{\text{def}}{=} \bigcap_{i \in \{1,2\}} \{\tau \in \vec{C}^\kappa \mid \tau \models_{\mathcal{C}_i} r_i\}.$$

In order to show that *Spec* is not empty, we consider both conditions from definition of independence:

- If $scop(r_1, \mathcal{C}_1) \cap scop(r_2, \mathcal{C}_2) = \emptyset$ then r_1 and r_2 constrain different channels. Further on, it holds for all $\tau, \tau' \in \vec{C}$ and $i \in \{1, 2\}$

  $$\tau' \lceil_{scop(r_i, \mathcal{C}_i)} = \tau \lceil_{scop(r_i, \mathcal{C}_i)} \Rightarrow (\tau \models_{\mathcal{C}_i} r_i \Leftrightarrow \tau' \models_{\mathcal{C}_i} r_i).$$

  Thus, there exists a trace τ such that

  $$\exists \tau' : \tau \lceil_{scop(r_1, \mathcal{C}_1)} = \tau' \lceil_{scop(r_1, \mathcal{C}_1)} \wedge \tau' \models_{\mathcal{C}_1} r_1,$$

  and

  $$\exists \tau'' : \tau \lceil_{scop(r_2, \mathcal{C}_2)} = \tau'' \lceil_{scop(r_2, \mathcal{C}_2)} \wedge \tau'' \models_{\mathcal{C}_2} r_2,$$

  and τ is present in both intersected sets.

- If the contexts are disjoint, i.e., $\mathcal{C}_1 \oslash \mathcal{C}_2 = \emptyset$, there exists a trace τ, which satisfies r_1 at $\cup_{(\tau, \mathbb{I}) \in \mathcal{C}_1} \mathbb{I}$ and r_2 at $\cup_{(\tau, \mathbb{I}) \in \mathcal{C}_2} \mathbb{I}$, since these sets are also disjoint. □

3.2 A Formal Model for System Specification

In the previous sections we were working directly on the semantic domain of reactive systems. This view on the system as an unstructured set of predicates is surely justified at the early stages, like analysis and documentation phases of requirements engineering. At that time only loose collection of requirements is available. However, later on in the development process, as more elaborate models of the system come into being, which structure and group the system functionality by different artifacts, like components, features, or packages, it makes sense to assign the time behavior to particular functional artifacts of these models [BJ05, Dam05]. This way it becomes possible to distribute the time constraints as certain obligations on the time behavior of particular artifacts, which have to be fulfilled by their respective implementations and, composed together, are conform to the original overall time requirements.

In the context of the vision sketched above, here, we provide a mechanism which allows to relate a time requirement and a component specification formally. In other words, we answer the question, when a component satisfies a time requirement.

Section 3.2.1 introduces the system model of structured component specifications. It resembles the idea of the FOCUS framework [Bro97, BS01a] in the settings of the domain of timed traces. Based on this system model we introduce in Section 3.2.2 composition and refinement of components. Finally, Section 3.2.3 shows the transition from individual time requirements to structured component specifications and relates the both notions of refinement: refinement of time behavior specifications and components refinement.

3.2.1 System Model

In Section 2.2.3 we have defined a timed system, or timed component, which communicates through directed typed channels from the set C sending and receiving messages from the set M^* with time stamps from the time domain $\mathbb{T}$ as a member of $\wp(\vec{C}^\kappa)$. We now provide mathematical means to describe the syntactic and semantic views on the systems or components within this system model. The essential novelty is the distinction and different treatment of input and output channel types.

The *black-box specification* of a component p with channels C over message set M and time domain $\mathbb{T}$ is a tuple $p = (I, O, F)$ with $C \stackrel{\text{def}}{=} I \cup O$ and $I \cap O = \emptyset$. Here, I, O determine the *syntactic interface* of p and the total relation F defines the semantics of p. The syntactic interface consists of a set of input channels I and a set of output channels O. Each channel $c \in C$ has an assigned type $ty(c) \in \wp(M^*)$. ty is a partition of M^* into different message types. Thus, a type is described by a set of message sequences.

The *black-box behavior* of a component is described by the relation

$$F : \vec{I}^\kappa \mapsto (\vec{O}^\kappa \mapsto \mathbb{B}),$$

which maps complete input histories to the sets of complete output histories. Thus, we also call F *relational component specification* or *I/O behavior*. When two histories $\iota \in \vec{I}$ and $o \in \vec{O}$

satisfy F, we write $F(\iota, o)$, or $F.\iota.o$ for short. $F(\iota)$ or $F.\iota$ stands for (a set of) all satisfying output histories of ι, in other words, we abbreviate $\{o \mid F.\iota.o\}$ by $F.\iota$. Several output behaviors from $\vec{O}^\kappa$ that satisfy $F.\iota$ for some $\iota \in \vec{I}^\kappa$ model non-determinism.

3.2.2 Causality, Composition, and Refinement

Here, we present the notions of causality, composition, and refinement for the relational component specifications from the previous section. The causality or time guardedness in the settings of FOCUS refers in the first place to the relation between inputs and outputs. At every point in time, it allows outputs to be dependent from previous inputs only. This asymmetry between inputs and outputs is captured by the following definition.

Definition 3.3 (Time guardedness, strong causality)**.** *The relational component specification* $F : \vec{I}^\kappa \mapsto (\vec{O}^\kappa \mapsto \mathbb{B})$ *is called* time guarded *or* strong causal *if*

$$\exists \delta \in \mathbb{T}_+ : \forall t \in \mathbb{T}, \iota_1, \iota_2 \in \vec{I}^\kappa : \iota_1{\downarrow_t} = \iota_2{\downarrow_t} \Rightarrow (F.\iota_1){\downarrow_{t+\delta}} = (F.\iota_2){\downarrow_{t+\delta}}. \qquad \diamond$$

The consequence of time guardedness is that the relation F is either always false or total, i.e., for any $\iota \in \vec{I}^\kappa$, there exists at least one $o \in \vec{O}^\kappa$ such that $F.\iota.o$. In the following we deal with the latter case only.

The composition is defined for *compatible* relational component specifications only. It is well-defined in the sense that it yields a relational component specification again [BS01a]. The notions of compatibility and composition are introduced next.

Definition 3.4 (Compatibility and composition of timed components)**.** *Given two subsystems* $p_1 = (I_1, O_1, F_1)$ *and* $p_2 = (I_2, O_2, F_2)$ *with the corresponding channel type mappings* $ty_i : (I_i \cup O_i) \mapsto \wp(M^*)$ *for* $i \in \{1, 2\}$*. Then, we call* p_1 *and* p_2 compatible *if and only if*

- *they do not share any input and output channels:* $I_1 \cap I_2 = \emptyset$ *and* $O_1 \cap O_2 = \emptyset$*,*
- *the homonymous channels have the same type:* $\forall c \in C : ty_1(c) = ty_2(c)$*,*

where $C = (I_1 \cup I_2) \cap (O_1 \cup O_2)$*. The composition of two compatible systems is denoted by*

$$p \stackrel{\text{def}}{=} p_1 \otimes p_2 = (I, O, F_1 \otimes F_2).$$

The syntactical interface of p *is*

$$I \stackrel{\text{def}}{=} (I_1 \cup I_2) \setminus (O_1 \cup O_2), \qquad O \stackrel{\text{def}}{=} (O_1 \cup O_2) \setminus (I_1 \cup I_2).$$

The semantics of p *is described by* $F_1 \otimes F_2 : I \mapsto (O \mapsto \mathbb{B})$ *as follows*

$$(F_1 \otimes F_2).(\tau\lceil_I).(\tau\lceil_O) \stackrel{\text{def}}{\Leftrightarrow} F_1.(\tau\lceil_{I_1}).(\tau\lceil_{O_1}) \wedge F_2.(\tau\lceil_{I_2}).(\tau\lceil_{O_2}). \qquad \diamond$$

The homonymous input/output channels disappear from the syntactical interface of the composed system. In the description of the system I/O behavior $F_1 \otimes F_2$ the affected channels are existentially quantified.

Finally, we define the refinement relation between relational component specifications as the set inclusion between outputs for every input.

Definition 3.5 (Property refinement)**.** *Given a pair of relational specifications* $F_0, F_1 : \vec{I}^\kappa \mapsto (\vec{O}^\kappa \mapsto \mathbb{B})$*. We say that* F_1 *is a* property refinement *of* F_0*, denoted by* $F_1 \rightsquigarrow F_0$ *if and only if*

$$\forall \iota \in \vec{I}^\kappa, o \in \vec{O}^\kappa : F_1.\iota.o \Rightarrow F_0.\iota.o$$

holds. ◇

We also write $p_1 \rightsquigarrow p_0$ if $p_0 = (I, O, F_0)$, $p_1 = (I, O, F_1)$, and $F_1 \rightsquigarrow F_0$. It is easy to see that the refinement relation is transitive and reflexive. Please also note that, in general, refinement as introduced above does not guarantee the preservation of time guardedness. However, in the rest of this thesis we will be dealing with time guarded specifications only. In particular, we expect the input totality of F_1, i.e., that for all $\iota \in \vec{I}^\kappa$, $F_1.\iota \neq \emptyset$. If F_1 is input total, then so is F_0. Finally, the composition operator $\otimes$ is monotonic with respect to the property refinement relation [BS01a], i.e.,

$$((F_1^c \rightsquigarrow F_1^a) \wedge (F_2^c \rightsquigarrow F_2^a)) \Rightarrow ((F_1^c \otimes F_2^c) \rightsquigarrow (F_1^a \otimes F_2^a)).$$

3.2.3 From Time Constraints to Component Specification

In the current section we formally relate a component as defined above and a time requirement, i.e., we define when the component satisfies the requirement. Moreover, we define when a change in time requirements produces a valid refinement step on the level of system components. In Section 3.1 we have met a number of transformation and derivation rules for timed behavior specifications. Given a time requirement/context pair $(r, \mathcal{C})$ and a component $p = (I, O, F)$ which realizes $(r, \mathcal{C})$ and provided r constraints channels from I only ($scop(r, \mathcal{C}) \subseteq I$), we answer the question, when a change of $(r, \mathcal{C})$ is a valid refinement step from p's perspective. In particular, we formally show that the refinement of r, $r_1 \rightsquigarrow r$, is *not* a valid refinement step for p, i.e., it cannot be guaranteed that p satisfies $(r_1, \mathcal{C})$. In contrast, the abstraction of r can be satisfied by a component p_1 such that $p_1 \rightsquigarrow p$. Such results allow us to work directly with time constraints, e.g., by applying methods from Section 3.1, and still to have the ability to evaluate the impact on the overall component-based specification, i.e., to produce valid refinements of the whole system model. This constitutes the time behavior as an independent but still integrated view on the system.

The basic scheme for deriving a relational, time-guarded component specification out of an untimed stream-based specification is described in [BK98]. It can be easily carried over to the timed domain. The resulting component is described in the A/G style (assumption/guarantee, cf. [Bro94, Pan90, MC81]), so that the liveness and safety specification parts of both the guarantee of the component and assumptions about the environment appear as separate predicates. Along the lines of [BK98], for a given timed property $r \subseteq \vec{C} \times \mathbb{T}$ and a context $\mathcal{C} \subseteq \vec{C}^\kappa \times [\mathbb{T}_\infty]$ we construct

the corresponding component $p(r,\mathcal{C}) = (I, O, F(r,\mathcal{C}))$ with disjoint $I, O \subseteq C$, as follows: For all $\iota \in \vec{I}^\kappa, o \in \vec{O}^\kappa$ holds

$$\begin{aligned} F(r,\mathcal{C}).\iota.o \overset{\text{def}}{\Leftrightarrow} \quad & ((\exists \tau \in \vec{C} : \tau\lceil_I = \iota \wedge \tau \models_{\mathcal{C}} r) \Rightarrow (\exists \tau \in \vec{C} : \tau\lceil_I = \iota \wedge \tau\lceil_O = o \wedge \tau \models_{\mathcal{C}} r)) \\ & \wedge (\forall t \in \mathbb{T} : \exists \delta \in \mathbb{T} : \\ & \qquad (\exists \tau \in \vec{C} : \iota\downarrow_t = (\tau\lceil_I)\downarrow_t \wedge \tau \models_{\mathcal{C}} r) \\ & \qquad \Rightarrow (\exists \tau \in \vec{C} : \iota\downarrow_t = (\tau\lceil_I)\downarrow_t \wedge o\downarrow_{t+\delta} = (\tau\lceil_O)\downarrow_{t+\delta} \wedge \tau \models_{\mathcal{C}} r)). \end{aligned}$$

The interpretation of the first conjunct is that a component has to fulfill a time requirement only for those traces, in which the valuation of input channels of $F(r,\mathcal{C})$ can be satisfied by $(r,\mathcal{C})$. Since time guarded components do not control their inputs[1], $F(r,\mathcal{C})$ is not responsible for unsatisfiable input valuations. The second conjunct says that every prefix, which has a continuation satisfying $(r,\mathcal{C})$, should also have a continuation, which satisfies $F(r,\mathcal{C})$. By this, the second conjunct treats also behaviors, not captured by the first one, namely traces, which satisfy the requirement only up to a certain finite point in time. On its part the second conjunct is too weak for full (i.e., infinite) traces satisfying $(r,\mathcal{C})$, in these cases the first conjunct applies.

Then, a relational component specification F fulfills a time requirement r in context $\mathcal{C}$, denoted by $F \models_{\mathcal{C}} r$ if and only if

$$F \models_{\mathcal{C}} r \overset{\text{def}}{\Leftrightarrow} F \rightsquigarrow F(r,\mathcal{C}). \tag{3.3}$$

By this, definition we immediately obtain property preservation during the property refinement. This fact is captured by the following corollary.

Corollary 3.9 ($\models_{(.)}$-relation monotony re property refinement)**.** *For any $F_1, F_0 : \vec{I}^\kappa \mapsto (\vec{O}^\kappa \mapsto \mathbb{B})$, $r \subseteq \vec{C} \times \mathbb{T}$ and $\mathcal{C} \subseteq \vec{C}^\kappa \times [\mathbb{T}_\infty]$. Provided $I, O \subseteq C$ and $I \cap O = \emptyset$, then*

$$F_0 \models_{\mathcal{C}} r \wedge F_1 \rightsquigarrow F_0 \Rightarrow F_1 \models_{\mathcal{C}} r.$$

Proof. Follows directly from the transitivity of refinement relation. □

There exist two ways to refine a specification written in A/G-style: The assumption can be relaxed, and/or the guarantee can be strengthened. We express this intuition by defining the following relation between requirement/context-pairs. It is defined relative to the directions of the channels in I and O: The weakening of the component's assumption corresponds to the increase of the number of different satisfying valuations from $\vec{I}^\kappa$. The strengthening of guarantee, corresponds to the reduction of satisfying valuations from $\vec{O}^\kappa$, which were satisfiable in the original requirement.

$$\begin{aligned} (r_1,\mathcal{C}_1) \overset{(I,O)}{\rightsquigarrow} (r_0,\mathcal{C}_0) \overset{\text{def}}{\Leftrightarrow} \quad & \{\tau\lceil_I \mid \tau \models_{\mathcal{C}_1} r_1\} \supseteq \{\tau\lceil_I \mid \tau \models_{\mathcal{C}_0} r_0\} \\ & \wedge \forall \iota \in \vec{I}^\kappa : \iota \in \{\tau\lceil_I \mid \tau \models_{\mathcal{C}_1} r_0\} \\ & \qquad \Rightarrow \{\tau\lceil_O \mid \tau\lceil_I = \iota \wedge \tau \models_{\mathcal{C}_1} r_1\} \subseteq \{\tau\lceil_O \mid \tau\lceil_I = \iota \wedge \tau \models_{\mathcal{C}_0} r_0\}. \end{aligned}$$

[1] We do not allow self-loops, thus a component cannot guarantee anything about its own future inputs.

Please note that the $\overset{((.),(.))}{\rightsquigarrow}$-relation can be *both* a refinement and an abstraction relation between a pair of timed properties. It is easy to prove that $\overset{((.),(.))}{\rightsquigarrow}$ is reflexive and transitive. In the remainder of this section, we study when the relation holds for our low-level requirements. The following proposition shows the equivalence between the $\overset{((.),(.))}{\rightsquigarrow}$-relation and the property refinement.

Proposition 3.10 (Relation between $\overset{((.),(.))}{\rightsquigarrow}$ and $\rightsquigarrow$)**.** *For any* $r_1, r_0 \subseteq \vec{C} \times \mathbb{T}$, $\mathcal{C}_1, \mathcal{C}_0 \subseteq \vec{C}^\kappa \times [\mathbb{T}_\infty]$, *and* $I, O \subseteq C$ $(I \cap O = \emptyset)$*:*

$$(r_1, \mathcal{C}_1) \overset{(I,O)}{\rightsquigarrow} (r_0, \mathcal{C}_0) \Leftrightarrow F(r_1, \mathcal{C}_1) \rightsquigarrow F(r_0, \mathcal{C}_0).$$

Proof. We note that for any r the definition of $F(r, \mathcal{C})$ can be rewritten using the set comprehension as follows

$$\begin{aligned} F(r,\mathcal{C}).\iota.o \Leftrightarrow \quad & (\iota \in \{\tau\lceil_I \mid \tau \models r\}) \Rightarrow (o \in \{\tau\lceil_O \mid \tau\lceil_I = \iota \wedge \tau \models r\}) \\ \wedge\ & \forall t \in \mathbb{T} : \exists \delta \in \mathbb{T} : (\iota|_t \in \{(\tau\lceil_I)|_t \mid \tau \models r\}) \\ & \Rightarrow (o|_{t+\delta} \in \{(\tau\lceil_O)|_{t+\delta} \mid (\tau\lceil_I)|_t = \iota|_t \wedge \tau \models r\}) \end{aligned}$$

for all $\iota \in \vec{I}^\kappa, o \in \vec{O}^\kappa$. By this, $F(r_1, \mathcal{C}_1) \rightsquigarrow F(r_0, \mathcal{C}_0)$ if and only if the set in the premise of $F(r_1, \mathcal{C}_1)$ is bigger, or the set in the conclusion of $F(r_1, \mathcal{C}_1)$ is smaller for valid input valuations then the respective sets in the definition of $F(r_0, \mathcal{C}_0)$ (or both). By this, we immediately obtain implication in both directions. □

We are now able to formulate the following result: If a component realizes some timed property then it realizes all timed properties, which stand in a $\overset{((.),(.))}{\rightsquigarrow}$-relation to it.

Corollary 3.11 (Satisfiability preservation re $\overset{((.),(.))}{\rightsquigarrow}$-relation)**.** *For any* $F : \vec{I}^\kappa \mapsto (\vec{O}^\kappa \mapsto \mathbb{B})$, $r_1, r_0 \subseteq \vec{C} \times \mathbb{T}$, $\mathcal{C}_1, \mathcal{C}_0 \subseteq \vec{C}^\kappa \times [\mathbb{T}_\infty]$, *and* $I, O \subseteq C$ $(I \cap O = \emptyset)$*:*

$$F \models_{\mathcal{C}_1} r_1 \wedge (r_1, \mathcal{C}_1) \overset{(I,O)}{\rightsquigarrow} (r_0, \mathcal{C}_0) \Rightarrow F \models_{\mathcal{C}_0} r_0.$$

Proof. Follows from Proposition 3.10 and the transitivity of the refinement relation. □

If we compare the above result with the refinement relations between low-level time requirements listed in Table 3.2 on page 58, we see that, e.g., provided the relation $i_p(A_1, \mathbb{I}_1) \overset{(I,O)}{\rightsquigarrow} i_p(A_1, \mathbb{I}_2)$ between a pair of inter-arrival time requirements holds, if $\mathbb{I}_2 \subseteq \mathbb{I}_1$, then $i_p(A_1, \mathbb{I}_1) \rightsquigarrow i_p(A_1, \mathbb{I}_2)$ and $i_p(A_1, \mathbb{I}_1) \nleftarrow i_p(A_1, \mathbb{I}_2)$, otherwise. This corresponds to the intuition that a refinement of a component is only allowed to realize more relaxed time requirements on its inputs and more stringent ones on its outputs.

By Corollary 3.9 we know that the refinement of component specification preserves the satisfaction of time properties. Corollary 3.11 guarantees that a change of time properties which is faithful to the $\overset{((.),(.))}{\rightsquigarrow}$-relation, preserves the satisfaction by component specification. In other words, the time and functional aspects of the system behavior can evolve independently, provided certain verifiable conditions are not violated.

3.3 Related Work

Below we compare our contributions with related approaches concerning the two main topics of the present chapter.

Analysis of Real-Time Requirements

To our knowledge there is not much work on the systematized and formally-founded defect detection for the real-time behavior on the level of requirements.

As a classical precursor to the formal analysis methods from Section 3.1, in [JLHM91] a set of criteria is defined to help find errors in software requirements specifications. Based on these criteria, analysis procedures for a state-machine-based modeling language are defined. Using this language analytical models of real-time process-control software requirements can be described formally. The criteria and methods from [JLHM91] are rather specific to the domain of process control and to the applied automata-based formalism.

The same critique also applies to the consistency checks supported by the SCR* tool set (cf. [HBGL95] as well as Section 2.5 for the description of SCR*). Moreover, these checks are able to demonstrate that a SCR specification is well-formed, but cannot be used to derive further requirements.

In the formal-methods community various verification techniques for real-time behavior, formalized using quantitative temporal logic, are proposed. They reach from model-checking algorithms to deductive systems (cf. [BMN00] for an overview). Deductive systems offer a set of axioms and deduction rules. Given a set of propositions, it is possible to check by the use of deduction rules whether other propositions are a logical consequence of the former set. These systems are usually purely syntax-deductive. On the one hand, this fact easies their computer-supported application; on the other hand, they operate with syntactical constructs of the logic and thus are too low-level compared with the derivation rules presented here.

From Unstructured to Structured Specifications

The transition from unstructured to structured specifications within the (untimed) FOCUS formalism is treated in [BK98]. There, unstructured specifications are represented by *interaction interfaces*. Similarly to the notion of specification from Section 2.3.4, an interaction interface describes the mutual interaction between two or more components by a number of allowed valuations of a channel set by infinite message streams (cf. Definitions in Section 2.1). The authors provide a transition procedure from interaction interfaces to untimed FOCUS components. This procedure was adapted for our trace-based semantics in Section 3.2.3. However, the approach of [BK98] pursues a pure top-down view on the system development and thus, does not consider the evolution of the unstructured specification in parallel with structured ones but rather treats it as an intermediate disposable product on the way to the component-based (structured) specification. By contrast, we believe that the unstructured specification and especially the unstructured

timed behavior specification is an independent artifact of the development and has to be aligned to and compared with different (structured) solutions created during the successive development steps. Thus, we relate refinement operators of unstructured and structured specifications at the end of Section 3.2.3.

3.4 Summary

In the current chapter, we have enriched the specification framework from Chapter 2 by formally founded analysis and transformation methods with the goal to involve the models created using our denotational formalism into quality- and conformance-assurance activities carried out during development.

We see the application areas of the proposed framework in the analysis phases of requirements engineering, design, implementation, and deployment of reactive real-time systems. The development during the requirements engineering usually begins with a collection of requirements, the system should realize. These requirements, grouped by use cases, build the basis of an unstructured timed behavior specification. Modeled within our framework, this specification can be analyzed with respect to its consistency and completeness.

During the subsequent development steps different (mostly operational) component- or function-based models of the system under construction arise. The corresponding denotational analytical models can be checked for fulfillment of the previously developed unstructured or structured time specification. For this purpose in the next chapter, we will map an operational design formalism to the semantic domain of real-time systems from Section 2.2.3 (cf. Section 4.1.3 and also [BP99]). Then, the fulfillment can be shown by proving the refinement relation between design and specification models.

Moreover, we presented methods for the distribution of time requirements over particular components of design, or implementation models in a consistent manner. By this, more detailed structured timed behavior specifications arise. Thus, during the deployment, where the actual time behavior of the realized system can be estimated with a higher degree of confidence, the time behavior obligations of single components or functions can be discharged by their implementations. Provided this was accomplished successfully, the overall time behavior of the system is provably fulfilled by the overall implementation. Otherwise, these exist two options: either another partition of time requirements exists, which is fulfilled by the particular time behaviors, or the time specification is missed by the implementation. Both results can be produced using the presented framework, its decomposition theorems, and refinement relation.

4 From Application Logic to Task Model

In this chapter we provide an automata-based model, which allows us to describe systems in an operational style. The model is instantiated for the semantics of the CASE tool called AutoFocus and for its novel extension called AutoFocus Task Model (AFTM). AFTM is more appropriate for schedulability analysis as well as for subsequent deployment, which is the topic of Chapter 5. The relationship between AutoFocus and AFTM models regrading property preservation is also covered by the current chapter.

Contents

In the previous chapter we have given a formal characterization *when* certain manipulations on the model result in a valid refinement step. The topic of this chapter as well as of Chapter 5 is to present a constructive transformation approach, which provably preserves properties of the system. The approach includes transition from pure application logic to function clusters, which group the functions of the application logic and wrap them, so that their subsequent distribution over time and space (i.e., deployment) does not change the overall application behavior. We discuss the property preservation of the clustering process at the end of the current chapter and the property preservation during deployment – in Chapter 5.

Using our approach models of application logic of distributed reactive real-time systems can be created independently from their deployment platform. Afterwards, they can be clustered to tasks, distributed and deployed onto a network of computation nodes in a way which provably preserves the original behavior of the application logic on a platform-independent level. Thereby, our main design directive was to provide the maximum amount of flexibility to the system engineers using our approach. This includes the flexible mapping of application logic components to tasks, consideration of established OS-, middle-ware- and bus-protocol-standards as deployment platform, and the flexibility in the distribution of tasks over nodes and schedule slots. Technically this is realized by formulating a number of constraints which have to be satisfied by clustering and deployment solutions. If this is the case, the property preservation is guaranteed. These constraints can be checked automatically for any solution created by the system engineer. In the next paragraphs we discuss the relevance of our approach to the practice of embedded systems design.

The separation of application logic and its technical realization is a promising approach pursued by umpteen emerging approaches, like AUTOSAR[1] [HSF+04], and CASE-tools, like StateMate[2] [HLN+90] or Rose-RT[3]. To the benefits of such separation belongs the improved portability of the models. This is an important quality attribute for embedded software, since the heterogeneity in the domain of embedded hardware controllers is rather high and their innovation cycles are rather short. Another motivating factor is the great number of different communication protocols, like FlexRay [Fle04], CAN [ISO94], TTCan[4] [FMHH01], ByteFlight[5], TTP/C[6] [KG94], MOST[7], *etc.*, designed for the inter-node communication in the embedded systems domain. Thus, the independence respective changes in the technical infrastructure, facilitates the reuse of the application logic. Moreover, the abstraction from technical details of the system provides a more concise and clear view on the system logic, and partitions the state space of the system into logical and technical parts. This allows the application of compositional verification techniques and makes the formal verification of industrial-size systems by the modern state-of-the-art methods, like model checking, SAT-solving, or theorem proving more feasible. Finally, the correctness proofs for a certain platform can be reused for the overall correctness

[1] see http://www.autosar.de (20.06.2008)
[2] see http://modeling.telelogic.com/modeling/products/statemate (20.06.2008)
[3] see http://www-01.ibm.com/software/rational (20.06.2008)
[4] see http://www.ttcan.com (20.06.2008)
[5] see http://www.byteflight.com (20.06.2008)
[6] see http://www.vmars.tuwien.ac.at/projects/ttp/ttpc.html (20.06.2008)
[7] see http://www.mostnet.de (20.06.2008)

of several systems, whose application logic has to be deployed on this platform. The same also applies for a particular application logic with different platforms as deployment target.

The defiances of the separation approach constitute the low budgets for available computation resources, like RAM, PROM, or CPU frequency. This is dictated by the will to bring the costs per unit down, since in many domains the embedded systems are integrated into the mass-production goods. The consequence of this are the code-optimizations which target to optimize the application code for a specific platform. However, the increasing complexity and safety-criticality of embedded systems from the domains, like automotive or avionics, performed a change in the strategies of the enterprises in the last years [SZ06, Gri03, PBKS07]. The emerging standards for time-triggered OSes (OSEKtime [OSE01b]), time-triggered buses (FlexRay, TTP/C, TTCan) as well as for middle-ware abstraction layers (AUTOSAR [HSF+04], MetaH [Hon98]) aim at reducing the complexity, by making the time behavior of the systems more predictable and realizing the separation, which we described above. Inevitably this goes at the expense of performance, but is a strongly needed trade-off in the face of exponentially growing amount of software, e.g., in the automotive domain.

Contributions and Outline. In the present chapter, in Section 4.1, we provide the formal foundation of an automata-based framework, which is suitable to formalize execution and communication semantics of different CASE tool paradigms as well as of different technical platforms. Within this framework, a system is modeled as a set of communicating Mealy machines. The actual communication and computation semantics is incorporated into the composition operator which, together with constituting machines, determines the overall system behavior. We instantiate this framework for the semantics of a CASE tool called AutoFocus in Section 4.2 and for the semantics of AutoFocus Task Model (AFTM), a novel extension of AutoFocus tailored to facilitate deployment, in Section 4.3.

AutoFocus tool [HSE97] described in Section 4.2 is well suited for the component-based design of reactive systems. Its formalization serves as a starting point for the subsequent transformations towards a deployed system. As an intermediate step on this way we introduce AFTM in Section 4.3. It was realized on the basis of AutoFocus tool as described in [BGH+06]. AFTM models consist of tasks, which group and adapt the components of the application logics modeled in AutoFocus. AFTM was designed as an abstraction of application logic models, which is adequate for both subsequent development activities according to our reference process from Figure 1.1: schedulability analysis and deployment. As a rationale of this claim Section 4.3 contains the description and categorization of the main aspects and characteristics of scheduling approaches and programmer's models of deployment platforms.

Finally, Section 4.4 treats the property preservation during the transition from AutoFocus to AFTM. There, we provide a special synthesis algorithm, which allows flexible distribution of AutoFocus components over tasks: a task can cluster several components and one task step can correspond to a finite run of these components. The correctness of the synthesis algorithm is shown by proving a special simulation relation between AutoFocus components and AFTM tasks.

4.1 Operational Semantics for Reactive Systems

We give the logical characterization of the operational-style description of the system behavior using *state transition systems* (*STS*) adapted from [BP99] and [PP05]. After the definition of the semantic domain in Section 4.1.1, we introduce the STS formalism in Section 4.1.2, and map STSs to the denotational domain of real-time systems from Chapter 2 in Section 4.1.3. Afterwards, the parallel composition and refinement of STSs are defined and their properties are proved in 4.1.4 and 4.1.5, respectively. Finally, a variant of STS, which is more suitable to describe technical systems is introduced in 4.1.6.

4.1.1 Variable Valuations

An STS works on a set of *variables*, which is disjoint partitioned into *input*, *output* and *local* variable subsets. Each variable has a type and can be mapped to one value of this type. In every step an STS changes the values of its local and output variables according to the values of its input and local variables. The notion of a variable and its valuation is fixed by the following definition.

Definition 4.1 (Variables and their valuations)**.** *A* variable *v is a symbolic name from the* name-space universe VAR. VAR' *is the universe of* primed *variable names, whereas* VAR *assumed to contain unprimed names only, i.e.,* $\mathsf{VAR} \cap \mathsf{VAR}' = \emptyset$.

The type *of a variable is a subset of the set of message sequences M^*, so it is described by all permitted valuations of the variable (its* valuation domain*). The function $ty(v) \in \wp(M^*)$ maps every $v \in \mathsf{VAR} \cup \mathsf{VAR}'$ to its type.*

Given a variable set $V \subseteq \mathsf{VAR}$ the variable set valuation *$\alpha\colon V \mapsto M^*$maps every variable name from V to a value. The valuation α of the set V is called* type correct *if and only if*

$$\forall v \in V : \alpha.v \in ty(v).$$

The set of all type correct valuations of a variable set V is denoted by $\Lambda(V)$.

If $v \in \mathsf{VAR}$ is a variable, then priming *yields a new variable $v' \in \mathsf{VAR}'$. The same applies for sets of unprimed variables $V \subseteq \mathsf{VAR}$: the set $V' \subseteq \mathsf{VAR}'$ is built by adding a prime sign to every variable name in V: $V' \stackrel{\text{def}}{=} \{v' \mid v \in V\}$. For a valuation α over V, α' stands for a valuation of V' such that*

$$\forall v \in V : \alpha.v = \alpha'.v'.$$

Analogously, we define the un-priming *operation, which works in the opposite direction, i.e., maps its argument from VAR' to VAR.* ◇

Given two not necessarily disjoint variable sets $V, W \subseteq \mathsf{VAR} \cup \mathsf{VAR}'$, two type-correct valuations $\alpha \in \Lambda(V)$ and $\beta \in \Lambda(W)$, α and β agree on some common subset $U \subseteq V \cap W$, denoted by $\alpha \stackrel{U}{=} \beta$ if they map every variable from U to the same value:

$$\alpha \stackrel{U}{=} \beta \stackrel{\text{def}}{\Leftrightarrow} \forall v \in U : \alpha.v = \beta.v.$$

A natural extension compares sets of valuations: $\mathfrak{A} \overset{U}{=} \mathfrak{B}$ if and only if

$$\forall \alpha \in \mathfrak{A} : \exists \beta \in \mathfrak{B} : \alpha \overset{U}{=} \beta \qquad \text{and} \qquad \forall \beta \in \mathfrak{B} : \exists \alpha \in \mathfrak{A} : \alpha \overset{U}{=} \beta.$$

In the next section we will describe STSs by means of *assertions*. An assertion is a logical formula Ψ defined over a set of free variables $\text{free}.\Psi \subseteq \mathsf{VAR} \cup \mathsf{VAR}'$. A valuation α *satisfies* the assertion Ψ if Ψ evaluates to true when all its free variables $v \in \text{free}.\Psi$ are replaced by $\alpha.v$: We write $\alpha \vdash \Psi$. If an assertion Ψ contains both, primed and unprimed variables, we write $\alpha,\beta' \vdash \Psi$ if it is satisfied by a pair of valuations for unprimed/primed variables from α/β', respectively. Alternatively, we also write $\alpha \vdash \Psi$ for $\alpha \in \Lambda(V \cup V')$.

