TU Clausthal
Clausthal University of Technology

Timed Automata-Based Verification of Consistency and Realizability of Real-Time Requirements Using the Example of Distributed Functions in the Automotive Domain

Überprüfung der Konsistenz und Realisierbarkeit von Echtzeit-Anforderungen durch Abbildung auf Timed Automata am Beispiel verteilter Funktionen im Fahrzeug

Bachelorarbeit

abgegeben am 19. Juni 2017

enthält kleine Korrekturen vom 31. Mai 2018

Institute for Applied Software Systems Engineering

Erstgutachter
Prof. Dr. Andreas Rausch

Zweitgutachter
Prof. Dr. Christian Siemers

Betreuer
Dipl.-Inf. Benjamin Cool
Dr. Falk Howar

Verfasser
Jan Toennemann
Matrikelnummer 444657

Studiengang
Informatik

Fakultät
Fakultät III
Mathematik/Informatik und Maschinenbau

Erklärung der Urheberschaft

Hiermit versichere ich, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe und dass alle Stellen dieser Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, als solche kenntlich gemacht wurden und dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsstelle vorgelegt wurde.

Des Weiteren erkläre ich, dass ich mit der öffentlichen Bereitstellung meiner Abschlussarbeit in der Instituts- und/oder Universitätsbibliothek einverstanden bin.

Ort, Datum	Unterschrift

Zusammenfassung

Im Verlauf der letzten Jahrzehnte haben sich eingebettete Systeme so weit verbreitet, dass sie inzwischen das Kernstück fast aller elektronischen Geräte darstellen, die in den letzten Jahren hergestellt wurden. Eine besondere Unterart dieser eingebetteten Systeme stellen die Echtzeitsysteme dar, welche vorhersagbar und zuverlässig funktionieren müssen. Diese Echtzeitsysteme werden hauptsächlich für sicherheitskritische Systeme verwendet, für welche garantiert werden muss, dass diese korrekt und wie erwartet arbeiten.

Im Automobilbereich werden diese Echtzeitsysteme genutzt, um viele wichtige Funktionen umzusetzen, hierzu gehören beispielsweise Fahrassistenzsysteme und x-by-Wire-Architekturen. In einem modernen Fahrzeug finden sich nicht nur ein oder zwei solcher Echtzeitsysteme, sondern ein Netzwerk bestehend aus oft hunderten einzelnen Steuergeräten. Es ist üblich, dass eine Funktion auf mehrere dieser Geräte aufgeteilt wird, sodass ein komplexes System voller möglicher Wechselwirkungen entsteht.

Da diese Systeme sicherheitskritisch sind und Fehlfunktionen den Verlust von Menschenleben zur Folge haben könnten, werden diese sehr ausgiebig getestet bevor die Serienproduktion des Fahrzeuges beginnt. Wenn die Ergebnisse dieser Tests zeigen, dass das Netzwerk aus Echtzeitsystemen nicht konsistent realisierbar ist, müssen mehrere Entwicklungsstadien erneut durchlaufen werden. Dies verzögert zum einen die Lieferzeit des Fahrzeuges und führt zum anderen dazu, dass zusätzliche Kosten in einem Projekt entstehen, das zu diesem Zeitpunkt eigentlich bereits fast vollendet sein sollte.

In dieser Arbeit wird ein Ansatz vorgestellt, mit dem die Konsistenz und Realisierbarkeit der Anforderungen, die an die Echtzeitsysteme gestellt werden, bereits am Ende der Planungsphase verifiziert werden können. Unser Ansatz unterstützt in der Spezifikation ebendieser Anforderungen, indem die Integrierbarkeit des Systems bei der Planung und Entwicklung bereits überprüft werden kann. Basierend auf den gegebenen Echtzeitanforderungen können zuverlässige Aussagen über die Integrität des fertigen Systems getroffen werden können.

Das vorgestellte Verfahren ermöglicht die automatisierte Verifikation eines modellierten Netzwerks von Echtzeitsystemen, um Informationen über die Konsistenz von Echtzeitanforderungen schon in der Planungsphase zu erlangen. Falls die Verifikation aufzeigt, dass die Anforderungen inkonsistent sind, so werden Informationen ausgegeben, die bei der Weiterentwicklung zu einem konsistenten System unterstützen.

Abstract

Over the course of the recent decades, embedded systems have become so prevalent that they now constitute the core of practically every electrical device manufactured. A special type of embedded systems are real-time systems, which are designed to be predictable and reliable. These real-time systems are commonly used in a lot of safety-critical systems and it is of utmost importance to guarantee that they work as expected.

In the automotive domain, these real-time systems are used to implement numerous functions including assistive systems and x-by-wire architectures. Modern cars are composed not of one or two real-time systems, but rather of a network consisting of hundreds of these systems. It is typical for a function to be distributed among multiple devices, creating a complex system full of interdependencies.

Since these systems are safety-critical and their malfunction could lead to loss of human life, very intricate testing is done before the main manufacturing stage is entered. When the testing results indicate that there is no way to realize the network of automotive software systems consistently, several development stages need to be repeated. This delays the final delivery and requires additional resources to be spent on a project which was supposed to be almost finished.

In this thesis we propose an approach to verify the consistency of requirements imposed on real-time systems earlier in the process, at the end of the planning stage. Our approach assists in the specification of requirements and helps during the development of automotive software systems by verifying the feasibility of these systems. Based on the given real-time requirements, reliable assertions can be made regarding the integrity of the final system.

The presented method allows for the automated verification of a modeled network of automotive software systems to gain information on the consistency of the requirements using data already available at the end of the planning stage in the process. Should the result of the verification indicate inconsistent requirements, detailed information is output to assist in the development of a consistent system.

Contents

1 Introduction

1.1 Motivation

More than two decades ago, the use of embedded systems was already commonplace in practically every product involving electronics. This includes everything from consumer-grade entertainment equipment like televisions, radios and phones up to critically important systems like aircraft autopilots, numerous life-critical medical devices as well as several software-based assistance systems inside of cars[25].

Especially for these safety-critical systems, the discipline of real-time computing has emerged, which aims to create fully predictable software systems, both in regards to timely execution as well as correct behavior[48]. Research on the usage of digital computing in environments where human life depends on it dates back to even before 1970, as the NASA already successfully used digital computers for it's Apollo missions to the moon[32]. Creating dependable embedded systems requires a lot of considerations and restrictions and their development follows strict guidelines. The vast importance of ensuring the reliability of these systems has been highlighted often, although assessment of the risks involved has shown to be rather problematic[14].

Starting in 1976, the automotive industry started introducing more and more digital systems as well as accompanying software into their cars[40]. Even fifteen years ago, cars already had up to 80 embedded controllers that all needed to be properly developed and tested[12]. The heterogeneous, largely supplier-based nature of automotive development processes has proven to be a challenging issue, as software and hardware from multiple sources needs to be integrated into a final, working product. Automotive systems, consisting of a large number of communicating Electronic Control Units (ECUs), are required to handle an ever increasing number of complex tasks and also need to fulfill a multitude of specific requirements related to safety and reliability[40].

In an attempt to create a common base for the development of these systems, the Automotive Open System Architecture (AUTOSAR) was founded as a development partnership and published its first major set of specifications in 2006. The goal of this partnership is to provide an open industry standard to narrow the gap between car manufacturers and individual suppliers[37]. But even with these common standards, the domain of automotive systems engineering still provides many unsolved challenges, especially in regards to real-time system development[13].

Following the AUTOSAR approach, the development process of automotive software systems is now largely function-oriented. This means that the development of software and the actual hardware it is deployed on is decoupled, allowing for great flexibility in regards to the actual configuration of the final system[20]. But this also introduces another problem domain, as such a function might behave differently when split among multiple ECUs in comparison to when it is being executed on a single one. Since dealing with these *distributed functions*, most failures of safety-critical systems can be traced back to problems in the interaction between the subsystems, not to failures of individual units[23].

As a result, there are even stricter and more complex restrictions in place for distributed functions as well as systems composed of a multitude of these. Dealing with requirements in the automotive domain has become a discipline in itself and the specification of feasible requirements for larger project often takes months and requires multiple iterations[11]. As a major problem domain of requirements in automotive software systems is timing, the AUTOSAR standard has been extended by the AUTOSAR Timing Exten-

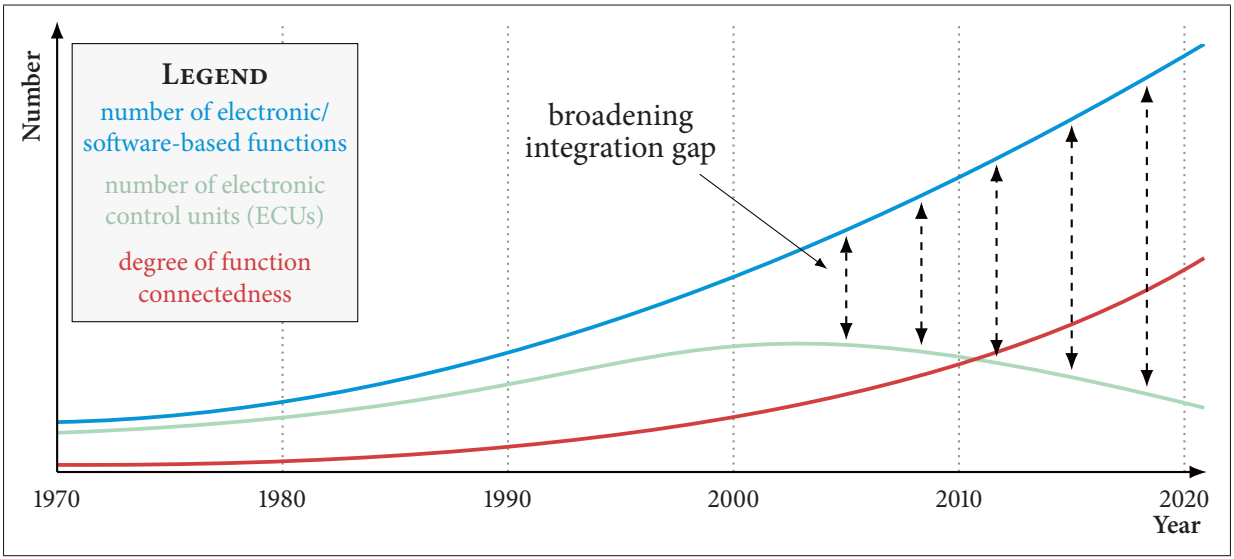

Figure 1.1: Evolution of Automotive Software Systems over Time[1]

sions (TIMEX) in an effort to create a common standard for their specification and characterization. This standard has been developed by bringing together results from multiple research projects, notably the ITEA 2 project TIMMO[49] and the doctoral thesis of Oliver Scheickl[45].

The topic of timing constraints in automotive software systems, especially distributed systems, has been a field of extensive research in recent years and still continues to be. Analyzing and simulating the behavior of a single real-time system is not trivial, but has been reliably accomplished decades ago. Characteristics of real-time systems to account for when doing so include task configuration and scheduling behavior, consideration of both internal as well as external interrupts and the reliability real-time clock itself. Even considering a processor with multiple cores and parallel execution, it is possible to ensure that the deployed software will perform reliably under all considered circumstances[15].

Considering a network of real-time systems introduces a whole new layer of complexity, resulting in a more complex simulation and analysis. For each new system introduced into the network an additional real-time clock needs to be considered, which might not run synchronous to that of the other systems in the network[57]. There exist various approaches to develop and test these inter-connected systems and generally, the analysis of distributed real-time systems inside certain bounds can also lead to reliable results[17]. But the thorough analysis that is necessary for these results requires a large amount of data about the system, requiring both the system development as well as its implementation and configuration to be already finished when starting the tests.

As each function is only tested on its own by the supplier, the behavior and timing influences of integrating a multitude of functions from various suppliers into a combined system is often very complex[45]. When also considering additional timing constraints for the different functions, a reliable simulation of the actual behavior and checks on whether these requirements are fulfilled can only be done very late in the process[22]. At this point, the results might indicate that with the current system composed of ECUs and the different functions, the timing constraints cannot be fulfilled. Since detecting these inconsistencies this late in the process requires major changes to the system as a whole, they introduce a lot of additional work

[1] based on the figure about the three phases of automotive software system development in [45]

and might delay a project substantially, severely increasing the cost[56]. In this thesis we will propose an approach to detect these inconsistencies earlier in the process, requiring only a small subset of information about the final system to verify whether fulfilling the constraints is possible.

1.2 Goals

Our first goal is the detection of inconsistencies in specified real-time requirements early in the process. In order to do that, we need to determine which inconsistencies can be detected in any set of specified real-time requirements and which data is required to do so. The more data we have, the more thorough of an analysis can be performed, making the results more accurate and profound. But since most of the data is only available later in the process, we are required to find a trade-off between the amount of data required and the fidelity of the analysis and verification.

The second goal is the automated detection of the inconsistencies that have been deemed identifiable. We want to develop a method that takes a set of requirements and a sample of basic parameters describing a planned system as input and gives information on whether this input represents consistent, feasible system or if there are requirements that contradict properties of the defined system, making it inconsistent.

This approach can then be used as one of the last steps in the planning process of distributed automotive software systems, in an attempt to prevent issues hindering the realization of the final system in the integration step near the end of the process. Considering a classic V-model-based process like shown in figure 1.2, this approach would shift the identification of inconsistencies from the *system integration* step to the *low-level system design* step of the process.

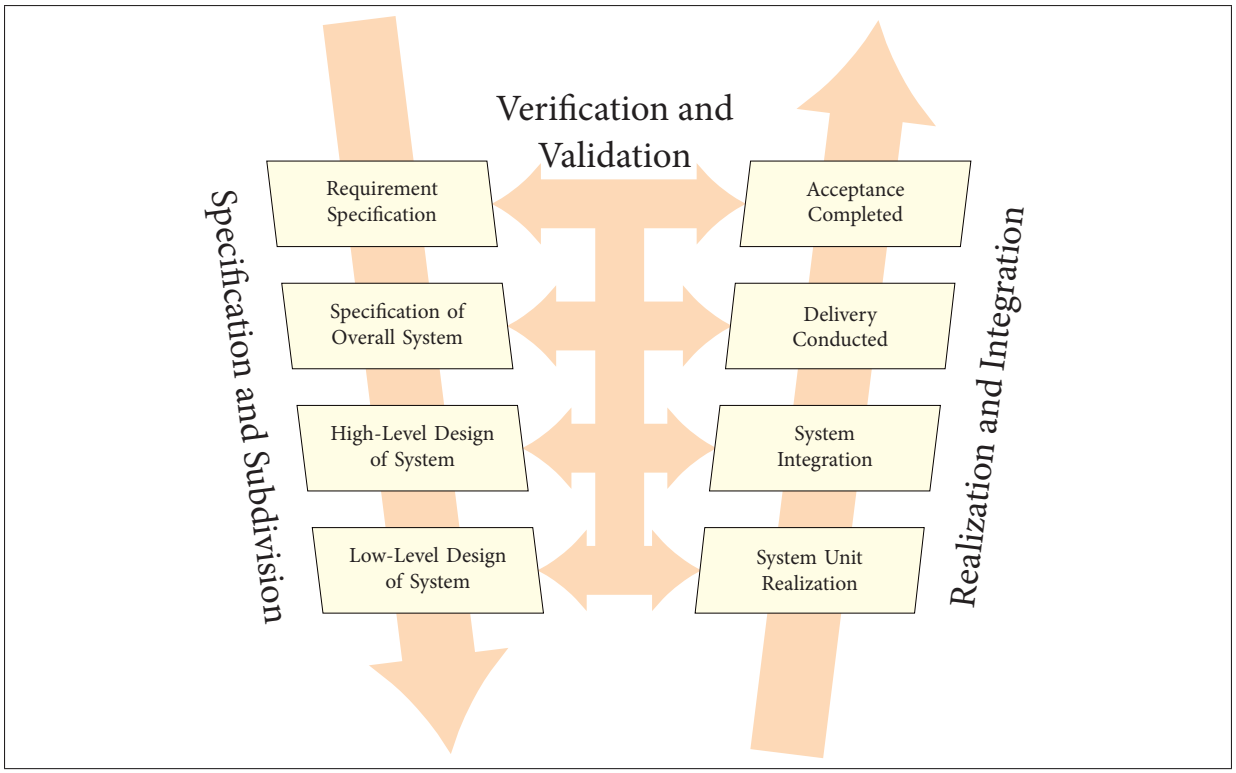

FIGURE 1.2: Process model based on V-model publications[10, 58, 59]

1.3 Approach

While basic inconsistencies in a set of requirements can be identified using simple arithmetic, most are complex enough to require at least a rough model of the planned system. Using data available in the planning stages of a process, we want to simulate the basic behavior of real-time systems in regard to timing as accurately as possible to check whether a planned system conforms to a given set of requirements. For this we require a time-aware model that is able to represent the discrete behavior of distributed real-time systems, for which we chose a state-based approach. Early concepts of state-based modelling of real-time systems dates back to 1990[2] and the automata concept which we will use in this thesis, that of Timed Automata (TA), has been proposed only four years later[3].

Timed automata are an extension of regular finite automata and allow for the simulation of time-aware behavior[6]. We use the tool UPPAAL[2], developed by researchers of the Swedish Uppsala University as well as the Aalborg University in Denmark, to allow for automated verification of such automata. The model-checking itself is done in UPPAAL by querying a single automaton or a system of multiple timed automata using a subset of Timed Computation Tree Logic (TCTL), a time-aware extension of CTL originally introduced in 1990[1].

The data we need for this verification is comprised of set of real-time requirements and a system plan, consisting of estimates for the task execution time bounds and a distribution of the tasks on ECUs as well as basic scheduling parameters. Apart from the specified requirements, each parameter can be variable and as such, it shall be possible to test a multitude of scenarios with only slight changes. As a result we want to obtain sound information about which, if any, requirements are not feasible as well as additional information on the counter-example encountered by the model-checker in case of a violated requirement.

Since we are using UPPAAL, the verification itself can be achieved in an automated manner, given a model and the relevant queries. To also allow for the automation of the formalization, we show how to map requirements given in the Timing Augmented Description Language 2 (TADL2) to our formalization, such that queries can be automatically generated from TADL2 specifications. TADL2 is a result of the ITEA 3 project TIMMO-2-USE and provides a way to denote real-time requirements in a textual format that can be mapped to AUTOSAR TIMEX, such that this approach can be integrated into the workflow current processes utilizing the AUTOSAR specification[4, 9].

1.4 Related Work

In the development of real-time systems, especially in safety-critical environments, model-checking is a common practice[1, 41, 47]. [1] primed the development of automata-based modelling of real-time systems[2] and timed automata[3]. Over the years several approaches have been developed using timed automata to model-check real-time systems.

Several papers deal with the aspect of *schedulability*, attempting to determine whether a system can be scheduled using a dynamic scheduling strategy and a set of scheduling parameters. The paper 'Timed Automata as Task Models for Event-Driven Systems'[38] proposes an approach to model real-time, task-based systems using timed automata. They perform schedulability analysis on real-time systems based on the Earliest Deadline First (EDF) approach and use UPPAAL for a case study showcasing automated verification.

[2] available under `http://uppaal.org/`

Another example of schedulability analysis can be found in [19], where an extension of timed automata called *task automata* is presented. These automata allow the modelling of tasks in software systems with execution time bounds and basic scheduling parameters as special states. The main goal of this research was to perform schedulability analysis not only of tasks with periodic triggering but also sporadic activation, supporting the extension by additional activation patterns. Decidability and undecidability of those systems as well as possible issues regarding schedulability are detailed. Automated model-checking was not covered for this new class of automata and at this time, there does not exist a tool at the time to do so, unfortunately making these not applicable for our approach.

'Formal Analysis and Testing of Real-Time Automotive Systems Using UPPAAL Tools'[29] approaches the analysis and testing of automotive software systems not only using the UPPAAL model checker, but also using the experimental statistical model checker of UPPAAL. In this article, a method is outline to apply methods of statistical model checking (SMC) to real-time systems, performing statistical evaluation and comparison as well as hypothesis testing. A case study is performed on by modelling a turn indication system using a network of timed automata and the verification of various properties including functional and timing properties using regular model checking as well as SMC. This approach requires very detailed modelling of real-time systems and is mostly applicable at late stages of system development.

Dissertations about model-based development of automotive software systems can be found in [31, 60]. The thesis in [60] proposes a model-based test framework based on MATLAB, Simulink and Stateflow called *MiLEST* and [31] proposes the *COLA automotive approach* of model-based development based on a newly created modelling language. Both theses include numerous case studies to showcase the development workflow using the respectively proposed approach.

Outside of model-checking and model-based development we also find research on the analysis of real-time systems as well as automotive software systems. An approach to the automated verification of real-time communicating systems using constraint-solving instead of model-checking is detailed in [21]. The paper in [57] presents an analysis technique for distributed real-time systems considering task-based systems and scheduling.

Another related work is [43], using formal and statistical analysis to determine schedule synthesis of time-triggered automotive communication buses. The analysis performed relates to timing in distributed architectures and the predictability of network communication in distributed automotive software systems. Similar is [27], utilizing discrete event simulation and network calculus to model and verify in-car communication for time-critical applications in the automotive domain.

The user documentation of Symtavision's tool 'Symbolic Timing Analysis for Systems' (SymTA/S) also covers the theory on which the tool is based[52]. SymTA/S can be used to perform statistical analysis on distributed real-time systems and a more in-depth comparison of our approach to that of SymTA/S is found later in section 6.2.

The papers in [10] and [11] provide research on requirements engineering for large automotive software systems, with [10] using the EAST-ADL requirements specification, which can also be mapped to AUTOSAR. Both reference the V-Model XT and detail a development process similar to the one assumed in this thesis.

Work regarding timing in automotive software systems including a timing model can be found in [22, 45], both of which are related to the specifications and concepts that are part of AUTOSAR TIMEX and

TADL2 as well[4, 9]. Being part of the TIMMO project, [50] proposes a design framework using these concepts to develop automotive systems with timing constraints. These works were used as references throughout this thesis as a foundation for timing in automotive real-time systems.

1.5 Structure of this Thesis

After the introduction in chapter 1, chapter 2 will introduce timed automata, TCTL and TADL2, which will be used throughout this thesis.

In the course of chapter 3 we will present an approach to the formalization of concepts in real-time systems and automotive software developments. We start by introducing events, event chains and functions in section 3.1, detail the formalization of actual software systems in section 3.2 and declare the real-time requirements covered in this thesis in section 3.3. The detection of some first inconsistencies in a set of real-time requirements using simple arithmetic operations and without any system model is explained in section 3.3.6.

Based on the formalization done in section 3.2, chapter 4 describes how to model processing environments using timed automata. In section 4.1 we explain necessary preparations and assumptions made in the model-building process. Section 4.2 contains details on how to create a model of timed automata from a formalized processing environment and in section 4.3 we implement this model using UPPAAL.

Using this model as a foundation, we show how the real-time requirements from section 3.3 can be transferred to TCTL queries for verification in chapter 5. This chapter shows how to verify requirements using only TCTL queries and the modeled system in section 5.1, proposes additional automata for state-based verification in section 5.2 and details how to resolve detected inconsistencies using the acquired results in section 5.4.

This workflow is applied to an example of a distributed brake-by-wire function in chapter 6, specifying a distributed function in section 6.1 and doing a case study starting in section 6.1.1, using to proposed approach to proceed from an initially assumed, inconsistent system plan to a realizable system distribution in multiple iterations. We will compare our approach to using SymTA/S on the same example in section 6.2, showing equivalences in achieved results and restrictions of applying a test-based approach in the conceptional stage.

The conclusion in chapter 7 consists of a summary and discussion of the developed method as well as proposals for additions to the model and future research in this field. Additional resources can be found in the appendix, including the bibliography and a list of symbols. Appendix A starting at page VII contains referenced material and Appendix B starting at page XIV includes complete code listings and UPPAAL templates.

2 Preliminaries

Before starting with our approach to formalization, we will introduce the basics of timed automata, the verification language TCTL as well as the textual description language TADL2 that allows to denote timing and timing requirements in automotive software engineering. The introductory definitions of Timed Automata to the most part follow those detailed in [6], incorporating the more basic definitions from [3] and also introducing concepts from [8].

2.1 Timed Automata

Originally proposed by R. Alur and D. L. Dill[2, 3], *Timed Automata* are an extension of the model of finite automata, designed to incorporate real-time properties using automata-internal clock simulation and timed guards dependent on these. A method to automatically verify Timed Automata was introduced by researchers from the Swedish Uppsala University[21], priming the development of a tool called UPPAAL, which has since been vastly extended.

Timed automata extend regular finite automata by clocks and since these clock are a fundamental part of the definitions in this chapter, we will start by introducing clocks themselves as well as possible operations on them.

Definition 2.1 (Clocks in Timed Automata)**.** Let $\mathcal{C}$ be a finite set of variables called clocks.

A *clock valuation* over $\mathcal{C}$ is a mapping $v : \mathcal{C} \mapsto \mathbb{R}_0^+$ which assigns to each clock a time value. Let $d \in \mathbb{R}_0^+$, the valuation $v + d$ is defined by $(v + d)(c) = v(c) + d, \forall c \in \mathcal{C}$.

For a $r \subseteq \mathcal{C}$, let $[r \mapsto 0]v$ denote the clock assignment that maps all clocks in r to 0 and conforms to v for all other clocks in $\mathcal{C} \setminus r$.

A *clock constraint* is a conjunctive formula of atomic constraints, either of the form $x \bowtie n$ or $x - y \bowtie n$ for $x, y \in \mathcal{C}, n \in \mathbb{N}_0, \bowtie \in \{<, \leq, =, \geq, >\}$. We will use $\mathcal{B}(\mathcal{C})$ to denote the set of clock constraints, ranged over by g. For verification purposes, we also introduce the set $\mathcal{B}'(\mathcal{C})$, the downwards closed set of clock constraints, with $\bowtie \in \{<, \leq\}$.

Definition 2.2 (Timed Automaton)**.** Let Σ be a a finite alphabet of actions.

A *timed automaton* $\mathcal{A}$ is a tuple $\langle N, l_0, E, I \rangle$ where

- N is a finite set of locations,
- $l_0 \in N$ is the initial location,
- $E \subseteq N \times \mathcal{B}(\mathcal{C}) \times (\Sigma \cup \{\epsilon\}) \times 2^{\mathcal{C}} \times N$ is the set of edges, and
- $I : N \mapsto \mathcal{B}'(\mathcal{C})$ assigns invariants to locations.

We will write $l \xrightarrow{g,a,r} l'$ when $\langle l, g, a, r, l' \rangle \in E$. Extending the notion of actions to $a \in (\Sigma \cup \{\epsilon\})$, we will allow empty actions. When a transition does not require input from the finite alphabet Σ, we will abbreviate $\langle l, g, \epsilon, r, l' \rangle \in E$ as $\langle l, g, r, l' \rangle \in E$ and denote transitions using $l \xrightarrow{g,r} l'$.

We consider a pair (t, a) with an action a and a point in time $t \in \mathbb{R}^+$ a *timed action* taken by a timed

automaton $\mathcal{A}$. The *time-stamp* t is measured from the beginning of $\mathcal{A}$'s start.

A *timed trace* is a possibly infinite, sequence of timed actions $\lambda = (t_1, a_1)(t_2, a_2)\ldots(t_i, a_i)\ldots$ where the time t is weak monotonically increasing for all $i \geq 1$, so that $t_i \leq t_{i+1}$ holds.

Invariants are restricted to the downwards closed set of clock constraints since this is a requirement for the verification of the system.

Definition 2.3 (Operational Semantics of a Timed Automaton)**.** The semantics of a timed automaton is defined as a *timed transition system* where states are pairs $\langle l, v\rangle$ and transitions are defined by the following rules:

- $\langle l, v\rangle \xrightarrow{d} \langle l, v + d\rangle$ if $v \in I(l)$ for any $d \in \mathbb{R}^+$
- $\langle l, v\rangle \xrightarrow{a} \langle l', v'\rangle$ if $l \xrightarrow{g,a,r} l', v \in g, v' = [r \mapsto 0]v$ and $v' \in I(l')$

To indicate a transition to another location based on both an action a and a delay d we will use the abbreviated notation $\langle l, v\rangle \xrightarrow{a,d} \langle l', v + d\rangle$. Further, we will allow *silent transitions*, where a is the empty word ϵ, as well as *zero-delay transitions*, where d is 0, so that transitions that require neither a delay nor any input are possible; these will use the notation $\langle l, v\rangle \xrightarrow{0} \langle l', v\rangle$, silent transitions will be denoted like this with a $d > 0$.

A *run* of a timed automaton $\mathcal{A} = \langle N, l_0, E, I\rangle$ with initial state $\langle l_0, v_0\rangle$ over a timed trace $\lambda = (t_1, a_1)(t_2, a_2)(t_3, a_3)\ldots$ is a sequence of transitions satisfying the condition $t_i = t_{i-1} + d_i$ for all $i \geq 1$:

$$\langle l_0, v_0\rangle \xrightarrow{a_1,d_1} \langle l_1, v_1\rangle \xrightarrow{a_2,d_2} \langle l_2, v_2\rangle \xrightarrow{a_3,d_3} \langle l_3, v_3\rangle \cdots$$

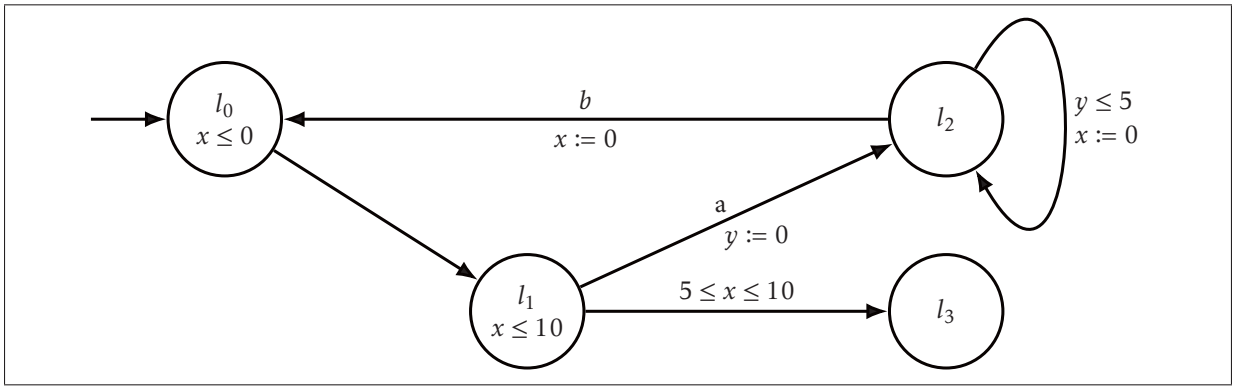

Figure 2.1: Simple timed automaton with four locations

In figure 2.1 we see an example automaton with the locations $l_0, \ldots, l_3$, the clocks x, y and the alphabet $\{a, b\}$. It initially starts in location l_0 with both clocks set to 0 and since $I(l_0) = x \leq 0$, the location must be left immediately, without any time advancing – since the only outgoing edge leads to l_1, the automaton must instantly enter this location after entering l_0. When in l_0, time may pass until the clock x reaches a value of 10; if it is between 5 and 10, the automaton may enter l_3, a location with no outgoing edges, causing a *deadlock*. Alternatively, when an a is given as input, the automaton may transition to location l_2, resetting the value of clock y in the process. Should the conditions of both outgoing edges from l_1 be true at the same time, the automaton chooses an edge non-deterministically, as known from non-deterministic finite

state automata. While in l_2 and y is still less than 5, the automaton can traverse a looped edge resetting the clock value of x without changing the location. Should it receive a b as input, the value x is reset as well and the automaton changes back to the initial location l_0, although after an iteration the clock value of y might now be larger than 0.

An example of a valid, finite run of this automaton would be:

$$\langle l_0, \begin{pmatrix}0\\0\end{pmatrix}\rangle \xrightarrow{\epsilon,0} \langle l_1, \begin{pmatrix}0\\0\end{pmatrix}\rangle \xrightarrow{a,7} \langle l_2, \begin{pmatrix}7\\0\end{pmatrix}\rangle \xrightarrow{\epsilon,3} \langle l_2, \begin{pmatrix}0\\3\end{pmatrix}\rangle \xrightarrow{b,2} \langle l_0, \begin{pmatrix}0\\5\end{pmatrix}\rangle \xrightarrow{\epsilon,0} \langle l_1, \begin{pmatrix}0\\5\end{pmatrix}\rangle \xrightarrow{\epsilon,10} \langle l_3, \begin{pmatrix}10\\15\end{pmatrix}\rangle$$

2.1.1 Timed Automata for Concurrent Systems

Since each timed automaton can only have one active state at a time, modelling concurrent systems using a single automaton would require a very complicated automaton. A more reasonable approach is to use a network of multiple timed automata and simulate their parallel execution, which is what is also done in UPPAAL. Definitions and techniques related to this topic will be covered in this subsection.

Definition 2.4 (A Network of Timed Automata). For a set of timed automata $A_1, \ldots, A_n$ called *processes*, let the parallel composition $A_1 \mid \cdots \mid A_n$, called a *network of timed automata*, be a single system defined by the parallel composition operator of the Calculus of Communicating Systems (CCS); for an in-depth explanation of CCS and this parallel composition please refer to [36]. Broadcast communication between processes in such a network is accomplished by utilizing broadcast channels, which require extending the alphabet Σ by the following:

- input actions denoted using $a?$,
- output actions labelled as $a!$, and
- internal actions being represented distinctly using η

Asynchronous communication is achieved using variables shared between two or more processes.

Definition 2.5 (Operational Semantics of a Network of Timed Automata). Since the network of timed automata is just a parallel composition of timed automata, we can give a similar definition of operational semantics in terms of transition systems.

Let $\langle l, v\rangle$ be the state of a network, where l is a vector of current locations in the network, one per process, and v is the clock assignment remembering the values of the clocks in the system. Let l_i be the ith element of a location vector l and let $l\left[l_i' \,/\, l_i\right]$ stand for the vector l with l_i being substituted with l_i'. Then, delay and action transitions can simply be defined as the following rules:

- $\langle l, v\rangle \xrightarrow{d} \langle l, v + d\rangle$ if $v \in I(l)$ and $(v + d) \in I(l)$ for any $d \in \mathbb{R}^+$, where $I(l) = \bigwedge_i I(l_i)$
- $\langle l, v\rangle \xrightarrow{\eta} \left\langle l\left[l_i' \,/\, l_i\right], v'\right\rangle$ if $l_i \xrightarrow{g,\eta,r} l_i', v \in g, v' = [r \mapsto 0]v$ and $v' \in I\left(l\left[l_i' \,/\, l_i\right]\right)$

Definition 2.6 (Shared Variables and Broadcast Channels). Considering clocks to simply be a variable of a specific type, we can introduce shared variables and arrays as known from traditional programming languages, represented in UPPAAL using a C-like syntax. To include shared integer variables, we need to define synchronized transitions with special regard to symmetry of input and output actions.

Since we are only using broadcast channels, that allow the emitter of a broadcast – the edge with the output action – to transition regardless of how many, if any, recipients there are, we introduce the following rules to our transition system:

- $\langle l, v\rangle \xrightarrow{\eta} \left\langle l\left[l_i' \,/\, l_i\right], v'\right\rangle$ if there exists an i such that $l_i \xrightarrow{g_i, a!, r_i} l_i'$ and $v \in g_i$
- $\langle l, v\rangle \xrightarrow{\eta} \left\langle l\left[l_i' \,/\, l_i\right]\left[l_j' \,/\, l_j\right], v'\right\rangle$ if there exists an $j \neq i$ such that
  1. $l_i \xrightarrow{g_i, a!, r_i} l_i'$ and $v \in g_i$,
  2. $l_j \xrightarrow{g_j, a?, r_j} l_j'$ and $v \in g_j$, and
  3. $v' = \left[r_i \mapsto 0\right]\left(\left[r_j \mapsto 0\right]v\right)$ and $v' \in I\left(l\left[l_i' \,/\, l_i\right]\left[l_j' \,/\, l_j\right]\right)$

The notion can be extended to an arbitrary amount of broadcast receivers. Each receiver *must* take the input transition when a broadcast is sent on the corresponding broadcast channel as long as the transition does violate neither transition guards nor invariants.

Definition 2.7 (Committed Locations in Timed Automata). We allow timed automata to have *committed locations*, which are locations in which no time may pass such that directly after entering the location the automaton is required to continue on an outgoing edge. In a network of timed automata, processes in committed locations can only be interleaved with other processes in a committed location.

Let each process A_i in a network have a subset $N_i^C \subseteq N_i$ of committed locations. On outgoing edges of locations in N_i^C, no clock constraints may be used. Considering committed locations, the following transition rules for a network of timed automata must be added, with $\rightarrow_c$ denoting the transition relation for a network with committed locations and $\rightarrow$ denoting the transition relation for the same network ignoring committed locations:

- $\langle l, v\rangle \xrightarrow{d}_c \langle l, v + d\rangle$ if $\langle l, v\rangle \xrightarrow{d} \langle l, v + d\rangle$ and $\bigcup_k \left(\{l_k\} \cap N_k^C\right) = \emptyset$
- $\langle l, v\rangle \xrightarrow{\eta}_c \left\langle l\left[l_i' \,/\, l_i\right], v'\right\rangle$ if
  1. $\langle l, v\rangle \xrightarrow{\eta} \left\langle l\left[l_i' \,/\, l_i\right], v'\right\rangle$, and
  2. either $l_i \in N_i^C$ or $\bigcup_k \left(\{l_k\} \cap N_k^C\right) = \emptyset$
- $\langle l, v\rangle \xrightarrow{\eta}_c \left\langle l\left[l_i' \,/\, l_i\right]\left[l_j' \,/\, l_j\right], v'\right\rangle$ if
  1. $\langle l, v\rangle \xrightarrow{\eta} \left\langle l\left[l_i' \,/\, l_i\right]\left[l_j' \,/\, l_j\right], v'\right\rangle$, and
  2. either $l_i \in N_i^C, l_j \in N_j^C$ or $\bigcup_k \left(\{l_k\} \cap N_k^C\right) = \emptyset$

For a single automaton a committed location is equivalent to adding a clock x to the location, having all incoming edges reset x to 0 and adding the invariant $x \leq 0$ to the location; refer to figure 2.1, where this was already used in the location l_0.

Figure 2.2 shows a network of timed automata, one automaton being an extension of the one shown in figure 2.1, with the initial location marked as committed and an input transition out of l_3, receiving messages on broadcast channel c. The other automaton in the network has only two locations l_0' and l_1' and

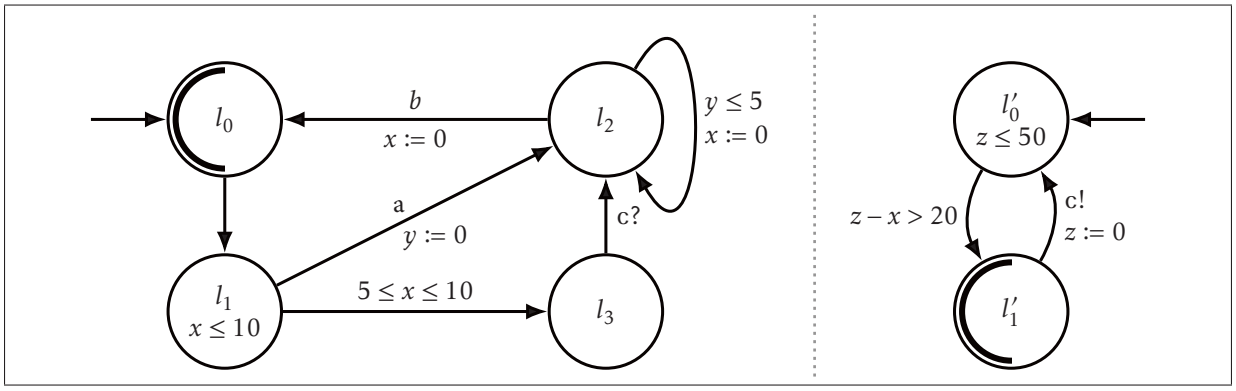

FIGURE 2.2: Network composed of two timed automata

introduces the additional clock z into the system. From the initial location l'_0 there is only one transition, which can only be taken when $z-x > 20$ holds true; note that this makes the clock x used by both automata. Having reached l'_1, which is a committed location, the automaton sends an output action on broadcast channel c and resets clock z; should the first automaton be in location l_3, it must take the transition marked with c? leading to l_2, since it receives the broadcast and must act upon it.

