Integrated Timing Verification for Distributed Embedded Real-Time Systems

Dissertation

zur Erlangung des Grades des
Doktors der Ingenieurwissenschaften
der Fakultät für Mathematik und Informatik
der Universität des Saarlandes
von

Pascal Montag

Saarbrücken 2018

Tag des Kolloqiums:	08.05.2018
Dekan:	Prof. Dr. Sebastian Hack
Prüfungsausschuß:	Prof. Dr. Jan Reineke (Vorsitzender) Prof. Dr. Reinhard Wilhelm (Gutachter) Prof. Dr. Peter Marwedel (Gutachter) Dr. Roland Leißa (Akademischer Mitarbeiter)

Abstract

More and more parts of our lives are controlled by software systems that are usually not recognised as such. This is due to the fact that they are embedded in non-computer systems, like washing machines or cars. A modern car, for example, is controlled by up to 80 electronic control units (ECU). Most of these ECUs do not just have to fulfil functional correctness requirements but also have to execute a control action within a given time bound. An airbag, for example, does not work correctly if it is triggered a single second too late. These so-called real-time properties have to be verified for safety-critical systems as well as for non-safety-critical real-time systems. The growing distribution of functions over several ECUs increases the amount of complex dependencies in the entire automotive system. Therefore, an integrated approach for timing verification on all development levels (System, ECU, Software, etc.) and in all development phases is necessary.

Today's most often used timing analysis method - the timing measurement of a system under test - is insufficient in many respects. First of all, it is very unlikely to find the actual worst-case response times this way. Furthermore, only the consequences of time consumption can thus be detected but not the potentially very complex causes for the consumption itself. The complexity of timing behaviour is one reason for the often late and thus expensive detection of timing problems in the development process.

In contrast to measurement with the mentioned drawbacks, there is the static timing verification which exists since many years and is applicable with commercial tools. This thesis studies the current problems of industrial applicability of the static timing analysis (effort, imprecision, over-estimation, etc.) and solves them by process integration and the development of new analysis methods.

In order to show the real benefit of the proposed methods, the approach will be demonstrated using an industrial example at every development stage.

Kurzfassung

Unser tägliches Leben wird immer stärker von Software-Systemen durchdrungen, die oftmals nicht als solche wahrgenommen werden, da sie in Nicht-Computer-Systeme (Waschmaschinen, Autos, usw.) eingebettet sind. So arbeiten in einem aktuellen PKW bis zu 80 Steuergeräte.
Diese müssen in vielen Fällen nicht nur funktional korrekt arbeiten, sondern eine geforderte Berechnung auch innerhalb vorgegebener Zeitschranken ausführen. Ein Airbag erfüllt seine Aufgabe beispielsweise nicht, wenn er auch nur eine Sekunde zu spät ausgelöst wird. Die so genannten Echtzeiteigenschaften müssen für sicherheitskritische Anwendungen und soweit wie möglich auch für alle anderen Echtzeitsysteme, abgesichert werden.
Insbesondere sorgt die steigende Verteilung von Funktionen über mehrere Steuergeräte hinweg zunehmend für komplexe Abhängigkeiten im gesamten Fahrzeugsystem. Dies macht eine im Entwicklungsprozess und auf allen Abstraktionsebenen der Entwicklung (System, Steuergeräte, Software, usw.) durchgängige Methodik der Zeitverifikation notwendig.
Das heute übliche Verfahren der Zeitmessung von Systemen während der Testdurchführung ist in vielerlei Hinsicht ungenügend. Zum einen werden die tatsächlichen Grenzwerte nur mit sehr geringer Wahrscheinlichkeit erreicht. Zum anderen werden auf diese Weise nur die Auswirkungen von Zeitverbräuchen gemessen, nicht aber deren Ursachen analysiert, die möglicherweise sehr komplex sein können. Dies führt auch dazu, dass Probleme erst spät im Entwicklungsprozess erkannt und folglich nur mit hohen Kosten behoben werden können.
Neben den Zeitmessungen mit den genannten Nachteilen gibt es die statische Zeitverifikation. Diese ist bereits seit vielen Jahren bekannt und auch über entsprechende Werkzeuge einsetzbar. In der vorliegenden Dissertation werden die Probleme der industriellen Anwendbarkeit der statischen Zeitverifikation (Aufwand, Ungenauigkeit, Überschätzung, usw.) untersucht und mit einer durchgängigen Prozessintegration sowie der Entwicklung neuer Analyse-Methoden gelöst.
Der hier vorgestellte Ansatz wird deshalb in jedem Schritt mit einem Beispiel aus der Industrie dargestellt und geprüft.

Acknowledgements

To my understanding wife Franziska, who went through this thesis' development with me and stood by my side during all the usual ups and downs in the process. To my sons Frederik and Jannik who taught me to appreciate the important things in life and to keep matters in perspective.
I also would like to express my special gratitude to my friend and supervisor Dr. Steffen Görzig for giving me the opportunity to work on static software analyses in such a great team and for supporting me all the way through. There are and were so many colleagues at Daimler that helped me grow personally and professionally that I cannot mention all of them. Most important to me are André Schmidt, Udo Gleich, Gordon Haak, Falk Vieweg, Gabriel Schwefer, Nader Alexan and Maciej Kicinski.
With great respect, I want to thank Prof. Reinhard Wilhelm for the fruitful discussions we had, for his patience and his insistence at times when I was about to give up.
I also want to thank Prof. Peter Marwedel for serving as referee on my thesis committee.
Last but not least, I want to thank my parents and my family for their continuous love and support.

Contents

1 Introduction

1.1 Motivation

1.1.1 Real-Time Requirements

The current decrease in deadly traffic accidents [Fed17] reflects the increasing number of active electronic safety systems in today's cars since they were first introduced with the ABS (antilock braking system).
Such systems do not only have to decide correctly whether or not to interfere in the braking process but have to do so within strictly defined time bounds. If the intervention is triggered too late, it may cost lives.
In general, adherence to real-time properties (cf. Section 2.1.2) is crucial when it comes to safety-critical systems. But they are also highly relevant to other automotive systems, like engine control, as injections have to be precisely timed with the engine's current rotation speed.
But there is also a tremendous amount of real-time systems outside of the automotive domain (e.g. multimedia, aerospace or medical devices), that can also benefit from the results of an integrated timing verification approach as introduced in this thesis.

Figure 1.1: The difference between measurement and static analysis methods (adapted from [Abs17]).

1.1.2 Timing Verification Instead of Measurement

Today's most used timing analysis technique is measurement, which means that the response time of a system's tasks (cf. Section 5.3) are profiled while it is executed. However, due to the strong dependency on the execution state of the processor architecture, the same execution path may lead to different measurement results when executed twice. So you might miss the worst-case even if you have a full path coverage. In addition, due to most systems' complexity and the resulting high amount of possible execution paths, it is highly unlikely that the measurement of the response time actually hits the worst possible case (cf. Figure 1.1).
Therefore, a safety margin is added to the measured values. For example, the Daimler development standard allows a maximum of 80% CPU usage at the start of production. However, nobody can guarantee that the chosen safety margin is always sufficient. That is why almost all ECUs use a watchdog to dynamically supervise the correct task execution as well.
A watchdog however, can only reset or shut off a system and will thus reduce the availability of fail safe functions like the airbag. While the automotive industry seems to agree that this is acceptable for systems like the airbag as they are just systems that reduce the impact of an already occurred accident, it will not be sufficient for fail operational functions like autonomous driving.
Thus, the timing requirements of fail operational functions have to be formally verified in order to ensure safety. Additionally, timing verification is desirable for all safety functions as it will improve their availability.
But there are also economical benefits of static timing verification over measurement. As they deliver results early in the development the resulting changes can be made similarly early and thus, often with lower costs. Furthermore, static timing verification delivers detailed results and reasons for a particular timing behaviour, thus allowing for better decisions to optimise the timing behaviour.

1.1.3 Process Integration

One of the biggest challenges of static timing verification compared to measurement is that it requires detailed system and software architecture knowledge as well as some implementation knowledge for each and every software function in the system.

While a measurement only requires the execution of the system as part of a test, static timing verification requires the collection of precise function Worst-Case Execution Times (WCETs) as well as the correct calculation of task and system response times based on those WCETs. The simple problem in a distributed development environment is that there is usually not a single person that has such detailed and broad knowledge.
That is why this thesis claims that, in addition to technical improvements of the static timing verification methods, an integrated approach for timing verification is key to the industrial application of the technology. Such an integrated approach requires a common understanding of timing concepts and definitions. Beginning with the definition of timing properties of system and software architectures, covering development and analysis tool-chains and leading into a development process that defines the necessary tasks for different development roles (and typically people) in order to achieve a timing verification of the entire system.

1.2 Basics and Definitions

In electronic control units (and other electronic systems), time consumption results from hardware resource access operations. Such operations are triggered by an action (e.g. hardware trigger, code instruction, software function, operating system task) which itself is triggered by input events. In turn, actions result in output events as depicted in Figure 1.2.
Moreover, specific actions have specific execution properties. Tasks, for example, can be preemptive, have a periodicity and a worst-case response time (i.e. the worst-case time consumption between input event and output event).
The functional behaviour of an embedded real-time system can be depicted as an action or a sequence of actions that fulfil a high-level function. For the end user, the most relevant time span is the one between such a function's input event(s) (e.g. collision sensor impact) and the desired output event (e.g. airbag release). We call this time span the end-to-end response time (E2ERT).
But in order to understand this duration and its dynamics it is necessary to study the execution times of software functions as given by the worst-case execution time (WCET and the best and average case execution times (BCET and ACET) respectively (cf. Figure 1.1). Based on the

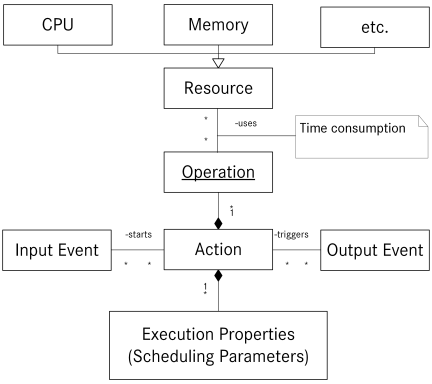


Figure 1.2: Real-time computational actions (adapted from [HGP+06])

execution times, different levels of worst-case response times (WCRTs) can be estimated based on the execution properties of tasks, interrupts and communication messages (cf. Figure 1.3).

1.3 Structure and Overview

In the next chapter (Chapter 2), we are going to discuss the current state-of-the-art in embedded real-time system (ERTS) development. There, we will discuss different approaches to software architecture design in order to identify the necessary concepts for improving timing predictability. Furthermore, we will introduce system and software development processes in general and the automotive V-model in particular. The specific characteristics of the automotive domain will be in focus throughout this thesis and will be emphasised in this chapter to highlight the potential differences to other domains.
Based on the current development of real-time systems, Chapter 3 will introduce the current challenges as we see them. These goals will later be used to evaluate existing approaches and to compare them to the results of this thesis.
As a thesis with industrial background, we will use several real world examples from the Daimler development throughout this work. Moreover,

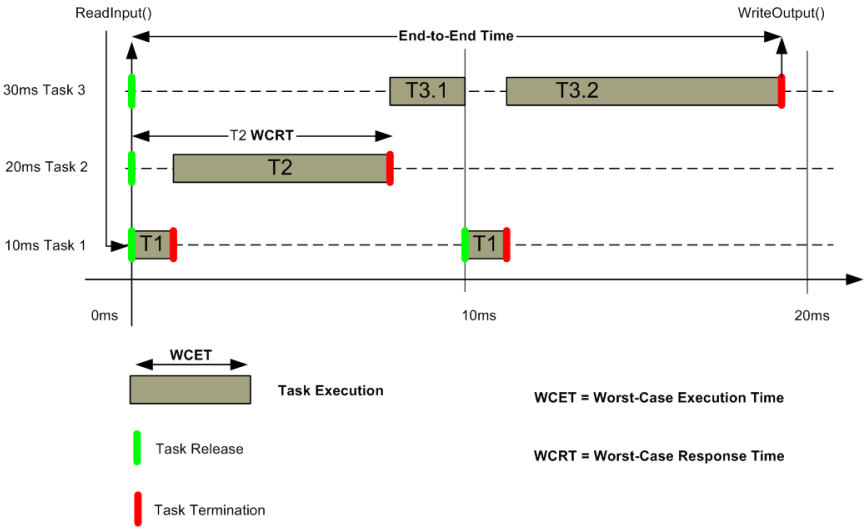


Figure 1.3: Schematic representation of timing analysis on task level.

in order to highlight the benefits of a continuous integration of the timing verification, we will use a central example, which will be introduced in Chapter 4.
In addition to the current state-of-the-art ERTS development, Chapter 5 introduces the state-of-the-art in timing analysis. Methods for the different levels of timing analysis (WCET, WCRT, E2E) will be introduced and discussed. This includes existing approaches for system-level timing definition and analysis, which we see as a key to an integrated approach.
Chapter 6 introduces the TOAD (Timing-Oriented Application Development) approach that represents the proposed solution of this thesis. In this chapter we argue that an integrated approach requires a defined software architecture, methods and tools as well as a process to be applied in an industrial context.
The TOAD approach overview is followed by our proposal for a TOAD architecture (TOAD-A in Chapter 7) and the TOAD tool suite (TOAD-T in Chapter 8). The architecture and tools form the infrastructure for an

integrated timing analysis that is used by the TOAD process (TOAD-P) which is described in Chapter 9.
Finally, we will sum up our results and compare them to existing approaches (cf. Chapter 10), followed by a discussion on the possible limitations of our approach (cf. Chapter 11).

2 Embedded Real-Time System Development

2.1 Introduction and Definition

Embedded Real-Time Systems (ERTS) represent the majority of automotive electronic systems. Within a car, which represents the embedding system, they perform specific tasks like powertrain, brake or airbag control.
ERTSs are embedded systems which have to meet real-time requirements in order to operate correctly. Hence, they represent the target domain for an integrated timing verification.
A common understanding of embedded systems and real-time systems as well as their architectures is established in the following sections. Furthermore, as an integrated timing analysis includes the integration into a development process, the current system development in the automotive domain is discussed with regard to its special characteristics.

2.1.1 Embedded Systems

An embedded system is a special-purpose hardware and software system, rather than a general-purpose computer (e.g. PC, smartphone). As its overall system design is focused on a small set of tasks such that these devices could be produced at a comparatively low cost in mass production. Embedded systems are usually not single devices but are rather built into an embedding system like a car, mobile phone or washing machine. Hence, people do not usually see the embedded systems, but rely on their correct behaviour. Only in error cases are people reminded of the complex electronic system which controls the device they use.
With the continuously increasing use of embedded systems that ease our lives and the simultaneous increase in the systems' complexity (e.g. by connecting single embedded systems to distributed systems), the amount of errors originating from embedded systems tends to increase. For example, the share of electronic-related car breakdowns from the overall breakdowns has increased from 35% in 2005 to 48% in 2015 (cf. Figure 2.1).

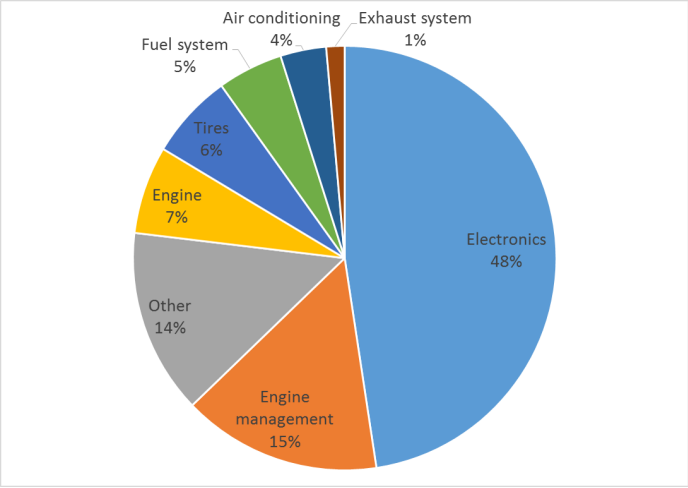


Figure 2.1: ADAC car breakdown statistic for Germany in 2015 [ADA16].

2.1.2 Real-Time Systems

Embedded systems use sensors to approximate the current state of the embedding system. Actions such as the acceleration towards a defined value are performed according to the current state (e.g. current speed) using actuators. As the state of the embedding system may change during the computation in the embedded system[1], the computation time has a direct influence on the correctness of the result.
H. Kopetz defines a real-time system as follows [Kop97]:

> A real-time (computer) system is a (computer) system in which the correctness of the system behaviour depends not only on the logical results of the computations, but also on the physical instant at which these results are produced.

The logical result of a computation originates from the functional requirements whereas the definition of the timing behaviour is a non-functional requirement that accompanies the functional requirements. A real-time

[1] I.e. the state may change before an appropriate action is computed.

system is not necessarily a system that reacts fast, but rather a system that complies with the defined non-functional timing constraints.
The often referred to airbag system, for example, also has lower time bounds on some requirements. For example, the crash detection conditions have to be present for a defined duration before a crash can be safely detected as such. Thus, an early airbag release would result in a potential release without the actual occurrence of a crash. This is regarded as an even worse scenario than that of the occurrence of a crash and the non-release of the airbag. This is because for the former scenario, the early airbag release could very well be the cause of occurrence of a crash.

Hard and Soft Real-Time Systems

Soft real-time systems are real-time systems where only the quality of the result is affected by deadline misses. Hence, deadline violations do not necessarily lead to system failures but can be tolerated within defined ranges. An example for a soft real-time system is a video player. A limited amount of skipped video frames due to deadline violations can be tolerated.
Hard real-time systems however, do not allow the violation of deadlines. Thus, most of them have to perform exception handling operations like entering fail-safe states as soon as a single deadline is missed. A fail-safe state is, for example, the deactivation of the airbag functionality and an appropriate indication of the deactivation in the driver's instrument cluster.
Most automotive ECUs contain some kind of hard real-time functionality. Most of them do so because they have to communicate in a defined frequency which is usually standardised and thus, easy to implement. But roughly half of them also have more complex safety-relevant functionalities that are time-dependent in almost all cases.
In opposite to fail-safe systems, there are fail-operational systems. Such systems have no fall-back fail-safe states for exception handling. Thus, systems like the Curiosity Mars-lander and upcoming autonomous cars make use of a redundant simplified functionality in case of exceptions like deadline misses. And those functionalities need guarantees on the defined time bounds.
Hence, the timing verification as it is presented in this thesis is crucial for fail-operational hard real-time systems. But it is also highly recommended

for all hard real-time systems and may even be helpful to ensure the quality and availability of soft real-time systems.

Event and Time-Triggered Systems

Every computer system is triggered by input events from external sensors and devices like buttons, radar sensors, clocks etc. The event of a clock tick is used for the so called time-triggered activations to start tasks or polled events periodically and thus, occur in a time deterministic manner. Event-triggered activations are therefore understood to be all non time-triggered activations. Such triggers are typically hardware interrupts that execute specific software-functions (i.e. interrupt service routines) in response to acyclic external events.
Most embedded systems today use event- and time-triggered functionalities together. The main part of the software functionality is often executed periodically in a time-triggered manner while concurrent functions (network communication, sensor information, etc.) result in interrupt events that deliver new input for the next main function cycle.
This mixture delivers the flexibility to react on non-deterministic inputs from the embedding system while the output events remain time deterministic.
The development of distributed time-triggered systems [Rus02] introduced globally-synchronised clocks (e.g. in the FlexRay network communication protocol [Fle07]) that allow time-determinism over several ECU-nodes with low latency as well as live-knowledge of the current E2E duration. This is very helpful for sensor-fusion functionalities, where all sensor inputs have to be synchronised in software. However, like watchdogs on ECU level, the knowledge of E2E violations can only result in an handling of such exceptions but cannot prevent them from happening. The latter is the intention of the proposed method in this thesis.

2.1.3 Architectures

Different Architectures are used on different development levels in order to structure the design and distribution of development work. Here, we consider the system-, hardware- and software-level. On each one of these levels, architectures are used to define basic rules and structures of interaction for lower level subsystems. Thus, they also define the necessary

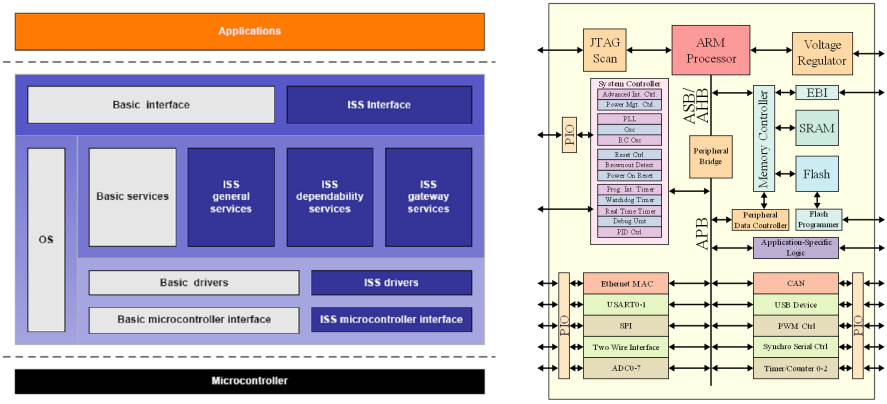


Figure 2.2: An example of a software architecture (left: architecture of Integrated Safety Systems (ISS), cf. [LHO+06]) and a hardware architecture (right: A system on-a-chip (SoC) for ARM).

collaboration of the teams developing these subsystems. And with regard to timing, it is the structure used to budget timing requirements and define scheduling rules.
The top level system in the automotive domain is the car itself, which consists of the physical system (e.g. mechanics and chemistry) and the embedded E/E system. The E/E system architecture (in the following only referred to as system architecture) defines the interplay of sensors, ECUs and actuators using communication connections.

System Architecture

The Carnegie Mellon University (CMU) defines a system architecture in [Car07] as

> a representation of a system in which there is a mapping of functionality onto hardware and software components, a mapping of the software architecture onto the hardware architecture, and human interaction with these components.

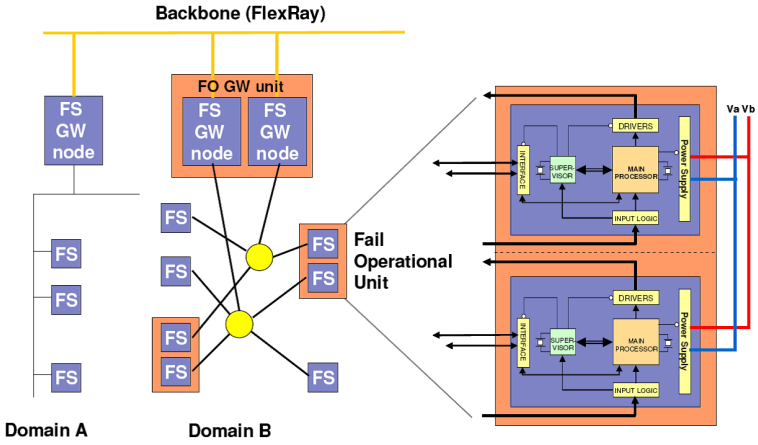


Figure 2.3: An example of an E/E system architecture (FS - Functional Software Component, GW - Gateway, cf. [LHO+06]).

For our use in the embedded real-time systems domain we adapt this definition to

> a representation of a system in which there is a mapping of functionality onto hardware and software components, a mapping of the software architecture onto the hardware architecture, and external interfaces that allow the interaction with these components.

In the phase of hardware/software co-design, functional requirements are assigned to hardware or software elements. Performance, flexibility and changeability are major aspects in this early decision process. The highest performance is usually gained by hardware implementations while the highest flexibility is given by software implementations.
Depending on the chosen software architecture, a late (re-)deployment of software components is possible. Such a deployment is depicted in the system architecture in Figure 2.3.
Timing requirements are derived from the physical system and assigned to

the system architecture first. Hence, timing verification needs to start on the system level as well.

Hardware Architecture

A hardware architecture defines the electrical interfaces and the layout of the different hardware components (see an example in Figure 2.2 on the right hand side). The following levels of hardware architectures are relevant in this thesis:

- Network topology: The structural topology of sensors, actuators and ECUs as well as the communication channels between them.
- ECU architecture: The structural organisation of memories, interfaces and CPUs on a physical controller.
- Processor architecture: The logic design or implementation of a specific instruction set architecture.

Successful timing verification is based on predictable processor architectures. As the automotive industry has abandoned the idea of developing automotive specific processor architectures that could focus on predictability (presumably due to the high costs), the commercially available processor architectures with increasingly elaborate caches tend to become less predictable with regard to timing. Therefore, those processor architectures may be used, but the lack of predictability limits their full potential.

Software Architecture

Similar to predictable processor architectures, software architectures can be more or less predictable with regard to timing.
Clements et al. define a software architecture in [BCK03] as:

> The set of structures needed to reason about the system, which comprises software elements, relations among them, and properties of both.

Hence, if we want to reason about the timing properties of a software architecture, we need to document structures, concepts and relationships that impact the timing behaviour. These are, for example, the scheduling strategy or component communication concepts.

2.2 Software Architecture Patterns

In this section, a short overview on common software architectural styles (cf. [SG96, CKK02, BCK03]) is given and evaluated with regard to their timing predictability and general usage in embedded real-time development.
According to [CKK02], an architectural style

- Describes a class of architectures or significant architecture pieces.
- Is found repeatedly in practice.
- Is a coherent package of design decisions.
- Has known properties that permit reuse.

Some of the most common software architectural styles are introduced in the following sections.

2.2.1 Event-Based Architectures

Event-based software architectures (also known as call and return architectures) are understood to be systems where functions are executed on demand instead of regularly in time. Representatives of this architectural style such as communicating processes or client server architectures are typically used to distribute systems on network nodes with a low computational overhead.
However, such systems tend to be unpredictable with regard to timing as there are no limits on the maximum frequency of possible requests (otherwise, one could choose a periodical execution accordingly). Hence, a timing verification is impossible until minimum distances between single event occurrences are defined (cf. Section 2.1.2).

2.2.2 Data-Flow Architectures

A data-flow oriented execution, as it is used for example in pipes and filter architectures or batch sequential architectures, assumes the execution of processing components (e.g. filters) to be triggered by the availability of new input data. Although this concept seems to be most efficient, it becomes challenging if there are concurrent executions on a single CPU. In such cases, scheduling and synchronisation concepts are required in order to achieve consistency among dependent data streams. Similar to event-based

architectures, unlimited arrival rates of new data makes predictability impossible.
For example, an Emergency Brake Assist (EBA, cf. Section 4.3) that adjusts the target speed (which is controlled by a Cruise Controller) according to obstacles in front of the vehicle and triggers the brakes in emergency situations requires a higher priority than the cruise controller, due to its safety relevance.
Thus, the data-flow orientation is only meaningful when it is used on a single consistent data stream. In this case, a timing verification can be performed if the data arrival rate is limited. That is why data-flow architectures can often be found as part of more complex architectures (e.g. a single component of a component architecture follows the data-flow pattern).

2.2.3 Blackboard Architectures

Blackboard architectures manage a central data-structure that contains the interface data of all processes and notifies processes of the arrival of new data according to predefined rules. Hence, the execution order depends not only on temporal distribution of input data, but also on potentially very complex rules.
The very high flexibility of this software architecture pattern results in a low and very difficult predictability.

2.2.4 Software Component Architectures

Software Component architectures (cf. Figure 2.4) focus on the definition of atomic units of execution, the components. These components can be triggered by data, events, requests or time. However, a time-deterministic software component architecture has to restrict the possible execution triggers, preferably to the periodic time trigger.
Software component architectures are usually built using a so-called runtime environment (RTE, also called middle-ware) which can be implemented as interpreter [2] or platform-dependent frameworks[3]. The use of interpreters usually leads to a lower performance than the use of platform-dependent frameworks. This is due to the fact that target specific hardware

[2] In this case, the code-instructions are executed without a target specific compilation.
[3] In this case, the components are compiled for the target hardware architecture.

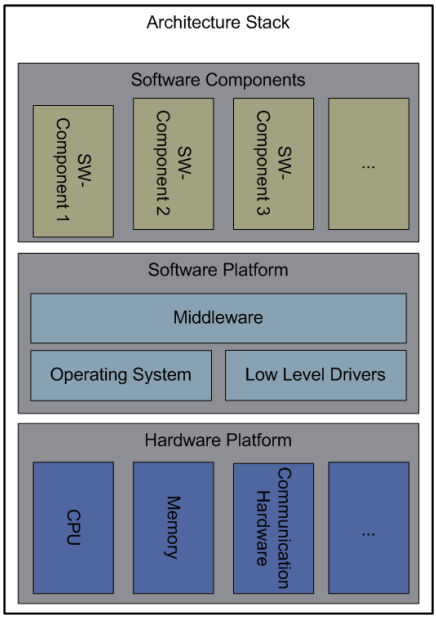


Figure 2.4: A common component architecture.

features cannot be used for the component execution by interpreters. On the other hand, interpreters are more flexible in case the support of a wide range of different hardware is a requirement. As embedded systems typically have higher performance and power-consumption requirements than flexibility requirements, most embedded systems' component frameworks are platform-dependent.
Nevertheless, software component architectures allow for a certain degree of hardware-platform independence as the RTE is used to abstract from low-level drivers and services. Such components are only aware of their formal interfaces. There is no hard link to where data inputs or function calls originated from. This is done by the software-architect who defines the RTE-configuration.
The high level of standardisation within a component framework allows for an easy deployment of components in a pre-existing system architec-

ture. But it also delivers a suitable infrastructure for system-level timing verification. The defined types and concepts allow the assignment of additional timing properties on an abstract level as well as the tracking of measurements and calculations on the implementation level.

2.3 Development Processes

> A development process defines the activities, methods and procedures that are necessary for the development and verification of a (software) system.
>
> Translation from [Bal98].

With regard to our timing verification approach, a domain independent process is required to define when and how different methods are to be applied to different work-products of a software development process (e.g. architecture, specification, code, etc.).
As explained earlier, we see one of the biggest challenges of timing verification in the fact that it cannot be assigned to a specific development step, but requires different actions on multiple development levels.
There are several standard process models for the software and system development (e.g. waterfall model, V-model, prototype model, etc.; cf. [Bal98]) whereas all of them describe the same three activities:

1. Requirements: The definition of the problem that the system/software(-component) should solve.
2. Specification/Implementation: The structure of the solution to a given problem, which may deliver sub-problems that are defined as requirements on the next refinement-level.
3. Test/Verification: Ensuring, that the solution actually solves the problem. Or, on the top level, the validation of the initial problem description.

With regard to timing verification, functional problem definitions on the top level are extended by timing requirements (i.e. latest, earliest or never definitions). All functional requirements that require an action to take place need a mandatory timing requirement. While "earliest"-requirements can be assigned to specific components that potentially delay an action,

"latest"-requirements need to be budgeted on all contributing sub-problems in the specification/implementation phases.
Actually, this is one of the biggest challenges of the timing verification. At this point, estimations are required that may turn out to be wrong. Or they may even lead to unbalanced efforts between different parts of the solution (e.g. component A struggles to achieve a timing requirement while component B has an additional budget available). The result is that architects avoid budgeting and hope for a good outcome in the late verification phases. This, however, has the negative side-effect that without budget-requirements, every component and every ECU tries to be just "fast". In doing so, CPU resources are wasted and cannot be retrieved easily in late phases.
Further questions arise in the context of specific domains. The following analysis of the automotive software development process aims at the identification of the specific challenges in the automotive domain.

