Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik

State-based Timing Analysis for Distributed Systems

Von der Fakultät für Informatik, Wirtschafts- und Rechtswissenschaften der Carl von Ossietzky Universität Oldenburg zur Erlangung des Grades und Titels eines

Doktors der Ingenieurwissenschaften (Dr. Ing.)

angenommene Dissertation von

Dipl.-Inform. Tayfun Gezgin

geboren am 12.02.1984 in Bremen

Gutachter:
Prof. Dr. Achim Rettberg
Prof. Dr. Bernhard Josko
Prof. Dr. Marco Wehrmeister

Tag der Disputation: 02.10.2017

Abstract

Functionalities of systems in safety-critical domains like the automotive are typically distributed over several computation units. Such systems have to work exactly as specified as a violation of a requirement could result in critical situations and in loss of human life. Error corrections after the start of production of a system could also lead to very high costs. A major aspect of correctness in safety-critical systems is the timeliness of computations. Systems have to finish certain critical computations in a timely manner in order to be safe and reliable. Thus, it is important to have rigorous analysis techniques such as formal verification. Unfortunately, interdependencies among functions and interferences on shared resources complicate the verification of such hard real-time properties. Moreover, changes affecting the specification and the implementation of a system might occur during the design process, which further complicate the verification process, as already performed analyses have to be repeated.

This thesis addresses these problems. A state-based approach for the analysis of timing constraints combining analytic and model checking methods is introduced. In analogy to model checking methods, the full state space for the analysis is taken into account. In classical scheduling analyses only the critical instance of a system is considered, which typically leads to highly pessimistic results. This could lead to expensive systems, as more computation units would be required to satisfy the timing requirements than actually needed. With the approach presented in this thesis exact response times are computed. This results in a reduced demand of computation resources, while guaranteeing that all timing constraints are still fulfilled. In order to alleviate the problem of state space explosion due to state unfolding performed by the presented approach, the state space of an architecture is constructed in an iterative manner. Minimization operations are applied on the interfaces between resources to keep the resulting state spaces as small as possible. To further boost the scalability, abstraction techniques on interfaces between dependent resources are worked out. The effects of the specific abstractions are evaluated with respect to the advantages for scalability and the adequacy of the results.

On top of this timing analysis an impact analysis approach is introduced in this thesis to minimize re-verification efforts of timing properties needed when the considered system is modified. Adaptations of the architecture of an already existing and analyzed system could be for instance the addition of new functionalities. Resulting tasks of these new functions could be allocated to existing resources of the system. As verification tasks are typically time consuming it is desirable to minimize the effort of a re-verification and to reuse previous results of analyses which have not been affected by architectural

changes. Two abstraction levels are defined which are affected through changes, i.e. the specification level and the implementation level.

On the *specification level* contracts are applied and a *virtual integration checking* technique is introduced. Contracts enable the designer to distinguish between *assumed* behavior, which must be offered by the environment of the system, and *guaranteed* behavior, for which the system itself is responsible. Certain characteristics of contracts are used in this thesis to enable a timed automaton-based verification approach. Contracts are specified by using a pattern-based language. An approach is introduced which transforms the language fragments automatically to timed automata. Besides automatically verifying the correct compositions and refinements of parts of a system based on the corresponding contracts, this technique allows to determine the impact of changes on previous analysis results on the specification level. If a contract is changed in such a manner that it refines the previous one, only the internals of the corresponding system have to be re-verified.

On the *implementation level* a refinement relation between state transition systems of interfaces of components is defined. If a change occurs on a resource the approach is able to determine whether the interfaces to dependent resources are affected. If the interfaces did change in a "good manner" the verification of dependent resources can be omitted, thus saving unnecessary verification times.

In a typical system design process, all introduced concepts are exploited in combination. Therefore, an overall methodology is presented which integrates all introduced verification techniques. Changing a part of a system encapsulating some functionalities implicates the integration of this part into its context, and the re-verification of its adapted implementation against the local specification.

Besides dedicated smaller examples and benchmark systems from related papers, the presented approach is evaluated by the application of an industrial driver assistance system case study.

Zusammenfassung

Funktionalitäten von Systemen in sicherheitskritischen Domänen wie dem Automobilsektor sind üblicherweise über viele Berechnungseinheiten verteilt. Solche Systeme müssen exakt nach deren Spezifikation arbeiten, da eine Verletzung einer Systemanforderung zu einer kritischen Situation führen und Menschenleben gefährden könnte. Außerdem führen Fehlerkorrekturen nach Produktionsstart zu hohen Kosten. Ein wesentlicher Aspekt von sicherheitskritischen Systemen ist, dass die Berechnungen rechtzeitig erfolgen müssen. Damit solche Systeme sicher und zuverlässig arbeiten, müssen deren Berechnungen rechtzeitig abgeschlossen werden. Deswegen ist es wichtig, dass diese Systeme durch beispielsweise der Anwendung von formaler Verifikation gründlich analysiert werden. Allerdings erschweren gegenseitige Abhängigkeiten zwischen Funktionen des Systems und Interferenzen zwischen verschiedenen Funktionen, die durch den Zugriff auf gemeinsam genutzten Ressourcen entstehen können, die Analysen harter Realzeiteigenschaften. Erschwerend kommt hinzu, dass es während der Systementwicklungsphase zu Änderungen an der Spezifikation und der Implementierung des Systems kommen kann. Bereits durchgeführte Analysen müssen dann wiederholt werden.

Im Rahmen dieser Dissertation werden solche Probleme in der Designphase von Systemen adressiert. In dieser Arbeit wird ein zustandsbasiertes Verfahren zum Analysieren des korrekten Schedulings vorgestellt, das analytische und Model Checking Verfahren kombiniert. Für die Analyse wird analog zu den Model Checking Verfahren der gesamte Zustandsraum berechnet. In klassischen Analysen wird ausschließlich die kritische Instanz eines Systems betrachtet. Dies führt zu pessimistischen Antwortzeiten, was wiederrum zu teureren Systemen führen kann, da mehr Berechnungseinheiten eingeplant werden müssen, um den zu hoch eingeschätzten zeitlichen Anforderungen gerecht zu werden. Mit dem in dieser Arbeit vorgestellten Verfahren können solche Kosten eingespart werden, da exakte Antwortzeiten der Tasks berechnet werden. Model Checking Ansätze haben typischerweise Probleme mit der Skalierbarkeit. Die Zustandsräume werden bereits für kleine Systeme sehr groß. Um dieses Problem zu verringern, wird der Zustandsraum der kompletten Architektur in dem hier vorgestellten Ansatz iterativ berechnet. Die Zustandsräume an den Schnittstellen von abhängigen Ressourcen werden dabei durch spezielle Methoden minimiert. Um die Skalierbarkeit weiter zu erhöhen, werden für solche Zustandsräume weitere Abstraktionstechniken vorgestellt. Die Effekte der einzelnen Abstraktionstechniken werden bezüglich der Zustandseinsparung und der Genauigkeit der erzielten Ergebnisse evaluiert.

Die Timing Analyse wird außerdem mit einer Impact Analyse erweitert, mit der

Re-Verifikationen von zeitlichen Eigenschaften verringert werden sollen. Solche Re-Verifikationen sind durch Änderungen im System erforderlich, die während der Entwurfsphase vorgenommen werden. Beispielsweise werden weitere bisher nicht eingeplante Funktionalitäten hinzugefügt, sodass weitere Tasks auf bereits existierende und analysierte Ressourcen allokiert werden. Da Verifikationsaufgaben typischerweise sehr zeitaufwendig sind, gibt es ein großes Potential Entwicklungszeiten durch das Wiederverwenden von vorherigen Analyseergebnissen, die durch Änderungen nicht beeinflusst worden sind, einzusparen.

Durch Änderungen werden im Wesentlichen die zwei Abstraktionsebenen *Spezifikation* und *Implementierung* beeinflusst. Auf der Spezifikationsebene wird eine Technik zur *virtuellen Integrationsprüfung* auf Basis von *Contracts* vorgestellt. Contracts ermöglichen es dem Entwickler, zwischen Verhalten, das durch den Kontext geliefert werden muss, und dem, das durch das System garantiert werden muss, zu unterscheiden. Einige charakteristische Eigenschaften der Contracts werden in dieser Ausarbeitung genutzt, um eine auf Timed Automaten basierende Verifikationstechnik zu ermöglichen. Dabei werden Contracts durch eine Pattern-basierte Sprache erfasst. Die einzelnen Pattern werden dann durch das Verfahren automatisch zu solchen Automaten transformiert. Zusätzlich zur Verifikation der korrekten Kompositions- und Verfeinerungsbeziehungen von Teilen des Systems ermöglicht diese Technik Auswirkungen von Spezifikationsänderungen auf bisherige Analyseergebnisse zu ermitteln.