2.2 Timed Computation Tree Logic

We will introduce the *Timed Computation Tree Logic (TCTL)*, an extension of the Computation Tree Logic used to verify properties of finite automata first specified in [1] – that paper is the base for the following definition. Since UPPAAL implements only a subset of TCTL[6], we will limit the definition here to the formulas supported in UPPAAL.

Definition 2.8 (Formulas of TCTL). Let P be a set of atomic propositions, $p \in P$, $c \in \mathbb{N}$ and $\bowtie \in \{<, \leq, =, \geq, >\}$. Then each valid formula of TCTL is inductively defined as follows:

$$\phi := p \mid \textbf{false} \mid \phi_1 \rightarrow \phi_2 \mid \exists \phi_1 \, \mathcal{U}_{\bowtie c} \, \phi_2 \mid \forall \phi_1 \, \mathcal{U}_{\bowtie c} \, \phi_2$$

In TCTL, $\exists \phi_1 \, \mathcal{U}_{\bowtie c} \, \phi_2$ expresses that there exists a computation path and a time length bound by $\bowtie \; c$ such that ϕ_2 holds when $\bowtie \; c$ is fulfilled and ϕ_1 holds in all the intermediate states. Similarly, $\forall \phi_1 \, \mathcal{U}_{\bowtie c} \, \phi_2$ denotes the same property but instead requires it to hold for every possible computation path. For $\phi_1 \rightarrow \phi_2$, ϕ_1 implies ϕ_2 and thus ϕ_2 must hold when ϕ_1 is fulfilled. The subset of TCTL available for verification in UPPAAL is listed in table 2.1.

TCTL Formula	UPPAAL Formula	Explanation
$\forall\Box\phi$	`A[]` ϕ	ϕ holds invariantly and is thus true in all reachable states of the network.
$\forall\Diamond\phi$	`A<>` ϕ	ϕ holds always eventually and is thus true at the end of each valid path.
$\exists\Box\phi$	`E[]` ϕ	There exists at least one valid path for which ϕ is true in all states.
$\exists\Diamond\phi$	`E<>` ϕ	There exists at least one valid path for which ϕ is true in the end.
$\psi \rightarrow \phi$	ψ `imply` ϕ	When ϕ holds, this implies that ψ must hold as well.

Table 2.1: Subset of TCTL queries implemented in UPPAAL

To ensure that 'something bad never happens' one needs to verify *safety* properties, such that for any formula ϕ that may never be true, the safety property $\forall\Box\neg\phi$ must hold. A special kind of safety property is *deadlock*-freeness, which, when verified, ensures that it is not possible for the system to reach a state from which it cannot further progress – this can be verified in UPPAAL using the query `A[] not deadlock`.

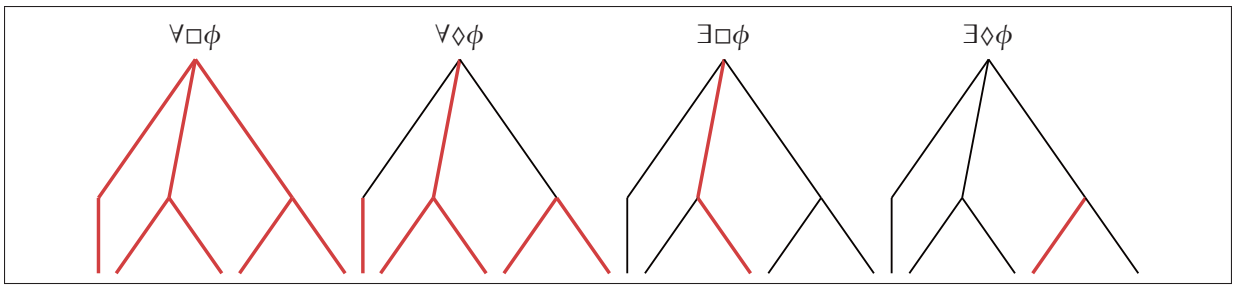

Figure 2.3: Visualization of the four main TCTL formula/query types

We have visualized verification paths for the first four query types introduced in table 2.1 in figure 2.3 using a tree-like depiction of possible system states. Starting at the top, the possible system states branch and the states where the corresponding query holds are marked bold red. If we assume $\phi \rightarrow \psi$ for any of the shown queries, then ψ would hold in at least all highlighted branches as well.

2.3 Timing Augmented Description Language (Version 2)

The Timing Augmented Description Language 2 (TADL2) is a result of the ITEA 3 project TIMMO-2-USE and an extension of the TADL developed as part of the original TIMMO project in ITEA 2[9]. It provides a standardized way of denoting concepts of timing in automotive software engineering as well as corresponding requirements in a textual format. The defined concepts and constraints can be mapped to that of AUTOSAR TIMEX and according to the AUTOSAR Timing Analysis document, 'TADL2 base concepts are quite equivalent to those of AUTOSAR TIMEX' [4, p. 30]. We will use TADL2 in chapter 3 to provide a mapping between textual requirements and our developed formalization. Relevant snippets of TADL2 declarations are included in section 3.1 and section 3.3, in this section we will only introduce necessary basics of a TADL2 specification that are shared between all other TADL2 declarations in this thesis.

To work with timing relations in TADL2, we need to define a universal time base, which serves as a

reference for the time base of systems and allows to differentiate between multiple, differently precise systems. Since our approach is applicable in the planning stage of a process and we do not have any data about differences in the precision of the systems, we will declare a fully-accurate, shared time base for all systems. The `universal_time` time base in listing 2.1 represents such a time base and will be considered to be the universal time base for all TADL2 declarations in this thesis.

```
Dimension physical_time {
    Units {
        micros{factor 1.0 offset 0.0},
        ms{factor 1000.0 offset 0.0 reference micros},
        second{factor 1000000.0 offset 0.0 reference micros}
    }
}

TimeBase universal_time {
    dimension physical_time,
    precisionFactor 0.1,
    precisionUnit micros
}
```

Listing 2.1: TADL2 description of a universal, fully-accurate time base

In section 3.1 we cover the definition of events and event chains in TADL2. Section 3.3 shows how timing constraints are represented in TADL2 and in chapter 6 we will show a full declaration of a distributed function in TADL2, including real-time requirements. This thesis only covers a small subset of the timing constraints that can be described using TADL2; a full listing can be found in [9].

3 Formalization of Automotive Software System Concepts

In this chapter we will introduce concepts common in automotive software engineering and propose a formalization approach with regards to timing analysis. We will start with the conceptual basics of timing in section 3.1, defining events, event chains and functions. In section 3.2 we will define how to formalize automotive systems as well as tasks and also explain relevant properties like scheduling. At the end of this chapter in section 3.3, we introduce the real-time requirements that will be examined in this thesis, show how they are expressed in TADL2 and how to transfer the definitions from our formalization to TADL2 and back.

3.1 Basics of Timing in Automotive Software Engineering

In order to define any real-time requirements, we need a common concept of timing in automotive systems. For this sake we will introduce the concept of events in this section and define event chains as well as functions in the timing context, allowing

3.1.1 Event

The core of timing analysis in automotive software engineering are events, they are a concept common to the timing mode of AUTOSAR TIMEX and TIMMO-2-USE[4, 9]. Events allow to specify observable actions in a software system, for which relations might exist. A comparison of different event models, and a more in-depth look at the concept in general, can be found in [45].

Definition 3.1 (Event). We consider an *event* to be an observable entity in temporal context. Each event e is distinctly identifiable and observable.

Definition 3.2 (Event Occurrence). We say that an event can *occur* at a point in time, which we will then refer to as an *occurrence* of that event. Each event occurrence $o = (e, t)$ is a tuple of an event e and a point in time $t \in \mathbb{R}_0^+$.

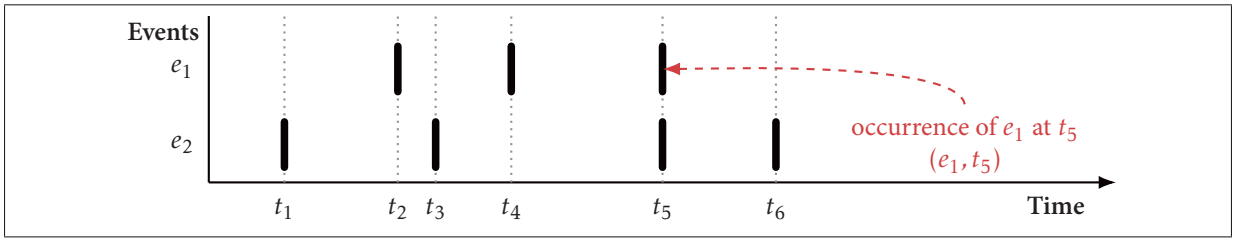

FIGURE 3.1: Multiple occurrences of different events

Figure 3.1 shows two sample events e_1, e_2 with various occurrences at the points in time t_1 to t_6. Note that event occurrences are comparable both for the same event as well as distinct events. With regard to figure 3.1, we can for example say that the occurrence of event e_1 at t_2 happened before that of e_2 at t_3 or that at t_5, the events e_1 and e_2 happened at the same time. To be able to properly refer to these relations, we define a preorder for sets of event occurrences of multiple events and a total order for sets of event occurrences of a single event.

Definition 3.3 (Preorder over Event Occurrences). Let $\preceq_t$ be a preorder over a set of event occurrences (e, t) where e may be any event and $t \in \mathbb{R}^+$ is a timestamp. Assuming any such set O, the preorder $\preceq_t$ exhibits the following properties:

- $(e_i, t_n) \preceq_t (e_j, t_m)$ holds true when $t_n \leq t_m$ holds true for any $(e_i, t_n), (e_j, t_m) \in O$
- $(e, t) \preceq_t (e, t)$ holds true for all $(e, t) \in O$ (reflexivity)
- $(e_i, t_n) \preceq_t (e_j, t_m)$ and $(e_j, t_m) \preceq_t (e_i, t_n)$ then $t_n = t_m$ for all $(e_i, t_n), (e_j, t_m) \in O$
- if $(e_i, t_n) \preceq_t (e_j, t_m)$ and $(e_j, t_m) \preceq_t (e_k, t_l)$ then $(e_i, t_n) \preceq_t (e_k, t_l)$ follows for all $(e_i, t_n), (e_j, t_m), (e_k, t_l) \in O$ (transitivity)

Definition 3.4 (Total Order over Event Occurrences). Let $\prec_t$ be a total order over a set of event occurrences (e, t) where e is one distinct event and $t \in \mathbb{R}^+$ is a timestamp. Assuming any such set O, the total order $\prec_t$ exhibits the following properties:

- $(e, t_n) \prec_t (e, t_m)$ holds true when $t_n < t_m$ holds true for any $(e, t_n), (e, t_m) \in O$
- $(e, t) \prec_t (e, t)$ holds true for all $(e, t) \in O$ (reflexivity)
- $(e, t_n) \prec_t (e, t_m)$ or $(e, t_m) \prec_t (e, t_n)$ holds for all $(e, t_n), (e, t_m) \in O$ (totality)
- if $(e, t_n) \prec_t (e, t_m)$ and $(e, t_m) \prec_t (e, t_n)$ then $t_n = t_m$ and thus $(e, t_n) = (e, t_m)$ for all $(e, t_n), (e, t_m) \in O$ (antisymmetry)
- if $(e, t_n) \prec_t (e, t_m)$ and $(e, t_m) \prec_t (e, t_l)$ then $(e, t_n) \prec_t (e, t_l)$ follows for all $(e, t_n), (e, t_m), (e, t_l) \in O$ (transitivity)
- the infimum is the element with the smallest timestamp such that $\inf O := (e, t_n) \in O$ with $t_n \leq t$ for any t in $(e, t) \in O$
- the supremum is the element with the largest timestamp such that $\sup O := (e, t_n) \in O$ with $t_n \geq t$ for any t in $(e, t) \in O$

Using these orders, we can say that in figure 3.1 the occurrences of e_1 fulfill the relation $(e_1, t_2) \prec_t (e_1, t_4) \prec_t (e_1, t_5)$ and that for example $(e_1, t_2) \preceq_t (e_2, t_3)$. Note that even in sets of occurrences of multiple different events, we can use the total order when only comparing occurrences of the same event.

In TADL2, events are defined using the simple statement `Event e {}`, where `e` is the name or descriptor of the event. These event definitions are the foundation of both event chains and real-time requirements and in each following snippet, we will assume the referenced events to have already been declared using this statement.

3.1.2 Event Chain

Event chains are a method to describe a course of events in the timing analysis of automotive software systems. They provide context and relation to the events, reacting to a *stimulus event*, following an event path and ending with a *response event*.

Definition 3.5 (Event Chain). We consider an event chain to be a totally ordered set $ec = (E, s, r)$ where

- E is a finite set of at least two distinct events and

- $s \in E$ is the stimulus event, and
- $r \in E$ is the response event.

Definition 3.6 (Total Order over Event Chains)**.** Let $\leq_{ec}$ be the total order over event chains. Assuming any event chain ec, the total order $\leq_{ec}$ exhibits the following properties:

- the stimulus s is the first element in the set E such that $s \leq_{ec} e$ for all $e \in E$
- the response r is the last element in the set E such that $e \leq_{ec} r$ for all $e \in E$
- $e \leq_{ec} e$ holds true for all $e \in E$ (reflexivity)
- $e_a \leq_{ec} e_b$ or $e_b \leq_{ec} e_a$ holds for all $e_a, e_b \in E$ (totality)
- if $e_a \leq_{ec} e_b$ and $e_b \leq_{ec} e_a$ then $e_a = e_b$ for all $e_a, e_b \in E$ (antisimmetry)
- if $e_a \leq_{ec} e_b$ and $e_b \leq_{ec} e_c$ then $e_a \leq_{ec} e_c$ follows for all $e_a, e_b, e_c \in E$ (transitivity)

When defining event chains, we assume E to be an ordered set of distinct events under the total order $\leq_{ec}$, such that for each event chain $ec = (E, s, r)$ with $E = \{e_1, \ldots, e_n\}$ the relation $e_1 \leq_{ec} \ldots \leq_{ec} e_n$ holds as well as $s = e_1, r = e_n$. Since s and r are implicitly given by the order of E, we will not explicitly declare them in the future and consider the definitions $ec = (E)$ and $ec = (E, s, r)$ to be equivalent.

Each occurrence of a stimulus event triggers the corresponding event chain. To trace a path of events through an event chain, we need an additional concept, describing a way from the occurrence of a stimulus event to the occurrence of a response event.

Definition 3.7 (Flow through an Event Chain)**.** Let $ec = \{e_1, \ldots, e_n\}$ be an event chain with $e_1 \leq_{ec} \ldots \leq_{ec} e_n$ and O a set of observed event occurrences. Each flow through the event chain consists of n elements from (e_1, t_1) to (e_n, t_n) where $t_1 < \ldots < t_n$ denote points in time. After an occurrence of (e_1, t_1) at any point in time, the *flow* of the event chain triggered by e_1 at t_1 is defined recursively as the following:

$$\mathrm{ef}_i(ec, t_1) := \mathrm{ef}_{i-1}(ec, t_1) \cup (e_i, t_i) \text{ with } (e_i, t_i) := \inf\left\{(e_i, t_j) \in O : t_j > t_{i-1}\right\} \text{ for } i \in [2, n]$$

When we denote the complete flow of an event chain, from the occurrence to the stimulus up to the occurrence of the response event, we will omit the parameter i. This means that we will consider $\mathrm{ef}(ec, t_1)$ and $\mathrm{ef}_n(ec, t_1)$ to be equivalent, for any occurrence of an event e at a point in time t_1.

Extending figure 3.1 by an additional event e_3, we can define the event chain $ec = \{e_1, e_2, e_3\}$. In figure 3.2 we can now see multiple occurrences of these events as well as three complete flows through the event chain ec:

$$\begin{aligned}
\text{ef}(ec, t_2) &:= \{(e_1, t_2), (e_2, t_3), (e_3, t_6)\}, \\
\text{ef}(ec, t_3) &:= \{(e_1, t_3), (e_2, t_5), (e_3, t_8)\}, \\
\text{ef}(ec, t_5) &:= \{(e_1, t_5), (e_2, t_7), (e_3, t_8)\}, \\
\text{ef}(ec, t_9) &:= \{(e_1, t_9), (e_2, t_{12}), (e_3, t_{13})\}, \\
\text{ef}(ec, t_{11}) &:= \{(e_1, t_{11}), (e_2, t_{12}), (e_3, t_{13})\}
\end{aligned}$$

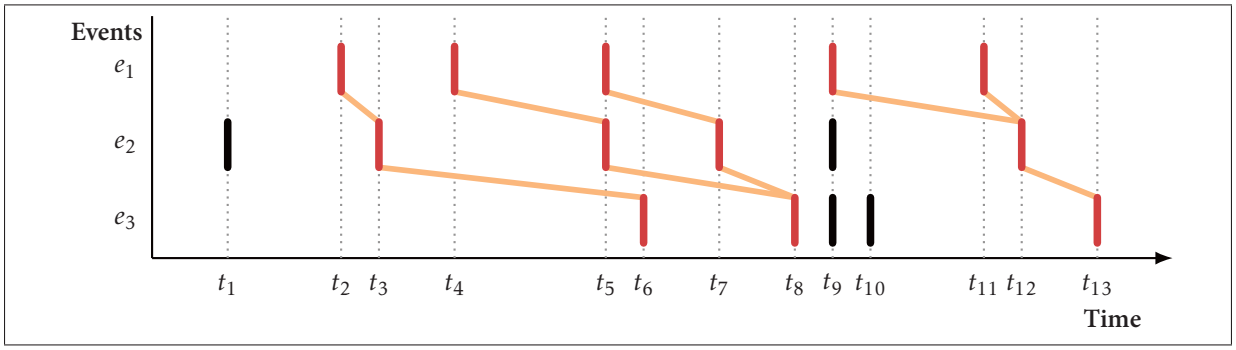

FIGURE 3.2: Several flows through $ec = \{e_1, e_2, e_3\}$ given sample occurrences of e_1, e_2, e_3

Since each occurrence in the flow needs to happen after the previous one, an event chain might be triggered or continued while multiple events happen at once, like at t_5 or t_{10} in figure 3.2, but at most a single event occurrence contributes to the flow then. Every flow of an event chain is unique and has a distinct occurrence of a stimulus event, but may share other events in the chain with other flows as seen with $\text{ef}(ec, t_3)$ and $\text{ef}(ec, t_5)$ sharing the same response occurrence (e_3, t_8). Note that each occurrence of a stimulus event *must* start a flow, but other events in the chain may happen in between, not being part of any flow; for example the occurrences of e_3 at t_9 and t_{10}.

In TADL2, event chains always consist of exactly one stimulus event and one response event. The event chain model in TADL2 is hierarchical, allowing the additional definition of segments to define a path of two-element event chains. Assuming we have already defined the relevant events as per the previous subsection, the definition of ec in TADL2 would look like shown in listing 3.1. Note that the hierarchical definition requires each segment to have the response event of the previous segment as its stimulus, otherwise the event chain definition would be invalid.

```
segA = EventChain {
    stimulus = e1,
    response = e2
}

segB = EventChain {
    stimulus = e2,
    response = e3
}

ec = EventChain {
```

```
        stimulus = e1,
        response = e3,
        segment = < segA, segB >
15  }
```

Listing 3.1: TADL2 model of the event chain $ec = \{e_1, e_2, e_3\}$

3.1.3 Function

Automotive software systems are not developed by defining and triggering events and event chains, but by implementing previously planned functions. The functions themselves usually describe what operations need to be performed, which input data is required for these calculations and which data is output at the end. It is then up to suppliers to implement these functions in a way that they exhibit the desired behavior. In the context of timing analysis, we do not need to know what or how functions implement this behavior and simply consider a function to be a high-level black-box concept.

Definition 3.8 (Function). We will consider a function $f = (start_f, finish_f)$ to be a special type of event chain consisting only of the stimulus event $start_f$ and the response event $finish_f$. The start and finish events of a function are in a 1:1–relationship, such that each $start_f$ must be followed by exactly one $finish_f$ event and no $finish_f$ may occur outside of a response.

When defining functions, we will consider the start and finish events to be explicitly given by the function name, such that the simple definition of a function f omitting the explicit events is equal to the definition $f = (start_f, finish_f)$ where f is the name of the function respectively.

Depending on the level of timing analysis performed, several different parts of a function implementation can be measured and examined. Figure 3.3 shows a sample of timing data that could be considered when doing a thorough timing analysis. For the sake of this thesis and the requirements covered in section 3.3, we will consider the f_{start} to occur at the start of the function execution time and the f_{finish} to be emitted at the end of the execution time. Referencing figure 3.3, the difference between the start and finish event of a function is equal to the *gross execution time*.

As functions are a more natural concept in describing the working of automotive software systems than event chains, we will from now on mostly deal with these. Since functions represent a special type of event chain, we will introduce a way to define event chains from given functions.

We will commonly use visualization of event chains defined from functions. An example for such an event chain $ec = \{f_1, f_2\}$ is shown in figure 3.5. Several flows through the function event chains f_1, f_2 as well as through this combined event chain $ec = \{f_1, f_2\}$ are depicted in figure 3.4, the connection between the two function event chains is highlighted separately.

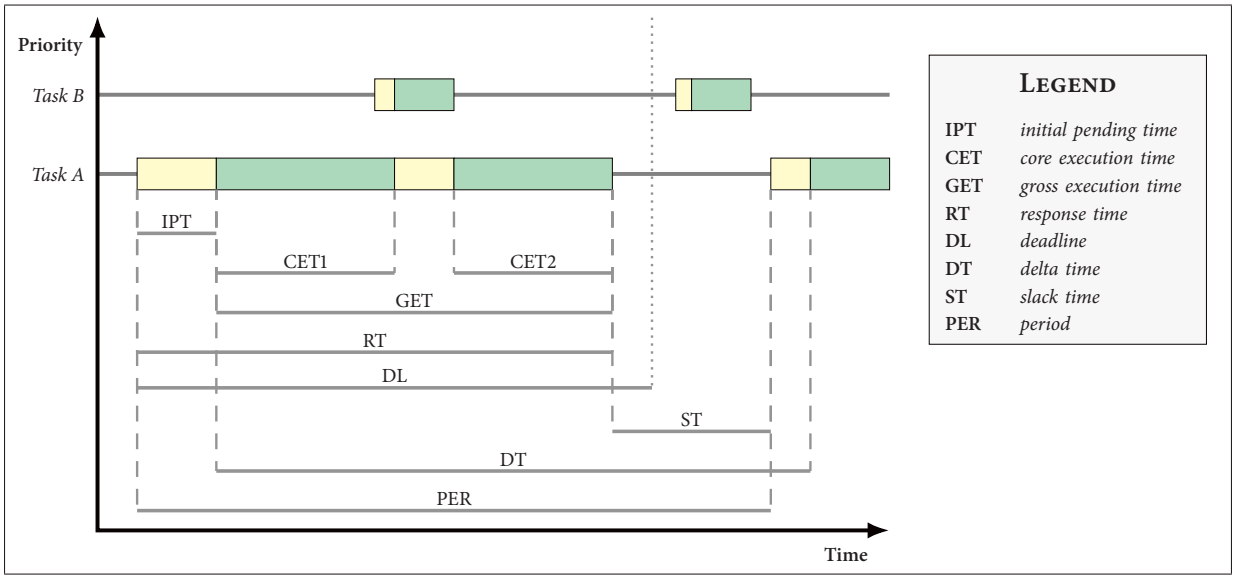

FIGURE 3.3: Information in intricate timing analysis of functions[1]

Definition 3.9 (Defining Event Chains from Functions). Considering an ordered set of implicitly defined functions $F = \{f_1, \ldots, f_n\}$, the event chain corresponding to this set is defined as

$$ec = \left(\bigcup_{f \in F} start_f \cup finish_f \right) = \left(\left\{ start_{f_1}, finish_{f_1}, \ldots, start_{f_n}, finish_{f_n} \right\} \right)$$

In this event chain, the start event of the first function is the stimulus and the finish event of the last function is the response. As such, $ec = (\{f_1, \ldots, f_n\})$ and $ec = \left(\{f_1, \ldots, f_n\}, start_{f_1}, finish_{f_2} \right)$ are equivalent.

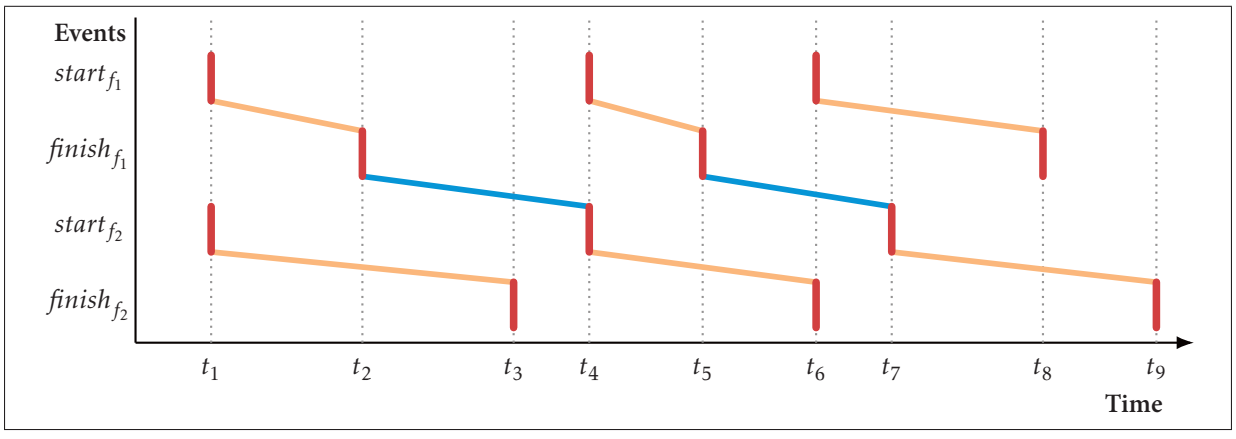

FIGURE 3.4: Flows through the event chain $ec = \{f_1, f_2\}$

In TADL2, the definition of this event chain does not only require the definition of the event chains representing f_1 and f_2, but also an additional chain to connect the response event of f_1 with the start event of f_2. A full example can be found in listing 3.2, assuming the events corresponding to f_1 and f_2 are already

[1] based on the graphical explanation of function timing in [26]

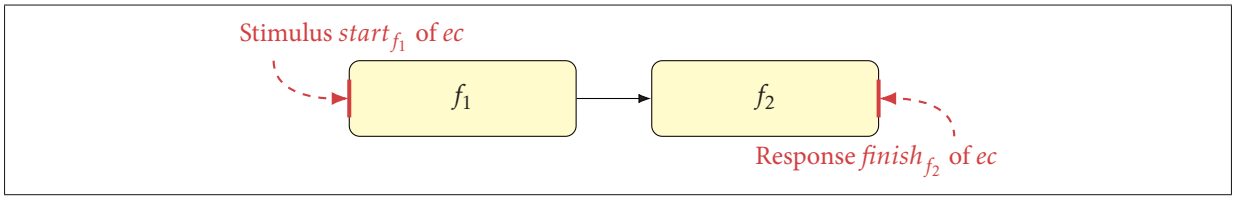

FIGURE 3.5: Visualization of the event chain $ec = \{f_1, f_2\}$

defined.

```
f1 = EventChain {
    stimulus = f1_start,
    response = f1_finish
}

f1_to_f2 = EventChain {
    stimulus = f1_finish,
    response = f2_start
}

f2 = EventChain {
    stimulus = f2_start,
    response = f2_finish
}

ec = EventChain {
    stimulus = f1_start,
    response = f2_finish,
    segment = < f1, f1_to_f2, f2 >
}
```

LISTING 3.2: Definition of the event chain $ec = \{f_1, f_2\}$ in TADL2

3.2 Properties of an Automotive Software System

The previously introduced definitions are conceptional and not part of an automotive software system. Any functions that were planned must be implemented, and in this step several limitations of the underlying system must be kept in mind; these concepts will be explained in this section.

Definition 3.10 (Processing Environment). We will consider a *processing environment* to be a tuple $pe = (T, \mathrm{TG}, TQ, \mathrm{c}, \mathbf{Sch})$ where

- T is a finite set of *tasks*,
- $\mathrm{TG} : T \mapsto \mathbb{R}^+$ is a function mapping each task in T to a period of time,
- TQ is the processing environment's *task queue*,
- c is the processing environment's *clock*, and
- **Sch** is a *scheduling function*.

Since we already introduced the concept of clocks in section 2.1 and are introducing another concept with the same name in the context of processing environments, it is important to properly differentiate between both; as such, clocks in the concept of processing environments are written in roman type, like c, and clocks in terms of timed automata in italic type, like c.

A processing environment serves as a basic representation of an automotive software system to run functions on, implemented as tasks. We assume a processing environment to be able to handle exactly one execution at a time.

3.2.1 Tasks

Tasks represent a realization of the function concept explained in section 3.1.3. We consider each task to be an implementation of exactly one function.

Definition 3.11 (Task). We define a *task* as tuple $\tau = (f, BCET, WCET, A)$ where

- f is a single function that is implemented by the task,
- $BCET \in \mathbb{R}^+$ is the *best-case execution time* of the task,
- $WCET \in \mathbb{R}^+$ is the *worst-case execution time* of the task with $WCET \geq BCET$,
- A, the task's scheduling attribute.

We will call any task $\tau = (f, BCET, WCET, A)$ an *implementation* of the function f.

We differ between the concept of a task, representing the implemented function on an automotive software system, and a task instance, which we consider to be an actual executable entity on a processing environment.

Since tasks are implementations of functions, we can specify bounds for the execution time of their corresponding task instances. In function context, the execution time of the task instance represents the time between the start event and the finish event of the implemented function. We will call the upper bound the *worst-case execution time (WCET)* and the lower bound the *best-case execution time (BCET)*; both are dependent on the implementation of the task as well as the underlying system and are fixed values given in the definition of the task. Due to this, we will consider tasks unique to a processing environment, and each processing environment is required to use its own, distinct set of tasks.

The scheduling attribute is used by the scheduling function **Sch** on the level of the process environment to determine the next task to execute; examples for schedulers and relevant scheduling attributes are given in section 3.2.4.

Unless otherwise noted, we will consider tasks to implement the functions defined by the same index, such that the tasks $\tau_1, \ldots, \tau_n$ implement the functions $f_1, \ldots, f_n$ respectively.

Definition 3.12 (Task Instance). We will consider a *task instance* to be a tuple $i = (\tau, s, et)$ where

- τ is the task of which i is an instance,
- $s \in \mathbb{R}_0^+$ is the the *start time* of the task instance, and
- $et \in [0, WCET_\tau]$ is the *execution time* of the task instance.

Since each task instance is bound to exactly one task τ we say that the task *executes* the function defined by f_τ. As such, the start of the execution of a task instance triggers the event start_f and the end of its execution triggers finish_f.

The execution time of a task instance is the amount of time that a processing environment spends executing this specific task instance. It is mainly determined by scheduling and will be covered in more detail in section 3.2.4.

3.2.2 Clock

In the context of real-time systems like automotive software systems, each system usually has its own clock to keep track of time. This concept is now formalized and applied to processing environments.

Definition 3.13 (Clock)**.** We consider a *clock* c to be the timekeeper of a processing environment. For a processing environment with a clock c, $t_c \in \mathbb{R}_0^+$ will define the *system time*.

Since each processing environment keeps track of its own local time, there might be differences in the time between various processing environment, resulting in an offset.

Definition 3.14 (Clock Offset)**.** We will call the difference between a clocks c_1 and another clock c_2 the *clock offset* between the clocks c_1 and c_2, defined as $\text{co}(\text{c}_1, \text{c}_2) = t_{\text{c}_2} - t_{\text{c}_1}$. For any c_1, c_2 the offset is symmetrical, that is $\text{co}(\text{c}_1, \text{c}_2) = -\text{co}(\text{c}_2, \text{c}_1)$ holds. Additionally, if $\text{c}_1 = \text{c}_2$ then $\text{co}(\text{c}_1, \text{c}_2) = -\text{co}(\text{c}_2, \text{c}_1) = 0$.

Since each processing environment has exactly one clock, we will also use $\text{co}(pe_1, pe_2)$ to denote the clock offset between any two processing environments pe_1, pe_2, which is equivalent to $\text{co}(\text{c}_{pe_1}, \text{c}_{pe_2})$.

Definition 3.15 (Reference System)**.** For any set PE of processing environments, a *reference system* is defined as a processing environment pe_i such that $\text{co}(pe_i, pe) \geq 0$ holds true for all $pe \in PE$.

As such, a reference system in a set of processing environments is a processing environment for which all clock offsets are non-negative. It will be used as a more accessible way of denoting clock offsets for larger systems, since all clock offsets can be declared starting at 0 and with a non-negative offset.

3.2.3 Time Grid

The time grid provides a timing-oriented approach to triggering tasks within a processing environment, with each task being assigned a fixed period in which an instance of it is added to the processing environment's task queue. In the definition of processing environments we defined the function TG, which assigns each task τ on a processing environment to a period of time $\rho \in \mathbb{R}^+$ in which it is supposed to be repeated. Tasks and time grids are in a one-to-many relationship, which means that each task must be assigned exactly one period, but multiple tasks can run in the same period. When inspecting properties of distributed functions on multiple systems, the time grid of a processing environment is dependent on that system's clock, so the clock offset is relevant and might shift the start of triggered tasks when seen in comparison.

In figure 3.6 we can see an example of two tasks sharing the same BCET, WCET and period, running on different systems with a slight clock offset. The tasks are defined as $\tau_1 = \{f_1, 6, 6, 1\}, \tau_2 = \{f_2, 6, 6, 1\}$

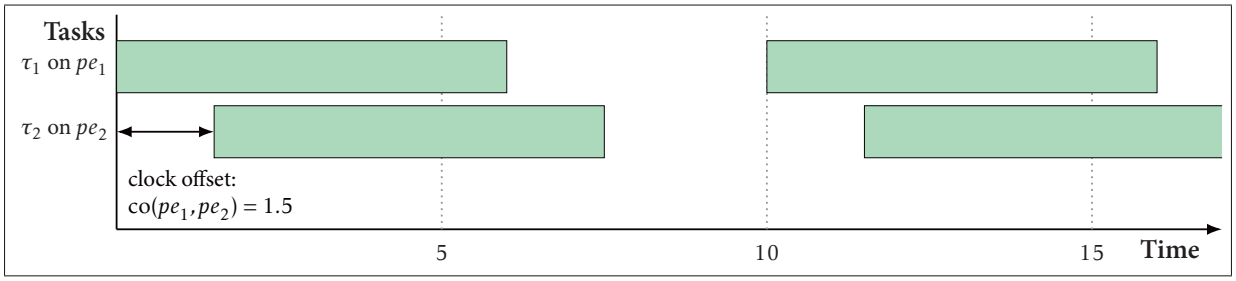

FIGURE 3.6: Two tasks running in the same time grid on different processing environments with a slight clock offset

and $TG(\tau_1) = TG(\tau_2) = 10$. Although both systems are running the same task in the same time grid, the task instances start and end at different points in time, each separated by the clock offset between the two systems. Note that in figure 3.6, the reference system is pe_1.

3.2.4 Scheduling

In real-time systems, a scheduler is used to both manages the task queue and to choose which task in the task queue shall be executed next, based on its scheduling strategy.

Definition 3.16 (Scheduler)**.** Each processing environment must have exactly one scheduling function **Sch** that, given a non-empty task queue TQ as input, returns exactly one task instance $i \in TQ$ chosen based on the scheduling strategy.

We will only cover fully deterministic scheduling functions, such that for any task queue TQ, a scheduling function applied to it always returns the same task instance i.

Any task instance in the queue is in exactly one of the following four states at any given time:

- *ready*: time spent after being added to the queue, but before the first execution
- *executing*: the instance is actively being executed by the system; each processing environment can have at most one task instance in this state at any given point in time
- *suspended*: after already being executed, the task is temporarily suspended so that another task can enter the *executing* state
- *done*: the instance was finished and will be dequeued; since we consider dequeuing to be an instantaneous process, no time is spent in this state

The states of a task instance and possible transitions are depicted in figure 3.7. Both scheduling strategies introduced in the following two subsections are *preemptive* scheduling strategies, which means that a task instance that is currently being executed might be temporarily suspended to allow for the execution of another task instance. The transitions marked with *start* and *finish* are the transitions in which the corresponding events of the implemented function occur.

Each processing environment has exactly one task queue TQ, which is filled by the time grid triggering tasks. Defining the state of a task queue at any given time requires a discrete time model and due to the varying execution time, the state of a task queue is non-deterministic even for a deterministic scheduling

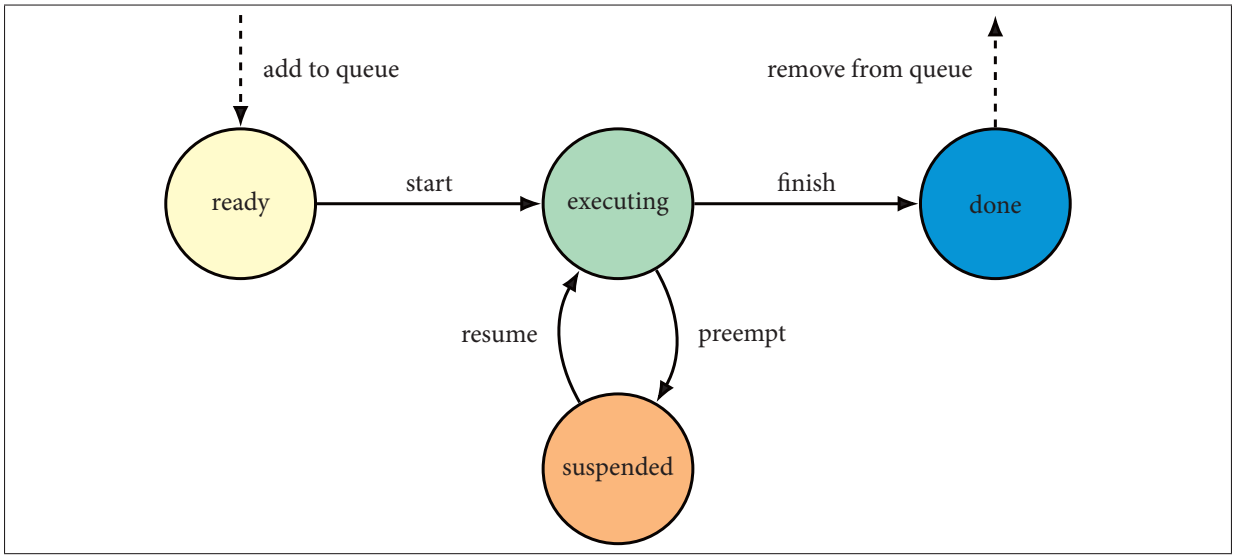

FIGURE 3.7: States of a task instance and available transitions in between

function. A formal approach to the definition of a task queue will be given in section 4.1, where preparations for the model are detailed.

OSEK Scheduling

A common scheduling strategy in automotive systems is that of OSEK OS, a real-time operating system aimed at distributed embedded control units with special support for automotive requirements[39]. The OSEK Scheduler is priority-based; here we will introduce preemptive OSEK scheduling, which allows instances of tasks with a higher priority to interrupt lower-priority ones. Since the priority of an OSEK task is given during task definition and does not change with time, OSEK scheduling is considered to be a *static* scheduling strategy[15].