4.1.2 State Transition Systems

We can now proceed to the definition of state transition systems. An STS over the message set M is described by a 6-tuple $\mathfrak{S}(M) \overset{\text{def}}{=} (I, O, L, \mathfrak{I}, \delta, d^{\varepsilon})$. By $V \overset{\text{def}}{=} I \cup O \cup L$ we denote the set of variables ($V \subseteq \mathsf{VAR}$), which is built by mutual disjoint subsets of input (I), output (O) and local (L) variables. The type system of V is defined over (sequences of) M. In the following discussion we will omit the parameter M in the definitions of STSs if the message set is unimportant or clear from the context. A state of the system is a type-correct valuation of V, i.e., an element from $\Lambda(V)$. $\mathfrak{I}$ is an assertion characterizing the initial states of the system. δ is a set of transition assertions and d^{ε} is a special complementing transition which makes STSs *input-enabled*. The rest of this section gives a detailed description of the elements from $\mathfrak{S}$'s tuple and introduces auxiliary functions and predicates which will be used for the reasoning about STSs in the rest of this chapter as well as in Chapter 5.

$\mathfrak{I}$ must be satisfiable by at least one valuation of V and it is allowed to constrain output and local variables only, i.e., the set of possible initial inputs is not constrained by $\mathfrak{I}$. Formally,

$$\exists \alpha \in \Lambda(V) : \alpha \vdash \mathfrak{I} \qquad \text{and} \qquad \forall \alpha,\beta \in \Lambda(V) : \alpha \overset{L \cup O}{=} \beta \Rightarrow (\alpha \vdash \mathfrak{I} \Leftrightarrow \beta \vdash \mathfrak{I}). \tag{4.1}$$

Transitions of the system are contained in the finite set δ. Each transition is an assertion over a subset of $V \cup V'$, i.e., the free variables of any transition $d \in \delta$ must fulfill: $\text{free}.d \subseteq V \cup V'$. The unprimed variables describe the current system state, while the primed ones constrain the valuations in the successor state. The variables from $V \cup V'$, which are not in $\text{free}.d$ are not subject to any restrictions, i.e., they are permitted to have arbitrary (type-correct) valuations.

Every transition (including d^{ε}) is an assertion which fires depending on the inputs and on the internal system state (values of local variables); and it is allowed to modify the local and output variables only, i.e., for all $d \in \delta \cup \{d^{\varepsilon}\}$

$$\forall \alpha,\beta \in \Lambda(V \cup V') : (\alpha \overset{I \cup L}{=} \beta \wedge \alpha \overset{L' \cup O'}{=} \beta) \Rightarrow ((\alpha \vdash d) \Leftrightarrow (\beta \vdash d)). \tag{4.2}$$

Please note that since V and V' are disjoint, the transition domain $\Lambda(V \cup V')$ is the same as $\Lambda(V) \times \Lambda(V')$. By Constraint (4.2) we disallow an STS to constrain its own future inputs, and enforce the separation between the local state and the outputs. In this context we call the members of I *external variables* and the members of $L \cup O$ *controlled variables*.

In a single transition step the system can process a finite sequence of inputs and produce a finite sequence of outputs. A single transition assertions can comprise several successor states with differently valuated controlled variables. By this, non-determinism can be modeled.

In order to be able to reason about STS transition steps and valuation sequences which they produce we define the set $\mathrm{Succ}(\alpha,\delta) \subseteq \Lambda(V)$ which contains all successor valuations reachable from the valuation $\alpha \in \Lambda(V)$ using the transitions from δ. Formally,

$$\mathrm{Succ}(\alpha,\delta) \stackrel{\text{def}}{=} \{\beta \in \Lambda(V) \mid \exists d \in \delta : \alpha,\beta' \vdash d\}.$$

If a transition $d \in \delta$ is *enabled* (can be fired) in some state α, we state that $\mathrm{Succ}(\alpha,\{d\}) \neq \emptyset$; otherwise, this set is empty. As a natural extension of the above set, we lift its domain to sets of valuations by defining

$$\mathrm{Succ}(\mathfrak{A},\delta) \stackrel{\text{def}}{=} \bigcup_{\alpha\in\mathfrak{A}} \mathrm{Succ}(\alpha,\delta) \quad \text{for } \mathfrak{A} \subseteq \Lambda(V).$$

The last member of the STS tuple is a special transition denoted by d^ε. It is called the *idle-loop transition* or the *idle loop* for short. In the course of this chapter, we also emphasize the special meaning of idle loops by referring to the members of δ using the term "non-idle transitions". Idle loop can be taken in any state if and only if no other transition is enabled. Formally,

$$\forall \alpha \in \Lambda(V) : \mathrm{Succ}(\alpha,\{d^\varepsilon\}) \neq \emptyset \Leftrightarrow \mathrm{Succ}(\alpha,\delta) = \emptyset. \tag{4.3}$$

By this constraint STSs become *input-enabled*, i.e., they can react to any unforeseen input in a predefined uniform manner. Typically d^ε should complement the input behavior of the STS and do not alter its internal state (i.e., the valuation of variables from L). In other words, it should offer the possibility to remain in any state of the STS for an arbitrary long time. We will meet similar definitions for idle loops in different computational models, which are formalized in the remainder of this thesis using the STS framework.

***Remark* 4.1** (A weaker definition of input-enabledness)**.** An alternative approach to define input-enabledness is to demand the *existence* of an enabled transition in any reachable valuation state. However, this is not sufficient for the preservation of input-enabledness upon composition. ∘

We map the behavior of the STS $\mathcal{S}$ to the semantic domain from Section 4.1.1 – a subset of $\Lambda(V)\times\Lambda(V)$ – by

$$\{(\alpha,\beta) \in \Lambda(V)\times\Lambda(V) \mid \alpha,\beta' \vdash \bigvee_{d\in\delta\cup\{d^\varepsilon\}} d\}.$$

The definition of idle loop (Constraint (4.3)) ensures that it is fired only if no regular transition is enabled, i.e., no "useful" work can be done. We can now define the language described by an STS.

Definition 4.2 (STS runs)**.** *The* run *of an STS* $\mathcal{S} = (I,O,L,\mathfrak{I},\delta,d^\varepsilon)$ *is an infinite sequence of timed valuations* $\rho = \langle \alpha_0\alpha_1\alpha_2\ldots\rangle$*, where*

$$\alpha_0 \vdash \mathfrak{I} \qquad \textit{and} \qquad \forall i \in \mathbb{N} : \alpha_{i+1} \in \mathrm{Succ}(\alpha_i, \delta\cup\{d^\varepsilon\}).$$

The ith valuation in the run ρ ($i \in \mathbb{N}$) is denoted by $\rho(i)$ or $\rho.i$ for short. The set of all *runs of* $\mathcal{S}$ *is denoted by* $\langle\langle \mathcal{S} \rangle\rangle$. $\diamond$

We extend our notion of equality over a variable set ($\overset{(.)}{=}$) for runs. A pair of runs ρ_1 and ρ_2 consisting of valuations over the sets V_1 and V_2, respectively coincide on a certain common subset $U \subseteq V_1 \cap V_2$, denoted by $\rho_1 \overset{U}{=} \rho_2$, if and only if $\forall i \in \mathbb{N} : \rho_1.i \overset{U}{=} \rho_2.i$. For two sets of runs $\mathfrak{R}_1$ and $\mathfrak{R}_2$, we define $\overset{(.)}{\subseteq}$ and $\overset{(.)}{=}$ as

$$\mathfrak{R}_1 \overset{V}{\subseteq} \mathfrak{R}_2 \overset{\text{def}}{=} \forall \rho_1 \in \mathfrak{R}_1 : \exists \rho_2 \in \mathfrak{R}_2 : \rho_1 \overset{V}{=} \rho_2,$$
$$\mathfrak{R}_1 \overset{V}{=} \mathfrak{R}_2 \overset{\text{def}}{=} \mathfrak{R}_1 \overset{V}{\subseteq} \mathfrak{R}_2 \wedge \mathfrak{R}_1 \overset{V}{\supseteq} \mathfrak{R}_2.$$

In order to improve the readability we will sometimes access the components of an STS $\mathcal{S} = (I, O, L, \mathcal{I}, \delta, d^{\varepsilon})$ by $\mathcal{S}.I$, $\mathcal{S}.\mathcal{I}$, $\mathcal{S}.\delta$, *etc.* We will also refer to the set of its variables by $\mathcal{S}.V$.

4.1.3 From Runs to Trace Semantics

STS offers means to realize the behavior reactive real-time systems. In this context an important question arises if the realization is a valid refinement of the requirements specification. In order to be able to answer this question, we relate the operational semantics of STSs to denotational-style description of timed systems from Chapters 2 and 3. For any system run $\rho \in \langle\langle \mathcal{S} \rangle\rangle$ we can easily construct a congruent timed trace $\tau_\rho \in \overrightarrow{I \cup O}^{\kappa}$ in the time domain of naturals by mapping every valuation to its index number, i.e.,

$$\forall c \in I \cup O : \tau_\rho.c \overset{\text{def}}{=} \{(\rho.i(c), i) \mid i \in \mathbb{N}\}.$$

The following definitions generalize the transition from runs to traces for arbitrary time domains. We allow arbitrary distribution of messages over the time line, provided it preserves the original (untimed) order. The (finite) time span between adjacent messages within a timed trace is filled with empty timed events. Thus, we need to distinguish between empty valuations within runs (i.e., valuations which map all variables to $\langle\rangle$) and non-empty ones. For this purpose we define the following auxiliary functions. $is\_ne(\alpha, U) \in \{0, 1\}$ returns 0 if the parameter valuation $\alpha \in \Lambda(V)$ contains empty valuations of variables from $U \subseteq V$ and 1, otherwise. Formally,

$$\forall \alpha \in \Lambda(V), U \subseteq V : is\_ne(\alpha, U) \overset{\text{def}}{=} \begin{cases} 1 & \text{if } \exists v \in U : \alpha.v \neq \langle\rangle, \\ 0, & \text{o.w.} \end{cases}$$

Next, analogous to the $nth(\tau, i)$-function from Section 2.2.2 which returns the ith non-empty timed event from the timed trace τ, we define the partial function $ith : \langle\langle \mathcal{S} \rangle\rangle \times \mathbb{N} \mapsto M^*$ such that $ith(\rho, i)$ returns the ith non-empty valuation from ρ if exists. Formally,

$$\forall \rho \in \langle\langle \mathcal{S} \rangle\rangle, i, n \in \mathbb{N} : ith(\rho, i) = \rho.n \overset{\text{def}}{\Leftrightarrow} is\_ne(\rho.n, I \cup O) = 1 \wedge i = \sum_{k \in [0,n]} is\_ne(\rho.k, I \cup O).$$

The ith non-empty valuation has the index n in which the number of non-empty valuations occurred in the run so far is equal to i. By the next definition we characterize every STS by the corresponding timed behavior specification.

Definition 4.3 (From runs to traces)**.** *Given the STS* $\mathcal{S} = (I, O, L, \mathcal{I}, \delta, d^{\varepsilon})$ *and the time domain* $\mathbb{T}$*, the set of timed observations* $\mathcal{S}$ *permitted in* $\mathbb{T}$ *is defined as*

$$\mathbb{P}_{\mathcal{S}}^{\mathbb{T}} \stackrel{\text{def}}{=} \{\tau \in \overrightarrow{I \cup O}^{\kappa} \mid \exists \rho \in \langle\langle \mathcal{S} \rangle\rangle : \forall i \in \mathbb{N} : nth(\tau, i) \stackrel{I \cup O}{=} ith(\rho, i)\},$$

where the equality between an atomic observation $A \in \overrightarrow{C_1}^{\mathcal{A}}$ *and a variable valuation* $\alpha \in \Lambda(C_2)$ *over the given common identifier subset* $C \subseteq C_1 \cap C_2$ *is defined as*

$$(A \stackrel{C}{=} \alpha) \stackrel{\text{def}}{=} (\forall c \in C : A.c = \alpha.c). \qquad \diamond$$

By this definition we allow arbitrary but finite delays between state changes of automata. We do *not* allow desynchronization of variable valuations, e.g., by the sequentialization of inputs and/or outputs of the system. The reason is that we cannot guarantee offhand the preservation of system behavior in this case. This is however a direct consequence of deployment of several tasks on the same computation node: they are executed sequentially and the order they produce their outputs and consume their inputs becomes linear. Section 4.4 and Chapter 5 are devoted to this problem.

As a consequence of input-enabledness of STSs, $\mathbb{P}_{\mathcal{S}}^{\mathbb{T}}$ is a total relation in the sense that for every input trace $\iota \in \vec{I}^{\kappa}$ there exists a trace $\tau \in \mathbb{P}_{\mathcal{S}}^{\mathbb{T}}$ such that $\iota = \tau\lceil_I$. By this, we can construct the *black-box abstraction* of the STS $\mathcal{S} = (I, O, L, \mathcal{I}, \delta, d^{\varepsilon})$ in the time domain $\mathbb{T}$ by $p(\mathcal{S}, \mathbb{T}) \stackrel{\text{def}}{=} (I, O, F_{\mathcal{S}}^{\mathbb{T}})$ (cf. Section 3.2.1), where

$$\forall \tau \in \overrightarrow{I \cup O}^{\kappa} : F_{\mathcal{S}}^{\mathbb{T}}.(\tau\lceil_I).(\tau\lceil_O) \stackrel{\text{def}}{\Leftrightarrow} \tau \in \mathbb{P}_{\mathcal{S}}^{\mathbb{T}}.$$

Since, according to the STS semantics there is always a delay of at least one transition step between an input and the corresponding output, $p(\mathcal{S}, \mathbb{T})$ is time guarded (cf. Definition 3.3).

4.1.4 Modeling Composite Systems with STS

The data exchange between STS automata is performed using homonymous input/output variable pairs. These variable pairs are regarded as new local variables of the composed system. By this, the communication between composed machines can be interpreted as information hiding or existential quantification. The communication is not instantaneous, but delayed by one transition step. This preserves the strict causality. The communication semantics is incorporated into the notion of *composition*, which is introduced next. First of all we fix the syntactical *compatibility* of STS automata, then we define formally the composition of compatible STSs.

Definition 4.4 (Compatibility and composition of STS)**.** *A pair of STSs* $\mathcal{S}_i = (I_i, O_i, L_i, \mathcal{I}_i, \delta_i, d_i^{\varepsilon})$ *with the corresponding variable type mappings* $ty_i \subseteq V_i \times \wp(M^*)$ *for* $i \in \{1, 2\}$ *are* compatible *if and only if*

(1) they do not share any output or local variables,
(2) no automaton may modify the local variables and read inputs of the another one, and
(3) the homonymous variables have the same type.

Formally, must hold

$$V_1 \cap V_2 = (I_1 \cup I_2) \cap (O_1 \cup O_2) \qquad \textit{and} \qquad \forall v \in (V_1 \cap V_2) : ty_1(v) = ty_2(v).$$

Provided $\mathcal{S}_1$ *and* $\mathcal{S}_2$ *are compatible, we denote their* composition *by* $\mathcal{S} = \mathcal{S}_1 \parallel \mathcal{S}_2$*, where* $\mathcal{S} \stackrel{\text{def}}{=} (I, O, L, \mathcal{I}, \delta, d^{\varepsilon})$ *is an STS with the* syntactical interface

$$I \stackrel{\text{def}}{=} (I_1 \cup I_2) \setminus (O_1 \cup O_2), \qquad O \stackrel{\text{def}}{=} (O_1 \cup O_2) \setminus (I_1 \cup I_2), \qquad L \stackrel{\text{def}}{=} L_1 \cup L_2 \cup (V_1 \cap V_2).$$

The semantics *of* $\mathcal{S}$ *is given by*

$$\begin{aligned} \mathcal{I} &\stackrel{\text{def}}{=} \mathcal{I}_1 \wedge \mathcal{I}_2, \\ \delta &\stackrel{\text{def}}{=} \{d_1 \wedge d_2 \mid d_1 \in \delta_1 \cup \{d_1^{\varepsilon}\} \wedge d_2 \in \delta_2 \cup \{d_2^{\varepsilon}\}\} \setminus \{(d_1^{\varepsilon} \wedge d_2^{\varepsilon})\}, \\ d^{\varepsilon} &\stackrel{\text{def}}{=} d_1^{\varepsilon} \wedge d_2^{\varepsilon}. \end{aligned}$$ ◇

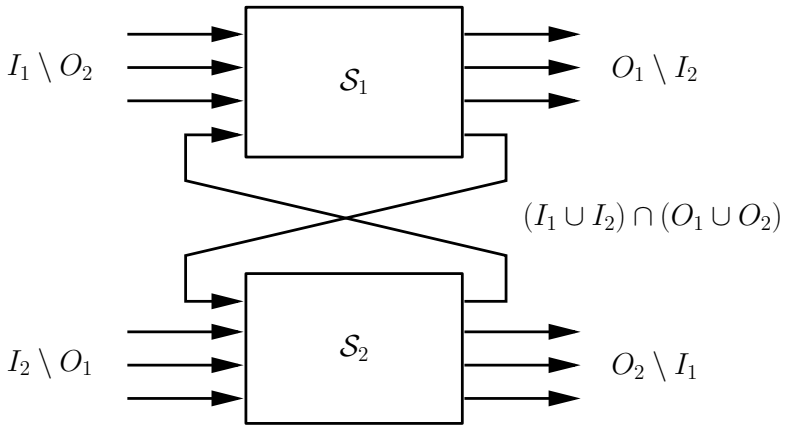


Figure 4.1: *Composition of STSs* ($\mathcal{S}_1 \parallel \mathcal{S}_2$)

Please note that compatible components are only allowed to share input/output variable pairs, i.e., to communicate with each other (cf. Figure 4.1). The input/output pairs become local variables of the resulting composite automaton. The internal state of the composed STSs remains encapsulated. Furthermore, the involved STSs may not modify the same output variables, or read the same inputs. By the former fact we prevent the possible conflicts in the output valuations already at the syntactical level. The latter restriction allows us to combine associativity of the composition operator, which is shown below, with the possibility of information hiding.

***Remark* 4.2** (One-to-one vs. one-to-n vs. one-to-$*$ communication)**.** By allowing broadcast to an arbitrary number of receivers, the interface of the composite system becomes a union of input and output variables of all its subsystems. This contradicts to the modularity, which a composition operator should exhibit: In order to support modular, compositional, and distributed development of STS models the internal implementation details of composite systems, like the

data flow between their constituting subsystems, should be hidden from the outer-world behind a well-defined interface.

The broadcast to a fixed number of receivers is a useful abbreviation of a fixed number of output variables, which have the same valuations in every reachable system sate. This description provides an indication of how the bounded broadcast can be implemented within the STS framework. ∘

The conjunctions in the definition of the composite initial-state assertion and the transition set imply that in the composite automaton $\mathcal{S}$ any initial state and any transition step are only allowed if the valuations of homonymous variables of $\mathcal{S}_1$ and $\mathcal{S}_2$ (i.e., $V_1 \cap V_2$) coincide. If neither $\mathcal{S}_1$, nor $\mathcal{S}_2$ can react to a certain combined input $\alpha \in \Lambda(V_1 \cup V_2)$, the idle-loop transition is fired.

The remainder of this section contains proofs of important properties of the introduced composition operator. First, we show that the composition of two compatible STSs yields an STS again.

Proposition 4.1 (Well-definedness of composition)**.** *Given a pair of compatible STSs* $\mathcal{S}_1 = (I_1, O_1, L_1, \mathcal{I}_1, \delta_1, d_1^\varepsilon)$ *and* $\mathcal{S}_2 = (I_2, O_2, L_2, \mathcal{I}_2, \delta_2, d_2^\varepsilon)$*, their composition* $\mathcal{S} = \mathcal{S}_1 \parallel \mathcal{S}_2 = (I, O, L, \mathcal{I}, \delta, d^\varepsilon)$ *yields an STS again, i.e., its variable sets are disjoint and it fulfills Properties* (4.1)*,* (4.2) *and* (4.3) *from Section 4.1.2.*

Also, $\mathcal{S}$ *has a non-empty set of behaviors:* $\langle\langle \mathcal{S} \rangle\rangle \neq \emptyset$*.*

Proof. Let $V \overset{\text{def}}{=} I \cup L \cup O$ and $V_i \overset{\text{def}}{=} I_i \cup L_i \cup O_i$ for $i \in \{1,2\}$.

- The disjointness of I, L, and O is obvious.
- The Property (4.1) demands the existence of at least one satisfying valuation and disallows to constrain possible valuations of input variables by initial state assertion.

  (1) Thus, the first property $\mathcal{I}$ has to fulfill is $\exists \alpha \in \Lambda(V) : \alpha \vdash \mathcal{I}$. Since $\mathcal{S}_1$, $\mathcal{S}_2$ are valid STSs, there exist some $\alpha_i \in \Lambda(V_i)$, which fulfill the respective $\mathcal{I}_i$ with $i \in \{1,2\}$ and the satisfaction of $\mathcal{I}_i$ does not depend on the valuation of the respective I_i sets. W.l.o.g. we define α as a valuation, which satisfies $\alpha \overset{L_1 \cup O_1}{=} \alpha_1$ and $\alpha \overset{L_2 \cup O_2}{=} \alpha_2$. The valuations of variables from I remain unconstrained. This definition is unambiguous, because $(L_1 \cup O_1) \cap (L_2 \cup O_2) = \emptyset$ and $(I_1 \cup L_1) \cap (I_2 \cup L_2) = \emptyset$. By this,

  $$\alpha \vdash \mathcal{I}_1 \wedge \alpha \vdash \mathcal{I}_2 \quad \Leftrightarrow \quad \alpha \vdash \mathcal{I}_1 \wedge \mathcal{I}_2 \quad \Leftrightarrow \quad \alpha \vdash \mathcal{I}.$$

  (2) The second property of $\mathcal{I}$ is $\forall \alpha, \beta \in \Lambda(V) : \alpha \overset{L \cup O}{=} \beta \Rightarrow (\alpha \vdash \mathcal{I} \Leftrightarrow \beta \vdash \mathcal{I})$. Let α be defined as in the previous point and the implication premise hold, i.e., $\alpha \overset{L \cup O}{=} \beta$. By this, β is constrained on the same variables as α, and it holds $\beta \overset{L_1 \cup O_1}{=} \alpha \overset{L_1 \cup O_1}{=} \alpha_1$ and $\beta \overset{L_2 \cup O_2}{=} \alpha \overset{L_2 \cup O_2}{=} \alpha_2$. Thus, we obtain $(\alpha \vdash \mathcal{I}_1 \wedge \mathcal{I}_2) \Leftrightarrow (\beta \vdash \mathcal{I}_1 \wedge \mathcal{I}_2)$.
- Property (4.2) states that no transition in $\delta \cup \{d^\varepsilon\}$ can choose its inputs or constrain previous outputs

  $$\forall d \in \delta \cup \{d^\varepsilon\}, \alpha, \beta \in \Lambda(V \cup V') : (\alpha \overset{I \cup L}{=} \beta \wedge \alpha \overset{L' \cup O'}{=} \beta) \Rightarrow ((\alpha \vdash d) \Leftrightarrow (\beta \vdash d)).$$

Assuming that $\alpha \overset{I \cup L}{=} \beta$ and $\alpha \overset{L' \cup O'}{=} \beta$ holds for some $\alpha, \beta \in \Lambda(V \cup V')$, we need to consider two symmetrical cases. Thus, we will prove that provided there is some $d \in \delta \cup \{d^\varepsilon\}$ such that $\alpha \vdash d$, then also $\beta \vdash d$. The proof of the other direction is analogous and thus omitted here.

By definition of composition we need to show that there exist $d_i \in \delta_i \cup \{d_i^\varepsilon\}$ $(i \in \{1,2\})$ such that $\beta \vdash d_1 \wedge d_2$, or equally that $\beta \vdash d_1$ and $\beta \vdash d_2$. Again, due to the symmetry of both conjuncts, we will show only the first one, i.e., that $\beta \vdash d_1$.

Let $\alpha_1 \overset{V_1 \cup V_1'}{=} \alpha$ and $\beta_1 \overset{V_1 \cup V_1'}{=} \beta$, then $\beta_1 \overset{I_1 \cup L_1}{=} \alpha_1$ and $\beta_1 \overset{L_1' \cup O_1'}{=} \alpha_1$, and the following property holds: $\alpha_1 \vdash d_1 \Leftrightarrow \beta_1 \vdash d_1$. The left-hand side of this logical equivalence is true, since the transition d_1, with $\alpha \vdash d_1 \wedge d_2$ is in $\delta_1 \cup \{d_1^\varepsilon\}$ and α_1 fixes all variables from free.d_1 the same way, as α does. Thus, $\beta_1 \vdash d_1$. Since the relation between β_1 and β is the same as between α_1 and α, by similar reasoning we obtain $\beta \vdash d_1$, and we are done in this case. As mentioned above, the other cases go analogously.

- The input-enabledness, expressed in Property (4.3) is proved by the following deduction chain

$$
\begin{aligned}
& \forall \alpha \in \Lambda(V) : \mathrm{Succ}(\alpha, \{d^\varepsilon\}) \neq \emptyset \Leftrightarrow \mathrm{Succ}(\alpha, \delta) = \emptyset \\
\Leftarrow \quad & (* \text{ Definition 4.4, definition of Succ, predicate calculus } *) \\
& \forall \alpha \in \Lambda(V) : (\mathrm{Succ}(\alpha, \{d_1^\varepsilon\}) \neq \emptyset \wedge \mathrm{Succ}(\alpha, \{d_2^\varepsilon\}) \neq \emptyset) \Leftrightarrow \mathrm{Succ}(\alpha, \delta) = \emptyset \\
\Leftarrow \quad & (* \text{ definition of Succ, predicate calculus } *) \\
& \forall \alpha_1 \in \Lambda(V_1), \alpha_2 \in \Lambda(V_2) : \alpha_1 \overset{V_1 \cap V_2}{=} \alpha_2 \Rightarrow ((\mathrm{Succ}(\alpha_1, \{d_1^\varepsilon\}) \neq \emptyset \wedge \mathrm{Succ}(\alpha_2, \{d_2^\varepsilon\}) \neq \emptyset) \\
& \qquad\qquad \Leftrightarrow (\mathrm{Succ}(\alpha_1, \delta_1) = \emptyset \wedge \mathrm{Succ}(\alpha_2, \delta_2) = \emptyset)) \\
\Leftarrow \quad & (* \text{ Property (4.3) for } \mathcal{S}_1 \text{ and } \mathcal{S}_2, \text{ predicate calculus } *) \\
& \text{true}
\end{aligned}
$$

- Finally, the property that $\langle\langle \mathcal{S} \rangle\rangle$ is not empty immediately follows from the input-enabledness of $\mathcal{S}$, proved above. □

The next result guarantees the property preservation upon composition. The composition operator preserves the (subsets of) runs of its operands. This results delivers the formal foundation behind the methodological meaning of component-based development: the behavior of an individual component is not changed but rather constrained by the composition.

Proposition 4.2 (Language inclusion and composition)**.** *Given a pair of compatible STSs* $\mathcal{S}_1 = (I_1, O_1, L_1, \mathcal{I}_1, \delta_1, d_1^\varepsilon)$ *and* $\mathcal{S}_2 = (I_2, O_2, L_2, \mathcal{I}_2, \delta_2, d_2^\varepsilon)$*, the following relations hold:*

$$\langle\langle \mathcal{S}_1 \,||\, \mathcal{S}_2 \rangle\rangle \overset{V_1}{\subseteq} \langle\langle \mathcal{S}_1 \rangle\rangle \qquad \textit{and} \qquad \langle\langle \mathcal{S}_1 \,||\, \mathcal{S}_2 \rangle\rangle \overset{V_2}{\subseteq} \langle\langle \mathcal{S}_2 \rangle\rangle,$$

where $V_i \overset{\text{def}}{=} I_i \cup L_i \cup O_i$ *for* $i \in \{1,2\}$.

Proof. Let $\mathbb{S}_1 \| \mathbb{S}_2 \stackrel{\text{def}}{=} \mathbb{S} = (I, O, L, \mathcal{I}, \delta, d^\varepsilon)$, with $V \stackrel{\text{def}}{=} I \cup L \cup O$. We will show that every $\rho \in \langle\langle \mathbb{S} \rangle\rangle$ is also in $\langle\langle \mathbb{S}_1 \rangle\rangle$. The remaining case is symmetrical. In accordance with Definition 4.2 we need to show that

$$\rho.0 \vdash \mathcal{I} \Rightarrow \rho.0 \vdash \mathcal{I}_1, \tag{4.4a}$$

$$\forall i \in \mathbb{N} : \rho.(i+1) \in \mathrm{Succ}(\rho.i, \delta \cup \{d^\varepsilon\}) \Rightarrow \exists \beta_1 \in \mathrm{Succ}(\rho.i, \delta_1 \cup \{d^\varepsilon\}) : \rho.(i+1) \stackrel{V_1}{=} \beta_1. \tag{4.4b}$$

- Property (4.4a) trivially holds by the definition of composition of initial valuations: $\mathcal{I} \Rightarrow \mathcal{I}_1$.
- Due to the definition of STS and Proposition 4.1, both d^ε and d_1^ε are enabled if and only if there are no enabled transitions in δ and δ_1, respectively. Thus, we obtain two sub-goals for Property (4.4b).

  $\underline{\rho.(i+1) \in \mathrm{Succ}(\rho.i, \delta)}$: In this case the property follows from the fact that every $d \in \delta$ has the form $d_1 \wedge d_2$ with $d_i \in \delta_i \cup \{d_i^\varepsilon\}$ $(i \in \{1,2\})$ and constrains more variables from V_1 than d_1 does, namely the variables from the set $I_1 \cap O_2$. Thus, $\alpha \vdash d \Rightarrow \alpha \vdash d_1$ holds for any $\alpha \in \Lambda(V \cup V')$ and by expanding the definition of Succ in (4.4b) and replacing β_1 by $\rho.(i+1)$ we obtain

  $$\begin{aligned} & \exists d \in \delta : \rho.i, (\rho.(i+1))' \vdash d \Rightarrow \exists d_1 \in \delta_1 \cup \{d_1^\varepsilon\} : \rho.i, (\rho.(i+1))' \vdash d_1 \\ \Leftrightarrow \quad & (* \text{ predicate calculus } *) \\ & \forall d \in \delta : \exists d_1 \in \delta_1 \cup \{d_1^\varepsilon\} : (\rho.i, (\rho.(i+1))' \vdash d \Rightarrow \rho.i, (\rho.(i+1))' \vdash d_1) \end{aligned}$$

  which is true.

  $\underline{\rho.i, (\rho.i+1)' \vdash d^\varepsilon}$: From to Definition 4.4 we can follow that if $d^\varepsilon = d_1^\varepsilon \wedge d_2^\varepsilon$ is enabled then so is d_1^ε. This proofs the Goal (4.4b) also for this case. □

The modularity in the development process of reactive systems using the STS formalism is facilitated by the next results, which show the associativity and commutativity of the STS composition.

Corollary 4.3 (Properties of composition)**.** *The composition operator "$\|$" is commutative and associative, i.e., for pairwise compatible STSs $\mathbb{S}_1$, $\mathbb{S}_2$ and $\mathbb{S}_3$ with $V_1 \cap V_2 \cap V_3 = \emptyset$ holds*

$$\mathbb{S}_1 \| \mathbb{S}_2 = \mathbb{S}_2 \| \mathbb{S}_1 \qquad \textit{and} \qquad (\mathbb{S}_1 \| \mathbb{S}_2) \| \mathbb{S}_3 = \mathbb{S}_1 \| (\mathbb{S}_2 \| \mathbb{S}_3).$$

Proof. Commutativity is obvious since STSs are treated symmetrically in the definition of composition.

The associativity of the syntactical interface follows from the fact that no "broadcast" is allowed by the definition of compatibility and the additional condition $V_1 \cap V_2 \cap V_3 = \emptyset$, i.e., no variable can be read or written by more than one STS at the same time. By this, the interface of the composite STS remains the same regardless of the composition order. Finally, the associativity of the composite behavior definition follows from the fact that the set-theoretic and logical operations used in that definition are associative and commutative.[8] □

[8] The definition of δ in 4.4 can be reformulated as $\{d_1 \wedge d_2 \mid d_1 \in \delta_1 \cup \{d_1^\varepsilon\} \wedge d_2 \in \delta_2\} \cup \{d_1 \wedge d_2 \mid d_1 \in \delta_1 \wedge d_2 \in \delta_2 \cup \{d_2^\varepsilon\}\}$.

Finally, we show the equality between the composition of black-box and white-box behaviors of the same STS automata. This result facilitates modular verification – every requirement satisfied by an STS automaton remains satisfied in the composition. Modularity in the verification is an important issue, since the modern verification techniques suffer under the infamous state-explosion problem. Thus, a specification formalism, which allows to bound the verification effort and guarantees the preservation of the proved properties is an important prerequisite for the establishment of formal verification as a quality assurance technique of software engineering.

Proposition 4.4 (Black-box behavior and composition)**.** *Given a pair of compatible STSs* $\mathcal{S}_1$ *and* $\mathcal{S}_2$ *with at least one common variable and a time domain* $\mathbb{T}$*, then for the corresponding black-box abstractions of* $\mathcal{S}_1$ *and* $\mathcal{S}_2$ *and for the black-box abstraction of their STS composition holds*

$$F^{\mathbb{T}}_{\mathcal{S}_1 \| \mathcal{S}_2} = F^{\mathbb{T}}_{\mathcal{S}_1} \otimes F^{\mathbb{T}}_{\mathcal{S}_2}.$$

Proof. Let $\mathcal{S}_i$ be defined as $\mathcal{S}_i = (I_i, O_i, L_i, \mathcal{I}_i, \delta_i, d_i^{\varepsilon})$ with $V_i = I_i \cup L_i \cup O_i$ $(i \in \{1,2\})$ such that $V_1 \cap V_2 \neq \emptyset$. Let $\mathcal{S}_1 \| \mathcal{S}_2 = (I, O, L, \mathcal{I}, \delta, d^{\varepsilon})$ with $V = I \cup L \cup O$ be the STS composition of $\mathcal{S}_1$ and $\mathcal{S}_2$. Finally, let the corresponding black-box abstractions of $\mathcal{S}_i$ be $p(\mathcal{S}_i, \mathbb{T}) = (I_i, O_i, F^{\mathbb{T}}_{\mathcal{S}_i})$ $(i \in \{1,2\})$ and the black-box abstraction of $\mathcal{S}_1 \| \mathcal{S}_2$ – $p(\mathcal{S}_1 \| \mathcal{S}_2, \mathbb{T}) = (I, O, F^{\mathbb{T}}_{\mathcal{S}_1 \| \mathcal{S}_2})$. We show the proposition by proving set inclusion in both directions.