Auf der Implementierungsebene wird eine Verfeinerungsbeziehung zwischen den Zustandsräumen voneinander abhängiger Komponenten vorgestellt. Wenn eine Änderung an einer Ressource durchgeführt wird, ist der vorgestellte Ansatz in der Lage zu entscheiden, ob Schnittstellen zu abhängigen Ressourcen betroffen sind und damit diese neu analysiert werden müssen oder ob unnötige Verifikationszeit eingespart werden kann. Alle vorgestellten Konzepte finden typischerweise im Entwurfsprozess eine kombinierte Anwendung. Deswegen wird in dieser Dissertation eine Methodik ausgearbeitet, die alle vorgestellten Verifikationstechniken integriert. Das Ändern eines Teilsystems führt zu einer Integrationsprüfung zum Restsystem und der Re-Verifikation der internen Struktur oder der geänderten Implementierung. Neben dedizierten, kleineren Beispielen und Benchmark-Systemen aus verwandten Veröffentlichungen wird das hier vorgestellte Verfahren an einer Fallstudie zu einem Fahrerassistenzsystem evaluiert.

Contents

1. Introduction **11**
1.1. Motivation 11
1.2. Objective of this Thesis 13
1.3. Context of this Thesis 17
1.4. Outline 18

2. Foundations **21**
2.1. Scheduling of Real-Time Tasks 23
2.1.1. Tasks and Task Dependency Graphs 24
2.1.2. Event Models 25
2.2. Timed Languages and Timed Automata 27
2.3. Modeling of System Architectures 33
2.3.1. Components and Resources 33
2.3.2. Modeling in MARTE 36
2.4. Specification of Requirements 38
2.4.1. Contract-based Design 39
2.4.2. Requirement Specification Language 44
2.5. Summary 49

3. State-based Timing Analysis **51**
3.1. Motivation 51
3.2. Related Work 53
3.2.1. Classical Analytical Approaches 53
3.2.2. Model Checking Approaches 56
3.2.3. Combination of Analytical and State-based Approaches 59
3.2.4. Contribution of this Chapter 60
3.3. General Approach 60
3.3.1. Iterative Analysis Approach 61
3.3.2. Symbolic Transition Systems of Resources 63
3.3.3. Simplification of Symbolic Transition Systems 67
3.4. Operations on Symbolic Transition Systems 70
3.4.1. Interface Computation 70
3.4.2. Composition Function 75

3.5. Analysis Algorithm . . . 79
3.5.1. Main Analysis Algorithm . . . 82
3.5.2. Successor Computation . . . 83
3.5.3. Completeness and Soundness of Algorithm . . . 91
3.5.4. Termination of Algorithm . . . 94
3.5.5. Minimization through Untimed Bisimulation, Timed Simulation Relation . . . 95
3.6. Abstraction Techniques . . . 97
3.6.1. Clock Resets and Duration Clocks . . . 98
3.6.2. Clocks of Interface STSs . . . 99
3.6.3. Abstraction through Simulation Relation . . . 102
3.6.4. Effects of Over-Approximations for Iterative Analysis Approach . . . 105
3.6.5. Testing: Abstraction through Under-Approximation . . . 105
3.6.6. Abstraction for Event Bursts . . . 107
3.7. Case Study: Driver Assistance System . . . 109
3.7.1. Overview . . . 109
3.7.2. Lane-Keeping-Support System . . . 112
3.7.3. Evaluation Results . . . 114
3.7.4. Observation on Scalability . . . 115
3.8. Summary . . . 115

4. Contract-based Impact Analysis **117**
4.1. Motivation . . . 117
4.2. Related Work . . . 118
4.2.1. Tool Support for Verification of Contract Specifications . . . 118
4.2.2. Impact Analysis . . . 121
4.2.3. Contribution of this Chapter . . . 123
4.3. Impact Analysis Methodology . . . 124
4.4. Impact Analysis on Specification Level . . . 129
4.4.1. Simplifying the Virtual Integration Condition . . . 130
4.4.2. Timed Automaton-based Analysis Approach . . . 134
4.5. Impact Analysis on Implementation Level . . . 142
4.5.1. Combining State-based Analysis with a Refinement Checking Technique . . . 143
4.5.2. Refinement through Simulation Relation . . . 145
4.5.3. Combining Impact Analysis with Abstractions . . . 146
4.6. Evaluation and Case Studies . . . 147
4.6.1. Contract-Level of the Driver Assistance System . . . 147
4.6.2. Impact Analysis on the Driver Assistance System . . . 150
4.6.3. Evaluation of Refinement on Implementation-Level . . . 151
4.7. Summary . . . 155

5. Summary and Outlook **157**

A. Tool Support **161**

B. Handling Architectures including Restricted Loop Structures **165**

C. Generated Timed Automata **169**

List of Figures **173**

List of Tables **177**

Index **179**

Nomenclature **181**

Bibliography **185**

1. Introduction

Over the last few years, the amount of functionality of systems in the hard real-time and safety-critical domains such as the avionics or the automotive which is realized by software heavily increased. The usage of more and more software intensive systems targets the increase of safety of vehicles and planes, and also targets the quality of traveling comfort and energy efficiency. The premise to increase the safety of vehicles is to guarantee correct system functionality. This is achieved by testing the system intensively or by performing exhaustive verifications.

A major aspect of correctness is the timeliness of computations. Systems have to finish certain critical computations in a timely manner in order to be safe and reliable. This thesis targets the efficient and systematic verification of such timing properties of safety-critical systems.

1.1. Motivation

Today up to 90 percent of all innovations, i.e. new and improved functionalities, are realized by the usage of electronic and software [1]. This trend will continue in future, as X-by-Wire systems and the increased interconnection of vehicles to their environments leading to cooperating traffic systems are already becoming commonplace in the market. An example in the avionics domain for this is the dynamic partitioning of the airspace with respect to time, which was investigated in the SESAR (Single European Sky ATM Research) program. The recent partitioning of the airspace is performed in a static manner, i.e. the trajectories of planes are not changed during landing approaches and takeoffs. The shift to a dynamic partitioning, which are called 4D-trajectories, involves an increased software-supported cooperation between the tower and the airplanes.

For safety-critical systems it is crucial that these adhere to their specifications as a violation of a requirement could result in critical situations leading to very high costs or even threats to human life. The verification of requirements in early design steps is a critical issue as the later an error is detected in a system the higher are the costs to perform corrections.

Recently, Nissan for example offers a car in the premium segment fitted with a steer-by-wire system, in which the steering commands are transmitted electrically to a control unit and then to an actuator which actually performs the steering movement. After the

[1] `http://www.presseportal.de/pm/67565/2723743` [Jan. 18, 2016]

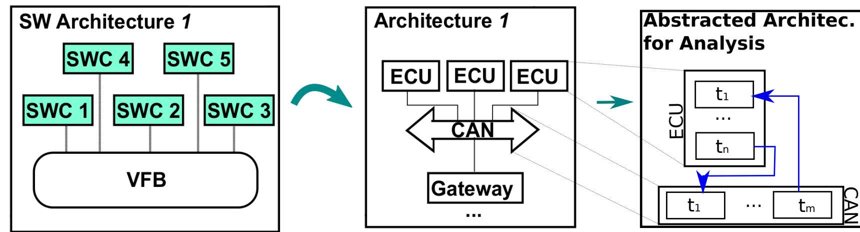


Figure 1.1.: General concept of the model-based design approach.

start of production in the year 2014 Nissan had to recall the vehicles due to some delays in the emergency program. As recalls are quite expensive, it is desirable to perform analyses at early design steps.

In order to cope with the complexity of adequately developing safety-critical systems, the model-based design paradigm was introduced and is widely used in development processes. Systems can be build up intuitively in a bottom-up or top-down fashion by the usage of so-called *components* as illustrated in Figure 1.1. A component is self-contained and provides a fraction of the functionalities of a system. It has a well defined interface and may contain a set of parts or subcomponents. To specify models in a reusable and interchangeable manner it is desirable to use domain specific modeling standards. Typically, the design of the overall system is performed by the original equipment manufacturer (OEM). In a first step the OEM designs the software components in form of logical architectures. The components and parts of this system are then realized and implemented by either the OEM itself or by various suppliers. In order to get adequate realizations from each supplier, the OEM has to specify the extra-functional properties and interfaces unambiguously. To capture the specification of a component many specification formalisms were introduced as for instance the language of Live Sequence Charts (LSC) [DH01] or the contracts-based specifications [Mey92]. Such formal languages offer a rigorous semantics enabling automatic verification.