2.4 Automotive Software Development

The development of electric and electronic systems (E/E systems) in the automotive industry generally adopts the V-model for system development [SZ03] which distinguishes between system and hardware/software development (cf. Figure 2.5). This model specifically understands the hardware- and software-development to be parallel and rather independent activities. It is assumed that the system-level is able to decide which problem is to be solved in hardware and which in software. This is actually the case only to a certain extent. Because there are interdependencies between the two, especially when it comes to timing requirements, such system-level requirements cannot be assigned to either of them independently. This is an additional reason, why budgeting from the system to the implementation level is often not considered to be meaningful.
In the following sections, the phases of the V-model are discussed from an automotive OEM (Original Equipment Manufacturer) point of view.

2.4.1 Analysis of the user/system requirements and specification of the logical architecture

The system requirements are specified as system functions with their respective timing requirements.

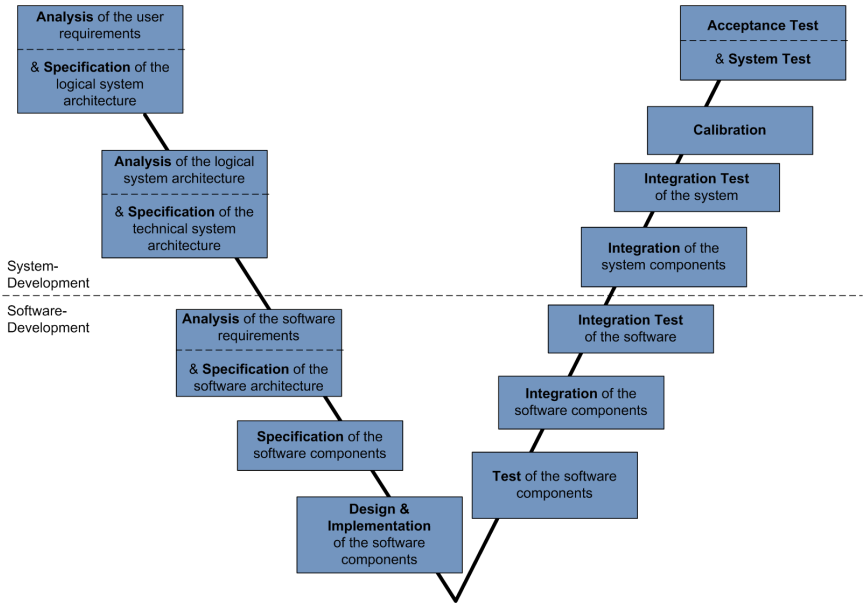


Figure 2.5: The standard process of an automotive development according to [SZ03].

2.4.2 Analysis of the logical architecture and specification of the technical architecture

The specification of the logical and technical architecture are typically not separated. Hence, the OEM refines and assigns the functions to the ECU level in one step. By doing so, sensors, actuators and communication interfaces between the ECUs are defined. If timing requirements were not explicitly defined, the communication interface indirectly defines them. The update rate of messages is assumed to correlate with the update rate of the actual calculated values to be sent.
In most cases, the system development on ECU, sensor and actuator level is performed by one or more suppliers. Especially in the case of multiple suppliers, budgeting of timing requirements is necessary in order to avoid legal disputes. However, this is not always done due to the same reasons

as mentioned before.
The supplier defines the ECU architecture as well as the underlying hardware and software architecture(s). In doing so, he defines whether functions are implemented in hardware, software or a mix of both. Often, the supplier is additionally responsible for sensors or actuators and thus, for an end-to-end timing of a specific function.
Today, the choice of a specific hardware platform is mainly driven by the knowledge gained in previous projects, costs and market availability. As innovative projects lack the knowledge of previous projects, they sometimes require costly hardware changes towards the end of the project.
This may be prevented with a more structured estimation of the required and available resources.

2.4.3 Analysis of the software requirements and specification of the software architecture

As the functionality of the software and the deployment on different hardware nodes has been defined at the previous stage, the sub-functions are now structured into software components with their respective interfaces on different software layers (e.g. hardware abstraction, basic-services and application[4]).
While the distribution of functionality into different software components can be found in most projects, they often lack the definition of system states with different functional and timing behaviour (e.g. initialisation, normal mode, error mode, etc.)[5]. Also, the definition of sequences for time and safety critical functions is not available in most projects. And in the few projects that specify system states or sequences we can almost never find refined timing requirements or budgeting.
The issue at hand is that functional partitioning in the course of software design is a well understood discipline. The budgeting of timing requirements is not. The time partitioning has to follow the functional partitioning. But unlike the functional partitioning it is based on uncertain assumptions, that the software architect has to make. These assumptions are based on existing code, assumed effects of hardware changes and best guesses for new functions.

[4] An application is understood as the set of system or project specific software functions.
[5] The statements here are based on the author's experiences in software architecture reviews for more than 30 projects.

Based on today's timing specification gap, it is obvious that it will be difficult to perform a module-based timing verification in later phases of the development.

2.4.4 Specification, Design and Implementation of Software Components

On the level of software components, interfaces and functionalities are specified according to the software architecture, further refined and implemented (often in a single model or a C-file). In the majority of the cases where the specification of the software architecture already missed the refinement of timing requirements, the software component design can not make up for that. So, we have actually never seen a single project that defined execution time limits on the software component level.
Such time limits are necessary, for example, if a task is shared by components from different development parties. In such cases a simple ratio of allowed time consumption can already be helpful in early stages. If a 10ms task is allowed to execute components for 2.5ms (as deducted from the software architecture's other tasks and functions) a split of 1ms and 1.5ms can be estimated by the software architect, based on measurements and static timing verification for existing code or assumptions based on the functionality.

2.4.5 Integration, Test and Verification

Depending on the project, the integration, test and verification steps can be found on different levels. These are typically the software, the ECU and one or more system levels (with the car as the top level system). Testing and (timing) verification however, are not necessarily performed on all levels. As timing requirements are usually not refined or budgeted in today's system development, timing verification can usually be performed only on a higher level of abstraction. The first one being the task level of an integrated software system.
Due to the most often missing low level timing requirements, intrinsic timing requirements are used instead. These are given by the fact that the response times of tasks shall usually not exceed the task's period. And due to today's mostly used dynamic timing verification, an integrated ECU is usually the first level for timing verification. There are several issues that

arise here. On one hand, such a level of abstraction makes it difficult to achieve worst-case timing scenarios (due to the already high level of complexity) and makes debugging of potentially found timing violations more difficult. On the other hand, meeting task deadlines does not necessarily result in meeting end-to-end deadlines. This can only be achieved by a combined functional and temporal reasoning.
In conclusion, we can see that because of various reasons, timing requirements are not refined to a level where they can easily be handled or verified. This thesis targets the elimination of the identified root causes for that issue.

2.5 Automotive Characteristics and Challenges

There are some specific characteristics of the automotive industry that make the timing verification issue more challenging. We have shown this in [MGL06b] for the static WCET verification with the tool *aiT* [Abs17].

2.5.1 Scalability

A main characteristic is the complexity of today's automotive systems. Although the lines of code do not form the most meaningful metric for software complexity, it is clear that an increase from some 10,000 lines of code in the last century to around 100 million lines of code per car in 2017 implies an immense increase in software and systems complexity. Especially software-centric functions like multimedia, navigation or driver assistance require new forms of software architectures to cope with the increasing complexity.
As the complexity tends to increase exponentially, so does the amount of system states and possible execution paths. Thus, timing verification has to rely on abstractions while remaining sound and precise, in order to scale with the complexity. For example, as most systems are not built from scratch, existing timing verification results (like loop-bounds or path constraint definitions) can be reused as well as the software components themselves. This, however, requires either abstract requirements on the possible execution contexts or a variant management detailed as detailed as the source code (e.g. parametrisable amount of gears for a gearbox software) which can be incorporated in the timing verification.

This requires in turn some level of standardisation on the software architecture and operating system level.

2.5.2 Distributed Development

Distributed development in the automotive domain does not only imply multiple development locations but also different development companies, mainly suppliers and sub-suppliers. This mainly cost-driven automotive characteristic is a de-facto standard for every bigger OEM.
The integrated timing verification is affected by the following aspects of the distributed development:

1. **Closed source development:** The OEM has usually no access to the supplier's source code. Hence, suppliers have to deliver partial timing verification results that can be integrated similar to their components and systems.
2. **Development scope:** There are systems that are completely developed by OEMs and there are systems that are completely developed by suppliers. And there are all shades of development scopes in between (cf. Figure 2.6). The partial timing verification results are required on every development level and need to be supplied from the OEM to the supplier in some special cases (e.g. OEM software component development and ECU integration on supplier side).

In general, code and verification results have to be reusable not only in different systems but also on different development levels with different levels of confidentiality.

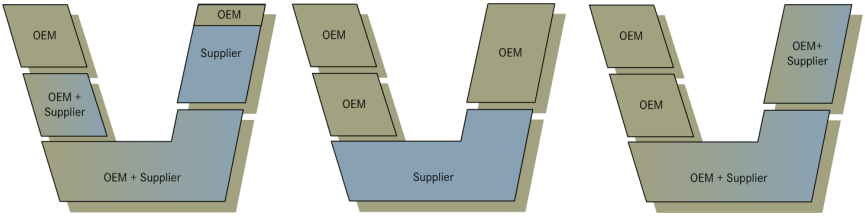


Figure 2.6: Several possible distributed development approaches.

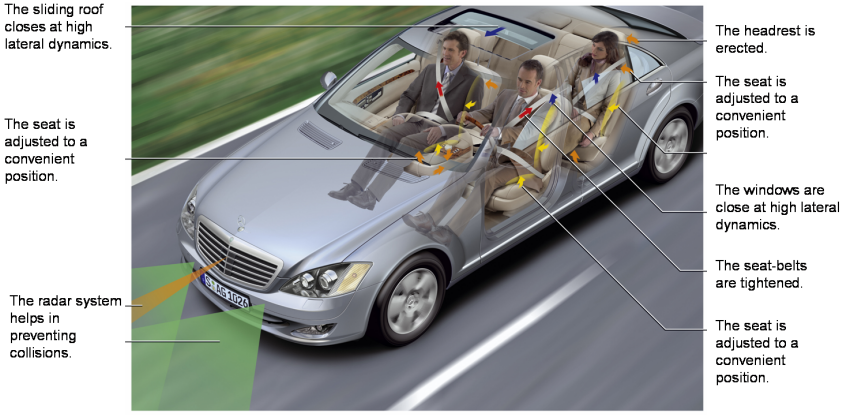


Figure 2.7: Distributed car functions that are activated at an imminent collision. Source: Daimler AG

2.5.3 Distributed Systems

As ECUs were developed similarly to mechanical components over a long period of time (i.e. independently by one supplier per ECU) the amount of ECUs has increased continuously to around 80. This high amount of separate controllers is not necessarily a functional requirement but more a reflection of historical mechanics centred development processes and responsibilities. Historically, there was no need for much communication between different development departments so that an ECU was typically developed by a single department. In turn, the ECU communication network can be seen as a reflection of the required communication between different OEM development departments.
However, the majority of today's innovative functions require increased communication in a network of different ECUs from different domains of the car (cf. Figure 2.7).

Therefore, it can be assumed, that the amount of distributed automotive systems will further increase in the future. The integrated timing verification approach has to support concepts for the timing analysis and verification of distributed systems. For safety-critical systems, the verifi-

cation of worst-case communication delay guarantee is required.
This requirement excludes non-time-deterministic distributed protocols, like TCP/IP, from the timing verification for safety-critical systems.

2.5.4 Hardware Variants and Configurations

Most vehicles are highly customisable products. The richness of the resulting hardware variants is directly reflected by the software. As the resulting software parameter configurations result in constraints on the possible execution paths, they also vary in their possible worst-case execution time.
On the one hand this leads to the necessity of static timing analysis as measurements cannot be carried out on all possible parameter combinations. Not to mention that correct guessing of worst-case variants is highly unlikely.
On the other hand, the timing verification of highly variant systems is currently not integrated into WCET analysis tools and requires formal variant descriptions which are not available for most projects.

2.6 Conclusion

This chapter introduced embedded real-time systems and their development processes in the way they are understood as a basis for this thesis. Various concepts for embedded systems are furthermore classified with regard to their timing behaviour and execution concepts. The technical foundation is further given by definitions and examples of software-, hardware- and system-architectures. As a conclusion, all development levels impact the integrated timing verification approach and will be considered accordingly in this thesis.
Automotive characteristics and challenges are introduced in order to highlight some requirements that seem to be unknown to the timing verification community so far. The solution of these specific challenges will increase the acceptance of the static timing verification in the automotive domain and could thus lead to a wider range of application of the methodology.

3 Challenges and Goals

3.1 Introduction and Definition

This chapter describes the identified challenges and defines resulting goals that are to be tackled in this thesis. These goals will be the basis for a later evaluation of the achievements of this thesis in comparison with existing approaches (cf. Section 5.4).

3.2 Main Goals

The main goal of this thesis is to define **a timing verification process that is industrially applicable**. As stated in previous chapters (cf. Section 2.3 and 2.4) we see the overall integration of timing aspects into the entire development process as a key to that.
Furthermore, as all electric/electronic (EE) development projects try to find a balance between product quality (e.g. functionality, safety etc.), development time (i.e. short time to market) and costs (i.e. development and production costs), these aspects are similarly relevant for our integrated timing verification approach. We believe that underestimating time and costs often prevents otherwise helpful approaches from being industrially applied.
In addition to the goal of the general industrial applicability, the second main goal is the **consideration of automotive specific requirements** (cf. Section 2.5 and [MGL06b]). The consideration of specific automotive industry related challenges aims at an industrial domain that is currently not as concerned with safety as the aeronautics and space domains. But the future of the automotive domain will lift it to a similar safety level as there won't be even a human driver on board anymore. The specific requirements are going to be considered specifically in Chapter 4.

3.3 Sub-Goals

3.3.1 Precision

As mentioned in Section 1.1.2 our goal is a timing verification that guarantees safe upper bounds for the timing estimations it produces. Furthermore, potential underestimations through wrong use of the analysis tools need to be indicated to the developer.
The method of choice for guaranteed upper time bounds is the static timing verification which faces the challenge of overestimation. So in order to be industrially applicable, our goal is to decrease the inherent overestimation. With regard to the specific automotive industry we see one major factor for an increased overestimation is the high amount of software and hardware variants. As the variants bear constraints on the possible execution paths, it is necessary to incorporate the potentially very complex constraint information which is often not explicitly given in the software.
Today's approaches to this problem are either to live with the overestimation or whenever that is not possible, to guess a worst-case variant and calculate its timing behaviour. The latter however, bears a high potential for underestimation. Hence, in order to achieve sound results, a variant-aware timing analysis is needed.

3.3.2 Generality and Flexibility

Using static analysis for timing verification is always limited by the supported hardware and software. And while our approach cannot support future hardware architectures out of the box, it can be generic and extendible to support these architectures.
Software architectures and operating systems are less diverse in the automotive industry. For real-time systems, the vast majority uses AUTOSAR [AUT07] nowadays which follows a component architecture approach (cf. Section 2.2). In the rare cases that AUTOSAR is not used, either a variant of OSEK is used or no dedicated operating system (OS) is used at all. Hence, this thesis will focus on AUTOSAR/OSEK systems only.

3.3.3 Usability and Automation

The academic background of automotive software developers is most often rather in mechanical engineering than in computer science. Thus, current

methods of formal specification or verification are very difficult to be applied without sufficient abstraction and automation on the user interface level. The tooling also needs to be integrated into existing development tools in order to increase the acceptance rate (e.g. into software architecture development tools).
Finally, also the analysis computation duration itself needs to be in acceptable ranges (rather hours than days) so that the development process is not unnecessarily affected.

3.4 Challenges

3.4.1 State Explosion

While the undecidability of the halting problem is a theoretical problem for the timing verification, the complexity of today's automotive embedded software is a practical one (cf. Section 2.5.1).
The analysis of a distributed real-time system in all its details (from source code to network communication) is not feasible today due to the state space explosion problem (cf. [MNL06]). The high amount of continuously changing input parameters that directly or indirectly influence the execution paths and thus, the timing behaviour, makes it nearly impossible to estimate safe time bounds in a reasonable time.
Hence, abstractions are necessary on many levels in order to reduce the number of relevant states considered in the timing verification. Every abstraction however, potentially reduces the precision of the analysis.

3.5 Limitations

3.5.1 Hardware Correctness

The real-time behaviour of a software can only be computed for a given hardware which executes it. As hardware operates in the physical environment, there is always a certain probability for hardware faults and defects. This is the case no matter how safe and redundant a hardware architecture is defined.
The guaranteed bounds resulting from timing verification always assume non-faulty hardware. For example, the CPU clock rate has to be assumed to be exact or at least to stay within specified borders.

In contrast to influences from the physical environment to the hardware, the correctness of hardware logic itself can and should be verified in similar ways as software correctness.

3.5.2 Software Correctness

Software and hardware logic follow semantics that are formally defined by the programming language (e.g. VHDL for hardware, ANSI C for software). External assumptions, like input contraints, are part of the specification. Thus, the semantics of the execution are well known, so that the correctness of the logic can be formally verified.
However, programming languages also bear undecidable properties as shown by Turing's proof on the halting problem in [Tur36].
The bottom line of the undecidability is that one cannot generally prove that an arbitrary algorithm will halt, though it is possible to prove it for specific subsets of algorithms.
Fortunately, most real world code can be verified to halt if it features specific properties. For example, fixed loop iteration bounds as well as prohibited use of recursions and function pointers can prevent straightforward termination issues. In addition, run-time error verification (e.g. pointer out of bounds write) and stack overflow protection prevent side effects that can lead to indirect termination issues.

3.6 Conclusion

This chapter discussed the main goals and challenges of this thesis. Those are:

1. The general **industrial applicability** goal of our approach requires an integration into current development processes and tools that are easy to use, precise and still able to guarantee time bounds.
2. The goal to tackle specific **automotive industry** related challenges requires the approach to take highly variant systems and their implications into consideration. Furthermore, the approach needs to be flexible and extendible in order to support the high amount of different hardware.

In addition to the aforementioned goals, the challenges of system complexity were discussed in this chapter.

4 Automotive Case Study

4.1 Motivation for an Example

PhD theses are meant to advance the current theoretical knowledge. Additionally showing the benefit of the gained knowledge in real world use is usually not a requirement. However, as this thesis focuses on the applicability of the developed approach, it is required to give a realistic proof of concept in order to explain the benefits for the industrial use-case.

4.2 Finding Suitable Case Studies

Not every example system is suitable to show what we want to achieve. As there are around 80 ECUs per car there will be some that do not exhibit the difficulties that most of the others do. Other ECUs may simplify the actual problems.
The following requirements can be derived from the main objectives of the TOAD approach (cf. Section 3.2).

- The system requirements have to include real-time properties.
- The system should be safety-relevant to necessitate a safe verification.
- The system should be non-trivial so that a manual verification is hardly possible.
- The system should cover the typical requirements of an automotive software system (mainly variability and distributed computing).

4.3 The Emergency Brake Assist (EBA)

A system which fulfils the identified requirements is the Emergency Brake Assist (EBA, cf. Figure 4.1) which is similar to an Active Brake Assist (ABA) in the Mercedes Benz Actros truck (cf. [JS06]). Such a system can be seen as an advanced adaptive cruise control (CC) with the special ability to detect and react to the emergency situations of impending unavoidable collisions.

The EBA uses a radar to detect obstacles in front of the vehicle and adjusts the vehicle's speed accordingly.

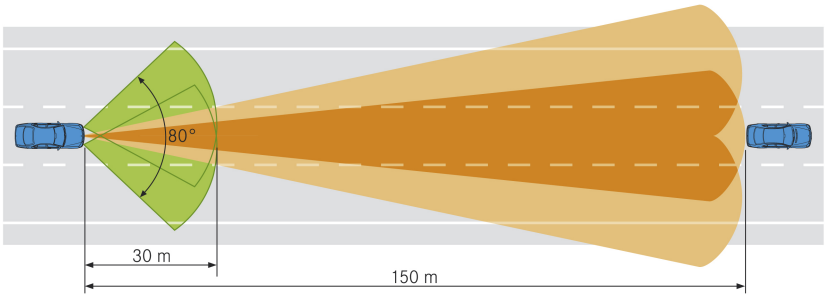


Figure 4.1: Schematic diagram of the radar lobes of the EBA system.

The logical architecture of an EBA system (cf. Figure 4.2) typically consists of six main components:

1. Radar - The radar processes the information of the various radar sensors (e.g. two close range and three long range radars) and transforms it into corresponding distances for particular angles according to each sensor.
2. Obstacle Tracker - The obstacle tracker uses the radar distance information to calculate the direction and relative speed of obstacles around the car (in our example the obstacles in front of the vehicle).
3. ACC - The Adaptive Cruise Controller evaluates the current situation in order to adjust the maximum allowed speed accordingly. In emergency situations it triggers the brake by requesting up to 100% brake pressure.
4. CC - The cruise controller takes the target speeds defined by the user and ACC and calculates the necessary torque on the powertrain (allows a limited deceleration compared to the brake).
5. Powertrain Controller - The powertrain component adjusts the torque on the engine according to the CC and opens the clutch in emergency brake situations.

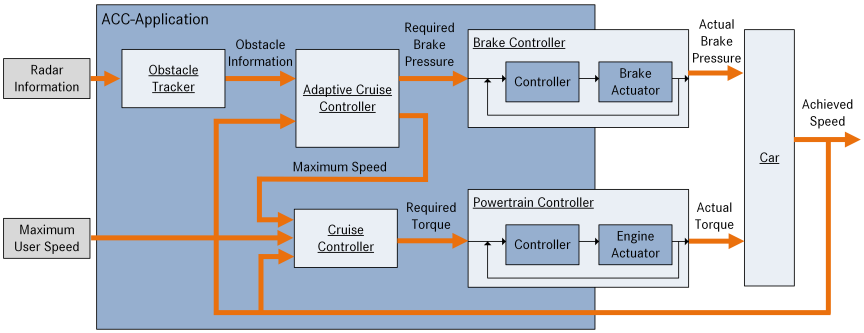


Figure 4.2: Schematic diagram of an ACC system with additional emergency brake functionality (EBA).

6. Brakes - The brakes translate the ACC's brake commands into brake actuator commands and executes them on the mechanics. For example, it contains the ABS functionality that prevents wheels from locking.

The potential deployment on different ECU nodes is mainly constrained by timing requirements. Thus, the radar preprocessing is either done directly within the radar sensor or on an ECU that is directly connected to the radar sensor. This saves bandwidth and thus reduces delays on the communication buses. Similarly, the powertrain and brakes software components are usually deployed on ECUs with direct access to the engine or the brakes respectively.
The obstacle tracker, CC and ACC software components on the other hand have rather limited data interfaces. The deployment decision can thus be made depending rather on the available CPU resources of the respective ECUs.
Today's adaptive cruise control systems are able to slow down the car in a specified range of deceleration by using the engine brake. Thus, no emergency braking is performed. This is mainly due to safety concerns as potentially faulty emergency brake interventions can lead to accidents instead of preventing them.
However, current developments in automotive system design are heading

towards a stronger interference in the cruising process as many accidents can only be prevented this way. An example of such a stronger interference is the Brake Assist PLUS in the actual Mercedes-Benz S-Class.
Such a system detects emergency situations with the radar sensors and increases the amount of brake pressure to the required level if needed as soon as the driver presses the brake pedal. Thus, the driver is still in charge of the trigger while the intensity of braking is controlled by the ECU.
The next step is a brake assist that is allowed to execute emergency braking independently of a driver's command. The Emergency Brake Assist (EBA) used for the demonstration of our methodologies is supposed to fulfil this requirement. Obviously, such an system requires a wide range of safety-specific development steps in addition to the timing verification.
Like most new automotive systems, the EBA is based on legacy software components and ECUs. In the case of the EBA these are the ACC ECU, which uses radar information to calculate obstacles and their distances, and the brake ECU, which calculates the necessary brake pressure depending on the current situation (e.g. different weight on each wheel) and a given deceleration value (by driver, ACC, etc.).
The new Emergency Brake Assist (EBA) represents an enhancement to the ACC controller as it adds an interface to the brake ECU. Hence, the existing brake ECU becomes part of the EBA system as it takes commands from the enhanced ACC component in addition to the driver's commands.
The EBA example features the typical elements of an automotive software system as it is distributed (and potentially further distributable) and contains several variant parameters.
Parameters that are relevant for the system timing are ECU hardware parameters like the CPU type, the clock rate of this CPU, the memory layout etc. The communication bus is usually the same for all system variants. Recent developments using laser and camera sensors however, lead to additional variant parameters for the used communication buses (e.g. type, protocol, bit rate).
Additional system parameters are the car or truck type that is meant to use the system (e.g. different maximum speed and weight) as well as the amount of radar sensors to be processed, the resolution of the obstacle detection lobe, etc.

4.4 Conclusion

This chapter discussed the need for an automotive case study with a suitable example. Requirements for such an example are given and the Emergency Brake Assist (EBA) is introduced as one such system that fulfils those requirements. The complexity of the system (in the sense of timing verification) can be seen in the distribution over several hardware components and the potentially high amount of system variants.
A detailed component specification for the EBA-system can be found in Chapter 8.

5 State of the Art in Timing Analysis

This chapter describes the current state-of-the-art by introduction and classification of timing analyses methods.

5.1 Introduction and Definition

Following the development of embedded real-time systems, timing verification methods are performed on the different development levels. Different verification approaches are applied for system, task and implementation level where the lower levels verify the assumptions made by the specifications of the higher levels (e.g. message response time verifications verify system level communication protocol specifications).
Starting on the system level, timing requirements are distributed using system architecture level models (c.f. Section 5.4). A verification on this level can already ensure, that the specified communication and ECU level computation times are consistent with the system requirements. The communication and ECU level computation times are then verified by WCRT analyses for the respective task or communication scheduling algorithm. The central input for the task response time is the WCET which represents an upper bound for the uninterrupted execution time of a machine code function on a particular hardware architecture.
WCET and WCRT analyses are well established methods in the timing analysis domain. Their current state and possible limitations will be discussed from Section 5.2.2 to 5.3. A less established timing verification level is the system architecture. While most developers assume that verified upper bounds on task and communication schedules sufficiently guarantee a system's timing behaviour, they miss the fact that safe bounds on end-to-end times require complete knowledge of all possible functional chains, synchronisation times between the tasks and communication messages, as well as all possible system states and system state transitions. Quite often error-, safe-, start- and shutdown states are not considered on end-to-end paths which results in unsafe system level timing although all single schedules are safe.

Hence, we distinguish three main classes of timing analyses: "WCET Analysis", "WCRT Analysis" and "System Architecture Level Timing Analysis".

5.2 WCET Analysis

The WCET represents an upper bound for the execution time of a specific software function, on a specific platform with a fixed set of parameters or external execution path constraints. A software function features varying worst-case execution times when executed on different execution hardware, but also for different sets of constraints (e.g. representing software or hardware variants).
Parameters or path constraints influencing the WCET are, for example, variant constraints (e.g. features a and b cannot be configured at the same time) or hardware constraints (e.g. the current project uses only one CAN (Controller Area Network [ISO93]) transceiver although the software supports up to four).

5.2.1 Preliminaries

The WCET does not take into account that there are possible interruptions of the software function under analysis by tasks or interrupt service routines. The methods for the calculation of the WCET value can be static (without executing the software on the target hardware, cf. Section 5.2.2) or dynamic (by executing the software on the target hardware, cf. Section 5.2.3). There are also approaches that try to incorporate the benefits of both paradigms [BCP03] but remain elaborated dynamic WCET analysis methods.

5.2.2 Static Worst-Case Execution Time Analysis

Time consumption occurs within information processing systems when the hardware executes machine-level commands and command sequences. That is why all static timing analyses (cf. Figure 5.1.) have models of specific hardware and use the software to be analysed as input.
A most precise timing analysis would take only executable paths into consideration while neglecting infeasible ones. But as the question of whether a path can actually be reached or not can be reduced to the halting problem, this problem is also undecidable. Safe WCET values however, are

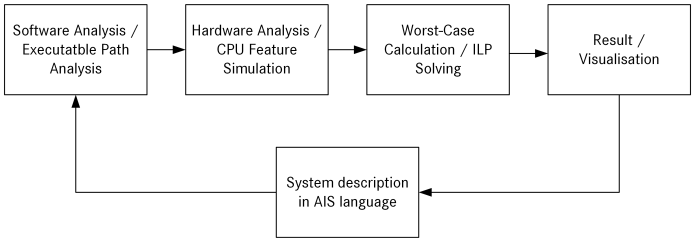


Figure 5.1: Workflow of a typical static WCET analysis exemplified on the tool *aiT* [Abs17].

not necessarily precise. Hence, tools like aiT (cf. [Abs17]) make use of the abstract interpretation (cf. [CC77]) which leads to safe but potentially overestimated sets of executable paths. Therefore, there will practically always be paths or loop iterations that cannot be executed but are part of the considered WCET path.
This overestimation cannot be determined as the actual WCET value is not known. However, an estimation can be performed by deriving the necessary input parameters for the calculated WCET path and executing such a worst-case scenario on the target hardware. That means, that by manually reviewing the calculated worst-case path, we can determine required inputs. Then we try to execute this path as closely as possible. Doing so might result in new insights on infeasible paths that can be modelled using specific constraint languages.
In order to determine a non-infinite WCET value, the upper bounds for the number of loop iterations and the number of possible recursion iterations have to be known at least. These values can be retrieved manually or partially automatically by static analyses.
But the calculation of a precise WCET estimate requires further restrictions of the control-flow than just the loop-bounds. It is necessary to know not only all possible value ranges of flow relevant variables but also the execution context in which specific values will occur.
The second step in the static timing analysis is the hardware and CPU feature analysis. There, safe overapproximations on the set of possible execution states are calculated for each program point in order to derive invariants for the estimation of a safe WCET with a low overestimation.

In the end, all feasible execution paths are weighted according to their execution time so that the calculation of a worst-case path can be performed. This is done by transforming the resulting control-flow graph into an optimisation problem. aiT uses the Implicit Path Enumeration (IPE) [LM99] to create an integer linear program (ILP) which can be solved with standard ILP solvers.
Successful case studies on static WCET verification have been applied to the automotive [MGL06b], space [HLS00, RSE+03] and avionics domains [The04, TSH+03]. Nevertheless, most of the case studies were performed on rather academic or small examples of code, whereas its application to highly parametric, complex industrial code has rarely been published so far.

5.2.3 Dynamic Worst-Case Execution Time Analysis/Profiling

Random Timing Measurement

Nowadays, the most often used timing analysis method is the dynamic execution time analysis, which is also called program profiling. It is most often used as it is the cheapest form of timing analysis and it is rather intuitive for the developers. Profiling is often performed by measuring the timing behaviour during functional tests that have to be performed anyway.
The profiling in the embedded domain sometimes requires code changes (i.e. instrumentation) so that the trace start and end points of tasks or routines can be logged. Modern development hardware is able to measure the execution time without generating additional CPU overhead.
A drawback of dynamic timing analyses is that the execution of embedded systems usually requires the completely integrated system to be run and hence, the execution of all tasks and interrupts. So the developers will only get late feedback on the timing behaviour and often not on the level of their own implementation but rather on a generalised task-level. This circumstance prevents an early or even the general detection of inefficient coding.
But the main issue is that it is virtually impossible to find an actual WCET by profiling, as the amount of feasible software execution paths combined with the possible amount of hardware execution states is enormously higher than the time for testing permits to execute.

Search-Based Timing Measurement

A more structured approach towards temporal behaviour testing is the evolutionary testing of real-time systems (cf. [Weg01, SBW01]). This approach is based on a structured selection of input parameters in order to maximise an optimisation function (also called fitness function). The fitness of a WCET analysis can be represented by the execution time which is to be maximised.
The selection of test parameter sets (individuals) can be performed by using ideas from the evolution theory [Dar59], in that parameter sets with high fitness values are selected (survive), recombined (parameter values are exchanged) and mutated (parameter values are randomly changed).
Wegener et al. showed in a case study [SBW01] that the WCET of all analysed tasks could be increased by an evolutionary search in comparison to the findings of the software developers. However, the case study was performed on rather small pieces of code (with a maximum of 119 lines of code per task).
It can be assumed that with increasing complexity of the analysed tasks, the determinism of the fitness function is reduced (due to interrupts, caching strategies or concurrent dependencies) and thus, the evolutionary search would be unstable and potentially degraded to a random search.
All dynamic timing analysis methods bear two important disadvantages: they cannot guarantee that a found WCET value is the actual worst-case and they cannot cope with highly variant systems as each variant would have to be tested separately in its specific environment (though pure software variants could be represented as inputs in simulated environments).