Definition 3.17 (OSEK Scheduling). We consider an *OSEK Task* a tuple $\tau = (F, BCET, WCET, P)$, where the scheduling information $P \in \mathbb{N}$ is the unique *task priority*. No two tasks on the same processing environment may be assigned the same priority.

An *OSEK Task Instance* is a tuple $i = (\tau, s, et)$, where τ is an OSEK task. The OSEK scheduling function **OSEK** is defined over a task queue of OSEK instances as $\mathbf{OSEK}(TQ) = \min_s(\max_{P_\tau}(TQ))$.

An OSEK scheduler always selects the task instance of a task with the highest priority P from a given queue. Should multiple instances of the same task be in the queue, the task instance with the lowest start time is selected. Since no two instances of the same task can be added to the queue at the same time, the OSEK scheduling function is deterministic by definition.

In figure 3.8 we see an example of an OSEK scheduled processing environment with three tasks. Task $\tau_1 = \{f_1, 2, 2, 3\}$ with $\mathrm{TG}(\tau_1) = 8$ has the highest priority and is always directly executed, spending no time in either the ready state nor in the suspended state. The task $\tau_2 = \{f_2, 3, 3, 2\}$ with $\mathrm{TG}(\tau_2) = 6$ has a lower priority than τ_1, but is often executed directly as well, sometimes waiting for an instance of τ_1; in the inspected time frame, only a single instance of τ_2 enters the suspended state. As a typical example of a low-priority task, $\tau_3 = \{f_3, 5, 5, 1\}$ with $\mathrm{TG}(\tau_3) = 20$ rarely spends time being executed; although it has

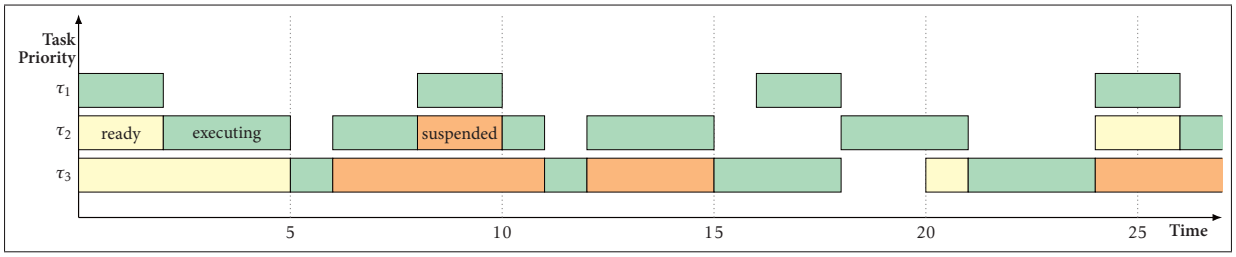

FIGURE 3.8: Preemptive OSEK scheduling of three tasks on a system

an execution time of just 5 time steps, the first instance of it finishes after spending a total of 18 time steps in the queue.

EDF Scheduling

A more recent trend in real-time systems is the application of the *earliest deadline first (EDF)* scheduling strategy, which selects the tasks from the queue which are closest to their deadline to be run next[42, 51]. Although each task is assigned a fixed relative deadline, scheduling is done based on the absolute deadline of each task instance, computed from the start time of the instance and the relative deadline of the task. As such, EDF scheduling is considered to be a *dynamic* scheduling strategy, since the final scheduling parameter is only available during runtime and not beforehand[15].

Definition 3.18 (EDF Scheduler). We consider an *EDF Task* a tuple $\tau = (F, BCET, WCET, D)$, where the scheduling information $D \in \mathbb{R}^+$ is a *relative deadline*. The deadline denotes that a successful execution of the task instance must have happened D time units after an instance of the task has been queued. If this rule is violated and can't be guaranteed, the system is considered to be *non-schedulable* with the current scheduling parameters.

An *EDF Task Instance* is a tuple $i = (\tau, s, et)$, where τ is an EDF task.. The EDF scheduling function **EDF** is defined over a task queue of EDF instances as $\mathbf{EDF}(TQ) = \min_s(\min_d(TQ))$, where $d = s + D_\tau$ is the *absolute deadline* of the task instance.

An EDF scheduler always selects the task instance with the nearest absolute deadline. Should multiple task instances with the same absolute deadline be in the queue at the same time, the scheduler selects the instance that was first added to the queue. While we are unable to exactly formalize this without a more formal definition of the task queue, it is covered in chapter 4.

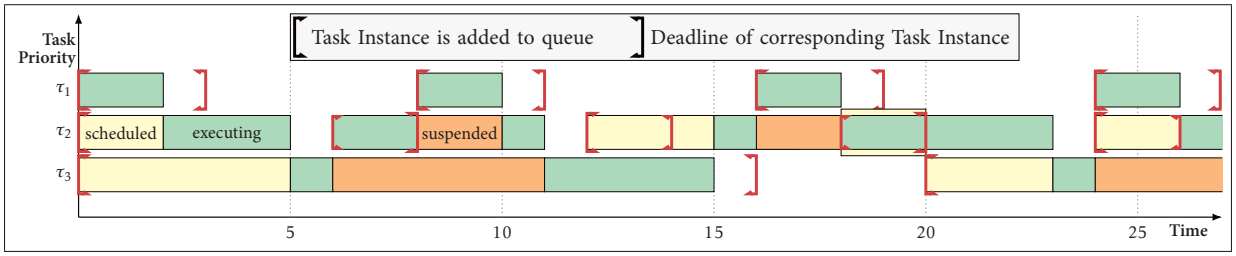

FIGURE 3.9: EDF scheduling of three tasks on a system

In figure 3.9 we see an example of EDF scheduling with tasks sharing the same BCET, WCET and period of those from figure 3.8, but with different scheduling parameters. The task $\tau_1 = \{f_1, 2, 2, 3\}$ has a relative deadline of 3, which indicates a high-priority, and its instances are thus always directly executed after being added to the queue, at least in the time frame shown. Task $\tau_2 = \{f_2, 3, 3, 8\}$ has a relative deadline of 8 but a period of 6, which means that there could be multiple instances of it in the queue at the same time; in the figure, this is the case in the time steps 18 to 20, indicated by the larger, second block in the background row. To easily identify the corresponding add and deadline markers, each odd pair for τ_2 is considerably smaller. The task $\tau_3 = \{f_3, 5, 5, 16\}$ is again an example of a low-priority task, but in contrast to OSEK scheduling, it is executed in the time frame from step 12 to 15, although an instance of τ_2 is already in the queue. This highlights the dynamic nature of EDF scheduling as well as an important feature, as this strategy is effectively avoiding *task starvation* – a problem with static scheduling strategies, where low-priority tasks are not or only very rarely executed due to high-priority tasks preempting them – and other problems of static scheduling strategies.

3.3 Real-Time Requirements

In this section several different types of requirements demanded from real-time systems are explained. All requirements are imposed on the concept level, over functions and event chains – the fulfillment of these requirements are up to their actual implementations, the tasks implementing these functions. This means that while we define each requirement on functions, we will later verify them on the tasks implementing these functions, running on processing environments.

3.3.1 Maximum Execution Time

The maximum execution time is a real-time requirement defining a limit for the execution time of a single function.

Definition 3.19 (Maximum Execution Time of a Function). For any function f we can define a *Maximum Execution Time* $\mathrm{MET}(f)$, which means that each task instance of a task implementing the function must finish within the specified time.

In other words, the maximum execution time is a constraint that specifies the maximum amount of time that may pass between an occurrence of the start event and the corresponding finish event for a function. Figure 3.10 shows how an example of how this constraint may be violated for a function f_1 with $\mathrm{MET}(f_1) = 4$.

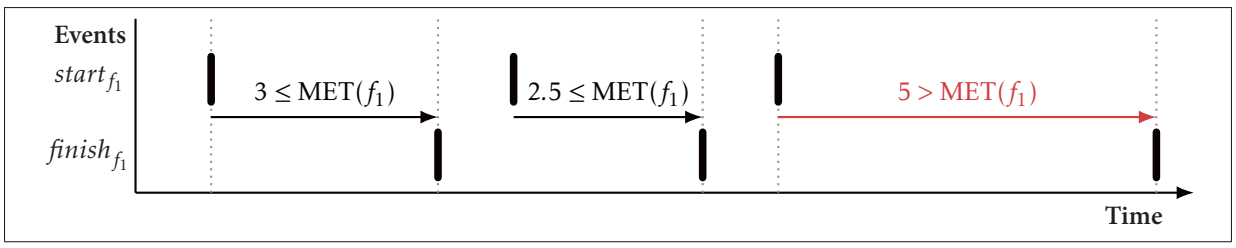

Figure 3.10: Events to consider when dealing with a function's maximum execution time

In TADL2, the maximum execution time is denoted using a *DelayConstraint*. Such a constraint defined

the time bounds in which a target event must happen after a source event occurred and when applied to the start and finish event of a function, specified it's MET.

In addition to the upper bound, a lower bound can be assigned to define a minimum execution time. Assuming the lower bound to have the default value, which is zero, we are able to define the requirement as shown in figure 3.10 using the TADL2 code listing 3.3

```
DelayConstraint met_f1 {
    source f1_start,
    target f1_finish,
    lower = 0,
    upper = (4 ms on universal_time)
}
```

Listing 3.3: DelayConstraint to define MET(f_1) = 4 ms

3.3.2 Maximum Reaction Time

The maximum reaction time is a real-time requirement defining a limit for the time of a full flow through an event chain.

Definition 3.20 (Maximum Reaction Time of an Event Chain). For any event chain *ec* we can define a *Maximum Reaction Time* MRT(*ec*), which declares the maximum time that may pass between the occurrence of a stimulus event at the start of a flow and the corresponding occurrence of the response event at the end of the flow.

In TADL2, a *ReactionConstraint* defines bounds for the time after the stimulus in which the corresponding response in an event chain must occur. It is similar to the *DelayConstraint* but includes the causal relation given by the event chain and take an event chain reference as scope rather than two single events. An example of an event chain flow was given in figure 3.5, to impose an MRT restriction upon the shown event chain and all valid flows using TADL2, one would proceed as shown in listing 3.4.

```
ReactionConstraint mrt_ec {
    scope ec,
    maximum = (20 ms on universal_time)
}
```

Listing 3.4: ReactionConstraint to define MRT(*ec*) = 20 ms

For a distributed function in an automotive software system, the maximum reaction time might be dependent on a large number of parameters, including the period of the tasks implementing the function, scheduling and variable runtimes as well as offset of processing environments. Figure 3.11 gives a glimpse of how complex the timing of event chains can get, even with just four systems without any offset, running only a single task each. This will be covered in more depth in section 5.2.2, where we will detail the verification of the MRT requirement and explain several dependencies based on the model developed in chapter 4.

3.3.3 Periodicity

Closely related to the time grid of a task is the periodicity requirement.

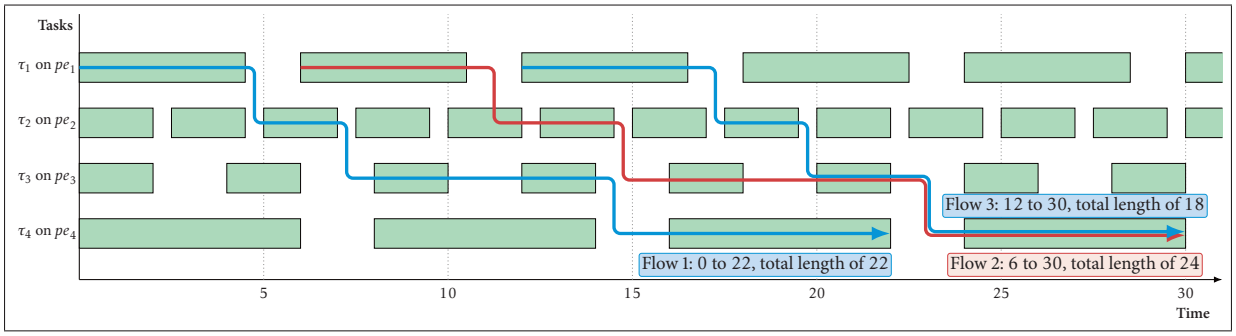

FIGURE 3.11: Reaction time of event chain flows using a path through $\tau_1, \tau_2, \tau_3, \tau_4$

Definition 3.21 (Periodicity of a Function). For any function f we can define a *Periodicity* requirement $\mathrm{PER}(f)$, declaring the maximum time considered to be valid between each occurrence of the finish event of f.

The periodicity requirement defines constraints for the time between two occurrences of the finish event of the corresponding function. In figure 3.12, occurrences of the start and finish events of a function f_1 are depicted. The requirement $\mathrm{PER}(f_1) = 5$ is imposed upon this function, such that figure 3.12 shows a valid as well as an invalid distance between the finish times of the function.

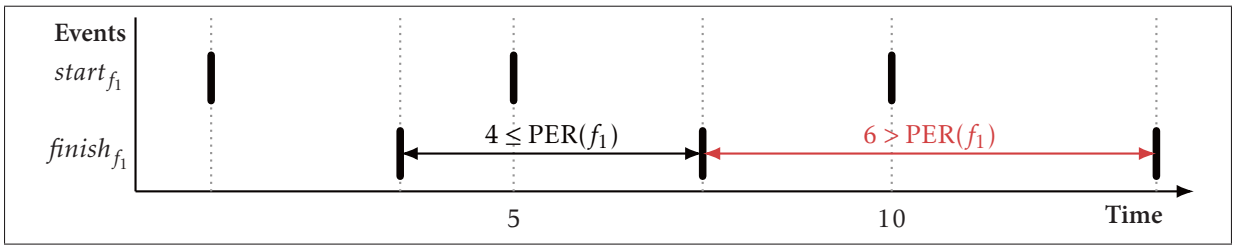

FIGURE 3.12: Visualization of periodicity based on events

In TADL2, the requirement can be described using the *RepeatConstraint*, which expresses the constraint of an event repetition inside given bounds, an example for $\mathrm{PER}(f_1) = 4$ is shown in listing 3.5. Apart from the event on which the constraint is imposed upon, the lower and upper bounds, as well as the span can be declared. The latter allows to not only specify that every occurrence of the event has to be periodic within the bounds, but that during the given time frame the designated amount of repetitions need to occur, without explicitly imposing restrictions upon the single occurrences. We will assume both the lower bound and the span to have their default values, which are 0 and 1 respectively, such that the *RepeatConstraint* can be used for the periodicity requirement as defined here.

```
RepeatConstraint per_f1 {
    event f1_finish,
    lower = 0,
    upper = (4 ms on universal_time),
    span = 1
}
```

LISTING 3.5: TADL2 snippet to declare $\mathrm{PER}(f_1) = 4\,\mathrm{ms}$

Figure 3.13 shows how the periodicity requirement might be violated on an automotive software system, considering low-priority tasks running in a tight time grid. The example shows of a single, OSEK-scheduled processing environment with $\tau_1 = (f_1, 4, 4, 3), \tau_2 = (f_2, 3, 3, 2), \tau_3 = (f_3, 1, 1, 1)$ within the time grid $\mathrm{TG}(\tau_1) = 12, \mathrm{TG}(\tau_2) = 10, \mathrm{TG}(\tau_3) = 5$. Although τ_3 has an execution time of only 1 time step, its execution is delayed for a long time by the other two tasks, so much that there often are two instances of τ_3 in the queue. When these execute in bursts they leave a lot of time between the occurrences, resulting in a violation of the example requirement $\mathrm{PER}(f_3) = 8$.

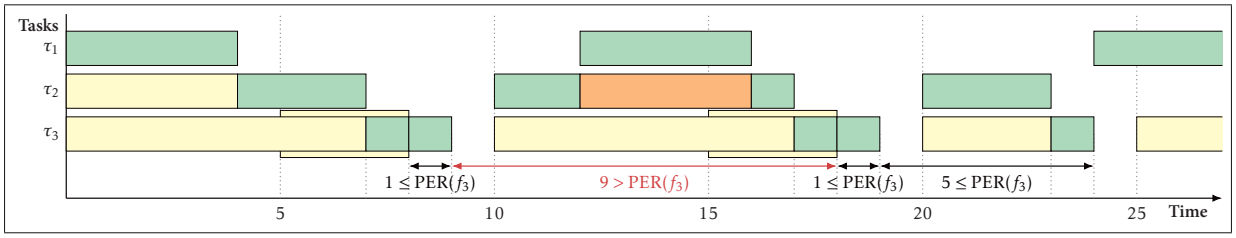

FIGURE 3.13: Violation of the periodicity requirement in a system with three tasks

3.3.4 Maximum Data Age

To set constraints relating to data that might travel between two functions, a maximum data age requirement may be specified.

Definition 3.22 (Maximum Data Age between a Function Pair). For any pair of functions f_1, f_2 we can define a *Maximum Data Age* $\mathrm{MDA}(f_1, f_2)$, declaring the maximum time that may have passed since the last occurrence of $finish_{f_1}$ when the event $start_{f_2}$ occurs.

Considering events, the maximum data age requirement represents an inverted constraint, such that the second event for which the constraint is imposed upon, $start_{f_2}$, triggers the verification and the time since the last occurrence of $finish_{f_1}$ is calculated. This means that the event $finish_{f_1}$ may occur multiple times in between and when $start_{f_2}$ occurs, the most recent occurrence is used for comparison against the constraint; an example is given in figure 3.14.

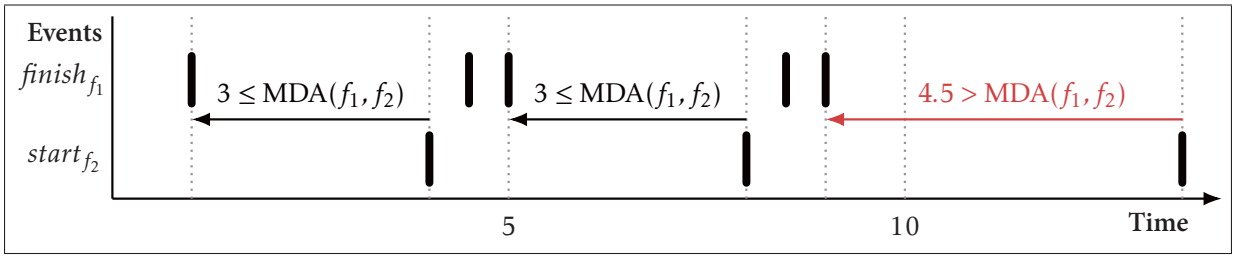

FIGURE 3.14: The maximum data age constraint visualized in event context with $\mathrm{MDA}(f_1, f_2) = 4$

To represent the MDA requirement in TADL2, we use an *AgeConstraint*, which imposes in which time frame before a response in an event chain the corresponding stimulus must have occurred – it can be imagined as an inverted *ReactionConstraint*. The example from figure 3.14 is specified as a TADL2 constraint in listing 3.6.

```
da_f1_f2 = EventChain {
    stimulus = f1_finish,
    response = f2_start
}

AgeConstraint mda_f1_f2 {
    scope = da_f1_f2
    minimum = 0
    maximum = (4 ms on universal_time)
}
```

Listing 3.6: TADL2 *AgeConstraint* to express $\mathrm{MDA}(f_1, f_2) = 4\,\mathrm{ms}$

In automotive software systems, the MDA denotes the maximum amount of time since the last finish of a task τ_1 implementing f_1 when a task τ_2 implementing f_2 is started. It is a way to logically define the dependency on recent information, read by f_2 and written by f_1. In figure 3.15 we have two example processing environments each with a single task running on them. Considering τ_1, τ_2 to implement the functions f_1, f_2 respectively, we can impose the maximum data age requirement $\mathrm{MDA}(f_1, f_2) = 8$ upon them and see a violation after just fifteen time steps.

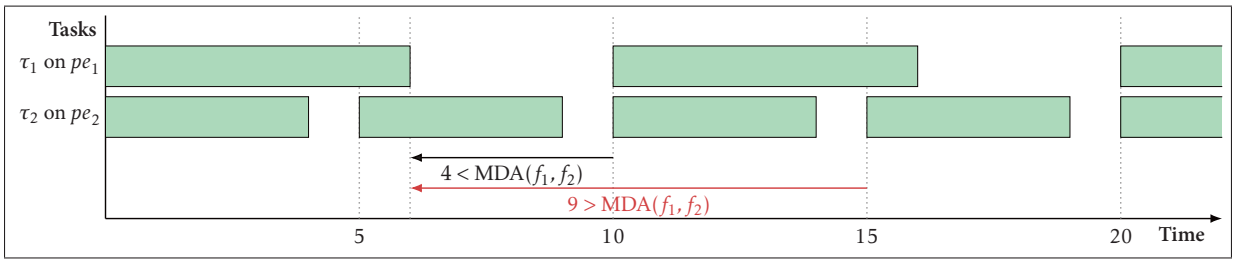

Figure 3.15: Two tasks showing a maximum data age violation

3.3.5 Synchronization

By imposing a *synchronization* requirement upon several functions $f_1, \ldots, f_n$ we require them to always finish close to each other.

An example of the synchronization constraint on the functions f_1, f_2, f_3 is given in figure 3.16, where the constraint $\mathrm{SYNC}(f_1, f_2, f_3) = 3{,}5$ is shown. After a first occurrence of one of the three finish event, the time frame in which the other two finish events need to occur is highlighted. These time frames may overlap as seen in the time steps 6 to 6,5 as long as in each time frame, the required events occur. A violation of the synchronization requirement is shown near the end of the depicted time, with the finish event of f_2 occurring only after the required time frame.

Definition 3.23 (Synchronization of a Set of Functions). For a set of two or more functions $f_1, \ldots, f_n$ we can define a *Synchronization* requirement $\mathrm{SYNC}(f_1, \ldots, f_n)$, which specifies that after an occurrence of one finish event from a function in the set, one finish event of all other functions in the set must occur within the given time frame.

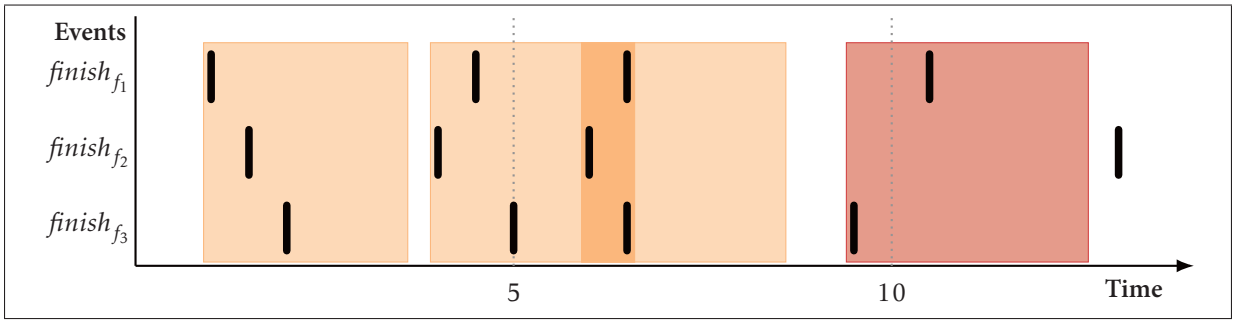

FIGURE 3.16: Finish events of the functions f_1, f_2, f_3 with $\text{SYNC}(f_1, f_2, f_3) = 3.5$

The TADL2 *SynchronizationConstraint* describes how closely several events need to occur together. During the defined tolerance frame, all events need to occur at least once; the TADL2 constraint for the requirement visualized in figure 3.16 can be found in listing 3.7.

```
SynchronizationConstraint f1_f2_f3_sync {
    events f1_finish, f2_finish, f3_finish
    tolerance = (3 ms on universal_time)
}
```

LISTING 3.7: Specification of $\text{SYNC}(f_1, f_2, f_3) = 3.5$ using TADL2

Having two systems with two different tasks each, figure 3.17 shows how the periodicity requirement might be violated in automotive software systems, taking system offset and scheduling into account and using the very strict constraint $\text{SYNC}(f_2, f_4) = 1$.

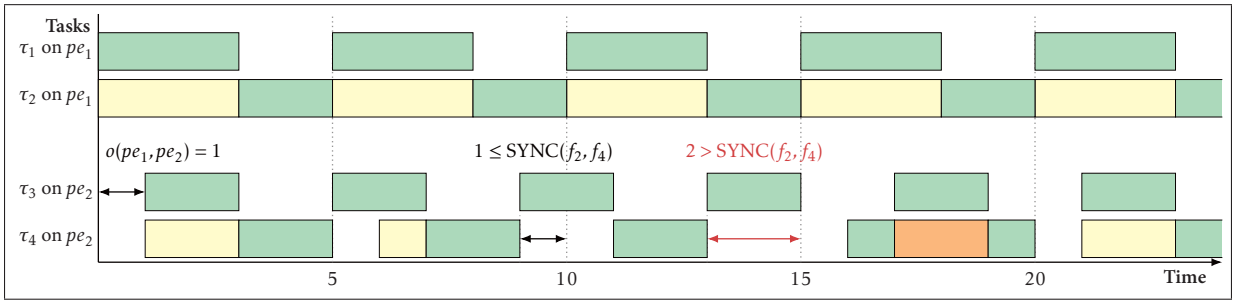

FIGURE 3.17: Violation of the synchronization constraint between two tasks on different processing environments with a slight offset

3.3.6 Arithmetically Detecting Inconsistencies

There are a couple of inconsistencies in real-time requirements that can be checked for using simple arithmetic comparisons, ignoring scheduling and an actual execution model of the processing environments. For the requirements detailed in section 3.3.1 to section 3.3.5, these trivial inconsistencies include the following:

- $\text{MET}(f) < WCET_\tau$: if the worst-case execution time of a task τ implementing a function f is larger than the maximum execution time defined as a requirement over f, the requirement cannot always be met and is infeasible; a simple counter-example to prove this is a task instance with the WCET as

execution time.

- $\mathrm{MRT}(\{f_1, \ldots, f_n\}) < \sum_{\tau \in \{\tau_1, \ldots, \tau_n\}} WCET_\tau$: if the maximum response timed of an event chain consisting of the functions $f_1, \ldots, f_n$ is smaller than the sum of worst-case execution time of the tasks $\tau_1, \ldots, \tau_n$ implementing these functions, the requirement cannot be met and is considered to be infeasible. Even if all tasks are running on the same processing environment with priorities defining the exact order as defined in the event chain, the requirement is violated when all task instances have the WCET as execution time.
- $\mathrm{PER}(f) < \mathrm{TG}(\tau)$: if the periodicity requirement of a task is smaller than the task's period, the requirement cannot be met and is inconsistent; it suffices to observe two task instances of such a task to prove this.
- $\mathrm{PER}(f) < WCET_\tau$: if the periodicity requirement of a task is smaller than the task's WCET, the requirement cannot be met and is considered to be inconsistent. Even if the period assigned to the task is equal to or larger than the requirement specified, the period is larger than the task's WCET, meaning that instances of the task just fill up the queue when instances of the task consistently finish with the WCET as execution time.
- $\mathrm{SYNC}(f_1, \ldots, f_n)$ and $\mathrm{TG}(\tau_a) \neq \mathrm{TG}(\tau_b)$ for any $\tau_a, \tau_b \in \{\tau_1, \ldots, \tau_n\}$: when a synchronization requirement is imposed upon a set of functions $f_1, \ldots, f_n$, all tasks $f_1, \ldots, f_n$ implementing these functions must run in the same time grid, otherwise the requirement is considered to be infeasible. If at least one task is running in another time grid, the instances of this task will get out of sync eventually.

The detection of more intricate feasibility issues requires at least basic modelling of the behavior of automotive software systems. We will develop such a model in chapter 4 using timed automata, based on the concepts introduced in section 3.2, and show how to use this model to identify unrealizable requirements in chapter 5.

4 Modelling Processing Environments Using Timed Automata

In this chapter we build a model of timed automata which will be used to provide a base for the verification. We start with preparations in section 4.1, introducing preparations for the model-building process that will be used throughout this chapter.

In section 4.2 we introduce a generic model of a processing environment to simulate the behavior of automotive software systems in regard to timing. This model is refined in section 4.2.2 to provide additional timing information to be used in the verification. Having established the formal model, we will transfer it to UPPAAL in section 4.3 and explain limitations that arise from the use of the tool.

During the course of this chapter, we will frequently use $:=$ to denote assignments to avoid confusion when $=$ is used for comparisons in the context of transition guards.

4.1 Preparations

As stated in section 1.2, we require a simulation of system behavior in regard to timing to be able to detect most inconsistencies in requirements. This means that we want to build a general model of a processing environment with which we can simulate the behavior of automotive software systems, including task execution and scheduling.

Real-time systems are discrete systems, which means that the number of states such a system can be in is finite. Discrete systems spend a certain time in a state and then move on to the next state and, due to the number of states being finite, can be represented as a finite state machine. In real-time systems, clocks operate on a tick-based basis, triggering periodically with a fixed delay between each trigger, called the *tick*.[30]

This allows us to define a discrete-time model of a processing environment using a timed automaton, for which we also define such a tick. Clock constraints in timed automata, that are used for both guards and location invariants, are required to be in $\mathbb{N}$ as defined in definition 2.1. We define the tick rate as $tick := 1$, since this allows us to have a common base for time that is both easy to understand and versatile. Any given best-case and worst-case execution time of tasks, which now also need to be in $\mathbb{N}$, can be divided by this tick rate.

Given this tick rate, we will define that a processing environment executes a task instance for exactly one tick before it determines the task instance to execute next. This effectively means that every tick, the scheduler chooses a task instance based on its scheduling strategy, enabling preemptive scheduling.

In addition to a clock c counting the system time, each automaton will have an additional clock tc with $v(tc) \in [0, tick]$. This *tick clock tc* will count the time up to each tick, and will be reset on a transition into the base location of the automaton model, enabling us to restrict the time of each action outside of this location to exactly one tick.

This discrete, tick-based time model allows us to recursively define the task queue dependent on both the system and the time as $TQ(pe, t)$ based on a start state TQ_0:

$$TQ(pe, 0) := TQ_0,$$
$$TQ(pe, tick\ n) := TQ(pe, tick\ (n-1)) \cup add \setminus remove$$

In this recursive definition, *add* is the set of elements to be appended to the task queue compared to the previous tick and *remove* is the set of elements to be removed since then. Note that this still does not allow us to deterministically determine the state of a task queue at any given time. As mentioned in section 3.2.4, the state of the task queue at any given time is non-deterministic even for a deterministic scheduling function, as long as at least one task with a variable runtime ($BCET_\tau \neq WCET_\tau$) is part of the processing environment.

For a processing environment *pe* we define the macro *NT* representing new task instances to be added to the queue. This macro is an essential part of the *add* set in each step, since it ensures that all tasks are triggered based on their assigned time grid, and corresponding task instances are added to the queue. We define it as the following:

$$NT := \left\{(\tau, v(c), 0) : v(c) \bmod \mathrm{TG}(\tau) = 0, \tau \in T_{pe}\right\}$$

This macro is deterministic, since the time grid of tasks is fixed. At any point in time given by the automatons internal clock c, it contains the set of task instances corresponding to tasks for which $v(c) \bmod \mathrm{TG}(\tau) = 0$ is true, initializing these instances with $s := v(c)$ and $et := 0$. This set is added each tick and simulates time-triggered, periodic task activations based on the time grid. Note that for $v(c) = 0$ this includes an instance of every task in the system, since $0 \bmod n = 0$ is true for any n.

Since we have already decided that the execution of a task instance is simulated for exactly one tick before rescheduling, we can define this execution as part of the *add* and *remove* macro. Assuming there is a task instance $i = (\tau, s, et), i \in TQ(pe, v(c))$ being executed. When we want to simulate the execution for one tick while keeping the instance in the queue, we define the task queue for the next tick as the following:

$$TQ(pe, v(c) + tick) := TQ(s, v(c)) \cup \{(\tau, s, et + tick)\} \cup NT \setminus \{i\}$$

This essentially removes the instance and re-adds it to the queue with increased execution time, as the instance has been executed for one tick. When the execution time of the instance has risen above or is at least equal to the corresponding task's BCET, the instance can be removed, which can be accomplished by simply not re-adding it to the queue:

$$TQ(pe, v(c) + tick) := TQ(s, v(c)) \cup NT \setminus \{i\}$$

Since each processing environment is simulated using its own timed automaton, we require a network of these automata to represent a distributed system. To take clock offset in distributed systems into consideration, we will incorporate an initial location that delays the actual execution and simulation of the processing environment until the internal clock's offset is equal to the offset to the reference system. For

this our initial location needs to wait until $c = \mathrm{co}(c_r, c)$ is met and then reset c in an outgoing transition.

With all these considerations in mind, we can start with the model-building process, creating a model based on parameters such that we can defined fixed rules to transfer processing environments to these automata.

4.2 Building the Model

This section will cover the transfer of a processing environment $pe = (T, \mathrm{TG}, TQ, \mathrm{c}, \mathbf{Sch})$ to a timed automaton, incorporating the preparations done in section 4.1. In addition to the properties we need a way to incorporate scheduling based on a scheduling strategy **Sch** as defined in section 3.2.4, which can be replaced by any of the two defined scheduling algorithms.

We will start by building a model to simulate the behavior of a processing environment in section 4.2.1 and will extend this model in section 4.2.2 to gain more insight into timing-relevant properties.

4.2.1 Structure of a Single Processing Environment Model

Starting with a model to simulate just the timing behavior of a single processing environment, we assume that we have a processing environment *pe* and a reference system pe_r with a clock c_r. We show how achieve such a model using a timed automaton with just four locations.

The first location in our automaton model is the *start* location, which will be used to introduce the clock offset as mentioned in section 4.2. For this purpose, we set $I(start) \coloneqq c \leq \mathrm{co}(c_r, c)$ and add an outgoing transition with the guard $c = \mathrm{co}(c_r, c)$, resetting the clocks. Additionally, this transition shall push an instance of every task on the system in the queue to simulate startup behavior, which can be achieved simply by resetting the clock c first and then setting $TQ(pe, 0) \coloneqq TQ_0 \coloneqq NT$.

The transition shall lead to the base location, a location in which the automaton returns after every tick and which leads to other locations based on scheduling. We will call this location *scheduling* and mark it as committed, thus no time may pass while inside this location, such that time only passes while simulating the execution of tasks or an idle state.

In order to simulate timing aspects of task execution we add a location in which a tick can pass, which is then added to the execution time of the currently selected task instance; we will call this location *exec*. This location shall have exactly one transition leading to it, in which a task instance from the task queue is selected based on the scheduling strategy. Depending on the execution time that has already passed for this task instance, the instance must either stay in the queue, could be removed or must be removed. If the worst-case execution time is not yet reached after the tick has passed, the task instance might stay in the queue, so we add an outgoing edge for this case keeping the instance with an updated execution time in queue. Should the best-case execution time already be reached, the task instance could be removed from the queue, so we add an outgoing edge removing the instance.

Until the BCET of the task is reached, only one edge is available, which keeps the instance in queue and only increases its execution time. When the BCET is reached, the transition removing the instance from the queue becomes available, but the edge to allow the task instance to stay in the queue remains available until the execution time of the instance is larger than the task's WCET. If the execution time of the current task instance is between the BCET and WCET of the corresponding task, both edges are available at the same time. This allows us to exploit non-determinism to simulate variable execution times, as all possible states the system could be in are part of the simulation. Since we do not simulate functional behavior but simply the behavior in regard to timing, this is a valid approach approach allowing us to simulate all possible final execution times of the task instances.

If there is no task in the queue, the system is in an idle state, for which we add an *idle* location. Just like during task execution, a tick passes while in this location, but the queue does not need to be manipulated as it is empty. We will add a transition to leave this location when the tick has passed, setting the queue to *NT* to simulate the addition of time-triggered tasks.

In addition to this, we also need to add transitions from the *schedule* location to the *exec* location and the *idle* location based on whether the task queue is empty, as well as invariants to both of these locations to force them to take an outgoing transition after each tick.

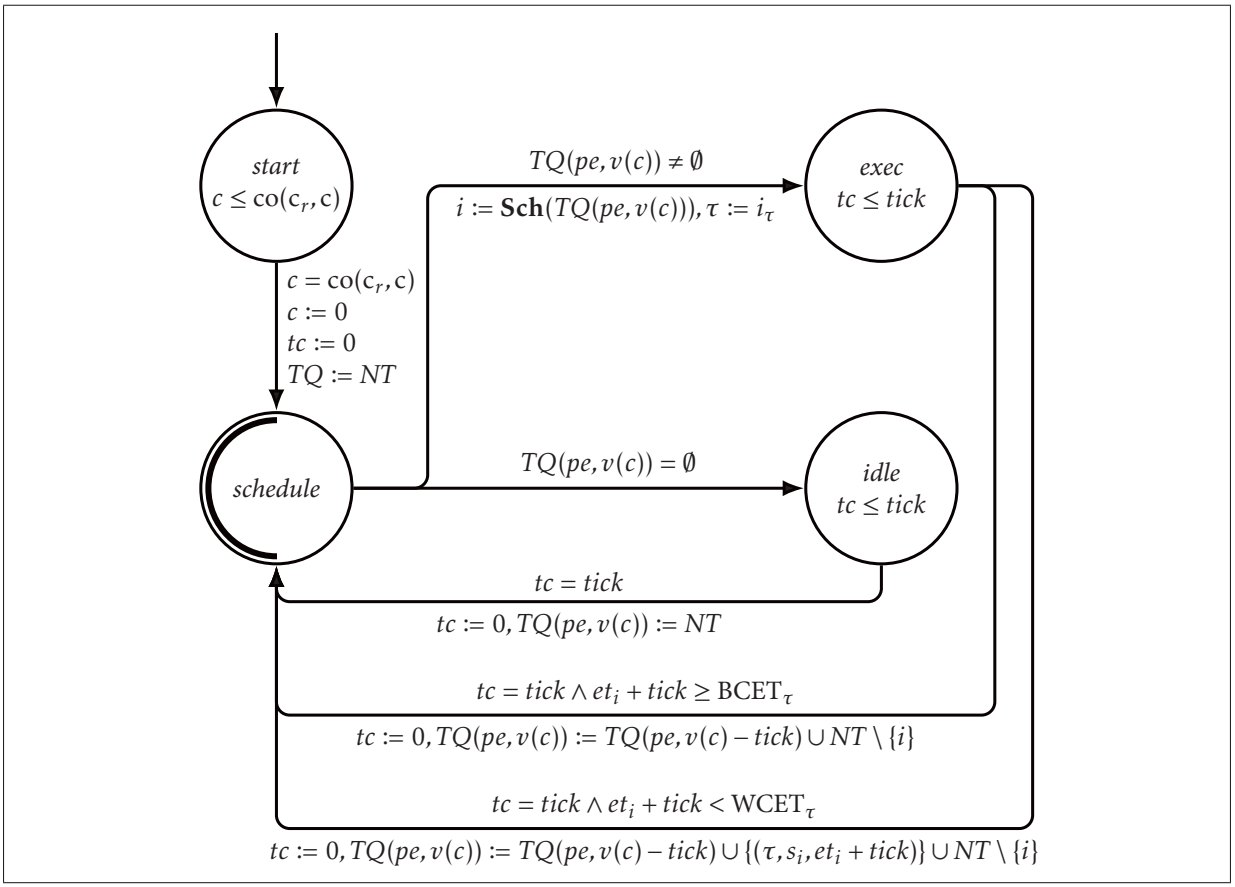

Figure 4.1: Model of a single Processing Environment

With all this combined we achieve the model shown in figure 4.1 with which it is already possible to simulate scheduled task execution with variable execution times, already incorporating the clock offset

when used in a network. The system works in a loop after leaving the initial *start* location, an example of a single run through this loop starting at $v(c) = i$ for a currently idle system could look like the following, with the clock vector being $\binom{c}{tc}$:

$$\cdots\rightarrow \langle schedule, \binom{i}{0}\rangle \xrightarrow{0} \langle idle, \binom{i}{0}\rangle \xrightarrow{tick} \langle idle, \binom{i+tick}{tick}\rangle \xrightarrow{0} \langle schedule, \binom{i+tick}{0}\rangle \rightarrow \cdots$$

But our goal isn't just the simulation of such a system, we would like to verify the real-time requirements in section 3.3, for which require more information about timing. We will specifically refine the *exec* location to get more detailed information about the process of task execution on the system.