When all suppliers deliver the implementations of the software components (SWCs), the OEM has to verify whether all SWCs *fit together*, i.e. he has to perform a consistency check in a black-box manner, and whether all higher level requirements which range over several SWCs are realized by the decomposition structure. After the implementation of all SWCs the logical architecture has to be allocated to a hardware architecture, on which the functionality shall be executed. The hardware architecture consists of electronic control units (ECUs) which are interconnected by bus systems. At this design stage technical details such as resource consumptions and timing latencies have to be verified. To perform such analyses, typically the architecture is abstracted in an appropriate manner. The kind of abstractions considered in this thesis is illustrated in the right part of Figure 1.1: ECUs and bus systems are treated logically equivalent in the sense that both represent

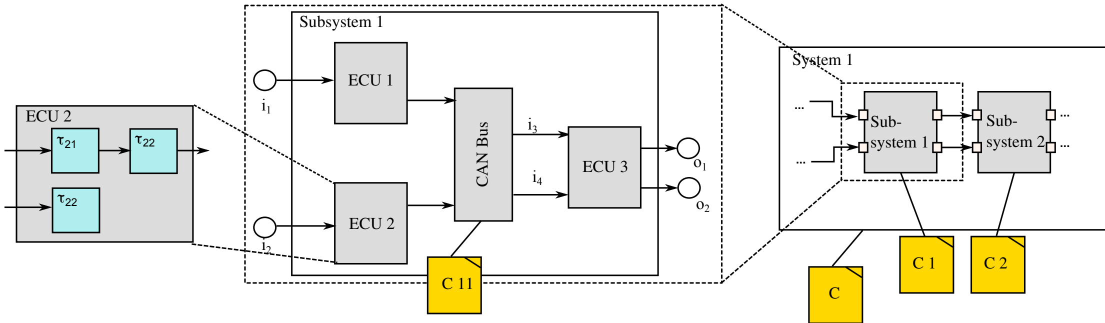


Figure 1.2.: Considered system architectures with assume/guarantee style contracts.

computation units on which a set of tasks are allocated. This is a valid approach, as a bus also needs some processing time to deliver a message to the correct recipient(s). The order of executions of the tasks is determined by the corresponding scheduling policy.

Specifications made during the development process of a system or a software component are typically subject of changes. New requirements from the OEM could be stated which were forgotten previously, existing requirements may get refined or corrected, or further functionalities have to be included into the system. Such changes on the system architecture, the parts of the implementations, or the specifications typically require that already performed analyses have to be repeated. As such tasks are time consuming it is desirable to minimize the effort of a re-verification.

This thesis targets the analyses tasks mentioned above. Modeling languages will not be detailed in this thesis as the scope is not the modeling but the analysis of hard real-time systems. The contributions of this thesis and the worked out analyses tasks are detailed in the following section.

1.2. Objective of this Thesis

The objective of this thesis is the analysis and verification of hard real-time systems. The focus is on the verification of timing requirements in early design steps, which is a critical issue, as late changes, which have to be performed due to design errors, typically lead to high costs.

System architectures consisting of sets of processing units (ECU) and bus systems are considered, on which a set of executable tasks is allocated as illustrated in Figure 1.2. These systems can be a part of a larger context such as *Subsystem 1* in the figure. Independent tasks are triggered by events of a corresponding event stream (ES). Event streams are characterized by a period and a jitter. Aperiodic event streams are not considered in this thesis. Event streams can be characterized by upper and lower occurrence

curves as introduced in the real-time calculus [TCN00]. The timing specifications of the systems are given in terms of assume/guarantee style contracts (abbreviated with C, C_i in Figure 1.2).

Regarding the semantics of models of the considered timed systems, discrete and continuous time domains can be applied. The discrete domain is closer to final implementations, while in the dense time semantics technical details like sampling times are abstracted. This is an advantage in early design steps, as the designer can keep the focus on the correct functionality and the overall timings. In this thesis the dense time semantics is considered.

The approach for scheduling analysis worked out in this thesis combines both analytical methods and model checking methods, where violations of timing constraints are decided through the concept of the reachability of bad states. The scheduling analysis is based on a model checking approach covering all reachable states of the system thus determining exact response times of the allocated tasks and end-to-end latencies of task chains.

More specifically, the goal is to determine response times for each task and whether some timing constraints of tasks (i.e. deadlines) and end-to-end latency constraints are violated. For this, the entire state space of a given system architecture, which includes all task interleavings and dependencies, is constructed in an iterative manner. The timing analysis thereby proceeds as follows: To build the state space of a resource, its *input behavior* has to be determined, which defines the *activation times* of all allocated tasks. State spaces are represented by symbolic transition systems (STS): The states determine a range of valuations of clock variables. Also, states include the information which task is currently running, is interrupted, or in the ready queue. A resource can have multiple sources for its inputs. To determine a *single* input state space for each resource, all these inputs have to be combined by an appropriately defined composition operation. The computed state space of a resource is then used as an input for *dependent* resources, i. e. for resources on which dependent tasks are allocated. To keep the interface between the resources as small as possible, parts of the state space that are not relevant for the input behavior of the dependent resources are abstracted.

Besides the timing analysis, an impact analysis approach is worked out, which is applied if changes in the already analyzed system architecture appear. Such changes are not unusual in a typical development process of software intensive systems. Sources of changes are for example the integration of new features to previously implemented software components, the update of implementations, or the change of requirements. In this thesis an impact analysis approach is introduced to reduce re-verification aspects for timing properties when such changes occur. The approach is able to handle changes on both the specification level and the implementation level.

Suppose that the specification of a system component shall be replaced. If the new contract of the adapted specification *refines* the previous contract, there is only a need to check the internals of the new component itself: If the considered component is decomposed in further subcomponents with local contracts, it has to be checked whether

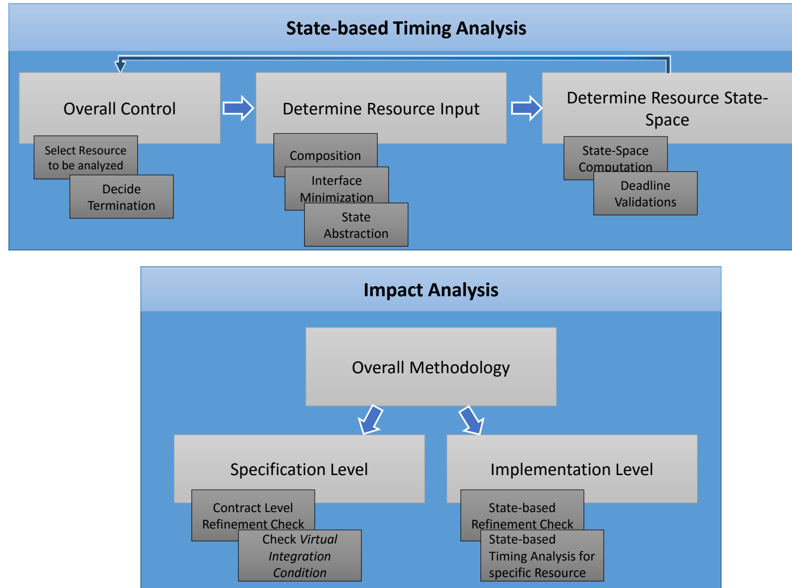


Figure 1.3.: Contributions of this thesis.

the new component contract is still fulfilled by the contracts of all subcomponents. For such scenarios a timed automaton-based virtual integration checking technique is introduced. Otherwise, if the new contract is not a refinement of the old one, an additional consistency check with all dependent parts of the system has to be performed, i.e. an additional virtual integration check has to be performed.

The presented approach on the implementation level is relevant for cases where certain aspects of the *realization* of a component are changed such as adding a new task on an existing resource, the merge of two tasks in a single one, or even the change of the complete implementation. When such a change occurs, the approach is able to determine whether the interfaces of dependent resources are affected through the concept of a refinement analysis: It is checked if the new interface between dependent resources *refines* the old interface. In such a case a re-verification of dependent resources is omitted.