Probabilistic Worst-Case Execution Time Analysis

The probabilistic WCET analysis (cf. [BCP03]) tries to incorporate data from the static program structure with timing measurements. Therefore, it uses the basic block graph and measurements for each basic block. The execution times of the basic blocks are represented by their discrete probability density function which is estimated along the paths taken during functional test cases.
Knowledge of the basic blocks allows the identification of the non-executed blocks as well as the calculation of a worst-case block combination that has not been experienced during testing. This knowledge together with

the tested executed paths allows an estimation of probabilities for different execution times.
The probabilistic analysis of the temporal behaviour results in a probability distribution as depicted in Figure 1.1. However, as software-based systems are no natural processes and thus don't follow Gaussian distributions, the estimated probabilities tend to hide the fact, that the actual WCET might be far from the mean value. Hence, the probabilistic WCET analysis is helpful for less safety-critical systems (e.g. engine or gearbox) that have to fulfil hard real-time requirements but are able to enter safe states (i.e. fail-safe modes) in case of timing violations.

5.3 WCRT Analyses

5.3.1 Preliminaries

Scheduling of tasks and messages as well as the schedulability analysis are classical disciplines of operations research and computer science (cf. [LSD89, LL73]).
A schedule in this domain is a protocol that defines how computational entities are assigned to resources like CPUs or buses. Goals of a schedulability analysis are, for example, to show that these entities can fulfil their timing requirements, to estimate available resources or to optimise the resource usage. The Worst-Case Response Time Analysis (WCRT) is the central part of the schedulability analysis, as it delivers the response times that are verified against the specification.

Basic Concepts

There are several scheduling concepts which form the basis of a scheduling protocol (cf. Figure 5.2).

- **Concurrent Execution/ Preemptive Execution** - If multiple processing resources are used to execute software in parallel this is called concurrent execution (e.g. multiple ECUs, CPUs or CPU cores).
  Otherwise, if different software functions are executed on a single processing resource, as tasks, their execution has to be scheduled sequentially. If the scheduler is able to preempt the execution of one

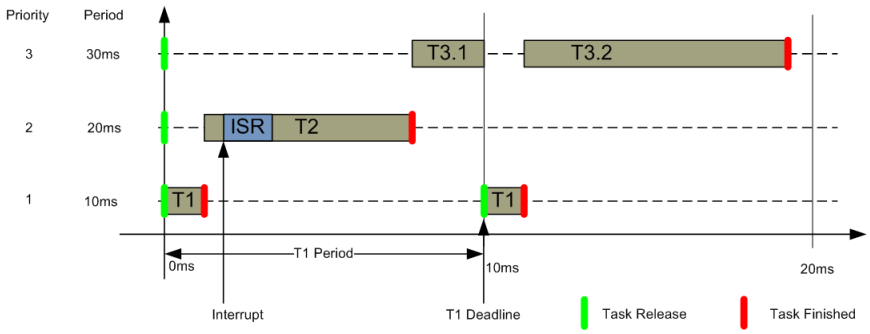


Figure 5.2: The basic concepts in a fixed priority preemtive task scheduling scheme using rate-monotonic scheduling.

task to allow another task to be executed, this is called preemptive scheduling. If the scheduler has to wait for a task to return from execution itself, this is called non-preemptive or cooperative scheduling. In the case of preemptive and concurrent execution, shared data has to be protected (e.g. by locks or semaphores) in order to prevent data inconsistencies. These mechanisms influence the WCRT-behaviour and need to be considered in WCRT-analyses.
It should be mentioned, that there are almost no pure non-preemptive schedulers used in embedded systems, as almost all of them use interrupt service routines that signal hardware events by interrupting the current execution.

- **Acyclic/ Periodic Execution** - The categorisation into acyclic and periodic tasks resembles the classification of event and time-triggered execution in Section 2.1.2. The input events of aperiodic actions (cf. Section 1.2) depend on external non-clock events while the input events of periodic actions are triggered by a clock in defined and fixed intervals, the **period**.
- **Aperiodic/Sporadic tasks** - Acyclic tasks can be further divided into sporadic and aperiodic tasks (cf. [Mar17]). In opposite to aperiodic tasks, sporadic tasks have a lower bound on the interval between successive releases that we call minimum task inter-arrival time. By this definition, aperiodic tasks can be released potentially infinitely

often at a certain point in time. Thus, they are generally not schedulable in a hard real-time system.

- **Interrupt** - An interrupt is a hardware triggered event representing the input event of an **interrupt service routine**. Therefore, interrupts are potential triggers for acyclic tasks.
- **Task** - A task represents an action in the model of Section 1.2.
- **Deadline** - A deadline is a scheduling parameter of a task which represents the time after which a task has to have finished its execution, including potential preemptions.
- **Response Time** - The delay between task/message release and task end/message arrival is called the response time.
- **Priority** - The priority is a task or message parameter that is used to regulate the order of execution or transmission. Low priority values usually indicate high priorities. The unique task priority of zero, for example, indicates that this task is executed first or is able to preempt other tasks that might still be executed.

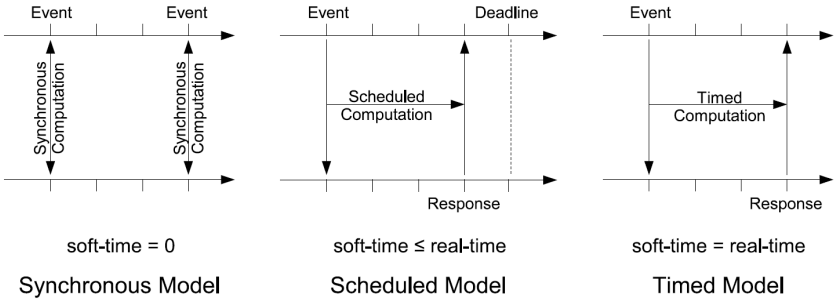


Figure 5.3: Real-Time Programming Models according to [Kir02].

Real-Time Programming Models

In [Kir02], C. Kirsch distinguishes three models of real-time programming by comparing soft-time (discontious time consumption of the embedded software) and real-time (continuous time flow in the embedding world).
In the synchronous model, the computation of a response starts as soon as

an input event arrives (cf. Figure 5.3). This model equals, for example, the concept of hardware interrupts and interrupt service routines.
The scheduled model requires a response to an input event to arrive before a given deadline has passed. This model is similar to most scheduled real-time tasks.
A timed model is a special case of a scheduled model where the previously computed response is not communicated before, but precisely at a given time (e.g. the deadline). This programming model can be helpful to reduce synchronisation overhead, as synchronisation takes only place at defined times. It can also improve the Worst-Case performance when potentially short event inter-arrival rates impact higher frequency tasks or messages (e.g. in CAN networks, cf. Section 8.6.3).

5.3.2 Classic Uniprocessor Schedulability Analysis

The classical approaches to scheduling analysis (e.g. [LSD89, LL73]) are usually specific for certain scheduling algorithms.

Common Scheduling Algorithms

This Section is meant to introduce only the three most typical scheduling algorithms for the embedded domain

Scheduling without Priorities One of the most simple scheduling algorithms is the round robin scheduling. Each task is queued in a predefined order and executed preemptively in same-sized time-slots following the first-in first-out (FIFO) principle.
Thus, the round-robin scheduling scheme is time deterministic but cannot handle asynchronous tasks like interrupts as that would reduce the fair share of CPU time.
That is why round robin is most often combined with the Fixed Priority Preemptive Scheduling in industrial applications. In these cases, a round-robin time-slot is executed during a single task period so that more time is available than a single slot would actually require.

Fixed Priority Preemptive Scheduling The probably most often used scheduling algorithm in the real-time world is the fixed priority scheduling (FPS). It has the advantage of remaining flexible through priorities while

being time deterministic. This algorithm is used, for example, by the standard automotive operating system OSEK [ISO05]. Here, a unique priority is assigned to every task. The active task with the highest priority is selected for execution as long as no higher prioritised task becomes active.
A scheduling is only possible if the tasks are released periodically or a minimum task inter-arrival time is known (e.g. for ISRs)[1]. The period or task inter-arrival time is used in the priority assignment protocol called rate monotonic scheduling which assigns the priorities monotonic to the rate (period) of the task. This algorithm was proven in [LSD89] to be optimal for tasks where the deadline equals their period (i.e. implicit deadline tasks).
For tasks with deadlines that are lower than their period, the deadline monotonic priority assignment[2] (DMS) is optimal (cf. [Tin00]).

Dynamic Priority Scheduling Dynamic priorities are used by the earliest deadline first (EDF) scheduling algorithm (c.f. [LL73]). The scheduling decision is made during execution by the dispatcher so that the task with the shortest remaining deadline is executed first. This algorithm has the advantage, that it is able to handle tasks with explicit deadlines (deadlines that are not equal to the task-period).
However, as this algorithm requires tasks with deadlines that are higher than their period to be efficient, it is not commonly used in industrial embedded real-time systems.

Response Time Analyses

The determination of a task's worst-case response time (WCRT) is the core task of a schedulability analysis. Knowing the WCRT and comparing it to the tasks deadline tells us whether a task is schedulable or not. The WCRT-analyses are usually very similar for the different scheduling algorithms (cf. [Tin00, TBW95]): The WCRT is the sum of the execution time (e.g. WCET) and possible delay times. As the length of the possible delays

[1] Otherwise every task could be released infinitely often at the same instance which would lead to an infinite response time.
[2] Priorities are assigned in order to their deadlines, starting with the highest priority for the shortest deadline.

depends on the WCRT itself, the solution is given by a fixed-point equation of the form:

$$WCRT^{n+1} = WCET + Delays(WCRT^n) \tag{5.1}$$

Examples will be given in the solution part of this thesis in Chapter 8.

5.3.3 Multiprocessor Schedulability Analysis

In recent years, multiprocessors were adapted by the automotive domain. Most of the automotive multiprocessor systems are identical or homogeneous multiprocessors (cf. definitions in [Mar17]) in the form of multi-core processors. But rather than using the different cores for an optimal distribution of existing tasks, the automotive use-cases today are independence of different development parties (with static assignments of cores to development parties) or independence of safety critical and non-safety critical functions.
Heterogeneous processor systems are used rather rarely in the automotive industry today (except for FPGAs where the specific tasks usually allow no task migration). Examples can only be found in infotainment and driver assistance systems. There, the different processors have dedicated tasks like car-communication, 3-d graphics rendering or object recognition.
Hence, homogeneous and heterogeneous multiprocessors in today's automotive systems have static task assignments so that each processor uses single core scheduling algorithms and the respective scheduling analyses.
Nevertheless, future high performance systems will need global task schedules that can balance the performance requirements between all available cores and potentially different processors. Then, earliest deadline first and rate monotonic scheduling need improvements as they are not optimal for multiprocessors anymore (cf. [Mar17]). Adapted scheduling algorithms like EDZL (earliest deadline until zero laxity) and RDZL (rate monotonic until zero laxity) give high priorities to tasks with zero laxity (time between a task's relative deadline and WCET) in order to improve the classic algorithms.
Further improvements like the incorporation of precedence constraints and support for heterogeneous processors can similarly be found in the literature.

5.3.4 A General Algebra for Scheduling Analysis

The classical scheduling analysis (e.g. [LSD89, LL73])) is concentrated on the analysis of concrete scheduling algorithms by describing resource usage using algebraic equations. In contrast, real-time calculus [TCN00] formulates a unified view on resource request and resource delivery to generalise the resource-usage and timing analysis.
Based on network calculus (c.f. Figure 5.4 and [Cru91]), Thiele et al. applied the max-plus algebra to compute the schedulability of several algorithms (cf. [TCN00]). Based on this Algebra, resource delivery and request functions need to be specified in order to analyse the design space of heterogeneous resource networks.
In order to deal with hard real-time bounds, the delivery curve β was in-

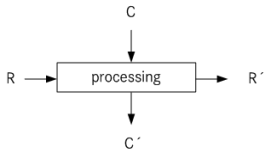


Figure 5.4: The basic model of network calculus [Cru91] with capacity function C, the remaining capacity function C', the request function R and the delivered computation function R'.

troduced as a lower bound to the possibly varying capacity functions C of a specific processor. Thus, β has to satisfy $C(t) - C(s) \geq \beta(t-s) \forall s \leq t$. The request curve α was similarly introduced as an upper bound for the request function R, and thus, has to satisfy $R(t) - R(s) \leq \alpha_\tau(t-s) \forall s \leq t$.

Using these formulas, a schedulability analysis can be derived, for example by proving $\beta(t) \geq \alpha(t) \forall t$.
The benefit of the real-time calculus approach is its general applicability for all kinds of resources. Even varying processor capacity can be modelled by this approach. However, the safe determination of delivery and request curves is similarly difficult as the analytic analysis of a specific scheduling algorithm.
If no safe determination is required, the request and delivery functions can be derived dynamically during tests.

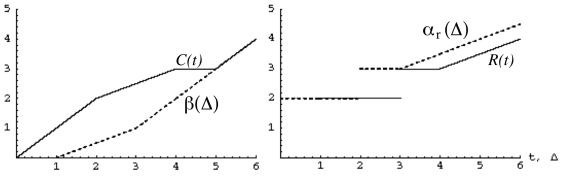


Figure 5.5: An example request function R and its request curve α_τ and a capacity function C and its delivery curve β (adapted from [TCN00]).

5.3.5 Scheduling of Distributed and Multi-Processor Systems

Many automotive systems are distributed. So their software-components can be located on different ECUs. The ECUs interact with each other by sharing sensor values (e.g. radar distances) or precalculated results (e.g. tracked objects) via network buses. A system's response time thus depends on the time used for the calculation on the ECUs and on the communication delay.
A response time calculation for communication messages can be performed similarly to the analysis of task schedules. As with task scheduling analyses, the difficulty of a message response time analysis depends mainly on the kind of network protocol.
Two major approaches to network scheduling can be found: asynchronous and synchronous message passing.

TDMA: Time Division Multiple Access The TDMA approach to message scheduling uses fixed time slots which are pre-defined for all messages on a bus.
In contrast to the task scheduling, the message delivery delay (comparable to the task WCET) is usually a known constant given by the bit-rate on the bus and the message length. Hence, the message scheduling on a TDMA protocol is deterministic by design.
Negative aspects of the TDMA approach are the high hardware costs due to the complex clock synchronisation on each network client as well as the inefficient resource usage due to potential empty message slots and only

partially filled messages. That is why even today the CAN bus [ISO93] (see below) is the most commonly used bus in automotive systems.

CSMA: Carrier Sense Multiple Access CSMA is a media access protocol strategy where the decision for a message transmission is made by the network nodes by waiting for an empty bus and by bus arbitration (identification of the highest priority message that currently awaits sending on a shared bus).
The most commonly used automotive bus protocol, the CAN bus [ISO93] uses this kind of media access protocol to schedule prioritised messages.
The disadvantage of the CAN protocol is its low deterministic timing behaviour. As the CAN nodes are not synchronised, all of them could theoretically start sending their messages at the same time, so that only a small number of highly prioritised messages has a non-infinite WCRT. Thus, a safe timing verification can only be estimated for a subset of the messages. Therefore, safety-critical systems can only be built upon the CAN protocol if few messages are to be transmitted and if non-zero message inter-arrival times can be guaranteed[3].
An example for the message response time analysis will be introduced in Chapter 8 and enhanced to enlarge the set of verified messages.

5.3.6 End-To-End Timing Analysis

Most system level timing requirements address end-to-end response times of system level functions. The end-to-end "ends" are usually the sensor input and the actuator output. The analysis seems straightforward as soon as the before-mentioned response times have been estimated.
But as already stated before, end-to-end analyses may be underestimated as complex system level states and interactions may result in more than a single end-to-end system path. It is the task of the System Architecture Level Timing (SALT) Analysis (cf. Section 5.4) to specify the set of possible system paths and to analyse their Worst-Case duration.

[3] This can be achieved by spreading the message delivery time on each client.

5.4 System Architecture Level Timing Analysis (SALT)

5.4.1 Preliminaries

In contrast to the timing analysis methods mentioned so far, the SALT analysis approaches are of a more general nature. They are not limited to specific development artefacts like messages, tasks or functions, but rather define functional chains of actions that link the before-mentioned artefacts on the system level.
In addition to high level timing aspects, they also cover functional and structural aspects of embedded real-time systems.
In doing so, they are central to the system development and thus close to this thesis' goal of an integrated timing verification. In the following sections, existing approaches will be introduced and compared with respect to the goals that were defined in Chapter 3.

5.4.2 AADL, EAST-EEA and MetaH

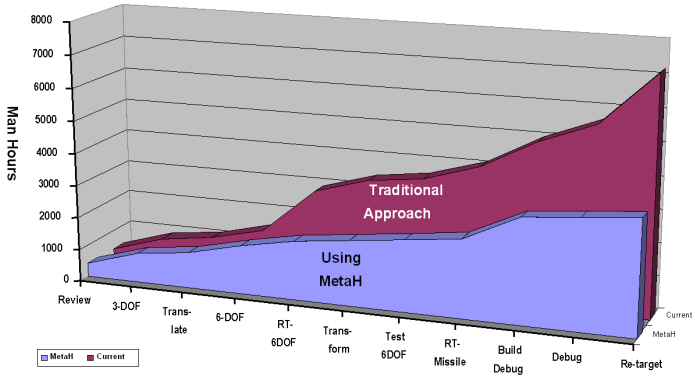


Figure 5.6: Benefits of the MetaH approach in man hours compared to traditional approaches. Diagram adopted from [Ves98].

The Architecture Analysis and Design Language (AADL, cf. [FLV03]) is an SAE (Society of Automotive Engineers) standard specification language for real-time component architectures. EAST-EEA (automotive AADL, cf.

[TEH+03]) and MetaH (aeronautic AADL, cf. [KVL98]) are closely related approaches which were brought together in the AADL.
Unlike the diagram-oriented UML (cf. [RJB04]), AADL is rather based on textual language-oriented definitions that make it more suitable for formal analyses than UML. AADL allows the definition of hardware and software elements on different architecture levels (ECU, processor, task). Based on these specifications, timing analyses are possible on the defined architecture levels. These analyses can be performed down to the level of tasks as it is shown in the work of Feiler et al. [FGHL04].
According to Vestal, the application of MetaH is beneficial when it comes to development costs (cf. [Ves98] and Figure 5.6). Hence, some requirements for industrial applicability seem to be fulfilled.
Like UML, AADL is generic and versatile in its application. It can be used to design every kind of real-time embedded system. So it forms an interesting candidate for our approach.

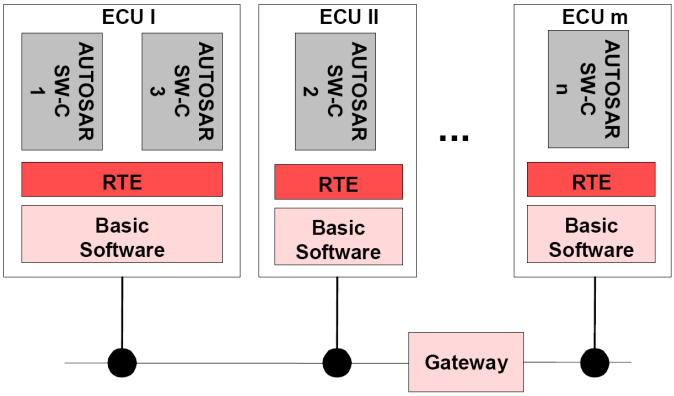


Figure 5.7: The AUTOSAR software component architecture (cf. [AUT07]).

5.4.3 AUTOSAR

In contrast to AADL, the goal of the Automotive Open System Architecture (AUTOSAR, [AUT07]) is not the definition of an architecture speci-

fication language but rather the definition of a concrete template software architecture for automotive systems. AUTOSAR is a standardisation attempt of several key players in the automotive domain (e.g. Daimler AG, Toyota, BMW etc.). Hence, it fulfils many automotive requirements as, for example, the variant description, reusability etc.
AUTOSAR is basically a software component architecture (cf. Figure 5.7) that focuses on portability by designing hardware-independent application components. The interfaces and properties of these components are formally defined using XML. This way, reusability of software components between different car lines and even OEMs is simplified. Nonetheless, timing verification is treated insufficiently by AUTOSAR, although all applications targeted by the standard bear real-time requirements. As this deficit was also realised by the AUTOSAR consortium members, the EU research project TIMMO[TIM07] was launched in 2007, aiming at this specific goal. The XML specifications could be extended to incorporate timing properties and variant information. A similar approach will be followed in this thesis.

5.4.4 Geodesic

Geodesic [dNR02] is a component framework for embedded real-time systems that features a "built-in" timing analysis engine called TimeWiz. The approach consists basically of an integrated design environment for the specification of functional and non-functional properties. An additional tool performs classic scheduling analyses on the specified architecture models.
Geodesic is rather a tool and as such it could be used as a basis for an integrated approach. But in order to be able to meet automotive requirements like distributed development and precise analysis of highly variant systems, many adaptations would be necessary that would be similar to a separate tool development.

5.4.5 SaveCCM

SaveCCM is a component architecture modelling language for embedded real-time systems developed at the Märladalen University (cf. [HÅCT04, CHP05]).
The approach uses formal methods for the verification of real-time systems during the development in the form of timed automata and the UPPAAL

model-checker. This approach is already quite close to an integrated approach, as the component model resembles the AUTOSAR approach and the verification using timed automata is generic enough to verify the entire event chain E2E response times.
SaveCCM features user defined attributes for the components which can be used to express variability. Furthermore, the reusability of SaveCCM components lead to a fulfilment of most of the automotive requirements.
Only with regard to usability (model checkers are difficult in a rather mechanics oriented domain) and highly complex systems do we see issues for a wide adoption of the approach. In order to analyse the capability of timed automata, this thesis performed a case study on the EBA system (cf. Section 8.3).

5.4.6 SymTA/S

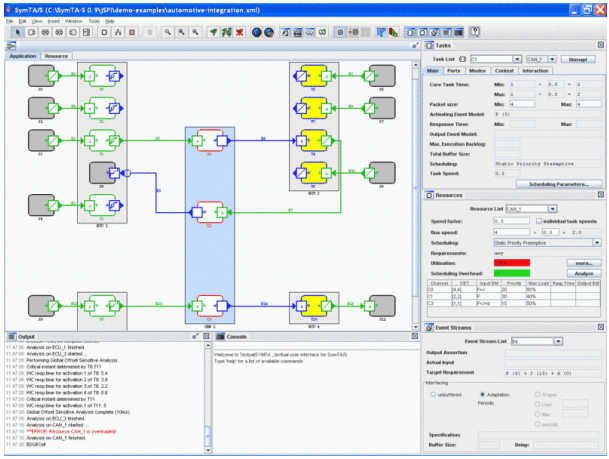


Figure 5.8: System specification using the SymTA/S tool suite according to [Sym07]

SymTA/S (cf. [HHJ+05]) is actually not a development approach but a system analysis tool developed by the University of Braunschweig and SymtaVision (cf. [Sym07]).

The computation theory behind the analysis engine is closely related to the event streams and arrival curves as they were proposed by Thiele et al. in [TCN00] and described in Section 5.3.4.
The necessary system description is set up in a graphical user interface as depicted in Figure 5.8. Tasks are created with their timing properties and assigned to processing nodes. The WCET property can be variable for different processing nodes so that the hardware platform's influence on the software tasks can be specified.
However, software components and their parameters cannot be modelled. Thus, a modelling of these influences is only possible by a detour on the definition of platforms for each possible configuration. This is an intricate task since the parameter influence has to be traced back manually, which conflicts with the goal of an integrated approach.
The benefits of SymTA/S can be seen in its additional ability to optimise a system's timing behaviour with regard to defined timing parameters and in the modelling of network comminication (including several bus protocols and gateways).

5.4.7 TDL and Giotto

The TDL (timing definition language) has been developed as a part of the MoDECS (Model-based development of Distributed Embedded Control Systems) project at the University of Salzburg. TDL is based on Giotto [HHK01, PPKT04] but offers a more convenient syntax, slightly changed semantics and an improved set of programming tools, according to [Nad05].
The TDL approach aims at a shift from a platform-oriented development towards a domain-oriented development of distributed real-time systems. Hence, the timing requirements are set independently of any hardware platform by the definition of platform-independent task execution times, the so-called logical execution time (LET), which equals response times like a task's period (cf. Figure 5.9).
TDL code is complied into so-called e-code in order to be platform-independent. Before this code is compiled, the specified timing properties are verified by a schedulability analysis based on constant WCETs, periods, priorities, etc.
The main idea behind TDL is the platform-independent execution that guarantees the same LETs on all supported plattforms. In order to do so, the generated e-code (which contains the scheduling information) is exe-

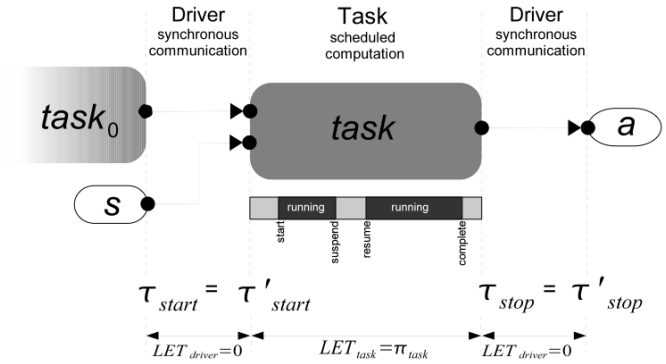


Figure 5.9: Input (e.g. sensor s) and output (e.g. actuator a) of a TDL task. (cf. [Nad05])

cuted by a platform-dependent interpreter, the e-machine.
However, as the WCET is not platform-independent, additional verification steps are necessary. Except for the LET concept, TDL can be seen as a tool like Geodesic (with a text-based description language rather than a GUI) that requires many changes so that it can be seen as an integrated approach (e.g. WCET calculation, variant awareness, bus communication). Nevertheless, the idea and concept for platform independence of TDL is interesting in the automotive domain. The AUTOSAR RTE can be seen as a similar approach, but with less focus on timing than on reusability.

5.4.8 Timed Automata/Model-Checking

The theory of Timed Automata [ACD93, AD94] is an automata theoretic framework with a dense time universe as a modelling language of real-time systems. The specification of real-time properties on timed automata can be done using the timed computation tree logic (TCTL) [ACD93], which is a timed adaptation of the computation tree logic (CTL, cf. [EC80]) from classical model-checking.
A timed automaton is a finite automaton equipped with a finite set of variables over the non-negative reals R+, called clocks. All clocks are syn-

chronized, that is, they run at the same speed, and can be reset to zero. A timed automaton is described by a finite set of locations and a finite set of jumps between locations. Time progresses when the automaton is at a certain location, whereas jumps are supposed to happen with no time elapsing.
The possibility of formal verification of real-time systems using Timed Automata has been demonstrated in several case studies. The three topics relating best to the embedded and automotive domain are [LPY01] (verification of a gear controller in UPPAAL), [HÅCT04] (a component model for embedded applications in vehicular systems; an adaptive simple cruise control system is analysed as an example) and [TB94, OS01, KH04] (verification of the CAN bus system). In all these case studies the considered applications are sub-systems of larger systems and could be verified as such, using tools like UPPAAL (cf. [LPY97]). Hence, analyses of complete EE system with bus communication has not been performed yet.
Timed Automata offer a powerful description language for real-time systems which makes them interesting for an integrated timing analysis. But in order to do this, timed automata have to be combined with static WCET analyses in order to incorporate hardware platform-dependent verifications on software function level. However, as we will show in Section 8.3, the analysis of Timed Automata suffers from the state explosion problem which currently prevents the applicability of Timed Automata for industrial level distributed and embedded systems and thus, for the automotive domain.

5.4.9 Comparison of the State-of-the-Art Approaches

There are many other tools and approaches to the problem of timing definition, analysis and verification that cannot be discussed here. Several of them are known to the author (e.g. VERTAF [HLS+02], TTP [HP02], DaVinci [Wer03] and Timed Petri Nets [BD91, RGdFE99]) but were not mentioned as the state-of-the-art in this chapter, as they do not provide additional features to satisfy the requirements that were set up in Chapter 3.
The goals of the TOAD approach that were discussed in Chapter 3 shall now be used for a comparison of the existing tools. The main goals of the TOAD approach are the *industrial applicability* and the *consideration of the automotive specific requirements* of scalability and variant awareness.
Depicted in Table 5.1 is our estimation of how the existing approaches ful-

Approach	Integrated Process	Industrial Applicability	Automotive Requirements
AADL, EAST-EEA, Meta-H	O	+	−
AUTOSAR	−	+	O
Geodesic	−	O	−
SaveCCM	O	−	+
SymTA/S	−	+	O
TDL/Giotto	−	O	−
Timed Automata	O	−	−

Table 5.1: Comparison of the current state-of-the art approaches.

fil our requirements using a simple −, O, + scale. And although none of them fulfil all our requirements, most of them bear interesting stand-alone concepts that will be considered by our approach.
Especially ideas from AUTOSAR, SaveCCM and SymTA/S will be recombined and extended with implementation level timing verification and variant awareness to form our TOAD approach.

5.4.10 Conclusion

This chapter introduces and evaluates several forms and levels of timing analyses. The three main categories WCET , WCRT and SALT analyses are introduced and examples for each of them are evaluated in detail. The results can be summed up as follows:

WCET Analysis

The biggest advantage of static timing verification is its ability to estimate safe upper bounds for execution times. This approach is most suitable for hard real-time requirements in safety-relevant systems. It is also the only technique that can efficiently address highly variant systems as given in the automotive software development. However, highly variant systems are rarely considered in the WCET research community yet. We introduced this topic in [MGL06a] and will discuss it in more detail in Chapter

8 of this thesis.
The main disadvantage of static timing verification that we can see is the inherent overestimation which results in unused resources and potentially high hardware costs.
The dynamic and hybrid approaches do not require hardware models as they use the actual hardware. However, they require test cases that cover all paths and states of a system to achieve safe results, which is virtually impossible. Thus, dynamic and hybrid approaches cannot verify Worst-Case Execution Times and neither can they cope with variability issues.
Therefore, dynamic and hybrid approaches are not considered for safety-critical or highly variant systems as they can be found often in the automotive domain. Hence, this thesis will focus on the static timing verification.

WCRT Analysis

For the WCRT analysis, we see many similarities within the analysis of scheduling algorithms and network access protocols. As a result, one can use the same general method (as introduced in [TCN00]) or the same scheme (the classic approach) to analyse worst-case response times (WCRT).
Whether or not to use a specific analysis method shall not be restricted by an integrated approach. Therefore, the solution part will discuss in detail, how timing analyses on different system levels (e.g. component, task, message, etc.) can be used in combination without a restriction to the kind of analysis method.
This thesis will make use of the existing approaches and show how they can be optimised when they are integrated.

System Architecture Level Timing Analysis

The System Architecture Level Timing Analysis approaches aim to integrate WCET and WCRT approaches in order to verify actual system level timing requirements. Thus, they are most interesting for our goal of an integrated timing verification. As none of the existing approaches is able to meet our requirements we need a combination and extension of the existing approaches.

6 TOAD Approach Overview

A comparison between the state-of-the-art (cf. Chapter 5) and this thesis' challenges and goals (cf. Chapter 3) shows a gap and thus, the need for an integrated timing analysis. The Timing Oriented Application Development approach (TOAD) aims to fill this gap.
As the TOAD approach is supposed to integrate with the entire development process, the single aspects of the solution spread over different development domains. Hence, this chapter sketches the overall idea that will be detailed in the following chapters.