4.2.2 Improved Model to aid Verification

To verify the real-time requirements we covered, we need to know when a task starts and when it ends. Since it does not suffice to know when any tasks start or ends, but we require that information separately for any task. Instead of a simple *exec* location to simulate the time spent executing a task, we require several locations for each task on the processing environment separately. We start by adding a $\tau exec$ location for the simulation of time spent executing a task like known from the previous model. In addition to this we also add a $\tau init$ location, which is the location where the incoming transition from the *scheduling* location will lead to.

To get information on when a task starts and finishes, we add the locations $\tau start$ and $\tau finish$ for each task, which need to be deterministically branched to. For the determination of the start time, we add a transition guarded by $et_i = 0$ from $\tau init$ to $\tau start$, a transition guarded by $et_i > 0$ from $\tau init$ to $\tau exec$ and an unguarded transition from $\tau start$ to $\tau exec$. With these transitions, the automaton deterministically branches into the $\tau start$ before the first execution of each task instance, and always transitions to the location $\tau exec$ regardless of the execution time. We mark the location $\tau init$ and $\tau start$ as committed, since we only want time to pass in the location simulating task execution.

For the $\tau finish$ location, we add the incoming transition that may be taken when the task instance's execution time is larger than its BCET, and add an outgoing transition to remove the instance from the task queue. This way, the $\tau finish$ must be entered before removing an instance from the queue and we have a reliable way to determine when a task instance finishes. We add an alternate edge that leaves the instance in the queue, just increasing its execution time, available as long as the WCET is not yet met – as already known from the previous model.

Adding an additional location $\tau exec'd$ after $\tau exec$ helps with a slightly more accessible logical segmentation of the locations, by having a single edge from the $\tau exec$ to the $\tau exec'd$, just reacting to the tick and increasing the execution time of the instance. Then we can check for the current, already incremented execution time in the two edges branching off this location.

Following all this, we obtain a task model like shown in figure 4.2. We have also added the location $\tau done$ to the end of this model such that this compound, representing a single task, can be added to the already existing model with just one incoming edge and one outgoing edge each. Both the $\tau exec'd$ and the $\tau done$ locations are marked committed as well, such that the $\tau exec$ location is the only one which is not committed. Although a total of six locations need to be added to the automaton representing the processing environment, only in a single one of them time may pass.

This model is designed in a way that guarantees that both the *start* and the *finish* locations are visited

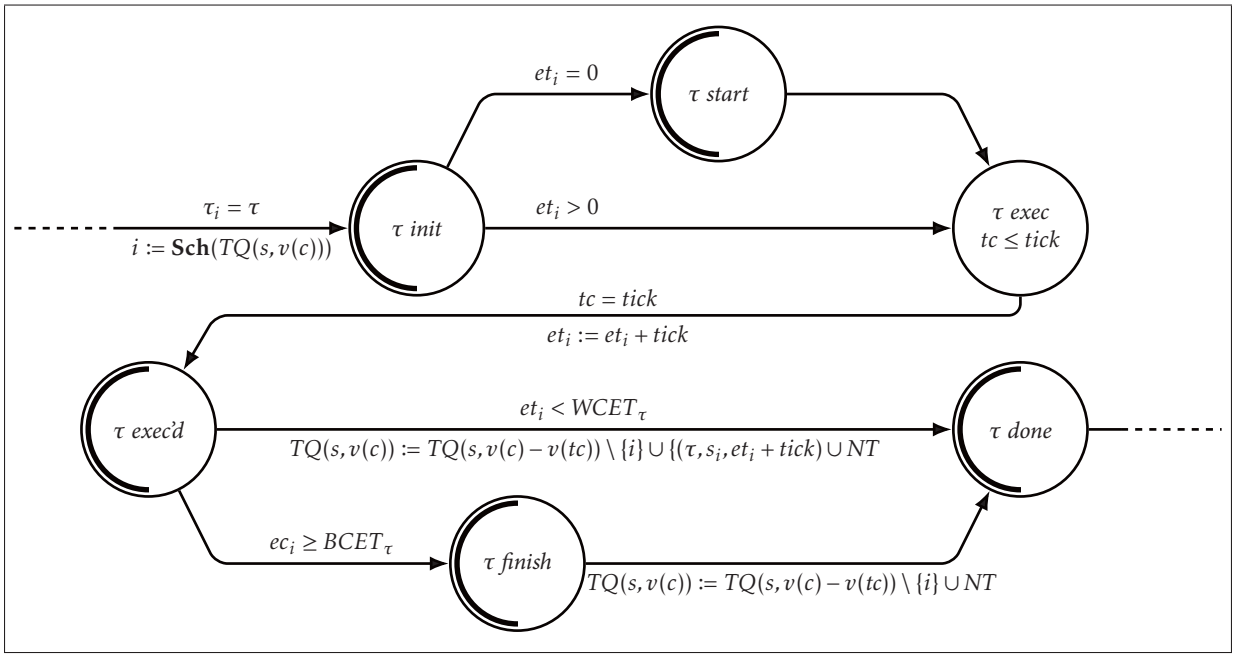

FIGURE 4.2: Detailed model of a task with separate *start* and *finish* locations

exactly once during each execution of a task instance corresponding to the task. In the given model, for each instance i of a task τ, we know that the automaton enters the location $\tau start$ when the scheduler decides that i shall be executed and the instance was either newly appended to the queue or the instance was in a ready state, which means that this marks the first time this instance is executed. Similarly, we know that when the $\tau finish$ location is entered, the corresponding instance will be removed from the queue in the only available outgoing edge, marking the last point of this instance's execution.

To gain easier access to runtime and data age information, we add two separate clocks per task measuring these times. Extending the model with data age clocks is rather simple, since we just need to add a clock $\tau_n da$ which is reset each time an instance of the task τ_n is finished and removed from the queue. Between these points in time, we have a steady increase in the data age, that is, in the time since the last finished execution of an instance of τ_n.

Adding a runtime clock to each task is not quite as trivial, since these shall only be increased while an instance of the task is currently executing or suspended, but not while the it is in the task queue or the system is idle and thus has an empty task queue. Since it is not possible to suspend a clock and prevent it from increasing, the runtime clocks that are not supposed to be increased need to be reset continuously. This has the side effect that we cannot reliably argue about the values of the runtime clocks while they are in the range $[0, tick]$.

To also grant other automata in a network access to start and finish information of tasks, we add broadcast channels for these actions on the outgoing edges of the corresponding locations.

In figure 4.3 we have a very detailed model of task execution based on the model shown in figure 4.3, extended by locations, clocks and broadcast channels.

While the addition of data age clocks and broadcast channels only affects one edge, we need to be pay extra attention to the runtime clock. Trivially, the runtime clock of τ_n needs to be reset when an instance of the task begins to be executed. But in each execution of a task, that is in each time advancement of a tick,

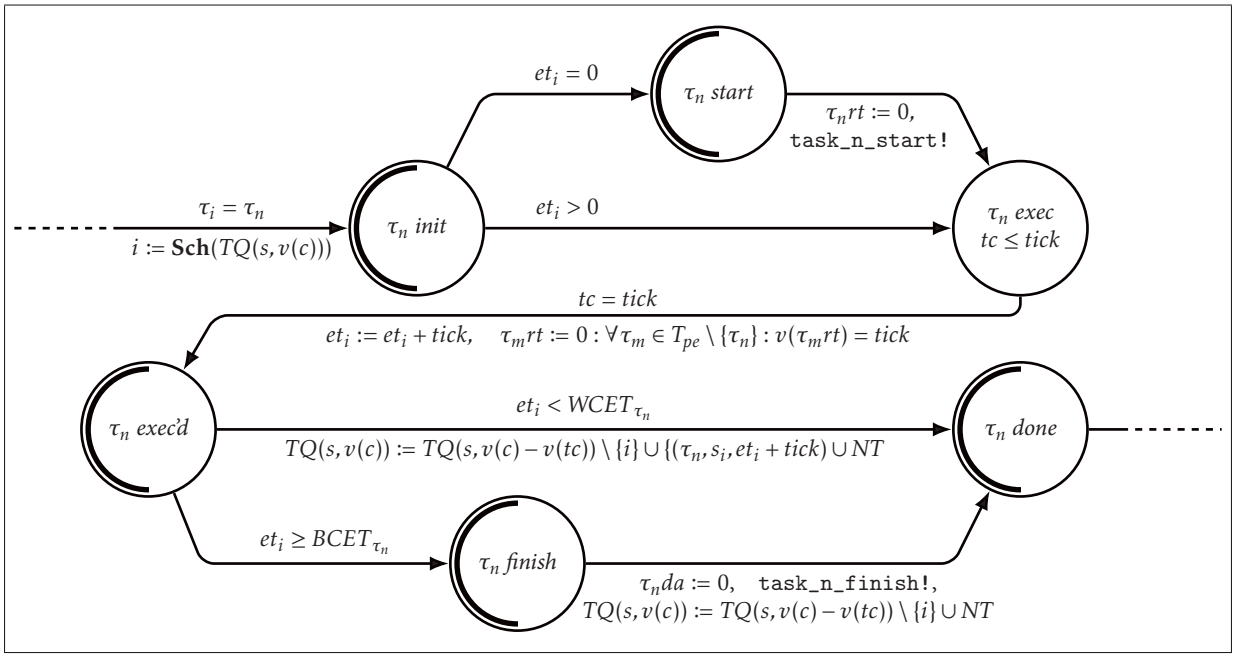

Figure 4.3: Very detailed model of a task with additional clocks and broadcast channels

only the clocks of the currently executing instance and of suspended task instances shall be incremented, the others shall be kept at zero. This is accomplished by resetting each runtime clock on the processing environment other than the one of the currently executed task to zero if its clock valuation equals exactly one tick when taking the outgoing edge of the execution location, in which one tick of time passes. We need to exclude the currently executing task since if this instance is being executed for the first time, its runtime clock will evaluate to one tick, but in this case this is a valid value. If instances of other tasks are currently suspended, their runtime clocks will have had a non-zero value before starting the execution of the current instances and thus their runtime clocks are larger than one tick after the execution, which is also desired behavior.

Unfortunately, this means that even tasks that never run, for example due to scheduling problems, also have a non-zero runtime clock valuation at times. Since timed automata require clock valuations to be in $\mathbb{R}^+$, we cannot prevent this by setting unused runtime clocks to $-tick$ before each execution. Because clock constraints are limited to comparisons to an $n \in \mathbb{N}_0$ we cannot implement edges to keep them indefinitely small, forcing their value to only be negligibly different from zero.

Integrating this advanced task model in the model for the single processing environment from figure 4.1, we obtain the model shown in figure 4.4. This figure shows how to model a system with a total of n tasks $\tau_1, \ldots, \tau_n$ and has a total of $2n+2$ clocks – the local timekeeper c, the tick clock tc and the runtime and data clocks for each task $\tau_m rt, \tau_m da \forall m \in \{1, \ldots, n\}$. Transitioning out of the idle location, all runtime clocks of tasks are set to zero, since that state is only entered when the task queue is completely empty and thus no task is currently being executed on the system. It is important to highlight that during the startup phase, while the offset is not yet reached, both the runtime and data age clocks need to be continually reset as well, once each tick. Otherwise, the values might rise close to or even above valid values encountered during simulation and verification, possibly invalidating verification queries.

An alternative display of the model can be found in figure A.1, where the locations corresponding to

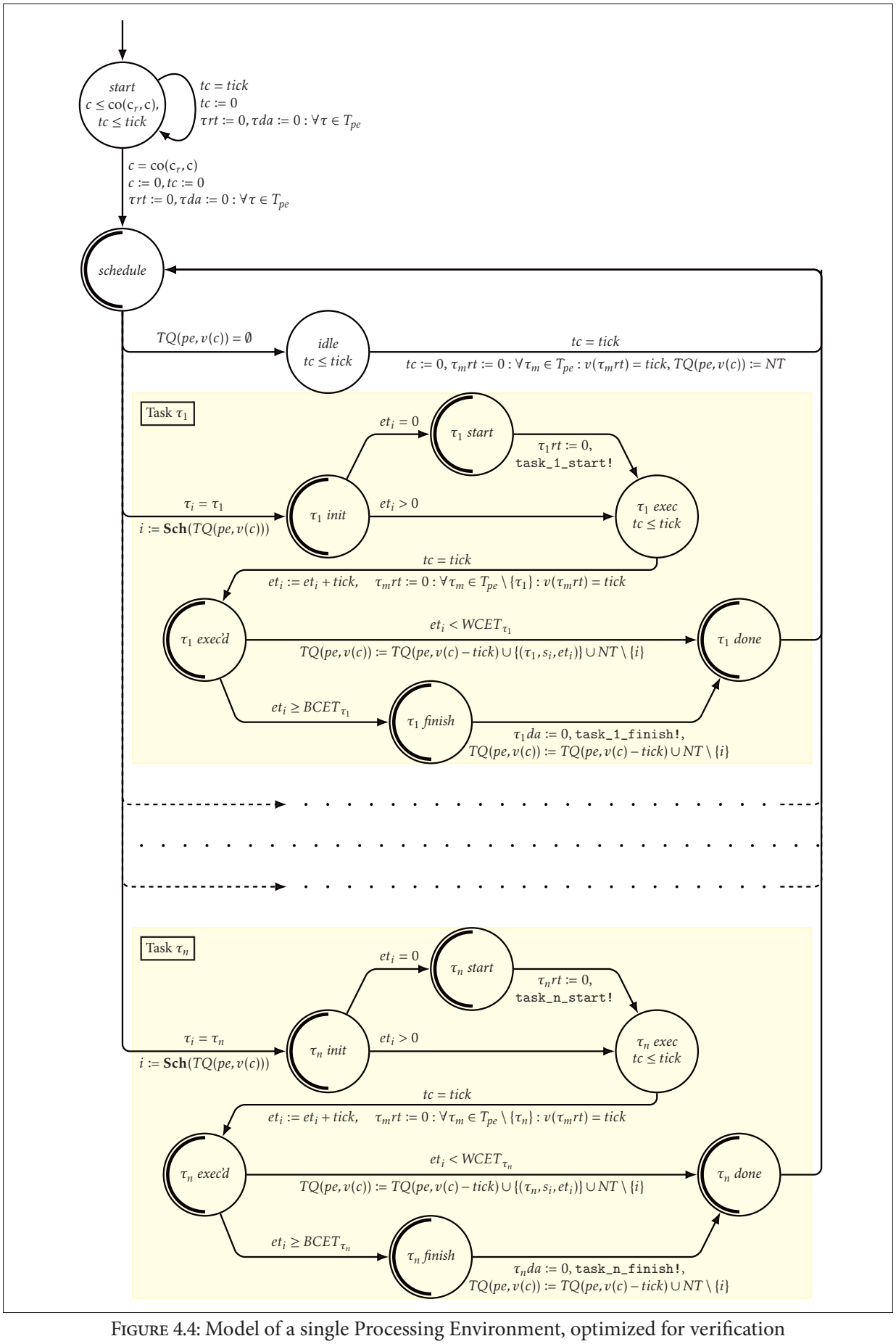

FIGURE 4.4: Model of a single Processing Environment, optimized for verification

each task are ordered as a row. We generally consider this a cleaner and easier-to-understand approach, but it is optimized for scrollable displays and not for printing on fixed-size paper, which is why it is not directly embedded here.

The generic approach to transfer a processing environment given as $pe = (T, \mathrm{TG}, TQ, c, \mathbf{Sch})$ to the automaton $\mathcal{A}_{pe} = \langle N, l_0, E, I \rangle$ is the following, assuming a reference system with clock c_r:

- the set of locations N composed of
  - the locations $start, schedule$ and $idle$,
  - the locations $\tau_n\ init,\ \tau_n\ start,\ \tau_n\ exec,\ \tau_n\ exec'd,\ \tau_n\ finish,\ \tau_n\ done$ for each task $\tau_n \in T$,
- the set of committed locations $N^C \in N$ composed of
  - the committed locations $schedule$,
  - the committed locations $\tau_n\ init,\ \tau_n\ start,\ \tau_n\ exec'd,\ \tau_n\ finish,\ \tau_n\ done$,
- the initial location $l_0 := start \in N$,
- the set of edges E composed of
  - $start \xrightarrow{tc=tick,\epsilon,r} start$ with $r := \{tc\} \cup \{\tau rt, \tau da : \forall \tau \in T\}$,
  - $start \xrightarrow{c=\mathrm{co}(c_r,c),\epsilon,r} schedule$ with $r := \{c, tc\} \cup \{\tau rt, \tau da : \forall \tau \in T\}$,
  - $schedule \xrightarrow{TQ(s,v(c))=\emptyset,\epsilon,\emptyset} idle$,
  - $idle \xrightarrow{tc=tick,\eta,r} schedule$,
    with $r := \{tc\} \cup \{\tau_m rt : \forall \tau_m \in T_{pe} : v(\tau_m rt) = tick\}, \eta := TQ(s, v(c)) = \emptyset$,
  - for each task $\tau_n \in T$ assuming $i := \mathbf{Sch}(TQ(s, v(c)))$
    1. $schedule \xrightarrow{\tau_i=\tau_n,\epsilon,\emptyset} \tau_n\ init$,
    2. $\tau_n\ init \xrightarrow{et_i=0,\epsilon,\emptyset} \tau_n\ start$,
    3. $\tau_n\ init \xrightarrow{et_i>0,\epsilon,\emptyset} \tau_n\ exec$,
    4. $\tau_n\ start \xrightarrow{\top,\texttt{task\_1\_start!},\{\tau_n rt\}} \tau_n\ exec$,
    5. $\tau_n\ exec \xrightarrow{tc=tick,et:=et_i+tick,r} \tau_n\ exec'd$
       with $r := \{tc\} \cup \{\tau_m rt : \forall \tau_m \in T_{pe} \setminus \{\tau_n\} : v(\tau_m rt) = tick\}$,
    6. $\tau_n\ exec'd \xrightarrow{et_i<WCET_{\tau_n},\eta,\emptyset} \tau_n\ done$
       with $\eta ::= TQ(s, v(c)) := TQ(s, v(c) - tick) \cup \{(\tau_1, s_i, et_i)\} \cup NT \setminus \{i\}$,
    7. $\tau_n\ exec'd \xrightarrow{et_i \geq BCET_{\tau_n},\epsilon,\epsilon} \tau_n\ finish$,
    8. $\tau_n\ finish \xrightarrow{\top,\texttt{task\_1\_start!} \wedge \eta,\tau_n da} \tau_n\ done$
       with $\eta ::= TQ(s, v(c)) := TQ(s, v(c) - tick) \cup NT \setminus \{i\}$,
    9. $\tau_n\ done \xrightarrow{\top,\epsilon,\emptyset} schedule$,

- the function $I : N \mapsto \mathcal{B}'(\mathcal{C})$ assigning the invariants to the locations with
  - $I(start) = c \leq \mathrm{co}(\mathrm{c_r}, \mathrm{c}) \wedge tc \leq tick$,

- $I(idle) = tc \leq tick$,
- $I(\tau_n\ exec) = tc \leq tick$ for each task $\tau_n \in T$

The notation $\eta ::= a := b$ is used to denote that the internal action η consists of the assignment $a := b$ to avoid confusion with the comparison operator = and the assignment operator used inside the action. An edge guarded by $\top$ is an unguarded edge, which can be taken at any time. Since these are used as the single outgoing edges of committed locations here, they must immediately be transitioned over.

4.3 Modelling in UPPAAL

To develop and test the model in UPPAAL we will use version 4.1.19, the newest development snapshot released on July 1st in 2014. Compared to the current official release from September 27th, 2010, it offers several small feature and performance improvements like the addition of a modulo operator and we did not experience any disadvantages compared to the older version.

Development in UPPAAL is generally done using templates, in which a single timed automaton can be constructed, and a corresponding definition file, which allows custom functionality to be added using a C-like syntax. Templates can have parameters and are then called parametrizable templates, which allow for the dynamic instantiation of a network of timed automata from a small number of templates and varying parameters. In addition to templates, global declarations can be made, which apply to all templates; a single shared version of the global definitions exists between all instantiated templates and any changes to these from a template's scope immediately affect all others as well. Apart from declarations in global and template scope, there are also system declarations, which are used to define and initialize the network of timed automata for simulation and verification. In the system declarations, each timed automaton is defined from a template, given parameters when necessary, and composed into the final system.

This section will detail the development process of the UPPAAL model, starting with the global declarations to setup generic behavior and commonly used functions, and then showing the process of creating templates and setting up the system. We will only show small snippets of the code here, listings of full code files can be found in section B.1.

4.3.1 Global Declarations

Before setting up the automata and their behavior, we will use the global declarations to implement a task model and scheduling functions, which can then be accessed from the templates.

During the development of the UPPAAL model, several constants were introduced to deal with often-needed numbers, these are also part of the global declarations. Since these do not contribute to the model itself, these can be found in listing A.1 in the appendix.

The language used in UPPAAL resembles C-code, so we define our types as `struct` using `typedef`. To be able to refer to tasks, we create the basic structure `Task` composed of a numeric identifier, the `ID` of the task, and further we enable the specification of the `BCET` and `WCET` as well. With this base, we create the structure `EDF_Task`, which is composed of a task, a relative deadline and a period, as well as an `OSEK_Task` structure, comprised of a priority and a period. Further, we define `EDF_Task_Instance` and `OSEK_Task_Instance`, both of which allow us to save their execution time as well as their start time; additionally, the `EDF_Task_Instance` also saves the absolute deadline, which is the absolute deadline

of the instance calculated from the start time and the relative deadline of the corresponding `EDF_Task`. We also define data types for the task queues, which are aliases for arrays of task instances.

Additionally, since we assume tasks to be unique, we define the runtime and data age clocks as arrays in the global declarations. We proceed the same way with the broadcast channels and also add a Boolean array to save whether a task is currently being executed, as we are unable to compare clock values outside of guards in UPPAAL, including custom functions.

```
typedef struct
{
    int[0, TASK_ID_MAX] ID;
    int[0, MAX_WCET] BCET;
    int[0, MAX_WCET] WCET;
} Task;

typedef struct
{
    Task t;
    int[0, MAX_REL_DEADLINE] rel_deadline;
    int[0, MAX_PERIOD] period;
} EDF_Task;

typedef struct
{
    Task t;
    int[0, MAX_T_PRIORITY] t_priority;
    int[0, MAX_PERIOD] period;
} OSEK_Task;

typedef struct
{
    EDF_Task edf_t;
    int[0, TIME_MAX + MAX_REL_DEADLINE] deadline;
    int[0, TIME_MAX] start;
    int[0, MAX_WCET] et;
} EDF_Task_Instance;

typedef struct
{
    OSEK_Task osek_t;
    int[0, TIME_MAX] start;
    int[0, MAX_WCET] et;
} OSEK_Task_Instance;

typedef EDF_Task_Instance EDF_Task_Queue[TASK_QUEUE_MAX];
typedef OSEK_Task_Instance OSEK_Task_Queue[TASK_QUEUE_MAX];

// verification helpers (clocks and channels for start/end events)
clock rt_c[TASK_ID_MAX];
clock da_c[TASK_ID_MAX];
```

```
bool is_running[TASK_ID_MAX];
broadcast chan task_start[TASK_ID_MAX];
broadcast chan task_finish[TASK_ID_MAX];
```

Listing 4.1: Type definitions, global clock and channel arrays

In addition to the type definitions, *null* types had to be created to indicate that no such entity is present. The null types are available using `NULL_[DATATYPE]` and their initial definition, along with other auxiliary definitions, can be found in listing A.2.

Next up are functions to generate these data types, manage and manipulate the queue and to actually perform scheduling on a task queue.

EDF Scheduling

To be able to work with the created data types, we want a straight-forward way to generate and initialize instances. In order to save memory and achieve consistency, we use pass-by-reference to link already-initiated tasks to EDF tasks using UPPAAL's `&` operator when defining the function parameters. The relative deadline and period of the EDF task are passed as values, and the generated `EDF_Task` is returned by the function `generate_EDF_Task(Task &t, int rel_deadline, int period)`.

To create an EDF task instance, we only pass the corresponding task as a reference and the local time of the processing environment as the parameters, as the rest can be calculated from these. The absolute deadline is calculated from the local time and the relative deadline of the EDF task, the start time can be assumed to be the passed local time, and the execution time is zero, since the instance has just been newly generated. This means that `generate_EDF_Task_Instance(EDF_Task &edf_t, int local_time)` shall be called from the templates when appending new instances to the queue and the returned `EDF_Task_Instance` is only valid for the processing environment from which the function call initiated.

Since UPPAAL does not have a default null type, we need to fill newly created task queues with the `NULL_EDF_TI` instance. In all methods dealing with a processing environment's task queue, we will pass the `EDF_Task_Queue` as a reference parameter and perform the operations directly on it. During the initialization of a template corresponding to a processing environment, `initialize_EDF_Task_Queue(EDF_Task_Queue &tq)` shall be called as early as possible, and must be called before any other operations are applied to the queue in order to avoid undefined behavior.

```
EDF_Task generate_EDF_Task(Task &t, int rel_deadline, int period) {
    EDF_Task new_task = { t, rel_deadline, period };
    return new_task;
}

// create an EDF task instance with a deadline based on the current time
EDF_Task_Instance generate_EDF_Task_Instance(EDF_Task &edf_t, int local_time) {
    EDF_Task_Instance new_ti = { edf_t, local_time + edf_t.rel_deadline, local_time, 0 };
    return new_ti;
}

// fill the EDF task queue given as a reference parameter with null values
```

```
void initialize_EDF_Task_Queue(EDF_Task_Queue &tq) {
    for (i : int[0, TASK_QUEUE_MAX - 1]) {
        tq[i] = NULL_EDF_TI;
    }
}
```

Listing 4.2: Auxiliary functions to help manage EDF data types

With an initialized system, we are working with EDF task instances in the task queue represented by the data type `EDF_Task_Instance`. Before actually declaring the scheduling function, we need the ability to properly maintain the task queue. We chose to represent the queue as an array, initialized with the elements `NULL_EDF_TI`, ordered in a way that we can consider the first encountered `NULL_EDF_TI` to be the end of the queue. This is a consistency requirement that needs to be considered when developing the functions to enqueue and dequeue the task instances.

This means that enqueuing is comparatively simple, as `EDF_enqueue(EDF_Task_Queue &tq, EDF_Task_Instance &edf_ti)` searches through the queue from the beginning and inserts the reference to the newly created task instance in the first free place found in the queue. Dequeuing changes the task instance at the given position to a `NULL_EDF_TI`, since the instance has been finished and its space in the queue is now considered to be empty. After this has been done `EDF_dequeue(EDF_Task_Queue &tq, int[0, TASK_QUEUE_MAX] ti_pos)` sifts through the queue after the deleted instance and shifts up all other instances to retain the consistency requirement up until the next `NULL_EDF_TI` is encountered, marking the actual end of the queue.

To get the number of items currently in a queue, we start enumerating it beginning from the first element, up until we find the first occurrence of `NULL_EDF_TI`, and return the variable used for enumeration. The queue that is enumerated is the one given as a reference argument to `count_EDF_queue_items(EDF_Task_Queue &tq)`.

```
int[0, TASK_QUEUE_MAX + 1] count_EDF_queue_items(EDF_Task_Queue &tq) {
    int[0, TASK_QUEUE_MAX + 1] i = 0;
    while(i < TASK_QUEUE_MAX && tq[i] != NULL_EDF_TI)
        i++;
    return i;
}

// insert the given EDF task instance in the first free space
// in the given EDF task queue
void EDF_enqueue(EDF_Task_Queue &tq, EDF_Task_Instance &edf_ti) {
    int[0, TASK_QUEUE_MAX + 1] i;
    i = count_EDF_queue_items(tq);
    tq[i] = edf_ti;
}

// dereference the given EDF task instance inside the queue and
// shift up the elements after it
void EDF_dequeue(EDF_Task_Queue &tq, int[0, TASK_QUEUE_MAX] ti_pos) {
```

```
        int[0, TASK_QUEUE_MAX] i = ti_pos;
20      tq[ti_pos] = NULL_EDF_TI;
        while(i < (TASK_QUEUE_MAX - 1) && tq[i + 1] != NULL_EDF_TI) {
            if(tq[i] == NULL_EDF_TI) {
                tq[i] = tq[i + 1];
                tq[i + 1] = NULL_EDF_TI;
25          }
            i++;
        }
    }
```

Listing 4.3: Functions to manage and manipulate the EDF Task Queue

The last function to be defined for EDF-scheduled tasks in the global declarations is the scheduling function. Since each task instance in the queue has information about its absolute deadline, the scheduling function simply moves through the queue and returns the instance with the lowest absolute deadline, that is, the next instance that needs to be finished. Should multiple instances have the same deadline, the index of the first one encountered is returned, which is the one with a lower index. The function `EDF_schedule(EDF_Task_Queue &tq)` does not return the instance, but rather its position in the task queue. This is due to a limitation in UPPAAL, which – while allowing references to be passed to a function – does not allow a function to return a reference. A way to circumvent this would be to pass another reference and set this to the selected instance, but both for consistency and compatibility reasons we chose the approach of just returning the index in the queue.

```
    // main scheduling function of EDF processing environments; selects the first
    // instance in the given queue that has the lowest absolute deadline
    int[0, TASK_QUEUE_MAX] EDF_schedule(EDF_Task_Queue &tq) {
        EDF_Task_Instance next_eti = tq[0];
 5      int[0, TASK_QUEUE_MAX] next_eti_pos = 0;
        int[1, TASK_QUEUE_MAX + 1] i = 1;
        // not run for empty queue due to tq[1] == NULL_EDF_TI
        while(i < TASK_QUEUE_MAX && tq[i] != NULL_EDF_TI) {
            if(tq[i].deadline < next_eti.deadline) {
10              next_eti = tq[i];
                next_eti_pos = i;
            }
            i++;
        }
15      return next_eti_pos;
    }
```

Listing 4.4: Scheduling function for EDF Task Queues

For correctly handled, consistent queues, all of the functions introduced for EDF are bound by the runtime complexity $\mathcal{O}(n)$, with n being the amount of elements in the queue.

OSEK Scheduling

Most basic functions for OSEK scheduling can be easily transferred from their EDF counterparts. When generating OSEK task instances, no additional scheduling parameter – like the absolute deadline in EDF – is encoded with the instance, since the priority is already part of the referenced `OSEK_Task`. For a complete reference of the auxiliary and queue manipulation functions for OSEK, please see the relevant parts of listing B.1.

The only major change when dealing with OSEK scheduling is the scheduling given in listing 4.5. While still mostly resembling the scheduling function for EDF given in listing 4.4, the attribute that is used for scheduling now is not encoded in the checked `OSEK_Task_Instance`, but instead in the referenced `OSEK_Task`. This again highlights the difference in static and dynamic scheduling strategies as explained in section 3.2.4. Like known from EDF, the queue is enumerated from the beginning, and the first occurrence of a task with the highest priority is returned, in case of multiple instances with the same priority the occurrence of the one with the lowest index.

```
// main scheduling function of OSEK processing environments; selects the first
// instance in the given queue that belongs to the task with the highest priority
// of all instances currently in the queue
int[0, TASK_QUEUE_MAX] OSEK_schedule(OSEK_Task_Queue &tq) {
    OSEK_Task_Instance next_osek_ti = tq[0];
    int[0, TASK_QUEUE_MAX] next_osek_ti_pos = 0;
    int[1, TASK_QUEUE_MAX + 1] i = 1;
    // not run for empty queue due to tq[1] == NULL_OSEK_TI
    while(i < TASK_QUEUE_MAX && tq[i] != NULL_OSEK_TI) {
        if(tq[i].osek_t.t_priority > next_osek_ti.osek_t.t_priority) {
            next_osek_ti = tq[i];
            next_osek_ti_pos = i;
        }
        i++;
    }
    return next_osek_ti_pos;
}
```

Listing 4.5: Scheduling function for OSEK Task Queues

For OSEK tasks, the runtime and space complexity classes equal those of their EDF counterparts. Since OSEK task instances have one less attribute to encode, they require slightly less space, but that difference is negligible.

4.3.2 Templates

Each automaton in UPPAAL is represented by a template, which defines the locations and edges including all information typically represented in a timed automaton model. Additionally, each template has its own declarations, which – similar to the global declarations – allow the definition of variables and functions, although in function scope. Since edges in the template are allowed to use functions with the `bool` return type as guards, and call any function in an update statement, the automata in UPPAAL can be setup to simulate rather sophisticated behavior.

An additional feature in UPPAAL is the parametrization of templates, such that a single template can make use of given parameters, used as placeholders, so that it can be instantiated multiple times with different arguments. This perfectly fits our use case, such that using parametrizable templates, we only need to create one processing environment template per amount of tasks run on the system and insert the tasks dynamically using the parameters.

The Automaton Template

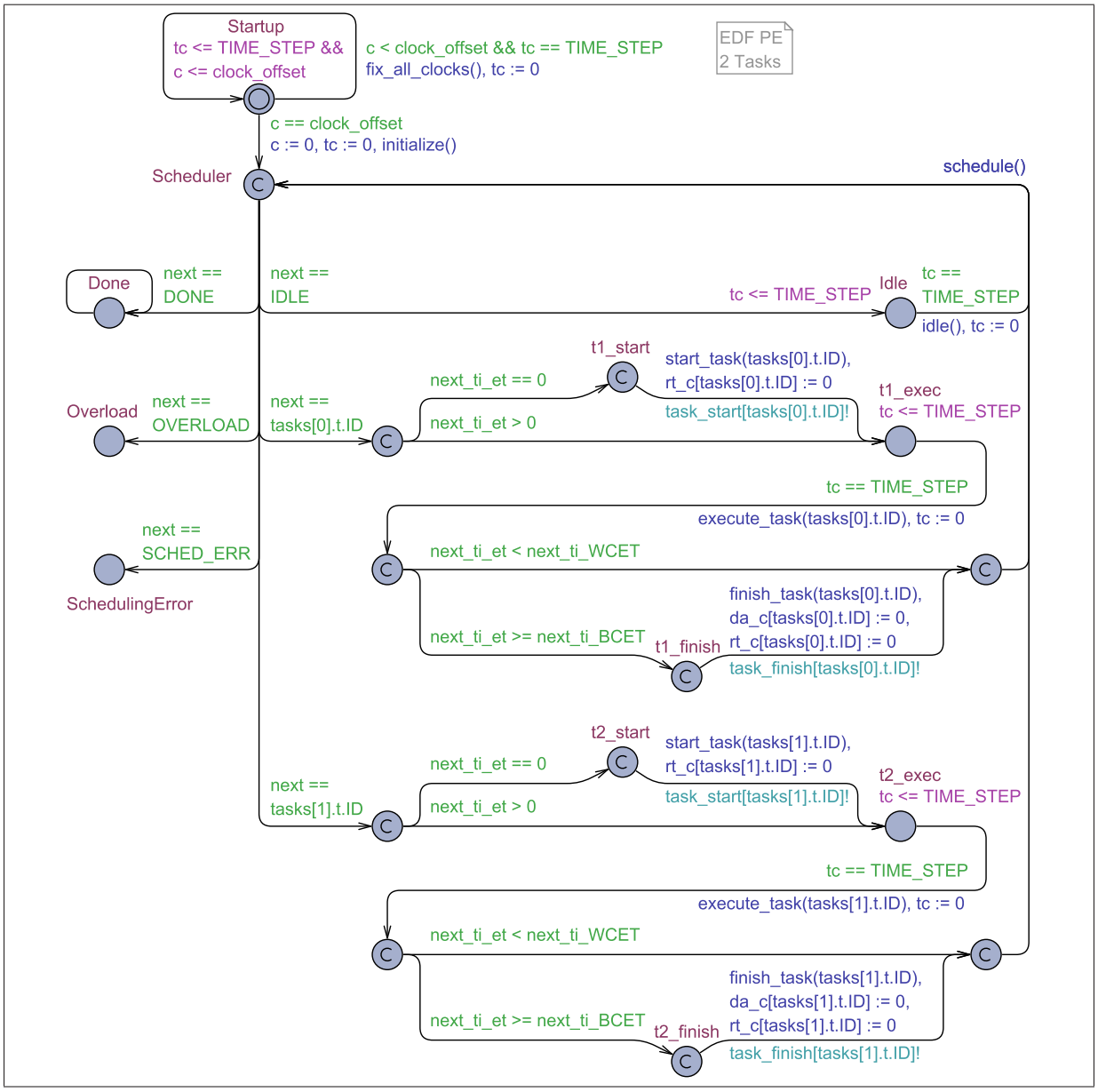

FIGURE 4.5: Example of an EDF-scheduled process environment template with two tasks

The templates itself don't differ based on the scheduling function used, only in the function declarations. This generic template shown in figure 4.5 calls the functions `initialize()` upon system initialization, `schedule()` after each time step and `idle()` when leaving the *Idle* location.

For each task, there is a complex resembling the one detailed in figure 4.3. This mostly works with

the numerical task identifiers that were already mentioned, and calls the functions `start_task(ID)`, `execute_task(ID)` and `finish_task(ID)` as well as sending over the relevant broadcast channels for the task ID.

It is worth noting that the automaton uses additional auxiliary locations compared to the one detailed in figure 4.4, namely the *Done*, the *Overload* and the *SchedulingError* locations. The first two are the result of limitations in UPPAAL, as both clock values and integer variables are limited by the `int16` bounds of the underlying C architecture, which amounts to 32767. This means that neither a clock value, nor the value of a local variable can rise above this bound. Since each automaton has the clock *c*, continuously counting the local time, this bound effectively limits the simulation steps possible. We have found a bound simulation time sufficient for all systems we tested and since a limit on the time reduces the state space, setting it to even lower values also means a faster verification. This is why the automata network, after having reached the time set by the `TIME_MAX` constant, forces all automata to enter the *Done* location, effectively reaching a verifiable end condition.

As long as all possible variations possible occur at least once during the time specified by `TIME_MAX`, this is not an issue. For systems with very large, different periods, issues can be encountered since not all combinations might be part of the simulation and verification process. Since this is a problem with the underlying architecture, it cannot be fixed trivially. A possible mitigation might be to introduce an additional integer variable into the system and increment it at a fixed interval, for example each 30000 time steps, resetting the clock in the process. This allows to keep track of way larger time spans, but also requires a lot of changes to the model.

The *Overload* location is reached when the queue length in a system exceeds the value of the constant `TASK_QUEUE_OVERLOAD`. The recommended value for this is at least twice the amount of total tasks available, since then we know that the system isn't schedulable in all cases. Should the task queue contain at least twice the amount of tasks on the processing environment, we know that – considering task instances always ending with the task's WCET as execution time – the task queue will be filled faster than it can be worked off, indicating a non-schedulable system.

The *SchedulingError* location is only part of EDF and not of OSEK templates, as it indicates an error regarding dynamic scheduling. This location is entered when during EDF scheduling inside of the template, the absolute deadline of a task instance cannot be met since it is in the past. When this state is encountered, changes to the scheduling parameters of the EDF tasks need to be made to achieve a schedulable system.