The complete approach and the contributions are illustrated in Figure 1.3:

- A state-based analysis approach for timing analysis is worked out, which consists of the computation of the input of resources and the computation of the state spaces of resources itself. Thereby, a main control mechanism determines the order of the resources to be analyzed and decides when the termination condition is reached. To realize the state-based analysis, some operations are defined:

  - A composition operation is defined to determine the input of a resource. Further, an interface minimization operation is presented, which abstracts from specific parts of the state space of a resource. The abstraction concerns such parts, which are not relevant to compute the input behavior of a specific dependent resource.
  - A set of abstraction techniques are introduced. These techniques are applied on the interfaces of dependent resources and target the scalability of the approach. The effects of the specific abstractions are evaluated with respect to the advantages for scalability and the adequacy of the results.
  - To compute the state space of a resource, an algorithm is worked out, with which it is possible to determine whether violations of timing constraints in the system can occur. The correctness of this algorithm is given through the proof of the soundness and completeness properties.

- An impact analysis approach is worked out, which determines affected parts of the system architecture when a change occurs.
  - On specification level a timed automaton-based analysis approach, which is based on the *virtual integration condition* of contracts, is worked out. Certain characteristics of contracts are used in this thesis to enable such a timed automaton-based verification approach. Contracts are specified by using a pattern-based language. An approach is presented which transforms the language fragments automatically to timed automata.
  - On implementation level, an appropriate refinement relation on the interfaces of resources is defined. It is shown that if a refinement is given, the satisfaction of timing properties is preserved.
  - The overall impact analysis methodology is worked out, which describes how the basic techniques are combined.

- The implementations of the introduced approaches are evaluated on dedicated benchmarks from related papers. Further, a driver assistance system case study is worked out on which the applicability of the approach is demonstrated.

The contribution to the corresponding research areas is the following:

- *State-based timing analysis*: In literature, typical approaches are based on holistic analyses, where the complete state set of the system is constructed in a single step. The difference of this work is to perform the analysis in an iterative fashion: First, dependencies between components are analyzed. Based on the dependency graph, the order of the state space construction of the involved resources is performed. Thus, the state space is constructed iteratively. This is a major advantage for scalability in contrast to the approaches in literature, as abstractions after each

analysis step are enabled, with which the overall state space can be kept smaller than in a holistic approach. Note that the goal of this thesis is neither to compare the introduced approach with state-of-the-art tools or to measure the effectivity in contrast to such tools as the prototype implementation of the concept is not suited and optimized for such purposes. By *improved scalability* we refer to the reduced state spaces with which the analysis concept has to deal in contrast to holistic approaches.

- *Impact analysis*: The combination of a scheduling analysis technique together with an impact analysis approach realizes a powerful verification methodology, with which implementation aspects (timing) and specification aspects (refinement, composition) are verified using the same formalism. Further, the impact analysis approach realizes the analysis of affected parts of a system when changes occur. This is an advantage as verification times of systems are typically time consuming.

In the appendixes of this document further helpful details concerning the tool implementations of the introduced analyses approaches are provided.

1.3. Context of this Thesis

The functional correctness of safety-critical systems heavily depends on the timeliness of computations, where computations have to be finished within the defined timing constraints. This defines the context of this thesis: First, to verify timing properties a verification technique is necessary. Second, a methodology which defines how to efficiently deal with re-verifications tasks has to be given.

The approach worked out in this thesis integrates in the classical development process as follows: During the design of real-time systems timing constraints for safety-critical functionalities are defined. By choosing low-performance computation units for example in order to save cost, the system could react too slow in certain situations leading to critical scenarios. Thus, after the design of a system architecture, where executable tasks are defined and allocated to resources, these constraints have to be verified. The approach introduced in this thesis targets this verification step.

In literature two types of real-time properties are defined: *Hard* real-time deadlines are considered to be safety properties. Missing such a deadline corresponds in a total system failure. In contrast to this, *soft* real-time deadlines are used to for performance requirements. These need not always to be satisfied but the response times shall be minimized to offer a good service quality. The focus of the thesis is on the hard real-time properties.

One state-of-the-art approach for such problems is the approach of SymTA/S, which was realized by Symtavision. The main idea behind SymTA/S is to transform event streams whenever needed and to exploit classical scheduling algorithms for local analyses.

Event streams describe the activation patterns for tasks by upper and lower occurrence curves, realizing a compositional analysis method. Unfortunately, this concept delivers pessimistic results when inter-ECU task dependencies exist, as the analysis abstracts completely from concrete state-based interdependencies. The main approach of this thesis picks up on the compositional concept of SymTA/S by realizing an iterative analysis.

An alternative approach is based on model checking, and has been illustrated for example in [FPY02, DILS09]. Here, all entities like tasks, processors, and schedulers are modeled in terms of timed automata. The most famous tool used for this approach is Uppaal. Analogous to [FPY02] the full state space for the timing analysis is considered in this thesis, where all interleavings and task dependencies are preserved. In contrast to these works, the approach presented here constructs the state space of the architecture in an iterative manner. With this iterative approach new minimization operations on the interfaces of dependent resources are enabled, such that a more scalable analysis technique is realized.

Technically speaking, the Uppaal DBM library [2] is used as the basis of the approach presented in this thesis. This library includes a clock zone implementation with all necessary basic operations such as zone intersection or reset of dedicated clocks. On top of these basic operations higher level functions to realize the computation of the state spaces of computation resources in an iterative manner are defined and implemented in this thesis.

Another aspect of this work is the re-verification of parts of the system architecture. During the design stage of such systems changes such as new or adapted requirements or new features which have to be offered occur. Thus, besides the timing analysis, verification tasks have also to deal with such changes. With this, there is a need for a methodology, which defines how to efficiently deal with such re-verifications tasks. Ideally, this methodology is based on the defined verification technique and extends it seamlessly by re-verification abilities.

1.4. Outline

Chapter 2: Foundations In Section 2.1 the basic aspects of the scheduling of real-time tasks are introduced. An overview of scheduling policies is given. Task characteristics are defined together with activation patterns, which trigger tasks at certain points in time. In Section 2.2 the formalism of timed automata and the semantics of networks of timed automata are introduced. Section 2.3 addresses the modeling of real-time systems. The concept of components and resources are illustrated in a formal way. As designer of systems do not use these formal constructs to design system architectures, the usage of a higher level modeling profile called Marte is illustrated. Section 2.4 deals with the specification of requirements in a pattern-based manner and introduces the contract-

[2] `http://people.cs.aau.dk/~adavid/UDBM/`

based design approach. The last section summarizes all concepts introduced in this chapter.

Chapter 3: State-based Timing Analysis This chapter starts with the review of related works which address the analysis of hard real-time systems. In Section 3.3 the idea of the general analysis approach is sketched, and the state space of a resource is defined. A simplified version of the symbolic transition systems (STS) introduced in the foundations chapter is presented. Operations on the symbolic transition systems needed to realize the analysis approach are introduced in Section 3.4, which are the abstraction and composition operations for STSs. Section 3.5 details the computation of the STS of a resource. In Section 3.6 abstraction techniques are introduced, which lead to more pessimistic response times but generally increase the scalability of the approach. In Section 3.7 the analysis technique is applied to a lane-keeping-support system (LKS) case study. Finally, a summary of this chapter is given.

Chapter 4: Contract-based Impact Analysis First, related works on verification tools for contract specifications, and on approaches reducing the effort of performing reverifications are given. In Section 4.3 the overall methodology of the impact analysis approach consisting of two basic verification techniques on the specification and the implementation level is presented. Both levels are detailed in the subsequent Sections 4.4 and 4.5. The approach is evaluated in Section 4.6 on a driver assistance system use-case, which extends the previously introduced LKS. Finally, all results of this chapter are summarized in Section 4.7.

Chapter 5: Summary and Outlook This chapter summarizes the main results and contributions of this work, and gives future research directions.