6.1 TOAD - Timing Oriented Application Development

As timing behaviour and worst-case timing prediction are closely related to the software development process itself, the TOAD approach addresses the individual elements of a typical automotive software development process (cf. Section 2.4).
Thus, we are going to establish a timing oriented development process (TOAD-P), that makes use of the technical elements of the TOAD approach (i.e. software architecture and tools).
As depicted in Figure 6.1, the TOAD approach consists of three different development domains: the software architecture (TOAD-A), the tool chain (TOAD-T) and the development process (TOAD-P).
These domains will be motivated in the following sections separately before they will be introduced in the consecutive chapters in depth.

6.2 The TOAD Architecture (TOAD-A)

The underlying architecture TOAD-A defines the necessary artefacts of a timing verification, i.e. hardware and software elements that are used to describe the system under analysis.
TOAD-A also defines common basic concepts like interface communication (e.g. among software components) or scheduling of tasks and messages.
Using such a common architecture definition and restricting some basic

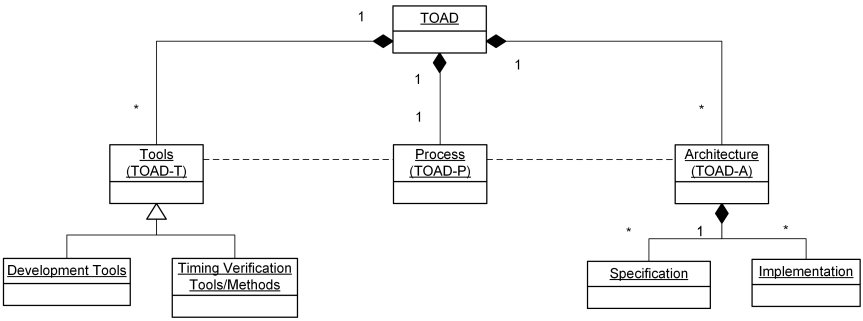


Figure 6.1: The TOAD approach for integrated timing verification.

concepts allows us to reuse existing verification methods without the need for a developer to delve into these methods himself (e.g. only a set of task scheduling mechanisms will be supported, but they are automatically verifiable).
So first we will derive software architectural requirements from the challenges defined in Chapter 3. Based on that, we will define an abstract software architecture. The intermediate level of abstraction intends to allow the adoption of commonly existing architectures and systems to our approach. Then, we will see in Chapter 7 how this abstraction will be adapted to a pre-existing software architecture.

6.3 The Tools and Methods (TOAD-T)

The tools and methods comprised by TOAD-T can be split up into the categories of development tools and methods as well as timing verification methods.

6.3.1 Development Tools and Methods

The development tools shall support the development of the non-functional timing aspects in addition to the functional aspects. The main requirement here is usability and process automation.
These tools will allow the model-based design of a software architecture

based on TOAD-A artefacts. This model-based approach allows us to restrict the development to the use of defined TOAD concepts as well as the use of verified code generators that guarantee interface conformance on the implementation level.
These development tools furthermore allow a single source concept for all architecture-related decisions and thus allows for the traceability of timing aspects on all development levels from the system architecture to the implementation.

6.3.2 Timing Verification Tools and Methods

Timing verification tools and their advancements form a central technical achievement of this thesis. These tools are integrated into the development tools in order to analyse the modelled systems or system elements.
Component timing properties like the WCET can be automatically verified by using aiT [Abs17] for example, while task and message timing properties can be verified by using different WCRT analysis algorithms.
The central tool-chain allows us to incorporate system-level information like variant constraints into all levels of timing verification. Thus, we are able to show the benefits of an integrated timing analysis.

6.4 The TOAD Development Process (TOAD-P)

One of the less technical but nonetheless important aspects of an integrated approach is the definition of a process that incorporates all the tools and concepts as a work package description. As we stated in our motivation, we see a main obstacle for wide adoption of timing verification in missing responsibility definitions, as well as missing technical competences when it comes to system level timing verification.
Our process defines the single activities of our integrated timing verification and how they can be distributed amongst different developers and potentially suppliers.
Starting with the definition of a required response time of an EE-system, identification of suitable hardware, design and implementation of the algorithms by different parties and verification of the specified timing properties are elements of this process.
Similar to TOAD-T this process will describe the abstract properties of our process so that existing processes can be easily adapted. Furthermore,

the automotive V-Model (cf. Chapter 2.4) shall be analysed and used as an example to show the adaptability of the TOAD-P process.

7 The TOAD Architecture (TOAD-A)

7.1 Introduction and Definition

The TOAD-A architecture defines abstract architecture artefacts (e.g. ECUs, software components, tasks, functions) and concepts (e.g. communication and scheduling protocols) necessary for a largely automated timing verification.
The formal definition also allows the description of system level constraints that spread down to software components and functions. This could be a variant constraint that is reflected by a software variable or a software architecture configuration parameter that is relevant to the timing verification only (e.g. component amount, interface property, task priority).
Hence, dependencies on timing behaviour can be made transparent proactively where today's timing analysis needs to trace back constraints from the code.

7.2 Requirements

7.2.1 Decomposition and Separation of Concerns

The abstract architecture definition of every timing relevant system concept and parameter used for TOAD-A allows a separation of different verification methods as well as the reuse of these methods on the same system elements in different architecture instances (e.g. particular schedulability analyses for OSEK-based architectures).
The integrated approach that was presented in [MNL06] used timed automata to model the timing behaviour of tasks and messages for a complete system. However, this approach does not only fail with regard to performance of the verification, but also introduces more possibilities for mistakes by the architects. This is due to the fact that basic concepts like communication transceivers often need several instances of the same state machine in order to describe concurrent behaviour.
Combined with the lack of computer science backgrounds in the automo-

tive domain, we strongly advocate the use of separation of concepts and timing analyses accordingly.

7.2.2 Explicit Timing Property Definition

In contrast to most existing architectures and programming languages, the TOAD approach makes timing properties and parameters relevant for timing behaviour on system, hardware and software level explicit.
As real-time consumption is a combined hardware-software property, constant execution times can only be defined for a particular hardware configuration. Parameters that influence timing behaviour like path constraints however are mostly hardware-independent. If there is a hardware dependency like an algorithm that iterates over the amount of available radar sensors, this dependency can be formally defined.
For flexibility reasons, the TOAD timing definition shall allow timing descriptions ranging from constant real-time values to parametric timing formulas that allow variant system components to be analysed.

7.2.3 Extensibility and Reusability

In addition to the flexible timing property definitions, TOAD-A and the derived architectures shall be extensible and offer a certain level of reusability. The TOAD-A architecture shall be portable to the majority of today's hardware and software architectures. This requires, for example, the hardware-independent definition of basic system types (e.g. uint8, sint16) and basic system functions (e.g. float multiplication).
This requirement should not be underestimated. A system that cannot be verified to fulfil its timing requirements on the current hardware and which is too expensive to be ported to higher performance hardware is at a dead end of its development. This worst-case situation might be one of the reasons that prevent today's developers from using static timing verification with its inherent overestimation.

7.2.4 Predictability and Accuracy

Restrictions on the hardware and software architecture may be necessary in order to achieve timing predictability. The timing behaviour of a common blackboard architecture [EHRLR80], for example, is hardly predictable,

since calculations can be performed event-based without idle times in between. And even if predictions are possible on such architectures, the worst-case scenario is typically far from the average-case.
The high difference between the average-case and the worst-case (shown in Figure 7.3) mainly results from missing discretisation and synchronisation of component execution. In order to be time predictable, acyclic tasks and interrupt service routines require defined and verified constraints on their arrival times.
An additional aspect of accuracy is the variance of system artefacts. As the worst-case timing behaviour depends on system parameters, the description of constraints on and between those parameters is needed in order to reduce overestimation (e.g. there can be either 5 standard or 3 high precision radar sensors on a system, but not 5 high precision radar sensors).
The run-time environment (RTE) of a software component architecture, for example, manages the communication among the application's components. Therefore, its timing behaviour depends on the amount of handled components, the communication and data types of the exchanged data. All these parameters need to be taken into account by the timing verification methods in order to accurately predict the worst-case times.

7.2.5 Automotive Characteristics

Finally, there are two central automotive characteristics that shall be supported by TOAD-A: the supplier management and the distributed computation.

Supplier Management

Due to the distributed development with many suppliers, composable software architectures and formally-defined interfaces are needed. With these concepts, logically separated and stakeholder specific system and software parts can be developed separately. But in order to achieve this, standardisation is required in the entire automotive industry. Hence, we follow the AUTOSAR approach as closely as possible.
In addition to standard functional interfaces, our approach requires all software components (supplier or OEM developed) to have verifiable timing properties that do not require additional implementation knowledge.

Hence, a standardised facility for timing meta information is to be added to the existing interfaces.

Distributed Computation

The typical automotive network architecture consists of around 80 ECU nodes. The functional distribution on this network is currently defined several years before the start of production. Timing verification however, may figure out bottle necks during the development that require a verified redeployment of functionality. Hence, TOAD-T shall support the virtual deployment of functionality.

7.3 TOAD-A

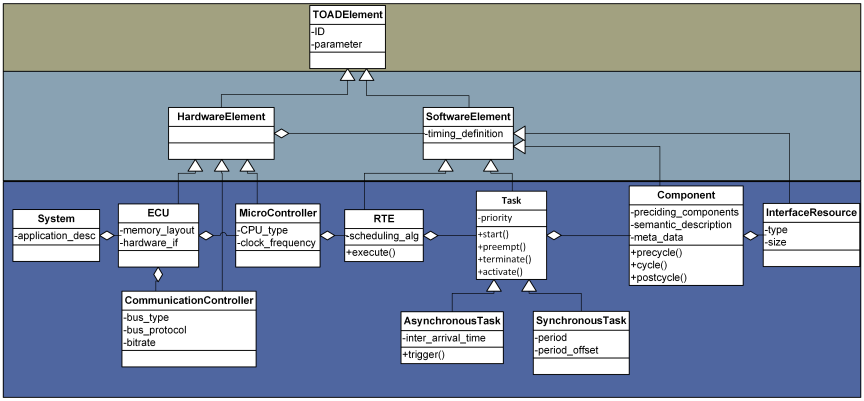


Figure 7.1: The basic elements of the TOAD-A architecture

According to Chapter 2, software component architectures like AUTOSAR are seen as most flexible and adequate to meet the requirements of an automotive system development. As it centres around the reuse of software components on potentially distributed hardware platforms by using formal interface definitions, this architecture shall be considered as a basis for TOAD-A.

However, in order to use the advantages of software component architectures in the timing verification domain, this thesis uses an extended definition of a component:

> A component is the aggregation of an interface description, defined semantics (functional information) and metadata (non-functional and system parameter dependency information).

TOAD-A (cf. Figure 7.1) is an abstract architecture that defines typical elements of a distributed real-time system as well as basic timing properties. This allows the TOAD architecture to be adapted by other architectures like Cadena [HDD+03] or AUTOSAR [AUT07].
Based on [BMR+96], an architecture can be characterised by four main views (logical, process, physical, development) which are discussed with regard to TOAD-A in the following sections.

7.3.1 Logical View

The logical view (cf. Figure 7.2 a)) of a TOAD application is defined by a set of components and resource elements. The resource elements are typed data interfaces that can be deployed on the same ECU or via bus network on multiple ECUs. By connecting the components with these data objects, a data interface is generated for each connected component in the code.

7.3.2 Process View

One of the most relevant views from the perspective of timing verification is the process view. This view defines the execution order (sequential, concurrent or preemptive execution) of functions, components and communication events.
These aspects are reflected by the parameters of the task (component order), OS/RTE (task and message control) and ECU (hardware platform) elements in TOAD-A. A task executes components in a sequential order according to given *precedence constraints*.
These are simple one-to-one relationships of the type "component 1 before component 2". As depicted in Figure 7.2 b), components 1 and 2 follow a precedence constraint which is indicated by an arrow.
The task scheduling algorithm in use is not limited in general by TOAD-A.

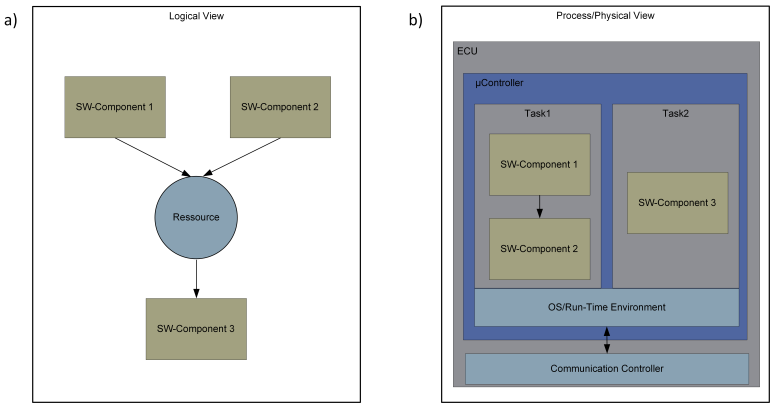


Figure 7.2: The logical and physical view for a TOAD application.

Hence, the adaptation of different concepts and operating systems is generally possible. However, as predictability comes mainly with synchronous or synchronised tasks, the prototype instance of TOAD-A uses the most common fixed priority preemptive scheduling algorithm.
A function-based synchronisation (e.g. start a particular task as soon as a particular other task has finished) can be achieved with the definition of preceding components within a task.
Although asynchronous tasks usually reduce predictability, they cannot be assumed to be non existing. They necessarily occur triggered by acyclic hardware events or remote requests which cannot be synchronised in time. As stated earlier, acyclic tasks need defined and verified constraints on their arrival times in order for the entire system to be predictable.
We favour the definition of minimum task inter-arrival times instead of statistical arrival times or arrival curves (cf. Section 5.4.6) in the industrial context. Statistical arrival times tend to be misunderstood or even misused by industrial practitioners as they invite for the definition of constraints without a sound basis. This is due to the fact that these statistics are usually derived from tests and the quality of these tests then defines the quality of the timing analysis. A safe and sound timing verification can thus not be guaranteed.

Arrival curves promise more flexibility than minimum inter-arrival times while defining fixed (and hopefully verified) upper arrival bounds compared to statistical arrival times. In contrast to statistical arrival times, they can be verified during execution using window watchdogs. But the definition of arrival curves in the industrial context would most likely be based on tests as well, which would basically mean that they are statistical arrival times undercover. Furthermore, each and every deviation from fixed task periods leads to higher worst-case response times or reduced available resources (e.g. a maximum of 3 executions within 30ms compared to fixed 10ms periods are depicted in Figure 7.3).
Minimum task inter-arrival times are simple to understand, verifiable during execution and can be enforced by hardware and software for systems that need such a high level of safety.
As asynchronous tasks should not be instantiated with levity, TOAD-A

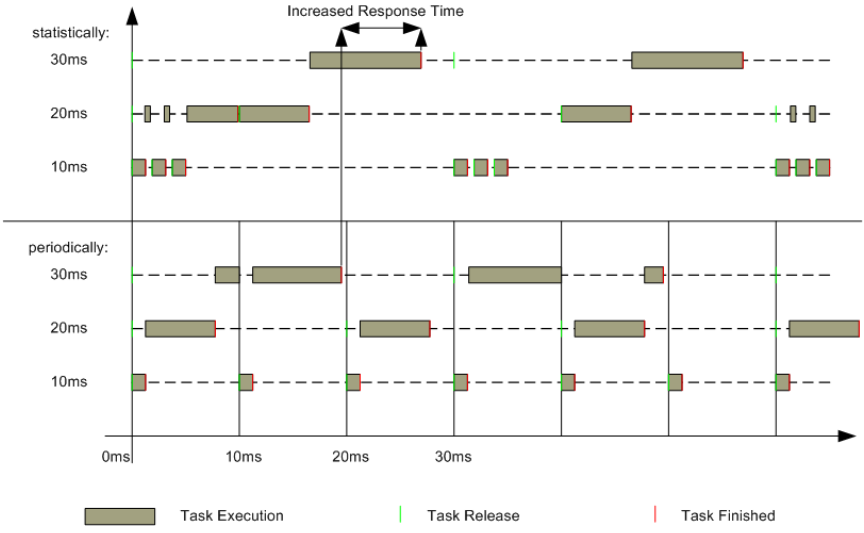


Figure 7.3: Synchronised task execution compared with unsynchronised execution using statistical frequency.

does not allow to deploy functional components within them. This limits

the circle of potential developers for asynchronous tasks to a selected group of software architects and hardware-related developers. This is one of several social and process-oriented means presented in this thesis that allows for an early development focus on predictability.
The special group of asynchronous task developers will split polling (asynchronous) and processing activities (synchronous) depending on timing and application requirements.
The specialists in such a group will also cover more hardware knowledge in order to be able to decide on the use of special hardware mechanisms like hardware interrupt schedulers as proposed in [RD05] or sporadic task servers (cf. [Mar17]). This way, asynchronous tasks could be shifted completely into hardware or at least their WCET could be vastly reduced.
TOAD-A incorporates transparent data exchange between components on different ECUs. On the process level, communication messages are available as soon as a component execution is finished and the respective interface variable is used by another component on a different ECU. Also the message protocol is not predefined. We will use the CAN protocol as an example as it is very common in the automotive industry.

7.3.3 Physical View

The physical view of a TOAD-A instance is displayed in Figure 7.2 b). The example shows only a single ECU although multiple ECUs are possible. The ECUs are containers for other hardware elements like micro processors or communication controllers.

Concurrent and Parallel Processing

The software components (as seen in the process view) are executed on a dedicated micro controller. A single software component is restricted to serial execution. Concurrency has to be achieved via separate software components in different tasks. Parallel execution can be achieved via separate or even the same software components running on different micro controllers or cores at once.
This is again due to the fact that most developers have difficulties in avoiding race-conditions and keeping predictability when using function-level parallelism. Nevertheless, experts in the field of parallel computing are able to provide specific basic software libraries if needed. In this case,

these experts will have to provide tools for the calculation of WCET and response times in TOAD-T (cf. Section 8) when using these libraries.

Additional Hardware

As it can be seen in Figure 7.1 a TOAD ECU may have a micro controller and a communication controller. However, most ECUs contain interfaces to different hardware like radars for the ACC system or GPS for a navigation system. These hardware components can be defined in TOAD-A by inheriting from the generic hardware or other existing hardware classes.

7.3.4 Development View

The Run-Time Environment

The run-time environment (RTE) handles the communication among the components so that they do not need to be aware of the physical location of the components they have interfaces with. Furthermore, the RTE triggers the execution of the tasks and components and thus defines the execution order within the tasks.
The RTE can be understood as an operating system abstraction which schedules the software component level and connects the defined data interfaces of all components.

The Enhanced Component Repository

A central aspect of the TOAD approach is the enhanced component repository as shown in Figure 7.4 on the right hand side. This repository functions as a central interface and data exchange hub for the entire TOAD development.
Any software component information is formally described in the repository via interfaces, metadata (e.g. timing information) and semantics description (i.e. source code). Hence, the repository also forms the basis for an easy reuse of its components.
The component interface description is defined by using the elements of the architecture definition in Figure 7.1. The timing is first defined as a different function for each software element (called timing_definition in Figure 7.1). Thus, the repository allows the definition of different scheduling algorithms for tasks or different WCET analysis methods for software

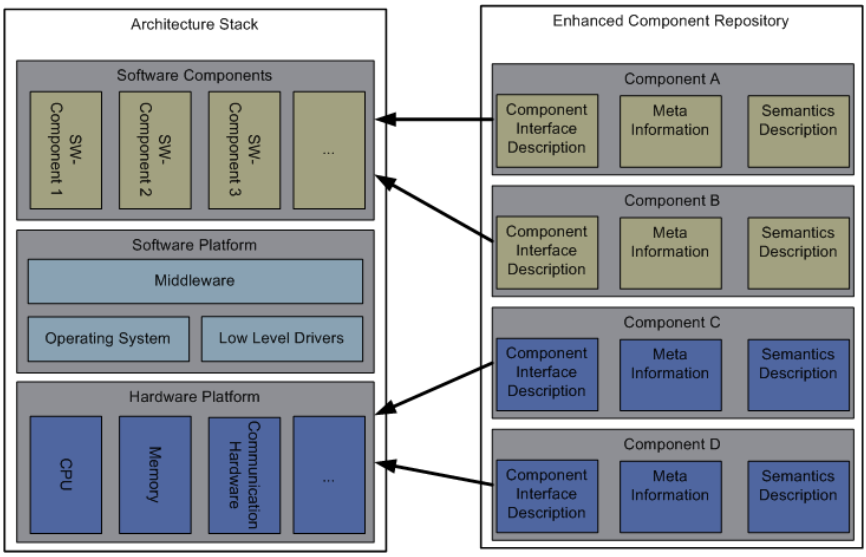


Figure 7.4: A common software component architecture (left) and an enhanced component repository (right).

components.
The semantic description is the least restricted information in the component repository. In general, this is a reference to the source code that implements the defined function interfaces. But many different semantic descriptions are possible and have been integrated into the repository. For example, feature models have been incorporated, that can be used to define the parameter dependencies which can be reused in the timing verification of highly variant systems.
The definition of additional meta-information or semantics descriptions can be accessed by the TOAD-T tools that implement methods of timing verification. The description of task properties, for example, can be accessed by task scheduling analysis algorithms.
In this regard, the enhanced component repository is an infrastructure for a variety of component information. The TOAD-T tool-chain uses this

information as an input for analyses and writes back the results to the repository for the potential use by other analysis methods.

Component Interfaces

According to the TOAD-A specification, every component implements three functions that are used for component initialisation, execution and destruction. These functions are called precycle, cycle and postcycle (cf. Figure 7.1). This basic functional interface is followed independently from the used programming language or code creation process (generated or manual implementation).
The interface functions for accessing interface variables and parameters can be generated. The respective code generator has been implemented for C and C++ code.
This interface knowledge is directly used for the timing verification. For example, the interface functions cannot be executed in the same task period. Hence, the WCET of the task executing a component equals the maximum of the WCETs of precylce, cycle and postcycle.

7.3.5 Differences to Existing Architectures

TOAD-A is based on a software component architecture that fulfils the requirements of flexibility (e.g. by parameterisable components) and reusability (by formal component interface definitions). Furthermore, it is well suited for the requirements of supplier management as the exchange of component information using the enhanced component repository delivers necessary information for a component integration without sharing too much intellectual property with the OEM.
In addition to common software component architectures, the component repository of TOAD-A contains additional meta-information as well as semantic information that is to be used by the timing verification but may also be used for other kinds of analyses. However, TOAD-A defines only generic and basic concepts that are required for all TOAD-A instances.
Based on these generic concepts, specific instances of hardware, scheduling algorithms or communication protocols have to be selected by the user of TOAD-A. Their timing behaviour is then defined using meta-information and analysis tools that interpret this meta-information.
In comparison to AADL, TOAD-A defines a concrete software component

architecture that limits the amount of potential interactions to the formally defined interfaces. The timing verification that is built on top of this software architecture can thus rely on the interface definition. This is especially true, if the RTE interface functions are generated the way it has been done for C and C++.
TOAD-A allows the incorporation of different software architecture styles into its components. Data-flow[1] or event-based[2] concepts can be realised using the TOAD concepts. Other software architecture styles such as the data-centric blackboard architecture however cannot be implemented on top of TOAD-A. But this is intentional, as such architectures are less time-deterministic and thus irrelevant for our approach.

7.4 Implementation of TOAD-A

7.4.1 TOAD-A RTE

The basis for a TOAD-A implementation is the TOAD-A RTE which is basically an abstraction of a specific operating system, like for example OSEK [ISO05]. Depicted on top of Figure 7.1, the RTE, like an OS, forms the interface between components running in tasks and the micro-controller.
The RTE executes the components in tasks depending on their defined period while preserving the defined precedence constraints. In addition, before and after the cyclic execution of components, the interface communication is handled by the RTE. According to the component's data resource usage, send and receive commands are used to update the respective interfaces to their most recent values. This includes the communication via bus networks.
The RTE could be implemented as an interpreter like a JAVA virtual machine. Such an RTE would read the TOAD configuration during execution and translate the timing and interface definitions into scheduling and communication commands. But due to resource limitations on most embedded devices, the generation of platform-specific code is typically the method of choice.
With the exception of the communication, TOAD-A does not define any other standard services like NVM (Non-volatile memory)-handling or diag-

[1] A TOAD system can be restricted to execute like a data-flow system.
[2] TOAD is mainly a time synchronous architecture but also allows the limited use of asynchronous tasks.

nosis. Those functions can be implemented as TOAD software components as they are not specifically relevant for the timing verification.

7.4.2 Adaptation of TOAD-A

The most efficient way of implementing TOAD-A is the adaptation of existing, similarly structured implementations. In this case, existing architecture concepts need to be mapped on TOAD-A elements and missing concepts need to be added to the implementation. As this is the most likely way that TOAD will be rolled out in the industry, this section shows an example for such an adaptation.
For that matter, the existing ANTS (Agent NeTwork System) architecture, which was proposed by Görzig in [Gör03], and its run-time environment (RTE) were adapted to the TOAD-A architecture. By doing so, TOAD-A directly supports two platforms: the OSEK/C166 and Linux/PC.
As depicted in Figure 7.5, most ANTS elements (for the OSEK/C166 plat-

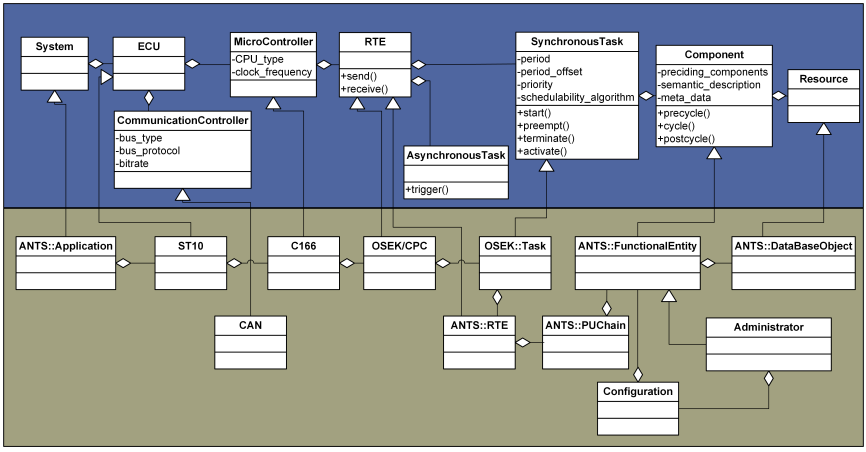


Figure 7.5: The mapping from the ANTS architecture running on the ST10/C166 platform (on the bottom) to TOAD-A (on top).

form) can be mapped to corresponding TOAD-A elements. The differences can typically be achieved by TOAD-A concepts similarly:

- The TOAD-RTE forms the OS-abstraction and thus triggers the task execution itself instead of using OS-specific tasks.
- The ANTS PU-chain concept (process unit chain) is covered by the TOAD component precedence constraints.
- The ANTS configuration defines sets of active and inactive components. Such sets form different application-modes (e.g. normal mode and diagnostic mode) and are resembled by different TOAD-A RTE configurations.
- The ANTS administrator is a specific ANTS component that triggers configuration changes. TOAD-A requires such changes to be performed on the software architecture level by experts with knowledge of the respective timing behaviour impact. In such cases, the RTE has to be adapted and the changes need to be reflected by TOAD-T tools for timing verification.

7.4.3 AUTOSAR

AUTOSAR [AUT07] is an automotive standard which resembles many commonalities with TOAD-A. Likewise with ANTS, AUTOSAR can be adapted to the TOAD-A architecture.
AUTOSAR is, like ANTS, very similar to the TOAD architecture. However, some differences can be found in the interface description and the execution concept. The AUTOSAR interface description is limited to the signal type and thus forms a subset of the TOAD-A interface description which allows complex data types (as shown by the mapping onto ANTS). The AUTOSAR execution and communication concept allows components to be executed periodically using a *sender-receiver* mode or event triggered using a *client-server* mode. Although the client-server concept can be implemented on top of TOAD-A's sender-receiver communication model, the worst-case timing behaviour would be equal to that of the sender-receiver model.

7.5 Developing TOAD-A Components

TOAD-A components are implemented on top of code frames that are generated from the component definition. Today, generators are implemented for ANSI-C or C++ files that contain the function prototypes for the precycle, cycle and postcycle functions as well as the get and set functionalities for each resource and attribute object.
The developer of a software component adds his functionality to the cycle functions where he makes use of the generated get and set functions.

7.6 Conclusion

In this chapter, the TOAD-A architecture and its core elements (run-time environment (RTE), enhanced component repository, etc.) were introduced.
As shown in this chapter, TOAD-A is based on a software component architecture that can be easily adapted to existing software architectures. Instances of TOAD-A are as flexible in use as possible (e.g. by using RTE generators and component specifications) but are also restricted to timing predictable software architecture concepts, so that the component developers have a limited impact on their application's worst-case timing behaviour.
The enhanced component repository allows TOAD-A components to be defined and implemented in distributed development scenarios while keeping necessary timing behaviour information transparent at all times. The explicit specification of component and interface parameters allows the definition of highly variant software components and applications. The central storage of all parameters and interfaces allows the later access of timing verification tools and thus the verification of highly variant systems.

8 Tools and Methods (TOAD-T)

8.1 Introduction and Definition

All tools and methods to be used by the TOAD approach are comprised within the TOAD tool suite (TOAD-T). What all the tools have in common is that they make use of the TOAD-A architecture and its component repository. The tools can be categorised into development tools (cf. Section 8.2) and timing verification tools (cf. Sections 8.4 to 8.8).
In the following sections, the words component and software component are used synonymously.

8.2 Development Tools

The main goal of the TOAD approach is an industrially applicable software development and timing verification process. In a first step the development process itself is automated, so that the timing verification can be performed as an extension to an existing tool chain.
Based on the specification of TOAD-A (Chapter 7), two tools are implemented for the development of TOAD applications: TOAD-CD (component definition) and TOAD-AD (for application definition).

8.2.1 TOAD-CD

TOAD-CD is a graphical user interface (GUI) for the component specification. This is done by defining the components' type, their communication interfaces and their parameters according to the TOAD-A architecture.
For example, the EBA system (cf. Chapter 4) requires an entity *ACC* of the *Component* type. As depicted in Figure 8.1, this ACC component references an *Obstacle* resource object containing the obstacle information from the radar.
Data objects types such as the *Obstacle* resource are defined in TOAD-CD

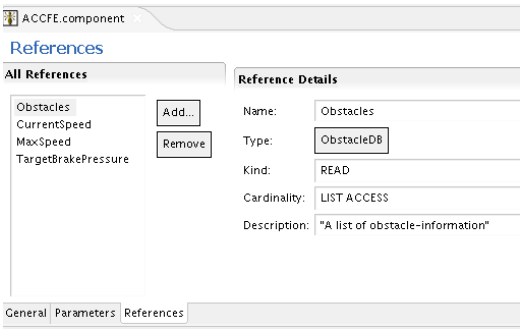


Figure 8.1: The component interface definition using the TOAD-CD tool.

similarly to functional components. Their basic type is the *InterfaceResource* as defined in TOAD-A. The resource types may contain parameters that are resource types themselves and thus allow the creation of complex type systems.
The *Radar* resource object (cf. Figure8.2) contains several parameters that represent the radar sensor information structure. A constant user-definable parameter is, for example, the resolution of the radar lobe.
Every component or resource that is described in TOAD-CD is stored in the enhanced component repository (cf. Section 8.2.3) so that they can be used for the component implementation and application development as well as their timing verification.
The necessary components and some example resource objects for the use in the EBA system are depicted in Figure 8.3 and Figure 8.2.