"$\supseteq$": From definition of the $\otimes$-operator (cf. Definition 3.4) we know that every trace from $F^{\mathbb{T}}_{\mathcal{S}_1} \otimes F^{\mathbb{T}}_{\mathcal{S}_2}$ restricted to the respective variable set is also contained in $F^{\mathbb{T}}_{\mathcal{S}_1}$ and in $F^{\mathbb{T}}_{\mathcal{S}_2}$. Formally, for all $\tau \in \overrightarrow{I \cup O}^{\kappa}$ if $(F^{\mathbb{T}}_{\mathcal{S}_1} \otimes F^{\mathbb{T}}_{\mathcal{S}_2}).(\tau\lceil_I).(\tau\lceil_O)$, then $F^{\mathbb{T}}_{\mathcal{S}_1}.(\tau\lceil_{I_1}).(\tau\lceil_{O_1})$ and $F^{\mathbb{T}}_{\mathcal{S}_2}.(\tau\lceil_{I_2}).(\tau\lceil_{O_2})$ and, thus, $\tau \in \mathbb{P}^{\mathbb{T}}_{\mathcal{S}_1}$ and $\tau \in \mathbb{P}^{\mathbb{T}}_{\mathcal{S}_2}$. Moreover, due to Definition 4.3 there exist a pair of runs ρ_1 and ρ_2 from $\langle\langle \mathcal{S}_1 \rangle\rangle$ and $\langle\langle \mathcal{S}_2 \rangle\rangle$, respectively such that for all $i \in \mathbb{N}$, $nth(\tau, i) \overset{I_1 \cup O_1}{=} ith(\rho_1, i)$ and $nth(\tau, i) \overset{I_2 \cup O_2}{=} ith(\rho_2, i)$. Thus, in every point ρ_1 and ρ_2 coincide on the valuations of shared variables (since STSs are compatible, their common variables build the set $(I_1 \cup I_2) \cap (O_1 \cup O_2)$). Due to the assumption that there exists at least one common variable, we conclude that for all $i \in \mathbb{N}$, $ith(\rho_1, i) \overset{V_1 \cap V_2}{=} ith(\rho_2, i)$. By this, due to the definition of $\|$-composition, there exists $\rho \in \langle\langle \mathcal{S}_1 \| \mathcal{S}_2 \rangle\rangle$, with $\rho \overset{I_1 \cup O_1}{=} \rho_1$ and $\rho \overset{I_2 \cup O_2}{=} \rho_2$. Finally, we obtain $\tau \in \mathbb{P}^{\mathbb{T}}_{\mathcal{S}_1 \| \mathcal{S}_2}$ and, thus, $F^{\mathbb{T}}_{\mathcal{S}_1 \| \mathcal{S}_2}.(\tau\lceil_I).(\tau\lceil_O)$.

"$\subseteq$": For every trace $\tau \in \overrightarrow{I \cup O}^{\kappa}$ such that $F^{\mathbb{T}}_{\mathcal{S}_1 \| \mathcal{S}_2}.(\tau\lceil_I).(\tau\lceil_O)$ and, consequently, $\tau \in \mathbb{P}^{\mathbb{T}}_{\mathcal{S}_1 \| \mathcal{S}_2}$, there exist a run $\rho \in \langle\langle \mathcal{S}_1 \| \mathcal{S}_2 \rangle\rangle$ such that for all $i \in \mathbb{N}$, $nth(\tau, i) \overset{I \cup O}{=} ith(\rho, i)$. From Proposition 4.2 we know that a corresponding restriction of ρ to V_1/V_2 belongs to $\langle\langle \mathcal{S}_1 \rangle\rangle / \langle\langle \mathcal{S}_2 \rangle\rangle$, respectively. Let $\rho_1 \in \langle\langle \mathcal{S}_1 \rangle\rangle$ and $\rho_2 \in \langle\langle \mathcal{S}_2 \rangle\rangle$ be these restrictions, then there exist corresponding traces $\tau_1 \in \mathbb{P}^{\mathbb{T}}_{\mathcal{S}_1}$ and $\tau_2 \in \mathbb{P}^{\mathbb{T}}_{\mathcal{S}_2}$ such that for all $i \in \mathbb{N}$, $nth(\tau_1, i) \overset{I_1 \cup O_1}{=} ith(\rho_1, i) \overset{I_1 \cup O_1}{=} ith(\rho, i)$ and $nth(\tau_2, i) \overset{I_2 \cup O_2}{=} ith(\rho_2, i) \overset{I_2 \cup O_2}{=} ith(\rho, i)$. By this, hold $F^{\mathbb{T}}_{\mathcal{S}_1}.(\tau\lceil_{I_1}).(\tau\lceil_{O_1})$ and $F^{\mathbb{T}}_{\mathcal{S}_2}.(\tau\lceil_{I_2}).(\tau\lceil_{O_2})$ and thus, in accordance with Definition 3.4, $(F^{\mathbb{T}}_{\mathcal{S}_1} \otimes F^{\mathbb{T}}_{\mathcal{S}_2}).(\tau\lceil_I).(\tau\lceil_O)$. □

***Remark* 4.3** (Composition and strict causality)**.** In order to check the fulfillment of time requirements by STS automata, we can map the automata to their respective behaviors within the trace domain and use the results from Section 3.2.3 for the subsequent verification. All those results

have the strict causality of components as a prerequisite (cf. Definition 3.3). As argued in Section 4.1.3 due to Property (4.2) timed behavior specifications which represent single STSs have this property. It is easy to show that the STS composition operator preserves the strict causality, i.e., given a pair of compatible STSs $\mathcal{S}_1$ and $\mathcal{S}_2$ and a time domain $\mathbb{T}$, then $F^{\mathbb{T}}_{\mathcal{S}_1 \| \mathcal{S}_2}$ is strict causal. This follows directly from the well-definedness of the STS composition operator (shown by Proposition 4.1). ◦

4.1.5 Refinement

In line with the denotational framework from Chapters 2 and 3 we model valid development steps in a top-down manner by ensuring the refinement relation between their respective outcomes. Using the notion of runs from Section 4.1.3 we can define the refinement relation between STSs as a language inclusion.

Definition 4.5 (Refinement of STSs)**.** *An STS* $\mathcal{S}_1$ *is a* refinement *of an STS* $\mathcal{S}_0$*, denoted by* $\mathcal{S}_1 \rightsquigarrow \mathcal{S}_0$ *if and only if*

$$V_0 \subseteq V_1 \qquad \text{and} \qquad \langle\!\langle \mathcal{S}_1 \rangle\!\rangle \overset{V_0}{\subseteq} \langle\!\langle \mathcal{S}_0 \rangle\!\rangle,$$

where V_0/V_1 *is a variable set of* $\mathcal{S}_0$/$\mathcal{S}_1$*, respectively.* ◇

By the above definition we allow the refinement to have a bigger state space than the corresponding abstraction. However, the possible runs on common variables are more restricted by the refinement. We can interpret this by stating that any variable v from $V_1 \setminus V_0$ is not constrained by $\mathcal{S}_0$, i.e., in every state of $\mathcal{S}_0$ it is allowed to have an arbitrary type-correct valuation. Thus, we obtain a subset relation as the definition of refinement. By this, the reflexivity and transitivity of STS refinement are obvious. Further on, we trivially obtain the following relationship between refinement and composition.

Corollary 4.5 (Refinement monotony re composition)**.** *Given two STS pairs* $(\mathcal{S}^a_1, \mathcal{S}^c_1)$ *and* $(\mathcal{S}^a_2, \mathcal{S}^c_2)$ *such that* $\mathcal{S}^c_1$ *and* $\mathcal{S}^c_2$ *are compatible and* $\mathcal{S}^a_i \rightsquigarrow \mathcal{S}^c_i$ *for* $i \in \{1, 2\}$*. Then,*

$$(\mathcal{S}^a_1 \| \mathcal{S}^a_2) \rightsquigarrow (\mathcal{S}^c_1 \| \mathcal{S}^c_2).$$

Proof. Due to Definition 4.5, $\mathcal{S}^a_1$ and $\mathcal{S}^a_2$ are compatible either. Then, the property follows from Definitions 4.4, 4.5 and Proposition 4.2. □

From the methodological point of view the compositionality of refinement is very important [Bro98], since it enables to independently refine the individual components without affecting the satisfaction of global properties by the whole system.

In the rest of this thesis we show the property preservation by proving simulation relations. A simulation relation between STSs is defined in A.6. We have already argued in Section 3.1.1 that the definitions of refinement and simulation relation coincide for operational formalisms,

like STS. In fact, for two STSs $\mathcal{S}_0$ and $\mathcal{S}_1$ such that $\mathcal{S}_1 \rightsquigarrow \mathcal{S}_0$, holds $\mathcal{I}_1 \Rightarrow \mathcal{I}_0$ and for all states $\alpha \in \Lambda(V_1)$ reachable in $\mathcal{S}_1$

$$\mathrm{Succ}(\alpha, \delta_1) \overset{V_0}{\subseteq} \mathrm{Succ}(\alpha, \delta_0).$$

Thus, we can easily show that $\mathcal{S}_0$ simulates $\mathcal{S}_1$ by constructing a corresponding simulation relation. Vice versa, if $\mathcal{S}_0$ simulates $\mathcal{S}_1$ then $\mathcal{S}_1$ refines $\mathcal{S}_0$.

4.1.6 STS for Technical Systems

The concepts, like the encapsulation of the internal state, the strong causality (i.e., a unit-length delay in the synchronization between outputs and inputs), as well as the parallel execution are well-established and useful abstractions for the design and the implementation of the reactive distributed systems. On the other hand, the actual application code is deployed onto a technical infrastructure, which uses quite different execution (scheduling) policies and communication mechanisms. In general, it can directly manipulate the "internal" state of the application, it can exhibit behaviors, in which the sending and receiving applications stay desynchronized for several system steps and it cannot always execute several applications in parallel. In other words, the composition semantics do not match. Moreover, the composition is partly dictated by the deployed components themselves: The components get assigned priorities or fixed activation times and by this *parametrize* their composition. Thus, this type of composition is in general not associative. The current section is devoted to the problem how technical systems can be modeled using the STS framework.

Scheduled STS

The task of proving property preservation between a model, e.g., an instance of a certain design paradigm and its realization (a technical system parametrized with the application code) demands for a formal representation of both composition paradigms within the same formal framework. In the present thesis STS is our formalism of choice. We define a *scheduled STS system* of n application automata $\mathcal{S}_1, \ldots, \mathcal{S}_n$ by an STS $\mathcal{S}\mathsf{ch} \overset{\text{def}}{=} (I, O, L, \mathcal{I}, \delta\mathsf{ch}, d^{\varepsilon})$. Let $C_I \overset{\text{def}}{=} \cup_{i\in[1,n]} I_i$, $C_O \overset{\text{def}}{=} \cup_{i\in[1,n]} O_i$, $C_L \overset{\text{def}}{=} \cup_{i\in[1,n]} L_i$ and $C_V \overset{\text{def}}{=} C_I \cup C_L \cup C_O$. Then, the following constraints must hold for $\mathcal{S}\mathsf{ch}$:

$$\mathcal{S}\mathsf{ch}.V \supseteq C_V, \qquad \mathcal{S}\mathsf{ch}.O \cap (C_I \setminus C_O) = \emptyset, \qquad \mathcal{S}\mathsf{ch}.I \cap (C_O \setminus C_I) = \emptyset.$$

The first formula states that the state space of $\mathcal{S}\mathsf{ch}$ cannot downsize. Both remaining constraints disallow variables to change their "direction". The scheduled automaton itself is allowed to have its own local state space as well as additional input and output variables. It is also allowed to hide the input and output variables of its constituents in its local state space as well as to make the local variables accessible by the environment. As argued above, in general we cannot state anything about the relation between the behavior of system constituents and the overall behavior of the composed system. The same is true for the initial states. However, we will define a number of scheduled STS systems, which respect the original transition functions, they are built of.

The STSs $\mathcal{S}_1, \ldots, \mathcal{S}_n$ need not to be (pairwise) compatible, nor the communication between them has to happen delayed by exactly one system step. The regulation of possible write conflicts, casting of mismatching variable types, artificially disjoining of overlapping state spaces, and the maintenance of discontinued data-flow links is left to the semantics of their scheduled composition. However, type casts lie beyond the scope of the presented work, thus in the rest of the thesis we will combine STSs which are at least *type compatible*. A pair of STSs is type compatible if and only if they fulfill the third property of the compatibility notion from Definition 4.4, i.e., all homonymous variable pairs have the same type.

Describing Complex Technical Systems

Next we provide *renaming* as a mechanism, which allows to decouple state spaces as well as certain data-flow dependencies. The assertion $\Psi[w/v]$ is generated out of the assertion Ψ by replacing the variables $v, v' \in \text{free}.\Psi$ by the variables w and w', respectively. The variables must have the same type. For a given STS $\mathcal{S} = (I, O, L, \mathfrak{I}, \delta, d^{\varepsilon})$, a variable $v \in V \stackrel{\text{def}}{=} I \cup L \cup O$, and the equally typed new variable w (i.e., $ty(v) = ty(w)$ and $w \notin V$) we define the STS

$$\mathcal{S}[w/v] \stackrel{\text{def}}{=} (\overline{I}, \overline{O}, \overline{L}, \mathfrak{I}[w/v], \overline{\delta}, d^{\varepsilon}[w/v])$$

with $\overline{V} \stackrel{\text{def}}{=} \overline{I} \cup \overline{O} \cup \overline{L}$ as

$$\forall X \in \{I, L, O\} : \overline{X} \stackrel{\text{def}}{=} \begin{cases} (X \setminus \{v\}) \cup \{w\} & \text{if } v \in X, \\ X, & \text{o.w.}, \end{cases}$$

$$\overline{\delta} \stackrel{\text{def}}{=} \{d[w/v] \mid d \in \delta\}.$$

We lift the renaming operations for the sets of variables for assertions and STSs by $\Psi[v_w/v]_{v \in U}$ and $\mathcal{S}[v_w/v]_{v \in U}$, respectively. The operations are defined as the subsequent renaming of every $v \in U$ by v_w. They are defined only if $U \subseteq V$, $\{v_w \mid v \in U\} \cap V = \emptyset$, and $\forall v \in U : ty(v) = ty(v_w)$.

The behavior of technical systems is rather complex. In order to make their formalization by scheduled STSs more comprehensible, we will split the description of their transition steps in several *transition segments*. For this purpose we define *compound* transitions. A compound transition has the form $d_1 \circ d_2$, where d_1 and d_2 are assertions over a variable set $V \stackrel{\text{def}}{=} I \uplus L \uplus O$ and their combination is defined as

$$\forall \alpha, \beta \in \Lambda(V) : \alpha, \beta' \vdash d_1 \circ d_2 \stackrel{\text{def}}{\Leftrightarrow} \exists \gamma \in \Lambda(V) : (\alpha, \gamma' \vdash d_1) \wedge (\gamma, \beta' \vdash d_2). \tag{4.5}$$

Please note that d_1 is allowed to constrain primed input variables and d_2 unprimed output variables. Their combination can still satisfy Condition (4.2), since the primed variables of d_1 and the unprimed variables of d_2 describe the intermediate states of compound transition.

Two Simple Scheduled STS Models

In the remainder of this section we present two simple scheduled systems for n pairwise compatible STS automata $\mathcal{S}_1, \ldots, \mathcal{S}_n$ without broadcast variables, i.e., for all $i, j, k \in [1, n]$ holds

$V_i \cap V_j \cap V_k = \emptyset$. Let $\mathcal{S} = \mathcal{S}_1 \parallel \cdots \parallel \mathcal{S}_n \stackrel{\text{def}}{=} (I, O, L, \mathcal{I}, \delta, d^{\varepsilon})$ be their parallel composition. Then, the STS

$$\mathcal{S}\mathsf{ch}^{par} \stackrel{\text{def}}{=} \mathcal{S}$$

corresponds to the parallel composition. This means that the parallel composition form Section 4.1.4 is a special case of the scheduled one. Further on, the STS

$$\mathcal{S}\mathsf{ch}^{int} \stackrel{\text{def}}{=} (I, O, L, \mathcal{I}, \delta\mathsf{ch}, d^{\varepsilon})$$

with

$$\delta\mathsf{ch} \stackrel{\text{def}}{=} \bigcup_{i \in [1,n]} \{d_i \wedge \bigwedge_{j \in [1,n] \setminus \{i\}} d_j^{\varepsilon} \mid d_i \in \delta_i\}$$

expresses the interleaved composition semantics, where in any state exactly one component is allowed to make a non-idle transition.

To summarize the current section, we defined a formal framework, which allows logical characterization of the operational-style description of system behavior. In the remainder of this thesis this framework will be instantiated several times with different modeling paradigms, which come into use at different development phases of reactive systems. The common mathematical foundation given by the STS framework allows us to show refinement relations between different modeling paradigms in form of simulation theorems.

4.2 Modeling Application Logic

In the line with the authors of [SPHP02a] we believe that

> "the model-based development is a paradigm for system development that besides the use of (graphical) domain-specific languages includes explicit and *operational* descriptions of the relevant entities that occur during development..."

However, we add a substantial amendment to this definition, namely that aspects of the system, which cannot be modeled by the operational semantics in certain development phases, e.g., the time behavior during the design, can be formulated in a *denotational* style, provided the integration with the rest of the system model is formally well-defined and well-founded. Our STS formalism allows to describe the operational semantics of reactive systems and is linked to their denotational domain presented in Chapter 2 (cf. Sections 4.1.3 and 4.1.4). This link allows to assign time behavior to particular STS automata. Thus, STS fits the extended definition of model-based development and will be used in the rest of this thesis to describe artifacts of our reference development process from Chapter 1. In particular the current section presents the semantics of the CASE tool AutoFocus [HSE97]. As argued by Schätz *et al.* in [SPHP02a, SPHP02b], this tool suffices the original definition listed in the quotation above and is formally founded by a restricted version of untimed FOCUS. Consequently, AutoFocus can be extended to fit our definition from above.

AutoFocus is designed for the graphical specification of reactive systems. The tool employs a formal time-synchronous operational semantics. That formal semantics permits the simulation of created models [HMR+98], their export to verification tools, like model checkers, SAT-solvers, theorem provers, *etc.* [PS99], as well as the generation of production code (e.g., ANSI C, Java, or Ada Code), cf. [Slo98, BHL+02]. With its graphical specification language, AutoFocus supports different views on the system, namely *structure*, *behavior*, *interaction* and *data-type view*. The formal semantics of the tool integrates these views in a concise and unambiguous way. This section gives a rather short introduction to AutoFocus. A more detailed description of AutoFocus and its diagram types can be found in [HSE97, BBR+05]. After an informal introduction in 4.2.1, the STS semantic of the AutoFocus design paradigm is given in Section 4.2.2.

4.2.1 Description Techniques in AutoFocus

Besides the time behavior, structure (also called architecture), functional behavior and data types are important views on a system. AutoFocus provides special diagram types for the specification of these views and integrates them to the overall system model using a certain computation and communication paradigm. All these ingredients are presented next.

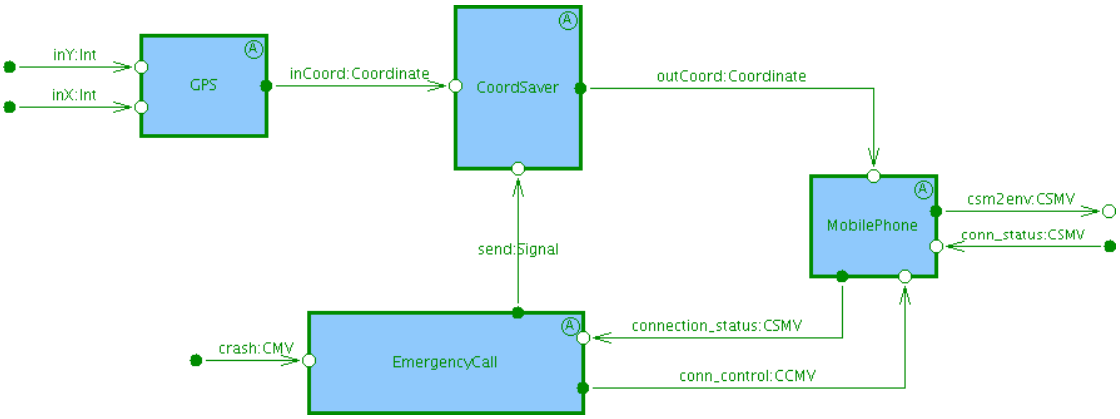


Figure 4.2: *The Component Architecture in AutoFocus (SSD) of the eCall Study*

Structural View: System Structure Diagrams (SSDs)

A system in AutoFocus is modeled by a network of components, graphically denoted as rectangles. Figure 4.2 shows the SSD of the emergency call application model (cf. Section 1.4). The communication interface of a component is defined by a set of typed directed *ports*. There are *input* and *output* ports, represented by empty and solid circles, respectively. The components communicate via typed *channels*, which connect input and output port pairs of matching types. Components networks are specified using the so called *system structure diagrams (SSDs)*. A component can be refined by a further component network. This results in a hierarchy of SSDs. By this, in a white-box view of a refined component, ports not connected to any sub-component

represent an interface to the rest of the world. These ports have counterparts in the black-box view with an opposite direction, e.g., solid circles in Figure 4.2, which represent output ports not assigned to a component become empty cycles, i.e., input ports in the black-box view of the component, specified by the SSD from the picture. The leaf components in the component hierarchy have assigned state space and behavior, described by *state transition diagrams*.

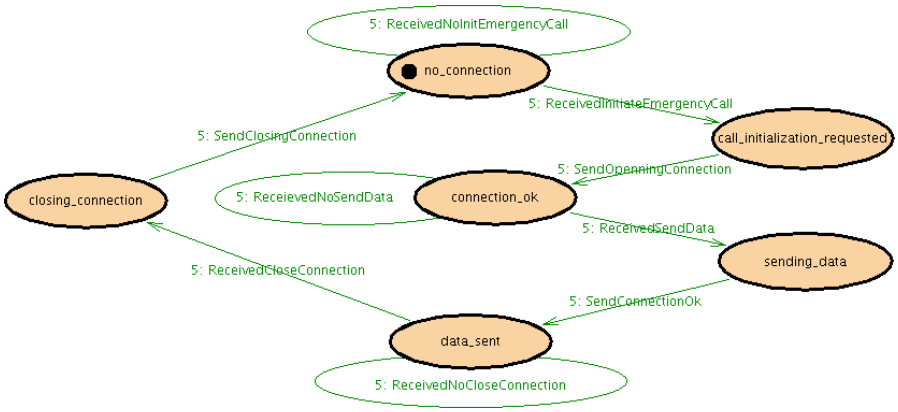


Figure 4.3: *State Transition Diagram (STD) of MobilePhone from eCall Study*

Behavioral View: State Transition Diagrams (STDs)

The behavior of a leaf component is defined by an extended I/O automaton [LT89]. It is based upon a set of *control states*, *local variables* and *transitions*. Graphically (cf. Figure 4.3), control states are denoted by ellipses and transitions by arrowed arcs. A black dot on the left side of an ellipse denotes an initial control state. For arcs with the same ellipse as source and destination the arrow head is omitted in the diagram.

An STD automaton is completely defined by its set of control states, the set of local variables along with their respective types, the initial state (of both control state and local variables), and the transition relation. The specification of a transition consists of four elements[9]:

pre-condition Transition precondition (a Boolean expression referring to local variables of the automaton and the ports of the corresponding component).

input patterns Input pattern that shows what message must be available on which port in order that the transition is triggered.

output patterns Output pattern that shows what message is written to which port as a result of this transition.

post-condition Transition post-condition (a Boolean expression referring to local variables of the automaton and the ports of the corresponding component).

[9] In order to improve the comprehensibility, the transitions in Figure 4.3 are decorated with symbolic labels, not their assigned specifications.

A transition can be fired in some state if the input ports specified in the input patterns have received the necessary input messages and the pre-condition is satisfied. The transition outputs data to output ports according to the output patterns, changes the local variables according to the post-condition, and puts the automaton into the next control state. Every of the four transition components from above are specified by a functional language expression (see below). A transition specification can contain special *temporary variables*, which are typically declared using type inference and pattern matching in the input patterns and have the scope bounded by one transition. The consistency mechanisms of AutoFocus ensure that the name space of these variables is disjoint with the names of local variables and ports, i.e., the scope overlay is not an issue in AutoFocus.

Example 4.1 (Specification of an AutoFocus transition)**.** The transition with label ReceivedNoCloseConnection from Figure 4.3 has the form:

```
(x != close_connection):
connection_control?x:
connection_status!data_sending_ok,idle!Present
```

It is fired if it gets a signal different from close_connection on port connection_control and sends the data_sending_ok signal through port connection_status and the Present signal via port idle. The postcondition is empty. x is the temporary variable of the transition. ◦

The AutoFocus automata are input-enabled – an idle-loop transition is implicitly[10] defined for every control state. It is fired only if no other transition is enabled in the current state. We will learn more about the behavior of AutoFocus (idle-loop) transitions in the course of defining the AutoFocus semantics (cf. Section 4.2.2).

Data-Type View: Data Type Definitions (DTDs)

AutoFocus models may be extended by user-defined data-type and function definitions in a Gofer-like ([Jon93]) functional language. The corresponding text documents are called *data type definitions* (*DTDs*). The symbolic function and data type names declared in these documents are referenced in the definitions of transitions, channels, ports, and variables.

Computation and Communication Semantics

AutoFocus components perform their computation steps simultaneously, driven by a global clock. Every computation step consists of two phases: First, a component reads values on the input ports and computes new values for local variables and output ports. After the time tick new values are copied to the output ports, where they can be accessed immediately via the input ports of connected components and the cycle is repeated. This results in a *time-synchronous* communication scheme with buffer size one.

[10]The idle loops are not shown in STDs.

***Remark* 4.4** (Further view)**.** The redundant view on the system not mentioned here is the interaction view, implemented in AutoFocus by the use of *extended event traces* (*EETs*). EETs capture exemplary simulation runs of the system in form of an extended version of sequence chart diagrams. ◦

eCall Case Study (*con'ed*). Figure 4.2 shows the SSD of the eCall application, which is introduced in Section 1.4. It consists of four components, namely

GPS determines the current coordinates of the vehicle and sends them to the CoordSaver component.

CoordSaver keeps the vehicle's position and sends it to the mobile phone upon the corresponding command from the eCallLogic component.

eCallLogic initiates an emergency-call procedure if a crash has happened. This is done in the cooperation with the MobilePhone component. The wireless-link for crash-data transmission can break at any stage of the emergency call procedure – in these cases eCallLogic resumes the procedure until the transmission succeeds.

MobilePhone establishes the wireless-connection with the remote operator and sends data through it. Thereby, it is controlled by the eCallLogic and reports the status (i.e., accomplished/canceled) of every operation to this component. The STD of the MobilePhone component is shown in Figure 4.3.

4.2.2 AutoFocus Semantics

This section describes the operational semantics of the AutoFocus CASE tool using our STS framework. We have made a number of simplifications in its definition.

- First of all, we observe that the component hierarchy built by SSDs is pure syntactic sugar: every hierarchical model can be flattened without any consequences to its behavior.
- Secondly, we will model the AutoFocus behavior as a sequence of atomic transition steps. We will not investigate how the state changes performed by these transitions are actually computed. From Section 4.2.1 we know that the firing of a transition in AutoFocus corresponds to the evaluation of a functional language expression. However, the behavior of transitions could be also realized, e.g., by ANSI C programs, – this is insignificant for the semantic model presented below. This fact explicitly enables to reason about the preservation of system behavior during the transition from a CASE tool model to the production code, under the assumption that the behavior of particular transitions remains the same (cf. Chapter 5 for a more in-depth discussion of this topic).
- Finally, we will not mention some special features of AutoFocus, like transition prioritization or special acyclic zero-delay channels, since they would groundless complicate the discussion.

We represent an AutoFocus (leaf) component as an STS $\mathcal{S}^{af} \stackrel{\text{def}}{=} (I, O, L, \mathfrak{I}, \mathcal{R}, \delta_{\mathsf{af}}, d^{\varepsilon})$, which has exactly one input/output variable for each input/output port, respectively. We will also refer to

the elements from I/O as *input/output port variables*, respectively, and to the STS itself as *AutoFocus STS (component)*. The type of every port variable has a special *bottom*-value, denoted by $\perp$, which encodes the absence of the actual signal. The local variables and the control state comprise the internal state of the STS component L, whereas we model the control state as a designated local variable with the set of all control states as its type domain. The initial state of the automaton corresponds to the initial state of the STS and fulfills Constraint (4.1).

In the initial state of an AutoFocus component no outputs are present. We express this by the following constraint on the initial state assertion of $\mathcal{S}^{af}$

$$\forall \alpha \in \Lambda(V) : \alpha \vdash \mathcal{I} \Rightarrow \forall v \in O : \alpha.v = \perp. \tag{4.6}$$

A light deviation from the STS communication semantics defined in Section 4.1.4 is the "flush"-semantics of AutoFocus input ports: An AutoFocus automaton produces a new output on every step on some of its output ports. The output may also depend from valuations of some of its input ports only. If no actual output is foreseen in some transition on some port, the $\perp$-value is sent out. By this, the old valuations of connected input ports are replaced by $\perp$ ("flushed"), signalizing the absence of input for the next system step. For these reasons we represent the "native" AutoFocus transition system by a set of predicates $\mathcal{R}$. Every of its elements $r \in \mathcal{R}$ represents one AutoFocus transition and it holds $\text{free}.r \subseteq I \cup L \cup L' \cup O'$. For instance, the transition presented in Example 4.1 constrains the output variables connection_status and idle and is dependent from the input variable connection_control, i.e.,

$$\text{free}.r_{\text{ReceivedNoCloseConnection}} = \{\text{connection\_control}, \text{connection\_status}', \text{idle}'\}.$$

Then, we define the AutoFocus STS transition system as a set δ_{af} which contains for every AutoFocus transition $r \in \mathcal{R}$ a corresponding transition $d \in \delta_{\text{af}}$ such that the following holds:

- For the valuations of primed and unprimed variables from $\text{free}.r$ d and r behave equivalently.
- The primed output variables not mentioned in r, i.e., the variables from the set $O' \setminus \text{free}.r$, are reset in d and the primed local variables from $L' \setminus \text{free}.r$ keep their values.

Formally, we define the following property:

$$\forall \alpha \in \Lambda(V \cup V') : \alpha \vdash d \overset{\text{def}}{\Leftrightarrow} (\alpha \vdash r) \wedge \bigwedge_{l' \in L' \setminus \text{free}.r} \alpha.l' = \alpha.l \wedge \bigwedge_{o' \in O' \setminus \text{free}.r} \alpha.o' = \perp. \tag{4.7}$$

For all $d \in \delta_{\text{af}}$ there exists $r \in \mathcal{R}$ such that Property (4.7) holds, and for all $r \in \mathcal{R}$ there exists $d \in \delta_{\text{af}}$ for which (4.7) holds, too. Please note that δ_{af} can be built out of the original set $\mathcal{R}$ fully schematically, by just binding the unconstrained variables in every AutoFocus transition to the respective $\perp$-/previous values. We assume that all transitions from $\mathcal{R}$ and by this all transitions from δ_{af} fulfill Property (4.2).

The idle loop in AutoFocus is defined as the complement of the regular transitions. It preserves the valuations of local variables and resets the output variables to $\perp$. This guarantees that the

valuation of every connected input variable was generated exactly one system step ago. Formally for all $\alpha,\beta \in \Lambda(V)$,

$$\alpha,\beta' \vdash d^{\varepsilon} \overset{\text{def}}{\Leftrightarrow} \left(\forall \gamma \in \Lambda(V) : \alpha,\gamma' \vdash \neg \bigvee_{d \in \delta\mathsf{af}} d\right) \wedge \bigwedge_{v \in L} \alpha.v = \beta.v \wedge \bigwedge_{v \in O} \beta.v = \bot.$$

It is easy to see that $\mathcal{S}^{af}$ is well-defined. In particular, it satisfies Properties (4.1)-(4.3).

AutoFocus Composition

An AutoFocus model is a network of interconnected AutoFocus components. Accordingly, we define *AutoFocus STS model* $\mathcal{S}\mathsf{ch}^{af}$ as a scheduled STS system parametrized by n pairwise compatible AutoFocus STS components. The channels are modeled as expected by homonymous output/input variable pairs. The components run in parallel and thus we compose AutoFocus components using the parallel composition from Definition 4.4:

$$\mathcal{S}\mathsf{ch}^{af} \overset{\text{def}}{=} \mathcal{S}\mathsf{ch}^{par}.$$

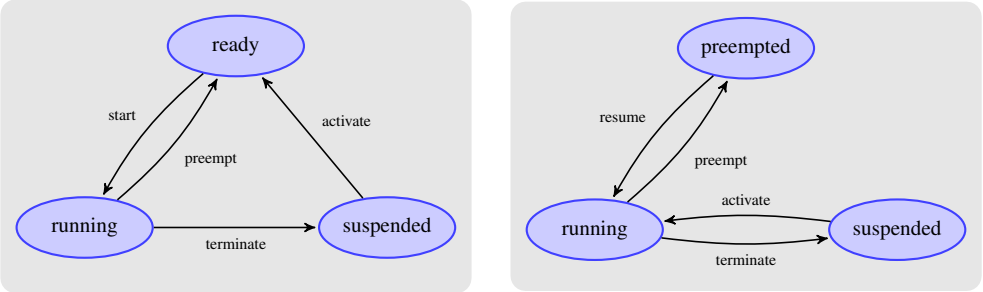
4.3 Clustering Application Logic

Clustering of the functionality is an intermediate step on the way from the model of pure application logic to the deployed system. The clustering process pursues several goals and applies different (heuristic) methods to achieve them. For example, strongly dependent components are grouped together in order to localize and by this reduce the inter-task communication and coordination effort. On the other hand, clustering can be done only to an extent, which allows to meet all time constraints imposed on the system by its specification. Thus, another heuristic groups functions with the similar activation rates together. There exist a number of further heuristic methods and best-practices, which help the system engineer to find an acceptable solution. In the present thesis we leave her alone with this problem, since it is highly application- and platform-dependent. We rather concentrate our attention on conditions, which every clustering and deployment solution has to fulfill in order to preserve the properties of the application logic.