All templates share common definitions as shown in listing 4.6, the two non-task clocks *c* and *tc*, a constant for the amount of tasks handled by the template, a variable saving the current local time and an array of the contained task's periods, used as triggers for adding them to the queue. In addition to these, shorthands are defined to access the next action, been determined by the scheduler, for the queue index of the next task instance, the amount of task instances currently in the queue and the execution time of the next task instance, along with the corresponding tasks' WCET and BCET.

```
// declaration of clocks
clock c;
clock tc;
int[0, TIME_MAX] local_time;
```

```
5
    // declaration of basic scheduling parameters
    EDF_Task_Queue tq;
    const int [0, TASK_AMOUNT] TASK_COUNT = 2;
    int[0, MAX_PERIOD] TG_triggers[TASK_COUNT] = { tasks[0].period, tasks[1].period };
10
    // variables determined by scheduling
    int[-3, TASK_AMOUNT] next;
    int[0, TASK_QUEUE_MAX] next_ti_pos;
    int[0, TASK_QUEUE_MAX] queue_item_count;
15  // shorthands for Template use
    int[0, MAX_WCET] next_ti_et;
    int[0, MAX_WCET] next_ti_WCET;
    int[0, MAX_WCET] next_ti_BCET;
```

Listing 4.6: Definitions for all templates, example showing a EDF PE with two tasks

Not only definitions, but also some functions are common to all templates, which are listed in listing 4.7. The function `fix_task_clocks()` is the equivalent to the mathematical operation used in the transition out of the execution locations in figure 4.4. It resets all runtime clocks of the tasks that are currently not running on the system, but uses a Boolean variable to check whether the clock shall be reset or not, since the clock values cannot be accessed in update statements. During the startup phase, `fix_all_clocks()` is called every tick to not only reset the runtime clocks, but also the data age clocks. There also is the `idle()` function, which is called in the update directive when leaving the *Idle* location. It increases the `local_time` variable to match the clock value, calls the aforementioned function to fix the runtime clocks and then calls the `schedule()` function to have the system determine the action for the next time step.

```
    void fix_task_clocks() {
        // reset runtime clocks of tasks on this PE
        // that are currently not running or suspended
        for (i : int[0, TASK_COUNT - 1]) {
5       if(is_running[tasks[i].t.ID] == false) {
                rt_c[tasks[i].t.ID] := 0;
            }
        }
    }
10
    void fix_all_clocks() {
        // reset runtime and data age clocks of tasks
        // on this PE continuously before starting
        for (i : int[0, TASK_COUNT - 1]) {
15          if(is_running[tasks[i].t.ID] == false) {
                rt_c[tasks[i].t.ID] := 0;
                da_c[tasks[i].t.ID] := 0;
            }
        }
20  }
```

```
    void idle() {
        // advance one step in time
        local_time += TIME_STEP;
25      fix_task_clocks();
        schedule();
    }
```

Listing 4.7: Functions for all templates

EDF Scheduling

While the following functions are present in all templates, regardless of the scheduling algorithm used, there are minor differences in the implementations because of different data types. This means that the the following functions – while based on EDF scheduling – are mostly applicable for all templates; a full reference can be found in section B.1.

The `initialize()` function serves as a basic setup for the system required at the start of the PE simulation. It calls the `initialize_EDF_Task_Queue(EDF_Task_Queue &tq)` function known from listing 4.2 to setup the task queue, resets the value of the `local_time` variable to 0 to match the value of the clock c which is reset in the update transition, and calls the `schedule()` function to determine the action for the next, the very first, time step for this automaton.

Then there are the three task-dependent functions `execute_task(int task_ID)`, `start_task(int task_ID)`, and `finish_task(int task_ID)`, which all take the ID of the corresponding task as parameter. Because of this parameter, task-specific behavior can be implemented by checking for the task ID and taking appropriate actions in each of these functions. In the current implementation `start_task(int task_ID)` just sets the token to not reset the runtime clock until the task is finished. The function `execute_task(int task_ID)` updates the execution time of the current task instance and its shorthand, adjusts the `local_time` variable to match the main clock again and resets the runtime clocks of non-running tasks. At the end of a task instance execution, `finish_task(int task_ID)` is used to reset the token so that the runtime clock is correctly reset, and removes the finished task instance from the queue.

```
    void initialize() {
        // initialize task queue and time
        initialize_EDF_Task_Queue(tq);
        local_time = 0;
 5
        schedule();
    }

    void start_task(int task_ID) {
10      is_running[task_ID] = true;
    }

    void execute_task(int task_ID) {
```

```
    // simulate running the task for one time step
    // task-specific behavior can be implemented by checking for task ID
    tq[next_ti_pos].et += TIME_STEP;
    // update shorthand
    next_ti_et = tq[next_ti_pos].et;
    // advance one step in time
    local_time += TIME_STEP;
    fix_task_clocks();
}

void finish_task(int task_ID) {
    // mark the end of task execution and dequeue the instance
    is_running[task_ID] = false;
    EDF_dequeue(tq, next_ti_pos);
}
```

Listing 4.8: Auxiliary functions in EDF Templates

After each time step, the `schedule()` function is called, which is responsible for a multitude of functions.

First, it checks the period of the tasks simulated in the processing environment and adds task instances to the queue which belong to tasks for which the time grid is met. This is the main reason for needing an additional `local_time` variable, as clock valuations cannot be checked against other variables using mathematical operations, like the time grid triggers, neither can they be passed to other functions, like when calling the function to generate a new EDF task instance.

Next, it calls the scheduling function known from listing 4.4 to determine the next task instance for the processing environment and sets up the shorthands used in the template. Then it checks if one of the three auxiliary locations needs to be entered; if the simulations has reached its end, the *Done* location is entered, if the amount of items in the task queue is larger than allowed by `TASK_QUEUE_OVERLOAD`, the *Overload* location is entered and if the deadline of the currently selected task instance is already in the past, the *SchedulingError* location is entered.

```
void schedule() {
    // generate and enqueue tasks for which the period is met
    for (i : int[0, TASK_COUNT - 1]) {
        if(local_time % TG_triggers[i] == 0) {
            EDF_Task_Instance ti;
            EDF_Task t = tasks[i];
            ti = generate_EDF_Task_Instance(t, local_time);
            EDF_enqueue(tq, ti);
        }
    }

    // determine next task
    next_ti_pos = EDF_schedule(tq);
    next = tq[next_ti_pos].edf_t.t.ID;
    next_ti_et = tq[next_ti_pos].et;
```

```
        next_ti_WCET = tq[next_ti_pos].edf_t.t.WCET;
        next_ti_BCET = tq[next_ti_pos].edf_t.t.BCET;

        // end simulation after the maximum simulation time is reached
        if((local_time + clock_offset) >= TIME_MAX)
            next = DONE;

        // switch into overload mode (essentially deadlock)
        // when the task queue is too full
        queue_item_count = count_EDF_queue_items(tq);
        if(queue_item_count > TASK_QUEUE_OVERLOAD)
            next = OVERLOAD;

        // detect a runtime scheduling error when a deadline is violated
        if((next != IDLE) && (next != DONE) && (tq[next_ti_pos].deadline < local_time))
            next = SCHED_ERR;
    }
```

Listing 4.9: Scheduling function for EDF Templates

OSEK Scheduling

The template code for OSEK scheduling is that already known of EDF scheduling, just with different data types and without the code to check for a scheduling error during runtime. Since there are no notable differences this time around, other than the missing check for an already passed deadline, the relevant code can be found in section B.1 and will not be listed here additionally.

4.3.3 System Declarations

The system declarations are the last set of declarations in UPPAAL and are used to describe the system, which is the network of timed automata that shall be simulated and verified. For this system, the process environments need to be defined by instantiating the relevant templates. The instantiation of templates works similar to the creation of objects in object-oriented languages, the parameters are given during creation. After all templates have been instantiated, the simulatable system must be defined using the `system` directive and a comma-separated list of already created templates.

```
// Task definitions (ID, BCET, WCET)
const Task T1 = {1, 5, 6};
const Task T2 = {2, 8, 11};
const Task T3 = {3, 4, 6};
const Task T4 = {4, 4, 5};

// EDF Task Definitions (Task, relative Deadline, Period)
const EDF_Task ET1 = { T1, 8, 12 };
const EDF_Task ET4 = { T4, 14, 20 };

// OSEK Task Definitions (Task, Priority, Period)
const OSEK_Task OT2 = { T2, 2, 30 };
```

```
const OSEK_Task OT3 = { T3, 3, 30 };

// Array Compositions
const EDF_Task PE1_Tasks[2] = { ET1, ET4 };
const OSEK_Task PE2_Tasks[2] = { OT2, OT3 };

// PE Definitions Template (Task Array, Offset to Reference PE)
PE1 = PE_2T_EDF(PE1_Tasks, 0);
PE2 = PE_2T_OSEK(PE2_Tasks, 2);

system PE1, PE2;
```

Listing 4.10: System declarations instantiating two processing environments with two tasks each

An example is given in listing 4.10; here, four tasks are created, two of them are embedded in EDF and OSEK tasks each. In order to pass these tasks to the templates, they are comprised into arrays of their respective data type. Then, the array containing the EDF tasks with the ID 1 and 4 is passed to the two-task template for EDF, instantiating the reference system. After that, the OSEK tasks with ID 2 and 3 are passed to a second processing environment template, instantiating an OSEK processing environment with a clock offset of 2 compared to the reference system. Both processing environments are then passed to the `system` directive, so that they both will be part of the simulation. We will continue using this small example in chapter 5 for example verification queries.

5 Verification of Real-Time Requirements Using TCTL

In chapter 4 we have built suitable for verifying the requirements specified in section 3.3. Using UPPAAL and TCTL, which was introduced in section 2.2, allows us to determine whether the given requirements are feasible for our system model or where inconsistencies occur.

To verify the requirements we will use queries to verify safety properties as mentioned in section 2.2, which means that a single state in which the requirement is not met will cause it to be considered inconsistent. We will use queries of the form $\forall\Box x$ or in UPPAAL `A[] x` to ensure that the requirement x always holds.

Should UPPAAL find a state in which the requirement is violated, we can get a full trace from the beginning of the simulation up to the point in which the system of timed automata encountered the state violating the requirement. UPPAAL supports this using the *diagnostic trace*, which – when a verification query is violated – saves the trace from the verification utility to the simulator to review the current state and the full path of transitions that lead to it. To enable this option, open Options » Diagnostic Trace and select either Some, Shortest or Fastest. The traces given in the course of this thesis have been obtained with the option set to Some.

All examples in this chapter build on the foundation of the small example system defined in listing 4.10. A more sophisticated, in-depth example will be given in chapter 6.

Due to the way the clocks in timed automata work, each runtime clock tc has a valuation of $v(c) \in [0, 1]$ when the corresponding task's simulation state is currently neither executing nor suspended. As a consequence, we are unable to reliably check whether a task has actually just started execution based on the runtime clocks and need to resort to the location names and the broadcast channels. Since UPPAAL itself does not allow dynamic or parametrizable location names, this needs to be taken into account for several verification queries – this will be mentioned in the individual subsections.

5.1 Verification of Properties Using TCTL

In this section, we will cover the requirements which we can verify using TCTL queries and a network of timed automata representing processing environments. The requirements covered here are requirements over a single or over two functions.

We start with the maximum execution time of a function, proceed with the data age and then show how to verify the periodicity requirement. In addition to the defined requirements, we will show how to verify whether a system is schedulable within the time bounds of the simulation and to check whether a task is actually ever run.

5.1.1 Maximum Execution Time of a Function

Due to the existence of runtime clocks, the maximum execution time of a function as defined in section 3.3.1 can easily be verified.

To check whether for a given system the execution time of a task τ_n implementing f_n does not exceed a given bound MET(f_n), we can use the following TCTL query:

$$\forall\Box\tau_n rt \leq \text{MET}(f_n)$$

As the UPPAAL model is time-bound by the constant `TIME_MAX`, we need to prepend a condition to account for this upper bound. Otherwise a system state in which the requirement is not fulfilled can always be found outside of the valid time bounds, as the automata enter the *Done* state and do not continue resetting the clocks.

We have several clocks to choose from that can act as global clocks to compare to this time bound, mainly the clock of the reference system and the runtime or data age clocks of task ID 0. Since the reference system may be declared with varying names, we will use `rt_c[0]` as the global clock for the following verification queries.

We can then transfer the TCTL query almost exactly, prepending this condition to account for the time-bound simulation. Assuming τ_n is represented using a task with the ID n in the UPPAAL model, we use query 5.1 to check for validity of the requirement.

```
A[] (rt_c[0] <= TIME_MAX) imply (rt_c[ n ] <= MET(f_n) )
```

Query 5.1: Verification query to check whether MET(f_n) holds true

For our example system, we can use the query `rt_c[2] <= 11` here, which evaluates to true. Since the task with the ID 2 has a WCET of 11 and the periods on the example systems do not lead to task preemption, the execution time never rises above the WCET. Choosing any value lower than 11 here obviously results in a verification failure.

5.1.2 Maximum Data Age

Just like with the function execution time, the verification of the data age requirement from section 3.3.4 was made easy in the model-building process by introducing the relevant clocks. Assuming f_n is implemented by τ_n and f_m is implemented by τ_m, we can verify the maximum data age requirement MDA(f_n, f_m) with using this TCTL query:

$$\forall\Box\tau_m start \rightarrow \tau_n da \leq \text{MDA}(f_n, f_m)$$

In UPPAAL, we need to know the task ID of τ_n, which we will consider to be n, as well as the processing environment ID i and the task array index j of τ_m – note that the indexing starts at the value 1. With this information, we can apply the template in query 5.2 to check whether MDA(f_n, f_m) is upheld by the given system.

```
A[] (PE i .t j _start imply (da_c[ n ] <= MDA(f_n,f_m) ))
```

Query 5.2: Verification query to check whether MDA(f_n, f_m) holds true

In our example, we could successfully check MDA(f_1, f_3) = 10 using `PE2.t2_start imply (da_c[1] <= 10)`, but trying the same with a value of 8 instead of 10 would fail.

5.1.3 Periodicity

The verification of the periodicity requirement defined in section 3.3.3 can be achieved easily as well, due to the fact that the data age clock is reset in the time step *after* the finish state, not before. We can use the following TCTL query to verify a periodicity requirement PER(f_n):

$$\forall\Box(\tau_n\, finish) \rightarrow \tau_n da \leq \mathrm{RPT}(f_n)$$

In UPPAAL, this query can be used as shown in query 5.3. For this we need to know the task ID n of τ_n, as well as the processing environment ID i and the task array index j of this task.

```
A[] (PE i .t j _finish imply (da_c[ n ] <= ( PER(f_n) )))
```

Query 5.3: Verification query to check whether PER(f_n) holds true

For our example system, we would check PER(f_3) = 32 using the query `A[] (PE2.t2_finish imply (da_c[3] <= 32))`, which holds true. Note that the query `A[] (PE2.t2_finish imply (da_c[3] <= 30))` will fail although TG(τ_3) = 30, due to the variable runtimes. If a task instance finishes within its task's BCET and the following instance finishes with the task's WCET, the span checked by PER(f_n) is larger than the period of τ_n.

5.1.4 Schedulability and Queue Overload

While not specifically a requirement, we can check whether a system might encounter an error during the simulation, namely a runtime scheduling error or a queue overload. As mentioned in section 4.3.1, the task queue of a system in UPPAAL cannot grow infinitely large, but is bound by the constant `TASK_QUEUE_MAX`. A processing environment automaton transitions to the overload state when the amount of instances in its task queue rises above the threshold determined by the constant `TASK_QUEUE_OVERLOAD`. While this is a limitation of the model, we have shown in section 4.3.2 that using the values proposed for the task queue limits, depending on the total number of tasks in the simulation, this indicates a non-schedulable system.

To check whether this is the case for a given processing environment i, the query shown in query 5.4 can be used.

```
A[] not PE i .Overload
```

Query 5.4: Verification query to check whether a processing environment is overloaded

A runtime scheduling error occurs when assumptions required for the correct functioning of a dynamic scheduling implementation are violated. Since we assume scheduling implementations to be functionally correct, such a scheduling error can only occur when the scheduling parameters are invalid or cannot be applied to the current system. For the two scheduling strategies implemented, the only scheduling error – apart from an ever-growing queue detected by the overload – that can occur during runtime is when an EDF system's local time is greater than any of the deadlines of task instances in the processing environment's queue. In hard real-time systems, which we are simulating, a passed deadline leads to a system being classified as non-schedulable using the current scheduling parameters[15, 30].

Since this error is detected from inside the EDF templates as shown in section 4.3.1, an EDF scheduling error can be detected using query 5.5 for an EDF processing environment automaton i, which can be

combined with the query for queue overload to query 5.6 to check for both missed deadlines as well as an overly full queue.

```
A[] not PE i .SchedulingError
```

Query 5.5: Verification query to check whether an EDF processing environment encounters a scheduling error

```
A[] not (PE i .SchedulingError or PE i .Overload)
```

Query 5.6: Verification query to check whether an EDF processing environment does not encounter an error during runtime

All auxiliary error states are states with no outgoing edges. While the *Done* state of a processing environment template can transition to itself, this is not possible in the error states. Since these error states are the only states in the whole automaton without outgoing edges, we can also check whether it is possible for the automaton to encounter a deadlock. To check whether a network of automata proceeds through the whole bound simulation without the possibility of any automaton entering an error state, we use query 5.7 to check for deadlock-freeness.

```
A[] not deadlock
```

Query 5.7: Verification query to check whether a network of automata is deadlock-free

5.1.5 Task Execution

To check whether a task instance is actually ever executed, we can use the following TCTL query:

$$\exists \Box \tau_n rt > tick$$

If there is a state in which $\tau_n rt \notin [0, tick]$, we know that the task was executed at least once as there exists at least one system state in which the corresponding runtime clock was not reset. Note that this only works for $WCET_\tau > tick$, otherwise we are confronted with the problem arising from the continuous increment of clocks as detailed in section 4.2.2.

In UPPAAL, this TCTL query can be transferred as Query 5.8, an example again requiring the conditional prefix known from section 5.1.1.

```
E[] (rt_c[0] <= TIME_MAX) and (rt_c[ n ] > 1)
```

Query 5.8: Verification query to check whether task τ_n is executed

5.2 Verification Using Additional Automata and TCTL

Properties that require information about a time span rather than a single point in time or need to react based on previous input or actions cannot be expressed using simple TCTL queries. All properties detailed in section 5.1 expressed that either at a certain point in time or at all times the corresponding property must hold. In this chapter, we will introduce additional automata into the simulated system to have the ability to react to multiple events in a single verification run, enabling state-aware verification for our model.

For the requirements covered in section 3.3, this is necessary to verify both the synchronization constraint as well as the maximum reaction time requirement using our model. Note that both of these can range over an arbitrary amount of functions and span multiple events over a time span. For both of these, there exists no single event we could observe from which we could check whether the requirement is valid by just using the TCTL subset as defined in section 2.2.

In [1] where TCTL was originally introduced, the ability to limit the scope of temporal operators to a certain time span, was introduced. As of now this has not been implemented in any model-checking tool publicly available, which is why we will use the approach covered in this chapter to allow for these requirements to be verified using UPPAAL.

We will create the templates for the verification automata as parametrizable templates just like with the automaton templates in section 4.3.2, to allow a fast instantiation of verification automata using a set of parameters defining the requirement.

5.2.1 Synchronization

As per section 3.3.5, the synchronization requirement specifies that a given set of tasks must finish within a given time span, in any order. To verify these requirements, we introduce an automaton template with an internal clock that starts counting when it receives a finish broadcast from any of the tasks in the given set and only resumes its continuous clock resets when a finish broadcast from all other tasks in the set was received. The maximum allowed time span shall be given as parameter, such that it is available from inside the automaton. With this available, the automaton can compare the current value of the internal clock to the given requirement each tick and switch into an error state if its clock value is larger, since then not all tasks finished within the given time span.

Given the template instantiation parameters `const int[1, TASK_AMOUNT] t_id[n], const int[0, MAX_PERIOD] sync_max` with $n = 2$, a template for a synchronization automaton can be found in figure 5.1, templates for the verification of more than two tasks can be found in appendix B starting on page XIV. The code for the templates of synchronization automata can be found in listing 5.1. It is rather short and requires only the change of a single constant `TASK_COUNT` in order to be adjusted to a different amount of tasks.

```
const int [0, TASK_AMOUNT] TASK_COUNT = 2;

clock c;
clock tc;
bool triggered[TASK_COUNT];

void trigger(int[0, TASK_COUNT] id) {
    triggered[id] = true;
}

// return the total number of task finishes that were already triggered in this run
int[0, TASK_COUNT] count_triggered() {
    int[0, TASK_COUNT] count = 0;
    for (i : int[0, TASK_COUNT - 1]) {
        if(triggered[i])
```

```
                count++;
        }
        return count;
    }

    // finish the run and reset all triggers
    void reset_triggers() {
        for (i : int[0, TASK_COUNT - 1]) {
            triggered[i] = false;
        }
    }
```

Listing 5.1: Code for the two task synchronization template from figure 5.1

From its initial location, the automaton transitions to a committed location with four outgoing edges once each tick. If the amount has not yet recorded a finish event in this run, the automaton returns to the initial location, resetting both its internal clock c as well as its tick clock tc. If at least one finish event has occurred during this run, but the clock is still supposed to be increased since not all tasks corresponding to the synchronization constraint have finished yet, the automaton also returns to the initial location, but only resets its tick clock tc. Should the automatons internal clock already be above the threshold given by the SYNC(f_1, f_2) requirement, the automaton enters the *Error* location to indicate that the requirement is not met.

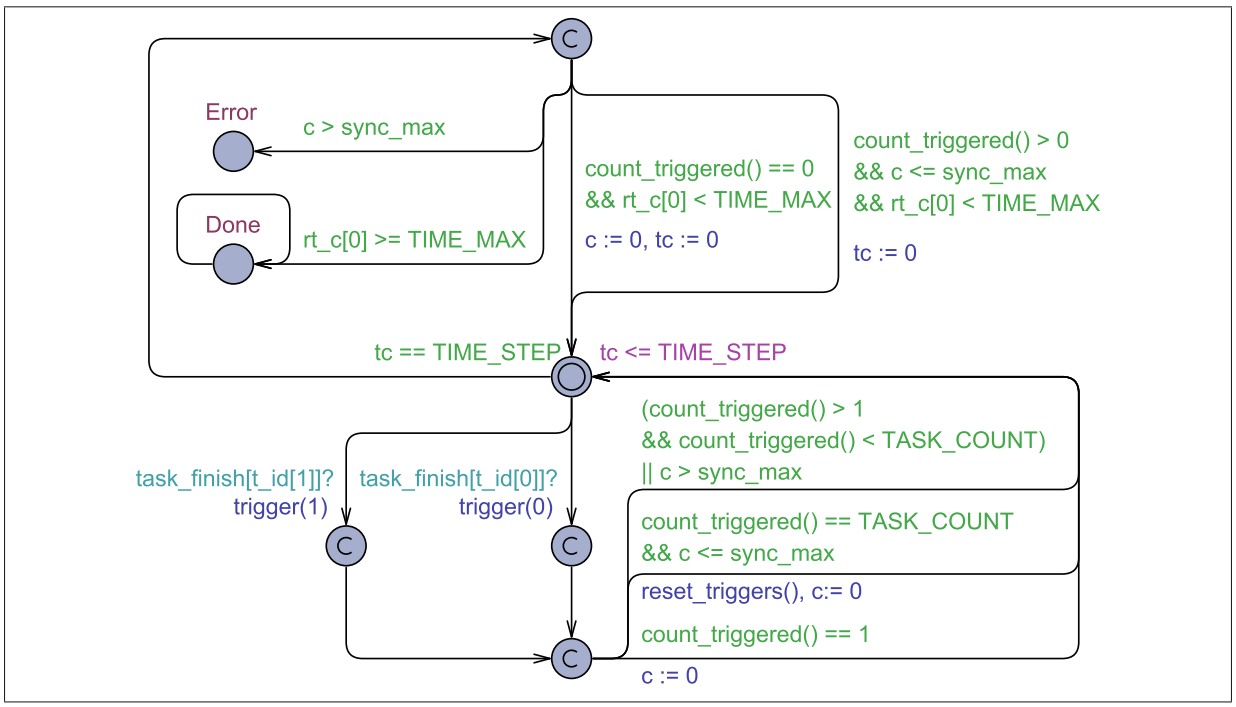

Figure 5.1: Template of a synchronization automaton for two tasks

When the automaton is in its initial location, it reacts to the broadcast on the channels belonging to the tasks given as parameters. Each time such an event is recorded, the `trigger(ID)` function shown in listing 5.1 is called, storing that the event has occurred. If this was the first event recorded in the run, the clock is reset to properly count the time since the first occurrence and to avoid off-by-one errors. If this was the last event to be recorded in this run, the triggers are reset and the internal clock c is reset as well, marking the end of this run. If it is anything in between, the automaton transitions back to the initial location without any updates or resets. Note that in the template shown in figure 5.1, this edge cannot be taken, since for two tasks there cannot be anything between the other two edges. But for synchronization automata with more than two tasks, this edge is important. An example automaton template for verifying the synchronization of three tasks is shown in figure B.3.

To incorporate the synchronization automaton into the system, we need to instantiate the template and append it to the system directive as shown in listing 5.2.

```
const int sync_t[2] = {2, 3};
sync1 = SYNC_2T(sync_t, 12);

system PE1, PE2, sync1;
```

Listing 5.2: Changes to the system declarations to incorporate the synchronization automaton

In the configuration shown, the synchronization automaton checks whether the tasks with the tasks with the IDs 2 and 3 always finish within a time range of 12 of each other and switches into the *Error* state when this is not given. Because of that, we can simply check whether the requirement is upheld using the following TCTL query:

```
A[] not sync1.Error
```

Query 5.9: Verification query to check whether the synchronization requirement is upheld

Given the system from section 4.3.3 and the synchronization automaton instantiated in listing 5.2, this query tells us that the property is satisfied. Reducing $\mathrm{SYNC}(f_2, f_3)$ to the value 10 by setting `sync1 = SYNC_2T(sync_t, 10)`, the query will fail.

5.2.2 Maximum Reaction Time of an Event Chain

Since event chains model a sequentially ordered set of events and we only consider task start and finish events here, we can also create parametrizable templates for event chains. The idea is to react to move through the locations receiving the start and the finish events of the contained tasks. To actually catch all valid flows through the event chain, we introduce non-determinism, such that the event chain automaton can switch to the start location from every other location using a non-guarded transition, resetting its internal clock as well as its tick clock. As the automaton is required to transition when receiving on the broadcast channel, and because the non-determinism introduces the ability for the automaton to always return to its start location, verification of the safety property using $\forall\Box$ ensures that there is no single valid event chain flow that violates the requirement.

As parameters to the template, we can simply pass an array of numeric task IDs representing the tasks in the order they appear in their event chain. For an event chain template of `n` tasks, we define the parameters

as `const int[1, TASK_AMOUNT] t_id[n]`, for n = 3 the template would look like figure 5.2, other examples can be found in appendix B, for example figure B.4. The code of each event chain automaton simply contains `clock c; clock tc;` and does not require any adjustments when changing the amount of tasks. When the automaton is properly defined, we can check for the MET requirement using the event chain automatons internal clock.

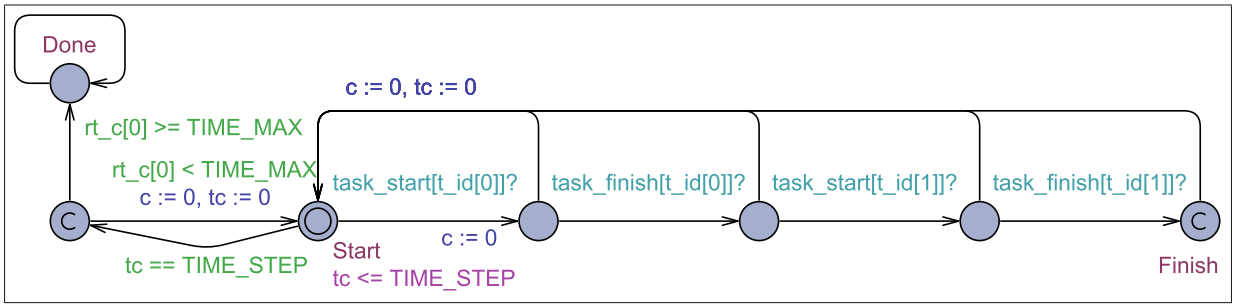

FIGURE 5.2: Template to verify an event chain consisting of three tasks

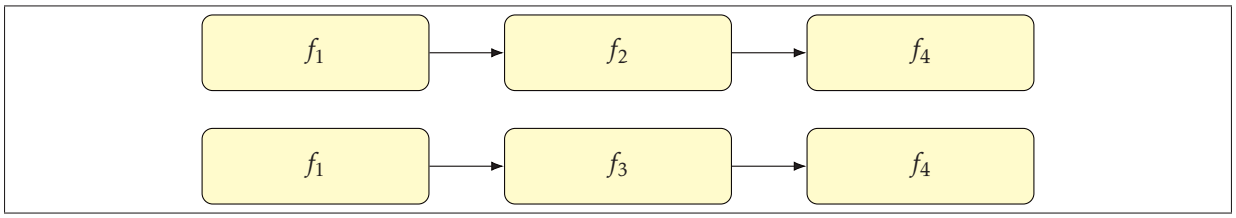

FIGURE 5.3: Simple event chains constructed from the four given functions

Given the example system already defined in section 4.3.3, we can construct two simple event chains $ec_1 = (\{f_1, f_2, f_4\}), ec_2 = (\{f_1, f_3, f_4\})$, shown in figure 5.3. We construct the two event chain automata and incorporate them into our system as shown in listing 5.3.

```
const int ec_v1[3] = {1, 2, 4};
const int ec_v2[3] = {1, 3, 4};
ec1 = EC_3T(ec_v1);
ec2 = EC_3T(ec_v2);

system PE1, PE2, ec1, ec2;
```

LISTING 5.3: Changes to the system declarations to incorporate the event chain automata

We can now use a query as shown in query 5.10 to check whether each complete run of the event chain was within the specified bounds.

```
A[] (ec1.Finish imply ec1.c <= MRT(ec1) )
```

QUERY 5.10: Verification query to check whether MRT(ec_1) is upheld

This of course only works when the event chain does reach a finish state, which might not be true in every case. An easy and fast query to check whether this happens is shown in Query 5.11, because of the non-determinism `E<>` needs to be used instead of `E[]`.

```
E<> ec1.Finish
```

Query 5.11: Verification query to check whether ec_1 finishes at least once

For our given example system, we can verify that the requirement $\mathrm{MRT}(ec_2) = 60$ holds true using the query `A[] (ec2.Finish imply ec2.c <= 60)`. Using a value lower than 60 leads to a verification error indicating an inconsistency. For ec_1 this limit is even higher as $\mathrm{MRT}(ec_1) = 72$ is the lower bound for a consistent requirement given the example system.

5.3 Limitations of the Model and the Verification

After having built a model in chapter 4 and having introduced how to verify the requirements introduced in section 3.3 based on this model, we will assess the limitations that come with this proposed model.

5.3.1 Complexity

The first problem we will cover is *complexity*, which unfortunately is a problem common to model-checking in general[16, 35]. For a single timed automaton, the verification problem has been shown to be **PSPACE**-complete[1]. The **PSPACE** complexity class describes decision problems that can be solved by a Turing machine using a polynomial amount of space[28].

The verification of a network of timed automata is done by computing the product automaton, applying state reduction as well as various other optimization techniques and applying the verification queries on this single automaton[18]. Since we know that **PSPACE** = **NPSPACE**[44] and that the verification problem for a timed automaton is **PSPACE**-complete, we can deduce that the verification of an arbitrarily large network of timed automata is **PSPACE**-complete as well. There has been a lot of work on making the verification of timed automata and timed systems in general more effective[5, 7], but **PSPACE**-complete problems are known to not scale well. Reducing a **PSPACE**-complete problem like verification in timed automata to a problem of a lower complexity class would essentially mean to solve all problems in **PSPACE** since they can be reduced to the **PSPACE**-complete problem[28].

The model of processing environments proposed in chapter 4 has been built in a way to keep the state space as small as possible. The automata act deterministic and only use non-determinism for the finish time of task instances to simulate variable execution times, but even this adds a non-negligible amount of complexity to the model.

To at least partially mitigate this issue, we recommend reducing the defined system on a per-query basis, only introducing the automata into the system which affect the result of the query. Since we only model timing behavior and no interaction between systems, the processing environment automata act entirely independent of each other. This means that for each query, we only need to enable the automata which contain parts of the property that is being verified.

For the maximum execution time and the periodicity we only need the one automaton representing the single processing environment on which the corresponding task is run. For the verification of the maximum data age we need to simulate up to two processing environments, such that the tasks corresponding to the functions in the MDA requirement are part of the system.

When verifying synchronization or maximum reaction time requirements, all processing environments

which contain tasks of the corresponding function sets need to be modeled as well as the synchronization or event chain automaton itself. Since the sets of functions for these requirements can grow arbitrarily large, the system might eventually get too complex to model-check. For the examples used in this thesis this turned out not to be a problem, but it is important to be aware of this possible issue when scaling this approach to larger systems.

5.3.2 Event Chains on the same Processing Environment

When we verify the maximum runtime of an event chain, there are slight differences in how broadcasts are handled in the verification between consecutive tasks running on the same processing environment and those running that are distributed to different processing environments. Since the model checker always assumes the worst possible case when verifying safety properties, the maximum runtime of a distributed event chain is based on the assumption that should a task finish on one processing environment and the consecutive task in the chain starts on another in the same time step, the second task broadcasts first and thus the event chain does not move forward two locations in this single time step.

When the consecutive functions are distributed to the same processing environment, this cannot be achieved, since the simulation of the finishing task must be done before the start of the next instance can be simulated. This effectively means that when the functions f_1, f_2 are implemented as tasks τ_1, τ_2 which are deployed on the same processing environment pe_1, an event chain automaton defined for an event chain composed of these two functions would receive both the `task_finish[1]?` and the `task_start[2]?` broadcasts sequentially in one tick and transition over both edges accordingly.

This is visualized in figure 5.4 using the functions f_1, f_2, f_3, which we assume to be implemented by τ_1, τ_2, τ_3 respectively with $T_{pe_1} = \{\tau_1, \tau_2\}$ and $T_{pe_1} = \{\tau_3\}$. We can observe that at t_2, both the finish and the start event are considered to be part of the event chain flow, since τ_1 and τ_2 are deployed on the same processing environment and there is no way for τ_2 to broadcast its start before τ_1 broadcasts its finish event – assuming that τ_2 does not preempt τ_1.

In t_3, we have a similar situation, with tasks adjacent in the chain but distributed to different processing environments. Like defined in section 3.1.2, the flow does not include two event occurrences at the same time here and instead the next occurrence of $start_{f_3}$, occurring at t_4, is included in the flow.

Although this violates the definition of an event flow given in section 3.1.2, this can also be considered to be a more accurate implementation of the behavior in distributed systems. When functions are deployed to multiple processing environments, we can assume that data must be transferred in the actual implementation of the automotive software system. Even if we assume that this data is sent and received in an instant, a *race condition* might occur, a state in which it cannot be accurately predicted whether the function on the receiving processing environment started before or after receiving the data, effectively creating an unpredictable system[30]. For functions on the same processing environment this cannot happen, since the data does not need to be transferred.

We will not consider it to be an actual error in the model, although it does violate the initial definition of the event flow, but it needs to be kept in mind when modelling and verifying systems. As will be seen in section 6.2, this behavior is consistent with that of mature testing tools.

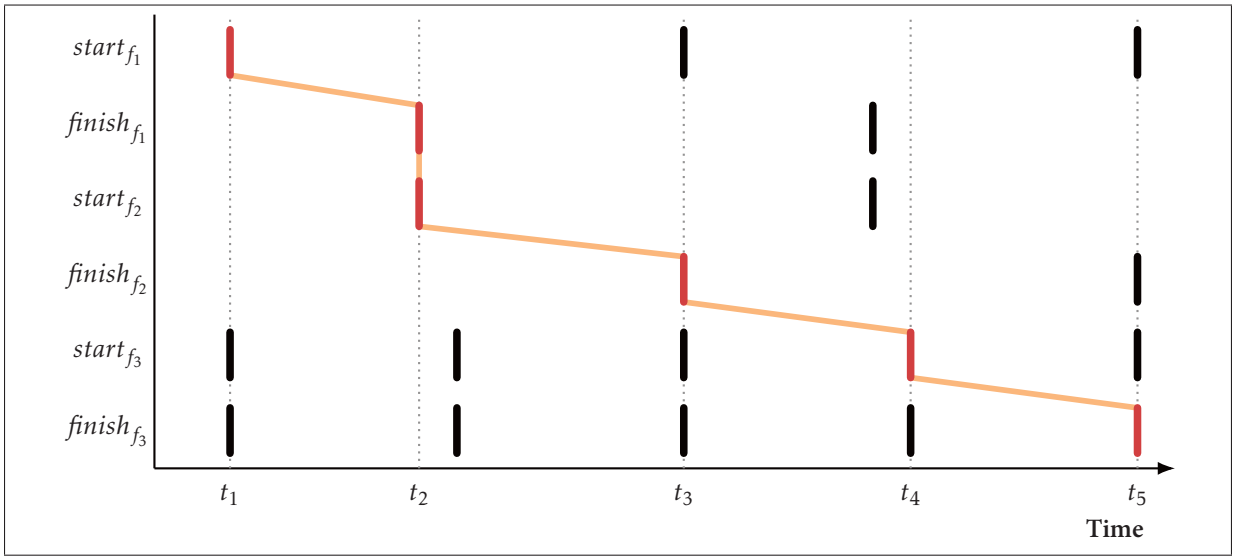

FIGURE 5.4: Verification of distributed event chains

5.4 Dealing with Infeasible Requirements

The verification method introduced in this chapter allows us to identify unrealizable requirements in distributed automotive software systems. Although actual solutions are unique to a system, we can give general strategies that help in dealing with requirements that have been deemed inconsistent.

Should the maximum execution time of a function not be met, it might help to adjust the corresponding task's scheduling parameter to assign a higher priority. It could also be moved to a less utilized processing environment to prevent frequent preemption.

Both of these are also valid strategies when the periodicity constraint of a function is not met, but in this case, it might also help to adjust the time grid, since the periodicity is very dependent on the chosen period.

When the maximum data age between two functions f_1, f_2 is violated, the easiest way to deal with this would be to distribute the tasks to the same processing environment and to choose the scheduling parameters in a way such that an instance of the task implementing f_2 is run directly after an instance of f_1. Since this often is not possible, other ways to possibly reduce the data age would be to lower the period of the task implementing f_1 or to adjust the scheduling parameters in a way that both are run in the same time grid, but each task instance of τ_2 runs after an instance of τ_1 in each period, which can be achieved by assigning a relatively low priority to the task τ_2 implementing f_2 and a high priority to the task τ_1 implementing f_1.

To achieve consistent synchronization of a set of functions and a maximum reaction time for an event chain, no generalized approach can be given, as solutions are heavily dependent on both the systems and the requirements defined. If the requirements are not inconsistent according to section 3.3.6, it is possible to create a system design consistent with these requirements.

It is worth noting that addressing some currently infeasible requirements might make it necessary to temporarily invalidate other, currently consistent requirements. Development of consistent systems is thus done in iterations, moving towards a final system design in several steps, possibly invalidating previously met requirements for several iterations in between to address others.

6 Case Study

Using the approach detailed in the previous two chapters, we will now perform a case study on a fictional brake-by-wire architecture.

We will start by outlining the example that will be used in this chapter and show how to formalize this according to section 3.3. In the course of this chapter, we will iteratively alter this model, using the developed UPPAAL model to assist in the development of a consistent system design to fulfill all given requirements.

6.1 A Distributed Brake-by-Wire Function

The example that we will use here is a fictional brake-by-wire architecture, inspired by the example given in [9]. A brake-by-wire system aims to replace the mechanical linkages between the driver and the vehicle utilizing electronical systems[34]. This would carry several benefits, including 'component reduction, weight reduction, potential for improved vehicle performance, increased cabin space, removal of the steering column, ergonomic and crash compatible mounting of controls, complete access to vehicle dynamics control, 'plug, play and bolt' modularity, software upgrading, and potential for better fuel economy' [34, p. 3].