2. Foundations

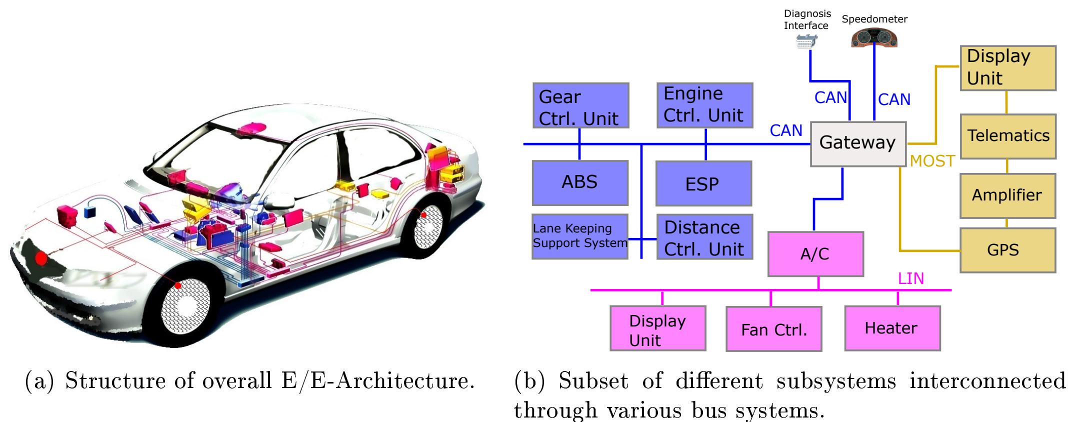
Typical real-time system architectures in the automotive and avionics domains consist of a set of control units, actuators, and sensors, which are interconnected directly or by bus systems like sketched in Figure 2.1(a). While in the nineties there were only a handful of control units used in cars for functions like the anti-blocking system (ABS), in today's cars the electrical/electronic architectures (E/E architectures in short) consist of up to 70 control units, interconnected by up to ten bus systems taking on tasks like the engine control or the dynamic stability control preventing to over- and under-steer the car [1].

In general, the overall architecture is separated in a set of networks using different bus protocols, e.g. CAN (Controller Area Network), LIN (Local Interconnect Network), or MOST (Media Oriented Systems Transport). This is illustrated in Figure 2.1(b). Protocols are chosen on certain required properties like the degree of predictable behavior, cost, or bandwidth. When data shall be exchanged between different networks, on which incompatible or compatible protocols are implemented, Gateways are used, which synchronize and translate the data adequately. The functionality of a gateway is in general realized by an already existing control unit on which some further tasks are allocated. In this work gateways are not considered explicitly. In this thesis only gateways that connect compatible protocols are considered such that these can be treated in a same manner as standard communication resources.

For communication resources in safety-critical systems, typically the CAN protocol is used. The advantage of this protocol is that messages from tasks with a high criticality may be processed before messages from tasks with less criticality, such that these computations are finished as early as possible. Besides this, the protocol allows to easily add further resources and tasks communicating over the corresponding bus without changing the architecture. For time division protocols this is not the case.

Various modeling formalisms have been worked out to model such systems like for example SysML (Systems Modeling Language), EAST-ADL (Electronics Architecture and Software Technology – Architecture Description Language), or MARTE (Modeling and Analysis of Real-Time and Embedded systems). In this work, modeling languages will neither be discussed in detail nor will be compared. The focus of this work is not the modeling but the analysis of hard real-time systems. This work is also not restricted to a specific modeling language. To enable the approach for a specific language an appropriate mapping of the contained model elements to the elements considered for the

[1] `http://www.autonews-123.de/`



(a) Structure of overall E/E-Architecture.

(b) Subset of different subsystems interconnected through various bus systems.

Figure 2.1.: E/E-Architecture of typical medium-sized cars.

analysis must be defined. The single restriction here might be that not the complete set of elements might be supported by the analysis approach.

Regarding the semantics of models of timed systems, discrete and continuous time domains can be considered. The discrete domain is closer to the final implementation of the system, where problems like the sampling time have to be considered. Anyway, in early design steps like the modeling phase the designer should not care about sampling times and the adequate choice of granularity. Thus, the dense time semantics has the advantage that the designer can focus on the correct functionality and the overall timing, and abstract from such implementation details. In this thesis the dense time semantics is considered.

For hard real-time systems it is crucial to specify requirements such as maximum end-to-end latencies in a formal manner to enable automatic analyses. In literature many methods and techniques have been worked out for this, as for instance temporal logics which are based on propositional logics, pattern-based languages, or sequence diagram-based visual formalisms. On top of these formalisms the *contract-based design* approach was introduced allowing to distinguish between assumed context behavior and guaranteed behavior of the system. Such unambiguous specification formalisms are especially crucial for larger systems, where several departments of a company and various suppliers on different design levels work in parallel, and all artifacts have to be integrated in later design steps. If interfaces are not specified appropriately, such integration approaches could fail and lead to expensive re-designs.

In this chapter, first the main aspects of real-time systems such as scheduling policies, task characteristics, and activation patterns are introduced, which trigger tasks at certain points in time. All basic terms used in this thesis will be defined and outlined and thus

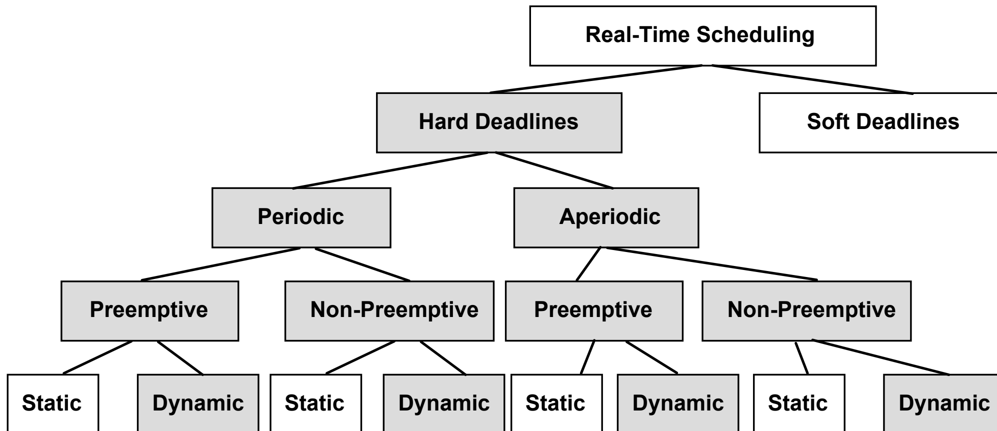


Figure 2.2.: General classes of scheduling algorithms [Mar06].

is a prerequisite to understand the content of this work. In Section 2.2 the formalism of timed automata is illustrated and the semantics of networks of timed automata is detailed. This formalism serves as the basis of the analyses approaches introduced in Chapters 3 and 4. Section 2.3 addresses the modeling of real-time systems. For this, components and resources are introduced in a formal manner. As designers of systems do not use these formal constructs to design a system architecture, the modeling profile Marte is presented and the usage in the context of this thesis is illustrated. This will ease the understanding of the case studies of this work. Section 2.4 deals with the specification of requirements in a pattern-based manner and introduces the contract-based design approach. The timing requirements specified in these formalisms serve as the input for both the timing analysis and the impact analysis approaches. The last section summarizes all concepts introduced in this chapter.

2.1. Scheduling of Real-Time Tasks

Scheduling is a crucial aspect in real-time systems, as a chosen scheduling algorithm directly affects the response times of the tasks on a computation resource. A scheduling policy is a strategy which determines the assignment order of tasks to the processor, such that each task is executed until its completion [But05]. A task is an abstraction of a (part of a) program that is executed by the resource on which it is allocated, i. e. it consumes processing time. A resource processes its allocated and activated tasks in a sequential manner [But05]. A schedule is *feasible* if all task executions can be completed before their corresponding *deadlines* expire. A deadline thereby determines the maximal allowed time frame from releasing a task to the termination of the task.

In [Mar06] the scheduling algorithms are classified with respect to their characteristics, which is also illustrated in Figure 2.2: Scheduling for hard deadlines requires that all task deadlines are fulfilled, as a violation could result in a catastrophic result, while soft deadlines need not always to be fulfilled but the response times may be minimized. As the focus of this thesis is the analysis of hard real-time properties occurring in systems like an engine control system, more details about soft deadlines will not be discuss in this work.

The scheduling policies distinguish between periodic and aperiodic tasks. Periodic tasks are activated with a fixed inter-arrival time defined by the corresponding period. The inter-arrival times may jitter with a specific amount of time. A sporadic (or aperiodic) task is in general a task which may appear at arbitrary points in time. As the focus here are predictable systems which can be analyzed by the concept of reachability checks, sporadic tasks are restricted in such a way that a minimum and a maximum inter-arrival time is always specified. These values bound the number of occurrences of aperiodic tasks.

A scheduler may be preemptive or non-preemptive. Preemptive schedulers allow that task executions can be interrupted at any point in time. If a critical task with high priority arrives while a non-critical task with a lower priority is executing, it can immediately start to run without delaying in the ready queue. For some scenarios non-preemptive scheduling policies have to be used, especially in cases in which tasks are considered to run in an atomic fashion, like tasks on a communication resource.