Recalling our design process depicted in Figure 1.1, page 5, the artifact produced by the clustering process is called task architecture (TA). It serves as an input for two subsequent activities: the schedulability analysis and the deployment. These activities and the ultimate goal to ensure the property preservation between application logic and their respective outcomes induce further requirements on an adequate modeling paradigm for TA. Beyond a simple grouping facility, the responsibilities of TA are twofold:

- It must allow to describe the modeled system on the level of abstraction, expected by the different analytical schedulability techniques.
- The deployable units also known as *tasks* must be produced out of the TA model, e.g., by code and wrapper generation and subsequent compilation.

In both cases the property preservation must be ensured; otherwise, it can happen that the results of the schedulability analysis are of no significance for the deployed system and/or the behavior of the deployed system deviates from the behavior of the application logic in an undesired manner. We will discuss the determining factors for the design of a TA paradigm in the following paragraphs.



(a) *OSEK OS (the Basic Version).* **(b)** *OSEKtime OS.*

Figure 4.4: *Task State Models*

The goal of the schedulability analysis is, given a task model, a set of real-time constraints and a description of the deployment platform, to find a *valid* schedule. A schedule is called valid if it meets all real-time constraints for a given set of tasks run on the given deployment platform. The task model expected by the schedulability analysis approaches, is an abstraction of the actual system behavior, which simplifies the state space of the function clusters (tasks) as well as their input and output behavior. The reason for this is that the operating systems and bus protocols cannot differentiate arbitrary complicated task activation conditions and communication schemes. The event-triggered run-time environments are usually able to recognize at most the *presence* of certain signals. The time-triggered systems react even only to one kind of input events – timeouts, which come from their own timers. Normally, the operating systems are also not aware of different internal states of their tasks. They rather operate with simplified task state models, e.g., cf. the task state models of OSEK [OSE01a] and OSEKtime [OSE01b] depicted in Figure 4.4. The transition function of these models is realized by a set of interrupt service routines (ISRs), provided by these OSes. Every state of a task state model is an abstraction of a set of the actual states which can be reached by a task.

The above discussion shows that both the technical systems and the analytical schedulability models work with abstractions of the actual behavior of their tasks. In this context one question naturally arises: Do these abstractions match? The problem of correct transition from a concrete application model and deployment platform behavior to an analytical abstract model is not covered by any approach so far. The usage of "stand-alone" formalisms, like timed automata [AD94, FKPY07], holds a risk of the wrong reproduction of an original problem without a possibility to guarantee the correctness of or to discover a deviation in its abstract representation.

Thus, in the face of our goal to guarantee property preservation in every development step, we must ensure that, on the one hand, the TA model respects the communication and computation

principles incorporated in technical systems. On the other hand, it must embed and adapt the application logic in a way, which does not alter its behavior.

In the next section we present a reference task model, which captures the simplified notion of the application and platform behavior, different schedulability techniques and technical platforms rely on. Afterwards, in Section 4.3.2 we present a TA model called AFTM. It takes the above considerations into account and instantiates the reference task model from Section 4.3.1.

4.3.1 A Reference Task Model

Different task models which can be found in the literature (cf. [SAÅ+04, Liu00, Fer03, Jos01, HSF+04]), use slightly different vocabularies to describe and systematize the concepts real-time systems are working with. This section gives a short introduction of these concepts with emphasis lying on the behavior of the task model, rather than its timing constraints.

A *task* is a portion of system functionality technical systems can deal with. During a lifeline of the system this functionality is executed by the platform in a (potentially infinite) series of finite runs, called *jobs*. We can see a job as an instance of a task: it is the time and functional behavior a task exhibits for certain inputs and internal state. Tasks and their jobs may have *parameters*, such as deadline, worst case execution time, *etc.*, as well as *release* or *activation conditions*, which describe when a task can be activated. Besides its tasks a technical platform is also characterized by a *resource model*, which describes the resources available to the tasks, and a scheduling algorithm, which determines the order of their execution. These aspects of task models are the topic of the next paragraphs.

Information Sources

An important aspect of a task model is the source the different information about the system comes from. We identify the time behavior specification, the application logic, the deployment platform, and the schedulability method itself as the main sources. The time constraints come, clearly, from the time specification, while the task activation conditions as well as the data and control flow models are extracted from the application logic. The deployment platform delivers the information about the supported scheduling strategies and, communication mechanisms, as well as context switch, and interrupt handling times. Among the outcomes of the schedulability analysis are the priorities of the individual tasks, as well as their start times. Certain task model parameters have mixed sources. For example, the WCET is usually obtained from the analysis of the application logic executed on the deployment platform.

Task Parameters and Timing Constraints

Task parameters can be assigned either to a task or to a job. An assignment of a parameter to a task makes it invariant for all jobs of this task. In contrast, the assignment to a job makes the parameter run-time dependent.

The most important task parameters are best- and worst-case execution time. The tight estimation of BCET and WCET for various hardware architectures is a non-trivial task and a wide area of research (cf. [WEE+08] for a comprehensive state-of-the-art and state-of-the-practice overview of tools and methods developed and applied in this area). Further parameters of tasks, like period or deadline, clearly correspond to the low-level time constraints, presented in Section 2.3.2. The refinement and decomposition methods from Chapter 3 allow to assign these constraints to individual tasks. Another parameter specific for a particular scheduling algorithm, like task start times, priorities, or communication schedules can be subsequently added by the system engineer after the analysis.

Activation Conditions

The classical literature distinguishes three types of task activation patterns, namely *periodic*, *aperiodic*, and *sporadic* patterns. Periodic tasks are activated repeatedly at regular time intervals. For instance, temperature monitoring in the vehicle cabin is a typical periodic task. Many real-time systems are required to respond to external events. Usually only the minimal inter-arrival time of these events is known. This is usually carried out by aperiodic and sporadic tasks of the system. These tasks are activated on certain input situations. The difference between both task types is that sporadic tasks have a hard deadline, while the aperiodic ones have a soft or no deadline. For example, an engine control task that activates the ignition is usually realized as a sporadic task whose activation time is dependent on piston position and crankshaft angular velocity. An example for aperiodic task is a task processing the input of human users.

During every activation a task consumes certain inputs and produces certain outputs. The inputs are made available by further tasks and/or environment and the outputs in turn are expected inputs of the environment/other tasks. Regardless of its activation type, a task should be activated, in a situation (or at a point in time), when all inputs, which it needs to produce outputs, are present[11].

The information about the expected inputs of the tasks is made explicit by the so called *task* or *precedence graphs*. The nodes in these graphs are tasks or jobs and the edges are causal dependencies (data and control flow) between them. Every node is attributed by an *activation condition*, also called *precedence constraint*. It describes either the input state of the task [HHK03], or the state of the predecessor tasks [Pop00], i.e., the tasks which according to the precedence graph deliver inputs must complete their execution before the current task can be activated. Clearly, the latter type of activation condition is the special case of the former one: The interrupt used by a task to suspend itself can be seen as an mandatory input for its successors. In the next section we will present a TA paradigm, which makes the above activation type distinction obsolete.

As discussed in the previous section, most scheduling approaches abstract from the concrete input-dependent task behavior (cf., e.g., approaches from [Jos01] or [Pop00]). They are dealing with the *presence* or *absence* of inputs, rather then with the actual message they contain. This allows to consider inputs and outputs of tasks as signals: a signal is either present or not. In

[11] The absence of some expected input at a certain point in time, is also a valuable information, which can be considered during the production of next outputs of the task.

order to capture activation conditions, we define the *abstract task interface* (*ATI*) as a set of input variable subsets $ATI \subseteq \wp(I)$. Every subset $ic \in ATI$ documents the activation condition of the task, i.e., which input signals have to be "present" or in terms of our STS framework which input variables have to be non-$\perp$, in order the task can produce meaningful results. Special cases are $ATI^{\mathsf{AND}} = \{I\}$, $ATI^{\mathsf{OR}} = \{\{v\} \mid v \in I\}$, and $ATI^{\top} = \wp(I)$. The first case requires all inputs to be present, for the second one suffices the presence of at least one signal, finally, the third case permits the arbitrary input behavior of a task.

Resource Model

Tasks have *active* and *passive* resources at their disposition. The most important active resources are processors, on which the tasks are executed, and transmission links which exchange data between processors. Processors and links build the topology of the target deployment platform, which can be represented by a graph, the so called *physical system architecture* [SZ06], with processors as nodes and transmission links as edges.

To the passive resources belong mutexes, semaphores, shared libraries, *etc.* Often these resources are *serially reusable*. This means that only one job can use each unit of resource at once. As a consequence, tasks often need to wait for the availability of a resource. This waiting time, also called blocking time, of course, flows into the overall response time of a task and complicates the timing analysis of real-time systems. Another, consequence is the infamous priority inversion problem: A lower priority task can prevent a higher prioritized one from running by locking certain resource, which they both use.

Scheduling Policies

The scheduling theory is a well-established branch of computer science and operations research. It describes and analyzes *scheduling policies*. Scheduling policies are sets of rules which, at any time, determine the order of task execution. In the literature, scheduling policies are classified according the following criteria:

preemptive vs. non-preemptive A non-preemptive task, once started, must be run to completion without any further interruption. A preemptive task, can be interrupted by another task with a higher priority.

off-line vs. on-line Off-line scheduling policies rely on scheduling tables which are computed before execution time. In contrast, on-line schedulers decide at execution time which task will be activated next.

static vs. dynamic In dynamic scheduling policies priorities of tasks change at execution time; priorities are fixed in the case of static scheduling policies.

There exist a number of scheduling policies, like rate monotonic scheduling (RMS), earliest deadline first (EDF), round-robin scheduling, *etc.* We refer, e.g., to [Liu00] or [SAÅ+04] for further particulars of scheduling theory and its policies.

4.3.2 AutoFocus Task Model (AFTM)

In order to instantiate the reference task model presented above information from different sources has to be brought together. One of the sources is the requirements specification which we treated in Chapter 2, another one concerning the activation and precedence constraints comes from the application model. This section presents a realization of this aspect of the reference task model from above. It is called AutoFocus Task Model (AFTM for short) and has a formal semantics defined within our STS framework.

As a task architecture paradigm for model-based development of reactive real-time systems, we suggest to model the clustered functionality of the system at the level of communicating tasks, the AutoFocus tasks. In [BGH+06] it is shown how the intended behavior of AutoFocus Task Model can be implemented using AutoFocus components and the composition operator of AutoFocus, i.e., how the semantics of AFTM can be emulated by the semantics of AutoFocus.

AutoFocus Tasks

We subdivide the run-time of a task in three phases. First, the task checks if all needed inputs are present, then if this is the case it makes its computations, and, finally, it writes its outputs. If the task cannot run in the given input state, it remains in the same state. No communication happens during the second phase, which usually needs more time than the other two. By ensuring that the reading (checking of inputs) and writing are atomic actions, which cannot be interrupted by the execution of other tasks, we can arbitrary interleave the task with other activities of the system without any impact on the results it will produce.

Another peculiarity of our task notion are explicit activation conditions associated with every individual task. In the previous section we described two different definitions of task activation conditions: A task is either activated after all/some of its predecessors complete their execution, or if a certain input situation emerges. Due to the run-time behavior of AutoFocus tasks described in the previous paragraph, the former type of activation condition becomes a special case of the latter one: Outputs of a task stay invisible for the outer world before it completes its execution. By this, the precedence constraints, expressed in terms of inputs dependencies can be translated for the schedulability methods, which work with task completion constraints, without a deformation of the original scheduling problem.

In AFTM we distinguish three different kinds of tasks: *AND*-, *OR*- and $\top$-tasks. In terms of our reference model from above every AutoFocus task realizes one of the following three abstract task interfaces: ATI^{AND}, ATI^{OR}, or $ATI^{\top}$. The function $ATI(\mathcal{S}^{aft})$ describes the ATI type of every task $\mathcal{S}^{aft}$ in an AFTM model. For any task $\mathcal{S}^{aft}$ and a given system state α the predicate $runnable(\mathcal{S}^{aft}, \alpha)$ evaluates to true if and only if the activation condition of this task is satisfied by α:

$$runnable(\mathcal{S}^{aft}, \alpha) \stackrel{\text{def}}{=} \{v \in I \mid \alpha.v \neq \bot\} \in ATI(\mathcal{S}^{aft}).$$

The input-enabledness of our STS automata has nice composition properties, but makes it complicated to express assumptions about the environment of the system. The solution is to model the environment explicitly by special $\top$-tasks. These tasks are always runnable and active. This

expresses the fact that the environment runs in parallel to the system. As discussed above, an *AND*-task may run only if *all* of its inputs are present, while for the activation of an *OR*-task the presence of a non-empty subset is sufficient.

The "flush" semantics of AutoFocus is a convenient means to model the inter-component communication in the models of application logic. It ensures that the *age* of an input message is exactly one, i.e., it was sent out in the last simulation step. However, in the technical systems the messages remain in the communication buffers until they are replaced by the next messages. We speak about a "keep" semantics in this case. Thus, if an AutoFocus task does not produce an output on a certain variable, it keeps its previous value.

Next we present the STS-based AutoFocus task semantics, which incorporates the ideas of the above informal introduction. Formally, an AutoFocus task (also called AFTM task) is an STS $\mathcal{S}^{aft} \stackrel{\text{def}}{=} (I, O, L, \mathcal{I}, \mathcal{R}, \delta\text{aft}, d^{\varepsilon})$.

In order to realize the keep semantics we need to store the past inputs. We realize this by a subset of dedicated local variables $L_I \subseteq L$ such that $L_I = \{v_l \mid v \in I\}$ holds, i.e., there is exactly one local variable in L_I for every input variable from I.

In the initial state no outputs are produced and no inputs were previously received, thus we demand $\mathcal{I}$ to satisfy the following constraint

$$\forall \alpha \in \Lambda(V) : \alpha \vdash \mathcal{I} \Rightarrow \forall v \in (L_I \cup O) : \alpha.v = \bot. \tag{4.8}$$

Similar to the definition of the AutoFocus semantics, in AFTM task STSs we model the individual transitions in a black-box manner. Thus, $\mathcal{R}$ denotes a set of "original" transitions, which encode the actual transformations carried out by the task. We assume that all transitions in $\mathcal{R}$ do not contain variables from L_I, i.e., $\forall r \in \mathcal{R} : \text{free}.r \cap L_I = \emptyset$ and that all transitions satisfy Property (4.2). δaft complements the behavior of $\mathcal{R}$ by removing underspecification in a schematic way, and installs the AFTM communication semantics, which is described next in more detail.

While a task is not runnable all non-$\bot$ inputs it receives are kept until the input state of the task satisfies its activation condition. A runnable task processes its inputs and flushes its input variables. This behavior reproduces the consumption of inputs by tasks run on a technical system: In every run an AFTM task reads its inputs once at the beginning. During its further execution, or while it is suspended new inputs may arrive. These inputs are gathered but not consumed by the task. They are considered by the task first at its next execution. On the other hand, the inputs already consumed by a task should not be kept in the system. Such inputs must disappear before the next run of this task. This naturally happens if a new input arrives – it always overwrites the old one. However, in the case that no input was produces the AFTM semantics ensures that the old inputs are overwritten with $\bot$. Exactly as in the AutoFocus semantics, the special message $\bot$ is interpreted as the absence of a message. In other words, AutoFocus tasks obtain the ability to notice the absence of certain inputs. Before we define the behavior of AutoFocus tasks, we fix the current input of a task in the state $\alpha \in \Lambda(V)$ by the valuation function $afTI(\alpha) \in \Lambda(I)$, which is defined as

$$\forall v \in I : afTI(\alpha).v \stackrel{\text{def}}{=} \begin{cases} \alpha.v & \text{if } \alpha.v \neq \bot, \\ \alpha.v_l, & \text{o.w.} \end{cases} \tag{4.9}$$

We characterize the set δ_{aft} by transitions which are built up of two segments, along the task execution phases described at the beginning of this section. Thus, for every transition $r \in \mathcal{R}$ we define an AutoFocus task transition $d \in \delta_{\mathsf{aft}}$ as $d \stackrel{\text{def}}{=} d^{rd} \circ d^{wt}$. The first segment of d reads the inputs and checks if the task is runnable. The second transition segment executes r, preserves the local variables and flushes outputs unconstrained by r, and, finally, flushes the local copies of input variables, since they are already consumed by the task. Formally for all $\alpha,\beta \in \Lambda(V)$,

$$\alpha,\beta' \vdash d^{rd} \stackrel{\text{def}}{\Leftrightarrow} runnable(\mathcal{S}^{aft}, afTI(\alpha)) \wedge \beta \stackrel{V\setminus I}{=} \alpha \wedge \beta \stackrel{I}{=} afTI(\alpha), \tag{4.10a}$$

$$\begin{aligned}\alpha,\beta' \vdash d^{wt} \stackrel{\text{def}}{\Leftrightarrow} \quad & (\alpha,\beta' \vdash r) \wedge \forall v' \in ((L' \setminus L'_I) \setminus \text{free}.r) : \beta.v = \alpha.v \\ & \wedge \forall v' \in (L'_I \cup (O' \setminus \text{free}.r)) : \beta.v = \bot.\end{aligned} \tag{4.10b}$$

Please note that neither d^{rd} nor d^{wt} are valid transitions in sense of Property (4.2), but their combination defined by (4.5) yields a valid transition.

If the task is not runnable, solely the idle-loop transition is enabled. The idle-loop transition does not alter valuations of the local variables, flushes the output variables, and stores the actual inputs in L_I. It is defined for all $\alpha,\beta \in \Lambda(V)$ as

$$\begin{aligned}\alpha,\beta' \vdash d^{\varepsilon} \stackrel{\text{def}}{\Leftrightarrow} \quad & \Big(\forall \gamma \in \Lambda(V) : \alpha,\gamma' \vdash \neg \bigvee_{d \in \delta_{\mathsf{aft}}} d\Big) \\ & \wedge \beta \stackrel{L\setminus L_I}{=} \alpha \wedge \forall v \in O : \beta.v = \bot \wedge \forall v \in I : \beta.v_l = afTI(\alpha).v.\end{aligned}$$

Please note that for the definition of d^{ε} it does not play any role, whether the output valuations are reset to $\bot$, or preserved – according to Equations (4.8) and (4.10b) their valuations will be $\bot$ anyway. By these definitions we ensure that $\neg runnable(\mathcal{S}^{aft}, afTI(\alpha)) \Rightarrow \text{Succ}(\alpha, \delta_{\mathsf{aft}}) = \emptyset$. We also observe that $\mathcal{S}^{aft}$ is well-defined and fulfills Properties (4.1)-(4.3).

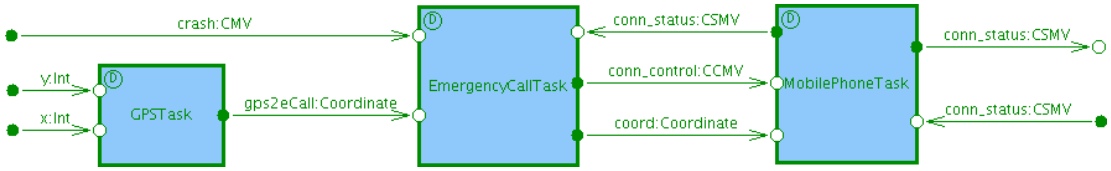
AFTM Composition

The scheduled AFTM system $\mathcal{S}\mathsf{ch}^{aftm}$ executes all AutoFocus tasks in parallel if and only if they are pairwise compatible, i.e.,

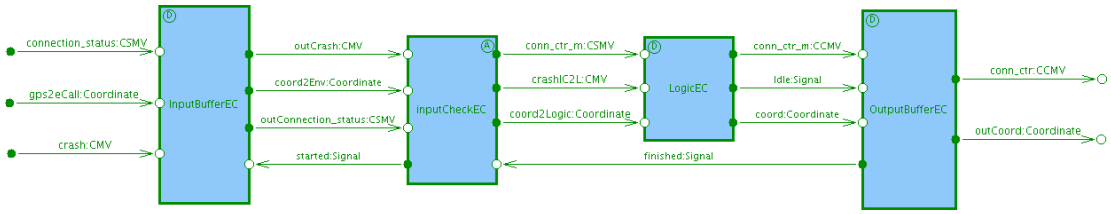
$$\mathcal{S}\mathsf{ch}^{aftm} \stackrel{\text{def}}{=} \mathcal{S}\mathsf{ch}^{par}.$$

To sum up, AFTM starts its tasks with output variables and local input copies flushed. In every step only runnable tasks make a regular transition step. The remaining tasks fire their idle loops and gather current inputs until they become runnable. Runnable tasks flush consumed inputs. If a task produces no output on a certain variable, it indicates this fact to its communication partner by resetting its value to $\bot$. From this description we can conclude that there must either exist unbounded input variables, or at least one $\top$-task in the AFTM model in order it can exhibit a behavior, which contains non-idle transition steps. This can be interpreted as: the AFTM models are started by environment.

eCall Case Study (*con'ed*). The AFTM semantics defined above can be emulated by the AutoFocus semantics from Section 4.2.2. The structure of an AFTM task in AutoFocus is shown



(a) *The Task Architecture.*



(b) *The Internal Structure of* EmergencyCallTask.

Figure 4.5: *Auto*Focus *Task Model of the eCall Study.*

in Figure 4.5b for the emergency-call task. The application logic is contained in the component LogicEC. It is surrounded by buffers, which gather the inputs (InputBufferEC) and outputs (OutputBufferEC) of the task during its execution or while it is idle. The component inputCheckEC realizes the *runnable*-predicate for this task – it decides to activate the task logic depending on the state in the input buffer. The task architecture of the eCall system is shown in Figure 4.5a. It consists of three tasks. The distribution of the four components from the AutoFocus component architecture (cf. Figure 4.2) to the AFTM tasks is described in the next section. For a more detailed description of the realization of the task architecture in AutoFocus we refer the reader to [BGH+06].

4.4 From AutoFocus to AFTM

In this section we discuss the transition from the pure application logic modeled in AutoFocus to an AFTM system. First, let us recapitulate and compare the properties of both modeling paradigms. The communication between AutoFocus tasks is a continuous flow of messages or, in terms of our STS formalism, – a continuous flow of input/output valuations. The messages must be consumed immediately after their arrival; otherwise, they are lost (flushed in the next communication round). In AFTM the messages are gathered by the receiving task. It consumes its inputs only if all inputs necessary for the computation are present. Messages can get lost only if they are overwritten by other non-$\perp$ messages.

We aim at clustering n AutoFocus components by one AFTM task. This degree of freedom allows the developer not to align the design of application logic respective of possible clustering or deployment strategies. Methodologically it is an important issue, since the logical time has no direct correspondence to the physical time: Consider, for example, two AutoFocus components

$\mathbb{S}_A^{af}$ and $\mathbb{S}_B^{af}$. $\mathbb{S}_A^{af}$ is realized in a way, that it needs two transition steps in order to react to a certain request. In contrast, $\mathbb{S}_B^{af}$ makes on every request only one step before the answer is produced. If we realize both components as separate tasks and deploy them, their response times will not necessarily preserve this relation. We have discussed and explained this phenomenon already, e.g., cf. Section 2.3.2. One possible solution for this problem is a constructive method, which always enforces the synchronization even between independent tasks after every logical system step on the deployment platform. This surely leads to over-constrained and inefficient implementations. Another one is a restrictive method, which demands to portion the functionality in a way, which produces components with a uniform response delay (e.g., delay one) only. We will discuss the disadvantages of such restriction in the next paragraph. In the present thesis we pursue an alternative approach, which allows us to capsule functionality arbitrarily and to reason about the causal dependencies of created tasks, without considering their logical execution times.

A commonality of AutoFocus and AFTM is their time-synchronous lockstep semantics. This fact suggests the establishing of a one-to-one correspondence between system steps in AutoFocus and AFTM. However, this would substantially restrict the design of the application logic as well as the number of possible clustering solutions. Provided, for example, the functionality of one task is realized by a network of AutoFocus components. Then, a response this task must produce on a certain request is not necessarily delayed by exactly one AutoFocus step. The clustering of the component networks results in tasks with internal communication links which, in accordance to our strict causal communication semantics, generate delays between a request and a corresponding response. In other words, given an AFTM task, which is realized by a network of AutoFocus components, then an external input of this task and the corresponding external output may lie more than one system step away from each other. This delay is also not necessarily bounded by the length of the data paths in the network – inputs may induce an intertwined data exchange between the internal AutoFocus components before the corresponding output is produced.

To sum up, the above considerations motivated us to establish an n-to-one relationship between AutoFocus components and tasks, as well as between AutoFocus and AFTM transition steps. A consequence of this decision is that a synchronized step of two AutoFocus tasks may correspond to the transition sequences of different length on the level of their respective "realizations" by AutoFocus component networks. In AFTM composition these sequences are aligned. In other words, the AFTM model runs in lockstep: every of its steps lasts as long as needed by the longest sequence. Meanwhile, the tasks which are ready with their execution stop and wait. This solution synchronizes the inter-task communication and allows to reason about precedence constraints of tasks without considering their logical execution times.

In Section 4.4.1 we give a precise mathematical problem statement of the property-preserving transition from AutoFocus to AFTM. In Section 4.4.2 we present a procedure that performs this transition and prove its correctness.

eCall Case Study (*con'ed*). The task architecture of the eCall system shown in Figure 4.5a consists of three tasks, which capture the application logic of the GPS module, of the emergency call itself, and of the mobile phone, respectively. Table 4.1 shows how the four components

Component	Task
GPS	GPSTask
CoordSaver eCallLogic	EmergencyCallTask
MobilePhone	MobilePhoneTask

Table 4.1: *Component to Task Mapping for the eCall System*

which build the application logic of the eCall system (shown in Figure 4.2) are distributed over its tasks.

4.4.1 Problem Statement

Due to the input-enabledness of STS automata every valuation of input variables can be accepted by any AutoFocus component or AFTM task. However, the clustering process should ensure that the desired outputs are produced as a response on specific inputs. These input/output pairs are documented in the *abstract component interface*, $ACI(\mathcal{S}^{af}) \subseteq \wp(I) \times \wp(O)$, of the AutoFocus component $\mathcal{S}^{af}$. A task *realizes*, or *obeys* the abstract component interface $ACI(\mathcal{S}^{af})$ only if holds: The task makes a non-idle transition step only if there exists a pair $(ic, oc) \in ACI$ such that

(1) the valuations of input variables from $ic \subseteq I$ are non-$\perp$ in the current state,
(2) produced output contains non-$\perp$ valuations of output variables from $oc \subseteq O$, and
(3) produced valuations of output variables from $O \setminus oc$ are $\perp$.

Obviously, the abstract task interface introduced in Section 4.3.1 is a projection of the abstract component interface on its first element.

We formulate the clustering problem as follows: Given an AutoFocus STS $\mathcal{S}^{af} = (I, O, L^{af}, \mathcal{I}^{af}, \mathcal{R}^{af}, \delta_{\mathsf{af}}, d^{\varepsilon}_{af})$ and an abstract component interface $ACI(\mathcal{S}^{af})$, we want to construct an AFTM task STS $\mathcal{S}^{aft} = (I, O, L^{aft}, \mathcal{I}^{aft}, \mathcal{R}^{aft}, \delta_{\mathsf{aft}}, d^{\varepsilon}_{aft})$, with $V^{aft} \supseteq V^{af}$, which obeys $ACI(\mathcal{S}^{af})$ and stands in a weak simulation relation (cf. Definition A.6) to $\mathcal{S}^{af}$. As discussed in the previous section, the simulation relation is allowed to relate finite run prefixes of $\mathcal{S}^{af}$ to the individual transitions of $\mathcal{S}^{aft}$. In the next paragraphs we will formally define this simulation relation.

***Remark* 4.5.** The one-to-one mapping between AutoFocus components and AFTM tasks is not a deficit of the presented approach and does not contradict to the n-to-one relation advocated in the previous section. The AutoFocus composition, which is equivalent to the parallel composition of STS and thus fulfills Proposition 4.1, allows us to cluster several AutoFocus components on one AFTM task and consider the composite AutoFocus STS in respect of simulation by the corresponding AFTM task. ◦

First, let us define the concatenation of variable valuations. For a given valuation pair $\alpha,\beta \in \Lambda(V)$, we define a new valuation $\alpha \frown \beta \in \Lambda(V)$ as the concatenation of valuations from α and β and the removal of $\perp$-values for every variable from V. Formally,

$$\forall v \in V : (\alpha \frown \beta).v \stackrel{\text{def}}{=} \begin{cases} \perp & \text{if } \alpha.v = \beta.v = \perp, \\ \alpha.v & \text{if } \alpha.v \neq \perp \text{ and } \beta.v = \perp, \\ \beta.v & \text{if } \alpha.v = \perp \text{ and } \beta.v \neq \perp, \\ \alpha.v \frown \beta.v, & \text{o.w.} \end{cases}$$

Next, we define the equivalence class of runs, which are equal up to "stuttering", i.e., idle loop steps. Since idle loops in AutoFocus and AFTM do not produce outputs and do not change the local state, they perfectly reproduce the notion of stuttering as introduced in [Lam83] for the specification approach TLA: the set of all possible behaviors after a stutter step (idle loop) remains the same. In order to define the above equivalence class formally, we mark the sequences of idle loops in runs by a special strict monotone indexing function $\eta\colon \mathbb{N} \mapsto \mathbb{N}_\infty$. $\eta(0) \stackrel{\text{def}}{=} 0$, and for any run ρ and index $i \in \mathbb{N}$, states from the interval $[\eta(i), \eta(i+1)-1)$ enable the idle-loop transition only; formally,

$$\forall j \in [\eta(i), \eta(i+1)-1) : \rho.j, (\rho.j+1)' \vdash d^\varepsilon.$$

If a non-idle transition is fired in state $\eta(i)$, we define $\eta(i+1) \stackrel{\text{def}}{=} \eta(i)+1$ and the above interval becomes empty. If a finite run prefix of length $\eta(i)$ is continued by an infinite sequence of idle loops, η maps all indexes starting with $i+1$ to ∞. The representative element of a stuttering class is a run $\overline{\rho}$ without idle-loop transitions, i.e., $\forall i \in \mathbb{N} : \neg(\overline{\rho}.i, (\overline{\rho}.i+1)' \vdash d^\varepsilon)$ or, equally, $\{\eta(i) \mid i \in \mathbb{N}\} = \mathbb{N}$ for $\overline{\rho}$. A run ρ belongs to the class of $\overline{\rho}$, denoted by $\rho \in IF(\overline{\rho}, d^\varepsilon)$ if and only if

$$\rho \in IF(\overline{\rho}, d^\varepsilon) \stackrel{\text{def}}{\Leftrightarrow} \exists \eta \in (\mathbb{N} \mapsto \mathbb{N}_\infty) : \quad \eta(0) = 0 \\ \wedge \forall i \in \mathbb{N} : \eta(i+1) < \infty \Rightarrow \overline{\rho}.i \stackrel{L \cup O}{=} \rho.\eta(i) \wedge \overline{\rho}.i \stackrel{I}{=} \rho.\eta(i+1)-1.$$

We denote the *stutter reduction* of a runs set $\mathfrak{R}$ by $SF(\mathfrak{R}, d^\varepsilon)$. It consists of stuttering-free runs, to whose equivalence classes the runs from $\mathfrak{R}$ belong. Formally,

$$SF(\mathfrak{R}, d^\varepsilon) \stackrel{\text{def}}{=} \{\overline{\rho} \mid \mathfrak{R} \cap IF(\overline{\rho}, d^\varepsilon) \neq \emptyset\}.$$

For the complete set of AFTM task runs it holds $SF(\langle\langle \mathcal{S}^{aft} \rangle\rangle, d^\varepsilon_{aft}) \subseteq \langle\langle \mathcal{S}^{aft} \rangle\rangle$. In fact, we can easily prove that every run in $\langle\langle \mathcal{S}^{aft} \rangle\rangle$ belongs to some equivalence class, whose representative element is also in $\langle\langle \mathcal{S}^{aft} \rangle\rangle$ and, vice versa, – for any stutter-free run ρ from $\langle\langle \mathcal{S}^{aft} \rangle\rangle$ it holds $IF(\rho, d^\varepsilon) \subseteq \langle\langle \mathcal{S}^{aft} \rangle\rangle$.

In order to define the simulation relation between components and tasks formally we define a special function, which relates every task step to the corresponding sequence of component steps. An AFTM task run $\rho \in SF(\langle\langle \mathcal{S}^{aft} \rangle\rangle, d^\varepsilon_{aft})$ is related to an AutoFocus component run $\overline{\rho} \in \langle\langle \mathcal{S}^{af} \rangle\rangle$, denoted by $\rho \subsetneq \overline{\rho}$ if and only if there exists a strict monotonic indexing function $\mu\colon \mathbb{N} \mapsto \mathbb{N}$ with $\mu(0) = 0$ and for all $i \in \mathbb{N}_+$:

$$\begin{aligned} afTI(\rho.i) &\stackrel{I}{=} \overline{\rho}.\mu(i) \frown \ldots \frown \overline{\rho}.(\mu(i+1)-1), \\ \rho.i &\stackrel{I^{af}}{=} \overline{\rho}.\mu(i), \\ \rho.i &\stackrel{O}{=} \overline{\rho}.(\mu(i-1)+1) \frown \ldots \frown \overline{\rho}.\mu(i). \end{aligned} \tag{4.11}$$

Mapping μ permits us to combine several AutoFocus transitions to one AFTM transition. Two valuation states are in the simulation relation according to μ if

(1) their common local states are equal,
(2) the concatenation of outputs produced so far by the AutoFocus component since the previous state in the simulation are equal to the outputs of the AFTM task, and
(3) the concatenation of future inputs of the AutoFocus component up to the next simulation state are equal to the input valuations from the task state.