We introduce a fictitious distributed function representing the software of such a brake-by-wire architecture. From a high-level point of view, this distributed function works by periodically polling the angle of the brake pedal to receive input from the driver, converting this to an amount of force to apply to the brakes, applying additional assistance systems like electronic brakeforce distribution, and then activating the corresponding actuators in the brakes with the desired force. Some of the assistance systems can be enabled and disabled by the driver, which is why we additionally pull the configuration data from the system to determine which assistance systems shall be applied.

In addition to this driver-based brake routine, our system shall include an emergency brake assistant, periodically analyzing data from various sensors of the car to detect an emergency, and activate the brake actuators as fast as possible in case of a detected emergency, bypassing the assistance systems to prevent any reduction of the brake force and instead only applying rudimentary stabilization measures.

A functional decomposition of our brake-by-wire architecture is shown in figure 6.1. For the functions shown, we are given several real-time requirements which shall be fulfilled in the final system:

- the function calculating the force that shall be applied to the brakes must always finish at most 28 ms after it started and the calculations must be done at least every 40 ms
- the sensor data used by the assistance systems may at most be 12 ms old and the already preprocessed data from the brake pedal may at most be 16 ms old when the calculations of the brake force start
- the driver-triggered brake routine, from the polling of the brake pedal angle to the finished activation of the brake actuators, must always finish within 110 ms

- the path from the main brake controller calculating the brake force up to the finished activation of the actuators may at most take 80 ms and the input data to the function calculating the brake force must always be from within a time frame of 10 ms
- the emergency brake routine, from the polling of the sensor data up to the finished activation of the brake actuators, must never surpass a total of 85 ms

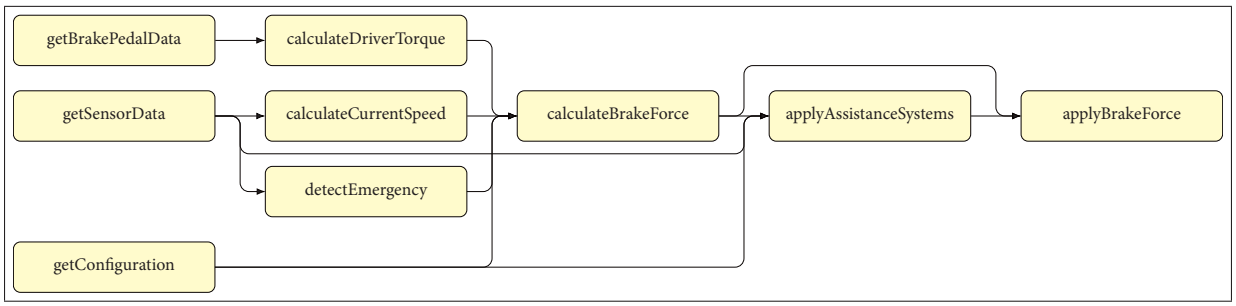

FIGURE 6.1: Functional decomposition of our brake-by-wire architecture

In section 7.3, we have included variations of this functional decomposition where the relevant paths mentioned here are highlighted. The event chain for standard, driver-triggered braking is shown in figure A.2, the one for the emergency brake routine in figure A.3 and the path from the main brake controller to the brake actuators.

6.1.1 Initial Formalization

We start with the formalization by mapping the functions shown in figure 6.1 to tasks, estimating budgets for their corresponding best-case and worst-case execution times. We assume that we have a homogeneous hardware architecture, where all tasks have a fixed BCET and WCET regardless of the processing environment they are currently deployed on.

Due to wiring constraints, we assume each additional processing environment introduced into the system to have a slight clock offset of 1 ms compared to the one added before, such that the offset between two processing environments pe_n, pe_m can be calculated as $\text{co}(pe_n, pe_m) = m - n$ ms, making the processing environment with the lowest index the reference system.

The requirements that were mentioned in the introduction to this section will be given to us in TADL2 format. A full listing of these can be found in listing A.3, which will be omitted here since all requirements were already mentioned. To get a better overview about the three event chains defined in these requirements, we have visualized them in figure 6.2.

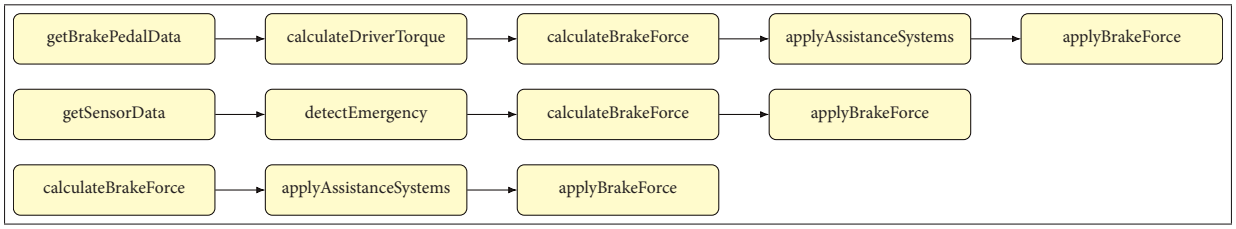

FIGURE 6.2: Event chains in the brake-by-wire architecture

Task	Function Name	BCET	WCET	Explanation
τ_1	getBrakePedalData	3	4	Receive and store information about the current brake pedal angle
τ_2	getSensorData	6	7	Receive and store information about the sensors (accelero-, gyrometer, camera, ultrasonic sensor, …)
τ_3	getConfiguration	3	5	Receive and store information about the currently selected user options (engine recuperation, assistance systems, …)
τ_4	calculateDriverTorque	2	3	Calculate relative torque from brake pedal angle
τ_5	calculateCurrentSpeed	8	10	Use stored sensor data to calculate the current speed
τ_6	detectEmergency	16	22	Use stored sensor data to detect whether an emergency situation is imminent
τ_7	calculateBrakeForce	19	26	Combine current information from the brake pedal, sensors and settings to calculate the force to apply to the brakes
τ_8	applyAssistanceSystems	13	28	Apply enabled assistance systems based on currently stored sensor data and already calculated brake force
τ_9	applyBrakeForce	7	9	Apply the final result of the force calculation to the brakes by activating the brake actuators

Table 6.1: Function mapping for our brake-by-wire architecture including estimated budgets

In order to be able to perform the verification in the next steps, we need to import this task model into UPPAAL. Apart from the task including their BCET and WCET budgets, we will also define the automata necessary to verify the requirements imposed upon the event chain reaction time and the synchronization. Now we are able to input this information in our UPPAAL system declarations as shown in listing 6.1. Since the requirements and tasks are static, these automata are static as well and do not need to be adjusted when changes are made to the system design later.

```
// Task definitions (ID, BCET, WCET)
const Task T1 = {1, 3, 4};   // getBrakePedalData
const Task T2 = {2, 6, 7};   // getSensorData
const Task T3 = {3, 3, 5};   // getConfiguration
const Task T4 = {4, 2, 3};   // calculateDriverTorque
const Task T5 = {5, 8, 10};  // calculateCurrentSpeed
const Task T6 = {6, 16, 22}; // detectEmergency
const Task T7 = {7, 19, 26}; // calculateBrakeForce
const Task T8 = {8, 13, 28}; // applyAssistanceSystems
const Task T9 = {9, 7, 9};   // applyBrakeForce

// Verification Automata
const int st1[3] = {3, 4, 5};
sync1 = SYNC_2T(st1, 10);

```

```
const int ec_v1[5] = {1, 4, 7, 8, 9};
const int ec_v2[4] = {2, 6, 7, 9};
const int ec_v3[3] = {7, 8, 9};
ec1 = EC_5T(ec_v1);
ec2 = EC_4T(ec_v2);
ec3 = EC_3T(ec_v2);
```

Listing 6.1: UPPAAL declaration of the given tasks and verification automata

Since we now have all definitions we need, we can start declaring processing environments and test the systems in UPPAAL. We will do this in multiple iterations, starting with a first concept and working our way towards a fully consistent, feasible system model.

Our first concept is composed of three processing environments, for which we split the tasks into the groups $T_1 = \{\tau_1, \tau_4, \tau_6, \tau_9\}$, $T_2 = \{\tau_2, \tau_3, \tau_7\}$ and $T_3 = \{\tau_5, \tau_8\}$. Since the accumulated WCET of the task sets is equal, we assign each task on each system the same period, just slightly over the total WCET the set. We also determine a simple OSEK scheduler with the priorities set in the order of the IDs and gain the system distribution as shown in figure 6.3.

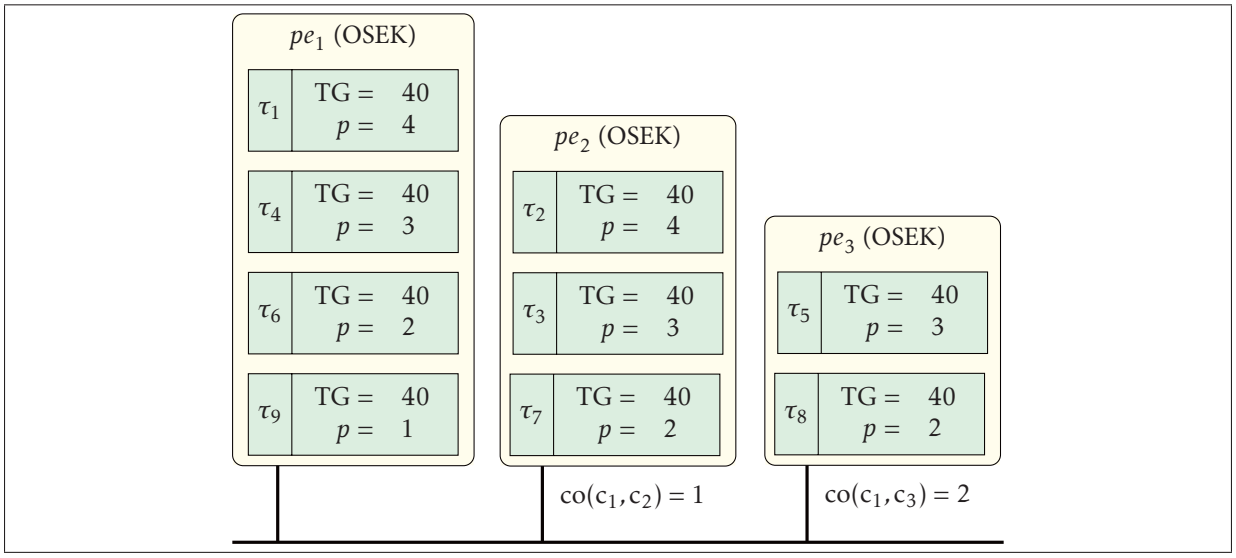

Figure 6.3: Task distribution and scheduling information of the initial model

Entering this into our UPPAAL model is straightforward, the additions to the system declarations are shown in listing 6.2. For verification purposes, we only introduce the automata required for the relevant queries into the system, as detailed in section 5.3.1. Because of this, the queries in table 6.2 have been verified using at most four automata in the system at once; the three processing environments and one verification automaton, if needed. The activated automata for each query are written as comments behind each `system` directive. Note that the query prefix for the maximum execution time has been shortened from `A[] (rt_c[0] <= TIME_MAX)` to `A[] (...)` to have a more compact and readable table.

```
// OSEK Task Definitions (Task, Priority, Period)
const OSEK_Task OT1 = { T1, 4, 40 };
const OSEK_Task OT2 = { T2, 3, 40 };
```

```
const OSEK_Task OT3 = { T3, 2, 40 };
const OSEK_Task OT4 = { T4, 3, 40 };
const OSEK_Task OT5 = { T5, 2, 40 };
const OSEK_Task OT6 = { T6, 2, 40 };
const OSEK_Task OT7 = { T7, 1, 40 };
const OSEK_Task OT8 = { T8, 1, 40 };
const OSEK_Task OT9 = { T9, 1, 40 };

// Array Compositions
const OSEK_Task PE1_Tasks[4] = { OT1, OT4, OT6, OT9 };
const OSEK_Task PE2_Tasks[3] = { OT2, OT3, OT7 };
const OSEK_Task PE3_Tasks[2] = { OT5, OT8 };

// PE Definitions Template (Task Array, Offset to Reference PE)
PE1 = PE_4T_OSEK(PE1_Tasks, 0);
PE2 = PE_3T_OSEK(PE2_Tasks, 1);
PE3 = PE_2T_OSEK(PE3_Tasks, 2);

//system PE1, PE2, PE3; // Queries 1-4 (MET, PER, MDA)
//system PE1, PE2, PE3, ec1; // Query 5 (MRT(ec1))
//system PE1, PE2, ec2; // Query 6 (MRT(ec2))
//system PE1, PE2, PE3, ec3; // Query 7 (MRT(ec3))
system PE1, PE2, PE3, sync1; // Query 8 (SYNC)
```

Listing 6.2: UPPAAL declaration of the system shown in figure 6.3

Property	UPPAAL Query	Satisfied?
$\mathrm{MET}(f_7) = 28$	`A[] (...) imply (rt_c[7] <= 28)`	✔
$\mathrm{PER}(f_7) = 40$	`A[] PE2.t3_finish imply (da_c[7] <= 4)`	✘
$\mathrm{MDA}(f_2, f_8) = 12$	`A[] PE3.t2_start imply (da_c[2] <= 12)`	✔
$\mathrm{MDA}(f_4, f_7) = 16$	`A[] PE2.t3_start imply (da_c[4] <= 16)`	✔
$\mathrm{MRT}(ec_1) = 110$	`A[] ec1.Finish imply ec1.c <= 110`	✘
$\mathrm{MRT}(ec_2) = 85$	`A[] ec2.Finish imply ec2.c <= 85`	✘
$\mathrm{MRT}(ec_3) = 80$	`A[] ec3.Finish imply ec3.c <= 80`	✘
$\mathrm{SYNC}(f_3, f_4, f_5) = 10$	`A[] not sync1.Error`	✔

Table 6.2: Queries and their results for the system specified in figure 6.3

As can be seen in table 6.2, the specified system does hold up to the maximum execution time requirement on the *calculateBrakeForce* function as well as both data age and the synchronization requirement. The first requirement is fulfilled because the system is designed in a way that no tasks are suspended and can run freely once in their assigned time grid. The other three requirements work because of the order in which the tasks are actually run, based on the priorities assigned to them. Using the knowledge gained from this example, we will start refining our system until we gain a consistent design.

6.1.2 Enhancing the System

It is apparent that using just three processing environments, we won't be able to satisfy all given real-time requirements. Since f_7 has both the $\mathrm{MET}(f_7) = 28$ and $\mathrm{PER}(f_7) = 40$ requirement with a WCET of 26,

we will introduce the new processing environment pe_4 with $T_4 = \{\tau_7\}$, so that τ_7 is the sole task running on this PE.

We look at figure 6.2 to determine the order in which the tasks would need to finish for the event chain requirements to be met. As τ_8 has such a high worst-case execution time, we redistribute the tasks to be able to let them run in a tighter time grid, leaving τ_8 as the sole to be the sole task on pe_3. Since both τ_7 and τ_8 are running on their own PE, we want to ensure that τ_4 always starts after τ_1 in a single iteration and that τ_9 starts at the beginning of an iteration in order to satisfy the constraint upon ec_1. As both τ_9 and τ_1 are running on the same PE, we assign them the highest two priorities, with τ_9 being the highest-priority task. Then, we assign τ_4 the lowest priority on its PE, such that the order is like that of the event chain.

This at the same time should take care of the MDA(f_4, f_7) requirement, and to satisfy MDA(f_2, f_8) we set the task priority of τ_2 to the lowest of its PE, so the time between its end and the start of τ_8 in the next period is as small as possible. With these changes, we gain the model shown in figure 6.4, which can be transferred to our UPPAAL as shown in listing A.4.

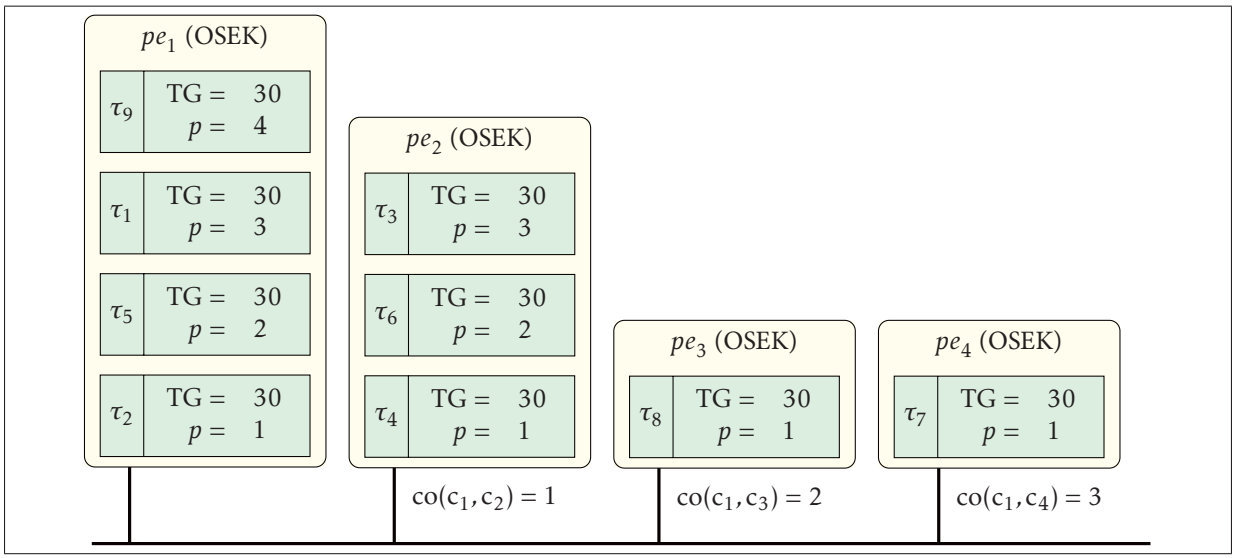

FIGURE 6.4: Task distribution and scheduling information after the first alterations

Property	UPPAAL Query	Satisfied?
MET(f_7) = 28	`A[] (...) imply (rt_c[7] <= 28)`	✓
PER(f_7) = 40	`A[] PE4.t1_finish imply (da_c[7] <= 40)`	✓
MDA(f_2, f_8) = 12	`A[] PE3.t1_start imply (da_c[2] <= 12)`	✓
MDA(f_4, f_7) = 16	`A[] PE4.t1_start imply (da_c[4] <= 16)`	✓
MRT(ec_1) = 110	`A[] ec1.Finish imply ec1.c <= 110`	✗
MRT(ec_2) = 85	`A[] ec2.Finish imply ec2.c <= 85`	✓
MRT(ec_3) = 80	`A[] ec3.Finish imply ec3.c <= 80`	✗
SYNC(f_3, f_4, f_5) = 10	`A[] not sync1.Error`	✗

TABLE 6.3: Queries and their results for the system specified in figure 6.4

When we run the queries again, we gain the results shown in table 6.3. In addition to the first four requirements, this new system also fulfills the MRT(ec_2) requirement. Unfortunately our alterations caused the SYNC(f_3, f_4, f_5) requirement to not be fulfilled anymore.

Using diagnostic traces, we obtain information about why MRT(ec_1) is not yet fulfilled, even though we arranged the tasks to execute in the defined order. We get a debug trace indicating that to satisfy this requirement over ec_1, we need to pay attention to the offset between the processing environments. Because τ_8 on pe_3 has a WCET of 28 and the PE has an offset of two to the reference system, the data does not arrive in time for the start of τ_9 on pe_1, which is now the first executed task there. Technically, both the start of τ_9 and the finish of τ_8 lie on the same tick, but because of the behavior of event chains explained in section 5.3.2, this does not contribute to a valid flow. Swapping the tasks of pe_2 and pe_3 should enable us to satisfy both MRT(ec_3) and at the same time MRT(ec_1).

Since τ_6 has such a large WCET and a large span between BCET and WCET, it is generally hard to predict its behavior, making it difficult to conform to the synchronization constraint. We introduce yet another processing environment, pe_4, and let τ_6 be the sole task on it. To have more fine-grained control in regard to the timing of the processing environments with multiple tasks, we let pe_1 and pe_3 be EDF-scheduled from now on. We initially distribute the deadlines in a way that represent the current OSEK priorities.

The last problem we need to deal with is the fact that the task τ_2 finishes too early in the period cycle, we would rather have it very close to the end. We solve this by introducing a simple delay task τ_{10} with $WCET_{\tau_{10}} = BCET_{\tau_{10}} = 13$ and change the scheduling parameters such that it is the first task of pe_1 to run. This enables the full synchronization including τ_5 and also restores the validity of all chains that require τ_2.

The task distribution shown in figure 6.5 shows the system design we arrive at, listing A.5 lists the system declaration in UPPAAL. As can be seen in table 6.4, this system now fulfills all requirements and all queries succeed.

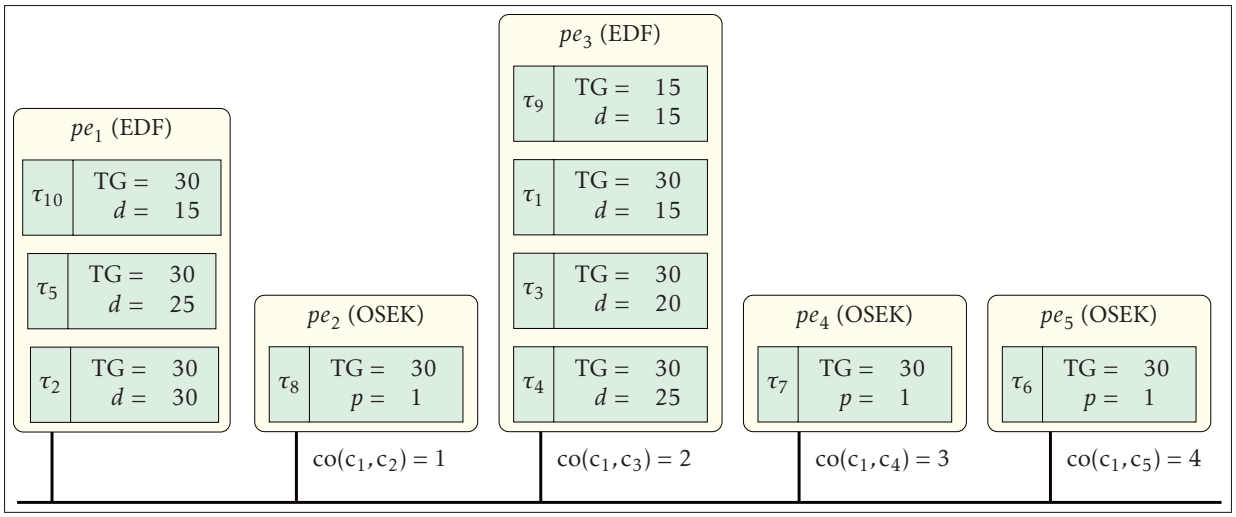

FIGURE 6.5: Task distribution and scheduling information after the second round of enhancements

In just two iterations we refined our initial model from figure 6.3 to a fully consistent system design, adding two process environments as well as a delay task. Between these iterations, we have varied the scheduling parameters heavily, showcasing possible solutions to problems that might occur. We have also shown that in the enhancement of systems, some requirements that were previously valid will be – at least temporarily – violated. Comparing the information about previous system designs with the current can give hints on why that is the case and how this problem might be solved.

Property	UPPAAL Query	Satisfied?
MET(f_7) = 28	`A[] (...) imply (rt_c[7] <= 28)`	✔
PER(f_7) = 40	`A[] PE4.t1_finish imply (da_c[7] <= 40)`	✔
MDA(f_2, f_8) = 12	`A[] PE2.t1_start imply (da_c[2] <= 12)`	✔
MDA(f_4, f_7) = 16	`A[] PE4.t1_start imply (da_c[4] <= 16)`	✔
MRT(ec_1) = 110	`A[] ec1.Finish imply ec1.c <= 110`	✔
MRT(ec_2) = 85	`A[] ec2.Finish imply ec2.c <= 85`	✔
MRT(ec_3) = 80	`A[] ec3.Finish imply ec3.c <= 80`	✔
SYNC(f_3, f_4, f_5) = 10	`A[] not sync1.Error`	✔

TABLE 6.4: Queries and their results for the system specified in figure 6.5

6.2 Comparison with SymTA/S

A tool to analyze timing in real-time automotive system using statistical analysis is *SymTA/S*, developed by Symtavision, which is now part of Luxoft. SymTA/S is a tool for the 'for a systematic description and solution of integration tasks when it comes to timing-related issues' [52]. It 'generates a mathematical model. After being solved quickly, it provides information about the system timing behavior, and identifies worst case configuration parameters automatically.' [52] Although the tool is mainly to be used in later stages of development, where a lot more data is already available, it is possible to model systems using the bare minimum of data available for scheduling analysis, which is the data we used as well.

To compare the result we got from our model to those that can be obtained using statistical analysis in SymTA/S, we want to model our final system design from figure 6.5, model it in the tool and compare the results we obtain to those can get using our UPPAAL model. Unfortunately SymTA/S does not yet implement an EDF scheduler, only the four OSEK variations 'GenericOSEK, ERCOsek, RTAOSEK, AutosarOS' [54] of which *GenericOSEK* is comparable to the OSEK scheduler introduced in section 3.2.4 and implemented in section 4.3.1. Since the tool does not support the simulation of systems with 100% utilization, we also had to slightly lower the worst-case execution times of the tasks to achieve a comparable system. The UPPAAL model that represents the system we used for the tests in SymTA/S can be found in listing A.6; this system model will be used for the comparisons in this chapter. Given the defined system, a single time step in UPPAAL represents 1 ms in SymTA/S, such that obtained results are arithmetically comparable.

In SymTA/S, each task can be synchronized to a time source and can have an offset assigned, relative to this clock. To incorporate the offsets between processing environments, we have defined one global reference clock in SymTA/S and manually given each task the offset corresponding to the clock offset of the processing environment it belongs to in our model.

The testing of the maximum execution time and periodicity requirements gave the exact results we obtained using the verification in UPPAAL. The verification of the maximum execution time as we defined it in section 3.3.1 is done by testing for the *max. Effective Execution Time* in SymTA/S, which is another name for the same concept, just like *Gross Execution Time* shown in figure 3.3. In addition to this requirement, SymTA/S also allows testing for the minimum or maximum *Total Load*, *Core Execution Time*, *Response Time*, *Activation Distance*, *Start Distance* and *End Distance*[53]. Using the *End Distance*, we could also easily test for the periodicity requirement as introduced in figure 3.13, which also returned the expected result – an indication that the requirement is met and the system passes the test.

When testing for these two requirements using SymTA/S, the tool generates a textual response to indicate whether the given system satisfied the constraint. In the case of a successful test, it provides several graphs of timing-related properties, for example an indication of the distribution of a task's execution time.

Using the implemented path system, it is possible to define event chains from a set of tasks. For these paths, some constraints can be tested, which are called the *ReactionSemantic*, *MaxAgeSemantic* and *StrictOrder*[55]. The first of these constraints allows us to determine the reaction time of event chains using paths in SymTA/S. For successfully tested systems, Gantt charts for the worst-case analysis of such a reaction time can be generated. An example is shown in figure 6.6, displaying the determined worst-case reaction time for the path corresponding to ec_1 in our model. For this worst-case analysis, the setting *Full Offset* has been used, which 'considers offsets in full accuracy and provides Gantt Charts' [54]. The documentation warns that '[t]his can slow down the analysis and increase memory requirements of SymTA/S - especially when many synchronized sources and many tasks are analyzed' [54], hinting that similar problems might be encountered compared to what we detailed in section 5.3.1 regarding our approach.

When we query the system defined in listing A.6 for the maximum reaction time of ec_1, we can go as low as $\mathrm{MRT}(ec_1) = 93$ before the query is returned as invalid. We use the query for $\mathrm{MRT}(ec_1) = 92$ to get a trace in order to obtain more information about this discrepancy.

With the first clock in the vector being the global clock and the second one being the internal clock c of the event chain automaton and omitting the names of anonymous locations, we get the following trace:

$$\cdots \rightarrow \langle \mathit{Start}, \begin{pmatrix} 8 \\ 1 \end{pmatrix} \rangle \xrightarrow{\texttt{task\_start[1]?}} \left\langle\!\left\langle \begin{pmatrix} 8 \\ 0 \end{pmatrix} \right\rangle\!\right\rangle \xrightarrow{\texttt{task\_finish[1]?}} \left\langle\!\left\langle \begin{pmatrix} 12 \\ 4 \end{pmatrix} \right\rangle\!\right\rangle \xrightarrow{\texttt{task\_start[4]?}} \left\langle\!\left\langle \begin{pmatrix} 15 \\ 17 \end{pmatrix} \right\rangle\!\right\rangle$$
$$\xrightarrow{\texttt{task\_finish[4]?}} \left\langle\!\left\langle \begin{pmatrix} 17 \\ 9 \end{pmatrix} \right\rangle\!\right\rangle \xrightarrow{\texttt{task\_start[7]?}} \left\langle\!\left\langle \begin{pmatrix} 33 \\ 25 \end{pmatrix} \right\rangle\!\right\rangle \xrightarrow{\texttt{task\_finish[7]?}} \left\langle\!\left\langle \begin{pmatrix} 52 \\ 44 \end{pmatrix} \right\rangle\!\right\rangle \xrightarrow{\texttt{task\_start[8]?}} \left\langle\!\left\langle \begin{pmatrix} 61 \\ 53 \end{pmatrix} \right\rangle\!\right\rangle$$
$$\xrightarrow{\texttt{task\_finish[8]?}} \left\langle\!\left\langle \begin{pmatrix} 77 \\ 69 \end{pmatrix} \right\rangle\!\right\rangle \xrightarrow{\texttt{task\_start[9]?}} \left\langle\!\left\langle \begin{pmatrix} 92 \\ 84 \end{pmatrix} \right\rangle\!\right\rangle \xrightarrow{\texttt{task\_finish[9]?}} \langle \mathit{Finish}, \begin{pmatrix} 101 \\ 93 \end{pmatrix} \rangle \rightarrow \cdots$$

The reason for the different result can be found in the difference in definitions between the worst-case reaction time of paths in SymTA/S and the maximum reaction time as we defined in section 3.3.2. In SymTA/S, the reaction time is calculated from the point in time at which the task instance is inserted into the queue. Since $\tau_1 = (f_1, 3, 4, p = 3)$ has a lower priority than the tasks $\tau_3 = (f_3, 3, 4, p = 2)$ and $\tau_4 = (f_4, 2, 3, p = 1)$, the task instance's execution is not started until after these are run. Subtracting their WCET from the result returned by SymTA/S, we arrive at an equivalent value compared to the value obtained through the verification using our model.

A similar case can be encountered when comparing the results for the worst-case reaction time analysis of ec_2 as shown in figure A.5. While the maximum reaction path determined by SymTA/S is 101 ms long, we obtain the result that $\mathrm{MRT}(ec_2) \leq 80$ and for the query attempting to verify $\mathrm{MRT}(ec_2) < 80$ we receive the following trace:

$$\cdots \rightarrow \langle \mathit{Start}, \begin{pmatrix} 21 \\ 1 \end{pmatrix} \rangle \xrightarrow{\texttt{task\_start[2]?}} \left\langle\!\left\langle \begin{pmatrix} 21 \\ 0 \end{pmatrix} \right\rangle\!\right\rangle \xrightarrow{\texttt{task\_finish[2]?}} \left\langle\!\left\langle \begin{pmatrix} 27 \\ 6 \end{pmatrix} \right\rangle\!\right\rangle \xrightarrow{\texttt{task\_start[6]?}} \left\langle\!\left\langle \begin{pmatrix} 34 \\ 13 \end{pmatrix} \right\rangle\!\right\rangle$$
$$\xrightarrow{\texttt{task\_finish[6]?}} \left\langle\!\left\langle \begin{pmatrix} 50 \\ 29 \end{pmatrix} \right\rangle\!\right\rangle \xrightarrow{\texttt{task\_start[7]?}} \left\langle\!\left\langle \begin{pmatrix} 63 \\ 42 \end{pmatrix} \right\rangle\!\right\rangle \xrightarrow{\texttt{task\_finish[7]?}} \left\langle\!\left\langle \begin{pmatrix} 82 \\ 61 \end{pmatrix} \right\rangle\!\right\rangle \xrightarrow{\texttt{task\_start[9]?}} \left\langle\!\left\langle \begin{pmatrix} 92 \\ 71 \end{pmatrix} \right\rangle\!\right\rangle$$
$$\xrightarrow{\texttt{task\_finish[9]?}} \langle \mathit{Finish}, \begin{pmatrix} 101 \\ 80 \end{pmatrix} \rangle \rightarrow \cdots$$

It is worth noting that the global clock has a value equal to the reaction time determined by SymTA/S, indicating that both the UPPAAL model and SymTA/S analyzed the same flow. The trace we obtained represents a flow of the event chain with mixed BCET and WCET of the task instances that broadcasted the corresponding events. Since the tasks $\tau_5 = (f_5, 8, 10, p = 2)$ and $\tau_1 0 = (delay, 13, 13, p = 3)$ are deployed to the same system as $\tau_2 = (f_2, 6, 6, p = 1)$, these are executed first due to the higher priority. If we deduct their combined BCET from the value of the global clock, we obtain our result for the MRT of the event chain. In this case, assuming the BCET of the tasks before τ_2 represents the worst case, as τ_5 starts and finishes earlier while the next task in the chain, τ_6, does not, increasing the time spent in between these two without executing anything relevant to this current event chain.

In addition to the reaction time of event chains, SymTA/S is also capable of determining the data age for a given path using the *MaxAgeSemantic*. Different to the definition in figure 3.15, SymTA/S measures data age paths from the start of the first task in the path to the end of the last task. To conform to our definition of the data age in section 3.3.4, we create a path of two functions. An example of the data age analysis for the requirement MDA(f_4, f_7) can be found in figure 6.7. The difference between the end of the first task and the start of the second is both calculated and also annotated, such that we can compare it to the result we obtain using our model.

Although the tool gives 2 as the maximum delay in the path as shown in figure 6.7, we get a result where the data age clock `da_c[2]` is at value 4. This is because our model also considers the BCET instead of only the WCET for the first task, which can lead to an increased data age, as the execution time of the task instance shifts slightly, starting at an earlier point in time. Due to this the finish broadcast is being sent earlier as well, resulting in a farther distance between the finish event of the first function and the start event of the second one. Due to the way that SymTA/S calculates the data age path, from the time that the first task has been added to the queue, including the BCET would not make a difference to their expected result in this case.

The tool unfortunately provides no way to check for the synchronization of tasks as defined in section 5.2.1. To summarize, using SymTA/S gives us identical or at least transferable results for tests that can be performed. When tests successfully run through, the tool can generate a multitude of visualizations including the Gantt charts for the Worst-Case of Event Chain Reaction Time or of the Data Age, for which we have seen the embedded examples. In cases where the tests indicate an invalid system, an error message is generated, showing which simulated ECU is overloaded and which tasks were problematic. When such a test run finishes, no additional information about the system state is given and no Gantt charts are generated for paths including tasks on overloaded systems.

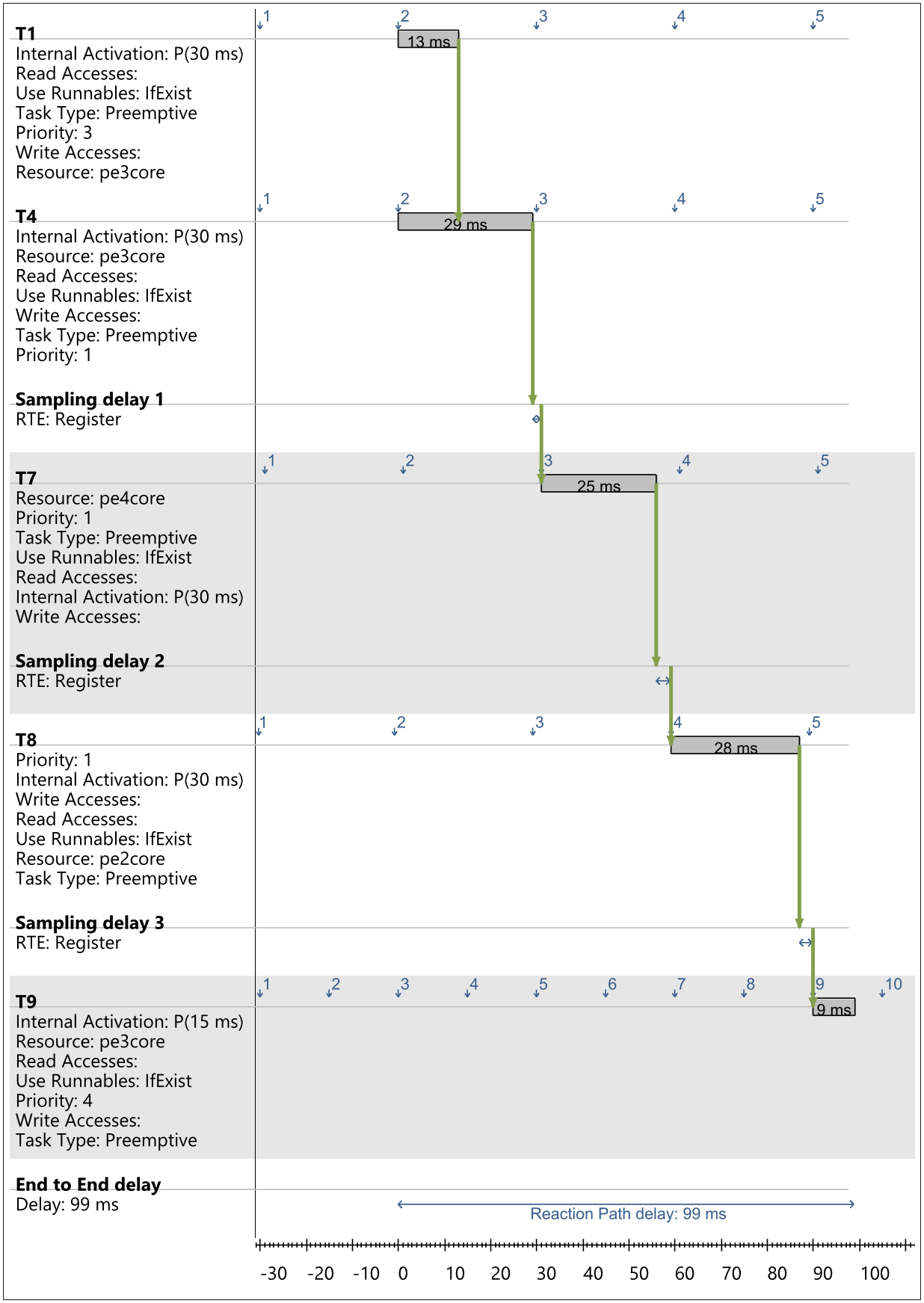

FIGURE 6.6: Results of SymTA/S WCET evaluation of ec_1

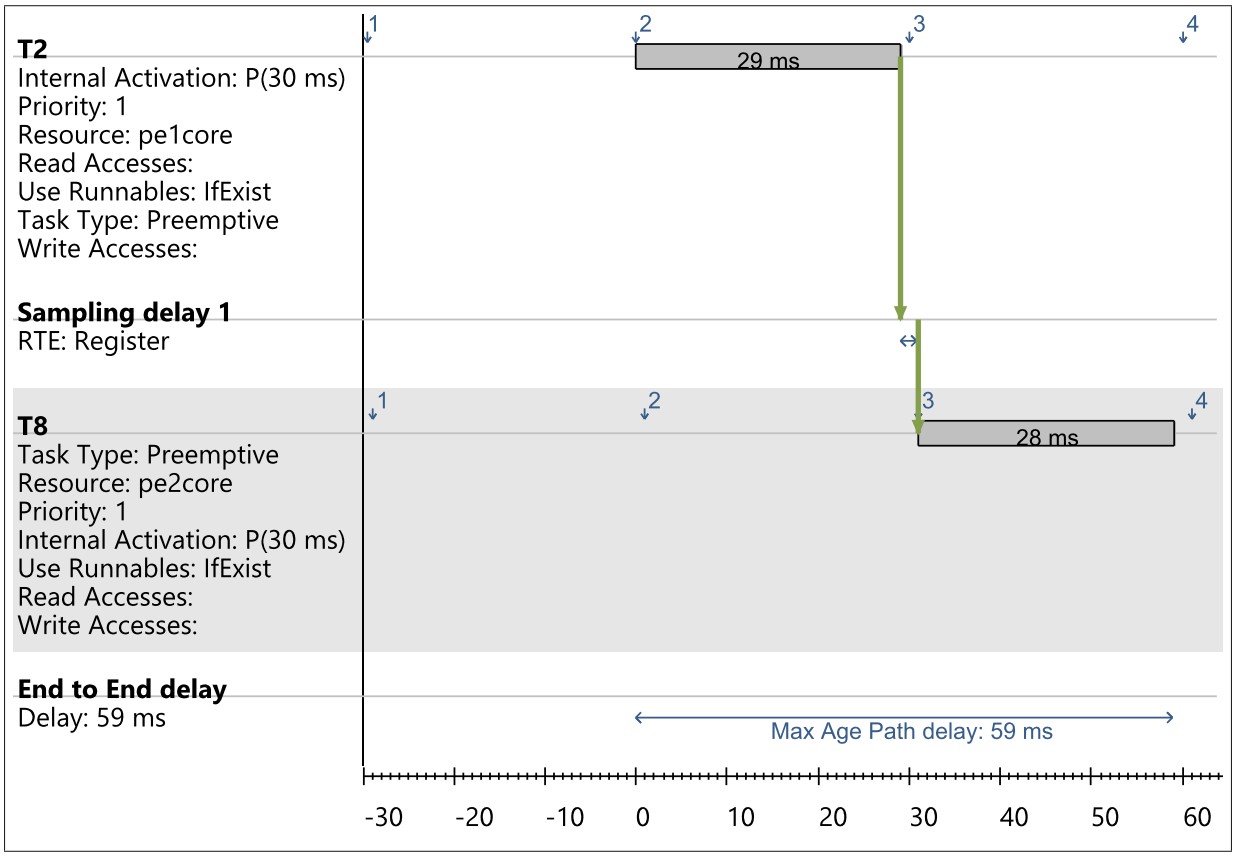

FIGURE 6.7: Gantt chart from SymTA/S visualizing data age

7 Conclusion

In this last part of this thesis, we will give a short summary of the previous chapters and our actual realization of the approach, discussing to which extent the goals specified in section 1.2 were met, which limitations the model currently exhibits and what prospects there are for future research.