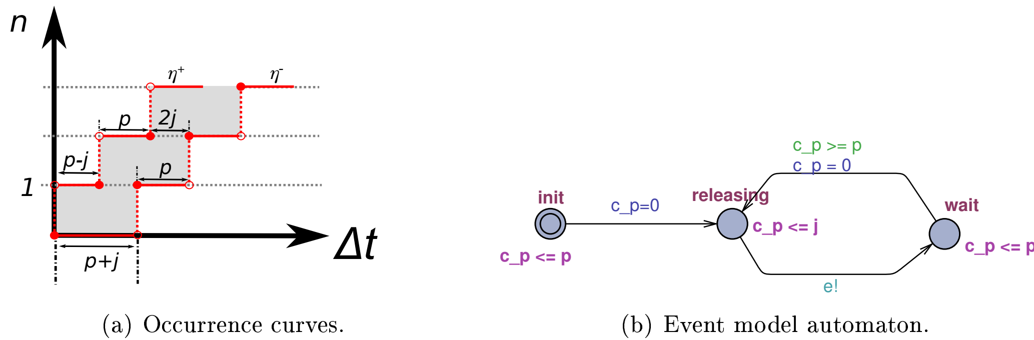
Another characteristic of a scheduler is whether it is static or dynamic. These terms are not unambiguous in literature. Here, it is referred to the definition of [Mar06]: In a static schedule the order and timings of tasks is defined at design time. For this, the start times and durations of tasks are encoded in a table. In contrast, dynamic scheduling determines at run-time, which task should be executed and which has to be enqueued in the ready list. The focus of this thesis are dynamic scheduling policies. In Figure 2.2 the class of schedulers which is of interest for this work are marked through boxes colored in gray.

In the next subsections the characteristics of tasks and their activation patterns is detailed.

2.1.1. Tasks and Task Dependency Graphs

A task is a tuple $\tau = (bcet, wcet, d, pr)$, where $bcet, wcet \in \mathbb{N}_{\geq 0}$ are the best- and worst-case execution times with respect to the allocated resource with $bcet \leq wcet$, $d \in \mathbb{N}_{\geq 0}$ is its relative deadline determining the maximal allowed time frame from release time to task termination, and $pr \in \mathbb{N}_{\geq 0}$ is the fixed priority of the task. We will refer to the elements of a tasks by indexing, e.g. $bcet_\tau$ for task τ. The set of all tasks is called T.

Other properties of a task can be inherited by the above ones: The *release time* of a task is the time, at which it becomes active, i.e. available for execution. The *response*



(a) Occurrence curves.

(b) Event model automaton.

Figure 2.3.: Periodical activation of independent tasks.

time is the minimal and maximal time frame between the task release and its completion. The lateness is the difference between the response time and the corresponding relative deadline.

Precedence constraints between tasks are modeled by task dependency graphs as defined in the following.

Definition 1 (Task Dependency Graph)**.** *Let* T *be a set of (distinct) tasks. A task dependency graph (TDG) is a directed acyclic graph* $G = (\mathsf{T}, E)$*, where* T *represent the vertices of the graph and* $E \subseteq \mathsf{T} \times \mathsf{T}$ *is the set of edges representing task dependencies. Further let* E^* *be the transitive closure of* E*. A task* t_1 *is called an* immediate *predecessor of* t_2 *if* $(t_1, t_2) \in E$*, and a predecessor if* $(t_1, t_2) \in E^*$*.*

A chain of task dependencies is a path in the corresponding TDG. In literature, such chains of tasks are also called *pipelines* [SSL$^+$14]. If for a task τ there exists no task τ' such that $(\tau', \tau) \in E$ holds, then τ is called independent . Independent tasks are triggered via event streams, which is the topic of the next section.

2.1.2. Event Models

Event models are models used to specify the allowed timings of events, which activate corresponding tasks. An (infinitely) long sequence of events is called *event stream*. Thus, event streams are more specific in the sense that event models define a (possibly infinite) set of event streams. There are several types of event models, like strictly periodic, periodic with jitter, or sporadic.

In this work, the *periodic with jitter* model is of interest, which is characterized by a period $p \in \mathbb{N}_{>0}$ and a jitter $j \in \mathbb{N}_{\geq 0}$. Such streams can be characterized by upper and lower occurrence functions $\{\eta^+, \eta^-\} : \mathbb{N} \rightarrow \mathbb{N}$ as introduced in the real-time calculus [TCN00] and illustrated in Figure 2.3(a). Occurrence functions are piecewise constant

step functions. They have unit-height steps of size one which corresponds to the occurrence of one event characterizing the activation of a corresponding task. For any length of a time interval $\Delta t \in \mathbb{N}$ the upper function $\eta^+(\Delta t)$ determines the maximum number of events that can occur within a time frame of such a length, while the lower function $\eta^-(\Delta t)$ determines the minimum number of events.

The *periodic with jitter* event model is specified through the following upper and lower occurrence functions [RRE03]:

$$\eta^+(\Delta t) = \left\lceil \frac{\Delta t + j}{p} \right\rceil, \tag{2.1}$$

$$\eta^-(\Delta t) = max(0, \left\lfloor \frac{\Delta t - j}{p} \right\rfloor). \tag{2.2}$$

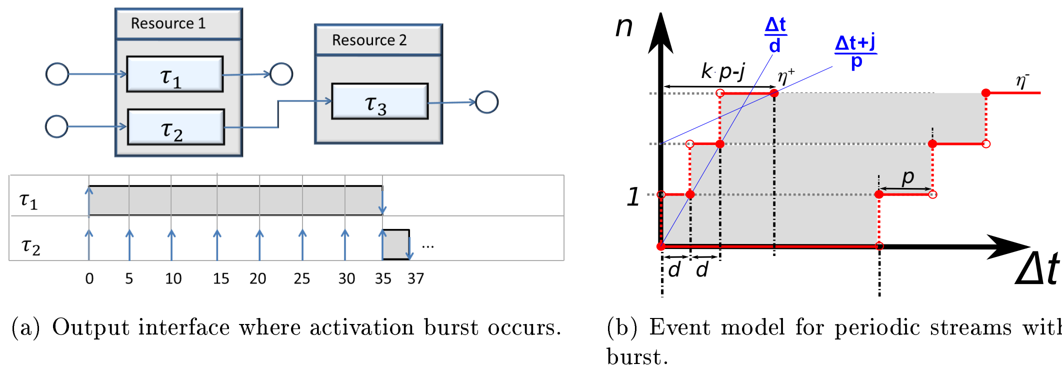
With this model also streams with jitter larger than the period can be captured. If the jitter is larger than the period, the initial step size of the upper function would be larger than one, which describes that more than one event is considered to occur simultaneously. This is a pessimistic approach, as in practical cases there is always a minimal distance between event arrivals. Jitter typically occur when a task is triggered from another task with a varying response time. Such a response time is always greater than zero, as a task will never finish its execution instantaneously.

Non-simultaneous arrivals of events were the topic of [Ric04]: In their work the authors introduced *minimum inter-arrival times* to the periodic model with jitter to support jitter larger than the period. The model is characterized by the parameters period p, jitter j and a minimum event distance d. The minimum event distance could be for example the best-case response time of a task, from which the considered one depends on. The model is defined as follows:

$$\eta^+(\Delta t) = min(\left\lceil \frac{\Delta t}{d} \right\rceil, \left\lceil \frac{\Delta t + j}{p} \right\rceil). \tag{2.3}$$

The first part of the function captures the burst behavior for small Δt, the second part captures the *long term* behavior of the stream which corresponds to the standard event model *periodic with jitter* of Formula 2.1. The lower function is the same as the standard event model for *periodic with jitter* of Formula 2.2.

This event model is relevant for event burst scenarios. Consider for example the scenario of Figure 2.4(a): On *Resource 1* two independent tasks τ_1, τ_2 are allocated, with periods $p_{\tau_1} = 50$ time units and $p_{\tau_2} = 5$ time units, and execution times $bcet_{\tau_1} = wcet_{\tau_1} = 35$ time units and $bcet_{\tau_2} = wcet_{\tau_2} = 2$ time units. As the priority of τ_1 is higher than the priority of τ_2 a large response jitter from activation of the task until completion is adhered at the output of the resource. On *Resource 2* the task τ_3 is allocated which depends on the output of τ_2, where the event model characterized above is used to describe the event burst activation behavior.



(a) Output interface where activation burst occurs.

(b) Event model for periodic streams with burst.

Figure 2.4.: Event streams with burst behavior.