For example, the mapping with the property $\forall k \in \mathbb{N} : \mu(k) = k$, establishes a one-to-one relationship between the members of both transition sets. We extend the definition of $\subseteq$ for the whole $\mathcal{S}^{af}$ and $\mathcal{S}^{aft}$ as follows

$$\mathcal{S}^{aft} \subseteq \mathcal{S}^{af} \stackrel{\text{def}}{=} \forall \rho \in SF(\langle\langle \mathcal{S}^{aft} \rangle\rangle, d^{\varepsilon}_{aft}) : \exists \overline{\rho} \in \langle\langle \mathcal{S}^{af} \rangle\rangle, \mu \in (\mathbb{N} \mapsto \mathbb{N}) : \rho \subseteq \overline{\rho}.$$

Then, for $\mathcal{S}^{af}$ and $ACI(\mathcal{S}^{af})$ the synthesis problem is to produce $\mathcal{S}^{aft}$ such that the following two conditions are satisfied:

$$\mathcal{S}^{aft} \subseteq \mathcal{S}^{af} \qquad \text{and} \qquad \forall d \in \delta\text{aft} : \forall \alpha, \beta \in \Lambda(V^{aft}) : \alpha, \beta' \vdash d \Leftrightarrow iconf(\mathcal{S}^{af}, \alpha, \beta), \tag{4.12}$$

where the predicate *iconf* checks if a pair of states is conform to the interface definition:

$$\forall \alpha, \beta \in \Lambda(V^{aft}) : iconf(\mathcal{S}^{af}, \alpha, \beta) \stackrel{\text{def}}{\Leftrightarrow} (\{v \in I \mid \alpha.v \neq \bot\}, \{v \in O \mid \beta.v \neq \bot\}) \in ACI(\mathcal{S}^{af}).$$

The simulation relation defined above relates every idle-loop-free run of the AFTM task to a condensed run of the AutoFocus component. The "condensation" of AutoFocus runs, i.e., the removal of $\bot$ values allows to group in principle unbounded sequences of idle-loops produced by the AutoFocus component and to relate this equivalence class of partial AutoFocus behaviors to one AFTM transition step. The idle loops of AFTM task are also excluded from the simulation relation. Thereby, our working hypothesis is that no useful work is done by components or tasks during idle loops and the simulation between respective idle-loop-free representatives captures the most important properties which are worth to be preserved.

4.4.2 AutoFocus Task Synthesis

In order to synthesize $\mathcal{S}^{aft}$ we are looking for the run prefixes of $\mathcal{S}^{af}$, in which a specified input is processed and the corresponding output is produced. Thereby, the individual variable valuations, which belong to the input and output pair from the task interface, can be arbitrary serialized, i.e., spread over several AutoFocus simulation steps, and no other outputs may be produced during that run prefix. In order to obtain a finite-state STS, we bound the length of prefixes in the above construction. This constraint is realistic in the settings of embedded systems since they have to be realized on platforms short for available memory resources. The rest of the current section contains the formalization of this idea as well as proofs of its correctness.

First, we define the n-bounded transitive closure ($n \in \mathbb{N}_+$) of a transition set δ:

$$\delta^n \stackrel{\text{def}}{=} \bigcup_{k \leq n} \{d_1 \frown \ldots \frown d_k \mid d_1, \ldots, d_k \in \delta\},$$

where $n \in \mathbb{N}$ and for all $d_1, d_2 \in \delta$, $d_1 \frown d_2$ is a new transition defined as

$$d_1 \frown d_2 \stackrel{\text{def}}{=} \{(\alpha, \beta') \mid \quad \exists \alpha_1, \alpha_2, \alpha_3 \in \Lambda(V) : \alpha_1, \alpha_2' \vdash d_1 \wedge \alpha_2, \alpha_3' \vdash d_2 \\ \wedge \alpha \stackrel{L^{af}}{=} \alpha_1 \wedge \beta \stackrel{L^{af}}{=} \alpha_3 \wedge \alpha \stackrel{I}{=} \alpha_1 \frown \alpha_2 \wedge \beta \stackrel{O}{=} \alpha_2 \frown \alpha_3\}.$$

The transition $d_1 \frown d_2$ starts from the current state described in d_1 and terminates in the successor state reachable by d_2. All outputs made by d_1 and d_2 are concatenated outputs of $d_1 \frown d_2$ (modulo $\perp$s). Please note that the concatenation on transition defined that way is associative. Provided the variable types are finite then δ^n is finite either.

We define the AFTM task STS $\mathcal{S}^{aft} = (I, O, L^{aft}, \mathcal{I}^{aft}, \mathcal{R}^{aft}, \delta_{\mathsf{aft}}, d^{\varepsilon}_{aft})$ as a task built as described in Section 4.3.2 over the set of transition predicates $\mathcal{R}^{aft}$ and the abstract task interface $ATI(\mathcal{S}^{aft})$. $\mathcal{R}^{aft}$ and $ATI(\mathcal{S}^{aft})$ are synthesized as follows

$$\mathcal{R}^{aft} \stackrel{\text{def}}{=} \{d \in \delta_{\mathsf{af}^n} \mid \exists n \in \mathbb{N} : \forall \alpha, \beta \in \Lambda(V^{af}) : \alpha, \beta' \vdash d \Leftrightarrow iconf(\mathcal{S}^{af}, \alpha, \beta)\},$$

$$ATI(\mathcal{S}^{aft}) \stackrel{\text{def}}{=} \{ic \mid (ic, oc) \in ACI(\mathcal{S}^{af})\}.$$

Please note that we can define $\mathcal{R}^{aft}$ over δ_{af} because it complements its original AutoFocus transitions by $\perp$s for unconstrained outputs and preserves the valuations of unconstrained local variables. Empty messages are eliminated during the construction of the transitive closure and the unconstrained local variables also do not change their values according to the AFTM semantics.

We say that the AutoFocus component $\mathcal{S}^{af}$ is *n-compatible* to the abstract component interface $ACI(\mathcal{S}^{af})$ if and only if the set of native transitions $\mathcal{R}^{aft}$ synthesized using the scheme from above is not empty.

By the next proposition we show that Simulation Relation (4.12) holds for every AutoFocus component and a synthesized AFTM task.

Proposition 4.6 (Property preservation during AFTM task synthesis)**.** *For a given AutoFocus component $\mathcal{S}^{af} = (I, O, L^{af}, \mathcal{I}^{af}, \mathcal{R}^{af}, \delta_{\mathsf{af}}, d^{\varepsilon}_{af})$, an interface $ACI(\mathcal{S}^{af})$ and an AFTM task $\mathcal{S}^{aft} = (I, O, L^{aft}, \mathcal{I}^{aft}, \mathcal{R}^{aft}, \delta_{\mathsf{aft}}, d^{\varepsilon}_{aft})$ synthesized using the procedure from above holds Simulation Relation (4.12).*

Proof. In order to show the simulation two properties have to be proved.

$\mathcal{S}^{aft} \subsetneq \mathcal{S}^{af}$: Let ρ be an arbitrary run from $SF(\langle\langle \mathcal{S}^{aft} \rangle\rangle, d^{\varepsilon}_{aft})$, then we must construct a run $\overline{\rho} \in \langle\langle \mathcal{S}^{af} \rangle\rangle$ such that $\rho \subsetneq \overline{\rho}$. We do this by induction over the prefix length $i \in \mathbb{N}$ of ρ. Since the initial states of $\mathcal{S}^{af}$ and $\mathcal{S}^{aft}$ are the same for V^{af}, we can define $\overline{\rho}.0 = \overline{\rho}.\mu(0) \stackrel{V^{af}}{=} \rho.0$.

For the induction step suppose for some $i \in \mathbb{N}$ there is $\mu(i) \geq i$ such that $\rho.i$ and $\overline{\rho}.\mu(i)$ are related according to (4.11). Then, since idle loops are never fired in ρ, it holds

$$\exists d \in \delta_{\mathsf{aft}} : \rho.i, (\rho.i + 1)' \vdash d.$$

Every regular transition of $\mathcal{S}^{aft}$ is based on some original transition predicate r from $\mathcal{R}^{aft}$. r in turn is built by a concatenated sequence $d_1 \frown \ldots \frown d_k$ with $1 \leq k \leq n$ of $\mathcal{S}^{af}$'s transitions.

Let k be the length of this sequence, then we set $\mu(i+1) \stackrel{\text{def}}{=} \mu(i)+k+1$ and define the valuation states of $\overline{\rho}$ in the interval $(\mu(i),\mu(i+1)]$ as

$$\forall m \in [1,k] : \overline{\rho}.\mu(i)+m, (\overline{\rho}.\mu(i)+m+1)' \vdash d_m.$$

By this, we obtain the fulfillment of the second and the third lines in Simulation Relation (4.12), i.e., local variables and outputs of $\rho.i+1$ are equal to the current local variables and past outputs in $\overline{\rho}.\mu(i+1)$, respectively. The relation between inputs of $\rho.i+1$ and future inputs between $\overline{\rho}.\mu(i+1)$ and $\overline{\rho}.\mu(i+2)$ follows from Property (4.2), i.e., AutoFocus components cannot constraint their future inputs. By this, the first line of the simulation relation can be proved for arbitrary $\mu(i+2) > \mu(i+1)$.

$\forall d \in \delta_{\mathsf{aft}} : \forall \alpha,\beta \in \Lambda(V) : \alpha,\beta' \vdash d \Leftrightarrow \mathit{iconf}(\mathcal{S}^{af},\alpha,\beta)$: The task interface of $\mathcal{S}^{aft}$ was defined as the projection of $\mathit{ACI}(\mathcal{S}^{af})$ on input variables. Thus, this property follows from the definition of AFTM task behavior in (4.10). □

The crucial point in the above proof is the fact that we simulate any sequence of idle-loop transitions in AFTM by the same state in AutoFocus. The reason for this is that an input not mentioned in the abstract component interface can be potentially (partially) accepted by AutoFocus. This would then lead to a new local valuation state in AutoFocus.

We still need to show that the simulation relation we defined for individual AutoFocus components and tasks is monotonic respective AutoFocus and AFTM composition. In the following proposition we show this for the case of two components and two tasks. The general case for n components/tasks follows then from the associativity of the parallel composition used for both, AutoFocus and AFTM composition operators.

Proposition 4.7 (Correctness of AutoFocus task generation)**.** *Given two compatible AutoFocus components $\mathcal{S}_0^{af}$, $\mathcal{S}_1^{af}$ and two compatible AFTM tasks $\mathcal{S}_0^{aft}$, $\mathcal{S}_1^{aft}$ such that $\mathcal{S}_i^{aft} \subsetneq \mathcal{S}_i^{af}$ for $i \in \{0,1\}$, then holds $\mathcal{S}\mathsf{ch}^{aftm} \subsetneq \mathcal{S}\mathsf{ch}^{af}$, where $\mathcal{S}\mathsf{ch}^{af}$ is the AutoFocus composition of $\mathcal{S}_0^{af}$ and $\mathcal{S}_1^{af}$ and $\mathcal{S}\mathsf{ch}^{aftm}$ the AFTM composition of $\mathcal{S}_0^{aft}$ and $\mathcal{S}_1^{aft}$.*

Proof. Both modeling paradigms compose their components/tasks using the parallel composition operator. Thus, we can use Proposition 4.2 and conclude in both cases that

$$\langle\langle \mathcal{S}\mathsf{ch}^{aftm} \rangle\rangle \overset{V_i^{aft}}{\subseteq} \langle\langle \mathcal{S}_i^{aft} \rangle\rangle \qquad \text{and} \qquad \langle\langle \mathcal{S}\mathsf{ch}^{af} \rangle\rangle \overset{V_i^{af}}{\subseteq} \langle\langle \mathcal{S}_i^{af} \rangle\rangle,$$

where V_i^{af} and V_i^{aft} are the variable sets of $\mathcal{S}_i^{af}$ and $\mathcal{S}_i^{aft}$, respectively, and $i \in \{0,1\}$. Then, form the above subset relation follows

$$SF(\langle\langle \mathcal{S}\mathsf{ch}^{aftm} \rangle\rangle, d^{\varepsilon}_{aftm}) \overset{V_i^{aft}}{\subseteq} SF(\langle\langle \mathcal{S}_i^{aft} \rangle\rangle, d^{\varepsilon}_{aft_i}),$$

where $d^{\varepsilon}_{aft_i}$, d^{ε}_{aftm} are the idle-loop transitions of $\mathcal{S}_i^{aft}$, $\mathcal{S}\mathsf{ch}^{aftm}$, respectively, with $i \in \{0,1\}$. And thus, $\mathcal{S}\mathsf{ch}^{aftm} \subsetneq \mathcal{S}_i^{af}$[12] for all $i \in \{0,1\}$.

[12] The $\subsetneq$-relation is well-defined for every STS pair $\mathcal{S}_0$, $\mathcal{S}_1$ only if $V_0 \supseteq V_1$. This is surely the case for $\mathcal{S}\mathsf{ch}^{aftm}$ and $\mathcal{S}_i^{af}$.

Suppose, there exists some $\rho \in SF(\langle\langle \mathcal{S}ch^{aftm} \rangle\rangle, d^{\varepsilon}_{aftm})$ such that there is no $\overline{\rho} \in \langle\langle \mathcal{S}ch^{af} \rangle\rangle$ with $\rho \subsetneq \overline{\rho}$. There must exist some $\overline{\rho}_i \in \langle\langle \mathcal{S}^{af}_i \rangle\rangle$ which is *not* in $\langle\langle \mathcal{S}ch^{af} \rangle\rangle$[13] with $\rho \subsetneq \overline{\rho}_i$. Then, in $\overline{\rho}_i$ there exists a pair of adjacent valuation states for which no unifiable transition of $\mathcal{S}^{af}_{|i-1|}$ exists. This can only happen if the shared output variable valuations of $\mathcal{S}^{af}_i$ in the first state, or the shared input variable valuations of $\mathcal{S}^{af}_i$ in the second state, cannot be accepted, or produced by $\mathcal{S}^{af}_{|i-1|}$, respectively. These shared variables coincide in both compositions (of AutoFocus and of AFTM) and become local in both cases. Thus, according to the definition of the $\subsetneq$-relation they must be equal in every AFTM state. Since $\mathcal{S}^{aft}_i \subsetneq \mathcal{S}^{af}_i$, we know that both transition sets produce the same sequences of local variables, i.e., $\langle\langle \mathcal{S}^{af}_i \rangle\rangle \overset{L^{af}}{=} \langle\langle \mathcal{S}^{aft}_i \rangle\rangle$. Thus, the valuations unifiable in the AFTM composition are also unifiable in the AutoFocus composition and we obtain a contradiction. □

The drawback of the presented method is that the n-compatibility of a pair of AutoFocus components to their respective abstract component interfaces does not guarantee that the AFTM composition of synthesized AutoFocus tasks exhibits behaviors different from pure idle-loop runs. Thus, we propose to apply the presented task synthesis algorithm iteratively until the common prefix length is found or the size of the model (e.g., in terms of data volume, which has to be exchanged between tasks) becomes impractical for the deployment on the target platform. This procedure exploits the fact that the n-compatibility implies also the $n+1$-compatibility.

4.5 Related Work

In the following we compare contributions presented in this chapter with related approaches from different fields of research and practice: We compare our STS framework with its direct precursors; we survey different clustering and deployment approaches provided by commercial CASE tools and proposed in the computer science research community; and we compare AFTM with existing task architecture models.

State Transition Systems

Different versions of the STS formalism are presented in [PP05] and [BP99]. The authors of [PP05] are interested in refactoring of automata behaviors. Thus, they deal with isolated STSs only and do not define the composition or refinement for this formalism. The main topic of [BP99] is the transition from operational automata models to denotational black-box specifications. There, the STS variables range over streams of values, i.e., in every transition step they extend the own valuation history. The composition operator presented in [BP99] corresponds to the interleaved composition $\mathcal{S}ch^{int}$ from Section 4.1.6 with additional fairness constraints.

[13] In other words, there is no run in $\langle\langle \mathcal{S}ch^{af} \rangle\rangle$ which coincides with $\overline{\rho}$ on V^{af}_i.

Task Architecture Models

The system behavior realized by AFTM is inspired by Henzinger's Giotto approach [HHK03]. The Giotto tasks, realized in C, are also activated only if all needed inputs are available. Their outputs are issued after the time of their worst case execution is elapsed. In order to provide such behavior, Giotto installs a low-level system driver, called *E-machine*, which takes over the role of input and output check during the run-time. However, in contrast to the presented approach, the data and control flow, which serves as an input for schedule synthesis, are extracted in a rather *ad hoc* manner, e.g., it cannot be proven, that they correspond to the actual behavior of the C-code tasks. For the AutoFocus Task Model these dependencies are made explicit and checked for their correctness. Furthermore, no additional middleware like the E-machine is needed in the presented work. This fact leaves more liberties for the subsequent deployment procedure and extends the set of possible platforms which are suited for the deployment of AutoFocus tasks.

Clustering and Deployment Approaches

As noted in [BBC+03], in CASE tools typically used for model-based software development in the automotive domain, like MATLAB/Simulink, Rose RT, AutoFocus, there is no explicit deployment concept. In other tools, like ASCET-SD or Cierto VCC, there is an ability to build a deployment model for one node only. However, these tools allow the modeling of the systems only on a very low level of abstraction. The application of such tools in the early design phases would lead to unnecessary over-specification.

AUTOSAR [HSF+04] (AUTomotive Open System Architecture) pursuits the goal of reducing the growing complexity of the electrical/electronic systems in vehicles. For this purpose it tries to reduce the platform dependencies of the automotive applications by defining a uniform syntactic interface for the inter-process communication. In terms of AUTOSAR an atomic software component is a system, which can be deployed on exactly one ECU (electronic control unit). Software components consist of primitive functions, called runnables, which can be arbitrary clustered to tasks on that ECU. No adaptation support is provided by the AUTOSAR approach for the clustering process. The software components of AUTOSAR communicate through the so-called Virtual Functional Bus (VFB) – a uniform syntactical communication interface –, which connects different platform-specific Run Time Environments (RTEs). An RTE wraps the software components and realizes the communication primitives of the VFB for a specific platform. At the moment, AUTOSAR is clearly implementation oriented. There is no development process based on it. In this sense, a promising approach would be to integrate our clustering and deployment procedures with AUTOSAR's architectural and run-time models.

The MetaH approach [Hon98] stems from the domain of avionics and flight control. It provides an architectural description language (ADL), which allows to describe software and hardware components of the system in a uniform manner. Software components are realized in Ada programming language. Moreover, the MetaH tool set provides the possibility of user-guided mapping of software to hardware components, i.e., an interactive tool-supported deployment procedure. A further part of MetaH is the MetaH Execution Environment, which is a combination

of a Real-Time Operating System (RTOS) and a run-time environment for Ada programs. The run-time environment executes Ada byte code and provides communication primitives for inter-process communication. Analogous to AUTOSAR, this pragmatic approach lacks of property-preservation evidence during deployment and the flexibility in the derivation of clustering solutions.

A number of approaches aim at preservation of system behavior during the transition between different computational models (cf. [Gir05] for a survey). The most prominent representative is the GALS approach of [BCG99]. GALS stands for Globally Asynchronous, Locally Synchronous and occupies with the problem of distributing time-synchronous models over an asynchronous network in a way which preserves their behavior. For this purpose [BCG99] formulates two properties which components of the system have to satisfy: endochrony and isochrony. If this is the case, the GALS approach guarantees property preservation. Compared with our task synthesis procedure, GALS pursuits a slightly different goal: Simulation Relation (4.12) produces a Locally Asynchronous, Globally Synchronous (LAGS) relationship between AutoFocus and AFTM models. In other words, every AutoFocus run sequence which is mapped to one AFTM step is desynchronized in AFTM, but the overall AFTM system runs in synchronous lockstep. It is a promising direction for future work to define sufficient and necessary conditions for LAGS which resemble endochrony and isochrony in GALS.

4.6 Summary

The contribution of this chapter can be subdivided into four main topics. Firstly, a generic automata-based framework is presented. It allows us to logically characterize the behavior of reactive systems and facilitates their component-based development, validation, and verification. Using STS the individual components can be specified and composed to overall systems by the formally defined composition operator. The operational semantics permits the validation of STS models by simulation. Finally, the logical characterization allows the verification of the systems specified in STS by techniques, like model checking, SAT-, and constraint solving, *etc.* The STS framework is embedded into the domain of reactive real-time systems from Chapter 2. Thus, in particular, the functionality modeled in STS can be checked respecting fulfillment of time constraints. Finally, the notion of STS refinement is introduced, which allows us to show the property preservation throughout the development process.

Secondly, the STS framework served as a formal foundation of CASE tool-supported model-based development of reactive systems. In particular, STS was instantiated by a concrete semantic specification scheme, which describes the behavior of the CASE tool AutoFocus. This scheme delivered a formal foundation for the development of application logic with AutoFocus and the starting point for the subsequent transformations of application-logic models towards deployed systems.

As an intermediate step towards a deployed system, serves the task architecture model. Thus, thirdly, the properties of TA were presented as a part of a reference task model. After that the novel TA called AFTM is introduced and characterized within the STS framework. AFTM

can be integrated into the aforementioned reference task model. It serves as a basis for both schedulability analysis as well as for code- and wrapper-generation for the target deployment platform.

Finally, fourthly, a transition procedure from an AutoFocus application logic model to an AFTM model was presented, and the property preservation of this procedure was proved. For a given AutoFocus model, clustering function of its components to tasks, and description of task activation conditions (abstract component interfaces) the presented approach generates a task architecture which respects the activation conditions and establishes a simulation relation between a subset of the original application-logic behavior and the task architecture.

To sum up, we have covered the transition from the logical architecture to the task architecture and prepared the basis for the subsequent step, according to our development process from Figure 1.1 (page 5). This step, i.e., the deployment, is the topic of the next chapter.

5 From CASE Tools to Integrated Systems

Based on the STS framework, we have established in the previous chapter, we introduce the formal model of a time-triggered distributed platform. It consists of a time-triggered bus (FlexRay), which connects nodes running a time-triggered OS (OSEKtime). Further on, we describe a special schedule and wrapper synthesis algorithms, which ensure the preservation of behavior of AFTM models deployed on this platform. Finally, we prove the correctness of these algorithms.

Contents

THE OUTCOME OF CHAPTER 4 IS THE CLUSTERED FUNCTIONALITY, which has to be deployed, i.e., has to be distributed over the nodes of the target platform. Then, for individual nodes bus and OS schedules have to be synthesized. Further on, the function clusters have to be wrapped again, in order to use the available communication primitives, provided by the middle-ware on the particular nodes and in order to adapt the functionality for the new execution environment. In order to be able to guarantee the preservation of properties, proved to hold for the application logic, we need to

(1) formalize the (abstract) behavioral models of the target platform (technical infrastructure),
(2) formulate the assumptions under which the property preservation is guaranteed to hold,
(3) prove the simulation relation between the task architecture model from the CASE tool (AFTM) and the model of the technical infrastructure, and
(4) discharge the assumptions from step *(2)* for the concrete application code.

The benefit of such procedure is, that for a fixed target platform steps *(1)*-*(3)* have to be accomplished once. Their results may be reused for the deployment of any application, which is modeled in AFTM. Another advantage is the possibility to automate schedule synthesis, since one of the constituents of the assumptions from step *(2)* is the formulation of the shapeliness of schedules. Every schedule, which obeys to them, is permissible and preserves the behavior of the deployed system.

To the disadvantages of the above procedure belong the potentially unnecessary strong constraints on subsystems, and/or on schedules, which may be avoided in certain cases. This can result in efficiency penalties for some applications. It is out of the scope of the present thesis to judge if these penalties are acceptable or not, since this is a highly application- and domain-dependent issue.

Contributions and Outline. The enumeration from above lists exactly the topics treated in the present chapter. We will exercise the sketched procedure for representatives of the time-triggered platform family, namely for the time-triggered OS OSEKtime [OSE01b] and the time-triggered bus FlexRay [Fle04]. Both of them are upcoming standards in the automotive industry. So, our target platform is a FlexRay network consisting of nodes with OSEKtime running on each of them. The STS models of OSEKtime and FlexRay as well as of their combination are given in Section 5.1. The procedure for schedule and wrapper synthesis, the constraints the AFTM models must fulfill, and the correctness proof of that procedure are given in Section 5.2.

The presented results were first published in [BBG+08]. Further on, in the scope of the Verisoft Automotive project[1] the AFTM, FlexRay, and OSEKtime models were formalized in Isabelle [NPW02], and substantial parts of the proof from Section 5.2 were carried out using this interactive theorem prover. Moreover, step *(4)* from the list above was exercised in Isabelle on the AFTM model of the eCall case study.

[1] see http://www.verisoft.de and http://www4.in.tum.de/~verisoft/automotive (20.06.2008)

5.1 Time-Triggered Deployment Platform

To master the inherent complexity of systems embedded in vehicles, the automotive industry came up with a number of standards based on the *time-triggered paradigm* [KG94]. They allow the realization of distributed systems with predictable time behavior, making them an appropriate deployment target for safety-critical real-time systems. In a time-triggered system actions are executed at predefined points in time. Further on, time-triggered communication protocols provide, using time synchronization, a global time base for the distributed communication partners. Thus, by combining time-triggered OS and communication system a deterministic system behavior with guaranteed response times can be achieved.

After an introduction of both standards and their combination in Section 5.1.1, we present their STS semantics in Sections 5.1.2 and 5.1.3.

5.1.1 OSEKtime & FlexRay: An Informal Introduction

Here, we give an informal introduction for the technologies designated to be the deployment target of our approach (OSEKtime OS and FlexRay bus). Their formalization is the topic of the subsequent section.

We also list simplifications we have made in the modeling of OSEKtime and FlexRay. The justification of these simplifications is that we are providing a programmer's model for these technical systems. From this perspective solutions, like several bus lines, redundant message transfer, or clock synchronization mechanisms, become transparent. Another reason is that in the combination of both technologies certain features become obsolete. For instance, the fault-tolerance mechanisms provided by the communication layer of OSEKtime and redundant FlexRay transmission links both aim at reducing the probability of incorrect message transfer. It is most likely that only one of these technologies will be applied at once.

OSEKtime OS

OSEKtime [OSE01b] is an OSEK/VDX[2] open operating system standard of the European automotive industry. OSEKtime OS supports cyclic fixed-time scheduling and provides a fault-tolerant communication mechanism.

An OSEKtime schedule defines when within a so-called scheduling *round* the dispatcher activates a user process. If another process is currently running at the scheduled activation time, it is preempted until the activated process has completed its computation. Thereafter the execution of the preempted process is resumed again. The task-state model of OSEKtime is shown in Figure 4.4b (page 96). In addition, OSEKtime monitors the deadlines of the processes, i.e., at predefined points within the scheduling round a process must have finished its computation; otherwise, an error hook is executed. The round-based scheduling procedure is repeated perpetually. All rounds have equal length and the scheduling tables for all rounds are the same.

[2] see http://www.osek-vdx.org (20.06.2008)

FTCom [OSE01c] is the fault-tolerant communication layer of OSEKtime that provides a number of primitives for the interprocess communication. Messages kept in FTCom are uniquely identified by their IDs. For every message ID FTCom realizes a buffer of length one. Applications send or receive messages by invoking the communication primitives ttSend and ttRecv, respectively, which are also provided by FTCom.

For the scope of this chapter we use the following simplifications compared to the OSEKtime OS standard:

- Every task is activated exactly once per scheduling round.
- The execution times of tasks do not overlap with each other, i.e., every task is scheduled after the worst-case execution time of the predecessor task. By this, we exclude preemption.
- The replication and RDA (Replica Determinate Agreement) [OSE01c] mechanisms of the FTCom are not used.

FlexRay Bus

FlexRay [Con03, Fle04] is a communication solution for safety-critical real-time automotive applications. It has been developed by FlexRay Consortium[3]. It is a static time division multiplexing network protocol that supports clock synchronization.

The static message transmission mode of FlexRay is based on FlexRay *rounds* consisting of a constant number of time slices of the same length, so called *slots*. A node can broadcast messages to other nodes within these slots. There can be at most one sender per slot.

For the STS model presented below several aspects of FlexRay are transparent as well:

- We use only a simple clock synchronization and FlexRay start-up algorithm. In other words, we consider the clock drift to be negligible.
- The model does not contain bus guardians [Con03] that protect channels on the physical layer from interference caused by communication that is not aligned with FlexRay schedules, e.g., from bubbling idiots.
- Only the static segment of the communication cycle of FlexRay is used, as we are mainly interested in time-triggered systems.
- Only one FlexRay channel is contained in the model, i.e., the redundancy caused by several channels is transparent to the application tasks.

Time-Triggered Architecture

The combination of the time-triggered OS and the time-triggered bus allows for synchronization of the computations and the communication. This is done by synchronizing the local clock with the help of FlexRay and by setting the length of the OSEKtime scheduling round to be a multiple

[3] see http://www.flexray.com (20.06.2008)

of the length of the FlexRay round. A unit of computation is then also one FlexRay slot. We denote the round length expressed in slots by *rl*.

The following design decisions were made for the combination of FlexRay and OSEKtime.

- Every task computation takes at most one scheduling slot.
- Both, the scheduling and the FlexRay slots have the same length.
- Both, the scheduling and the FlexRay rounds have the same length.

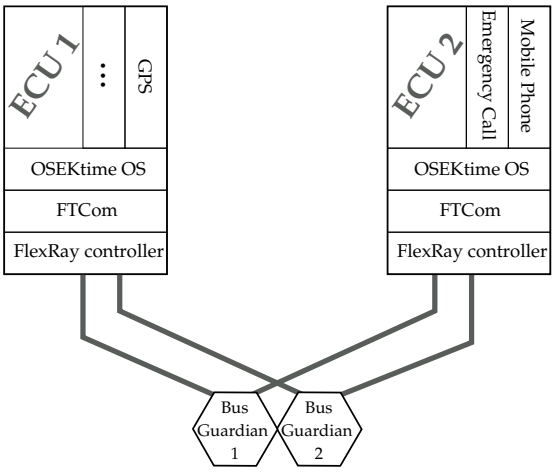


Figure 5.1: *Target Deployment Platform Architecture*

These restrictions of our model preserve the idea of time-triggered communication and computation, and thus do not restrict the applicability of the presented ideas to real systems. Figure 5.1 shows an sample network with three tasks of the eCall system deployed on two OSEKtime ECUs (electronic control units) connected by a double-redundant FlexRay link.

5.1.2 Time-Triggered Execution: OSEKtime

The formal model of OSEKtime OS is given by a special scheduled STS, which composes OSEKtime tasks according to the specification in [OSE01b].

OSEKtime Tasks

According to the standard [OSE01b, OSE01c] OSEKtime tasks communicate using special primitives: ttSend writes a message and ttRecv reads a message. The messages are kept in a special buffer of the FTCom middle-ware. Unlike the ports in AutoFocus/AFTM and input/output variables in STS, there are no constraints on reading or writing of FTCom entries: in fact, one OSEKtime task is allowed to read and write the same entry. In order to construct the STS $\mathcal{S}^{ot} \stackrel{\text{def}}{=} (I, O, L, \mathfrak{I}, \mathcal{R}^{ot}, \delta_{\text{ott}}, d^{\varepsilon})$ for an OSEKtime task we gather the entries mentioned in all ttRecv-statements of a task in the set R and all entries from ttSend-statements in S. The variables from

the intersection of both sets can be both read and written by the component. However, according to the definition of STS these sets have to be disjoint, thus we add common variables to the local variable set.[4] We define

$$I \stackrel{\text{def}}{=} R \setminus S, \qquad O \stackrel{\text{def}}{=} S \setminus R, \qquad L \supseteq R \cap S.$$

The task to make the valuations of the variables from $R \cap S$ visible to other tasks is incumbent on the scheduled STS system, which models the behavior of OSEKtime OS. It is introduced later in this section. To the set of local variables L also belong the variables used in the implementation of the OSEKtime task. We assume that they are disjoint to the set of FTCom-entries.

In the initial state of an OSEKtime task all controllable FTCom variables have to be set to $\perp$:

$$\forall \alpha \in \Lambda(V) : \alpha \vdash \mathfrak{I} \Rightarrow \forall v \in (R \cup S) \setminus I : \alpha.v = \perp. \tag{5.1}$$

Similar to the AutoFocus and AFTM models we assume that the "original" functionality of an OSEKtime task is described by a set of predicates from $\mathcal{R}^{ot}$. Every predicate captures a state change of certain variables between an activation of the task and an invocation of a special ttFinished system call, which returns the control back to the operating system and sets the task into suspended state (cf. Figure 4.4b on page 96). Like in the AFTM semantics, the values of the remaining output variables are not flushed. After the outputs are issued and the control is given over to the dispatcher the state of the OSEKtime task does not change (modulo valuations in $R \cup S$) until the next activation of the task. Thus, the transition set δ_{ott} from the $\mathcal{S}^{ot}$ tuple is built of transitions which in pairs with corresponding members of $\mathcal{R}^{ot}$ fulfill the following constraint.

$$\forall \alpha \in \Lambda(V \cup V') : \alpha \vdash d \stackrel{\text{def}}{\Leftrightarrow} (\alpha \vdash r) \wedge \forall v' \in (L' \cup O') \setminus \text{free}.r : \alpha.v' = \alpha.v. \tag{5.2}$$

More precisely, for every $d \in \delta_{\text{ott}}$ there exists $r \in \mathcal{R}^{ot}$ such that (5.2) holds and vice versa for every $r \in \mathcal{R}^{ot}$ there exists $d \in \delta_{\text{ott}}$, for which Property (5.2) is satisfied.