7.1 Summary

During the course of this thesis, we have successfully developed an approach to verify the realizability of real-time requirements. Figure 7.1 gives an overview of the developed workflow.

We have successfully identified a multitude of inconsistencies in requirements that can be detected during the planning phase in the process and have provided examples of contradictory requirements that can already be detected using simple arithmetic. Examples of these inconsistencies include a requirement for the synchronization of tasks which run in different time grids and choosing a maximum reaction time for an event chain that is smaller than the sum of the worst-case execution time of included tasks. These can be considered to be design errors and do not require in-depth analysis to be discovered and corrected.

More intricate inconsistencies can be detected with just a set of requirements and a basic system design consisting of estimated task budgets, scheduling parameters and a distribution of tasks to systems. In order to do this, we have built a general model of timed automata and transferred this to UPPAAL to allow for automated model-checking. We have shown how using this model, some requirements can be verified using TCTL queries, and how some more sophisticated requirements can be checked with the help of additional timed automata. This automated approach allowed us to receive results that either indicate a system that can be successfully integrated, or it reveals infeasibilities. In the case of a verification failure, the model-checker allows us to return a full trace, from the simulation start up to the point where it encountered the system in a state where it violates the requirement. This counter-example and the path that led to it assists in the enhancement of the system design, allowing to create refined systems consistent with the requirements.

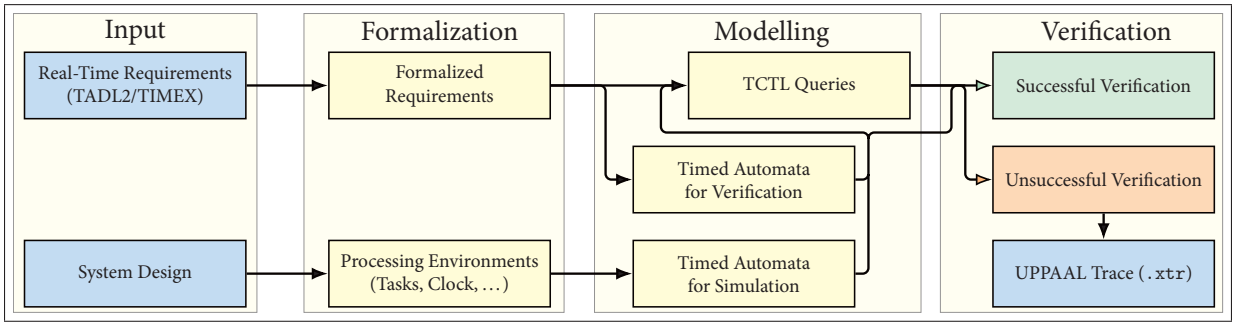

FIGURE 7.1: Workflow with the proposed approach

A case study was performed on the example of a brake-by-wire function, using the proposed approach to get from an initial system model to a sound system distribution, consistent with the imposed requirements. We compare some results we got from our approach to that obtained from the timing analysis tool SymTA/S, which performs statistical analysis, detailing both the validity of the results as well as the applicability in the planning stage.

With regard to the last research question, at least a part of the proposed workflow is automated of right now, the verification of the requirements. Using parametrizable templates in UPPAAL, we have allowed for the easy and automatable creation of the proposed automata, since dynamic properties like arrays can be inserted without requiring any alterations to the template. Considering figure 7.1, the whole workflow can be fully automated when a fixed input format for the system designs can be guaranteed. Since we have already used requirements in the form of TADL2 and all other steps in the workflow up to the verification can be described by predetermined rules, this allows the whole workflow to be automated. The diagnostic trace obtained from an unsuccessful query verification is given in `.xtr` format representing a path in the given automata. Given the system description, which was part of the input, it could also be transferred to another part or visualized to allow easier access to the results.

7.2 Discussion

Existing tools used to test automotive software systems are able to quite accurately simulate the system's behavior and often give very detailed results of tests in the form of exportable statistics and graphs. For systems which cannot be successfully simulated or for requirements that are not fulfilled, these tools often return only a very small set of information that could assist in the development of a refined system model.

In contrast to this, using the model we developed allows us to obtain in-depth information in form of a validation trace, when a requirement for a system is not met. While existing approaches are designed to work with working systems, we consider a working system to be the final state and only output information indicating the validity of the requirement in this case. For systems for which inconsistencies are detected given the requirements imposed upon them, we offer the ability to obtain information about the case in which such a conflict is encountered, assisting in the development of a consistent system conforming to the requirements.

Extending the proposed model is comparably easy, we have already shown how to implement both a static as well as a dynamic scheduling strategy and introduced two clocks and corresponding locations to assist in the verification of the covered requirements. By adding clocks to capture the time between occurrences of other events, as well as the implementation of supplementary task properties, the model can be vastly extended to incorporate a large variety timing-related behavior and allow for a multitude of additional requirements. With additional scheduling strategies, the simulation and verification can be expanded to a large variety of existing real-time systems.

It is worth noting that just because we find a system to be consistent, this does not guarantee that the final system realized based on the verified system design can be integrated successfully. There are a multitude of parameters affecting large-scale automotive software systems, only some of which can be predicted. Aside from abiding to timing requirements, a real-time system also requires to perform functionally correct, which requires a different tool suite to assess.

Especially during the planning phase, a non-negligible numbers of parameters affecting the final system is still unknown. Since our approach uses model-checking, the verification is strict and manages to always identify the worst-case situations. In the case of a verification success, we can at least say that under the given assumptions, the system should conform to the timing constraints, but cannot give a definite answer or guarantee. When an inconsistency is encountered, though, we can unambiguously say that there are issues in the system design that could lead to a violation of the constraints. These inconsistencies, when

ignored at the planning stage, will most likely be need to be fixed later in the process, such that we can claim our approach to be a valuable addition to the process as a whole.

7.3 Prospects and Future Research

While we have met the goals in section 1.2, we can still identify a multitude of items that allow for future research on the topic of this thesis. A trivial point is the addition of more requirements; while we have already covered most of the requirements of TADL2 that deal with periodic functions, there are still several left that could also be verified. Some of these might require alterations to the model proposed in chapter 4 to gain access to even more in-depth timing information, like the *ExecutionTimeConstraint* of TADL2 which specified the maximum core execution time of a function, ignoring the time the implementing task spends in suspended state.

A non-negligible property of actual distributed systems is clock drift[30], which describes a synchronization issue between clocks such that they will eventually get out of sync. The clock drift is usually so slight that a second on one ECU might take a couple of nanoseconds more on another one, but even slight drift adds up over time. We can emulate the clock drift over time by testing for a span of clock offsets, but actually simulating the clock offset using our model is currently not possible. Since the clock drift is so small, we would need to represent the offset using non-integer numbers, which is not possible in the current definition of timed automata which require integers in the comparisons of clock constraints. In the formal model we could simply multiply all guards by the same factor to achieve a model simulating clock drift, but since this still requires a small time step it would mean an enormous amount of possible states. Due to the constraints of UPPAAL, namely the upper bound for clocks and variables defined by `INT_MAX = 32767`, handling such a large state space is currently not possible in context of automated model checking of timed automata.

In recent years many manufacturers of ECU have switched to a multi-core approach[24, 33, 46], where each processor has multiple cores and allows for parallel execution. While we can to some extent model these cores using multiple processing environments, this requires that each core has it's own task queue, which does not represent the behavior of real multi-core systems. In addition, other properties of multi-core systems need to be considered; a processor with two cores usually doesn't have a 100% increase in processing power compared to a single core processor, which means that modelling both cores as separate processing environments results in a model far away from being a representation of the real system.

Many tasks in automotive software systems do not run periodically in a fixed time grid, but are triggered by events in a non-deterministic matter[42]. Using classic model-checking, these cannot be reliably accounted for, since the worst case assumptions made in the process would be that the event is constantly triggered, resulting in an extremely overloaded system, which is not even close to situations that occur in real-world examinations. There are also propositions to only realize safety-critical using periodically triggered tasks[23, 43], but as of right now, event-triggered tasks are considered to be an integral part of automotive software systems[42, 45]. Several approaches exist to apply statistical analysis to include these type of tasks, and [29] uses the experimental statistic model-checking toolkit integrated in the current UPPAAL development snapshots to verify such event properties inside UPPAAL. The model given in [29] is a very detailed representation of the system including a representation of the functional behavior in addition to the timing properties and unfortunately not transferable to a general model, but such a model

can be achieved using a similar development approach to the one we used.

To apply our approach to large-scale industrial processes and systems with several hundred functions and ECUs, the performance needs to be optimized heavily. In section 5.3.1 we have discussed the complexity of this approach, because model-checking over timed automata is used, the problem was found to be **PSPACE**-complete. Ways to mitigate this were covered in the same subsection, but when complex requirements like the reaction time of an event chain need to be checked on very large systems, either another modelling approach or an additional tool is required to perform a very optimized state reduction specially developed for this use case.

Bibliography

[1] R. Alur, C. Courcoubetis, and D. Dill. 'Model-checking for real-time systems'. In: *Proceedings. Fifth Annual IEEE Symposium on Logic in Computer Science*. Institute of Electrical and Electronics Engineers (IEEE), 1990. DOI: 10.1109/lics.1990.113766 (cit. on pp. 4, 11, 59, 63).

[2] Rajeev Alur and David Dill. 'Automata for modeling real-time systems'. In: *Automata, Languages and Programming*. Springer Nature, 1990, pp. 322–335. DOI: 10.1007/bfb0032042 (cit. on pp. 4, 7).

[3] Rajeev Alur and David L. Dill. 'A theory of timed automata'. In: *Theoretical Computer Science* 126.2 (Apr. 1994), pp. 183–235. DOI: 10.1016/0304-3975(94)90010-8 (cit. on pp. 4, 7).

[4] AUTOSAR. *Timing Analysis*. Tech. rep. AUTOSAR, 2015 (cit. on pp. 4, 6, 12, 14).

[5] Johan Bengtsson. *Reducing memory usage in symbolic state-space exploration for timed systems*. Tech. rep. 2001 (cit. on p. 63).

[6] Johan Bengtsson and Wang Yi. 'Timed Automata: Semantics, Algorithms and Tools'. In: *Lectures on Concurrency and Petri Nets*. Springer Science + Business Media, 2004, pp. 87–124. DOI: 10.1007/978-3-540-27755-2_3 (cit. on pp. 4, 7, 11).

[7] Johan Bengtsson et al. 'Partial order reductions for timed systems'. In: *CONCUR'98 Concurrency Theory: 9th International Conference Nice, France, September 8–11, 1998 Proceedings*. Ed. by Davide Sangiorgi and Robert de Simone. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 485–500. ISBN: 978-3-540-68455-8. DOI: 10.1007/BFb0055643 (cit. on p. 63).

[8] Béatrice Bérard. 'An Introduction to Timed Automata'. In: *Lecture Notes in Control and Information Sciences*. Springer Science + Business Media, 2013, pp. 169–187. DOI: 10.1007/978-1-4471-4276-8_9 (cit. on p. 7).

[9] Hans Blom et al. *Timing Model – Tools, algorithms, languages, methodology, USE cases. TIMMO-2-USE Deliverable D11*. Tech. rep. Version 1.2. Aug. 30, 2012 (cit. on pp. 4, 6, 12–14, 66).

[10] Jean-Louis Boulanger and Van Quang Dao. 'Requirements engineering in a model-based methodology for embedded automotive software'. In: *2008 IEEE International Conference on Research, Innovation and Vision for the Future in Computing and Communication Technologies* (June 7, 2017). IEEE, 2008. DOI: 10.1109/rivf.2008.4586365 (cit. on pp. 3, 5).

[11] Peter Braun et al. 'Guiding requirements engineering for software-intensive embedded systems in the automotive industry'. In: *Computer Science - Research and Development* 29.1 (Oct. 2010), pp. 21–43. DOI: 10.1007/s00450-010-0136-y (cit. on pp. 1, 5).

[12] Manfred Broy. *Automotive software engineering*. Ieee, 2003. ISBN: 0-7695-1877-X (cit. on p. 1).

[13] Manfred Broy et al. 'Engineering Automotive Software'. In: *Proceedings of the IEEE* 95.2 (Feb. 2007), pp. 356–373. DOI: 10.1109/jproc.2006.888386 (cit. on p. 1).

[14] R.W. Butler and G.B. Finelli. 'The infeasibility of quantifying the reliability of life-critical real-time software'. In: *IEEE Transactions on Software Engineering* 19.1 (1993), pp. 3–12. DOI: 10.1109/32.210303 (cit. on p. 1).

[15] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems*. Springer US, 2011. DOI: 10.1007/978-1-4614-0676-1 (cit. on pp. 2, 24, 25, 57).

[16] Stephen A. Cook. 'The Complexity of Theorem-proving Procedures'. In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC '71. Shaker Heights, Ohio, USA: ACM, 1971, pp. 151–158. DOI: 10.1145/800157.805047 (cit. on p. 63).

[17] T. Cucinotta et al. 'A Real-Time Service-Oriented Architecture for Industrial Automation'. In: *IEEE Transactions on Industrial Informatics* 5.3 (Aug. 2009), pp. 267–277. DOI: 10.1109/tii.2009.2027013 (cit. on p. 2).

[18] Alexandre David et al. 'A Tool Architecture for the Next Generation of Uppaal'. In: *Formal Methods at the Crossroads. From Panacea to Foundational Support*. Springer, 2002, pp. 352–366. ISBN: 3-540-20527-6 (cit. on p. 63).

[19] Elena Fersman et al. 'Task automata: Schedulability, decidability and undecidability'. In: *Information and Computation* 205.8 (Aug. 2007), pp. 1149–1172. DOI: 10.1016/j.ic.2007.01.009 (cit. on p. 5).

[20] Bastian Florentz. 'Software and system architecture evaluation and analysis in the automotive domain'. PhD thesis. Technische Universität Braunschweig, 2008 (cit. on p. 1).

[21] Wang Yi, Paul Pettersson, and Mats Daniels. 'Automatic Verification of Real-Time Communicating Systems By Constraint-Solving'. In: (1994). Ed. by Dieter Hogrefe and Stefan Leue, pp. 223–238 (cit. on pp. 5, 7).

[22] Patrick Frey. 'A timing model for real-time control-systems and its application on simulation and monitoring of AUTOSAR systems'. PhD thesis. Universität Ulm, 2011. DOI: 10.18725/OPARU-1743 (cit. on pp. 2, 5).

[23] Jelena Frtunikj. 'Safety framework and platform for functions of future automotive E/E systems'. In: *Automotive and Engine Technology* (July 2016). DOI: 10.1007/s41104-016-0007-z (cit. on pp. 1, 80).

[24] Thomas Fuhrman et al. 'On Designing Software Architectures for Next-Generation Multi-Core ECUs'. In: *SAE Int. J. Passeng. Cars – Electron. Electr. Syst.* 8 (Apr. 2015), pp. 115–123. DOI: 10.4271/2015-01-0177 (cit. on p. 80).

[25] D.D. Gajski and F. Vahid. 'Specification and design of embedded hardware-software systems'. In: *IEEE Design & Test of Computers* 12.1 (1995), pp. 53–67. DOI: 10.1109/54.350695 (cit. on p. 1).

[26] GLIWA GmbH. 'Timing Poster'. In: Poster. Feb. 20, 2013 (cit. on p. 19).

[27] Thomas Herpel. 'Performance Evaluation of Time-Critical Data Transmission in Automotive Communication Systems. Leistungsbewertung zeitkritischer Datenübertragung in automobilen Kommunikationssystemen'. PhD thesis. Friedrich-Alexander-Universität Erlangen-Nürnberg, 2009 (cit. on p. 5).

[28] John Hopcroft, Wolfgang Paul, and Leslie Valiant. 'On Time Versus Space'. In: *J. ACM* 24.2 (Apr. 1977), pp. 332–337. ISSN: 0004-5411. DOI: 10.1145/322003.322015 (cit. on p. 63).

[29] Jin Hyun Kim et al. 'Formal Analysis and Testing of Real-Time Automotive Systems Using UPPAAL Tools'. In: *Formal Methods for Industrial Critical Systems*. Springer International Publishing, 2015, pp. 47–61. DOI: 10.1007/978-3-319-19458-5_4 (cit. on pp. 5, 80).

[30] Hermann Kopetz. *Real-Time Systems*. Springer US, 2011. DOI: 10.1007/978-1-4419-8237-7 (cit. on pp. 33, 57, 64, 80).

[31] Stefan Kugele. 'Model-Based Development of Software-intensive Automotive Systems'. Dissertation. München: Technische Universität München, 2012 (cit. on p. 5).

[32] J.H. Lala and R.E. Harper. 'Architectural principles for safety-critical real-time applications'. In: *Proceedings of the IEEE* 82.1 (1994), pp. 25–40. DOI: 10.1109/5.259424 (cit. on p. 1).

[33] Patrick Leteinturier, Simon Brewerton, and Klaus Scheibert. 'MultiCore Benefits & Challenges for Automotive Applications'. In: *SAE Technical Paper*. SAE International, Apr. 2008. DOI: 10.4271/2008-01-0989 (cit. on p. 80).

[34] Chris Line, Chris Manzie, and Malcolm Good. 'Control of an Electromechanical Brake for Automotive Brake-By-Wire Systems with an Adapted Motion Control Architecture'. In: *SAE Technical Paper Series*. SAE International, May 2004. DOI: 10.4271/2004-01-2050 (cit. on p. 66).

[35] Kenneth Lauchlin McMillan. 'Symbolic Model Checking: An Approach to the State Explosion Problem'. UMI Order No. GAX92-24209. PhD thesis. Pittsburgh, PA, USA, 1992 (cit. on p. 63).

[36] Robin Milner. *A Calculus of Communicating Systems*. Ed. by Robin Milner. Springer Berlin Heidelberg, 1980. DOI: 10.1007/3-540-10235-3 (cit. on p. 9).

[37] Jürgen Mössinger et al. 'Autosar – a Worldwide Standard Is on the Road'. In: (June 7, 2017). 2009 (cit. on p. 1).

[38] C. Norstrom, A. Wall, and Wang Yi. 'Timed Automata as Task Models for Event-Driven Systems'. In: *Proceedings Sixth International Conference on Real-Time Computing Systems and Applications. RTCSA'99 (Cat. No.PR00306)*. Institute of Electrical and Electronics Engineers (IEEE), 1999. DOI: 10.1109/rtcsa.1999.811218 (cit. on p. 4).

[39] OSEK. *OSEK/VDX Operating System Specification*. Ed. by OSEK. 2005 (cit. on p. 24).

[40] Alexander Pretschner et al. 'Software Engineering for Automotive Systems: A Roadmap'. In: *Future of Software Engineering (FOSE '07)* (June 7, 2017). IEEE, May 2007. DOI: 10.1109/fose.2007.22 (cit. on p. 1).

[41] V. Ratan et al. 'Safety analysis tools for requirements specifications'. In: *Proceedings of 11th Annual Conference on Computer Assurance. COMPASS '96* (June 8, 2017). IEEE, 1996. DOI: 10.1109/cmpass.1996.507883 (cit. on p. 4).

[42] Achim Rettberg et al., eds. *Analysis, Architectures and Modelling of Embedded Systems*. Springer Berlin Heidelberg, 2009. DOI: 10.1007/978-3-642-04284-3 (cit. on pp. 25, 80).

[43] Florian Sagstetter. 'Schedule Synthesis for Time-Triggered Automotive Architectures'. Dissertation. München: Technische Universität München, 2016 (cit. on pp. 5, 80).

[44] Walter J. Savitch. 'Relationships between nondeterministic and deterministic tape complexities'. In: *Journal of Computer and System Sciences* 4.2 (1970), pp. 177–192. ISSN: 0022-0000. DOI: http://dx.doi.org/10.1016/S0022-0000(70)80006-X (cit. on p. 63).

[45] Oliver Scheickl. 'Timing Constraints in Distributed Development of Automotive Real-time Systems'. Dissertation. München: Technische Universität München, 2011 (cit. on pp. 2, 5, 14, 80).

[46] Rolf Schneider, Simon Brewerton, and Denis Eberhard. 'Multicore vs Safety'. In: *SAE Technical Paper*. SAE International, Apr. 2010. DOI: 10.4271/2010-01-0207 (cit. on p. 80).

[47] M. Shaw et al. 'Abstractions for software architecture and tools to support them'. In: *IEEE Transactions on Software Engineering* 21.4 (Apr. 1995), pp. 314–335. DOI: 10.1109/32.385970 (cit. on p. 4).

[48] K.G. Shin and P. Ramanathan. 'Real-time computing: a new discipline of computer science and engineering'. In: *Proceedings of the IEEE* 82.1 (1994), pp. 6–24. DOI: 10.1109/5.259423 (cit. on p. 1).

[49] Friedhelm Stappert. *Managing in-car timing constraints. TIMMO Innovation Report*. Tech. rep. 2009 (cit. on p. 2).

[50] Friedhelm Stappert et al. 'A Design Framework for End-To-End Timing Constrained Automotive Applications'. In: *Embedded Real-Time Software and Systems*. 2010 (cit. on p. 6).

[51] N. Stoimenov, S. Perathoner, and L. Thiele. 'Reliable mode changes in real-time systems with fixed priority or EDF scheduling'. In: *2009 Design, Automation & Test in Europe Conference & Exhibition* (June 12, 2017). IEEE, Apr. 2009. DOI: 10.1109/date.2009.5090640 (cit. on p. 25).

[52] Symtavision GmbH. *Analysis Introduction and Theory. Symbolic Timing Analysis for Systems User Documentation - Version 4.0.0*. 2017 (cit. on pp. 5, 73).

[53] Symtavision GmbH. *Main. Symbolic Timing Analysis for Systems User Documentation - Version 4.0.0*. 2017 (cit. on p. 73).

[54] Symtavision GmbH. *OSEK. Symbolic Timing Analysis for Systems User Documentation - Version 4.0.0*. 2017 (cit. on pp. 73, 74).

[55] Symtavision GmbH. *System Distribution. Symbolic Timing Analysis for Systems User Documentation - Version 4.0.0*. 2017 (cit. on p. 74).

[56] G. Tassey. *The Economic Impacts of Inadequate Infrastructure for Software Testing*. Diane Publishing Company, 2002. ISBN: 9780756726188 (cit. on p. 3).

[57] H. Thane and H. Hansson. 'Testing distributed real-time systems'. In: *Microprocessors and Microsystems* 24.9 (Feb. 2001), pp. 463–478. DOI: 10.1016/s0141-9331(00)00099-5 (cit. on pp. 2, 5).

[58] V-MODELL®AUTHORS. *V-Modell XT. Fundamentals of the V-Modell XT, Version 1.3*. 2006 (cit. on p. 3).

[59] Verein zur Weiterentwicklung des V-Modell XT e.V. (Weit e.V.) *Das V-Modell XT. Das deutsche Referenzmodell für Systementwicklungsprojekte, Version 2.1*. Apr. 20, 2017 (cit. on p. 3).

[60] Justyna Zander-Nowicka. 'Model-based testing of real-time embedded systems in the automotive domain'. PhD thesis. Technische Universität Berlin, 2009. DOI: 10.14279/depositonce-2126 (cit. on p. 5).

Appendix

List of Symbols

$\mathcal{C}$	set of clocks in a timed automaton
v	clock valuation
$\mathcal{B}(\mathcal{C})$	set of clock constraints
$\mathcal{B}'(\mathcal{C})$	downwards closed set of clock constraints
$\mathcal{A}$	timed automaton
c	clock of a timed automaton
N	set of locations in a timed automaton
l_0	initial location of a timed automaton
E	set of edges of a timed automaton
I	invariant function of a timed automaton
λ	timed trace
$a!$	output action (broadcast emission) in a network of timed automata
$a?$	input action (broadcast reception) in a network of timed automata
η	internal action in a network of timed automata
$\forall\Box$	*always globally* operator in TCTL
$\forall\Diamond$	*always finally* operator in TCTL
$\exists\Box$	*exists globally* operator in TCTL
$\exists\Diamond$	*exists finally* operator in TCTL
$\phi \rightarrow \psi$	implication operator in TCTL
$\mathbb{N}$	set of natural numbers (often used as indices)
$\mathbb{N}_0$	set of natural numbers including 0
$\mathbb{R}$	set of real numbers
$\mathbb{R}^+$	set of positive real numbers
$\mathbb{R}_0^+$	set of non-negative real numbers
e	event
o	event occurrence
ec	event chain

$\mathrm{ef}(ec, t)$	event flow through event chain ec with stimulus occurring at t
f	function
pe	processing environment
τ	task
T_{pe}	set of tasks on the processing environment pe
$\mathrm{TG}(\tau)$	time grid assigned to task τ
TQ_{pe}	task queue of processing environment pe
c_{pe}	clock of processing environment pe
$\mathbf{Sch}(TQ)$	generic scheduling function applied to task queue TQ
f_τ	function f implemented by task τ
$BCET_\tau$	best-case execution time of task τ
$WCET_\tau$	worst-case execution time of task τ
A_τ	scheduling parameter of task τ
i	task instance
s_i	start time of a task instance i
et	Execution time of a task instance
$\mathrm{co}(\mathrm{c}_1, \mathrm{c}_2)$	clock offset between the clocks $\mathrm{c}_1, \mathrm{c}_2$
$\mathbf{OSEK}(TQ)$	OSEK scheduling function
P	priority of OSEK tasks
$\mathbf{EDF}(TQ)$	EDF scheduling function
D	relative deadline of EDF tasks
d	absolute deadline of EDF task instances
$\mathrm{MET}(f)$	maximum execution time requirement for function f
$\mathrm{MRT}(ec)$	maximum reaction time requirement for event chain ec
$\mathrm{PER}(f)$	periodicity requirement for function f
$\mathrm{MDA}(f_1, f_2)$	maximum data age requirement for function pair f_1, f_2
$\mathrm{SYNC}(f_1, \ldots, f_n)$	synchronization requirement for functions $f_1, \ldots, f_n$

Referenced Material

```
const int INT_MAX = 32767; // from C's limits.h

// Bounds for manual adjustment; choose as low as possible to reduce state space

// time at which the simulation shall be stopped
const int TIME_MAX = 10000;
// 'tick rate' of the simulation; setting this to anything
// other than 1 might cause undesired behavior
const int TIME_STEP = 1;
// largest clock offset between the reference system and another
const int MAX_OFFSET = 10;

// amount of tasks to be simulated (should be exact, but
// must at least be equal to number of different tasks)
const int TASK_AMOUNT = 10;
// accounts for NULL_TASK blocking the first ID and task IDs starting with 1
const int TASK_ID_MAX = TASK_AMOUNT + 1;

// maximum WCET of a single task
const int MAX_WCET = 50;
// maximum relative deadline of an EDF task
// (the absolute deadline is bound by TIME_MAX + MAX_REAL_DEADLINE)
const int MAX_REL_DEADLINE = 100;
// upper bound for priorities of an OSEK task
// (using TASK_ID_MAX here allows setting a unique priority for each task)
const int MAX_T_PRIORITY = TASK_ID_MAX;
// maximum period for a single task
const int MAX_PERIOD = TIME_MAX - 1;

// hard limit for the amount of tasks in the queue,
// simulation/verification crashes when this is surpassed
const int TASK_QUEUE_MAX = 3 * TASK_AMOUNT;
// amount of items that are permitted in the task queue
// before the automaton transitions to the OVERLOAD location
const int TASK_QUEUE_OVERLOAD = 2 * TASK_AMOUNT;
```

Listing A.1: Constants in UPPAAL Global declarations

```
// define 'NULL' values for types, since 'NULL' doesn't exist in UPPAAL (yet)
const Task NULL_TASK = { 0, 0, 0 };
const EDF_Task NULL_EDF_T = { NULL_TASK, 0, 0 };
const OSEK_Task NULL_OSEK_T = { NULL_TASK, 0, 0 };
const EDF_Task_Instance NULL_EDF_TI = { NULL_EDF_T, 0, 0, 0 };
const OSEK_Task_Instance NULL_OSEK_TI = { NULL_OSEK_T, 0, 0 };

// automaton state helpers
const int IDLE = 0;
const int DONE = -1;
const int OVERLOAD = -2;
const int SCHED_ERR = -3;
```

Listing A.2: Additional definitions in UPPAAL Global declarations

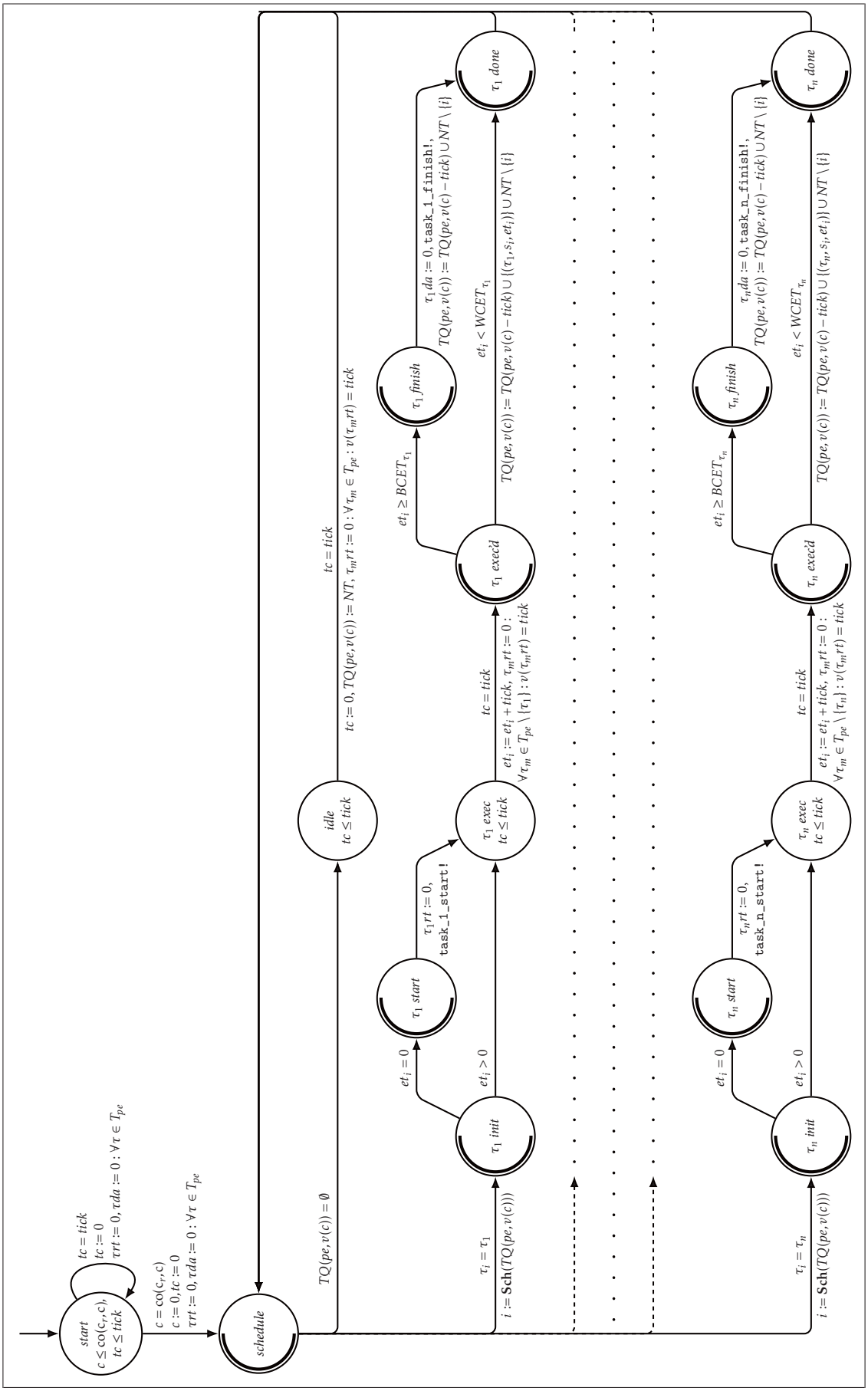

FIGURE A.1: Model of a single Processing Environment, optimized for verification

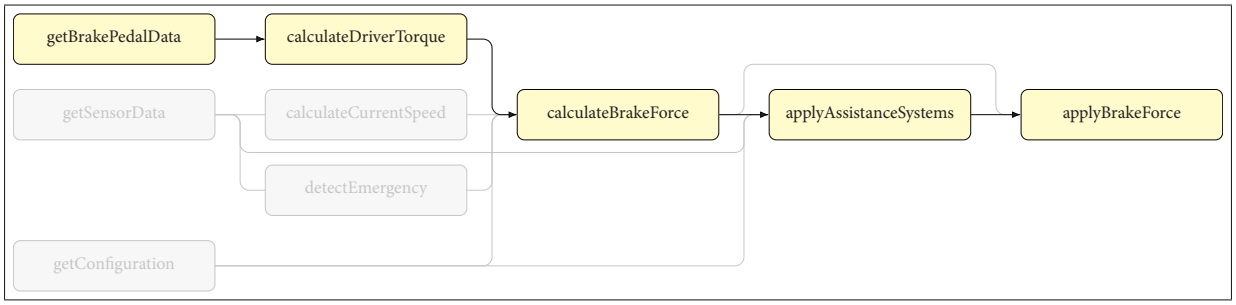

FIGURE A.2: Standard brake routine of the brake-by-wire architecture introduced in section 6.1

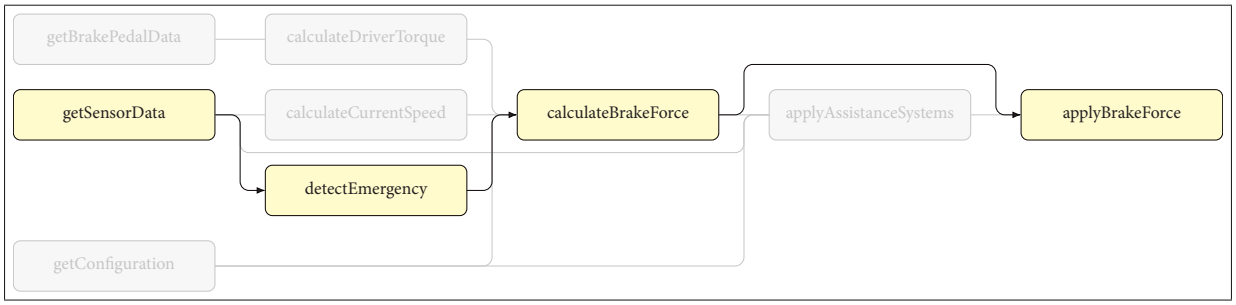

FIGURE A.3: Emergency brake routine of the brake-by-wire architecture introduced in section 6.1

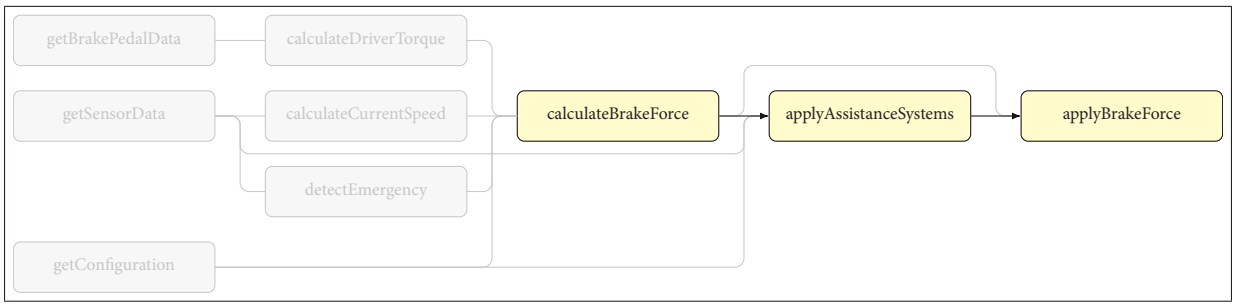

FIGURE A.4: Path of functions from the main brake controller to the brake actuators in the brake-by-wire architecture introduced in section 6.1

```
...

DelayConstraint brakeCalculationDelay {
    source calculateBrakeForceStart,
    target calculateBrakeForceFinish,
    upper = (28 ms on universal_time)
}

RepeatConstraint periodicBrakeInput {
    event calculateBrakeInputFinish,
    upper = (40 ms on universal_time)
}

AgeConstraint driverTorqueDataAge {
    scope = driverTorqueDataChain,
    minimum = 0,
    maximum = (16 ms on universal_time)
}

AgeConstraint assistiveSensorDataAge {
    scope = assistiveSensorDataChain,
    minimum = 0,
    maximum = (12 ms on universal_time)
}

ReactionConstraint standardBrakeConstraint {
    scope = standardBrakeFunction,
    upper = (110 ms on universal_time)
}

ReactionConstraint emergencyBrakeConstraint {
    scope = emergencyBrakeFunction,
    upper = (85 ms on universal_time)
}

ReactionConstraint mainBrakeConstraint {
    scope = mainBrakeChain,
    upper = (80 ms on universal_time)
}

SynchronizationConstraint syncInputCalculations {
    events calculateDriverTorqueFinish, calculateCurrentSpeedFinish, getConfigurationFinish
    tolerance = (10 ms on universal_time)
}
```