In this work, occurrence curves are modeled through timed automata like illustrated in Figure 2.3(b). According to the occurrence functions, the first event can arrive immediately ($t = 0$) or can wait a full period in state *init* and jitter in state *releasing*. This is achieved through the invariant in state *init* specifying that the clock of the automaton c_p may not exceed the period, and the invariant in state *releasing* stating that c_p may not exceed the jitter. Note that by switching from state *init* to *releasing* the clock is reset. All successive events may arrive a full period after the last event at the earliest, delayed by jitter j at the latest, which is achieved through the invariant $c_p \leq p$ in state *wait* and the guard $c_p \geq p$ on the transition from state *wait* to *releasing*. The details of timed automata are discussed in the next section.

2.2. Timed Languages and Timed Automata

A (formal) language $\mathcal{L}$ is in general a set of finite words over some finite alphabet Σ and is thus a subset of Σ^*, which is the set of all finite words over Σ. For non-terminating systems like control units or operating systems, infinite words instead of finite ones are used to characterize their behavior. Such languages are called $\omega-$languages. Thus, an $\omega-$language over Σ is a subset of Σ^ω, the set of all infinite words over Σ. If an $\omega-$language is accepted by some *Büchi Automata* then it is called $\omega-$regular. A Büchi Automaton is mainly a finite state machine with an additional acceptance criterion such that infinite words can be recognized. In a Büchi Automaton a subset of the states are marked as *accepting*. A Büchi Automaton accept words, for which there is an infinite sequence of states (also referred to as *run*) of the automaton, of which some are accepting and are visited infinitely often. Regular and $\omega-$regular languages are widely used in information systems as these are closed under all set-theoretic Boolean operations such as intersection and complement, and also under several other operations such as concatenation and

Kleene star.

To specify the behavior of *timed* systems, Büchi Automata were extended by real-valued variables called clocks in [AD94]. These automata accept timed $\omega-$regular languages over a given alphabet and an infinite time sequence.

Timed Languages

Timed words are infinite sequences of symbols of some finite alphabet, enriched by a monotonically increasing infinite time sequence. The time values determine the occurrence times of the corresponding symbols.

Definition 2 (Timed Word and Language)**.** *A timed word (or trace) over a non-empty finite alphabet* Σ *is a pair* $\sigma = (\rho, \tau)$ *of an infinite sequence* $\rho = \rho_0, \rho_1, \cdots$ *of symbols from* Σ*, and an infinite sequence* $\tau = \tau_0, \tau_1, \cdots$ *of non-negative real values from* $\mathbb{R}^+$*, such that*

1. $\forall i \in \mathbb{N}: \ \tau_i \leq \tau_{i+1},$ *(Monotonicity)*
2. $\forall t \in \mathbb{R}^+ \ \exists i \in \mathbb{N}: \ \tau_i > t.$ *(Progress)*

The set of all timed words over Σ *and a time sequence* τ *is denoted by* $(\Sigma, \tau)^\omega$*.*

As we will only use *timed* words in the following, we will abbreviate the set of all timed words with Σ^ω. A timed language $\mathcal{L}$ over Σ is a set of timed words with $\mathcal{L} \subseteq \Sigma^\omega$. In many cases it is important to only consider finite prefixes of (infinite) timed words. This is defined in a recursive fashion.

Definition 3 (Trace Prefix)**.** *Let* $\sigma = (\rho_0, \tau_0)(\rho_1, \tau_1) \cdots$ *be a timed trace over* Σ*. The finite prefix of* σ *of length* $n+1$ *with* $n \in \mathbb{N}_{>0}$ *is recursively defined as follows.*

$$\begin{aligned} pre(\sigma, 0) = &\quad (\rho_0, \tau_0), \\ pre(\sigma, n) = &\quad pre(\sigma, n-1)(\rho_n, \tau_n). \end{aligned}$$

Further for $i \geq 0$ *let* $\sigma^i := (\rho_i, \tau_i)$ *determine the* i^{th} *element of a trace* σ*.*

Timed Automata

In this work, the formalism of timed automata are used for several purposes: First, event streams defining the activation behavior of independent tasks are characterized in terms of timed automata as described in the previous subsection. Second, the worked out timing analysis approach which is the topic of Chapter 3 is based on the symbolic representation of the induced state spaces of timed automata. Third, the virtual integration check introduced in Chapter 4 will also be based on this formalism.

Timed automata are finite automata extended by a finite set of real-valued variables called *clocks*. The formalism of timed automata was introduced by Alur and Dill in [AD94] in order to define a modeling concept for real-time systems. Here, the syntax

and semantics of timed automata are defined as employed by UPPAAL [LPY97]. UPPAAL adapts timed safety automata introduced in [HNSY92]. In such automata, progress is enforced by means of *local invariants* instead of an accepting condition. States – also referred to as locations – may be associated with such invariants, which are timing constraint defining upper bounds on clocks. In the sense of the classical Büchi Automata [AD94] all runs of a safety automaton are considered to be accepting.

Let C be a set of clocks. A clock constraint is a conjunction of upper and lower bounds of clock variables and differences of clocks. Formally, a clock constraint is defined by the syntax

$$\varphi ::= \ c_1 \sim t \mid c_1 - c_2 \sim t \mid \varphi \wedge \varphi, \tag{2.4}$$

where $c_1, c_2 \in C$, $t \in \mathbb{Q}_{\geq 0}$ and $\sim \in \{\leq, <, =, >, \geq\}$. The set of all clock constraints over the set of clocks C is denoted by $\Phi(C)$.

A valuation of a set of clocks C is a function $\nu : C \to \mathbb{R}_{\geq 0}$ assigning each clock in C a non-negative real number. By $\nu \models \varphi$ it is denoted that a clock constraint φ evaluates to true under the clock valuation ν. The notion 0_C is used to denote the clock valuation $c = 0$ for all $c \in C$. A time shift denotes the passage of time and is abbreviated by the notion $\nu + d := \nu(c) + d$ for all $c \in C$. By the notion $\nu[\varrho \mapsto 0]$ resets for a subset of clocks $\varrho \subseteq C$ are denoted, where $\nu[\varrho \mapsto 0](c) = 0$ if $c \in \varrho$, and $\nu[\varrho \mapsto 0] = \nu(c)$ if $c \notin \varrho$.

With these ingredients the formalism of timed automata together with their semantics is enabled to be defined in the following.

Definition 4 (Timed Automaton)**.** *A* Timed Automaton *(TA) is a tuple $A = (L, l^0, \Sigma, C, R, I)$ where*

- *L is a finite, non-empty set of locations, and $l^0 \in L$ is the initial location,*
- *$\Sigma = \Sigma_! \cup \Sigma_?$ is a finite alphabet of channels partitioned in sending ($\Sigma_!$) and receiving ($\Sigma_?$) events,*
- *C is a finite set of clocks,*
- *$R \subseteq L \times \Sigma \cup \{\varepsilon\} \times \Phi(C) \times 2^C \times L$ is a set of transitions. A tuple $r = (l, \sigma, \varphi, \varrho, l')$ represents a transition from location $l \in L$ to location $l' \in L$ annotated with an action $\sigma \in \Sigma$, a constraint $\varphi \in \Phi(C)$, and a set $\varrho \subseteq 2^C$ of clocks which are reset.*
- *$I : L \to \Phi(C)$ is a mapping which assigns an invariant to each location.*

The semantics of timed automata is given by timed transition systems.

Definition 5 (Timed Transition System)**.** *Let $A_i = (L_i, l_i^0, \Sigma_i, C_i, R_i, I_i)$ with $i \in \{1, ..., n\}$ be a network of timed automata with pairwise disjoint sets of clocks. The semantics of such a network is defined in terms of a timed transition system $\mathcal{T}(A_1 \parallel ... \parallel A_n) = (Conf, Conf^0, C, \Sigma, \to)$, where*

- *Conf* $= \{(l, \nu) \mid l \in L_1 \times ... \times L_n \;\wedge\; \nu \models \bigwedge_{j=1}^{n} I_j(l_j)\}$ *is the set of configurations, and* $Conf^0 = (l^0, 0_C)$ *is the initial configuration, where* $l^0 = (l_1^0, ..., l_n^0)$ *is the initial location and* 0_C *is the initial clock valuation,*
- $C = C_1 \cup ... \cup C_n$,
- $\Sigma = \Sigma_1 \cup ... \cup \Sigma_n$,
- $\rightarrow \subseteq Conf \times (\Sigma \cup \mathbb{R}_{\geq 0}) \times Conf$ *is the transition relation. A transition* $((l, \nu), \lambda, (l', \nu'))$, *also denoted by* $(l, \nu) \xrightarrow{\lambda} (l', \nu')$, *has one of the following types:*
  - *A flow transition* $(l, \nu) \xrightarrow{t} (l, \nu + t)$ *with* $t \in \mathbb{R}_{\geq 0}$ *can occur, if* $\nu + t \models \bigwedge_{j=1}^{n} I_j(l_j)$.
  - *A discrete transition* $(l, \nu) \xrightarrow{\lambda} (l', \nu')$ *with* $l' = l[l_{\{i,j\}} \rightarrow l'_{\{i,j\}}]$ *can occur, if for some* $i, j \in \{1, ..., n\}$ *and* $\lambda \in \Sigma_i \cap \Sigma_j$ *it holds that* $(l_i, \lambda!, \varphi_i, \varrho_i, l'_i) \in R_i$, $(l_j, \lambda?, \varphi_j, \varrho_j, l'_j) \in R_j$, *such that* $\nu \models \varphi_i \wedge \varphi_j$, $\nu' = \nu[\varrho_{\{i,j\}} \mapsto 0]$ *and* $\nu' \models \bigwedge_{j=1}^{n} I_j(l'_j)$.