The idle-loop of OSEKtime is defined for all $\alpha, \beta \in \Lambda(V)$ as the complement of δ_{ott}, which preserves the state of controllable variables:

$$\alpha, \beta' \vdash d^{\varepsilon} \stackrel{\text{def}}{\Leftrightarrow} \Big(\forall \gamma \in \Lambda(V) : \alpha, \gamma' \vdash \neg \bigvee_{d \in \delta_{\text{ott}}} d\Big) \wedge (\forall v \in L \cup O : \beta.v = \alpha.v). \tag{5.3}$$

***Remark* 5.1.** In the rest of this chapter we will deploy AutoFocus task STSs, which are not allowed to have self-loops. Thus, we can assume that their realization as OSEKtime tasks will have no homonymous input/output variables, i.e., $R \cap S = \emptyset$. However, in order to retain the generality of our formal OSEKtime model, we will also address this special case in the definition of the OSEKtime composition below. ◦

Idle Task. A special type of OSEKtime tasks is the *idle task*. It is executed when no regular task needs to run. Thus, it can carry out lower prioritized activities, when free resources are available. For example, the OSEKtime OS specification [OSE01b] proposes to start an instance of OSEK OS [OSE01a] as an idle task. An idle task is suspended as soon as a regular task needs to be activated. We denote the idle task by $\mathcal{S}^{ot}_{\text{idle}}$ and assume that it does not interact with regular tasks. In other words, their state spaces (including the input and output variables) are disjoint.

[4] A local STS variable can be seen a self-loop channel: in every step the STS automaton sends its new value to itself.

OSEKtime Composition

The scheduled STS for OSEKtime parametrized by n tasks $\mathcal{S}_1^{ot}, \ldots, \mathcal{S}_n^{ot}$ with

$$\mathcal{S}_i^{ot} \stackrel{\text{def}}{=} (I_i, O_i, L_i, \mathcal{I}_i, \mathcal{R}_i^{ot}, \delta\mathsf{ott}_i, d_i^{\varepsilon}) \quad \text{for all } i \in [1,n]$$

and an idle task

$$\mathcal{S}_{\text{idle}}^{ot} \stackrel{\text{def}}{=} (I_{n+1}, O_{n+1}, L_{n+1}, \mathcal{I}_{n+1}, \mathcal{R}_{n+1}^{ot}, \delta\mathsf{ott}_{n+1}, d_{n+1}^{\varepsilon})$$

is denoted by $\mathcal{S}\mathsf{ch}^{ot} \stackrel{\text{def}}{=} (I, O, L, \mathcal{I}, \delta\mathsf{ot}, d^{\varepsilon})$. For the tasks we expect a weak version of pairwise compatibility: The tasks have to be mutually type compatible (cf. Section 4.1.6), and the local variables besides the self-loop variables from $R_i \cup S_i$ have to be disjoint. However, nothing can be assumed about the communication flow between tasks, i.e., tasks can share any input, output, or local FTCom variable.

The set of local variables L consists of three parts:

- The set of FTCom entries $FT \subset L$. Any input or output variable of any task belongs to FT. Thus, FTCom provides the only interface for the inter-task communication. Formally,

$$\bigcup_{i \in [1,n+1]} (R_i \cup S_i) \subseteq FT.$$

- The local variables of building STSs remain in the set of local variables of $\mathcal{S}\mathsf{ch}^{ot}$:

$$\bigcup_{i \in [1,n+1]} L_i \subset L.$$

- The counter for the current slot $s \in L$. W.l.o.g. we assume that $s \notin FT \cup \bigcup_{i \in [1,n+1]} L_i$. The slot counter is incremented modulo round length rl, i.e., $ty(s) = \{i \in \mathbb{N} \mid 0 \leq i < rl\}$.

OSEKtime communicates with its environment through hardware buffers, which are provided by the controllers of the physical node, it runs on. We assume that for every peripheral device (e.g., a bus link or a directly connected sensor) there exists exactly one controller, which provides buffers for reading and for writing messages. For our programmer's model it is adequate to model buffers of every controller as input/output variables of OSEKtime STS. Every variable is typed with the whole valuation domain (M^*), i.e., every message can be sent through every controller. This solution abstracts from common procedures accomplished by FTCom and/or controllers during the inter-node communication, like packing/unpacking messages or splitting/merging them in the case if the hardware buffers are either smaller or bigger than the FTCom entries. This design decision is sustainable because those procedures are transparent for the application tasks.

For the rest of this thesis we will work with computation nodes, which are equipped with a FlexRay controller. An OSEKtime system has to copy values from FTCom to its output variables and from input variables to FTCom. This task is carried out by the FlexRay *device driver* according

to the global communication schedule. By this, we model the FlexRay buffer in OSEKtime by one input and one output variable:

$$frb^{in} \in I \qquad \text{and} \qquad frb^{out} \in O.$$

For every scheduled OSEKtime STS $\mathcal{Sch}^{ot}$ we need a schedule for its tasks, a (global) communication schedule, which determines slots, in which the node is allowed to send as well as a mapping from the current slot number to an FTCom entry, whose message is transported on the bus at this time. This information is kept in the following functions.

$$\begin{aligned} nxt^{exe} &: [0, rl) \mapsto [1, n+1], \\ nxt^{com} &: [0, rl) \mapsto \mathbb{B}, \\ map^{com} &: [0, rl) \mapsto \mathsf{VAR}. \end{aligned}$$

In every slot s the task with the number $nxt^{exe}(s)$ is executed. The task with the number $n+1$ is the idle task. It is scheduled in every free slot. By this, nxt^{exe} is a total function.

If $nxt^{com}(s)$ is true in slot s, the message $map^{com}(s) \in FT$ is transported (e.g., copied into frb^{out}). Otherwise, the value of frb^{in} is copied into the FTCom entry $map^{com}(s)$. Intuitively, this means that in one slot a node can either send or receive a message through FlexRay, but not both. We assume that for every slot there exists a variable whose value is scheduled for sending over bus. By this, the communication map map^{com} is a total function. However, if we restrict its range to FT, it needs not to be total. This expresses the fact that not all messages transported by the bus are addressed to the subsystem, deployed on the considered OSEKtime node.

The initial state of the OSEKtime system is described by the following predicate. The slot counter is set to 0 and the FlexRay output buffer is set to $\perp$. Further on, due to Constraint (5.1) which every initial condition of an OSEKtime task has to fulfill, the task-controllable variables from FT are set to $\perp$ as well. Additionally, it must be ensured that the non-task-controllable FT-entries also obtain the $\perp$-valuation:

$$\forall \alpha \in \Lambda(V) : \alpha \vdash \mathcal{I} \overset{\text{def}}{\Leftrightarrow} \Big(\alpha \vdash \bigwedge_{i \in [1,n]} \mathcal{I}_i\Big) \wedge \alpha.s = 0 \wedge \alpha.frb^{out} = \perp \wedge \forall v \in FT : \alpha.v = \perp. \tag{5.4}$$

Every transition in the set $\delta\mathsf{ot}$ is built by the composition of a computation transition d^{exe} and a communication transition d^{com}. Thus, $\forall d \in \delta\mathsf{ot} : d \overset{\text{def}}{=} d^{com} \circ d^{exe}$ with the parts defined for all $\alpha \in \Lambda(V \cup V')$ as following:

$$\begin{aligned} \alpha \vdash d^{exe} \overset{\text{def}}{\Leftrightarrow} \quad & \exists d^{loc} \in \delta\mathsf{ott}_{nxt^{exe}(\alpha.s)} \cup \{d^{\varepsilon}_{nxt^{exe}(\alpha.s)}\} : \alpha \vdash d^{loc} \\ & \wedge \forall i \in [1,n] : i \neq nxt^{exe}(\alpha.s) \Rightarrow \forall v \in (L_i \cup O_i) \setminus V_{nxt^{exe}(\alpha.s)} : \alpha.v' = \alpha.v \qquad (5.5a) \\ & \wedge \alpha.frb^{in\prime} = \alpha.frb^{in} \wedge \alpha.frb^{out\prime} = \alpha.frb^{out} \wedge \alpha.s' = \alpha.s, \end{aligned}$$

$$\begin{aligned} \alpha \vdash d^{com} \overset{\text{def}}{\Leftrightarrow} \quad & \alpha.s' = \alpha.s + 1 \mod rl \\ & \wedge \forall v \in FT : \alpha.v' = \begin{cases} \alpha.frb^{in} & \text{if } v = map^{com}(\alpha.s), \\ \alpha.v, & \text{o.w.} \end{cases} \qquad (5.5b) \\ & \wedge \alpha.frb^{out\prime} = \begin{cases} \alpha.map^{com}(\alpha.s) & \text{if } nxt^{com}(\alpha.s'), \\ \alpha.frb^{out}, & \text{o.w.} \end{cases} \end{aligned}$$

d^{exe} executes the task scheduled for the current slot according to the scheduling table in nxt^{exe}. Since the idle-loop of a task is defined as a complement of the transition set, Property (4.3) is not violated, i.e., the idle-loop is only fired if nothing useful can be done for the current input. All other tasks do not change their states.

d^{com} increases the slot counter modulo round length, and copies the message in the FlexRay input buffer to the corresponding FTCom entry. Finally, if the node may send in the next FlexRay slot, the message to be sent is copied into the FlexRay output buffer. If the node sends some FTCom entry on the bus, we assume that it also receives this entry. This is ensured by the FlexRay bus protocol, described in the next section.

Please note that the last line of Equation (5.5a) does *not* violate Property (4.2) on page 77, which states that a transition cannot constrain input valuations in the next state. The reason is that the valuations of primed variables, which satisfy d^{exe}, describe an intermediate automaton state. The final successor state is given by d^{com}.

The idle-loop transition of OSEKtime is never fired. This is explained by the fact that there exists an ordinary transition for every input valuation. Thus, in order to fulfill Property (4.2) we define $d^{\varepsilon} \stackrel{\text{def}}{=} \text{ff}$.

5.1.3 Time-Triggered Communication: FlexRay

Next we would like to compose OSEKtime nodes from the last section using the FlexRay protocol. We formalize this by the scheduled FlexRay STS $\mathcal{Sch}^{OF} \stackrel{\text{def}}{=} (I, O, L, \mathcal{J}, \delta_{\text{of}}, d^{\varepsilon})$. Given n scheduled OSEKtime STSs $\mathcal{Sch}_1^{ot}, \ldots, \mathcal{Sch}_n^{ot}$ with

$$\mathcal{Sch}_i^{ot} \stackrel{\text{def}}{=} (I_i, O_i, L_i, \mathcal{J}_i, \delta_{\text{ot}_i}, d_i^{\varepsilon}), \qquad V_i \stackrel{\text{def}}{=} I_i \cup L_i \cup O_i,$$

pair-wise disjoint variable sets, and associated scheduling functions nxt_i^{exe}, nxt_i^{com}, and map_i^{com} for all $i \in [1,n]$, we compose them to $\mathcal{Sch}^{OF}$ by imposing following constraint on the global communication schedule:

$$\forall s \in [0, rl), i, j \in [1,n] : i \neq j \Rightarrow \neg(nxt_i^{com}(s) \wedge nxt_j^{com}(s)). \tag{5.6}$$

This constraint formally captures the fact that in each slot at most one node is allowed to send. Constraint (5.6) allows us to unite all communication schedules in an unambiguous way. We define the global communication schedule function $nxt^{FR} : [0, rl) \mapsto [0,n]$ as

$$\begin{aligned} \forall s \in [0, rl) : \quad & (\forall i \in [1,n] : nxt^{FR}(s) = i \stackrel{\text{def}}{\Leftrightarrow} nxt_i^{com}(s)) \\ & \wedge (nxt^{FR}(s) = 0 \stackrel{\text{def}}{\Leftrightarrow} \neg \bigvee_{i \in [1,n]} nxt_i^{com}(s)). \end{aligned} \tag{5.7}$$

If no node sends a message in a certain slot, we set nxt^{FR} to 0.

***Remark* 5.2** (Combining FlexRay systems)**.** This special case $nxt^{FR}(s) = 0$ does not indicate the absence of communication, but rather marks communication that does not concern the current

system. The semantics of OSEKtime ensures that this communication does not influence the system behavior. By this, we could define mechanisms for the combination of several communication networks which use the same physical link to a further scheduled FlexRay STS. However, this would go beyond the scope of the present thesis. Thus, we simply leave this case underspecified in the semantics definition bellow. ◦

The local variables of the FlexRay STS contain all input and output buffers as well as all local variables of OSEKtime STSs. Please note that the fact that the scheduled OSEKtime STSs have pairwise disjoint variable sets implies that the names of their FTCom entries are disjoint also. We will take care of this fact in the section on adaptation and wrapper generation (5.2.3). Formally, holds

$$\bigcup_{i\in[1,n]} (L_i \cup \{frb_i^{in}, frb_i^{out}\}) \subseteq L.$$

The initial state of the OSEKtime/FlexRay system is simply described by the conjunction of the initial states of all OSEKtime subsystems,

$$\forall \alpha \in \Lambda(V) : \alpha \vdash \mathcal{I} \overset{\text{def}}{\Leftrightarrow} \alpha \vdash \bigwedge_{i\in[1,n]} \mathcal{I}_i. \tag{5.8}$$

Every transition in the set $\delta\mathsf{of}$ makes a transition step for every OSEKtime subsystem $\mathcal{S}\mathsf{ch}_i^{ot}$ and broadcasts the value from the FlexRay output buffer of the node, which is allowed to send according to the communication schedule, to the input variables of all OSEKtime notes. Formally, for all $d \in \delta\mathsf{of}$ holds

$$\begin{aligned}\forall \alpha \in \Lambda(V \cup V') : \alpha \vdash d \overset{\text{def}}{\Leftrightarrow} \quad & \forall i \in [1,n] : \exists d_i \in \delta\mathsf{ot}_i : \alpha \vdash d_i \\ & \wedge nxt^{FR}(\alpha.s) > 0 \Rightarrow \forall i \in [1,n] : \alpha.frb_i^{in'} = \alpha.frb^{out}_{nxt^{FR}(\alpha.s)}\end{aligned} \tag{5.9}$$

Finally, the idle-loop transition is defined as $d^{\varepsilon} \overset{\text{def}}{=} \wedge_{i\in[1,n]} d_i^{\varepsilon} = \mathrm{ff}$.

5.2 Propety Preserving Deployment

The problem we are dealing with in the next sections is: Given n pairwise compatible AutoFocus task STSs $\mathcal{S}_1^{aft}, \ldots, \mathcal{S}_n^{aft}$ with $\mathcal{S}_i^{aft} \overset{\text{def}}{=} (I_i, L_i, O_i, \mathcal{I}_i^{aft}, \mathcal{R}_i^{aft}, \delta\mathsf{aft}_i, d_i^{aft\varepsilon})$ for all $i \in [1,n]$ and n pairwise compatible OSEKtime task STSs $\mathcal{S}_1^{ot}, \ldots, \mathcal{S}_n^{ot}$ with $\mathcal{S}_i^{ot} \overset{\text{def}}{=} (I_i, L_i, O_i, \mathcal{I}_i^{ot}, \mathcal{R}_i^{ot}, \delta\mathsf{ott}_i, d_i^{ot\varepsilon})$ for all $i \in [1,n]$ such that they stand in a pairwise refinement relation to each other:

$$\forall i \in [1,n] : \mathcal{S}_i^{aft} \rightsquigarrow \mathcal{S}_i^{ot}.$$

How can we build the scheduled OSEKtime/FlexRay STS

$$\mathcal{S}\mathsf{ch}^{OF} = (I^{OF}, L^{OF}, O^{OF}, \mathcal{I}^{OF}, \delta\mathsf{of}, d^{OF\varepsilon}),$$

which provably refines the scheduled AFTM STS

$$\mathcal{S}\mathsf{ch}^{aftm} = (I^{aftm}, L^{aftm}, O^{aftm}, \mathcal{I}^{aftm}, \delta\mathsf{aftm}, d^{aftm\varepsilon}),$$

both built out of the corresponding AutoFocus/OSEKtime tasks? We will answer this question in three steps. Firstly, we describe a mapping from a network of AutoFocus tasks (AFTM model) to a network of nodes running OSEKtime and connected by FlexRay (cf. Section 5.2.1). Secondly, in Section 5.2.2 we provide a schedule synthesis algorithm, which preserves the original behavior of AFTM models in a given OSEKtime/FlexRay network and for a given mapping on it. Finally, in Section 5.2.3 we adapt AutoFocus tasks for their new run-time environment and prove the property preservation between the AFTM model and the deployed system.

***Remark* 5.3** (Correctness of generated task code)**.** The refinement relation between individual AFTM and OSEKtime tasks can be achieved using code generators of AutoFocus mentioned in Section 4.2. In order to ensure the correctness of code generation without proving the correctness of the code generator, the well-known technique called *translation validation* [PSS98] can be applied. The correctness of an AutoFocus task relies on the correctness of individual transition steps of the underlying I/O automaton. Their correctness can be guaranteed by generating assertions for every piece of code which implements an individual transition step. These assertions have to be proved for the real code.

The procedure sketched above was applied to the eCall study in the scope of the Verisoft Automotive project. Therefore, C-code and assertions ware generated automatically out of the AFTM model of the emergency call. Afterwards, the assertions were discharged for the eCall code using the Hoare-logic verification environment of the Isabelle prover [Sch06]. ◦

5.2.1 Mapping AFTM to OSEKtime/FlexRay

The outcome of this section is a *deployment function*, which captures information needed to initialize an OSEKtime/FlexRay network by an AFTM model. More precisely, it describes the data-flow network of the AFTM model on an OSEKtime/FlexRay platform. There, the nodes are simulated by corresponding OSEKtime tasks, ports by FTCom entries and channels by FlexRay slots. This function gives us a possibility to describe and reason about deployment solutions.

As described as the beginning of Section 5.2, AFTM tasks are replaced by OSEKtime tasks, which stand in the abstraction/refinement-relation to each other. Thus, the remaining constituents of a deployment solution are

- a distribution of AFTM tasks over an OSEKtime/FlexRay network,
- a mapping of AutoFocus ports to FTCom entries, and
- a mapping of AutoFocus channels to sending slots.

These mappings underlie certain constraints. In fact we want to deal with deployment functions only, which map *all* tasks, channels and ports in an *unambiguous* way. In this context we speak about *valid* deployment functions – these are functions, which fulfill all constraints listed below.

Distribution

Given n pairwise compatible OSEKtime task STSs $\mathcal{S}^{ot}_1,\ldots,\mathcal{S}^{ot}_n$ and m OSEKtime OS nodes $\mathcal{S}\mathsf{ch}^{ot}_1,\ldots,\mathcal{S}\mathsf{ch}^{ot}_m$, we use the total surjective function $depl\colon [1,n] \mapsto [1,m]$ to map every task to

a node. Thus, every scheduled OSEKtime STS $\mathcal{S}\mathsf{ch}_j^{ot}$ with $j \in [1,m]$ is built over a task set $\{\mathcal{S}_i^{ot} \mid depl(i) = j\}$. Because *depl* is surjective, this set is not empty for any j.

***Remark* 5.4** (Coexistence with another systems)**.** By the definition of *depl* we do not disallow the presence of alien functionality (e.g., non-AFTM tasks deployed on the same OSEKtime nodes). We just ensure that every of n tasks is uniquely assigned to one of m nodes and every node obtains at least one task. For reasons of simplicity we assume that the presence of alien functionality does not influence the behavior of the system. On the one hand, this is guaranteed by the resource management mechanisms of OSEKtime and FlexRay, on the other hand, we demand that FTCom entries which belong to the system are never modified by alien tasks. ◦

The identifier number of the task $i \in [1,n]$ on the node $depl(i)$ is defined by $locid(i)$. In other words, the mapping *locid* transforms task indexes from the interval $[1,n]$ to the local numbering on each node.

Scheduling and Communication

In order to be able to reason about scheduling of particular AFTM tasks, we define here the global computation and communication schedules. Moreover, we describe additional constraints on communication schedules of a valid deployment function. For these purposes we make the following auxiliary definitions. We define the set of all *external* variables, i.e., variables which must be sent through the bus as

$$EXT \stackrel{\text{def}}{=} \{v \mid \exists i, j \in [1,n] : depl(i) \neq depl(j) \wedge v \in (I_i \cap O_j)\} \cup I^{aftm} \cup O^{aftm}.$$

We recall that I^{aftm} and O^{aftm} are the input and output variables, respectively, in the AFTM composition of the tasks $\mathcal{S}_1^{aft}, \ldots, \mathcal{S}_n^{aft}$. *EXT* contains all input/output variable pairs, which belong to the tasks deployed on different nodes and all variables, which are sent to/received from environment. Further on, we gather the set of all remaining variables in $INT \stackrel{\text{def}}{=} V^{aftm} \setminus EXT$, where V^{aftm} is the set of all variables in the AFTM composition.

Following constraints must hold for any valid deployment function. We demand that every input/output variable, which must be sent over the bus, is mapped to a separate sending slot. All other variables may not be mapped to any slot:

$$\{map^{com}(s) \mid s \in [0,rl)\} \supseteq EXT \qquad \text{and} \qquad \{map^{com}(s) \mid s \in [0,rl)\} \cap INT = \emptyset. \tag{5.10}$$

Moreover, the predicates nxt^{com} on every node must provide send slots for every output variable sent over FlexRay and must disallow writing on the bus, when inputs are transferred:

$$\forall s \in [0,rl), i \in [1,n] : nxt^{com}_{depl(i)}(s) \Leftrightarrow map^{com}(s) \in O_i. \tag{5.11}$$

Finally, we introduce two functions, which are inverse to the scheduling functions of OSEKtime/FlexRay. We define $schd^{exe}\colon [1,n] \mapsto [0,rl)$ and $schd^{com}\colon \mathsf{VAR} \mapsto [0,rl)$ such that for all $s \in [0rl)$ and $i \in [1,n]$

$$schd^{exe}(i) = s \overset{\text{def}}{\Leftrightarrow} nxt^{exe}_{depl(i)}(s) = locid(i),$$

$$\forall v \in EXT : schd^{com}(v) = s \overset{\text{def}}{\Leftrightarrow} \begin{cases} map^{com}(s) = v & \text{if } v \in I^{aftm}, \\ \exists i \in [1,n] : nxt^{exe}_{depl(i)}(s) = locid(i) \wedge v \in O_i, & \text{o.w.} \end{cases}$$

We demand $schd^{exe}$ to be total and injective for all tasks mapped to the same node:

$$\forall i,j \in [1,n] : (depl(i) = depl(j) \wedge i \neq j) \Rightarrow schd^{exe}(i) \neq schd^{exe}(j).$$

Thus, every task is scheduled exactly once at a unique slot number.

The function $schd^{com}$ maps every input/output variable to the slot, when it is updated by the next value: If the variable is an external input variable, this is the slot when it is transported on the bus. Otherwise, it is the slot when the variable is produced. Since every task is allowed to produce in one step several outputs (on different output variables), the function $schd^{com}$ can potentially map multiple local variables to the same slot.

5.2.2 Scheduler Synthesis Procedure

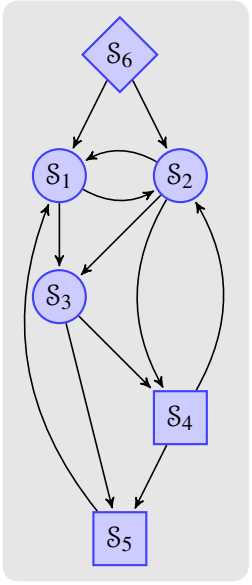
In the previous section we have provided means to describe deployment functions, which map AFTM models to OSEKtime/FlexRay networks. The constraints, which every valid deployment function must fulfill, ensure that all tasks are deployed unambiguously and the data-flow between the tasks is recreated correctly. In this section we describe the conditions, which all valid deployment functions must satisfy, in order to preserve the behavior of the AFTM model. These conditions address OSEKtime and FlexRay schedules. Thus, in the following we describe the scheduler synthesis procedures for both constituents of our target platform.

Example 5.1. We will illustrate introduced concepts using the example from Figure 5.2a. It is designed to exhibit the most interesting causal relations between tasks. There, the circles denote *OR*-tasks, the rectangles *AND*-tasks, and the diamond denotes the environment ($\top$-task). Directed edges stand for the data flow between tasks (homonymous input/output variable pairs). For the reasons of simplicity, we assume that if two tasks are connected by an edge, they share exactly one variable. ◦

OSEKtime Schedule Synthesis

To simulate AFTM by a deployed system a scheduling constraint is necessary: If a task sends its outputs through FlexRay, the FlexRay transfer must be scheduled *after* the producing task. Formally,

$$\forall i \in [1,n], v \in (O_i \cap EXT) : schd^{exe}(i) < schd^{com}(v). \tag{5.12}$$



(a)

Slot	Node 1	Node 2	Communication
0			$(\mathcal{S}_6, \mathcal{S}_1)$
1	$\mathcal{S}_1$		$(\mathcal{S}_6, \mathcal{S}_2)$
2			$(\mathcal{S}_1, \mathcal{S}_2)$
3		$\mathcal{S}_2$	
4			$(\mathcal{S}_2, \mathcal{S}_3)$
5	$\mathcal{S}_3$		$(\mathcal{S}_2, \mathcal{S}_1)$
6			$(\mathcal{S}_3, \mathcal{S}_4)$
7		$\mathcal{S}_4$	
8			$(\mathcal{S}_4, \mathcal{S}_5)$
9	$\mathcal{S}_5$		

(b)

to \ from	$\mathcal{S}_6$	$\mathcal{S}_1$	$\mathcal{S}_2$	$\mathcal{S}_3$	$\mathcal{S}_4$	$\mathcal{S}_5$
$\mathcal{S}_1$	1	-	1	-	-	0
$\mathcal{S}_2$	1	0	-	-	0	-
$\mathcal{S}_3$	-	1	1	-	-	-
$\mathcal{S}_4$	-	-	1	1	-	-
$\mathcal{S}_5$	-	-	-	1	1	-

(c)

Figure 5.2: *Auto*Focus *Task Graph (a) with a Sample Schedule (b) and Delays (c)*

Example 5.1 (*con'ed*). Assuming the tasks from Figure 5.2a are deployed on two nodes ($\mathcal{S}_1$, $\mathcal{S}_3$, $\mathcal{S}_5$ on node 1 and $\mathcal{S}_2$, $\mathcal{S}_4$ on node 2), then Table 5.2b shows a sample schedule for these tasks. The round length is 10. The unallocated computation slots can be used by other tasks which do not belong to this task graph. The environment task is not deployed explicitly. It is merely represented by the FlexRay slots for the input/output messages it produces/consumes. The assumption is that the behavior of the environment task in AFTM and the real environment (which can be realized by several nodes/tasks) are equivalent. ◦

Communication Schedule Synthesis

While an AFTM system is allowed to have arbitrary communication cycles (except for self-loops), according to the OSEKtime specification, there exists only one cycle with the fixed length of one round. For example, in AFTM the cycle between tasks $\mathcal{S}_1$, $\mathcal{S}_3$, and $\mathcal{S}_5$ from Figure 5.2a means that the environment task $\mathcal{S}_6$ will send its output four times to $\mathcal{S}_1$ before the first input from

$\mathcal{S}_5$ (together with the fifth one from $\mathcal{S}_6$ and the third one from $\mathcal{S}_2$) will arrive. In this context we will speak about the *age* of an input: the number of messages arrived through a specific channel so far. On the other hand, in the corresponding OSEKtime system, having a schedule in which every task is scheduled exactly once (cf. Table 5.2b), the first input from $\mathcal{S}_5$ will be processed by $\mathcal{S}_1$ together with the second input from $\mathcal{S}_6$. By this, a naïve deployment approach would lead to deviations from the communication semantics established by AFTM.

This problem is mastered by installing *delay buffers* of length one on communication links in the deployed system. They exactly simulate the communication semantics of AFTM. For a given channel between two tasks $\mathcal{S}_i^{ot}$ and $\mathcal{S}_j^{ot}$ the communication has to be either delayed by one if the data sent through this channel arrives *before* the task $\mathcal{S}_j^{ot}$ is started, or the delay emerges naturally, otherwise. By this, the set of all *input delays* $delay$: $\mathsf{VAR} \mapsto \mathbb{B}$ is defined for all input variables by:

$$\forall i \in [1,n] : \forall v \in I_i : delay(v) = \begin{cases} \text{tt} & \text{if } schd^{com}(v) < schd^{exe}(i), \\ \text{ff}, & \text{o.w.} \end{cases} \tag{5.13}$$

As the result of the deployment, every task $\mathcal{S}_i^{ot}$ gets $|\{v \in I_i \mid delay(v)\}|$ dedicated buffers for its corresponding inputs. Table 5.2c shows the buffer lengths for each channel from our example. They are initialized with an $\perp$. At the beginning of its activation the task reads its inputs from FTCom (*FT*) and puts them into the corresponding buffer. The values which fall out of the buffer are the actual inputs for the task logic.

(r,s)	$(0,1)$	$(1,1)$	$(2,1)$	$(3,1)$	$(4,1)$
1. *FT*	$i_6^{(1)}, \perp_2, \perp_5$	$i_6^{(2)}, i_2^{(1)}, \perp_5$	$i_6^{(3)}, i_2^{(2)}, \perp_5$	$i_6^{(4)}, i_2^{(3)}, \perp_5$	$i_6^{(5)}, i_2^{(4)}, i_5^{(1)}$
$(\mathcal{S}_6, \mathcal{S}_1)$	$i_6^{(1)}$	$i_6^{(2)}$	$i_6^{(3)}$	$i_6^{(4)}$	$i_6^{(5)}$
2. $(\mathcal{S}_2, \mathcal{S}_1)$	$\perp_2$	$i_2^{(1)}$	$i_2^{(2)}$	$i_2^{(3)}$	$i_2^{(4)}$
$(\mathcal{S}_5, \mathcal{S}_1)$	–	–	–	–	–
3. inputs	$\perp, \perp, \perp_5$	$i_6^{(1)}, \perp_2, \perp_5$	$i_6^{(2)}, i_2^{(1)}, \perp_5$	$i_6^{(3)}, i_2^{(2)}, \perp_5$	$i_6^{(4)}, i_2^{(3)}, i_5^{(1)}$

Table 5.1: *Message/Input Buffer Content & Inputs of the Task* $\mathcal{S}_1$

Example 5.1 (*con'ed*). The described behavior is illustrated by Figure 5.1 for the task $\mathcal{S}_1$ from Figure 5.2a. There, the inputs for five invocations of the task according to the schedule from Table 5.2b are listed. The time point of every invocation is denoted by a round number/slot number-pair (r,s). The upper index denotes the age of an input, while the lower one is the index of its producer. The absence of a message from the task $\mathcal{S}_i$ is indicated by $\perp_i$. The first line shows the content of the message buffer at the specified points in time. These values are inserted into the buffers of $\mathcal{S}_1$, shown on the second line. The values, which fall out thereby, as well as the input from the task $\mathcal{S}_5$ (for which no buffering is needed, cf. Table 5.2c), are the actual inputs of $\mathcal{S}_1$, listed in line three. This demonstrates that with the help of these buffers the tasks in AFTM and OSEKtime will work on consistent inputs. This statement will be proved as a part of the simulation in Section 5.2.3. ∘

***Remark* 5.5** (Diversity vs. efficiency)**.** The delay buffers slow down the data flow in the system. Every message transfer lasts exactly one round. This solution is surely superfluous for applications with acyclic data flow dependencies. In these cases there always exists a sequentialization, which allows to establish slot-wise communication delays. On the other hand, forbidding data-flow cycles would substantially constrain the set of deployable models and a buffer-free deployment procedure would reduce the flexibility of the schedule synthesis. It depends on the concrete application domain, which solution is more preferable. ◦

5.2.3 Simulation

From the previous sections we know that in general the behavior of AutoFocus tasks has to be adapted before it can be deployed on an OSEKtime/FlexRay network in order to preserve its behavior. The reasons are (1) the deviations in the communication and execution semantics and (2) special communication primitives (e.g., ttRecv and ttSend), which have to be inserted instead of shared input/output variables. This is done in the current section before the simulation proof between AFTM and OSEKtime/FlexRay.

Wrapper and Code Generation

In order to incorporate the delay buffers introduced above into the OSEKtime system, we have two options: To modify the OSEKtime semantics or to wrap the original STS components. Here, we choose the second one, because it promises a better integration with other (legacy) subsystems.

For every OSEKtime task STS $\mathcal{S}^{ot} = (I, L, O, \mathcal{I}^{ot}, \mathcal{R}^{ot}, \delta_{\mathsf{ott}}, d^{ot\varepsilon})$ ($V = I \cup L \cup O$) with the set of input variables, which have to be delayed, $Q \subseteq I$, we define a wrapped STS

$$\mathcal{W}(\mathcal{S}^{ot}, Q) \stackrel{\text{def}}{=} (I, L \cup B_q, O, \mathcal{I}^{w}, \delta_{\mathsf{wott}}, d^{w\varepsilon})$$

with the set of distinguished buffer variables:

$$B_q \stackrel{\text{def}}{=} \{v_q \mid v \in Q\} \qquad \text{such that} \qquad B_q \cap V = \emptyset \qquad \text{and} \qquad \forall v \in Q : ty(v) = ty(v_q).$$

The initial state of $\mathcal{W}(\mathcal{S}^{ot}, Q)$ is defined for all $\alpha \in \Lambda(V^w)$, where V^w is the set of all variables of the wrapped task, as

$$\alpha \vdash \mathcal{I}^{w} \stackrel{\text{def}}{\Leftrightarrow} \alpha \vdash \mathcal{I}^{ot} \wedge \forall v_q \in B_q : \alpha.v_q = \bot. \tag{5.14}$$

Before we define the behavior of wrapped OSEKtime tasks, we will fix the current input of a task in the state $\alpha \in \Lambda(V^w)$ by the valuation $otTI(\mathcal{W}(\mathcal{S}^{ot}, Q), \alpha) \in \Lambda(I)$ defined as

$$\forall v \in I : otTI(\mathcal{W}(\mathcal{S}^{ot}, Q), \alpha).v \stackrel{\text{def}}{=} \begin{cases} eti(Q, \alpha).v & \text{if } eti(Q, \alpha).v \neq \bot, \\ \alpha.v_I, & \text{o.w.}, \end{cases} \tag{5.15a}$$

$$\forall v \in I : eti(Q, \alpha).v \stackrel{\text{def}}{=} \begin{cases} \alpha.v_q & \text{if } v \in Q, \\ \alpha.v, & \text{o.w.} \end{cases} \tag{5.15b}$$

where the variable $v_l \in L_I$ is the local copy of the previous input, as defined in Section 4.3.2. The valuation function returns the input, which comes either from a buffer variable v_q (if exists) or from an FTCom entry v if and only if this input is not $\perp$. Otherwise, the old value stored in v_l is returned.