8.2.2 TOAD-AD

The TOAD-AD tool allows a GUI-assisted definition of TOAD applications on the basis of TOAD-A elements and the software components specified using TOAD-CD. A screenshot of the data flow definition is depicted in Figure 8.4 on the left hand side.
The connection between the functional entities (indicated by the FE suffix) is established through the usage of data objects of defined complex types. The process view (Figure 8.4 on the right hand side) allows the assignment

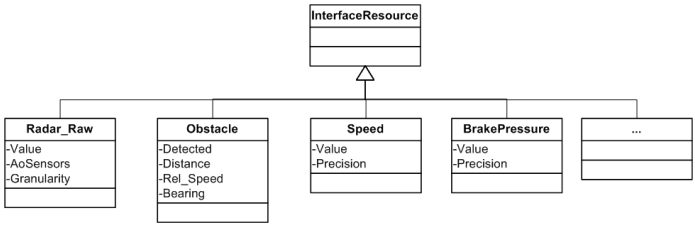


Figure 8.2: The partial resource object specification for the EBA system.

of the software components to tasks that are executed on defined processing nodes (ECUs). The tool also allows the setting of initial values for each system parameter (for components, ECUs, tasks, etc.).
TOAD-AD stores the application information in an XML file that contains the formal component definitions, the component's deployment and the data flow between the components.
Based on this description, the timing verification can be performed for the thus defined application.

8.2.3 The Enhanced Component Repository

The enhanced component repository (see also Section 7.3.4) is no tool itself, but a central data store for the communication between the different TOAD tools (verification methods, TOAD-CD, TOAD-AD). Additionally, it is a means for communication between TOAD developers (e.g. OEM and supplier). As mentioned in Section 7.3.4, the repository stores every component's relevant information (interface, semantics and metadata).
The storage structure of the architecture specification is derived from the TOAD-A definition. The metadata contains parameter specifications that are used for example for the timing verification. Semantics information is currently given by a reference to the source code which is a necessary source for a timing verification. Furthermore, references to design specifications such as timed automata can be given here (cf. Section 8.3).
The repository information is currently used by the following tools:

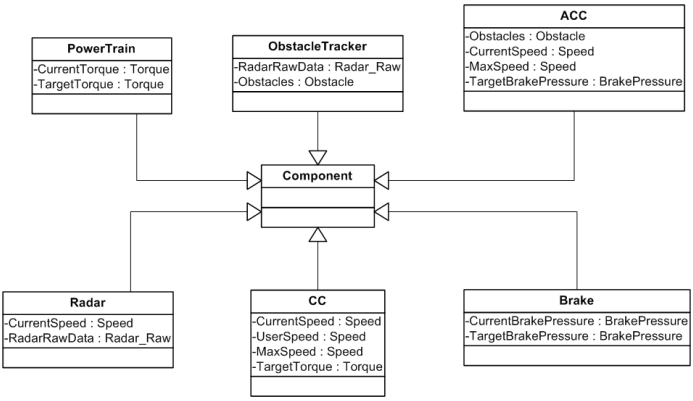


Figure 8.3: The component specification for the EBA system.

Code Generation

After the definition of software components using TOAD-CD, source code frames are generated as described in Chapter 7.

Application Development

The application development uses the repository as a system library and allows a model-based system development down to the software component level.

Timing Verification

The repository stores the timing-relevant information of each element of the system. Software components gain WCET values or WCET estimation methods for each interface function, task properties include deadlines, priorities, etc. and communication controller properties are for example bit rates of a bus.
The timing analysis and verification of a concrete application can be performed solely based on the application description and the repository information.

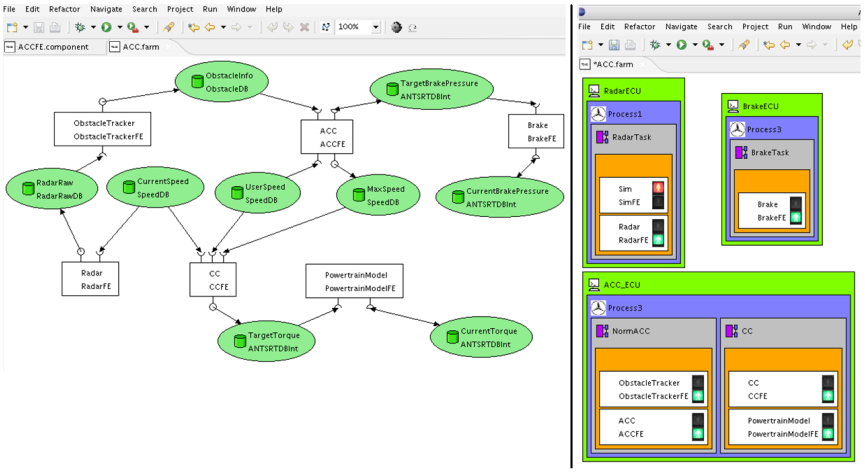


Figure 8.4: The data flow (left) and process view (right) of the EBA system in the TOAD-AD tool. As the simulation component is deactivated in the given configuration, it is not visible in the data flow view.

The next sections deal with the calculation of timing properties that are typically part of a system's timing requirements (e.g. end-to-end time).

8.3 Integrated Timing Verification using Timed Automata

The development tools and concepts mentioned so far are used to develop TOAD applications. Timing verification can be thought of as a bottom-up (e.g. WCET verification, task WCRT calculation, etc.) or a top-down approach (e.g. end-to-end definition, system timing budgeting, etc.). We recommend at least a top-down start as it results in earlier warnings in the case of problems and the consideration of timing issues in the entire development process.
Our integrated timing verification approach requires some central facility

for timing verification and a consistent model that represents the interplay between different system elements.
As our approach is to be capable of safeguarding highly safety critical systems with regard to timing, the formal verification capabilities of timed automata (cf. Section 5.4.8) makes them an interesting basis for the central representation of TOAD applications. In order to evaluate the feasibility of an integrated timing verification using timed automata, we performed a case study on the EBA system (cf. Section 4) that was presented in detail in [MNL06].

As mentioned before, the EBA system is a distributed system running

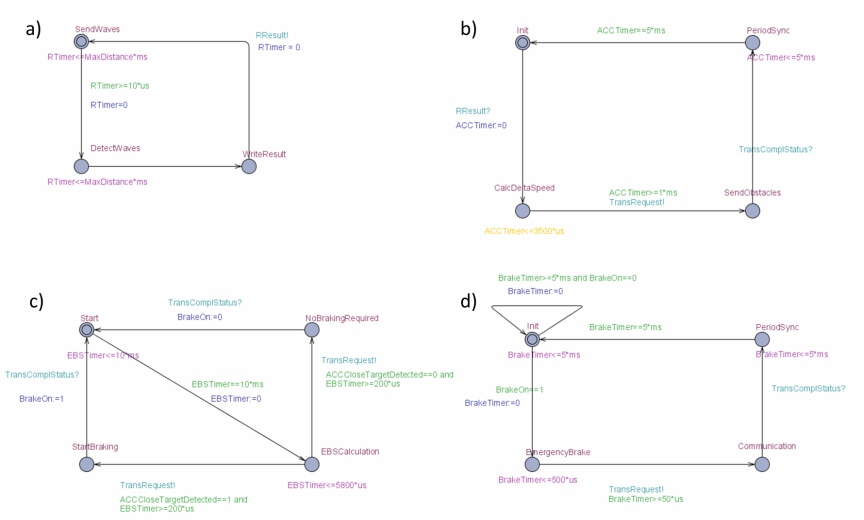


Figure 8.5: Basic models of the Radar a), ACC b), EBA c) and Brake d).

on three separate ECUs (ACC, EBA and Brake; cf. Figure 8.5). The communication is performed via a CAN bus, that is represented by a bus and a transceiver automata (cf. Figure 8.6).
The following timing properties were defined to be verified on the modelled EBA system:

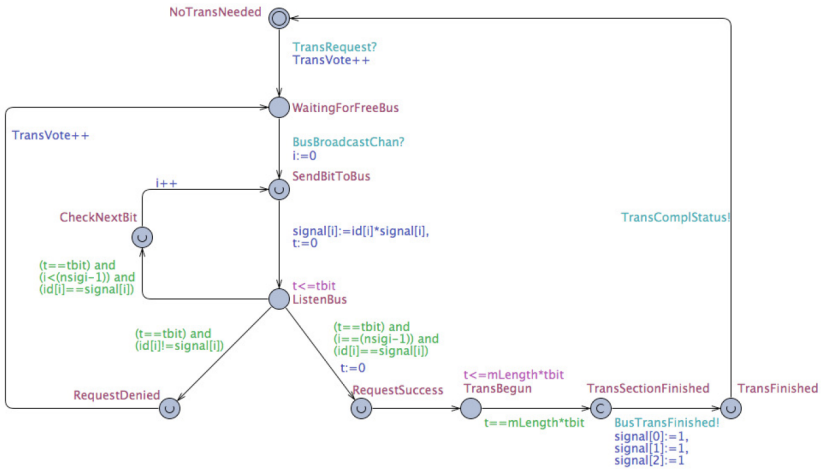


Figure 8.6: The CAN transceiver as adopted from [KH04].

- Is the emergency brake activated after an emergency situation has been sensed for a maximum time of 30ms?
- Is the system deadlock-free?
- Do ECU tasks consume less time than their period lengths (deadline)?

Using a time base of $1\mu s$ per tick, these properties are specified by the following UPPAAL TCTL formulas:

1. Radar.Close –> (EBA.EmergencyBrake and CloseTimer <= 30000)
2. A[](not deadlock)
3. A[](ECU.TaskFinished imply ECU.Timer <= Period)

The verification results given by UPPAAL look as follows:

1. A[](not deadlock) [**satisfied**]
2. A[](ACC.TaskFinished imply (ACCTimer <= 5000)) [**not satisfied**]
3. A[](EBA.TaskFinished imply (EBATimer <= 10000)) [**satisfied**]
4. A[](Brake.TaskFinished imply (BrakeTimer <= 5000)) [**satisfied**]

5. Radar.Close –> (Brake.EmergencyBrake and (CloseTimer <= 30000)) **[satisfied]**

As property 2 is not satisfied, the tool reveals a counter example that indicates the reason for the property violation. The given trace shows a response time of 5344 μs instead of the allowed 5000 μs, which is in turn a result of incorrect CAN priorities.
By changing these priorities in favour of the ACC, every property can be satisfied by the given system.
After the first step of system specification and verification, the model components, such as the ECUs, are to be refined. This leads to the addition of more functional details to the system. In a typical automotive development, this task can be performed by distributed development teams for the different ECUs.
Refinement can be achieved in various ways. It is possible to add locations, transitions, variables and new concurrent processes[1]. Figure 8.7 shows a possible refinement of the basic radar model (Figure 8.5 a)).
New locations and transitions were added to describe the influence of the

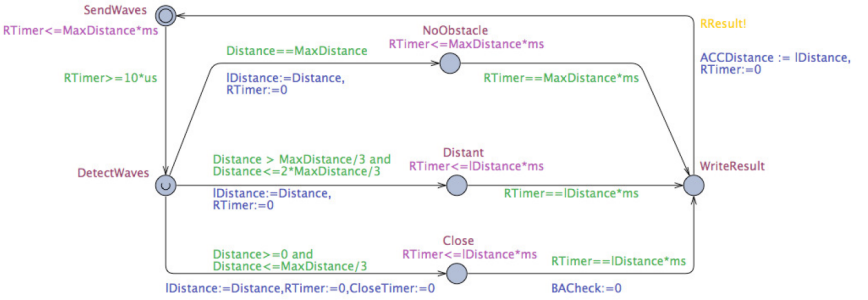


Figure 8.7: The refined radar model.

obstacle distance (NoObstacle, Distant, Close) on the detection speed. In the presented model the radar accesses a distance variable, which is provided by a new environment simulator model. This simulator changes the speed of two simulated cars to calculate a distance change in a predefined

[1] For example, an environment simulator for the component only.

range.
The timing properties of the basic and refined model are supposed be identical as the newly introduced locations together represent the former DetectWaves location, and the union of the new locations fulfils the formerly given invariant. However, the correctness of the transformation still has to be verified formally. In addition to the already mentioned refinements, a radar calculated distance value is now passed to the refined ACC. The refined ACC calculates the relative speed, which is sent to the refined EBA. There, the decision of whether or not an emergency braking is required is taken. This decision does not only depend on the relative speed (higher speeds lead to earlier emergency braking) but also on the distance to the obstacle. The result is sent to the bus so that a previously activated emergency brake might also be stopped. In contrast to the basic version, the refined brake actually retards the speed in the case of an emergency braking so that functional properties with regard to the distance between obstacle and car can be verified.
However, if one tries to verify the properties which we have already proved to be true for the basic model, the verifier tool fails, as the memory limit of a 4 GB RAM machine is exceeded. Here, we reach the limit of the UPPAAL application with our example. Nevertheless, some local properties could still be shown using larger computation resources. However, the main point of our example is that the use of a formal tool does have its advantages when employed in the early phase of a design process. As we have seen above, a design flaw was discovered early on and eliminated. On the other hand, the use of basic level models is limited. Model abstractions might reach a point where they are too general to deliver meaningful results.
Our analysis showed, for example, that the use of the presented CAN model is necessary on every abstraction level. A simpler and more abstracted communication model, e.g. without arbitration, leads to highly overestimated time bounds, especially for the safety-relevant components that are treated with higher priority by the arbitration process. If we use a model without arbitration, i.e. choosing the winning transceiver randomly, it is not possible to prove the Worst-Case deadline compliance anymore.
The most important part of the refinement process is the need to keep the already verified properties valid. That is, one would have to check that the refined model is related in some way to the abstract one in order to allow certain results from the verification of one model to carry over to the

other one. There exist several approaches to relate a refined model to its abstract one; see for example [Cer92, GSSL94, HK95, TAKB96].
For our purpose, the most intuitive one is probably the following: Let A be the abstract version of the refined timed automata B, then the behaviour of B should be mimicked by A, that is, every step of A can be done by B as well, for B is supposed to be a more detailed description of the more generic design A. We say that B simulates A. This idea of refinement is captured in the following two definitions.

Definition 1 (Simulation Relation - Transition Systems)**.** *A transition system $(S, \Sigma, \Rightarrow_S)$ where S is the set of states, Σ is the finite set of actions and $\Rightarrow$ the set of transitions is simulated by a transition system $(T, \Gamma, \Rightarrow_T)$, if a relation $R \subseteq S \times T$ exists such that for all $(s,t) \in R$ we have $s \Rightarrow_S^a s'$ implies $\exists t' \in T$ such that $t \Rightarrow_T^a t'$ and $(s', t') \in R$. The relation R is called a simulation relation.*

Definition 2 (Simulation Relation - Timed Automata)**.** *A timed automaton A is simulated by a timed automaton B, denoted by $A \leq B$, if the transition system $M(A)$ is simulated by $M(B)$ with a simulation relation R such that for every initial state s_0 in $M(A)$ there exists an initial state t_0 in $M(B)$ such that $(s_0, t_0) \in R$.*

The language $L(A)$ of an automaton A is the set of all execution sequences of A. Note that simulation is more strict than language inclusion, that is, $A \leq B$ implies $L(A) \subseteq L(B)$. However, the converse does not hold. Indeed, the language inclusion problem for timed automata is undecidable (cf. [AD94]), whereas the simulation relation (and also the bisimulation relation [Cer92]) can be decided in EXPTIME (cf. [TAKB96]). The authors of [TAKB96] have implemented a simulation checking procedure in the COSPAN verifier, that is, given a relation between two automata we can check if it is a simulation relation. However, giving the simulation relation in each design step is probably not very practicable. A fully automatic check would be more reasonable.
However, as we have seen with a rather small example that uses no variant mechanisms at all, this approach is not capable to handle complex automotive system. Hence, the timing verification using a timed automata model on all scopes will prevent us from the goal of an integrated timing analysis for industrial applications. Hence, scope-specific methods (categorised into

WCET analysis, WCRT analysis and end-to-end analysis) will be evaluated in the following sections.

8.4 Static WCET Analysis

As introduced in Section 5.2, the WCET represents the single, uninterrupted execution time of a software function. Unlike most industrial WCET verification attempts known by the author, our approach is to cover all software functions from platform- over RTE- to the software component-level. This section will introduce the WCET analyses for TOAD-A applications and consider variations of the standard WCET analysis approach, in order to increase the analysis performance depending on the degree of variability.

8.4.1 Platform Analysis

Platform services like drivers, error management or NVM-handling that are executed independently from the application make contributions to the response time of the system. Therefore, interrupt service routines (ISRs), system tasks and context switches require separate WCET analyses.

ISR Overhead

ISRs are asynchronous tasks with priorities typically higher then those of the synchronous tasks. These scheduling properties usually require the ISRs to be simple and straightforward, thus reducing the execution time jitter (i.e. the difference between BCET and WCET). Their WCETs can usually be calculated with low effort using tools like aiT. A case study by the author (cf. [MGL06b]) revealed that generally the ISRs do not require sophisticated AIS annotations (AbsInt Specifications, cf. [FHT03]). Hence, for each platform (i.e. hardware and software platform) a single WCET value is calculated and stored in the TOAD repository.

Context Switch Overhead

A more complex impact on a platform's WCET occurs due to context switches that occur on every task preemption. The WCET of a potential context results from a possible worst-case task context consisting of cache and register content. Note that the worst-case context does not necessarily

imply filled pipelines or empty caches (cf. timing anomalies in [RWT+06]). This problem was addressed by [Sch02] and can be incorporated in the analysis tools for the respective platforms. Due to the fact that our evaluation platform (C166/ST10) supports single cycle context switching, the resulting constant overhead is regarded accordingly in the WCRT analysis.

Operating System Library Calls

The platform-dependent operating system (OS) typically provides library functionality that can be used by application software running on the OS. As TOAD-A prevents direct access from application components to platform-dependent functions, OS library calls are analysed in the context of their use by TOAD-A RTE functions. This way, the context specific WCET will most likely bear a lower overestimation (cf. Figure 8.8).

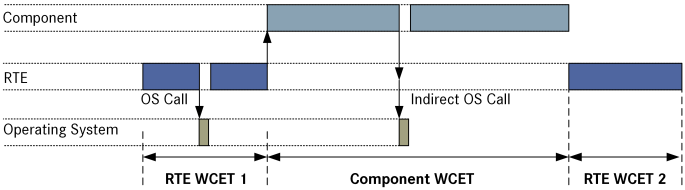


Figure 8.8: The separation of WCET analysis on the RTE and component level.

8.4.2 RTE Analysis

Although it is possible to analyse a complete set of components that is executed by the RTE, a separate analysis of each component and the RTE has several benefits. First, independent analyses shall be possible by different development parties. Second, component developers shall not need functional or timing knowledge beyond their own component. Furthermore, the separate calculation of WCETs for independent software components is usually faster and can even be more precise than a combined analysis (cf. [MGL06b]). Finally, independent WCET analyses for software components are also made easy by the TOAD-A architecture due to the defined

interfaces and parameters.
Different configurations of the RTE lead to different WCET values. Each additional software component to be executed by the RTE, for example, leads to additional call overhead on the RTE. Thus, the RTE's WCET is only bounded for a given configuration.
The estimation of an RTE's WCET requires at least six loop annotations. The given maximum loop-bounds depend, for example, on the number of components and the maximum amount of data references by each component. Hence, these loop-bounds can be automatically derived from the component repository.
The case study revealed that aiT is not able to calculate the loop-bound automatically for a given configuration. Hence, it is not able to calculate context-dependent loop-bounds. For example, if there are two components and one of them references two data entities and the second references just a single data entity, aiT assumes the same maximum amount of iterations for registering two data entities for both components as it is not able to distinguish the execution contexts. Therefore, the overestimation increases with the difference between the average and the maximum amount of interface references used by a component. Our approach reduces this overestimation by using configuration knowledge from the TOAD repository.
Reducing the overestimation and additionally improving the RTE-WCET estimation's performance is a method that we call *analytical analysis*. That is, aiT and AIS annotations are used to analyse the WCET for a number of loop-bound combinations. From the mapping $l_{i,j} \rightarrow WCET(l_{i,j})$, where $l_{i,j}$ is the loop-bound for loop i in configuration j, we approximate an upper bound algebraic equation for the WCET calculation as described in [MGL06a]. For the TOAD-RTE, these equations are the following:

$$t_{Core} = t_{CoreO} + k \cdot (t_{CoreC} + t_{ChainO}) + \sum_{i \in l} t_{ChainC} + t_{C_i} \tag{8.1}$$

$$t_{C_i} = 2 \cdot (t_{DataO} + m_i \cdot t_{DataC}) \tag{8.2}$$

According to these formulas, the WCET of the RTE-core routine (t_{Core}) results from a core routine offset (t_{CoreO}) and the number of executed component chains[2] k. The component chains are executed in a loop with an iteration WCET (t_{CoreC}). Every component chain executes the com-

[2]The component chain is an ANTS concept that has been taken over in the adaptation.

ponents in a loop which results in an additional offset (t_{ChainO}) and an iteration time t_{ChainC} per component.
Each data entity of a component is locked before and unlocked after an execution cycle. That leads to an offset (t_{DataO}) and a loop iteration time for each data entity (t_{DataC}).
All values t_{CoreO}, t_{CoreC}, t_{ChainO}, t_{ChainC}, t_{DataO} and t_{DataC} can be calculated using aiT for a given micro-processor as constant coefficients in CPU cycles. The only variable values are k, the number of component chains, and m_i, the number of data entities for each of the l components. These variables can be derived automatically from the TOAD-A repository. When performing this kind of analytical analysis using aiT on a specific generated configuration, it is important that no values that are volatile in the TOAD configuration are fixed constants in the generated code or otherwise available to the aiT analysis. Complex flow dependencies (e.g. by nested loops) and variable dependencies (dependencies that are estimated by aiT) have to be considered in the analytical analysis for other TOAD-A derived architectures.

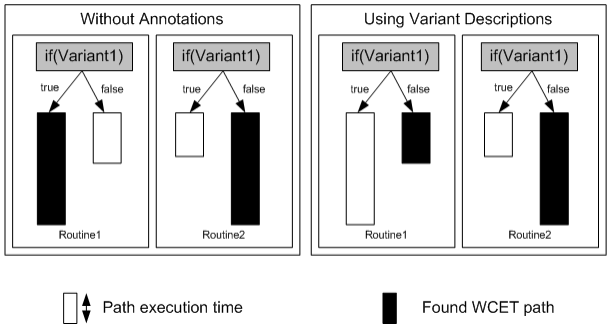


Figure 8.9: Timing overestimation due to software execution variants and the proposed solution using variant descriptions.

8.4.3 Software Component Analysis

The interface routines (precycle, cycle, postcycle) are to be analysed for each component separately. The influence of variant parameters on timing

analyses, as we have seen for the RTE, is also a source of WCET overestimation for most TOAD components. loop-bounds that cannot be determined by aiT are often indicators for variation points. In such cases, the loops are often bound by volatile variables that resemble a variant interface for external use.
The overestimation by static WCET analysis can only be assumed by comparison with timing measurements. A part of this overestimation can be attributed to variant parameters as depicted in Figure 8.9. In Section 8.5 we will see several examples for the degree of variant-related overestimation.
The identified variables can be further divided into "system variables" or "mode variables":

- System variables are constant variables during the system's lifetime. They often represent the hardware variant of the embedding system.
- Mode variables can be changed during the execution of an application. I.e. by pre- or post-cycle functions or by other software components via the RTE. Mode variables (typically interface variables or parameters) represent system execution states or modes that potentially define mode-specific infeasible paths.

In the best case, the component developer knows all execution variants or has a formal variant description. The TOAD approach eases the knowledge of variants since every component is developed with regard to a defined interface and parameter specification. Thus, variant configurations that should only depend on interface variables or component parameters are easily identifiable.
A WCET analysis case study on a powertrain system revealed that developers also use internal modes in order to reduce the execution time. Different modes were used in the form of component-level time slicing. In our example, we could see that the four time slices used by the developer were unbalanced and thus inefficient for the entire system. This finding further strengthens our belief that component developers should not be responsible for timing analysis or optimisation in general.
Using the TOAD approach, we can even argue that such potentially erroneous concepts are not necessary at all:

- Component-level time slicing is often used out of fear for dependencies within the same task that prevent an easy redeployment. One

of the central TOAD concepts is the transparency of interface and parameter dependencies that eases the redeployment of components.
- Time slices are also used to pass intermediate results to other components in a given time slice. Such constructs indicate suboptimal component design, as intermediate results that are needed by other components indicate functional bounds suitable for a component split up.

However, there are further code constructs that lead to WCET overestimations than mode parameters. This appeared in our analyses with complex control-flow dependencies such as nested loops where the inner loop-bound depends on the outer loop-bound.
aiT cannot use relative control-flow annotations on loops (e.g. loop2 iterates half as often as loop1) and thus requires memory annotations instead (e.g. instruction A is half as often reached as instruction B). What seems to be just inconvenient for a user, prevents the automated use of such annotations in component architectures. This is due to the fact that the memory addresses are likely to change when components are redeployed in different systems.
Two solutions to this problem are possible: either the annotation language of aiT is extended or the analyses are performed using the analytical method that was introduced in Section 8.4.2.
However, analytical analyses cannot always be used. Especially for highly complex components with many non-linear dependencies the analytical method tends to be error-prone.
That is why we came up with a second method that is able to cope with more complex dependencies than the analytical method. As Integer Linear Programs (ILP) are used to describe the program flow dependencies by aiT, dataflow dependencies can be found in the ILP similar to the dependencies described by the algebraic equations of the analytical method. Hence, the aiT-generated ILPs can be analysed and parameterised for the variant timing estimation.
An initial ILP can be found by using aiT on the component under analysis with the respective annotations. The variability information is to be found manually in one or multiple ILP constraints of the form $ax_1 + bx_2 <= cx_3$. These constraints are found by analysing the difference between the resulting ILPs of different configurations. The differences are then parameterised so that the ILP can be solved for different configurations.

Method	Parameter Complexity	Performance ([s] on our example)	Error-Prone
Fixed Values	no variability	highest (0)	low
Analytical	low	high (10^{-7})	high
ILP Analytic	medium	medium (17)	medium
AIS Parametric	high	low (54)	low

Table 8.1: Time determination method comparison.

The analysis of our example showed the general applicability of this method, but also revealed potential overestimations in some special cases (but no underestimation). These cases can only be identified by manual reasoning on the pipeline behaviour. Configurations leading to such cases have to be estimated using aiT instead.
In [MGL06a], the author showed that this method works well with nested loops on the C166 processor. Other microprocessor architectures may increase the effort for safe ILP analyses with low overestimation.
Finally, the existing methods (fixed value and AIS parametric) are compared with the newly presented methods (analytical and ILP analytic). As the selection of a suitable method depends on several parameters, the main decision criteria are displayed in Table 8.1. An analysis method is assumed to be error-prone when the manual effort and the necessary considerations (like cache and pipeline behaviour) are high. Hence, the parameterisation of AIS annotations (AIS parametric) and the fixed values are just slightly error-prone, as they form the typical case of an aiT analysis and form the basis for the newly introduced methods.
If high analysis performance is required and high parameter complexity is present, the newly introduced methods should be used. The example of the RTE analysis shows the successful application of the analytical method. As the RTE is used in every TOAD application, the necessary effort to ensure the correctness of the derived equations is easily justified.
Finally, all four methods for WCET verification are available for component developers: Fixed Values (there are no parameter dependencies or the dependencies can be set to fixed values), Analytical (dependencies are linear and simple), ILP Analytic (dependencies are more complex) and AIS Parametric (dependencies are complex and can be described using AIS annotations).

Since our analysis, further research on parametric timing analyses has been going on, for example, in [EA11]. In his work, Altmeyer derives the execution variants directly within the analysis tool by searching for variables that are read before they are written. This approach is less error-prone than the presented ILP analytic method since no user interaction is required. However, Altmeyer's approach cannot take external information on parameter dependencies into account and thus, reduces only a part of the overestimation. Furthermore, the approach is currently restricted to small portions of code (due to the parametric LP solver).

8.4.4 Improving the Loop-Bound Detection Rate

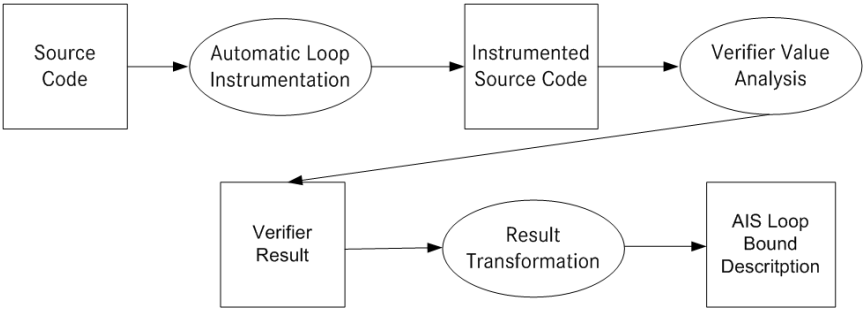


Figure 8.10: Automatic loop-bound estimation using the abstract interpretation on source code level.

Our case study using aiT has shown that the highest manual effort is needed for the loop-bound detection and annotation. These efforts have to be repeated on each new version of the software in order to ensure that the loop-bound values are still correct. As high manual effort is typically linked to potential human errors, complex loop conditions might not be evaluated safely.
In order to achieve industrial acceptance of timing verification and increase the soundness of its results, an improved automatic loop-bound detection is required. We propose a method based on a static source code analysis using abstract interpretation that has been implemented and tested on our example application. As depicted in Figure 8.10, we analyse the source

code with our own parser to instrument the loops automatically with a counter. Performing this instrumentation leads to the following changes to the existing loops:

```
counterL1 = 0;//add a unique counter variable
for(...;...;...)
{
   ...//existing loop body
   ++counterL1;//increment the counter on each iteration
}
```

The code analysis using *PolySpace Verifier* (cf. [The07]) as an abstract interpretation engine provided value ranges for the newly introduced loop counter variables. Thus, the upper bound of the counter variable as identified by that analysis engine equals the loop-bound. A transformation into equivalent AIS annotations completes the automated tool chain for the loop-bound detection. Using our prototype implementation, we were able to find 47 out of 51 loop-bounds that had not been found by aiT which is an increase in precision of 92 percent.
The precision of our loop-bound detection method now depends on the precision of the abstract interpretation engine for a given implementation and tool setup.

8.5 Variant-Aware WCET Analysis

In the previous sections, we considered the impact of parameters on the WCET verification in general and solutions for the integration with our approach. In this section we go further by incorporating external explicit variant constraint definitions into the WCET estimation.

8.5.1 Definition

Highly variant systems are characterized by an amount of variants that render the validation and verification of each single variant instance impossible. Hence, sophisticated methods for testing and analysis have to be applied to the entire ECU which includes all possible software variants.
The implementation of the software variants ranges from the use of EEPROM (Electrically Erasable Programmable Read-Only Memory) parameters to self-parameterization by detection of the hardware environment (similar to

plug-and-play).
When applying static WCET analyses to highly variant systems, conservative assumptions can lead to significant overestimations [MGL06b].