Listing A.3: Requirements for the example in chapter 6 in TADL2 format

```
// OSEK Task Definitions (Task, Priority, Period)
const OSEK_Task OT1 = { T1, 3, 30 };
const OSEK_Task OT2 = { T2, 1, 30 };
const OSEK_Task OT3 = { T3, 3, 30 };
const OSEK_Task OT4 = { T4, 1, 30 };
const OSEK_Task OT5 = { T5, 2, 30 };
const OSEK_Task OT6 = { T6, 2, 30 };
const OSEK_Task OT7 = { T7, 1, 30 };
```

```
const OSEK_Task OT8 = { T8, 1, 30 };
const OSEK_Task OT9 = { T9, 4, 30 };

// Array Compositions
const OSEK_Task PE1_Tasks[4] = { OT1, OT2, OT5, OT9 };
const OSEK_Task PE2_Tasks[3] = { OT3, OT4, OT6 };
const OSEK_Task PE3_Tasks[1] = { OT8 };
const OSEK_Task PE4_Tasks[1] = { OT7 };

// PE Definitions Template (Task Array, Offset to Reference PE)
PE1 = PE_4T_OSEK(PE1_Tasks, 0);
PE2 = PE_3T_OSEK(PE2_Tasks, 1);
PE3 = PE_1T_OSEK(PE3_Tasks, 2);
PE4 = PE_1T_OSEK(PE4_Tasks, 3);

//system PE4; // Queries 1, 2 (MET, PER)
//system PE1, PE3; // Query 3 (MDA(f2,f8))
//system PE2, PE4; // Query 4 (MDA(f4,f7))
//system PE1, PE2, PE3, PE4, ec1; // Query 5 (MRT(ec1))
//system PE1, PE2, PE4, ec2; // Query 6 (MRT(ec2))
//system PE1, PE3, PE4, ec3; // Query 7 (MRT(ec3))
//system PE1, PE2, PE3, sync1; // Query 8 (SYNC)
```

Listing A.4: UPPAAL declaration of the system shown in figure 6.4

```
const Task T10 = {10, 13, 13}; // delayTask

// EDF Task Definitions (Task, relative Deadline, Period)
const EDF_Task ET1 = { T1, 15, 30 };
const EDF_Task ET2 = { T2, 30, 30 };
const EDF_Task ET3 = { T3, 20, 30 };
const EDF_Task ET4 = { T4, 25, 30 };
const EDF_Task ET5 = { T5, 25, 30 };
const EDF_Task ET9 = { T9, 15, 15 };
const EDF_Task ET10 = { T10, 15, 30 };

// OSEK Task Definitions (Task, Priority, Period)
const OSEK_Task OT6 = { T6, 1, 30 };
const OSEK_Task OT7 = { T7, 1, 30 };
const OSEK_Task OT8 = { T8, 1, 30 };

// Array Compositions
const EDF_Task PE1_Tasks[3] = { ET2, ET5, ET10 };
const OSEK_Task PE2_Tasks[1] = { OT8 };
const EDF_Task PE3_Tasks[4] = { ET1, ET3, ET4, ET9 };
const OSEK_Task PE4_Tasks[1] = { OT7 };
const OSEK_Task PE5_Tasks[1] = { OT6 };

// PE Definitions Template (Task Array, Offset to Reference PE)
PE1 = PE_3T_EDF(PE1_Tasks, 0);
PE2 = PE_1T_OSEK(PE2_Tasks, 1);
PE3 = PE_4T_EDF(PE3_Tasks, 2);
PE4 = PE_1T_OSEK(PE4_Tasks, 3);
PE5 = PE_1T_OSEK(PE5_Tasks, 4);

//system PE4; // Queries 1 & 2
```

```
//system PE1, PE2; // Query 3
//system PE3, PE4; // Query 4
//system PE2, PE3, PE4, ec1; // Query 5
//system PE1, PE3, PE4, PE5, ec2; // Query 6
//system PE2, PE3, PE4, ec3; // Query 7
//system PE1, PE3, sync1; // Query 8
```

Listing A.5: UPPAAL declaration of the system shown in figure 6.5

```
// Task definitions (ID, BCET, WCET)
const Task T1 = {1, 3, 4};      // getBrakePedalData
const Task T2 = {2, 6, 6};      // getSensorData
const Task T3 = {3, 3, 4};      // getConfiguration
const Task T4 = {4, 2, 3};      // calculateDriverTorque
const Task T5 = {5, 8, 10};     // calculateCurrentSpeed
const Task T6 = {6, 16, 22};    // detectEmergency
const Task T7 = {7, 19, 25};    // calculateBrakeForce
const Task T8 = {8, 13, 28};    // applyAssistanceSystems
const Task T9 = {9, 7, 9};      // applyBrakeForce
const Task T10 = {10, 13, 13}; // delayTask

// OSEK Task Definitions (Task, Priority, Period)
const OSEK_Task OT1 = { T1, 3, 30 };
const OSEK_Task OT2 = { T2, 1, 30 };
const OSEK_Task OT3 = { T3, 2, 30 };
const OSEK_Task OT4 = { T4, 1, 30 };
const OSEK_Task OT5 = { T5, 2, 30 };
const OSEK_Task OT6 = { T6, 1, 30 };
const OSEK_Task OT7 = { T7, 1, 30 };
const OSEK_Task OT8 = { T8, 1, 30 };
const OSEK_Task OT9 = { T9, 4, 15 };
const OSEK_Task OT10 = { T10, 3, 30 };

// Array Compositions
const OSEK_Task PE1_Tasks[3] = { OT2, OT5, OT10 };
const OSEK_Task PE2_Tasks[1] = { OT8 };
const OSEK_Task PE3_Tasks[4] = { OT1, OT3, OT4, OT9 };
const OSEK_Task PE4_Tasks[1] = { OT7 };
const OSEK_Task PE5_Tasks[1] = { OT6 };

// PE Definitions Template (Task Array, Offset to Reference PE)
PE1 = PE_3T_OSEK(PE1_Tasks, 0);
PE2 = PE_1T_OSEK(PE2_Tasks, 1);
PE3 = PE_4T_OSEK(PE3_Tasks, 2);
PE4 = PE_1T_OSEK(PE4_Tasks, 3);
PE5 = PE_1T_OSEK(PE5_Tasks, 4);
```

Listing A.6: UPPAAL declaration of the system equivalent to the one modeled in SymTA/S

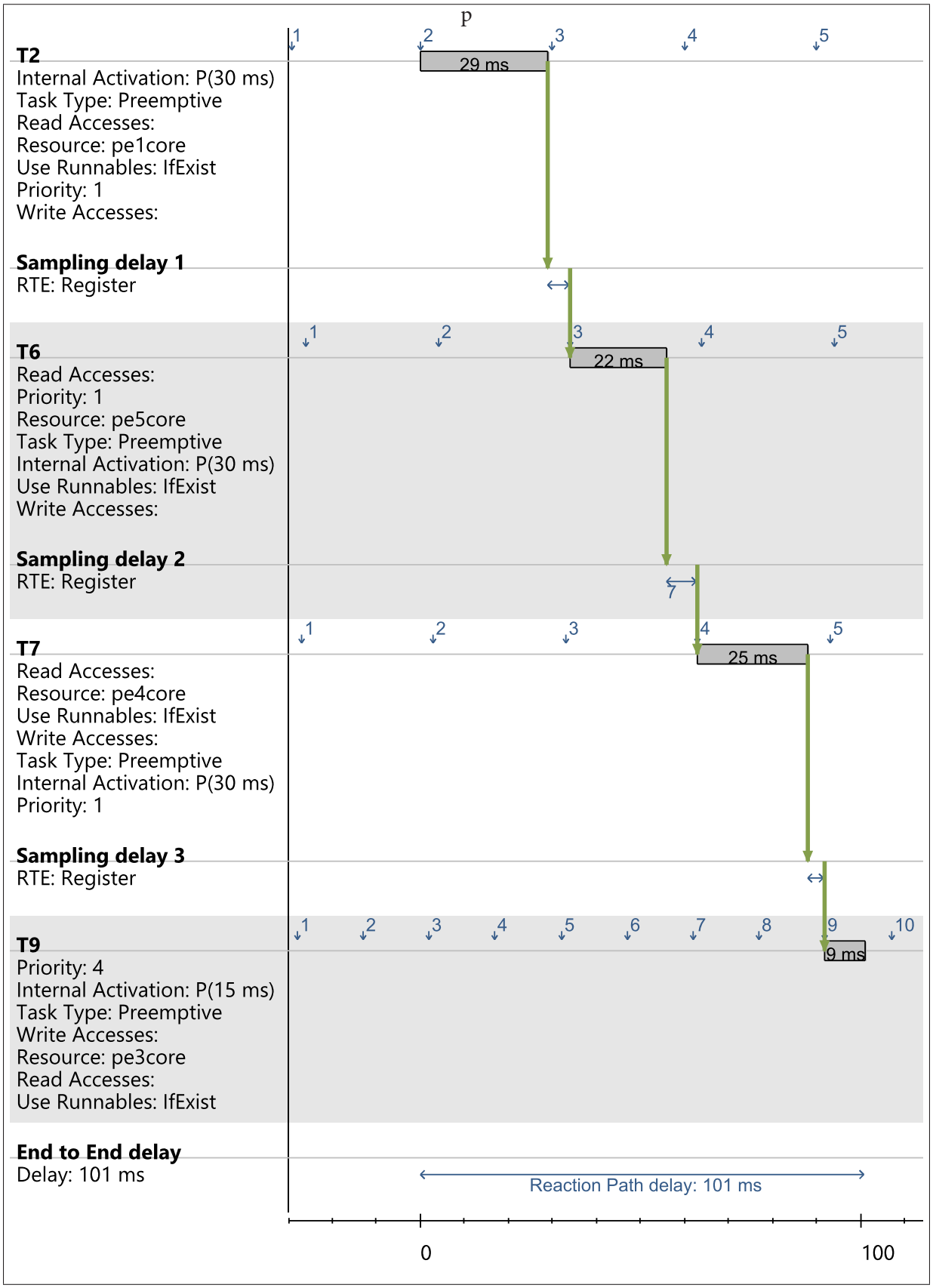

FIGURE A.5: SymTA/S WCET Gantt chart for ec_2

Code Listings and Templates

B.1 Full Code Listings

```
const int INT_MAX = 32767; // from C's limits.h

// Bounds for manual adjustment; choose as low as possible to reduce state space

// time at which the simulation shall be stopped
const int TIME_MAX = 10000;
// 'tick rate' of the simulation; setting this to anything
// other than 1 might cause undesired behavior
const int TIME_STEP = 1;
// largest clock offset between the reference system and another
const int MAX_OFFSET = 10;

// amount of tasks to be simulated (should be exact, but
// must at least be equal to number of different tasks)
const int TASK_AMOUNT = 10;
// accounts for NULL_TASK blocking the first ID and task IDs starting with 1
const int TASK_ID_MAX = TASK_AMOUNT + 1;

// maximum WCET of a single task
const int MAX_WCET = 50;
// maximum relative deadline of an EDF task
// (the absolute deadline is bound by TIME_MAX + MAX_REAL_DEADLINE)
const int MAX_REL_DEADLINE = 100;
// upper bound for priorities of an OSEK task
// (using TASK_ID_MAX here allows setting a unique priority for each task)
const int MAX_T_PRIORITY = TASK_ID_MAX;
// maximum period for a single task
const int MAX_PERIOD = TIME_MAX - 1;

// hard limit for the amount of tasks in the queue,
// simulation/verification crashes when this is surpassed
const int TASK_QUEUE_MAX = 3 * TASK_AMOUNT;
// amount of items that are permitted in the task queue
// before the automaton transitions to the OVERLOAD location
const int TASK_QUEUE_OVERLOAD = 2 * TASK_AMOUNT;

////////////////////
// Type Definitions
////////////////////
typedef struct
{
    int[0, TASK_ID_MAX] ID;
    int[0, MAX_WCET] BCET;
    int[0, MAX_WCET] WCET;
} Task;

typedef struct
{
    Task t;
    int[0, MAX_REL_DEADLINE] rel_deadline;
    int[0, MAX_PERIOD] period;
} EDF_Task;

typedef struct
{
    Task t;
    int[0, MAX_T_PRIORITY] t_priority;
    int[0, MAX_PERIOD] period;
} OSEK_Task;

typedef struct
{
    EDF_Task edf_t;
```

```
    int[0, TIME_MAX + MAX_REL_DEADLINE] deadline;
    int[0, TIME_MAX] start;
    int[0, MAX_WCET] et;
} EDF_Task_Instance;

typedef struct
{
    OSEK_Task osek_t;
    int[0, TIME_MAX] start;
    int[0, MAX_WCET] et;
} OSEK_Task_Instance;

typedef EDF_Task_Instance EDF_Task_Queue[TASK_QUEUE_MAX];
typedef OSEK_Task_Instance OSEK_Task_Queue[TASK_QUEUE_MAX];

// define 'NULL' values for types, since 'NULL' doesn't exist in UPPAAL (yet)
const Task NULL_TASK = { 0, 0, 0 };
const EDF_Task NULL_EDF_T = { NULL_TASK, 0, 0 };
const OSEK_Task NULL_OSEK_T = { NULL_TASK, 0, 0 };
const EDF_Task_Instance NULL_EDF_TI = { NULL_EDF_T, 0, 0, 0 };
const OSEK_Task_Instance NULL_OSEK_TI = { NULL_OSEK_T, 0, 0 };

// automaton state helpers
const int IDLE = 0;
const int DONE = -1;
const int OVERLOAD = -2;
const int SCHED_ERR = -3;

// verification helpers (clocks and channels for start/end events)
clock rt_t[TASK_ID_MAX];
clock da_t[TASK_ID_MAX];
bool is_running[TASK_ID_MAX];
broadcast chan task_start[TASK_ID_MAX];
broadcast chan task_finish[TASK_ID_MAX];

//////////////////
// EDF Scheduling
//////////////////
EDF_Task generate_EDF_Task(Task &t, int rel_deadline, int period) {
    EDF_Task new_task = { t, rel_deadline, period };
    return new_task;
}

// create an EDF task instance with a deadline based on the current time
EDF_Task_Instance generate_EDF_Task_Instance(EDF_Task &edf_t, int local_time) {
    EDF_Task_Instance new_ti = { edf_t, local_time + edf_t.rel_deadline, local_time, 0 };
    return new_ti;
}

// fill the EDF task queue given as a reference parameter with null values
void initialize_EDF_Task_Queue(EDF_Task_Queue &tq) {
    for (i : int[0, TASK_QUEUE_MAX - 1]) {
        tq[i] = NULL_EDF_TI;
    }
}

int[0, TASK_QUEUE_MAX + 1] count_EDF_queue_items(EDF_Task_Queue &tq) {
    int[0, TASK_QUEUE_MAX + 1] i = 0;
    while(i < TASK_QUEUE_MAX && tq[i] != NULL_EDF_TI)
        i++;
    return i;
}

// insert the given EDF task instance in the first free space in the given EDF task queue
void EDF_enqueue(EDF_Task_Queue &tq, EDF_Task_Instance &edf_ti) {
    int[0, TASK_QUEUE_MAX + 1] i;
```

```
    i = count_EDF_queue_items(tq);
    tq[i] = edf_ti;
}

// dereference the given EDF task instance inside the queue and shift up
// the elements after it, if necessary
void EDF_dequeue(EDF_Task_Queue &tq, int[0, TASK_QUEUE_MAX] ti_pos) {
    int[0, TASK_QUEUE_MAX] i = ti_pos;
    tq[ti_pos] = NULL_EDF_TI;
    while(i < (TASK_QUEUE_MAX - 1) && tq[i + 1] != NULL_EDF_TI) {
        if(tq[i] == NULL_EDF_TI) {
            tq[i] = tq[i + 1];
            tq[i + 1] = NULL_EDF_TI;
        }
        i++;
    }
}

// main scheduling function of EDF processing environments; selects the first instance
// in the given queue that has the lowest absolute deadline
int[0, TASK_QUEUE_MAX] EDF_schedule(EDF_Task_Queue &tq) {
    EDF_Task_Instance next_eti = tq[0];
    int[0, TASK_QUEUE_MAX] next_eti_pos = 0;
    int[1, TASK_QUEUE_MAX + 1] i = 1;
    // not run for empty queue due to tq[1] == NULL_EDF_TI
    while(i < TASK_QUEUE_MAX && tq[i] != NULL_EDF_TI) {
        if(tq[i].deadline < next_eti.deadline) {
            next_eti = tq[i];
            next_eti_pos = i;
        }
        i++;
    }
    return next_eti_pos;
}

//////////////////
// OSEK Scheduling
//////////////////
OSEK_Task generate_OSEK_Task(Task &t, int tpriority, int period) {
    OSEK_Task new_task = { t, tpriority, period };
    return new_task;
}

OSEK_Task_Instance generate_OSEK_Task_Instance(OSEK_Task &ot, int local_time) {
    OSEK_Task_Instance new_task_instance = { ot, local_time, 0 };
    return new_task_instance;
}

// fill the OSEK task queue given as a reference parameter with null values
void initialize_OSEK_Task_Queue(OSEK_Task_Queue &tq) {
    for (i : int[0, TASK_QUEUE_MAX - 1]) {
        tq[i] = NULL_OSEK_TI;
    }
}

int[0, TASK_QUEUE_MAX + 1] count_OSEK_queue_items(OSEK_Task_Queue &tq) {
    int[0, TASK_QUEUE_MAX + 1] i = 0;
    while(i < TASK_QUEUE_MAX && tq[i] != NULL_OSEK_TI)
        i++;
    return i;
}

// insert the given OSEK task instance in the first free space in the given OSEK task queue
void OSEK_enqueue(OSEK_Task_Queue &tq, OSEK_Task_Instance &osek_ti) {
    int[0, TASK_QUEUE_MAX + 1] i;
    i = count_OSEK_queue_items(tq);
```

```
        tq[i] = osek_ti;
    }

    // dereference the given OSEK task instance inside the queue and shift up
    // the elements after it, if necessary
    void OSEK_dequeue(OSEK_Task_Queue &tq, int[0, TASK_QUEUE_MAX] ti_pos) {
        int[0, TASK_QUEUE_MAX + 1] i = ti_pos;
        tq[ti_pos] = NULL_OSEK_TI;
        while(i < (TASK_QUEUE_MAX - 1) && tq[i + 1] != NULL_OSEK_TI) {
            if(tq[i] == NULL_OSEK_TI) {
                tq[i] = tq[i + 1];
                tq[i + 1] = NULL_OSEK_TI;
            }
            i++;
        }
    }

    // main scheduling function of OSEK processing environments; selects the first instance
    // in the given queue that belongs to the task with the highest priority of all instances
    // currently in the queue
    int[0, TASK_QUEUE_MAX] OSEK_schedule(OSEK_Task_Queue &tq) {
        OSEK_Task_Instance next_osek_ti = tq[0];
        int[0, TASK_QUEUE_MAX] next_osek_ti_pos = 0;
        int[1, TASK_QUEUE_MAX + 1] i = 1;
        // not run for empty queue due to tq[1] == NULL_OSEK_TI
        while(i < TASK_QUEUE_MAX && tq[i] != NULL_OSEK_TI) {
            if(tq[i].osek_t.t_priority > next_osek_ti.osek_t.t_priority) {
                next_osek_ti = tq[i];
                next_osek_ti_pos = i;
            }
            i++;
        }
        return next_osek_ti_pos;
    }
```

Listing B.1: Global Declarations for UPPAAL model

```
    // EDF scheduled Processing Environment with three Tasks
    // Parameters:
    // EDF_Task tasks[3]: Array of three EDF tasks
    // int clock_offset: Clock offset relative to reference PE

    // declaration of clocks
    clock c;
    clock tc;
    int[0, TIME_MAX] local_time;

    // declaration of basic scheduling parameters
    EDF_Task_Queue tq;
    const int [0, TASK_AMOUNT] TASK_COUNT = 3;
    int[0, MAX_PERIOD] TG_triggers[TASK_COUNT] = { tasks[0].period, tasks[1].period, tasks[2].period };

    // variables determined by scheduling
    int[-3, TASK_AMOUNT] next;
    int[0, TASK_QUEUE_MAX] next_ti_pos;
    int[0, TASK_QUEUE_MAX] queue_item_count;
    // shorthands for Template use
    int[0, MAX_WCET] next_ti_et;
    int[0, MAX_WCET] next_ti_WCET;
    int[0, MAX_WCET] next_ti_BCET;

    void fix_task_clocks() {
        // reset runtime clocks of tasks on this PE that are currently
        // not running or suspended
        for (i : int[0, TASK_COUNT - 1]) {
```

```
            if(is_running[tasks[i].t.ID] == false) {
                rt_c[tasks[i].t.ID] := 0;
            }
        }
    }

    void fix_all_clocks() {
        // reset runtime and data age clocks of tasks on this PE continuously
        // before starting
        for (i : int[0, TASK_COUNT - 1]) {
            if(is_running[tasks[i].t.ID] == false) {
                rt_c[tasks[i].t.ID] := 0;
                da_c[tasks[i].t.ID] := 0;
            }
        }
    }

    void schedule() {
        // generate and enqueue tasks for which the period is met
        for (i : int[0, TASK_COUNT - 1]) {
            if(local_time % TG_triggers[i] == 0) {
                EDF_Task_Instance ti;
                EDF_Task t = tasks[i];
                ti = generate_EDF_Task_Instance(t, local_time);
                EDF_enqueue(tq, ti);
            }
        }

        // determine next task
        next_ti_pos = EDF_schedule(tq);
        next = tq[next_ti_pos].edf_t.t.ID;
        next_ti_et = tq[next_ti_pos].et;
        next_ti_WCET = tq[next_ti_pos].edf_t.t.WCET;
        next_ti_BCET = tq[next_ti_pos].edf_t.t.BCET;

        // end simulation after the maximum simulation time is reached
        if((local_time + clock_offset) >= TIME_MAX)
            next = DONE;

        // switch into overload mode (essentially deadlock)
        // when the task queue is too full
        queue_item_count = count_EDF_queue_items(tq);
        if(queue_item_count > TASK_QUEUE_OVERLOAD)
            next = OVERLOAD;

        // detect a runtime scheduling error when a deadline is violated
        if((next != IDLE) && (next != DONE) && (tq[next_ti_pos].deadline < local_time))
            next = SCHED_ERR;
    }

    void initialize() {
        // initialize task queue and timekeeper
        initialize_EDF_Task_Queue(tq);
        local_time = 0;

        fix_all_clocks();
        schedule();
    }

    void start_task(int task_ID) {
        // mark the start of the task execution
        is_running[task_ID] = true;
    }

    void execute_task(int task_ID) {
        // simulate running the task for one time step
```

```
95      // task-specific behavior can be implemented by checking for taskID
        tq[next_ti_pos].et += TIME_STEP;
        // update shorthand
        next_ti_et = tq[next_ti_pos].et;
        // advance one step in time
100     local_time += TIME_STEP;
        fix_task_clocks();
    }

    void finish_task(int task_ID) {
105     // mark the end of task execution and deque the instance
        is_running[task_ID] = false;
        EDF_dequeue(tq, next_ti_pos);
    }

110 void idle() {
        // advance one step in time
        local_time += TIME_STEP;
        fix_task_clocks();
        schedule();
115 }
```

Listing B.2: UPPAAL Code for PE template to simulate three EDF-scheduled tasks

```
    // OSEK scheduled Processing Environment with three Tasks
    // Parameters:
    // OSEK_Task tasks[3]: Array of three OSEK tasks
    // int clock_offset: Clock offset relative to reference PE
 5
    // declaration of clocks
    clock c;
    clock tc;
    int[0, TIME_MAX] local_time;
10
    // declaration of basic scheduling parameters
    OSEK_Task_Queue tq;
    const int [0, TASK_AMOUNT] TASK_COUNT = 3;
    int[0, MAX_PERIOD] TG_triggers[TASK_COUNT] = { tasks[0].period, tasks[1].period, tasks[2].period };
15
    // variables determined by scheduling
    int[-2, TASK_AMOUNT] next;
    int[0, TASK_QUEUE_MAX] next_ti_pos;
    int[0, TASK_QUEUE_MAX] queue_item_count;
20  // shorthands for Template use
    int[0, MAX_WCET] next_ti_et;
    int[0, MAX_WCET] next_ti_WCET;
    int[0, MAX_WCET] next_ti_BCET;

25  void fix_task_clocks() {
        // reset runtime clocks of tasks on this PE that are currently
        // not running or suspended
        for (i : int[0, TASK_COUNT - 1]) {
            if(is_running[tasks[i].t.ID] == false) {
30              rt_c[tasks[i].t.ID] := 0;
            }
        }
    }

35  void fix_all_clocks() {
        // reset runtime and data age clocks of tasks on this PE continuously
        // before starting
        for (i : int[0, TASK_COUNT - 1]) {
            if(is_running[tasks[i].t.ID] == false) {
40              rt_c[tasks[i].t.ID] := 0;
                da_c[tasks[i].t.ID] := 0;
```

```
            }
        }
    }

    void schedule() {
        // generate and enqueue tasks for which the period is met
        for (i : int[0, TASK_COUNT - 1]) {
            if(local_time % TG_triggers[i] == 0) {
                OSEK_Task_Instance ti;
                OSEK_Task t = tasks[i];
                ti = generate_OSEK_Task_Instance(t, local_time);
                OSEK_enqueue(tq, ti);
            }
        }

        // determine next task
        next_ti_pos = OSEK_schedule(tq);
        next = tq[next_ti_pos].osek_t.t.ID;
        next_ti_et = tq[next_ti_pos].et;
        next_ti_WCET = tq[next_ti_pos].osek_t.t.WCET;
        next_ti_BCET = tq[next_ti_pos].osek_t.t.BCET;

        // end simulation after the maximum simulation time is reached
        if((local_time + clock_offset) >= TIME_MAX)
            next = DONE;

        // switch into overload mode (essentially deadlock)
        // when the task queue is too full
        queue_item_count = count_OSEK_queue_items(tq);
        if(queue_item_count > TASK_QUEUE_OVERLOAD)
            next = OVERLOAD;
    }

    void initialize() {
        // initialize task queue and timekeeper
        initialize_OSEK_Task_Queue(tq);
        local_time = 0;

        fix_all_clocks();
        schedule();
    }

    void start_task(int task_ID) {
        // mark the start of the task execution
        is_running[task_ID] = true;
    }

    void execute_task(int task_ID) {
        // simulate running the task for one time step
        // task-specific behavior can be implemented by checking for taskID
        tq[next_ti_pos].et += TIME_STEP;
        // update shorthand
        next_ti_et = tq[next_ti_pos].et;
        // advance one step in time
        local_time += TIME_STEP;
        fix_task_clocks();
    }

    void finish_task(int task_ID) {
        // mark the end of task execution and deque the instance
        is_running[task_ID] = false;
        OSEK_dequeue(tq, next_ti_pos);
    }

    void idle() {
        // advance one step in time
```

```
        local_time += TIME_STEP;
        fix_task_clocks();
110     schedule();
    }
```

Listing B.3: UPPAAL Code for PE template to simulate three OSEK-scheduled tasks

```
    // Verification automaton to check for the synchronization requirement imposed upon three tasks
    // Parameters:
    // int[1, TASK_AMOUNT] t_id[3]: Array of three task IDs
    //     for which the synchronization shall be checked
 5  // int[0, MAX_PERIOD] sync_max: maximum total offset between
    //     each finish of the tasks referenced by t_id

    const int [0, TASK_AMOUNT] TASK_COUNT = 3;

10  clock c;
    clock tc;
    bool triggered[TASK_COUNT];

    void trigger(int[0, TASK_COUNT] id) {
15      triggered[id] = true;
    }

    // return the total number of task finishes that were already triggered in this run
    int[0, TASK_COUNT] count_triggered() {
20      int[0, TASK_COUNT] count = 0;
        for (i : int[0, TASK_COUNT - 1]) {
            if(triggered[i])
                count++;
        }
25      return count;
    }

    // finish the run and reset all triggers
    void reset_triggers() {
30      for (i : int[0, TASK_COUNT - 1]) {
            triggered[i] = false;
        }
    }
```

Listing B.4: UPPAAL Code for a 3 task synchronization automaton template

B.2 Templates

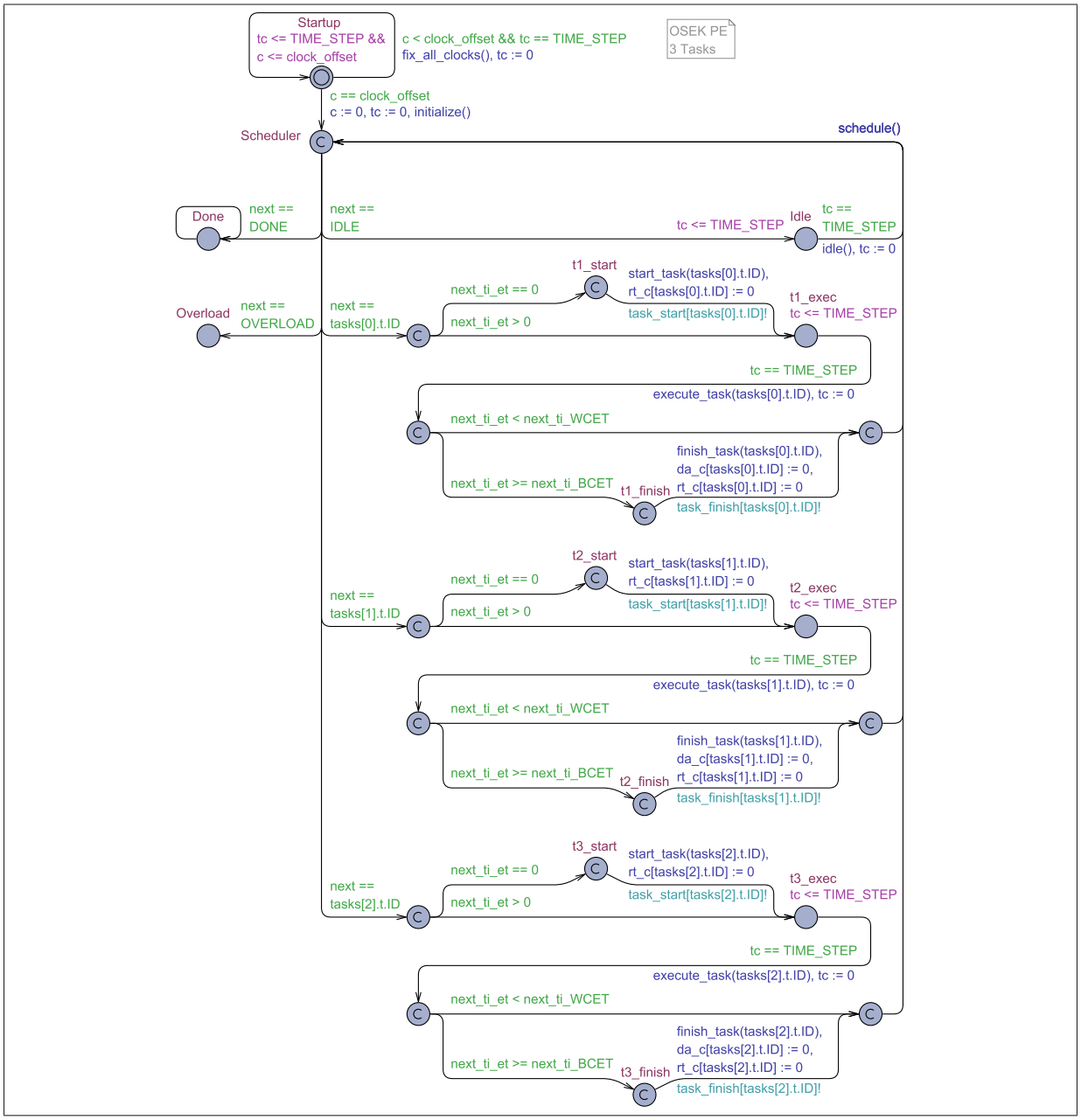

FIGURE B.1: Example of an OSEK-scheduled process environment template with three tasks

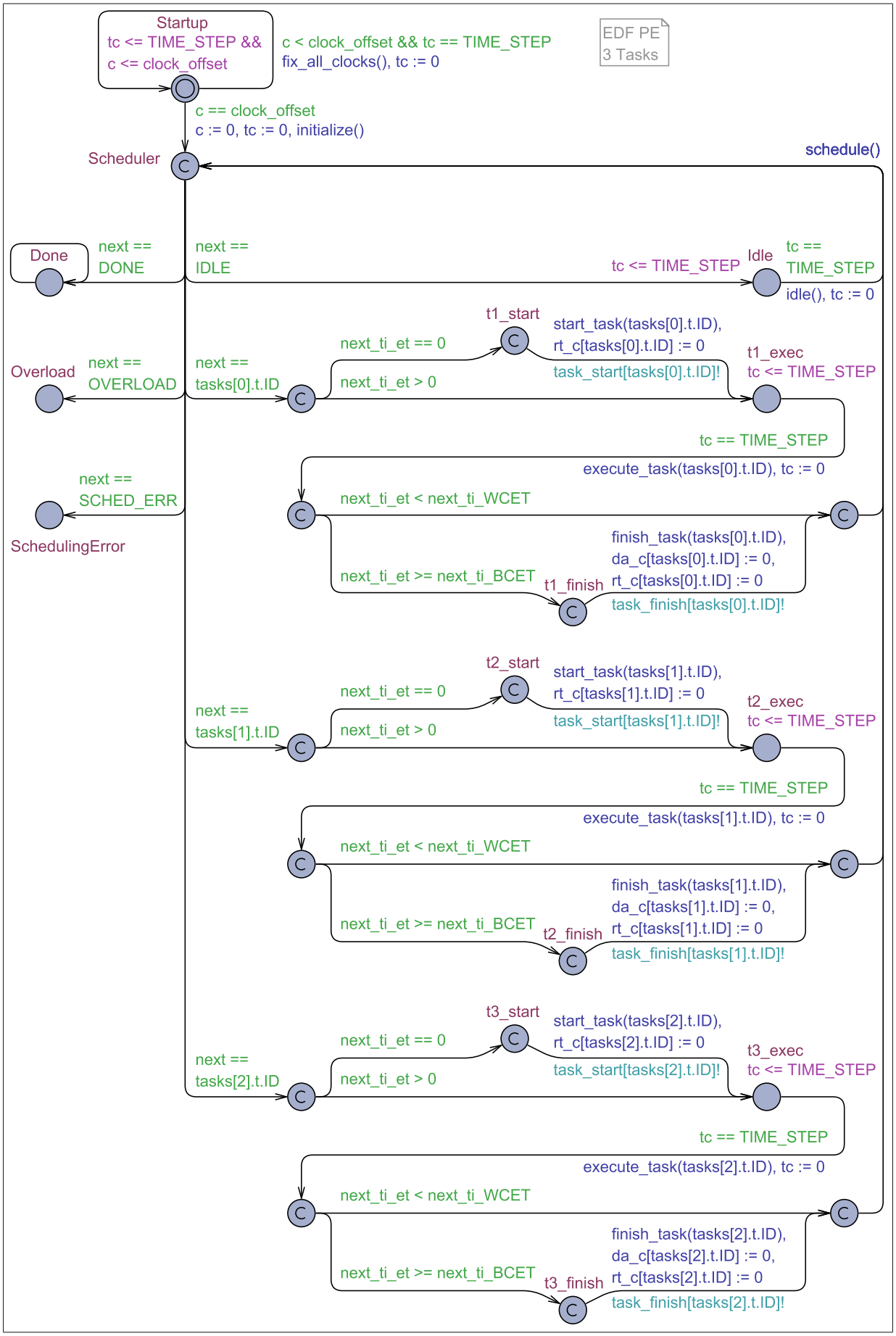

FIGURE B.2: Example of an EDF-scheduled process environment template with three tasks

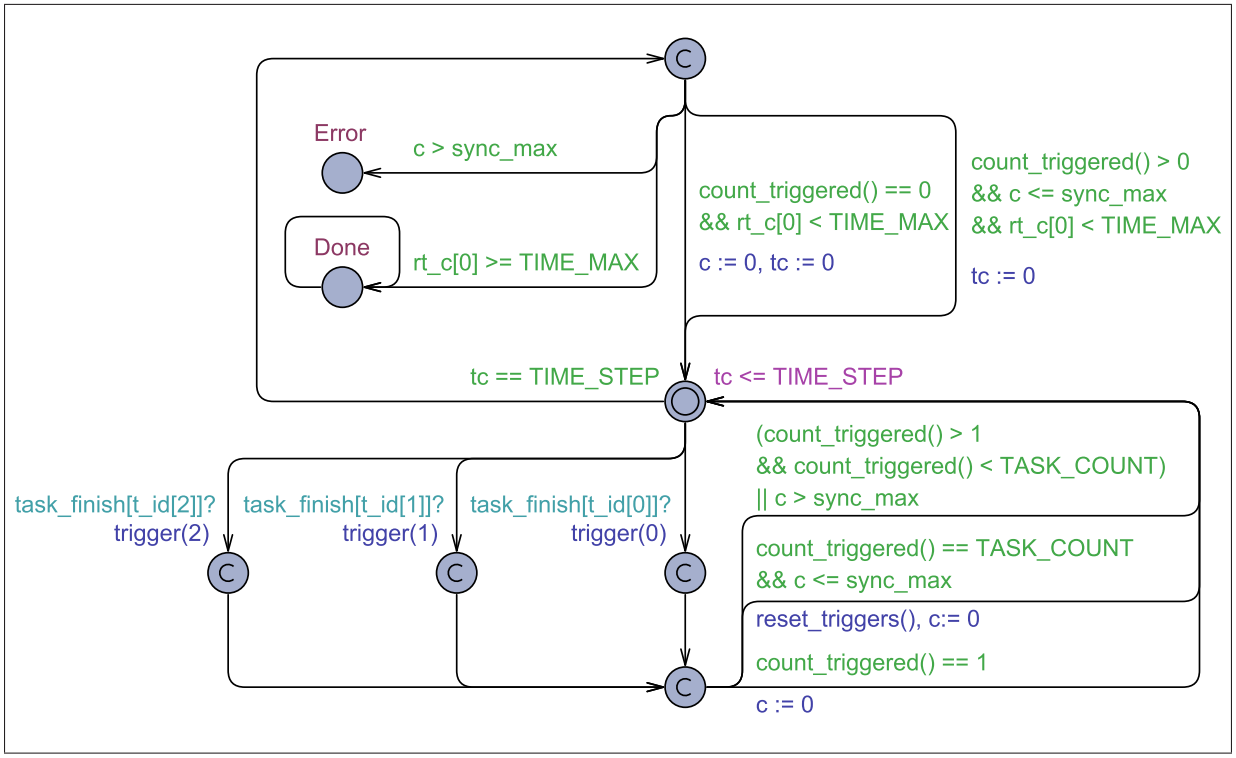

FIGURE B.3: Template of a synchronization automaton for three tasks

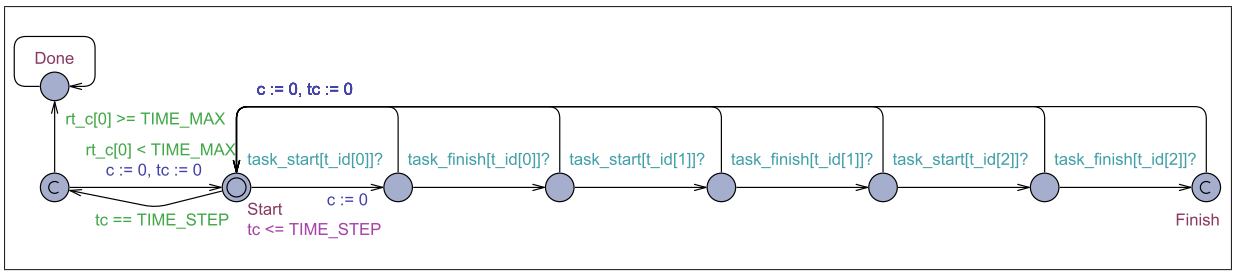

FIGURE B.4: Template of an event chain automaton for three tasks

List of Definitions

List of Figures

List of Listings