The wrapped component behaves the same way as the original STS with input values replaced by the valuations from *eti*:

$$\begin{aligned}\forall d \in \delta_{\mathsf{wott}}, \alpha \in \Lambda(V^w \cup V^{w\prime}) : \\ \alpha \vdash d \overset{\text{def}}{\Leftrightarrow} \quad & \exists d^{loc} \in \delta_{\mathsf{ott}}, \beta \in \Lambda(V \cup V') : \beta \vdash d^{loc} \\ & \wedge \beta \overset{I}{=} eti(Q, \alpha) \wedge \beta \overset{L \cup O}{=} \alpha \wedge \beta \overset{V'}{=} \alpha \\ & \wedge \forall v \in Q : \alpha.v'_q = \alpha.v.\end{aligned}$$

The above definition consists of two parts: The first part describes the inputs and outputs of the wrapped OSEKtime task. This is done as explained above. The second part defines the successor states of additional local variables, introduced by the wrapper: The external inputs from the FTCom are copied into the buffer variables.

Finally, the idle-loop yields the idle-loop of the OSEKtime task and preserves the values of all new variables:

$$\forall \alpha \in \Lambda(V^w \cup V^{w\prime}) : \alpha \vdash d^{w\varepsilon} \overset{\text{def}}{\Leftrightarrow} (\alpha \vdash d^{ot\varepsilon}) \wedge \forall v_q \in B_q : \alpha.v'_q = \alpha.v_q.$$

***Remark* 5.6** (Disjointness of physical state spaces)**.** Another preliminary for the successful composition is the disjointness of variable sets of scheduled OSEKtime STSs. The communication mechanism of AFTM via homonymous variables is now replaced by the scheduled OSEKtime and FlexRay composition. While the tasks deployed on one OSEKtime node can share their I/O state space (the FTCom buffer), the state space of physically distributed tasks gets naturally disjoint and the valuations of the homonymous input/output-variables become occasionally de-synchronized. The natural solution of this problem is the renaming of homonymous variable pairs in all physically distributed tasks. For this purpose we can use the renaming mechanism for STSs presented in Section 4.1.6. However, in order to keep the discussion and proofs simple, we do not provide the formal foundation for the underpinning of this idea. We merely assume that the refinement relation between OSEKtime and AFTM tasks and the $\overset{(.)}{=}$-relation consider these necessary renamings, i.e., compare AFTM task variables with their corresponding uniquely renamed counterparts in OSEKtime task. ◦

Simulation Proof

The subsequent theorem employs the following notation: for a given OSEKtime/FlexRay run $\rho \in \langle\langle \mathcal{S}\mathsf{ch}^{OF} \rangle\rangle$ we access the state *at the beginning* of the sth slot in the rth round (where $s \in [0, rl)$ and $r \in \mathbb{N}$) by

$$\rho.(r, s) \overset{\text{def}}{=} \rho.(r \cdot rl + s).$$

Further on, we define the increment and the decrement of a round/slot-pair as

$$(r,s)+1 \stackrel{\text{def}}{=} \begin{cases}(r,s+1) & \text{if } s < rl-1,\\ (r+1,0), & \text{o.w.}\end{cases} \qquad \text{and} \qquad (r,s)-1 \stackrel{\text{def}}{=} \begin{cases}(r,s-1) & \text{if } 0 < s,\\ (r-1,rl-1), & \text{o.w.,}\end{cases}$$

respectively.

Finally, given n AutoFocus and OSEKtime tasks $\mathfrak{S}_1^{aft},\ldots,\mathfrak{S}_n^{aft}$ and $\mathfrak{S}_1^{ot},\ldots,\mathfrak{S}_n^{ot}$ such that $\mathfrak{S}_i^{aft} \rightsquigarrow \mathfrak{S}_i^{ot}$ for every $i \in [1,n]$, we denote the corresponding variable sets by V_i for every involved task pair i. The same is done for the input, local, and output variable sets, which are denoted by I_i, L_i, and O_i, respectively. OSEKtime tasks wrapped according to the *delay*-predicate are denoted by $\mathcal{W}_q(\mathfrak{S}_i^{ot})$ and defined as

$$\forall i \in [1,n] : \mathcal{W}_q(\mathfrak{S}_i^{ot}) \stackrel{\text{def}}{=} \mathcal{W}(\mathfrak{S}_i^{ot}, \{v \in I_i \mid delay(v)\}).$$

Theorem 5.1 (Simulation: OSEKtime/FlexRay vs. AFTM)**.** *Given a scheduled AFTM system* $\mathcal{Sch}^{aftm} \stackrel{\text{def}}{=} (I,L,O,\mathcal{I},\delta_{\mathsf{aftm}},d^{\varepsilon})$ *built over n pairwise compatible AutoFocus tasks* $\mathfrak{S}_1^{aft},\ldots,\mathfrak{S}_n^{aft}$ *and an OSEKtime/FlexRay system* $\mathcal{Sch}^{OF}$ *with round length* $rl \in \mathbb{N}$*, parametrized by n pairwise compatible wrapped OSEKtime tasks deployed on m OSEKtime nodes*

$$\mathcal{W}_q(\mathfrak{S}_1^{ot}),\ldots,\mathcal{W}_q(\mathfrak{S}_n^{ot}) \qquad \textit{and} \qquad \mathcal{Sch}_1^{ot},\ldots,\mathcal{Sch}_m^{ot},$$

respectively, and provided $\mathfrak{S}_i^{ot} \rightsquigarrow \mathfrak{S}_i^{aft}$ *for all* $i \in [1,n]$*, and given a valid deployment function, described by mappings*

$$depl\colon [1,n] \mapsto [1,m], \qquad schd^{exe}\colon [1,n] \mapsto [0,rl), \qquad \textit{and} \qquad schd^{com}\colon \mathsf{VAR} \mapsto [0,rl),$$

we prove that for every OSEKtime/FlexRay run $\rho^{OF} \in \langle\!\langle \mathcal{Sch}^{OF} \rangle\!\rangle$ *there exists a run of AFTM* $\rho^{aftm} \in \langle\!\langle \mathcal{Sch}^{aftm} \rangle\!\rangle$ *such that for all* $i \in [1,n]$ *and* $r \in \mathbb{N}$ *the following simulation relation holds between every pair of states* $(\rho^{aftm}.r, \rho^{OF}.(r, schd^{exe}(i)))$*:*

$$\rho^{aftm}.r \stackrel{L_i}{=} \rho^{OF}.(r, schd^{exe}(i)), \tag{5.16a}$$

$$\rho^{aftm}.r \stackrel{O_i}{=} \rho^{OF}.(r, schd^{exe}(i)), \tag{5.16b}$$

$$afTI(\rho^{aftm}.r) \stackrel{I_i}{=} otTI(\mathcal{W}_q(\mathfrak{S}_i^{ot}), \rho^{OF}.(r, schd^{exe}(i))). \tag{5.16c}$$

Proof. The theorem is obviously true for the initial system states:

$$\begin{aligned}
& \mathcal{Sch}^{OF}.\mathcal{I} \Rightarrow \mathcal{Sch}^{aftm}.\mathcal{I}\\
\Leftrightarrow \quad & (* \text{ definition of } \mathcal{Sch}^{aftm}.\mathcal{I}, \text{ Definition (5.8) } *)\\
& \wedge_{i\in[1,m]} \mathcal{Sch}_i^{ot}.\mathcal{I} \Rightarrow \wedge_{i\in[1,n]} \mathfrak{S}_i^{aft}.\mathcal{I}\\
\Leftarrow \quad & (* \text{ Definitions (5.4), (5.14), (5.1), predicate calculus } *)\\
& \forall \alpha \in \Lambda(\mathcal{Sch}^{OF}.V \cup \mathcal{Sch}^{OF}.V') : (\alpha \vdash \wedge_{i\in[1,n]} \mathfrak{S}_i^{ot}.\mathcal{I})\\
& \qquad \wedge \forall w \in \textstyle\bigcup_{i\in[1,n]} (\{v_q \mid v \in I_i \wedge delay(v)\} \cup \mathfrak{S}_i^{ot}.FT) : \alpha.w = \bot \Rightarrow \alpha \vdash \wedge_{i\in[1,n]} \mathfrak{S}_i^{aft}.\mathcal{I}\\
\Leftarrow \quad & (* \text{ theorem assumption, definition of refinement, Definitions (4.1), (4.8) } *)\\
& \text{true}
\end{aligned}$$

From that implication Properties (5.16a) and (5.16b) immediately follow. Property (5.16c) follows from the definitions of *afTI*- and *otTI*-functions and from the fact that according to the above derivation chain all input and buffer variables in OSEKtime/FlexRay and AFTM are initialized with $\perp$.

Induction Step. Let us assume that the simulation relation holds in some round $r \in \mathbb{N}$ for some task $i \in [1,n]$. We show that the AFTM system after one step simulates the OSEKtime/FlexRay system after one round step again. Every following lemma has this induction assumption as a premise.

Lemma 5.1 (Local state invariance for inactive tasks in OSEKtime/FlexRay)**.** *For all wrapped OSEKtime tasks* $\mathcal{W}_q(\mathcal{S}_i^{ot})$ *with* $(i \in [1,n])$ *holds*

$$\rho^{OF}.(r, schd^{exe}(i)) + 1 \overset{WL_i}{=} \rho^{OF}.(r+1, schd^{exe}(i)),$$

where $WL_i \overset{\text{def}}{=} L_i \cup \{v_q \mid v \in I_i \wedge delay(v)\}$ *is the local state space of* $\mathcal{W}_q(\mathcal{S}_i^{ot})$.

Proof. The local state of a task in OSEKtime/FlexRay is changed only once per round according to the definition of the scheduling function in Section 5.2.2 and to the OSEKtime semantics given by (5.5a) and (5.5b) in Section 5.1.2. □

Now we prove the theorem by considering different execution phases of the AFTM and OSEKtime/FlexRay systems. Each of the following lemmas considers one part of the simulation relation to be proved.

Lemma 5.2 (State equivalence of controllable variables)**.** *For every successor state*

$$\alpha^{ot} \in \text{Succ}(\rho^{OF}.(r, schd^{exe}(i)), \delta ott_i)$$

of the OSEKtime task there exists a successor state of the corresponding AutoFocus *task STS* $\alpha^{aft} \in \text{Succ}(\rho^{aftm}.r, \delta \mathsf{aft}_i)$ *such that*

$$\alpha^{ot} \overset{L_i}{=} \alpha^{aft} \qquad \textit{and} \qquad \alpha^{ot} \overset{O_i}{=} \alpha^{aft}.$$

Proof. This is a direct consequence of the induction hypotheses (5.16a) and (5.16b) and the refinement relation between $\mathcal{S}_i^{aft}$ and $\mathcal{S}_i^{ot}$. □

Lemma 5.3 (Output equivalence)**.** *For all tasks* $\mathcal{S}_i^{aft}$ $(i \in [1,n])$*, for all their output variables* $v \in O_i$*, and for all slots* $(\overline{r}, \overline{s})$ *which lie in the interval*

$$[(r, schd^{com}(v)) + 1 : (r+1, schd^{com}(v))]$$

the following predicate holds:

$$\exists \alpha^{aft} \in \text{Succ}(\rho^{aftm}.r, \delta \mathsf{aft}_i) : \rho^{OF}.(\overline{r}, \overline{s}).v = \alpha^{aft}.v.$$

Proof. Due to the second conclusion of Lemma 5.2, and the semantics of OSEKtime and FlexRay introduced in Sections 5.1.2 and 5.1.3, respectively, we have

$$\exists \alpha^{aft} \in \text{Succ}(\rho^{aftm}.r, \delta\text{aft}_i) : \rho^{OF}.(r, schd^{exe}(i)) + 1.v = \alpha^{aft}.v.$$

Furthermore, Scheduling Constraint (5.12) ensures that the value will be transported to all nodes before the end of the round r and will not be changed until the next broadcast according to $schd^{com}(v)$ □

Lemma 5.4 (Broadcast correctness)**.** *For any pair of tasks $\mathbb{S}_i^{aft}$ and $\mathbb{S}_j^{aft}$ with $i, j \in [1,n]$ and a common variable $v \in I_i \cap O_j$ holds: The valuation of v in step $r+1$ is correctly broadcast in OSEKtime/FlexRay:*

$$\exists \alpha_i^{aft} \in \text{Succ}(\rho^{aftm}.r, \delta\text{aft}_i), \alpha_j^{aft} \in \text{Succ}(\rho^{aftm}.r, \delta\text{aft}_j) :$$
$$\alpha_i^{aft}.v = \alpha_j^{aft}.v = \begin{cases} \rho^{OF}.(r+1, schd^{exe}(i)).v_q & \text{if } delay(v), \\ \rho^{OF}.(r+1, schd^{exe}(i)).v, & o.w. \end{cases}$$

Proof. We prove the right equation first. According to the distinction above we need to consider two cases:

$\underline{\neg delay(v)}$: In the case that the message is broadcast after the start of the task $\mathbb{S}_i^{ot}$, the following obviously holds:

$$(r+1, schd^{exe}(i)) \in [(r, schd^{com}(v)) + 1 : (r+1, schd^{com}(v))].$$

Thus, according to Lemma 5.3 we obtain

$$\exists \alpha_j^{aft} \in \text{Succ}(\rho^{aftm}.r, \delta\text{aft}_j) : \alpha_j^{aft}.v = \rho^{OF}.(r+1, schd^{exe}(i)).v.$$

$\underline{delay(v)}$: In the case that the message is broadcast before the start of the corresponding task $\mathbb{S}_i^{ot}$ we have

$$(r, schd^{exe}(i)) \in [(r, schd^{com}(v)) + 1 : (r+1, schd^{com}(v))]$$

And thus:

$$\begin{aligned}
& \exists \alpha_j^{aft} \in \text{Succ}(\rho^{aftm}.r, \delta\text{aft}_j) : \alpha_j^{aft}.v = \rho^{OF}.(r+1, schd^{exe}(i)).v_q \\
\Leftrightarrow \quad & (* \text{ since } v_q \in WL_i, \text{ Lemma 5.1 } *) \\
& \exists \alpha_j^{aft} \in \text{Succ}(\rho^{aftm}.r, \delta\text{aft}_j) : \alpha_j^{aft}.v = \rho^{OF}.(r, schd^{exe}(i)) + 1.v_q \\
\Leftarrow \quad & (* \text{ definition of } \mathcal{W}(\mathbb{S}^{ot}, Q) *) \\
& \exists \alpha_j^{aft} \in \text{Succ}(\rho^{aftm}.r, \delta\text{aft}_j) : \alpha_j^{aft}.v = \rho^{OF}.(r, schd^{exe}(i)).v \\
\Leftarrow \quad & (* \text{ Lemma 5.3 } *) \\
& \exists \alpha_j^{aft} \in \text{Succ}(\rho^{aftm}.r, \delta\text{aft}_j) : \alpha_j^{aft}.v = \rho^{OF}.(r, schd^{com}(v)) + 1.v \\
\Leftarrow \quad & (* \text{ since } v \in O_j \text{ by Lemma 5.2 } *) \\
& \text{true}
\end{aligned}$$

We can move from α_j^{aft} to α_i^{aft} in both cases, using Constraint (4.2) – it states that the future input of an STS automaton cannot be constrained by its transition function and thus

$$\forall \alpha_j^{aft} \in \text{Succ}(\rho^{aftm}.r, \delta\text{aft}_j) : \exists \alpha_i^{aft} \in \text{Succ}(\rho^{aftm}.r, \delta\text{aft}_i) : \alpha_i^{aft}.v = \alpha_j^{aft}.v.$$

By this, the statement is true in both cases. □

To prove the theorem, we still need to move from a member of $\text{Succ}(\rho^{aftm}.r, \delta\text{aft}_i)$ to $\rho^{aftm}.(r+1)$. The AFTM composition is defined as the parallel one. Thus, it is easy to see that for any task $\mathcal{S}_i^{aft}$ $(i \in [1,n])$ the AFTM composition operator does not constraint any controllable variable. Formally, let $U_i \stackrel{\text{def}}{=} V_i \setminus \cup_{j\in[1,n]\setminus\{i\}} O_j$ be this set, then

$$\forall \alpha^{aft} \in \text{Succ}(\rho^{aftm}.k, \delta\text{aft}_i) : \exists \alpha^{aftm} \in \text{Succ}(\rho^{aftm}.k, \delta\text{aftm}) : \alpha^{aft} \stackrel{U_i}{=} \alpha^{aftm}. \tag{5.17}$$

The above relation combined with Lemma 5.3 proves Property (5.16b) to hold in step $r+1$. Property (5.16a) is also proved by Property (5.17) combined with the first part of Lemma 5.2 and the fact that according to the semantics of OSEKtime and FlexRay the local variables cannot be manipulated by other tasks. The unconnected input variables, which are also part of U_i, satisfy Property (5.16c) in step $r+1$, since Scheduling Constraint (5.12) and the semantics of the task wrapper guarantee that the (unconstrained) input is delivered to the OSEKtime task with the same rate as in AFTM. Finally, for connected input variables Property (5.16c) holds in $r+1$, since according to the AFTM semantics for any valuation states of AFTM tasks which coincide on homonymous variables, there always exists a valuation state in AFTM which combines the both task states. That coincidence of common variables is the consequence of Lemma 5.4. Thus, all members of $\text{Succ}(\rho^{aftm}.r, \delta\text{aftm})$ also exhibit this property. By this, Property (5.16c) holds also for connected input variables. □

Simplified versions (cf. [BBG+08]) of AFTM, OSEKtime and FlexRay were formalized using the interactive theorem prover Isabelle [NPW02] within the scope of the Verisoft Automotive project[5]. Afterwards, the proofs of the substantial parts of the above theorem were accomplished.

5.3 Related Work

In this chapter we have presented a concept for separating verification of application logic and infrastructure. The necessity of this separation is also argued for by the authors of [STY03]. They introduce a formal modeling framework and a methodology, addressing the analysis of correct deployment and timing properties. The additional value of our task concept is the explicit modeling of task dependencies and explicit statements about task activation conditions.

There also exist other approaches for the verification of distributed real-time software. [Rus99, Rus02] present a framework for a systematic formal verification of time-triggered communication. That framework allows to prove a simulation relationship between an untimed synchronous

[5] see http://www.verisoft.de and http://www4.in.tum.de/~verisoft/automotive (20.06.2008)

system, consisting of a number of communicating components (“processors”) and its implementation based on a time-triggered communication system. However, that approach considers only a one-to-one relationship between components and physical devices they run on, i.e. no OS, and no sequentialization of component execution is taken into account. For current verification issues as encountered in the automotive industry this approach is insufficient because it neglects the current praxis of automotive software development: OS, bus and application logic are developed by different suppliers and therefore should be treated separately.

5.4 Summary

In this chapter we have covered the final step in the developed process, depicted in Figure 1.1 at the beginning of this thesis: We have presented the transition from the task architecture to the deployed system and proved the correctness of this step in the sense of property preservation. This was accomplished for a time-triggered architecture as the target deployment platform and AFTM from the previous chapter as the task architecture.

Using the STS framework presented in Chapter 4 we have defined the semantics of OSEKtime operating system and FlexRay communication protocol. By integrating both STS models we were able to describe a network of computation nodes connected by the FlexRay bus with OSEKtime running on each of them. All created models constitute the programmer’s point of view on the deployment platform. The integrated model can be parameterized by a set of tasks and messages as well as by communication and computation schedules.

For the instantiation of the integrated OSEKtime/FlexRay platform by AFTM models a special schedule synthesis algorithm was presented. It allows to distribute an AFTM model over a network in a way that preserves the overall behavior of this model. The procedure demands for special buffering wrappers around the particular tasks. The wrapper generation was described also using the STS framework as a special scheduled STS parametrized with one AFTM task.

Finally, the simulation proof between AFTM and OSEKtime/FlexRay models was provided. Major parts of this proof are already formalized and proved in the automated theorem prover Isabelle [NPW02]. This is done in the Verisoft Automotive project. Other proofs carried out within the scope of this project allowed to build a model stack, reaching from the gate-level up to the CASE tool models. This stack provides a mathematical model for each level and states that every level of the system can be simulated by the underlying one. In particular, this stack proves that the programmer’s model presented here corresponds to the actual behavior of the technical infrastructure. In a broader view, this theorem stack implies that the system modeled in a CASE tool is simulated by the system running on the real hardware. Combined with the verification of the model accomplished on the level of application logic in AutoFocus and translation validation for the generated code, this yields a completely verified system.

6 Conclusions

In the preceding chapters we have studied the preservation of time and functional properties along the development process. Here, we summarize the results of this thesis and give directions for further work.

6.1 Thesis Summary

In the introduction of this thesis we set out to work towards property preservation during the development of safety-critical reactive real-time systems. Thereby our focus lay on the time behavior of these systems. In Section 1.2, we identified four interesting questions concerning its modeling and preservation, namely

(1) How can real-time requirements be analyzed respecting their consistency?

(2) How can real-time requirements be mapped to a design model and distributed over its artifacts (components, classes, *etc.*)?

(3) How can a design model be analyzed in respect of fulfillment of time requirements in the view of different time axes?

(4) How can a design model be mapped to a technical platform, without loss of functional or real-time properties of this model?

We offered answers to these questions in Chapters 2 through 5.

In Chapter 2 we introduced the semantic domain of reactive real-time systems and the specification language MTL which allows to specify real-time constraints on the systems from this domain. Using MTL we formalized a taxonomy of low-level real-time requirements commonly used to describe the behavior of reactive systems on the technical level. We also combined individual requirements to specifications by embedding them into contexts. This withdrawal of time requirements from their functional embedding established the time behavior as a special view on the system which is yet integrated with the functional one.

Using the denotational framework form Chapter 2, we presented a number of analysis and transformation methods in Chapter 3 which can work with individual time requirements from our taxonomy. The goal of Chapter 3 was to support the quality and conformance assurance measures during the system development and, in particular, to deliver answers to the questions one and two from above. For these purposes, we defined the notion of refinement. Further on, we provided rules for inference, dissipation, and distribution of requirements. Finally, we investigated the transition from unstructured set of requirements to system specification structured by components. In order to facilitate this process we formulated a special relation which holds when a component specification realizes a real-time requirement. Furthermore, in order to preserve the time behavior requirements as an independent view, we formulated a special relation between requirements. This relation allows to perform changes directly within the real-time-requirements view which provably yield a correct refinement of the whole component-based system specification.

In Chapter 4 we introduced the STS formalism, an automata-based framework for the realization of reactive systems. The systems can be specified using STSs in a modular way using an associative composition operator. A mapping to the domain of real-time systems allows to ensure the compliance of the specification by the realization and, by this delivers an answer to the third question from above. STS is designed for the formalization of design models as well as implementation models which include the behavior of technical systems. By this, reasoning about property preservation during the transition from design to implementation can be done within the same framework. We instantiated STS by the design model of the CASE tool AutoFocus.

In the same chapter we presented a reference task model, which describes and orders different kinds of information needed for the adequate description of a deployment problem and solution. We also explained how and why the actual behavior of the system is abstracted and simplified within different task models. Within the scope of the reference task model we identified the simplified model of application logic the technical systems and schedulability approaches can deal with. This aspect of the task model, referenced as the task architecture model, was instantiated by the AFTM modeling paradigm. The semantics of AFTM was formalized using the STS framework. Finally, as the last contribution of Chapter 4, the transition from AutoFocus to AFTM was described and the property preservation of this transition was proved. By this, we are able to cluster and adapt application logic modeled in AutoFocus to a deployable AFTM model without falsification of the intended behavior.

Chapter 5 contains the formal model of a time-triggered deployment platform, consisting of an operating system OSEKtime and a communication protocol FlexRay. Both technical systems as well as their combination were formalized using STSs. Further on, scheduling- and wrapper-synthesis algorithms for AFTM models were presented. Provided a deployment solution satisfies Scheduling Constraint (5.12), the deployed system preserves the behavior of the original AFTM model in the sense that there exists a simulation relation between them. This was formally proved at the end of Chapter 5. By this, Chapters 4 and 5 delivered an answer to the last question in the list from above.

To sum up our contributions, the requirements on reactive real-time systems can be modeled in the denotational semantic domain using the temporal logic developed in Chapter 2. Certain

common types of time requirements are already formalized and can be used as building blocks of specifications. Individual requirements can be integrated in formal specifications using the flexible context embedding mechanism. It allows the separation of functional and timing concerns of the system. Specifications can be formally analyzed in regard to their consistency and completeness using the methods from Chapter 3. Moreover, specifications of individual sub-systems can be derived using those methods. The presented denotational component-based specification formalism can be used as an analytical system model at different development stages of the system: the mapping of the STS formalism to the semantic domain of reactive real-time systems allows to create denotational models out of operational ones and to verify their conformance to requirements specification. In particular, this includes the possibility of conformance checks for time requirements during design phase, in which logical-time axes are used. Time requirements can be formally related to denotational component specifications, in a way which allows independent but yet integrated development of both components and requirements. These contributions constitute the time behavior as an independent but yet integrated view on the system throughout development process.

The operational models of application logic, task architecture, and time-triggered deployment platform are formally related by synthesis algorithms presented in Chapters 4 and 5. By this, the actual realization of the system can be concentrated on the level of application logic – clustering and deployment of application code can be done computer-aided in a schematic manner. These results leverage the idea of model-based development from narrow, implementation-near domains to a model stack reaching from application logic to an integrated system. Altogether, the introduced model-based formally-founded approach facilitates the construction of correct reactive real-time systems.

6.2 Outlook

Results presented in Chapters 2 through 5 serve as the foundation for future work of both practical and theoretical nature. For the ultimate goal of establishing the model-based development of reactive real-time systems especially in the early phases the following points are of particular interest.

Modeling Scheme for Time Requirements

In Chapter 2 we presented the semantic framework for the specification of time requirements. Its purpose is to provide the foundation of a specification language. Besides the temporal logic presented in that chapter, there exist a number of well-established graphical notation schemes like *use case diagrams* or *message sequence charts* (*MSC*) which are suited for the capture of requirements. Their foundation using the developed stream-based semantics as well as their extension by timing annotations are promising investigation fields on the way to the tool support for requirements engineering which is described below.

A complementary work direction is the adaptation of one of the various structured-English approaches for the settings of our semantics, MTL syntax, and requirements taxonomy. These approaches (e.g., [SACO02, Fuc00]) aim at generating explanations to the formulas using a fixed English vocabulary subset and a number of sentence-structure rules.

Tool Support

Automation and computer-support of different tasks in software engineering are as important as their foundation by formal methods. The combination of these two factors is the best way to produce correct systems, fulfilling the customer's needs, at affordable cost. AutoFocus tool is used to emulate AFTM models, as shown in [BGH+06]; however, there is still no support for the transformation and clustering of AutoFocus models by AFTM models. The affiliate transition towards the OSEKtime/FlexRay architecture is partially supported by the code generator developed within the scope of the Verisoft Automotive project. However, computation and communication schedules are still need to be created and checked for validity by hand.

The connection to available schedulability tools, like TIMES tool [AFM+03] or SymTA/S [HJRE06], is a further important issue. This is also an important ingredient of the extension of the presented deployment approach towards the support of event-triggered platforms which is discussed below.

The automation and tool support for requirements engineering provided by commercially available tools are at their best restricted to the structuring of text [Bro06b]. The modeling approach from Chapter 2 facilitates the documentation, analysis and validation of specifications based on formal models and assisted by computer. For these purposes the following problems have to be treated

(1) the existing CASE tools have to be extended by new modeling techniques, e.g., by the aforementioned annotated MSCs or use cases,

(2) the existing system views like AutoFocus's STDs have to be linked to or annotated by the relevant time requirements, and

(3) the verification backends, e.g., model checker or theorem prover, have to support the reasoning within the semantic domain of reactive real-time systems from Section 2.2.3.

Further Deployment Platforms

So far, we were dealing with time-triggered technical architectures only. The next reasonable step is to consider event-triggered deployment platforms, like OSEK OS [OSE01a] as an operating system or CAN bus [ISO94] as a communication protocol. The behavior of event-triggered systems is less predictable respective time behavior; however, they often permit more efficient solutions and, thus, event-triggered systems are often more preferable for applications where cost concerns and legacy integration issues are more dominant compared to safety requirements. Thus, event-triggered systems still play a prominent role for the bulk good manufactures, like the automotive industry.

In the case of event-triggered systems the correctness of deployment solution can be either

- guaranteed under certain assumptions, e.g., about the maximal latency of messages on the bus,
- guaranteed with a certain probability, e.g., under the assumption that a certain percentage of messages is transmitted on the bus correctly and in time, or
- enforced by emulating a more predictable communication protocol, e.g., cf. [RB04, BCG+02].

Another interesting challenge is to model heterogeneous systems, consisting of time- and event-triggered parts. This situation is typical, e.g., for on-board networks in vehicles [SZ06]: Networks embedded in premium cars consist of up to 80 ECUs (electronic control units) running different (time- and event-triggered) operating systems and connected by different types of transmission links, like event-triggered CAN bus or time-triggered FlexRay. Apart from the inherent complexity of real-world systems the problem of modeling and analyzing heterogeneous systems is also interesting from the theoretical point of view [HS07].

Finally, more elaborate deployment and clustering synthesis procedures would offer even more flexibility during these development steps. In particular, the abstract component interface which describes a valid AutoFocus component abstraction can be refined by dynamical aspects, e.g., by causal dependencies between the input/output presence combinations. This would allow to establish certain communication protocols between AutoFocus tasks, like request-response, or client-server. The deployment model can approach the reality by allowing task run-times of more than one slot, or by modeling the data-loss during the inter-node communication as well as the fault-tolerance mechanisms of the middle-ware which may cause additional unforeseen delays in the communication.

Case Studies

Extended case studies of substantial size promise to give insight about the scalability and efficiency of the presented techniques. In particular, the quantification of effort put into the creation of formal requirements model must be compared with the actual quality improvement of the resulting system and with the overall effort for ensuring the correctness.

Another potential field of investigation is the estimation of performance penalties entailed by generic schedule and code generation schemes. The defensibleness of these penalties decides about the applicability of the presented approach in practice.

A Auxiliary Definitions and Proofs

In Chapters 2 and 4 we have omitted some proofs and definitions for better readability of the "main" text. In the following sections we make up for this omission.