Variants can be seen as system level constraints. For example, engines using a turbo charger require different handling of airflow and pressure control than engines without turbo chargers. However, due to system level constraints the combination of different airflow and pressure control functionalities is not possible. This mutual exclusion (cf. Figure 8.11 b)) is often not taken into account by the WCET analysis. This is mainly due to the fact that system level constraints are set by external variables that are not visible to the analysis. A further reason for WCET overestimation arises from the necessity of context approximations by the WCET analysis. As it is practically impossible to trace each and every reachable program context, system-wide dependencies are prone to be neglected by analysis optimizations. Here, we will present a variant-aware timing analysis, that

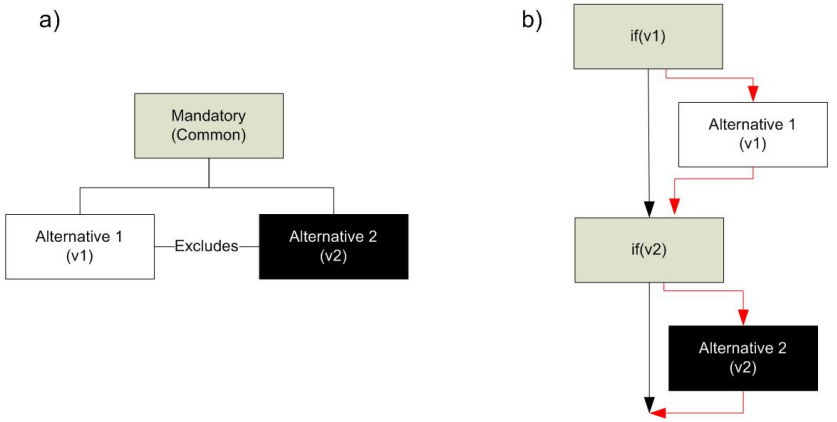


Figure 8.11: a) Variant constraint model b) Variant algorithm with infeasible WCET path.

derives a) a reduced but sound WCET bound and b) the corresponding worst-case variant that leads to this bound. The analysis consists of three steps. We first determine the variant dependent control-flow structures (loops and conditionals) and the type of dependency. Then we derive a

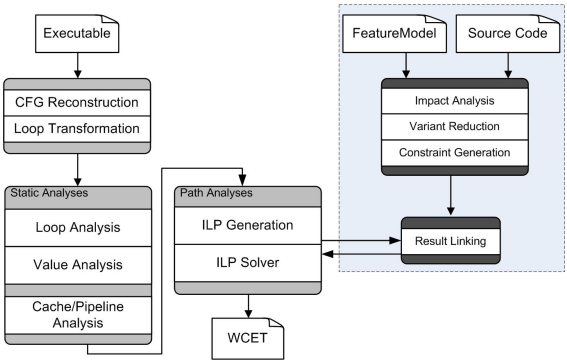


Figure 8.12: The original aiT toolchain and the approach's extensions (in the dashed box).

set of candidates for the worst-case variant. In the last step, we extend the computation of the worst-case execution path from the original timing analysis to derive the WCET bound and the worst-case variant at once.

In the next section we will present the original timing analysis on which we base our approach. The formal setting of the variant-aware timing analysis is given in Section 8.5.3 and Section 8.5.4 presents the analysis in detail. Evaluations are then given in Sections 8.5.5 to 8.5.7.

8.5.2 The aiT Framework

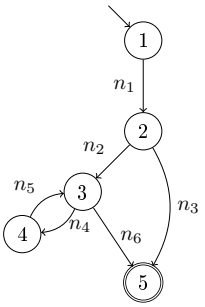
We build our variant-aware timing analysis on top of the *aiT-Framework* as depicted in Figure 8.12. It consists of a set of different tools that can be subdivided into three main parts: *CFG Reconstruction*, *Static Analyses* and *Path Analyses*.

The *CFG reconstruction* builds the control-flow graph (CFG), the internal representation, out of the binary executable [WGR+09]. This CFG consists of the so-called *basic blocks*. A basic block is a sequence of instructions such that the basic block is always entered at the first and left at the last instruction. To make sophisticated interprocedural analysis techniques applicable, loop structures have to be transformed into tail-recursive routines. Additionally, user annotations, such as upper bounds on the number of loop iterations which the analysis cannot automatically derive, are processed during this step.

The static analysis part consists of three different analyses: *loop analysis*, *value analysis*, and a combined *cache and pipeline analysis*. The *value analysis* determines the effective addresses of memory accesses and also supports the loop analysis to find upper bounds on the number of loop iterations [MAWF98]. For this purpose, the analysis derives intervals for all variables at each program point.

The *loop analysis* collects invariants for all potential loop counters. This means it computes for all the variables changed within a loop, how much they change during one iteration. Then it evaluates the loop exits, requests start and end values for these potential loop counters from the value analysis and thus derives upper bounds on the number of loop iterations.

The *cache and pipeline analysis* performs the so-called *low-level analysis*. It simulates the processor's behavior in an abstract fashion to determine an upper bound on the execution time of each basic block [FMWA99, LTH02].



$$n_1 = 1;$$
$$n_1 = n_2 + n_3;$$
$$n_2 + n_5 = n_4 + n_6;$$
$$n_4 = n_5;$$
$$n_4 <= b_{loop} n_2;$$
$$n_3 + n_6 = 1;$$

$$\max : \sum_i \left(\sum_{\forall j : n_j \ enters \ B_i} c_i n_j \right)$$

Figure 8.13: Control-flow graph and the corresponding flow constraints.

The *path analysis* combines the timing information of each basic block and all loop-bounds, then searches for the longest path within the executable. In this fashion, it computes an upper bound on a task's execution time. Searching for the longest path is done by using a technique called *implicit path enumeration (IPET)* [LM95]: the control flow graph and the loop-bounds are transformed into *flow constraints*. The upper bounds for the execution times of the basic blocks as computed in the cache and pipeline analysis are used as weights. Figure 8.13 provides an example. The variables n_i, also called traversal counts, denote how often a specific

edge is traversed. The first and the last basic block are left, respectively entered, exactly once ($n_1 = 1$ and $n_3 + n_6 = 1$). For all other basic blocks, the sum of the traversal counts entering equals the sum leaving. The loop body (basic block 4) is executed at most b_{loop} times as often as the loop is entered ($n_4 <= b_{loop} n_2$). The constant c_j denotes the cost of the basic block j. The maximum sum over the costs of a basic block times the traversal counts entering it determines the final WCET bound.

8.5.3 Variant Constraint Specification

A common language for the description of variants is the feature model [CE00]. This modeling language describes abstract features of a product and the feature relationships as a tree. The abstract features are either mandatory (necessary in all products), optional (optionally available) or alternative (exactly one of a given set). These feature types form a basic constraint language that can be enhanced by more elaborate constraints.

A feature is anything that a customer can experience as a functional entity. Hence, an engine is a mandatory feature, whereas the engine type is an alternative decision. A sunroof is usually an optional feature, that can only be chosen if the car is not a convertible. Obviously, there are lots of similar constraints in the software that controls all these functionalities. While feature models are usually seen as marketing decision models, fur-

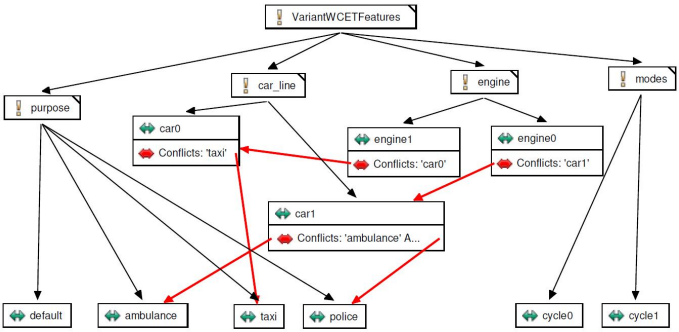


Figure 8.14: An example feature model.

ther use-cases beyond customer features can be imagined. We see them in software internal constraint specifications like operating mode relationships or task internal time-slicing dependencies. The decision space of a feature

model in current real-life systems may easily reach up to several million valid combinations. We now provide a formal definition of a variant.

Definition 3 (Variant)**.** *Given a set of features $\mathbb{F}$, a variant is a mapping from features to boolean values, depending on whether or not the feature is selected:*

$$V : \mathbb{F} \to \mathbb{B}$$

where $\mathbb{B}$ is the set of boolean values true and false. If a variant obeys all constraints $\mathbb{C}$ determined by the variant model, the variant is said to be valid. The set of all variants is given by $\mathbb{V}$ and the set of all valid variants by $\mathbb{V}_v \subseteq \mathbb{V}$.

Note that we do not need to consider mandatory, optional or alternative features separately as such requirements can be considered to be modeled by constraints contained in $\mathbb{C}$. For the sake of simplicity, we represent a variant V as a set of features that evaluate to true: $f \in V \Leftrightarrow V(f) = T$ and $f \notin V \Leftrightarrow V(f) = F$. Given the example in Figure 8.14, we have the following set of features $\mathbb{F}$ and set of constraints $\mathbb{C}$:

$$\begin{aligned}\mathbb{F} = \{&car0, car1, engine0, engine1, cycle0, cycle1,\\ &default, ambulance, taxi, police\}\end{aligned}$$

$$\begin{aligned}\mathbb{C} = \{&(car0 \otimes car1), (engine0 \otimes engine1), (cycle0 \otimes cycle1),\\ &\neg(car0 \wedge engine1), \neg(car1 \wedge engine0), \neg(car0 \wedge taxi),\\ &\neg(car1 \wedge police), \neg(car1 \wedge ambulance)\\ &(\exists! x \in \{default, taxi, police, ambulance\} : x)\}.\end{aligned}$$

For instance, $V = \{car1, engine1, taxi, cycle0\}$ is a valid variant. Note that within this small example there are already 2^{10} variants from which only 10 are valid.

Timing analysis may derive a variant-depending timing bound. Given a variant V, the specific information about all identifiers can be encoded to be considered by the timing analysis. The resulting time bound is then only valid for this variant V. In such a case, we write WCET(V). If no information about a specific variant or a set of variants is taken into account, we write $\widehat{\text{WCET}}$ for the variant independent time bound. Note

that $\forall V \in \mathbb{V} : \widehat{\text{WCET}} \geq \text{WCET}(V)$. The problem of variant-aware timing analysis can then be seen as follows

$$\text{Find } \hat{V} \in \mathbb{V}_v \text{ s.t. } \forall V' \in \mathbb{V}_v : \text{WCET}(\hat{V}) \geq \text{WCET}(V')$$

This variant $\hat{V}$ is the so-called worst-case variant and provides a precise global bound for the analysed software. Determining this variant by exhaustively deriving all $\text{WCET}(V)$ is computationally infeasible due to the high number of different variants and the complexity of the timing analysis. Hence, we first reduce the search space and then aim for an approximate solution.

8.5.4 Variant-Aware WCET Analysis Approach

Our approach for a variant-aware static WCET analysis (cf. the dashed box in Figure 8.12) consists of the following steps:

- Impact Analysis: Identify code to feature dependencies that deliver major impacts on the WCET.
- Variant Reduction: Take the identified features and search the feature model for further dependencies (feature to feature dependencies). All independent features can be removed.
- Constraint Generation/Result Linking: The remaining feature dependencies (i.e. variants) are transformed into ILP control-flow constraints. Hence, a variant-enriched ILP can be generated which is then used to estimate $WCET(\hat{V})$.

Impact Analysis

Two different control-flow structures may influence the timing behavior depending on the chosen variant: loops and conditionals. In case of conditionals, the taken branch may depend on whether or not a feature is selected. In case of loops, the loop iteration bound may be determined by the feature selection. To analyse these dependencies, we first need to identify the feature-dependent control-flow structures and in a second step analyse how these structures are influenced. The set of identified conditionals is denoted as $\mathbb{CO}$ and the set of loops as $\mathbb{LO}$. Note that for the sake of simplicity, we restrict the description of the approach to *for*- and *while*-loops as well as to simple *if*-statements. Other statements can be

seen as extensions to our basic structures. For example, switch-cases can be seen as combinations of multiple if-else structures.

We represent the variant-dependencies in the following way:

$$Cond : \mathbb{CO} \rightarrow (\mathbb{F} \rightharpoonup \mathbb{B})$$

A conditional c is assigned a partial mapping of features to boolean values. $(Cond(c))(f)$ evaluates to boolean value b, iff conditional c always evaluates to b in case feature f is selected.

$$Loop : \mathbb{LO} \rightarrow (\mathbb{F} \rightharpoonup \mathbb{N})$$

A loop l is assigned a partial mapping of features to integers. $(Loop(l))(f) = n$ means that loop l is bounded by n if feature f is selected.

More complex dependencies between features and conditionals are possible. Consider a conditional that evaluates to true, iff two out of three features are selected. Such cases do not fit into this notion. The domain of the partial mapping $(\mathbb{F} \rightharpoonup \mathbb{B})$ for such a conditional would be empty.

Listing 8.1: Variant Depending Code Fragment

```
    ...
11: const char clutch[2] =
12: /*   car0    car1 */
13:     {  10,    5 }; /*clutch line*/
    ... [Block1] ...
37:   for(i=0;
38:         i<clutch[c_car_type];
39:         ++i)
40: {
41:     if(c_purpose==TAXI){...[Block2]...}
    ...[Block3]...
94: }
    ...[Block4] ...
```

As a short example, let us consider the feature model in Figure 8.14 and the code snippet from Listing 8.1. In the example (stripped down from existing code) we notice that the dependency between the feature model and code variables is not always given directly. Hence, these relationships have to be established via naming rules or manual mapping. In the above

case, for example, the prefix 'c' indicates a variant configuration parameter (*c_engine*, *c_car_type* and *c_purpose*). There are several possible ways to detect the basic structures in the code. We decided for pattern matching as the following step of constraint generation does not rely on a complete set of variant dependencies. Undetected dependencies on loops and conditionals will lead to missing constraints and thus to an overapproximation of the result (but never to an underapproximation). The impact analysis first finds the loop in line $L37$ and the conditional in line $L41$ and then evaluates both expressions: $Loop(L37) = \{(car0, 10), (car1, 5)\}$ and $Cond(L41) = \{(taxi, true)\}$.

Variant Reduction

Given the functions $Cond$ and $Loop$, we can perform the variant reduction in order to reduce the search space for our worst case variant.

Removing Timing-Irrelevant Features We partition the set of features $\mathbb{F}$ into timing-relevant $\mathbb{F}_r$ and timing-irrelevant $\mathbb{F}_{\neg r}$. A feature f is timing-relevant if there is at least one loop l or conditional c such that $Cond(c)(f)$ or $Loop(l)(f)$ is defined. In our example, $\mathbb{F}_r = \{car0, car1, taxi\}$. Note that we also remove all constraints from the set $\mathbb{C}$ that contain at least one timing-irrelevant feature: $\mathbb{C}_r = \{(car0 \otimes car1), \neg(car0 \wedge taxi)\}$. Since the set of constraints only limits the possible variants, removing constraints may only lead to an overapproximation and is thus sound.

Definition 4 (Reduced Variant)**.** *A variant is called reduced, iff it only defines values for timing-relevant features $\mathbb{F}_r$:*

$$V_R : \mathbb{F}_r \to \mathbb{B}$$

The set of all reduced variants is denoted as $\mathbb{V}_r$.

The set of reduced variants for our example is

$$\mathbb{V}_r = \{\{\}, \{car0\}, \{car1\}, \{taxi\}, \{car0, car1\}, \\ \{car0, taxi\}, \{car1, taxi\}, \{car0, car1, taxi\}\}.$$

The set $\mathbb{V}_{r,v} = \big\{\{car0\}, \{car1\}, \{car1, taxi\}\big\}$ gives all valid reduced variants. All other variants do not comply with $\mathbb{C}_r$.

Dominating Features In order to reduce the set of possible worst-case variants even further, we need to identify dominant features with respect to timing.

Definition 5 (Feature Domination). *A feature f is said to dominate feature f' ($f \sqsupseteq f'$), iff*

$$\forall c \in \mathbb{CO} : (Cond(c))(f) = (Cond(c))(f')$$

and

$$\forall l \in \mathbb{LO} : (Loop(l))(f) \geq (Loop(l))(f')$$

In our example, $engine0 \sqsupseteq engine1$ since the loop-bound of the loop in line 38 is higher in case $engine0$ is selected (while all other loops and conditionals remain unchanged).

Using the feature domination, we can safely exclude some variants from the search space for the worst-case variant. Hence, we lift the domination relation to variants.

Definition 6 (Variant Domination). *A variant V is said to dominate variant V' with $V \neq V'$, iff*

$$\forall f' \in V' : f' \in V \vee (\exists f \in V : f \sqsupseteq f')$$

In such a case, we write $V \sqsupseteq V'$

Assuming that a variant V is dominated by another variant V', we know that if V leads to the highest WCET-bound then also V' does. In our example variant $\{car0\}$ dominates $\{car1\}$, but not $\{car1, taxi\}$.

Variant Search Space The variant search space $S \subseteq \mathbb{V}_{r,v}$ is a subset of the set of all reduced variants. In addition, we can exclude all dominated variants from the search space.

$$\forall V \in S : \nexists V' \in S : V' \neq V \wedge V' \sqsupseteq V$$

For the given example, the resulting search space is given by

$$S = \big\{\{car0\}\{car1, taxi\}\big\}.$$

Constraint Generation / Result Linking

In the last step, we extend the IPET model to derive the worst-case variant.

In order to derive the worst-case path, we introduce within the ILP one variable V_i for each remaining variant from the search space S and one variable f_j for each timing relevant feature from the set $\mathbb{F}_r$. Constraint (8.3) ensures that at most one variant is selected:

$$\sum_i V_i = 1 \tag{8.3}$$

The set of constraints (8.4) links the remaining variants (from the search space S) to their active features:

$$\forall f_j : f_j = \sum_{V_i \in V^f} V_i \tag{8.4}$$

where V^f is the set of variables representing variants where feature f is selected. We now need to link the remaining variants and features to the IPET model. For a feature-dependent loop $l \in \mathbb{LO}$, we add a constraint which bounds loop l by $(Loop(l))(f)$, in case feature f is selected:

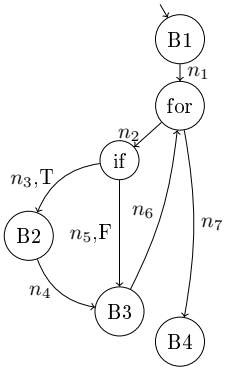
$$\forall l \in \mathbb{LO} : \forall f \in Dom(Loop(l)) : \\ n_{loop-body(l)} \leq (Loop(l))(f) \cdot n_{loop-entry(l)} + (1 - f) \cdot C_{big} \tag{8.5}$$

Constant $C_{big} \in \mathbb{N}$ is a big integer used to 'deactivate' constraint (8.5) if the corresponding feature is not selected. Given that C_{big} is chosen big enough, the constraint does not influence the final result. In the same manner, we add a constraint for each feature-dependent conditional $c \in \mathbb{CO}$. Let $(Cond(c))(f)$ evaluate to b. If feature f is active, traversal count for the $\neg b$ edge (denoted as $n_{\neg b}$) is set to 0.

$$\forall c \in \mathbb{CO} : \forall f \in Dom(Cond(l)) : \\ n_{\neg(Cond(c))(f)} \leq (1 - f) \cdot C_{big} \tag{8.6}$$

Again $C_{big} \in \mathbb{N}$ is used to 'deactivate' the constraint, in case f is not selected.

Figure 8.15 depicts the control-flow graph and the corresponding constraints for the code fragment from Listing 8.1.



$$
\begin{aligned}
n_1 &= 1;\\
n_1 + n_6 &= n_2 + n_7;\\
n_2 &= n_3 + n_5;\\
n_3 &= n_4;\\
n_6 &= n_4 + n_5;
\end{aligned}
$$

$$
\begin{aligned}
v_1 + v_2 &= 1\\
f_{car_0} &= v_1\\
f_{car_1} &= v_2\\
f_{taxi} &= v_2
\end{aligned}
$$

$$
\begin{aligned}
n_5 &\leq f_{taxi} \cdot C_{big}\\
n_2 &\leq 10 \cdot n_1 + (1 - f_{car0})C_{big}\\
n_2 &\leq 5 \cdot n_1 + (1 - f_{car1})C_{big}
\end{aligned}
$$

$$\max : \sum_i \left(\sum_{\forall j : n_j \ enters \ B_i} c_i n_j \right)$$

Figure 8.15: Control-flow graph and the corresponding flow constraints.

The solution to the ILP determines the worst-case variant $\hat{v}$ and an upper bound on the execution time valid for this variant $WCET(\hat{v})$.

Correctness

Each static WCET analysis can only estimate the actual worst-case execution time. It is, however, crucial that the actual execution time is never underestimated. The constraints introduced in Section 8.5.4 model restrictions on the control-flow graph and thus reduce the WCET bound. We now need to argue that this reduction is sound, i.e. that the WCET bound is still a safe approximation.

The allowed combinations of features (and thus of loop-bounds and conditionals) depend on the set of variants $\mathbb{V}_{r,v}$. Although the set of reduced and valid variants is much smaller than the set of variants $\mathbb{V}$, $\mathbb{V}_{r,v}$ represents an overapproximation of the $\mathbb{V}_v$ with respect to timing. By construction, concentrating only on timing-relevant features does not influence the WCET analysis. In step three, we also reduce the set of constraints $\mathbb{C}_r$. This, however, may only lead to variants falsely considered valid (less

constraints lead to more valid combinations). Hence, we approximate the set of variants considered in the final ILP on the safe side only.

The constraints link loops $\mathbb{LO}$ and conditionals $\mathbb{CO}$ to features $\mathbb{F}_r$, which again are linked to reduced and valid variants $\mathbb{V}_{r,v}$. Both sets $\mathbb{LO}$ and $\mathbb{CO}$ only contain control-flow structures for which it was possible to determine a feature dependency. Loops and conditionals for which we were unable to determine a dependency remain unchanged and are not influenced by our newly generated constraints. Again, we approximate only on the safe side and may only deliver an overapproximation, but no underapproximation.

Note that the derived worst-case variant $\hat{V}$ is only valid with respect to the timing analysis. It may happen that the actual worst-case execution time occurs for variant $V \neq \hat{V}$. However, the bounds computed by our analysis are still valid, i.e., $WCET(\hat{V}) \geq WCET(V)$ holds for all variants V, where $WCET(V)$ denotes the execution time bound and not the actual worst-case execution time.

8.5.5 Evaluation Project 1 - Automatic Gear Shifting (AG)

The introduced variant-aware WCET analysis was applied to several functions of the Daimler Trucks automatic gear shifting system (AG). The existing AG feature model was previously used for documentation- and test-case-generation purposes. An existing feature model should be the typical use case for the introduced approach so that no further effort is necessary for the description of the variant constraints. The used feature modeling tool is called *pure::variants* [PS17] which is currently being rolled out in several Daimler projects.

The entire feature model consists of 72 features that lead to an estimated amount of 6.4 million valid variants, according to the modeling tool. A stepwise analysis of all single valid variants is thus not reasonable, which makes the AG a more than appropriate use case for our approach. For comparability reasons, we apply the variant-aware WCET analysis to three separate AG functions (i.e. *ecomain*, *preparation* and *target_range*).

Impact Analysis

The input for the impact analysis is a simple mapping between features and variable names (in the C-code). In our case, this manual work is rather simple as the variant-related code structures can be found in central pa-

rameterization header files where similar names are used for features and variables. This way, 60 out of 72 features are mapped to one or (less often) multiple variables. The remaining 12 features are either structural model elements or features without direct code impact. The automatic impact analysis applied to the entire software identifies 14 out of 122 loop counters that are variant-dependent (11 percent) while 499 of 3658 conditions are variant-dependent (14 percent). Hence, if we assume a similar distribution for the other control-flow statements (i.e. do-while, switch etc.), approximately every 10^{th} control-flow decision depends on constant system constraints.

Variant Reduction

Our implementation of the variant reduction removes all features that are directly or indirectly irrelevant to the analysed source code. Applying the reduction introduced in Section 8.5.4, only 7 features are directly relevant in the case of the function *preparation* and 10 features in the cases of *ecomain* and *target_range*. Additionally, we implemented a dependency analysis that checks if two features exhibit a transitive dependency. Assuming that, for example, only the features *amount of forward gears* and *amount of backward gears* are used in the code, the additional feature *gear box* is required to identify the constraints of their transitive relationship.

The analysis reveals that 55 features are required to describe the dependencies among the 7 original features in case of the function *preparation*. Respectively, 58 features are required for *ecomain* and *target_range*. We see that between 19% and 26% of the features of our feature model can be discarded from the analysis which might improve performance but may also result in a decrease of analysis precision. Note, that a loss of precision only implies a higher amount of remaining overestimation while we always keep sound results.

Constraint Generation and Solution

The feature model, the dependent loops and conditions are then transformed into single ILPs per analysed function. The size of these ILPs ranges from 43 (*preparation*) to 61 (*target_range*) constraints. Before these ILPs can be solved, they are linked to the intermediate aiT result ILP.

Analyzing the function *preparation* without variant constraints reveals a WCET of 201 μs. Adding the variant constraints leads to a WCET of 193 μs, a reduction of almost 4 percent. In addition, the ILP solution also provides a worst-case variant which is the so-called *Unimog truck* using a particular engine and gear box. A portion of the 4 percent reduction can be traced back to the fact that many loops iterate over the amount of forward and backward gears. The gearboxes are available with up to 16 forward and 8 backward gears. However, a gearbox that exhibits both features at the same time is not supported by the software. Hence, a gearbox with 16 forward and 4 backward gears was automatically identified as the worst-case.

The function *target_range* was analysed in a similar way. A variant-unaware WCET of 258 μs can be seen as highly overestimated when compared to the 129 μs variant-aware WCET. This is a total of 50 percent. Furthermore, the result analysis reveals a different 'worst-case truck' than before. The function's loops still iterate on the amount of gears, but the particular truck (with a maximum of 6 forward gears) requires special linearization functions to be called within each loop iteration which leads to the found worst-case.

Finally, also *ecomain* revealed an improvement of more than 3 percent. The main reason here can be seen in the extensive use of mode variables that trigger different functions in four separate execution cycles. The result allows us to identify the worst-case cycle and thus, to improve the distribution of functions within these cycles to gain a real WCET benefit.

Generally, we can see a very diverse range of potential reduction (3 to 50 precent) in the overestimation of WCET caused by software variability. An explanation for this result is the range of variant impact on the source code and thus the WCET itself. With the three use cases, our approach has proven to safely calculate reduced WCET estimates and to additionally deliver worst-case variants that were previously unknown.

8.5.6 Evaluation Project 2 - Model-Based Automatic Gear Shifting (MAGIC)

A second project was chosen in order to verify the previous findings and to evaluate the potential of the variant-aware timing analysis on model-based developments. Again, a feature model already existed for the project.

Feature Model Analysis

A first analysis of the existing feature model for the MAGIC project revealed that it is incomplete in several dimensions. Due to its current use as a generator for a small set of fixed configuration variants, the model contains only a basic set of features (i.e. gear box and engine types) but no implications that are relevant on source code level such as the amount of supported gears. Furthermore, the constraints are very basic as well, as they describe simple relationships, for example, different engines cannot be present at the same time. Dependencies between engines and gearboxes are not available as in example project 1.
The given feature model can be seen as an overapproximation of the actual variant space. Hence, the chances of reducing variant related WCET overestimation are reduced by this model. Fortunately, evaluation projects 1 and 2 target the same truck model range with the same gear boxes and engine types. Hence, we were able to combine the two models in order to retrieve actual feature dependencies.
Nevertheless, there may be constraints and dependencies present, that we do not see due to the limitation of the feature model.

Variant Impact

The computed impact based on the two feature models is limited to the number of available forward and backward gears as gear boxes or engines are never explicitly referenced in the code. But as the majority of the loops in the code iterate on the amount of gears and as we expect loops to have a major impact on the variant-dependent WCET overestimation, we assume that the detected impact is sufficient enough for our evaluation.
To our surprise, all calculating functions were designed to be independent of the actual amount of gears. I.e. each loop that iterates on gears iterated on the maximum amount of possible gears, which again is 16.
Even for purely backward-gear loops (maximum amount of 4), 16 iterations are performed. Hence, 75 % of such loops' actual WCET is wasted by design.
The variant-aware WCET analysis only achieves improvements if the worst-case variant's WCET is actually smaller than the variant-independent WCET: $\forall V \in \mathbb{V} : \widehat{\text{WCET}} > \text{WCET}(V)$. With the example at hand, this does not seem to be the case. Every variant has the same set of feasi-

ble paths that are partially executed on empty data.
For our analysis, we added variant constraints in the ILP that simulate an optimised code. The results of this analysis can be seen in Table 8.2.
Evaluation project 2 shows that the variant-aware WCET analysis cannot only reduce static WCET verification's overestimation, but also help to identify low performance in the code, in the case that variant impacted loops or conditions do not reflect this variance.

Function	Impacted Loops	Original WCET	Optimised WCET	Optimisation [%]
AutomaticGearEstimation	3	2,42	2,41	0,04
LimitationEstimation	4	0,42	0,31	26,45
TargetConditions	2	2,21	2,21	0,00
SignalPreprocessing	2	0,55	0,54	1,42

Table 8.2: Optimisations by variant-aware design.

8.5.7 Evaluation Project 3 - Stop Start

With the Stop Start project a third project has been evaluated with the variant-aware WCET analysis approach. The project already uses feature models for the project's parameterisation in order to adapt it to different car models and powertrains. Static WCET analysis however, is not applied in the project so far. Hence, we picked two main Stop Start functions for a dedicated WCET analysis, the *Starter Coordinator* and the *Automatic Start*.

Feature Model Analysis

Compared to evaluation project 2, the Stop Start feature model is very detailed. Furthermore, it explicitly describes the links between features and code parameters with the so called family model, allowing the complete variant-aware WCET analysis approach to be automated.

Impact Analysis

The impact analysis revealed that the two analysed functions bear distinct sets of variant-relevant parameters as referenced by the family model. Out

of the set of referenced variables, almost one third from the Starter Coordinator are WCET-relevant whereas only five percent from Automatic Start are WCET-relevant. Again, we can see a diverse impact of software variants onto different functions.

Constraint Generation and Solution

When formulating the variant constraints using AIS annotations we found an issue with the timing verification. After defining constraints for particular variant configurations we found that the resulting WCET was sometimes higher than the WCET of the unconstrained software, which is counterintuitive.
We identified the root cause for this issue in the variant parameters of the code that are defined as constants with default values (e.g. const int a = 0) instead of volatile constants.
As most users will not realise that these variables need to be specified as volatile, either in the code or at least in AIS annotations, aiT will calculate the WCET for the particular code-given default variant, which in our case is not even a valid variant configuration and leads to an underestimated WCET.
Now, in order to prevent this issue, aiT could raise a warning as soon as it detects a particular amount of infeasible paths due to constants. But as this is not a guarantee to identify all such issues, the software development process should enforce the definition of variant parameters in the code as volatile. Such rules are best enforced and verified by integrated timing verification processes.
Applying the variant-aware WCET analysis to the two analysed functions leads to a reduction of 2.3 percent for the Automatic Start (from 12.49μs to 12.2μs) and 4.8 percent for the Starter Coordinator (from 37.26μs to 35.48μs).
Summarising the results of all three evaluation projects, we can claim that the variant-aware WCET analysis approach helps to safely reduce WCETs in variant-rich software. This is done by reducing the previously unknown overestimation (projects 1 and 3) or by reducing the actual WCET in the design phase of the software (project 2).
The reduced overestimation is typically in the range of 2 to 5 percent which is seen as relevant for the safe estimation of WCETs. Similarly relevant

is the fact that potential underestimations were identified and fixed using our approach.

8.6 Static WCRT Analysis

8.6.1 Definition

The Worst-Case Response Time (WCRT) or Schedulability Analysis (cf. Chapter 5.3) is a wide ranged field of analyses as it covers every timing behaviour of resource allocating processes that are expected to respond in a given time. For the TOAD approach, such entities can be components, tasks, communication messages or complete systems.
Hence, the response time analysis can be further categorised following these concepts.