A.1 Definitions and Proofs for Chapter 2

Definition A.1 (Partial order/partially ordered set (poset))**.** *For a set S and a binary relation $\preceq\colon S \times S$ written in infix notation, $(S, \preceq)$ is a* partial order*, iff for all $a, b, c \in S$, we have that:*

(1) $a \preceq a$, *(reflexivity)*

(2) *if* $a \preceq b$ *and* $b \preceq a$ *then* $a = b$, *(antisymmetry)*

(3) *if* $a \preceq b$ *and* $b \preceq c$ *then* $a \preceq c$. *(transitivity)*

A set with a partial order is called a partially ordered set *(also called a* poset*).* ◇

Definition A.2 (Total/linear order)**.** *A partial order $(S, \preceq)$ is called* total *or* linear order*, iff for all $a, b \in S$ holds:*

$$a \preceq b \quad or \quad b \preceq a \qquad \text{(totality)}.$$ ◇

Definition A.3 (Infimum/greatest lower bound (glb))**.** *For a partial order $(S, \preceq)$ and a subset $T \subseteq S$, $s \in S$ is called a* lower bound *of T iff*

$$\forall x \in T : s \preceq x.$$

s is called a greatest *lower bound, written $s \stackrel{\text{def}}{=} \bigsqcap T$ or $s \stackrel{\text{def}}{=} \operatorname{glb} T$ iff*

(1) s is a lower bound of T, and
(2) for all lower bounds $\overline{s}$ of T, $\overline{s} \preceq s$. ◇

Definition A.4 (Supremum/least upper bound (lub))**.** *For a partial order* $(S, \preceq)$ *and a subset* $T \subseteq S$*,* $s \in S_\infty$ *is called an* upper bound *of* T *iff*

$$\forall x \in T : x \preceq s.$$

s *is called a* least *upper bound, written* $s \stackrel{\text{def}}{=} \bigsqcup T$ *or* $s \stackrel{\text{def}}{=} \operatorname{lub} T$ *iff*

(1) s *is an upper bound of* T*, and*
(2) for all upper bounds $\overline{s}$ *of* T*,* $s \preceq \overline{s}$*.* ◇

Definition A.5 (Operations on timed channel valuations)**.** *Given simultaneous channel valuations* $\tau, \tau', \tau'' \in \vec{C}$ *of an arbitrary non-empty channel set* C*, a timed event* $a \in \vec{C}^{\mathcal{A}}$*, an interval* $\mathbb{I} \in [\mathbb{T}_\infty]$ *from the timed domain* $\mathbb{T}$*, and a natural number* $i \in \mathbb{N}$*. We lift the definitions of operations introduced in Section 2.2.2 for timed channel valuations as follows:*

$$\begin{aligned}
\#\tau &\stackrel{\text{def}}{=} \#\phi(\tau), \\
\tau' = \tau|_{\mathbb{I}} &\stackrel{\text{def}}{\Leftrightarrow} \forall c \in C : \tau'.c = \tau.c|_{\mathbb{I}}, \\
len(\tau) &\stackrel{\text{def}}{=} \max_{c \in C} len(\tau.c), \\
nth(\tau, i) = a &\stackrel{\text{def}}{\Leftrightarrow} \quad (\forall c \in C : \tau.(\phi(a)).c = a.c) \\
&\qquad \wedge \big(\phi(a) = \min_{c \in C}\{\phi(nth(\tau.c, i)) \mid i \leq len(\tau.c)\} \quad \text{if } i \leq len(\tau), \\
\tau'' = \tau \frown \tau' &\stackrel{\text{def}}{\Leftrightarrow} \forall c \in C : \tau''.c = \tau.c \frown \tau'.c.
\end{aligned}$$

◇

Proposition A.1 (Correctness of equivalencies in Table 2.4)**.** *For a pair of MTL formulas* φ, ψ*, observation predicates* $e, e_1, e_2 \in \wp(\vec{C})$*, and a pair of intervals* $\mathbb{I}, \mathbb{J} \in [\mathbb{T}_\infty]$ *hold Properties (2.1a) through (2.1h) listed on page 30 in Table 2.4.*

Proof. For all $\tau \in \vec{C}$ and $t \in \phi(\tau)$ hold:

- Property (2.1a) holds due to the following derivation:

$$\begin{aligned}
& (\tau, t) \models \Diamond_{\mathbb{I}}(\varphi \vee \psi) \\
\Leftrightarrow \quad & (* \text{ definitions of } \Diamond_{(.)} \text{ and } \vee \ *) \\
& (\tau, t) \models \mathrm{tt}\, \mathrm{U}_{\mathbb{I}}\, \neg(\neg\varphi \wedge \neg\psi) \\
\Leftrightarrow \quad & (* \text{ semantics of } \mathrm{U}_{(.)}, \neg, \text{ and } \wedge \ *) \\
& \exists t' \in \phi(t) : \big((\tau, t') \models \varphi \vee (\tau, t') \models \psi\big) \wedge t' - t \in \mathbb{I} \\
\Leftrightarrow \quad & (* \text{ predicate calculus } *) \\
& \big(\exists t' \in \phi(t) : (\tau, t') \models \varphi \wedge t' - t \in \mathbb{I}\big) \vee \big(\exists t' \in \phi(t) : (\tau, t') \models \psi \wedge t' - t \in \mathbb{I}\big) \\
\Leftrightarrow \quad & (* \text{ semantics of } \mathrm{U}_{(.)}, \text{ definition of } \Diamond_{(.)} \ *) \\
& (\tau, t') \models \Diamond_{\mathbb{I}}\varphi \vee (\tau, t') \models \Diamond_{\mathbb{I}}\psi \\
\Leftrightarrow \quad & (* \text{ definition of } \vee, \text{ semantics of } \neg, \wedge, \text{ predicate calculus } *) \\
& (\tau, t') \models \Diamond_{\mathbb{I}}\varphi \vee \Diamond_{\mathbb{I}}\psi
\end{aligned}$$

- Property (2.1b) holds due to the following derivation:

$$
\begin{aligned}
& (\tau, t) \models \Box_{\mathbb{I}}(\varphi \wedge \psi) \\
\Leftrightarrow \quad & (* \text{ definition of } \Box_{(.)}, \text{ predicate calculus } *) \\
& (\tau, t) \models \neg \Diamond_{\mathbb{I}}(\neg\varphi \vee \neg\psi) \\
\Leftrightarrow \quad & (* \text{ Property (2.1a) } *) \\
& (\tau, t) \models \neg(\Diamond_{\mathbb{I}}\varphi \vee \Diamond_{\mathbb{I}}\psi) \\
\Leftrightarrow \quad & (* \text{ definitions of } \Box_{(.)}, \wedge, \text{ predicate calculus } *) \\
& (\tau, t) \models \Box_{\mathbb{I}}\varphi \wedge \Box_{\mathbb{I}}\psi
\end{aligned}
$$

- Provided $\mathbb{I} \cup \mathbb{J}$ is a valid interval, then we prove Property (2.1c) as follows:

$$
\begin{aligned}
& (\tau, t) \models \Diamond_{\mathbb{I}\cup\mathbb{J}}(\varphi) \\
\Leftrightarrow \quad & (* \text{ definition of } \Diamond_{(.)} *) \\
& (\tau, t) \models \mathrm{tt}\, \mathrm{U}_{\mathbb{I}\cup\mathbb{J}}\, \varphi \\
\Leftrightarrow \quad & (* \text{ semantics of } \mathrm{U}_{(.)} *) \\
& \exists t' \in \phi(t) : (\tau, t') \models \varphi \wedge t' - t \in \mathbb{I} \cup \mathbb{J} \\
\Leftrightarrow \quad & (* \text{ set theory, predicate calculus } *) \\
& (\exists t' \in \phi(t) : (\tau, t') \models \varphi \wedge t' - t \in \mathbb{I}) \vee (\exists t' \in \phi(t) : (\tau, t') \models \varphi \wedge t' - t \in \mathbb{J}) \\
\Leftrightarrow \quad & (* \text{ semantics of } \mathrm{U}_{(.)}, \text{ definition of } \Diamond_{(.)} *) \\
& (\tau, t') \models \Diamond_{\mathbb{I}}\varphi \vee (\tau, t') \models \Diamond_{\mathbb{J}}\varphi \\
\Leftrightarrow \quad & (* \text{ definition of } \vee, \text{ semantics of } \neg, \wedge, \text{ predicate calculus } *) \\
& (\tau, t') \models \Diamond_{\mathbb{I}}\varphi \vee \Diamond_{\mathbb{J}}\varphi
\end{aligned}
$$

- Provided $\mathbb{I} \cup \mathbb{J}$ is a valid interval, then we prove Property (2.1d) as follows:

$$
\begin{aligned}
& (\tau, t) \models \Box_{\mathbb{I}\cup\mathbb{J}}(\varphi) \\
\Leftrightarrow \quad & (* \text{ definition of } \Box_{(.)} *) \\
& (\tau, t) \models \neg \Diamond_{\mathbb{I}\cup\mathbb{J}} \neg\varphi \\
\Leftrightarrow \quad & (* \text{ Property (2.1c) } *) \\
& (\tau, t) \models \neg(\Diamond_{\mathbb{I}} \neg\varphi \vee \Diamond_{\mathbb{J}} \neg\varphi) \\
\Leftrightarrow \quad & (* \text{ definitions of } \Box_{(.)}, \wedge, \text{ predicate calculus } *) \\
& (\tau, t) \models \Box_{\mathbb{I}}\varphi \wedge \Box_{\mathbb{J}}\varphi
\end{aligned}
$$

- Property (2.1e) which states that $\neg\Box_{\mathbb{I}}\varphi = \Diamond_{\mathbb{I}}\neg\varphi$ holds follows directly from the definition of the $\Box_{(.)}$-modality.

- Property (2.1f) holds due to the following derivation:

$$
\begin{aligned}
& (\tau,t) \models (\neg\psi \,\mathrm{U}_{\mathbb{I}}\, (\neg\varphi \wedge \neg\psi)) \vee \Box_{\mathbb{I}}\, \neg\psi \\
\Leftrightarrow \quad & (* \text{ semantics of } \mathrm{U}_{(.)},\ \neg,\ \text{definitions of } \Box_{(.)},\ \Diamond_{(.)}\ *) \\
& (\exists t' \in \phi(t) : (\tau,t') \models (\neg\varphi \wedge \neg\psi) \wedge t' - t \in \mathbb{I} \wedge \forall t'' \in t + \mathbb{I} : t'' < t' \Rightarrow (\tau,t'') \not\models \psi) \\
& \vee \neg(\exists t' \in \phi(t) : (\tau,t') \models \psi \wedge t' - t \in \mathbb{I}) \\
\Leftrightarrow \quad & (* \text{ predicate calculus } *) \\
& \neg\Big((\forall t' \in \phi(t) : (\tau,t') \models (\varphi \vee \psi) \vee t' - t \notin \mathbb{I} \vee \exists t'' \in t + \mathbb{I} : t'' < t' \Rightarrow (\tau,t'') \models \psi) \\
& \quad \wedge (\exists t' \in \phi(t) : (\tau,t') \models \psi \wedge t' - t \in \mathbb{I})\Big) \\
\Leftrightarrow \quad & (* \text{ renaming } \forall\text{-quantified } t' \text{ to } t''',\ \text{predicate calculus } *) \\
& \neg\Big(\exists t' \in \phi(t) : (\tau,t') \models \psi \wedge t' - t \in \mathbb{I} \\
& \quad \wedge (\forall t''' \in \phi(t) : (\tau,t''') \models (\varphi \vee \psi) \vee t''' - t \notin \mathbb{I} \vee \exists t'' \in t + \mathbb{I} : t'' < t''' \Rightarrow (\tau,t'') \models \psi)\Big) \\
\Leftrightarrow \quad & (* \text{ predicate calculus, interval arithmetic } *) \\
& \neg\Big(\exists t' \in \phi(t) : (\tau,t') \models \psi \wedge t' - t \in \mathbb{I} \\
& \quad \wedge (\forall t''' \in t + \mathbb{I} : (\tau,t''') \models (\varphi \vee \psi) \vee \exists t'' \in t + \mathbb{I} : t'' < t''' \Rightarrow (\tau,t'') \models \psi)\Big)
\end{aligned}
$$

  The last conjunct follows from the first one when $t''' = t'$ or $t''' > t'$. Further on, if for a given t' with $(\tau,t') \models \psi \wedge t' - t \in \mathbb{I}$ there exists some $\bar{t} \in t + \mathbb{I}$ such that $\bar{t} < t'$ and $(\tau,\bar{t}) \models \psi$ then for all $t''' \geq \bar{t}$ the last conjunct yields true, too. Thus, we can simplify the above logical expression to:

$$
\begin{aligned}
& \neg(\exists t' \in \phi(t) : (\tau,t') \models \psi \wedge t' - t \in \mathbb{I} \wedge \forall t''' \in t + \mathbb{I} : t''' < t' \Rightarrow (\tau,t''') \models \varphi) \\
\Leftrightarrow \quad & (* \text{ semantics of } \mathrm{U}_{(.)},\ \neg\ *) \\
& (\tau,t) \models \neg(\varphi \,\mathrm{U}_{\mathbb{I}}\, \psi)
\end{aligned}
$$

- Property (2.1g) holds since for any $\updownarrow \in \{\downarrow,\uparrow\}$:

$$
\begin{aligned}
& (\tau,t) \models (e_1 \cap e_2)^{\updownarrow} \\
\Leftrightarrow \quad & (* \text{ semantics of } \downarrow/\uparrow\ *) \\
& \tau\updownarrow_t \in (e_1 \cap e_2) \\
\Leftrightarrow \quad & (* \text{ set theory, predicate calculus } *) \\
& (\tau\updownarrow_t \in e_1) \wedge (\tau\updownarrow_t \in e_2) \\
\Leftrightarrow \quad & (* \text{ semantics of } \wedge,\ \downarrow/\uparrow\ *) \\
& (\tau,t) \models (e_1^{\updownarrow} \wedge e_2^{\updownarrow})
\end{aligned}
$$

- Property (2.1h) holds since for any $\updownarrow \in \{\downarrow,\uparrow\}$:

$$
\begin{aligned}
& (\tau,t) \models (\neg e)^{\updownarrow} \\
\Leftrightarrow \quad & (* \text{ semantics of } \downarrow/\uparrow,\ \neg\ *) \\
& \neg(\tau\updownarrow_t \in e) \\
\Leftrightarrow \quad & (* \text{ semantics of } \downarrow/\uparrow,\ \neg\ *) \\
& (\tau,t) \models \neg(e^{\updownarrow})
\end{aligned}
$$

□

A.2 Definitions and Proofs for Chapter 4

Definition A.6 (Simulation on STSs)**.** *For two STS automata* $\mathcal{S}_1 = (I_1, L_1, O_1, \mathcal{I}_1, \delta_1)$ *and* $\mathcal{S}_2 = (I_2, L_2, O_2, \mathcal{I}_2, \delta_2)$ *with* $V_1 \supseteq V_2$*, we say that a binary relation* $H \subseteq \Lambda(V_1) \times \Lambda(V_2)$ *is a* simulation relation *[Mil71] between* $\mathcal{S}_1$ *and* $\mathcal{S}_2$ *iff for all* $\alpha_1 \in \Lambda(V_1)$ *and* $\alpha_2 \in \Lambda(V_2)$ *if* $(\alpha_1, \alpha_2) \in H$ *then the following conditions hold.*

(1) $\alpha_1 \overset{V_2}{=} \alpha_2$.

(2) $\forall \beta_1 \in \mathrm{Succ}(\alpha_1, \delta_1) : \exists \beta_2 \in \mathrm{Succ}_2(\alpha_2, \delta_2) : (\beta_1, \beta_2) \in H$.

H is a weak simulation relation *if we replace the second condition from above by the following.*

$$\forall n \in \mathbb{N} : \forall \beta_1 \in \mathrm{Succ}^n(\alpha_1, \delta_1) : \exists k \in \mathbb{N} : \exists \beta_2 \in \mathrm{Succ}^k(\alpha_2, \delta_2) : (\beta_1, \beta_2) \in H.$$

$\mathcal{S}_2$ simulates $\mathcal{S}_1$ *if there exists a simulation relation H such that for every initial state* $\alpha_1 \vdash \mathcal{I}_1$ *there is an initial state* $\alpha_2 \vdash \mathcal{I}_2$ *for which* $(\alpha_1, \alpha_2) \in H$. ◇

Bibliography

[ABG95] Pascalin Amagbégnon, Loïc Besnard, and Paul Le Guernic. Implementation of the data-flow synchronous language signal. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 163–173, New York, NY, USA, 1995. ACM.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[AFM+03] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Times: a tool for schedulability analysis and code generation of real-time systems. In *Proceedings of the 1st International Workshop on Formal Modeling and Analysis of Timed Systems, FORMATS 2003*, 2003.

[AH92] Rajeev Alur and Thomas A. Henzinger. Logics and models of real time: A survey. In *Proceedings of the Real-Time: Theory in Practice, REX Workshop*, pages 74–106, London, UK, 1992. Springer-Verlag.

[All83] James F. Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843, 1983.

[Alu92] Rajeev Alur. *Techniques for automatic verification of real-time systems*. PhD thesis, Department of Computer Science, Stanford University, Stanford, CA, USA, 1992.

[BBC+03] P. Braun, M. Broy, M. V. Cengarle, J. Philipps, W. Prenninger, A. Pretschner, M. Rappl, and R. Sandner. *The automotive CASE*, pages 211 – 228. Wiley, 2003.

[BBG+08] Jewgenij Botaschanjan, Manfred Broy, Alexander Gruler, Alexander Harhurin, Steffen Knapp, Leonid Kof, Wolfgang Paul, and Maria Spichkova. On the correctness of upper layers of automotive systems. *Formal Aspects of Computing (FACS)*, 20(6):561–662, December 2008.

[BBR+05] Andreas Bauer, Manfred Broy, Jan Romberg, Bernhard Schätz, Peter Braun, Ulrich Freund, Núria Mata, Robert Sandner, and Dirk Ziegenbein. AutoMoDe — notations, methods, and tools for model-based development of automotive software. In *Proceedings of the SAE 2005 World Congress*, Detroit, MI, April 2005. Society of Automotive Engineers.

[BCG99] A. Benveniste, B. Caillaud, and P. Le Guernic. From synchrony to asynchrony. In J.C.M. Baeten and S. Mauw, editors, *Proceedings of CONCUR'99, Concurrency Theory, 10th International Conference*, volume 1664 of *LNCS*, pages 162–177. Springer, 1999.

[BCG+02] A. Benveniste, P. Caspi, P. Le Guernic, H. Marchand, J.-P. Talpin, and S. Tripakis. A protocol for loosely time-triggered architectures. In *Proceedings of EMSOFT 2001*. Springer-Verlag, 2002.

[BFG+08] Manfred Broy, Martin Feilkas, Johannes Grünbauer, Alexander Gruler, Alexander Harhurin, Judith Hartmann, Birgit Penzenstadler, Bernhard Schätz, and Doris Wild. Umfassendes architekturmodell für das engineering eingebetteter software-intensiver systeme. Technical Report TUM-I0816, Technische Universität München, 06 2008.

[BGH+06] Jewgenij Botaschanjan, Alexander Gruler, Alexander Harhurin, Leonid Kof, Maria Spichkova, and David Trachtenherz. Towards Modularized Verification of Distributed Time-Triggered Systems. In *Formal Methods 2006*, McMaster University, Hamilton ON, Canada, August 23 - 25, 2006.

[BHL+02] A. Blotz, F. Huber, H. Lötzbeyer, A. Pretschner, O. Slotosch, and H.-P. Zängerl. Model-based software engineering and ada: Synergy for the development of safety-critical systems. In *Proc. Ada Deutschland Tagung*, Jena, 2002.

[BJ05] Jewgenij Botaschanjan and Jan Jürjens. MoDeII: Modeling and Analyzing Time-Constraints. In *Proceedings of the 12th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems, ECBS*. IEEE, 2005.

[BK98] Manfred Broy and Ingolf Krüger. Interaction Interfaces – Towards a scientific foundation of a methodological usage of Message Sequence Charts. In J. Staples, M. G. Hinchey, Shaoying Liu Hinchey, and Shaoying Liu, editors, *Formal Engineering Methods (ICFEM'98)*, pages 2 – 15. IEEE Computer Society, 1998.

[BKKS05] J. Botaschanjan, L. Kof, Ch. Kühnel, and M. Spichkova. Towards Verified Automotive Software. In *ICSE, SEAS Workshop*, St. Louis, Missouri, USA, May 21 2005.

[BMN00] P. Bellini, R. Mattolini, and P. Nesi. Temporal logics for real-time system specification. *ACM Comput. Surv.*, 32(1):12–42, 2000.

[Boe81] Barry W. Boehm. *Software Engineering Economics*. Prentice-Hall, Upper Saddle River, NJ, USA, 1981.

[BP99] M. Breitling and J. Philipps. Black box views of state machines. Technical Report TUM-I9916, Institut für Informatik, Technische Universität München, 1999.

[Bro94] Manfred Broy. A Functional Rephrasing of the Assumption/Commitment Specification Style. Technical Report TUM-I9417, Technische Universität München, 1994.

[Bro97] M. Broy. Refinement of Time. In M. Bertran and Th. Rus, editors, *Transformation-Based Reactive System Development, ARTS'97*, number 1231 in LNCS, pages 44 – 63. TCS, 1997.

[Bro98] Manfred Broy. Compositional Refinement of Interactive Systems Modelled by Relations. In *International Symposium Compositionality*, number 1536 in LNCS, pages 130 – 149. Springer, 1998.

[Bro06a] Manfred Broy. Challenges in automotive software engineering. In Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa, editors, *ICSE*, pages 33–42. ACM, 2006.

[Bro06b] Manfred Broy. Requirements engineering as a key to holistic software quality. In *Computer and Information Sciences ISCIS 2006*, volume Volume 4263/2006, pages 24–34. Springer Berlin / Heidelberg, 2006.

[BRS04] J. Botaschanjan, J. Romberg, and O. Slotosch. *Formal Methods and Models for System Design – A System Level Perspective*, chapter MoDe: A Method for System-Level Architecture Evaluation. Kluwer Academic Publishers, 2004.

[BS01a] M. Broy and K. Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer, 2001.

[BS01b] Manfred Broy and Oscar Slotosch. From requirements to validated embedded systems. In *EMSOFT '01: Proceedings of the First International Workshop on Embedded Software*, pages 51–65, London, UK, 2001. Springer-Verlag.

[BW95] A. Burns and A. Wellings. *HRT-HOOD: A Structured Design Method for Hard Real-Time Ada Systems*. Elsevier Science, 1995.

[Car02] Jan Carlson. Languages and methods for specifying real-time systems. Technical report, Mälardalen Real-Time Research Centre, Department of Computer Science and Engineering, Mälardalen University, Sweden, August 2002.

[CES83] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, pages 117–126, Austin, Texas, January 24–26, 1983. ACM SIGACT-SIGPLAN.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[Cha88] K. Mani Chandy. *Parallel program design: a foundation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[Con03] FlexRay Consortium. FlexRay Overview. http://www.flexray.com/products/protocol%20overview.pdf, 2003. accessed 15.12.2006.

[DAC98] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property specification patterns for finite-state verification. In *FMSP '98: Proceedings of the second workshop on Formal methods in software practice*, pages 7–15, New York, NY, USA, 1998. ACM.

[Dam05] Werner Damm. Controlling speculative design processes using rich component models. In *ACSD '05: Proceedings of the Fifth International Conference on Application of Concurrency to System Design*, pages 118–119, Washington, DC, USA, 2005. IEEE Computer Society.

[EMSS92] E. A. Emerson, A. K. Mok, A. P. Sistla, and J. Srinivasan. Quantitative temporal reasoning. *Real-Time Syst.*, 4(4):331–352, 1992.

[Eur03] European Commission (DG Enterprise and DG Information Society). eSafety forum: Summary report 2003. Technical report, eSafety, March 2003.

[Fer03] E. Fersman. *A generic approach to schedulability analysis of real-time systems*. PhD thesis, Uppsala University, 2003.

[FHL+01] Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm. Reliable and precise WCET determination for a real-life processor. In *EMSOFT '01: Proceedings of the First International Workshop on Embedded Software*, pages 469–485, London, UK, 2001. Springer-Verlag.

[FKPY07] Elena Fersman, Pavel Krcal, Paul Pettersson, and Wang Yi. Task automata: Schedulability, decidability and undecidability. *Information and Computation*, Volume 205(8):1149–1172, August 2007.

[Fle83] S. B. Flexner, editor. *Random House Webster's Unabridged Dictionary*. Random House, Inc., second edition, 1983.

[Fle04] FlexRay Consortium. *FlexRay Communication System - Protocol Specification - Version 2.0*, 2004.

[FMHH01] T. Führer, B. Müller, F. Hartwich, and R. Hugel. Time triggered CAN (TTCAN). In *SAE 2001, Detroit*, number 2001-01-0073 in SAE, 2001.

[Fuc00] Norbert E. Fuchs. Attempto controlled english. In *WLP*, pages 211–218, 2000.

[GEB03] Jinghong Guo, Stephen H. Edwards, and Dusan Borojevich. Comparison of dataflow architecture and Real-Time Workshop Embedded Coder in power electronics system control software design. In *CPES 2003 Power Electronics Seminar and NSF/Industry Annual Review*, April 2003.

[GGJZ00] Carl A. Gunter, Elsa L. Gunter, Michael Jackson, and Pamela Zave. A reference model for requirements and specifications. *IEEE Software*, 17(3):37–43, 2000.

[Gir05] A. Girault. A survey of automatic distribution method for synchronous programs. In F. Maraninchi, M. Pouzet, and V. Roy, editors, *International Workshop on Synchronous Languages, Applications and Programs, SLAP'05*, ENTCS, Edinburgh, UK, April 2005. Elsevier Science.

[Gli07] M. Glinz. On non-functional requirements. *Requirements Engineering Conference, 2007. RE '07. 15th IEEE International*, pages 21–26, 15-19 Oct. 2007.

[Gri03] Klaus Grimm. Software technology in an automotive company: major challenges. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 498–503, Washington, DC, USA, 2003. IEEE Computer Society.

[GTTW02] Erich Grädel, Wolfgang Thomas, and Thomas Wilke, editors. *Automata, Logics and Infinite Games*, volume 2500 of *Lecture Notes in Computer Science*. Springer, 2002.

[HBGL95] C. Heitmeyer, A. Bull, C. Gasarch, and B. Labaw. Scr: a toolset for specifying and analyzing requirements. *Computer Assurance, 1995. COMPASS '95. 'Systems Integrity, Software Safety and Process Security'. Proceedings of the Tenth Annual Conference on*, pages 109–122, Jun 1995.

[HD98] Patrick Heymans and Eric Dubois. Scenario-based techniques for supporting the elaboration and the validation of formal requirements. Technical report, Technical Report of the University of Namur, October 1998.

[Hei95] C. Heitmeyer. Requirements specifications for hybrid systems. In *Proc. Workshop on Verification and Control of Hybrid Systems*, pages 304–314, 1995.

[HHK03] Th. A. Henzinger, Ben. Horowitz, and Ch. M. Kirsch. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91:84–99, 2003.

[HJRE06] Arne Hamann, Marek Jersak, Kai Richter, and Rolf Ernst. A framework for modular analysis and exploration of heterogeneous embedded systems. *Real-Time Systems*, 2006.

[HLN+90] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. STATEMATE: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.

[HMR+98] F. Huber, S. Molterer, A. Rausch, B. Schätz, M. Sihling, and O. Slotosch. Tool supported specification and simulation of distributed systems. In B. Krämer, N. Uchihira, P. Croll, and S. Russo, editors, *Proc. Intl. Symp. on Software Engineering for Parallel and Distributed Systems*, pages 155–164. IEEE, 1998.

[Hon98] Honeywell Technology Center. *MetaH User's Manual*, 1998.

[HRR91] N. Halbwachs, P. Raymond, and C. Ratel. Generating Efficient Code from Data-Flow Programs. In J. Maluszyński and M. Wirsing, editors, *Proceedings of the Third International*

Symposium on Programming Language Implementation and Logic Programming, number 528 in LNCS, pages 1–13207–218. Springer Verlag, 1991.

[HS07] Thomas A. Henzinger and Joseph Sifakis. The discipline of embedded systems design. *Computer*, 40(10):32–40, 2007.

[HSE97] F. Huber, B. Schätz, and G. Einert. Consistent Graphical Specification of Distributed Systems. In *Industrial Applications and Strengthened Foundations of Formal Methods (FME'97)*, LNCS 1313, pages 122–141. Springer Verlag, 1997.

[HSF+04] H. Heinecke, K.-P. Schnelle, H. Fennel, J. Bortolazzi, L. Lundh, J. Leflour, J.-L. Maté, K. Nishikawa, and T. Scharnhorst. AUTomotive Open System ARchitecture – an industry-wide initiative to manage the complexity of emerging automotive E/E-architectures. In *Proceed. Convergence 2004, International Congress on Transportation Electronics, Detroit*, October 2004.

[IM07] James Ivers and Gabriel A. Moreno. Model-driven development with predictable quality. In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications companion*, pages 874–875, New York, NY, USA, 2007. ACM.

[ISO94] ISO: Int'l Organization for Standardization, Geneva. *ISO 11898, Road Vehicles: Interchange of Digital Information-Controller Area Network (CAN) for High-speed Communication*, 1994.

[JLHM91] Matthew S. Jaffe, Nancy G. Leveson, Mats Per Erik Heimdahl, and Bonnie E. Melhart. Software requirements analysis for real-time process-control systems. *IEEE Trans. Software Eng.*, 1991.

[JMS88] Farnam Jahanian, Aloysius K. Mok, and Douglas A. Stuart. Formal specification of real-time systems. Technical report, University of Texas at Austin, Austin, TX, USA, 1988.

[Jon93] M. P. Jones. *An Introduction to Gofer*, August 1993.

[Jos01] Mathai Joseph, editor. *Real-time Systems: Specification, Verification and Analysis*. Prentice Hall, revised version with corrections edition, June 2001.

[KC05] Sascha Konrad and Betty H. C. Cheng. Real-time specification patterns. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 372–381, New York, NY, USA, 2005. ACM.

[KG94] Hermann Kopetz and Günter Grünsteidl. TTP — a protocol for fault-tolerant real-time systems. *Computer*, 27(1):14–23, 1994.

[KLSV06] D.K. Kaynar, N.A. Lynch, R. Segala, and F.W. Vaandrager. *The Theory of Timed I/O Automata*. Morgan & Claypool Publishers, 2006. Synthesis Lecture on Computer Science, 101pp, ISBN 159829010X.

[Koy90] Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Syst.*, 2(4):255–299, 1990.

[KR02] Kowalewski and Rittel. Real-time service allocation for car periphery supervision. Technical report, Robert Bosch GmbH. Research and Development – FV/SLD, 2002.

[Krü00] I. Krüger. *Distributed System Design with Message Sequence Charts*. PhD thesis, Technische Universität München, 2000.

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. on Software Engineering*, 2:125–143, March 1977.

[Lam83] L. Lamport. What good is temporal logic. In R. E. A. Mason, editor, *Information Processing 83: Proceedings of the IFIP 9th World Congress*, pages 657–668, 1983.

[Lam94] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[Lan97] G. Le Lann. An analysis of the ariane 5 flight 501 failure-a system engineering perspective. *ecbs*, 00:339, 1997.

[LH95] Yassine Lakhneche and Jozef Hooman. Metric temporal logic with durations. *Theor. Comput. Sci.*, 138(1):169–199, 1995.

[Liu00] Jane W. S. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[LL73] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1), 1973.

[LT89] N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, 9 1989.

[LvL02] Emmanuel Letier and Axel van Lamsweerde. Deriving operational software specifications from system goals. *SIGSOFT Softw. Eng. Notes*, 27(6):119–128, 2002.

[MA95] S.M. Mahmud and A.I. Alrabady. A new decision making algorithm for airbag control. *Vehicular Technology, IEEE Transactions on*, 44(3):690–697, Aug 1995.

[MC81] J. Misra and K.M. Chandy. Proofs of networks of processes. *Software Engineering, IEEE Transactions on*, SE-7(4):417–426, July 1981.

[Mil71] Robin Milner. An algebraic definition of simulation between programs. Technical report, Stanford University, Stanford, CA, USA, 1971.

[MP92] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer, New York, 1992.

[MP95] Zohar Manna and Amir Pnueli. Models for reactivity. *Acta Informatica*, 30(7):609–678, 1995.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[OSE01a] OSEK VDX consortium. *OSEK/VDX Operating System Version 2.2*, 2001.

[OSE01b] OSEK VDX consortium. *OSEK/VDX Time-Triggered Operating Specification 1.0*, 2001.

[OSE01c] OSEK/VDX. *Fault-Tolerant Communication - Specification 1.0*, 2001.

[Pan90] P. K. Pandya. Some comments on the assumption-commitment framework for compositional verification of distributed programs. In *REX workshop: Proceedings on Stepwise refinement of distributed systems: models, formalisms, correctness*, pages 622–640, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[PBKS07] Alexander Pretschner, Manfred Broy, Ingolf H. Kruger, and Thomas Stauner. Software engineering for automotive systems: A roadmap. In *FOSE '07: 2007 Future of Software Engineering*, pages 55–71, Washington, DC, USA, 2007. IEEE Computer Society.

[PM95] David Lorge Parnas and Jan Madey. Functional documents for computer systems. *Science of Computer Programming*, 25(1):41–61, 1995.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77)*, pages 46–57, Providence, Rhode Island, October 31–November 2 1977. IEEE Computer Society Press.

[Poh07] Klaus Pohl. *Requirements Engineering. Grundlagen, Prinzipien, Techniken*. Dpunkt Verlag, 2007.

[Pop00] Paul Pop. *Analysis and Synthesis of Communication-Intensive Heterogeneous Real-Time Systems*. PhD thesis, Institute of Technology, Linköping University, Linköping, Sweden, 2000.

[PP05] Alexander Pretschner and Wolfgang Prenninger. Computing refactorings of behavior models. *Model Driven Engineering Languages and Systems*, pages 126–141, 2005.

[PS99] Jan Philipps and Oscar Slotosch. The quest for correct systems: Model checking of diagrams and datatypes. In *APSEC'99: Asian Pacific Software Engineering Conference*, pages 449 – 458. IEEE Computer Society, 1999.

[PSS98] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *LNCS*, 1998.

[RB04] J. Romberg and A. Bauer. Loose synchronization of event-triggered networks for distribution of synchronous programs. In *Proceedings 4th ACM International Conference on Embedded Software*, September 2004.

[Rus99] John Rushby. Systematic formal verification for fault-tolerant time-triggered algorithms. *IEEE Trans. Softw. Eng.*, 25(5):651–660, 1999.

[Rus02] John M. Rushby. An overview of formal verification for the time-triggered architecture. In *FTRTFT '02: Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 83–106, London, UK, 2002. Springer-Verlag.

[SAÅ+04] Lui Sha, Tarek Abdelzaher, Karl-Erik Årzén, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, and Aloysius K. Mok. Real time scheduling theory: A historical perspective. *Real-Time Syst.*, 28(2-3):101–155, 2004.

[SACO02] Rachel L. Smith, George S. Avrunin, Lori A. Clarke, and Leon J. Osterweil. Propel: an approach supporting property elucidation. In *ICSE'02: Proceedings of the 24th International Conference on Software Engineering*, pages 11–21, New York, NY, USA, 2002. ACM.

[Sch06] Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL.* PhD thesis, Technische Universität München, 2006.

[Slo98] O. Slotosch. Overview over the project Quest. In *Proc. of FM Trends 98*, LNCS 1641. Springer-Verlag, 1998.

[SPHP02a] Bernhard Schätz, Alexander Pretschner, Franz Huber, and Jan Philipps. Model-based development. TUM-I 0204, Technische Universität München, May 2002.

[SPHP02b] Bernhard Schätz, Alexander Pretschner, Franz Huber, and Jan Philipps. Model-based development of embedded systems. In J.-M. Bruel and Z. Bellahsene, editors, *Advances in Object-Oriented Information Systems*. Springer, 2002.

[STY03] Joseph Sifakis, Stavros Tripakis, and Sergio Yovine. Building models of real-time systems from application software. *Proceedings of the IEEE*, 91(1):100–111, 2003.

[SZ06] Jörg Schäuffele and Thomas Zurawka. *Automotive Software Engineering*. Vieweg Friedr. + Sohn Ver, 2006.

[TBW94] K. W. Tindell, A. Burns, and A. J. Wellings. An extendible approach for analyzing fixed priority hard real-time tasks. *Real-Time Syst.*, 6(2):133–151, 1994.

[vL03] Axel van Lamsweerde. From system goals to software architecture. In M. Bernardo and P. Inverardi, editors, *Formal Methods for Software Architectures*, LNCS 2804, pages 25–43. Springer-Verlag, 2003.

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem — overview of methods and survey of tools. *Trans. on Embedded Computing Sys.*, 7(3):1–53, 2008.

[Yov98] Sergio Yovine. Model checking timed automata. In *Lectures on Embedded Systems, European Educational Forum, School on Embedded Systems*, pages 114–152, London, UK, 1998. Springer-Verlag.

[ZCA91] Z. Chaochen, C.A.R. Hoare, and A.P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1991.

List of Figures

List of Tables