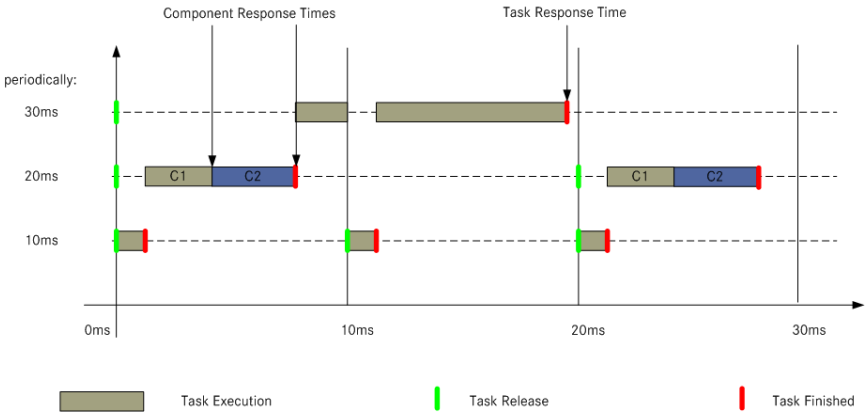


Figure 8.16: The component and task response times in a TOAD application

Task Response Time (TRT)

As depicted in Figure 8.16, the task response time denotes the duration between a task release and a task finish. Thus, preemptions of other tasks

and interrupts are taken into account.
As mentioned in Section 8.4, the task-switching overhead due to preemptions is also taken into account by the TRT, but its safe determination is potentially difficult due to cache-related preemption delays.

Component Response Time (CRT)

The name "component response time" is used within this thesis to describe the duration between a component's release within a task and the component's termination time. The TOAD specific concept of CRTs will later be used to reduce the overestimation of message response times (cf. Section 8.6.3) and end-to-end times (cf. Section 8.8).

Message Response Time (MRT)

The message response time denotes the duration between the queuing of a message and the end of the message transmission.

8.6.2 TOAD Task Scheduling and Scheduling Analysis

Although the scheduling concept of TOAD is configurable, the basic scheduling is the static, preemptive, fixed priority scheduling (cf. [Tin94]). The research field of task scheduling is already very mature and delivers methods for optimal priority assignments ([LSD89, Tin00], schedulability analysis[3] and response time analysis. The following basic form of response time formula can be seen as a de facto standard:

$$R_i = C_i + I_i \tag{8.7}$$

where R_i is the response time of task i, C_i the WCET of task i and I_i the preemption time due to other tasks and ISRs. The preemption time is computed as follows:

$$I_i = \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{8.8}$$

where the function $hp(i)$ returns the set of priorities that are higher than i.

[3] Decides whether a task set is schedulable at all.

The formula for the preemption time is derived from the fact that within the response time (R_i) the higher prioritised tasks j can preempt the execution $\left\lceil \frac{R_i}{T_j} \right\rceil$ times with a WCET of C_j, where T_j is the period of task j. However, due to the shared use of resources in TOAD, the priority inheritance protocol (cf. [LRL90]) is used in order to prevent priority inversion (cf. [Ree98]). This protocol adds an additional blocking time due to the resource usage of lower prioritised tasks as follows:

$$R_i = B_i + C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{8.9}$$

where B_i is the resource blocking time, given by:

$$B_i = \max_{\forall k \in lp(i), \forall s \in locks_{k,i}} (t_{k,s}) \tag{8.10}$$

As only one resource lock can block the execution of the analysed task at a given time, the maximum blocking time ($t_{k,s}$) of all possible resource locks represents the Worst-Case. The possible blocking time due to locks can only originate from lower prioritised tasks (k) that have a higher ceiling priority than the priority of the analysed task (these locks are returned by $locks_{k,i}$).
Using these formulas, the typical TOAD applications can be analysed given the necessary information from the TOAD repository. In our example system, not all information is available. As it is quite often the case, the minimum task inter-arrival times for interrupt service routines are not known and have to be either assumed (which is unsafe) or artificially controlled by the RTE using polling (cf. Section 7.3.2).

Component Response Time Analysis

TOAD components are executed in sequential order within the tasks. Thus, the CRTs depend on the components' position in this sequence.
To calculate the CRT, the TRT analysis can be used with the sum of the WCETs of the component under analysis and all preceding components.

8.6.3 Distributed and Concurrent Systems

As introduced in Section 5.3.5, synchronous (e.g. TDMA) and asynchronous (e.g. CSMA) message passing are the two major scheduling strategies that can be seen in network communication. However, since the analysis of TDMA protocols can be done in a straight forward way (due to the synchronised system times) and CSMA protocols are more broadly used in today's automotive systems, they will be analysed in more detail.

Automotive Distributed Systems

In contrast to common real-time systems, automotive systems are typically distributed. As introduced in Section 2.5, a modern car contains up to 80 ECUs that are highly interdependent (cf. Figure 8.17).
The most commonly used network protocols in the automotive domain are CAN [ISO93], LIN [SR00, LIN07], MOST [MOS06] and increasingly FlexRay [Fle07]. The MOST protocol will not be analysed further as its application domain is in multi-media and thus not in the hard real-time domain.
The LIN protocol is inherently time-deterministic by its master-slave TDMA approach (cf. [RW03]). This results in a schedule that directly reveals the WCRT by its specification. The same applies to the static part of the FlexRay protocol. However, a user-definable dynamic part additionally offers a less time-deterministic event-triggered communication. As FlexRay is assumed to be the future standard bus, its impact on the TOAD approach will be discussed in the following section.
Nonetheless, the currently most often used and rarely time-deterministic CAN protocol can also be used in hard real-time environments. Hence, it's message response time verification will be discussed in Section 8.6.3.

FlexRay Integration

The time-triggered FlexRay communication protocol synchronises the clocks of the communication nodes to allow the application of a static communication schedule. Hence, every message gains a fixed time slot in a system-wide communication period. The fixed communication times make the WCRT analyses redundant, as no interference can delay a communication.

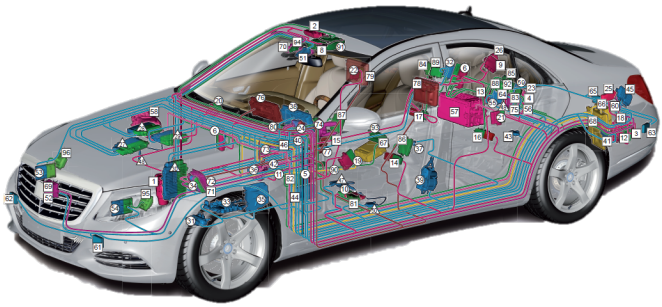


Figure 8.17: Distribution of ECUs in a modern car. Source: Daimler AG

Timing-oriented approaches like TOAD are particularly suited for time-triggered communication protocols, as in such systems all messages have known worst-case release times (corresponding to the component response times). These times are furthermore spread within the message-generating task which is particularly helpful for optimised schedules. A FlexRay schedule can be automatically generated by specialised tools like TTX-Plan by TTTech (cf. [TTT08]) according to given communication requirements. Within our approach, this process can be fully integrated and automated in the development process.
The dynamic part of a FlexRay frame can be used if acyclic messages are to be generated. A standard TOAD software component however, will not generate acyclic messages as asynchronous tasks have no access to data resources[4].

Analysis of CAN Message Response Times

The CAN protocol uses the Carrier Sense Multiple Access paradigm. Thus, an arbitration process is used to decide which message is to be sent. Hence, the following formula estimates the worst-case message response time R_i as presented in [TB94, THW94, TBW95]:

$$R_i = J_i + Q_i + C_i \tag{8.11}$$

[4] This behaviour was explicitly chosen to ensure timing predictability in TOAD applications.

where J_i is the queuing jitter[5] of message i, Q_i is the queuing delay of message i and C_i is message i's transmission time.
While the queuing jitter is a known value and the transmission time can be derived from the bus bit-rate and the message length, the queuing delay Q_i requires the analysis of the arbitration protocol.
As an active message transmission cannot be interrupted, the blocking time B_i equals the longest transmission time of messages with lower priorities ($lp()$).

$$B_i = \max_{k \in lp(m)} (C_k) \tag{8.12}$$

Within the arbitration process, the messages that have higher priorities (messages j) are transmitted first and thus can delay the start of the transmission of message i $\left\lceil \frac{Q_i + J_j + \tau_{bit}}{T_j} \right\rceil$ times. τ_{bit} represents the transmission time of a single bit and is used as a maximum difference in the arbitration start time. Thus the following formula is used to calculate the complete queuing delay for message i:

$$Q_i = B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{Q_i + J_j + \tau_{bit}}{T_j} \right\rceil C_j \tag{8.13}$$

However, since the message itself can delay the transmission of higher prioritised messages the $hp(i)$ term has to be extended to $hep(i)$ which returns the messages of higher and equal priorities (hence the message itself). This flaw in the original formula was published 13 years after the first publication in 1994 (cf. [DBBL07]) and thus shows that even the established analyses do not necessarily lead to a verification if wrong assumptions are made. Hence the formula used in the example TOAD implementation is given as follows:

$$Q_i = B_i + \sum_{\forall j \in hep(i)} \left\lceil \frac{Q_i + J_j + \tau_{bit}}{T_j} \right\rceil C_j \tag{8.14}$$

If an error detection and correction mechanism is used, an additional error term is be added to the formula.
Concluding the analytical MRT calculations, the timing behaviour of CAN messages can be similarly described as the Worst-Case task response time

[5] The queuing jitter represents the maximum variation in queuing time.

calculation. Parameters that can be derived from the TOAD repository have to be filled into the WCRT formula and calculated using a fixed-point iteration. Due to the fact that the formula is non linear, the system timing optimisation cannot use operations research optimisation algorithms such as simplex solvers.
In contrast to typical task scheduling, CAN-message scheduling can result in sets of messages that have finite response times as well as sets of messages that cannot be guaranteed to be delivered at all. If a message of the latter set is part of an end-to-end path, the end-to-end time is also unbounded. This can be a valid result for non-safety or non-real-time systems.

Reducing Message Response Time Pessimism

The calculation of the queuing delay Q_i currently assumes that every message can occur simultaneously. In order to reduce the queuing delay, we can release messages at defined times. With the TOAD approach we know the Worst-Case CRT which is the time, that a result is definitely available for release. So even without a time-triggered operating system, we can optimise message response times by using basic alarms.
In the concept of real-time calculus (cf. Section 5.3.4) we adapt the request function by tightening the message release jitter and spreading the message release times.
Considering a system with just two messages, each having a length of 135

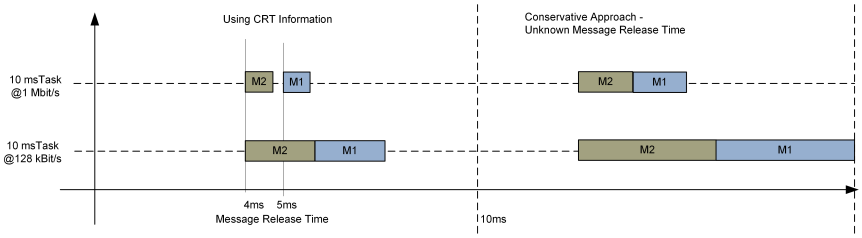


Figure 8.18: CAN messages that cannot interfere with each other are known by the TOAD approach.

bits as an example. When the messages are released in a 10ms task with a Worst-Case CRT of 4ms (ID 2) and 5ms (ID1) the message response times

are on average twice as high if the TOAD knowledge is not applied in the analysis (cf. Figure 8.18 and Table 8.3).

Bitrate	MessageID	TOAD MRT [ms]	Common MRT [ms]
1 MBit	1	0.135	0.27
	2	0.135	0.27
128 kBit	1	1.11	2.11
	2	1.055	2.11

Table 8.3: An example for possible MRT optimisation resulting from the TOAD approach.

The adapted response time algorithm is as follows:

1. Determine the set of independent messages for each message (by using the existing MRT-analysis algorithm).
2. Calculate the response time for each message but exclude the independent messages from the calculation.
3. Refresh the sets of independent messages according to the calculated response times (add new independent messages).
4. Return to step 2 until the sets of independent messages do not change on refresh.

The worst-case result (no improvement) occurs if the algorithm stops with an empty set of independent messages. In that case, the result equals that of the standard algorithm.

8.6.4 Implementation

Some of the algorithms mentioned so far are already implemented in existing tools (e.g. [HHJ+05] and [HGP+06]). But due to the fact that the existing tools are too inflexible for adaptation within our approach, scheduling and response-time analyses were implemented anew.
The current implementation does not calculate specific preemption overheads as the currently supported target (C166) has a constant preemption overhead. Furthermore, the C166 implementation of the TOAD-RTE does not implement polling for ISRs and thus uses measurements as basis for assumptions on minimum ISR inter-arrival times.

8.7 Variant-Aware WCRT Analysis

8.7.1 Definition

As shown in Section 8.5, the awareness of variants in the design, implementation and analysis can improve timing performance and reduce WCET overestimation. As a result of the variant-aware WCET analysis, component and task WCETs are estimated for a specific variant configuration, the so called Worst-Case variant. Using these WCET estimates in the WCRT calculation is safe but bears potential overestimation as well. As with the WCET estimation, the variant configuration cannot change during task execution. If two components' variant-aware WCET calculation results in different Worst-Case variant configurations, we know that only one of them can occur at a time.
Hence, it is necessary to take the variant configurations into account in the task response time analysis as well.

8.7.2 Calculating Variant-Aware Response Times

The difficulty of the Variant-Aware Response Time Analysis Approach is that it cannot be performed independently of the WCET calculation. Consider the following standard response time formula:

$$R_i = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j.$$

In addition to the non-linearity given by the $\lceil\rceil$ operator, C_i and C_j now depend on the variant model. And their value may in turn depend on the amount of task preemptions:

$$m = \left\lceil \frac{R_i}{T_j} \right\rceil.$$

Thus, in order to solve the WCRT equation, the typical fixed-point iteration has to be adapted to work on changing ILPs, where the basic step is given as follows:

$$R_i^{n+1} = C_i^n + \sum_{\forall j \in hp(i)} m_j C_j^n.$$

A safe, but potentially overestimated start value for R_i^0 is the variant unaware WCET of task i (C_i). In contrast to the calculation with constant WCET values, each R_i^{n+1} iteration is calculated as an ILP optimization problem with the goal of maximizing $C_i + \sum_{\forall j \in hp(i)} m_j C_j$ for the n_j derived in the previous step. The ILP calculation now takes into account the weights of all basic blocks of all relevant task-functions including the constraints given in the feature model.

8.7.3 Evaluation

As we have no suitable project with variant-dependent tasks at hand, we chose to simulate such a system by taking the industrial examples from Section 8.5 and placing them virtually in different task contexts. Choosing the task parameters of period and priority leads to different possible values of m. Thus, for a combination of two tasks, each containing a single function, we are able to directly calculate the WCRT for different values of m. The results are depicted in table 8.4. The reduction of the overestimation

m	WCRT	Variant WCRT	Overestimation
1	0,4528	0,45255	0,0552%
2	0,647	0,6465	0,0773%
10	2,2006	2,1981	0,1136%
100	19,6786	19,6536	0,1270%

Table 8.4: Overestimation reduction by variant-aware WCRT analysis.

for our particular example is just in the range of 0.1 %. This shows the basic feasibility of the variant-aware response time analysis but should not be taken as an indicator of potential savings in other projects. The potential savings can be close to 100% on artificial examples. Hence, further real world examples are necessary in order to gain insight into more realistic expectations.
An evaluation of the estimated Worst-Case variant configurations shows further interesting insights in that they indicate a potential reason for the lower than expected additional savings. According to Table 8.5 both tasks bear a high similarity in the chosen worst-vase variant for their WCET. Hence, one would expect the similarities to prevail in the combined worst-case variant for task response times. However, the result is different in the

main aspects like vehicle line, engine and gear-box. Nevertheless, when it comes to the timing results, the chosen configuration is still very close to the original configuration.

Feature	Task A	Task B	WCRT Combination
Line	Unimog	Unimog	Actros
Engine	OM 904	OM 904	OM 501
Gear Box	UG 100 Split	UG 100	G241
# FW Gears	16	8	16
# BW Gears	4	8	4
Application	Basic	Basic	Basic

Table 8.5: Worst-Case configurations estimated by the variant-aware WCET (Task A,B) and combined WCRT calculation.

8.8 End-To-End Timing Analysis

As defined in Section 5.3.6, the end-to-end timing is typically the actual goal of a timing verification. The end-to-end timing represents the Worst-Case duration from sensor inputs to one or multiple actuator outputs.
In TOAD applications sensors and actuators can be defined with the use of special interfaces. A particular write-only interface is used for sensors while a particular read-only interface is used for actuators. Thus, an end-to-end algorithm can automatically identify these components by their type.
The potential Worst-Case end-to-end routes can then be estimated in the longest route from a sensor to an actuator. Cyclic (sub-)routes should be prevented by design, but sometimes cannot be prevented. In these cases, special cycle-bound annotations are required for the end-to-end estimation.
As soon as a route is identified or was manually defined, the analysis of the Worst-Case end-to-end is straightforward: Every response time on the route is summed up together with the synchronisation overhead resulting from the respective synchronisation mechanisms (e.g. [Sun97]).
As received messages and tasks are not synchronised in our implementation, a sent communication message that is read by a task on a different ECU is assumed to arrive shortly after the task has started its execution in the worst case. Hence, the new input can only be processed in the following task cycle.

The EBA example in Figure 8.19 shows two possible routes (indicated by

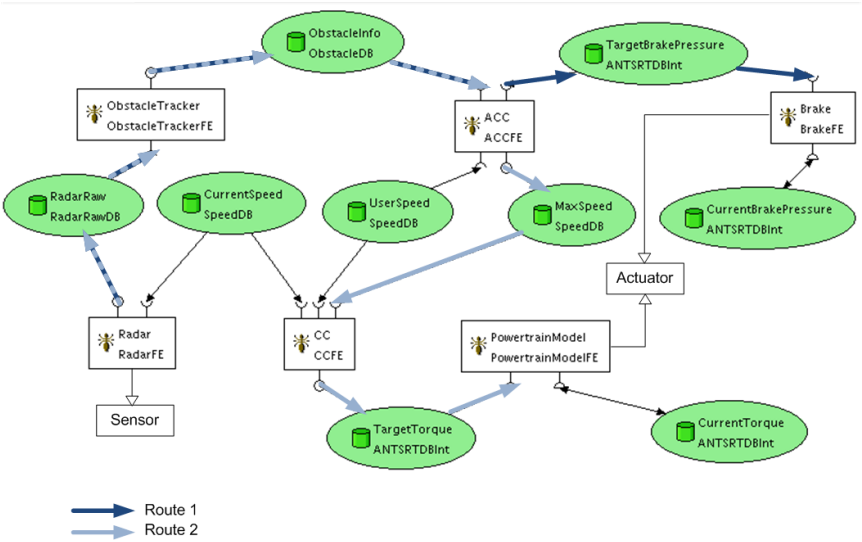


Figure 8.19: Finding end-to-end routes in TOAD applications by definition of sensors and actuators.

dark and light arrows). An obstacle that is sensed by the radar unit results in a torque adaptation using either the powertrain or the brake.

8.9 Optimisation

As shown in the previous sections, it is possible to determine many different timing properties for a given TOAD application.
Due to the fact that the timing properties often depend on parameters that can be changed by the software architect or the developer, several optimisations can be performed. For example:

1. Find the cheapest hardware platform to run the current application according to the timing specification.
2. Find the optimal deployment of an application's components on a given platform, so it can meet the timing specification.

3. Find the maximum amount of input sensors that can be handled by the application on a given platform.

All these optimisation goals can be found by changing the available parameters manually or by transforming the analysis system into an optimisation problem. In contrast to existing timing optimisation approaches (e.g. SymTA/S [HHJ+05]), the TOAD approach is the only approach known to the author that is capable of taking all system parameters into account, down to the software component level.

8.10 Conclusion

This chapter introduced tools and methods for the automation of the TOAD development process and its integrated timing verification on the different TOAD architecture levels. Several novel WCET analysis methods were introduced. The parametric WCET analysis, for example, allows high performance analyses that are especially helpful for automatic optimisations. The variant-aware WCET analysis takes external variant constraints into account and thereby reduces overestimations of current WCET analysis methods by up to 50% of the original WCET.
The automatic WCET analysis was further improved by a loop-bound detection on source code level, that showed a 92% loop-bound detection increase on our example application. The safe estimation of WCETs was widely improved, as manual and error-prone work could be reduced largely.
All response time analysis methods receive their inputs from other analysis scopes within the TOAD repository. Thus, all analyses are automated and integrated within the TOAD tool set so that they can be used transparently by software component developers that do not require specific timing verification knowledge.
We were also able to show that the results on the different scopes can be used to reduce the actual worst-case behaviour. For example, CAN messages are released with regard to their component's worst-case response time. Thus, creating independent messages that improve the overall worst-case message response time.
The final discussion on system optimisations shows the further potential of our integrated analysis approach.

9 The TOAD Development Process (TOAD-P)

9.1 Introduction and Definition

Our integrated approach to timing verification provides means to discover timing-related issues early in the development. But in order to realise such benefits, a development process is required that enforces the use of our approach. Such a process describes the creation, flow and use of timing related information at different development stages, performed by different roles.
The TOAD-P process specifically aims at the distributed automotive software development process (cf. Section 2.4).

The following should be answered by the TOAD process:

- When and by whom are timing properties specified in the context of a distributed development?
- Which forms of verification can be performed at which development stage?
- How are timing verification results exchanged between different development parties?
- If the timing verification fails at some point, what are the possible consequences?

We take the process definition from Section 2.3:

> A development process defines the activities, methods and procedures that are necessary for the development and verification of a (software) system.
>
> Translation from [Bal98].

As a first step, we need to analyse the existing activities, methods and procedures in order to identify the process interfaces for the TOAD approach.

9.2 On Existing Processes

Development processes for embedded real-time systems are not as divergent as, for example, the hardware architectures they use. Commonly, these processes are following the steps of requirements engineering, specification, implementation and testing on different system levels.
The embedded systems development is typically subdivided into system, hardware and software development domains (cf. Section 2.4 with dedicated development teams for each domain [SZ03]).
All these domains define their own architectures in order to structure their technical (sub-)system as well as the work packages required to implement their (sub-)system. Thus, these architectures form the basis for the distributed development. And the interfaces they define form the basis for information exchange between the different development parties.
The TOAD architecture defines concepts on the system, hardware and software level that are used by the respective developers and stored in a common central repository, so that every development party has access to all required information.

9.3 The TOAD-P Process

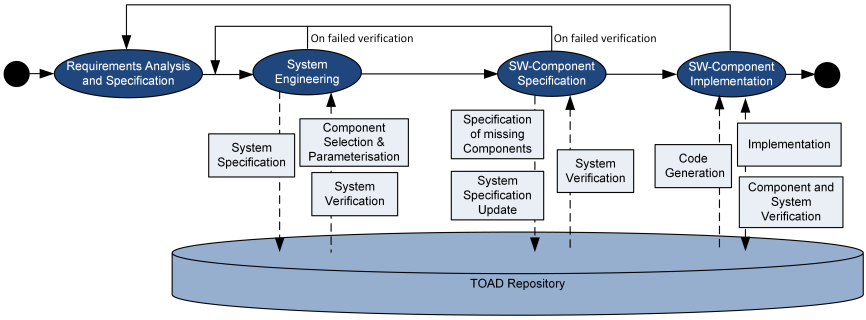


Figure 9.1: The TOAD-P process.

The TOAD-P process (Figure 9.1) is strongly related to the TOAD-A architecture as the formal architecture element definitions are used for the

communication between developers of the different development stages.
All process steps beginning with the system engineering produce work products that are stored in the repository. This way, a system verification can be performed in each step of the development. But only the implementation phase generates verification results (namely the WCET calculations) that are propagated back to the repository.

9.3.1 Requirements Analysis and Specification

Unfortunately, one of the most important steps in the timing verification process cannot be enforced by a tool-chain: The specification of timing requirements can only be enforced by the requirement specification process and the respective review processes.
Generally, each requirement that states that something shall happen on the system level, requires the definition of a time until it shall happen. That sounds easy. But several reviews by the author revealed that even functional safety requirements, that are required to contain fault tolerant time intervals (cf. [ISO11]), do not always do so.
The early definition of an upper limit of the end-to-end response time usually requires elaborative estimations or costly customer-oriented experiments if human reaction time plays a crucial role. However, when it comes to safety, these issues should not prevent the estimation of the required response times.
As soon as the required response times are defined, budgeting is to be applied along the execution paths through the system (which is defined during the System Engineering phase, see Section 9.3.2). Existing timing information from reused system parts helps doing so with high accuracy. Using the TOAD approach, such information is easily accessible for succeeding projects.
In the case of the EBA (cf. Chapter 4), for example, the existing ACC functionality was only enhanced so that it uses the brakes in addition to the engine in order to reduce the vehicle speed. Thus, the ACC's previous timing behaviour is known and is used in the budgeting process.
The brake control requirement, for example, is defined as follows:

> The EBA shall prevent accidents at speeds between 30 and 130km/h, relative speeds of up to -60km/h to a vehicle in front and a minimum obstacle detection distance of 80m.

As a result of this requirement, the worst-case time to collision is 4.8s at a speed of 130km/h. Assuming a worst-case deceleration of $4m/s^2$ a time of 4.2s is required after full activation of the brakes. Hence, a required worst-case response time of the system of 0.6s can be estimated.
If the system is able to react faster than that, even higher vehicle speeds or lower obstacle detection distances are possible. Nevertheless, the minimum boundaries of the system's capability need to be defined very early in the development in order to ensure safety but also to ensure the sales potential of the final product. For example, maximum speeds of 80km/h may help with safe assumptions but may not be useful for most customers.
If, on the other hand, the system is not able to achieve the defined timing requirements, the following timing verification process is able to identify this issue as early as possible. Furthermore, it will help to identify root causes for the timing violations and thus help to find suitable solutions.

9.3.2 System Engineering

The system engineering is supported by TOAD-T tools which are used for the system's architecture definition. The system architect develops the architecture based on the top-level requirements and the existing platforms and components in mind in order to maximise reuse. Again, it is supported by the TOAD repository that contains the existing components and their timing behaviour descriptions.
For new hardware or software components, the system architect has to define the functional requirements, interfaces, variant dependencies and timing behaviour assumptions. After a successful automatic verification of the timing requirements on the newly defined system architecture, the timing assumptions can be transferred into timing requirements on component level.
However, if the timing verification fails, the system architect has to identify possible solutions. Assuming possible software optimisations in this early stage is usually considered too optimistic. And it also does not leave room for potential additional functions that may be introduced during the development. Hence, we expect changes of the hardware platform in such situations. These changes are also assisted by the TOAD approach. Here, the same functional architecture can be simulated with regards to timing on different hardware platforms in order to find the best fitting one.

9.3.3 Software Component Specification

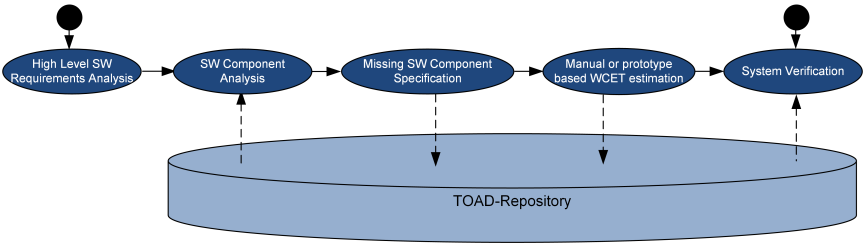


Figure 9.2: The TOAD process of software component specification.

While the system architect already defines software components in order to derive software requirements, the software architect will further refine those components based on the high level software requirements, existing software components and structural requirements like coupling and cohesion.
The EBA example is split up, for example, into the functions of object detection, control decision and actuation (brake and powertrain). Software components are derived, for example, from the object detection function (e.g. radar control and obstacle tracker).
The software component definition includes the definition of software component requirements, interfaces, variant constraints and timing behaviour. As a central repository is used for the architecture definition, the changes made by the software architect are directly reflected on the system level. Furthermore, the software architect can be a supplier who is thus able to communicate his interface requirements to the OEM, a process that is often performed but usually not explicitly defined.
As soon as the software component level is completely defined according to the high level software requirements, the system level timing verification can be performed automatically once again. However, if we cannot verify the timing behaviour in this step, the software architect will have to consult with the system architect in order to find a proper solution. This quick step back to the system level is usually not considered by existing processes but can save an entire development cycle.
On the software component level, the most crucial components can be im-

plemented as prototypes in order to test the functional assumptions and to have early estimates for their timing behaviour. For the EBA system we developed the EBA software component as a prototype and simulated different driving scenarios with the prototype. As depicted in Figure 9.3, the emergency brake assist engages at a speed of 130km/h and a distance to the obstacle in front of 40m. However, independent of specific scenarios, our system is verified to safely react in all scenarios in the boundaries of the defined requirements.

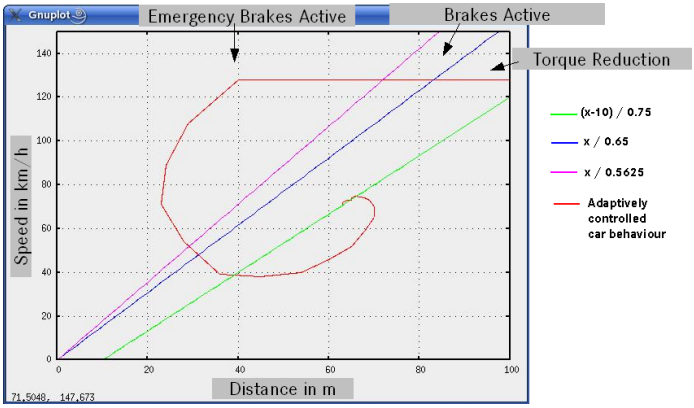


Figure 9.3: The EBA controlling a car, driving at 130km/h while approaching a car in front driving at 70km/h.

9.3.4 Software Component Implementation

The implementation part of the development starts with the automatic generation of a code or model frame based on the repository's existing definitions (cf. Figure 9.4).
Then, the component's functionality is either implemented by hand-coding or modelling. If the interface needs to be changed during implementation, the software architect has to be informed so that the changes can be performed on the component specification and new code or model frames can be generated. Our code generator supports round-trip engineering, i.e. we can adapt the generated interface during code implementation.

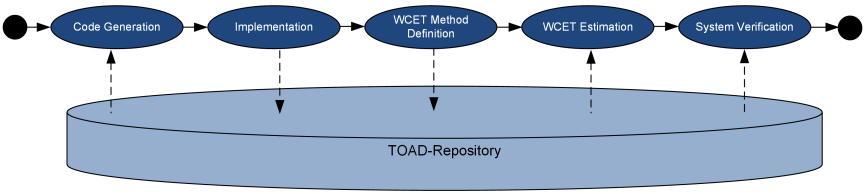


Figure 9.4: The Software Component Implementation process.

For the implementation, we recommend that the following coding rules are enforced in order to achieve predictable execution and to ease the static timing verification:

- Functions shall not call themselves, either directly or indirectly (MISRA-C 2012, Rule 17.2, [AS13]).
- The goto statement should not be used (MISRA-C 2012, Rule 15.1, [AS13]).
- A for loop shall be well-formed (MISRA-C 2012, Rule 14.2, [AS13])
- Use verifiable loop-bounds for all loops meant to be terminating (JPL Rule 2.3, [Lab09]).

The developer performs the WCET analysis by selecting one or multiple analysis methods as presented in Sections 8.4 and 8.5. If the developer is not able to perform this task, this will be noticed as soon as there is code in the repository and the WCET verification is missing. In such cases, he will be supported by WCET verification experts.
In every implementation step, the developer is able to perform a system verification based on his current implementation. This can also be automated by using nightly builds. Hence, the entire development group has constant transparency on the current timing behaviour.

9.4 Supplier Management

With the use of the TOAD repository, all tasks of the system development and timing verification can be distributed while the necessary result information is centrally available. The development and verification progress

is transparent between different roles in the process so that issues can be identified early.
With the repository and the TOAD process, the supplier has the additional ability to formally request specification interface or timing requirement changes is which unambiguous and saves development time.

9.5 Conclusion

We believe that the timing verification, which affects all engineering steps of a system development, can only be performed properly if the development process ensures that each development group has clear timing requirements and is able to verify these requirements for their contribution to the entire system.
We achieve that goal within TOAD-P mainly by transparency given by the TOAD repository which helps us furthermore in reusing existing components and thus in early and precise timing estimation.
TOAD-P allows OEMs and suppliers to work on different levels of the system development while having a formal exchange of the specification and keeping a high level of transparency when it comes to timing verification.

10 Results

10.1 Introduction

In this chapter, we want to evaluate our general achievements (see Section 10.2.1) as well as the achievement of the goals that we defined in Chapter 3 (see the following Sections).
These achievements consider the entire approach consisting of TOAD-A, TOAD-T and TOAD-P. While these achievements are rather in the field of Software Engineering, Section 10.3 shows technical improvements of the static timing verification that were discovered during the application and implementation of our approach.

10.2 Evaluation

In Chapter 3, we formulated our main goal as a timing verification process that is industrially applicable and considers automotive specific requirements. We will now argue that we have developed an integrated timing verification process (Section 10.2.1), that is industrially applicable (Section 10.2.2) and considers automotive requirements (Section 10.2.3).

10.2.1 Integrated Timing Verification Process

We understand the integrated timing verification as a development process that regards timing verification in every engineering step. I.e. timing requirements are specified during the requirement analysis phase and refined during specification and implementation, in order to allow a system level timing verification.
Central to our solution is our TOAD-A architecture (cf. Chapter 7) that allows a generalised formal specification of our system that can be stored in a central repository. This repository, in turn, allows a transparent development which is the basis for an actual and monitored application of the timing verification in each development step.

The TOAD-T tools highly automate the timing verification process wherever possible so that all developers can contribute independent of their technical background. The tools also reduce the overestimation of static timing estimations and thus help to increase the acceptance of our approach.
Finally, we defined the integrated timing verification process (cf. Chapter 9) and thus shown the benefits of our integration with regard to transparency, reusability and distributed development.

10.2.2 Industrial Applicability

The industrial applicability of the TOAD approach is mainly achieved by the TOAD-P process that enables the integration of static timing verification into existing development processes. TOAD-P's transparency and ability to assist all developers improves the acceptance of our approach compared to other approaches. Furthermore, the TOAD approach improves the communication in distributed development projects and thus reduces development costs.
The underlying TOAD-A architecture can be adapted to existing architectures, but it mainly targets software component architectures as shown in Chapter 7 with the adaptation of the ANTS architecture.
The automation, optimised timing and reduced overestimation achieved by the TOAD-T tools further drive the industrial applicability of the approach. And finally, the EBA example shows the general applicability of the integrated approach to industrial development projects.

10.2.3 Consideration of Automotive Requirements

The TOAD approach focuses on automotive specific aspects like scalability, distributed development and variant handling. It is furthermore intended for the use with distributed automotive network architectures.
TOAD-A is a scalable component architecture that also forms the basis for variant handling (via component specification) and distributed development (via exchangeable interface specifications and the central repository).
The state-of-the-art in timing analysis (cf. Chapter 5) indicates that variant rich systems have not played a big role in timing verification research so far. With our variant-aware WCET analysis we have shown that the WCET overestimation induced by highly variant systems on production

code can be up to 50 percent which can be regained with our approach.

10.2.4 Comparison with State-of-the-Art Approaches

The previously mentioned results help us to compare our TOAD approach with the state-of-the-art approaches as depicted in Table 10.1. Here, we use the table from Section 5.4 and add the TOAD approach.

Approach	Integrated Process	Industrial Applicability	Automotive Requirements
AADL, EAST-EEA, Meta-H	O	+	−
AUTOSAR	−	+	O
Geodesic	−	O	−
SaveCCM	O	−	+
SymTA/S	−	+	O
TDL/Giotto	−	O	−
Timed Automata	O	−	−
TOAD	+	+	+

Table 10.1: Comparison of the current state-of-the art approaches.

10.3 Technical Improvements

10.3.1 Parametric Timing Analysis

The parametric timing analysis has been found to be a key technique for the timing analysis of highly parametric automotive embedded real-time systems. We introduced two novel methods, the Analytical and ILP Analytic method and thus allow the developers to pick one out of four possible parametric timing analysis methods suitable for different parameter complexities (cf. Table 10.2). These methods were applied, for example, to the run-time environment of the TOAD-A architecture. There, the Analytical timing analysis method was applied. As a result, the WCET estimation can be performed by solving the algebraic equation below.

Method	Degree of Variability	Calculation Speed (example calculation)	Error-Prone
Fixed Values	no variability	zero (0 s)	low
Manual Analytic	low	high (10^{-7} s)	high
ILP Analytic	medium	medium (17 s)	medium
AIS Parametric	high	low (54 s)	low

Table 10.2: Time determination method comparison.

$$t_{Core} = t_{CoreO} + k \cdot (t_{CoreC} + t_{ChainO}) + \sum_{i \in l} t_{ChainC} + t_{C_i} \quad (10.1)$$

$$t_{C_i} = 2 \cdot (t_{DataO} + m_i \cdot t_{DataC}) \quad (10.2)$$

The parameters of this WCET formula are derived for particular platforms using commercial static WCET analysis tools (in our case for a C166 target).

10.3.2 Variant-Aware Timing Analysis

With our variant-aware timing analysis we are able to reduce the overestimation induced by infeasible execution paths resulting from variant constraints. The approach uses existing variant modelling constraints in the form of feature models. These constraints are transformed and linked into ILP constraints for the WCET analysis.
In three different case studies we were able to show that the WCET overestimation can be reduced by up to 50 percent. In one case study we could even show that not only overestimation can be reduced but also the actual execution time. This is due to the fact that the algorithm itself did not consider existing variant dependencies.
We extended these findings in the domain of WCET analyses to the variant-aware WCRT analysis which can reduce the timing estimates even further. Due to the lack of case study subjects we could only show an improvement of 0.1% for the one case that we analysed.

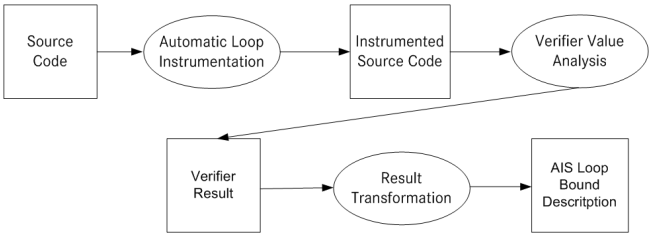


Figure 10.1: Automatic loop-bound estimation using abstract interpretation on the source code level.

10.3.3 Improved Loop-Bound Detection

As introduced in Section 8.4.4 and depicted in Figure 10.1, we developed a loop-bound detection method on the source code level. With our prototype tool chain, we were able to find 47 out of 51 loop-bounds that had not been found by aiT for our industrial example. Hence, we were able to increase the precision by 92 percent.
The introduced method lacks the support for hardware-dependent loop conditions by design and requires a proper setup of the source code analysis. Both aspects are treated by the TOAD approach as the application components have to be implemented hardware-independent and the source code analysis setup can be automated using the component interface specification.
As the manual loop-bound detection is an error-prone task, the integrated and automated estimation of upper loop-bounds also increases the safety of the WCET verification.

10.3.4 Improved Message Response Times

In Section 8.6.3, we introduced a benefit of integrated timing analysis that estimates execution and response times on all system levels. Here, we reduced the message response time pessimism through knowledge of the minimum message release intervals of CAN messages.
Minimum message release intervals are enforced by queueing the messages with timers at their worst-case component response time (CRT). Hence, we know a set of independent messages for each message (cf. Figure 10.2).

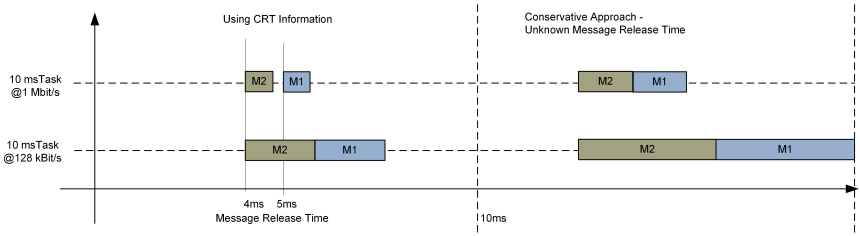


Figure 10.2: CAN messages that cannot interfere with each other are known by the TOAD approach.

A simple example of two messages, each having a length of 135 bit, shows the benefit of this approach. When the messages are released in a 10ms task with a Worst-Case CRT of 4ms (ID 2) and 5ms (ID1) the message response times are twice as high in the case that the TOAD approach is not applied (cf. Table 10.3).

Bitrate	MessageID	TOAD MRT [ms]	Common MRT [ms]
1 MBit	1	0.135	0.27
	2	0.135	0.27
128 kBit	1	1.11	2.11
	2	1.055	2.11

Table 10.3: An example for possible MRT optimisation resulting from the TOAD approach.

The adapted response time algorithm is as follows:

1. Determine the set of independent messages for each message (by using the existing MRT-analysis algorithm).
2. Calculate the response time for each message but exclude the independent messages from the calculation.
3. Refresh the sets of independent messages according to the calculated response times (add new independent messages).
4. Return to step 2 until the sets of independent messages do not change on refresh.

The worst-case result (no improvement) occurs if the algorithm stops with an empty set of independent messages. In that case, the result equals that of the standard algorithm.

10.3.5 Application of Timed Automata for Complex and Distributed Systems

In Section 8.3 we studied the applicability of model-checking using timed automata in the context of integrated timing verification. Furthermore, we modelled the EBA to study the industrial applicability. The results can be summed up as follows:

- The timing verification using timed automata can be performed if rather small, less complex systems are used (examples are given). The state explosion problem circumvents an application in more complex systems, as they occurred by refinement of high level models in our example. Thus, we introduced a method for the refinement of timed automata using simulation relations that allows the shifting of verification limitations.
- The integrated verification using the timed automata approach has the benefit of possible functional property verification in addition to the pure timing verification.
- Successful verifications are valid for the modelled timed automata. But the transformation into source code as well the execution on a given hardware have to be verified in addition.

Considering these findings, we realised that timed automata cannot not be used in the near future for the verification of today's complex and highly variant automotive systems.

11 Conclusion

11.1 Summary

The Timing Oriented Application Development (TOAD) approach deals with several yet unsolved problems in the timing verification domain for embedded and distributed real-time systems. Motivated by several safety-relevant and time-critical functions (e.g. air-bag, ABS or ESP), but mainly by the upcoming autonomous driving which has no human fall-back, we present our approach as a solution for a safe timing verification of complex, variant-rich and distributed real-time systems.
In contrast to existing approaches, the TOAD approach focuses on the aspect of integration in that it considers *all* relevant timing properties throughout the *entire* system development process. Thus, the current lack of proper timing verification, which we see also originating from a lack of defined system-wide responsibilities, is addressed and also covered by the TOAD process.
Our approach is based on a reference software architecture called TOAD-A and a set of new analysis and verification methods consolidated in the TOAD-T tool suite.
Most importantly, the TOAD approach increases the acceptance of timing verification as

- TOAD-A limits the use of asynchronous execution on the application level and thus ensures verifiability in early stages.
- TOAD-A automates the development process and ensures transparency on all development levels.
- TOAD-T reduces the WCET and WCRT overestimation in complex and highly variant systems.
- TOAD-T reduces the calculation time and allows a system level timing verification in each development step.
- TOAD-P defines roles and process steps that enforce timing relevant activities from an early consideration of timing aspects in the requirements down to a programming style that reduces timing verification efforts.

- TOAD-P improves the transparency in the internal as well as external communication in distributed development settings.

Finally, the TOAD approach shows that integrated timing verification can be applied even for complex real-time systems like the EBA example that is used throughout this thesis.

11.2 Limitations of the Approach and Future Work

By definition, our approach is limited to embedded real-time systems. Although the approach can be extended for the use with less deterministic architectures (event-based architectures, IP-networks, etc.), the results may not be verifiable safely.
The basis for our approach is a complete system description using TOAD-A. Mixed systems (TOAD-A together with other software architectures) have not been considered, as they require application specific adaptations of the TOAD-T tools.
The same is the case if other processor architectures or network protocols are to be applied. While most adaptations are relatively easy (e.g. a different aiT hardware target for WCET verification or a different message WCRT calculation), others may require more effort (e.g. calculation of task preemption overhead for complex processor architectures) or are not feasible at all (e.g. analytical WCET analysis for complex processor architectures).
In typical industrial scenarios, we expect that the introduction of the TOAD process will most often be performed gradually with the need for some kind of adaptation.
The complete description of timing properties in the TOAD approach is currently used for the verification of timing properties. In the future, we can imagine that this system is also used for optimisations such as finding cheap or lightweight hardware platforms for a given system with certain time bounds.
A next big step would also be the adaptation of TOAD-A to AUTOSAR architectures in order to dramatically widen the field of potential development projects.

Glossary

AADL	Architecture Analysis and Design Language, 50
ACET	Average-Case Execution Time, 3
ANTS	Agent NeTwork System, 75
AUTOSAR	AUTomotive Open System ARchitecture, 67
BCET	Best-Case Execution Time, 3
CAN	Controller Area Network, 37
CRT	Component Response Time, 115
CSMA	Carrier Sense Multiple Access, 49
CTL	Computation Tree Logic, 55
E2ERT	End-to-End Response Time, 3
EE	Electric/Electronic, 26
EEPROM	Electrically Erasable Programmable Read-Only Memory, 96
ERTS	Embedded Real-Time System, 7
LET	Logical Execution Time, 54
LIN	Local Interconnect Network, 117
MRT	Message Response Time, 115
NVM	Non-Volatile Memory, 74
OEM	Original Equipment Manufacturer, 18
OS	Operating System, 27
RTE	Run-Time Environment, 75

TCTL Timed Computation Tree Logic, 55
TDMA Time Division Multiple Access, 48
TOAD Timing Oriented Application Development, 59
TRT Task Response Time, 114

WCET Worst-Case Execution Time, 2, 3
WCRT Worst-Case Response Time, 3

Bibliography

[Abs17] AbsInt. *Company Homepage*. www.absint.com, 2017.

[ACD93] R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Inform. and Comput.*, 104(1):2–34, 1993.

[AD94] R. Alur and D. Dill. A theory of timed automata. *Theoret. Comput. Sci.*, 126(2):183–235, 1994.

[ADA16] ADAC. Pannenstatistik 2016. Technical report, ADAC, 2016.

[AS13] Motor Industry Software Reliability Association and Motor Industry Software Reliability Association Staff. *MISRA C:2012: Guidelines for the Use of the C Language in Critical Systems*. Motor Industry Research Association, 2013.

[AUT07] AUTOSAR. *Consortium Homepage*. www.autosar.org, 2007.

[Bal98] Helmut Balzert. *Lehrbuch der Softwaretechnik - Softwaremanagement, Software Qualitätssicherung, Unternehmensmodellierung*. Spektrum Akademischer Verlag, Heidelberg, Berlin, 1998.

[BCK03] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture In Practice, Second Edition*. Addison-Wesley, 2003.

[BCP03] G. Bernat, A. Colin, and S. Petters. pWCET, a tool for probabilistic WCET analysis of real-time systems. In *Proceedings of the 3rd International Workshop on WCET analysis.*, Porto, July 2003.

[BD91] Bernard Berthomieu and Michel Diaz. Modeling and verification of time dependent systems using time petri nets. *IEEE Trans. Softw. Eng.*, 17(3):259–273, 1991.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.

[Car07] Carnegie Mellon University's Software Engineering Institute. Online glossary: `www.sei.cmu.edu/opensystems/glossary.html`, April 2007.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages*, Los Angeles, January 1977.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming. Methods, Tools and Applications*. Addison-Wesley, 2000.

[Cer92] Karlis Cerans. Decidability of bisimulation equivalences for parallel timer processes. In *CAV*, volume 663 of *Lecture Notes in Computer Science*, pages 302–315. Springer, 1992.

[CHP05] J. Carlson, J. Håkansson, and P. Pettersson. SaveCCM: An analysable component model for real-time systems. In *Proceedings of the 2nd Workshop on Formal Aspects of Components Software (FACS 2005)*, Electronic Notes in Theoretical Computer Science. Elsevier, 2005.

[CKK02] Paul Clements, Rick Kazman, and Mark Klein. *Evaluating Software Architectures - Methods and Case Studies*. Addison-Wesley, 2002.

[Cru91] R. Cruz. A calculus for network delay. In *IEEE Transactions on Information Theory*, volume 37, pages 114–141, 1991.

[Dar59] Charles Darwin. *The Origin of Species by Means of Natural Selection*. John Murray, 1859.

[DBBL07] Robert I. Davis, Alan Burns, Reinder J. Bril, and Johan J. Lukkien. Controller area network (can) schedulability analysis: Refuted, revisited and revised. *Real-Time Syst.*, 35(3):239–272, 2007.

[dNR02] Dionisio de Niz and Ragunathan Rajkumar. Geodesic - a reusable component framework for embedded real-time systems. In *Submitted to the 8th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2002.

[EA11] Rouven Naujoks Ernst Althaus, Sebastian Altmeyer. Precise and efficient parametric path analysis. In *LCTES '11: Proceedings of the ACM SIGPLAN/SIGBED 2011 conference on Languages, compilers, and tools for embedded*, 2011.

[EC80] E. Allen Emerson and Edmund M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pages 169–181, London, UK, 1980. Springer-Verlag.

[EHRLR80] Lee D. Erman, Frederick Hayes-Roth, Victor R. Lesser, and D. Raj Reddy. The hearsay-ii speech-understanding system: Integrating knowledge to resolve uncertainty. In *Computing Surveys*, June 1980.

[Fed17] Federal Statistical Office Germany. Number of traffic accident fatalities again expected to decline in 2017 `https://www.destatis.de/EN/PressServices/Press/pr/2017/12/PE17_442_46241.html`, 2017.

[FGHL04] Peter H. Feiler, David P. Gluch, John J. Hudak, and Bruce A. Lewis. Embedded system architecture analysis using SAE AADL. Technical report, SEI, June 2004.

[FHT03] C. Ferdinand, R. Heckmann, and H. Theiling. Convenient user annotations for a WCET tool. In *3rd International Workshop on Worst-Case Execution Time Analysis*, 2003.

[Fle07] FlexRay Consortium. Flexray protocol specification v2.1 rev. a. www.flexray.com, 2007.

[FLV03] Peter Feiler, Bruce Lewis, and Steve Vestal. The SAE AADL standard: A basis for model-based architecture-driven embedded systems engineering. In *Workshop on Model-Driven*

Embedded Systems, Real-Time Application Systems (RTAS) Conference. Washington, D.C., May 2003.

[FMWA99] Christian Ferdinand, Florian Martin, Reinhard Wilhelm, and Martin Alt. Cache behavior prediction by abstract interpretation. *Sci. Comput. Program.*, 35(2-3), 1999.

[Gör03] Steffen Görzig. *Eine generische Software-Architektur für Multi-Agentensysteme und ihr Einsatz am Beispiel von Fahrerassistenzsystemen.* PhD thesis, Universität Stuttgart, 2003.

[GSSL94] Rainer Gawlick, Roberto Segala, Jørgen F. Søgaard-Andersen, and Nancy A. Lynch. Liveness in timed and untimed systems. In *ICALP*, volume 820 of *Lecture Notes in Computer Science*, pages 166–177. Springer, 1994.

[HÅCT04] H. Hansson, M. Åkerholm, I. Crnkovic, and M. Törngren. SaveCCM Ð a component model for safety-critical real-time systems. In *Euromicro Conference, Special Session Component Models for Dependable Systems*, Rennes, France, 2004. IEEE.

[HDD+03] J. Hatcliff, X. Deng, M. Dwyer, G. Jung, and V. Ranganath. Cadena: An integrated development, analysis, and verification environment for component-based systems. In *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)*, pages 160–173. IEEE, 2003.

[HGP+06] Michael González Harbour, José Javier Gutiérrez, José Carlos Palencia, José María Drake, Julio Medina, and Patricia López. Mast: A timing behavior model for embedded systems design processes. In *ARTIST workshop on Models of Computation and Communication MoCC, Zurich*, 2006.

[HHJ+05] Rafik Henia, Arne Hamann, Marek Jersak, Razvan Racu, Kai Richter, and Rolf Ernst. System level performance analysis - the symta/s approach, 2005.

[HHK01] Thomas A. Henzinger, Benjamin Horowitz, and Christoph Meyer Kirsch. Giotto: A time-triggered language

for embedded programming. *Lecture Notes in Computer Science*, 2211:166+, 2001.

[HK95] M. Henzinger and P. Kopke. Computing simulations on finite and infinite graphs. In *FOCS*, 36th Annual Symposium on Foundations of Computer Science, pages 453–462. IEEE Computer Society, 1995.

[HLS00] N. Holsti, T. Långbacka, and S. Saarinen. Using a worst-case execution time tool for real-time verification of the debie software. In *Proceedings of the DASIA 2000 Conference(Data Systems in Aerospace)*, September 2000.

[HLS+02] Pao-Ann Hsiung, Trong-Yen Lee, Win-Bin See, Jih-Ming Fu, and Sao-Jie Chen. Vertaf:an object-oriented application framework for embedded real-time systems, April 2002.

[HP02] Ensio Hokka and Markus Plankensteiner. Rapid prototyping und simulation verteilter echtzeitsysteme rapid prototyping und simulation verteilter echtzeitsysteme (in german). *Design & Verification*, 10, October 2002.

[ISO93] ISO. ISO 11898: Road vehicles - Interchange of digital information; Controller area network (CAN) for high-speed communication. Technical report, International Organization for Standardization, Geneva, CH, 1993.

[ISO05] ISO. ISO 17356: Open interface for embedded automotive applications. Technical report, International Organization for Standardization, Geneva, CH, 2005.

[ISO11] ISO. Road vehicles – Functional safety, 2011.

[JS06] Ilona Jüngst and Susanne Spotz. Dem fahrer stets zu diensten (in german). *trans aktuell*, 2006(20), 2006.

[KH04] J. Krákora and Z. Hanzálek. Timed automata approach for CAN verification. In *11th IFAC Symposium on Information Control Problems in Manufacturing (INCOM)*, 2004.

[Kir02] Christoph M. Kirsch. Principles of real-time programming. In *Proceedings of the 2nd International Worshop Embedded Software (EMSOFT), LNCS 2491*. Springer Verlag, 2002.

[Kop97] Hermann Kopetz. *Real-Time Systems. Design Principles for Distributed Embedded Applications*. Kluwer International Series in Engineering & Computer Science, 1997.

[KVL98] J.W. Krueger, S. Vestal, and B. Lewis. Fitting the pieces together: system/software analysis and codeintegration using metah. In *Digital Avionics Systems Conference, 17th DASC. The AIAA/IEEE/SAE*, pages C33/1–C33/8 vol. 1, 1998.

[Lab09] JPL Jet Propulsion Laboratory. JPL Institutional Coding Standard for the C Programming Language, 2009.

[LHO+06] Vera Lauer, Martin Hiller, Massimo Osella, Marko Auerswald, and Jürgen Lucas. Easis - electronic architecture and system engineering for integrated safety systems. In *Proceedings of the Transport Research Arena (TRA)*, 2006.

[LIN07] LIN Consortium. Lin specification, version 2.1. www.lin-subbus.org, 2007.

[LL73] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in hard real-time environments. In *Journal of the ACM, pp. 46-61*, 1973.

[LM95] Yau Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *in Proceedings of the 32nd ACM/IEEE Design Automation Conference*, 1995.

[LM99] Y. Li and S. Malik. *Performance Analysis of Real-Time Embedded Software*. Kluwer Academic Publishers, 1999.

[LPY97] K. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *Int. J. on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[LPY01] M. Lindahl, P. Pettersson, and W. Yi. Formal design and analysis of a gear controller. *Int. J. on Software Tools for Technology Transfer*, 3(3):353–368, 2001.

[LRL90] Sha L., R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers, p. 1175.*, 1990.

[LSD89] J. P. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm - exact chracterization and average case behavior. In *Proceedings of the 10th Real-Time Systems Symposium, pp. 166-171*, 1989.

[LTH02] Marc Langenbach, Stephan Thesing, and Reinhold Heckmann. Pipeline modeling for timing analysis. In *Proceedings of the Static Analyses Symposium (SAS)*, volume 2477 of *LNCS*, Madrid, Spain, 2002.

[Mar17] P. Marwedel. *Embedded System Design*. Springer-Verlag, 2017.

[MAWF98] Florian Martin, Martin Alt, Reinhard Wilhelm, and Christian Ferdinand. Analysis of loops. In Kai Koskimies, editor, *CC*, volume 1383 of *Lecture Notes in Computer Science*. Springer, 1998.

[MGL06a] Pascal Montag, Steffen Görzig, and Paul Levi. Applying static timing analysis to component architectures. In *Proceedings of the 28th International Conference on Software Engineering , Shanghai (China)*, May 2006.

[MGL06b] Pascal Montag, Steffen Görzig, and Paul Levi. Challenges of timing verification tools in the automotive domain. In *2nd IEEE International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, Paphos (Cyprus)*, November 2006.

[MNL06] Pascal Montag, Dirk Nowotka, and Paul Levi. Verification in the design process of large real-time systems: A case study. In *Automotive Safety and Security 2006, Stuttgart (Germany)*, October 2006.

[MOS06] MOST Cooperation. Most specification rev. 2.5. www.mostcooperation.com, 2006.

[Nad05] Andreas Naderlinger. A plug-in architecture for platform-specific code generation from tdl components. Master's thesis, University of Salzburg, October 2005.

[OS01] M. Van Osch and S. A. Smolka. Finite-state analysis of the can bus protocol. In *HASE '01: The 6th IEEE International Symposium on High-Assurance Systems Engineering*, pages 42–54. IEEE, 2001.

[PPKT04] Gustav Pomberger, Wolfgang Pree, Christoph Kirsch, and Josef Templ. *Software Engineering - Architektur-Design und Prozessorientierung*. Hanser Fachbuch Verlag, 2004.

[PS17] Pure-Systems. *Company Homepage*. http://www.pure-systems.com/, 2017.

[RD05] John Regehr and Usit Duongsaa. Preventing interrupt overload. In *LCTES '05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 50–58, New York, NY, USA, 2005. ACM Press.

[Ree98] Glenn Reeves. Re: What really happened on mars? Risks-Forum Digest, Volume 19: Issue 58., 1998.

[RGdFE99] Valentin Valero Ruiz, Fernando Cuartero Gomez, and David de Frutos Escrig. On non-decidability of reachability for timed-arc petri nets. *pnpm*, 00:188, 1999.

[RJB04] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.

[RSE+03] Manuel Rodríguez, Nuno Silva, João Esteves, Luis Henriques, Diamantino Costa, Niklas Holsti, and Kjeld Hjortnaes. Challenges in calculating the WCET of a complex on-board satellite application. In *WCET*, pages 11–15, 2003.

[Rus02] John Rushby. An overview of formal verification for the time-triggered architecture. In *Proceedings of 7th International Symposium on Formal Techniques in Real-Time and*

Fault Tolerant Systems, volume 2469 of *LNCS*, pages 83–105. Springer, 2002.

[RW03] Anders Rylander and Erik Wallin. Lin - local interconnect network - for use as sub-bus in volvo trucks. Master's thesis, Chalmers University of Technology, Göteborg, 2003.

[RWT+06] Jan Reineke, Bjoern Wachter, Stephan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A definition and classification of timing anomalies. In *Workshop on Worst-Cacse Execution Time (WCET) Analysis 2006 (WCET 06)*, 2006.

[SBW01] Harmen Sthamer, André Baresel, and Joachim Wegener. Evolutionary testing of embedded systems. In *Proceedings of the 14th International Internet & Software Quality Week*, 2001.

[Sch02] Jörn Jakob Schneider. *Combined Schedulability and WCET Analysis for Real-Time Operating Systems*. PhD thesis, Universität des Saarlandes, 2002.

[SG96] Mary Shaw and David Garlan. *Software Architecture - Perspectives on an Emerging Discipline*. Prentice Hall, Upper Saddle River, N. J., 1996.

[SR00] J. Will Specks and Anatal Rajnák. Lin - protocol, development tools, and software interfaces for local interconnect networks in vehicles. In *Proceedings of the 9th International Conference on Electronic Systems for Vehicles, Baden-Baden*, 2000.

[Sun97] Jun Sun. *Fixed-Priority End-To-End Scheduling in Distributed Real-Time Systems*. PhD thesis, University of Illinois at Urbana-Champaign, 1997.

[Sym07] SymtaVision. *Company Homepage*. www.symtavision.com, 2007.

[SZ03] Jörg Schäuffele and Thomas Zurkawka. *Automotive Software Engineering: Grundlagen, Prozesse, Methoden und Werkzeuge*. Friedr. Vieweg & Sohn Verlag /GWV Fachverlage GmbH, Wiesbaden, Juli 2003.

[TAKB96] Serdar TaŞiran, Rajeev Alur, Robert P. Kurshan, and Robert K. Brayton. *Verifying abstractions of timed systems*, pages 546–562. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.

[TB94] K. Tindell and A. Burns. Guaranteed message latencies for distributed safety-critical hard real-time control networks. YCS 229, University of York, Computer Science, 1994.

[TBW95] K. Tindell, A. Burns, and A.J. WEllings. Calculating controller area network (can) message response times. In *Control Engineering Practice*, pages 1163–1169, 1995.

[TCN00] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *Circuits and Systems, 2000. Proceedings. ISCAS 2000 Geneva., pp. 101-104*, 2000.

[TEH+03] T. Thurner, J. Eisenmann, M. Haneberg, S. Voget, R. Geiger, and U. Virnich. The east-eea project - a middleware based software architecture for networked electronic control units in vehicles. In *Electronic Systems for Vehicles, Baden-Baden, Germany, VDI Berichte 1789*, 2003.

[The04] Stephan Thesing. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models.* PhD thesis, Universität des Saarlandes, 2004.

[The07] The Mathworks. *Company Homepage.* www.mathworks.com, 2007.

[THW94] K. Tindell, H. Hansson, and A.J. Wellings. Analysing real-time communications: Controller area network (can). In *Proceedings of the 15th IEEE Real-Time Systems Symposium*, pages 259–265, 1994.

[TIM07] TIMMO. *Project Homepage.* www.timmo.org, 2007.

[Tin94] Ken Tindell. An extensible approach for analysing fixed priority hard real-time tasks. *Journal of Real-Time Systems*, 1994.

[Tin00] Ken Tindell. Deadline monotonic analysis. *Embedded Systems Programming (www.embedded.com)*, 2000.

[TSH+03] Stephan Thesing, Jean Souyris, Reinhold Heckmann, Famantanantsoa Randimbivololona, Marc Langenbach, Reinhard Wilhelm, and Christian Ferdinand. An abstract interpretation-based timing validation of hard real-time avionics software. *dsn*, 00:625, 2003.

[TTT08] TTTech. *Company Homepage*. www.ttautomotive.com, 2008.

[Tur36] Alan Turing. On computable numbers, with an application to the Entscheidungsproblem. In *Proceedings of the London Mathematical Society, Series 2, 42, pp 230-265*, 1936.

[Ves98] Steve Vestal. MetaH Avionics Architecture Description Language. An emerging standard language for specifying avionics architectures. A toolset for design, modeling, analysis, system integration. Presentation of Honeywell Labs, Minneapolis, MN, USA, 1998.

[Weg01] Joachim Jens Wegener. *Evolutionärer Test des Zeitverhaltens von Realzeit-Systemen*. PhD thesis, Humboldt-Universität zu Berlin, 2001.

[Wer03] M. Wernicke. New design methodology from vector simplifies the development of distributed systems. Vector Informatik Press Release, June 2003.

[WGR+09] Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 28(7), July 2009.