Note that in this definition local steps of a single timed automaton, i. e. edges of timed automata annotated with the ε-action, have been left out. Local transitions are treated as a special case of a discrete transition where only a single automaton is involved. This will ease the description of the concepts introduced in this thesis. The distinction of receiving and sending events is only relevant for timed automata to define participants of communication. For timed transition systems such a distinction is not necessary. The function $l' = l[l_{\{i,j\}} \rightarrow l'_{\{i,j\}}]$ for a location vector $l = (l_1, ..., l_i, ..., l_j, ..., l_n)$ represents the location vector $l' = (l_1, ..., l'_i, l_{i+1}, ..., l'_j, ..., l_n)$, i. e. where only the locations of A_i and A_j are changed. While the set of configurations is generally infinite, Alur and Dill worked out a finite representation which is called region graph [AD94]. In [Dil90, LLPY97] a more efficient data structure called symbolic transition system (or zone graph) was presented. This is the topic of the next subsection.

Symbolic Transition Systems

A *clock zone* represents the maximal set of clock valuations satisfying a corresponding clock constraint. In other words, a clock zone represents a solution set of a corresponding clock constraint. Let $g \in \Phi(C)$ be a clock constraint, the induced set of clock valuations $D_g = \{\nu \mid \nu \models g\}$ is called a clock zone. The intersection between two zones D_1 and D_2 is defined as follows:

$$D_1 \cap D_2 = \{\nu \mid \nu \in D_1 \wedge \nu \in D_2\}. \tag{2.5}$$

Let $D^{\uparrow} = \{\nu + d \mid \nu \in D \wedge d \in \mathbb{R}_{\geq 0}\}$ be the zone where all clocks in D are advanced by d. Let further $D[\varrho \rightarrow 0] = \{\nu[\varrho \mapsto 0] \mid \nu \in D\}$ be the zone where all clocks in ϱ are reset to 0.

The finite representation of a timed transition system is given by a symbolic transition system defined as follows:

Definition 6 (Symbolic Transition System). *Let A be a network of timed automata with pairwise disjoint sets of clocks, such that $C_i \cap C_j = \emptyset$ for $i \neq j \in \{1, ..., n\}$. The symbolic transition system of A is a tuple $STS(A) = (S, S^0, C, \Sigma, \rightarrow)$ where*

- *$S = \{\langle l, D_\varphi \rangle \mid l \in L_1 \times ... \times L_n, \varphi \in \Phi(C)\}$ is the symbolic state set, and $S^0 = \langle l^0, 0_C \rangle$ the initial state with $l^0 = (l_1^0, ..., l_n^0)$,*
- *$C = C_1 \cup ... \cup C_n$,*
- *$\Sigma = \Sigma_1 \cup ... \cup \Sigma_n \cup \{\Delta\}$, where Δ is a dedicated symbol representing the passage of time,*
- *$\rightarrow \subseteq S \times \Sigma \times S$ is the symbolic transition relation with*
  - *$\langle l, D \rangle \xrightarrow{\Delta} \langle l, D^\uparrow \cap D_{I(l)} \rangle$, where $I(l) = \bigwedge_{j=1}^{n} I_j(l_j)$*
  - *$\langle l, D \rangle \xrightarrow{\lambda} \langle l', (D \cap D_{\varphi_i} \cap D_{\varphi_j})[\varrho_{\{i,j\}} \rightarrow 0] \cap D_{I(l')} \rangle$ where $l' = l[l_{\{i,j\}} \rightarrow l'_{\{i,j\}}]$, if there are some $i, j \in \{1, ..., n\}$ with $\lambda \in \Sigma_i \cap \Sigma_j$ such that $(l_i, \lambda!, \varphi_i, \varrho_i, l'_i) \in R_i$ and $(l_j, \lambda?, \varphi_j, \varrho_j, l'_j) \in R_j$.*

Note that all states have always a unique time successor, which is defined by the invariants of the locations. In contrast to the unique time successor, a state may have a set of discrete steps. Instead of using the Δ-symbol, the word *timeFlow* is used in some figures to ease the understanding of the STSs.

In general, the transition relation may lead to infinite number of zones, if unbounded clocks are considered, i.e. clocks which have no upper constraints – also called the clock ceiling – in the states of the corresponding timed automaton. In such cases, the valuation of a clock which has no upper bound constraint in a state may drift away leading to arbitrarily valuations. Thus, for such cases normalization operations on zones are necessary, which map zones containing arbitrarily valuations for such clocks to representatives, which are zones with bounded valuations. These representatives for such clocks are determined based on the maximal values occurring in the clock constraints of the automaton. Such normalization operations have been worked out in the work of [BY04]. Nevertheless, for the purposes of this thesis the above definition is sufficient as always ceilings for all clocks are given in the considered models.

Clock zones have to be in a *canonical* form. A canonical form is given, if no constraint in the considered zone can be strengthened without reducing the solution set of the zone [BY04]. The canonical zones are unique, which is a necessary property to be able to minimize the state representations. If the zones would not be transformed to their canonical forms, infinite number of states of the transition systems would be computed for zones, which represent the same solution set. To compute the canonical form of zones, in [BY04] a graph-based operation to compute the tightest representations of clock constraints has been defined. Clocks represent the nodes of such a graph. Edges between two clocks are given, if a difference constraint is defined between these clocks. The

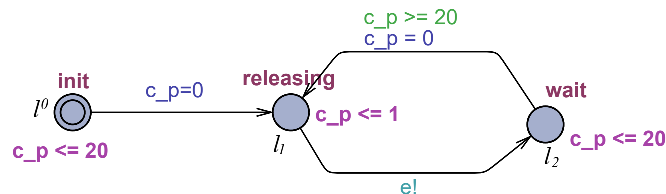


Figure 2.5.: Event stream automaton with a period $p = 20$ and a jitter $j = 1$.

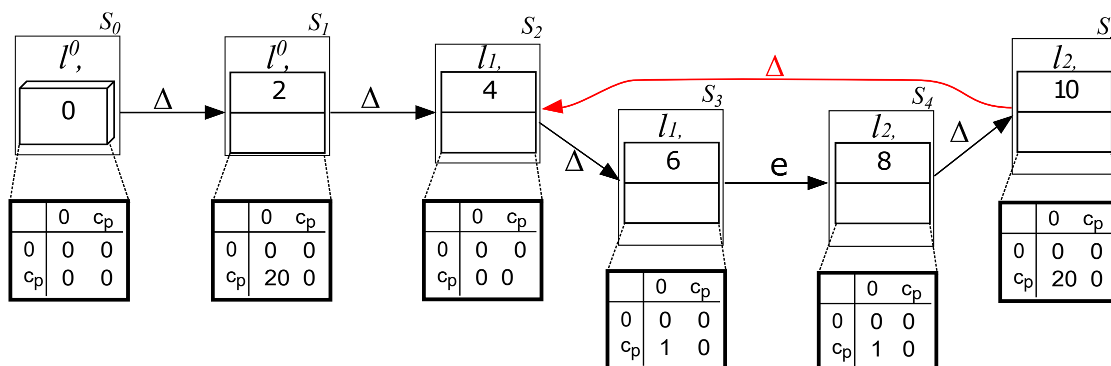


Figure 2.6.: STS for an event stream automaton with $p = 20, j = 1$.

corresponding edges are weighted by the constraint values. The tightening of constraints is then realized by the computation of the shortest path between two nodes. Note that the complexity of this operation is cubic in the number of clocks of a zone.

The DBM (difference bound matrices) representation of zones is used in this thesis. DBMs are matrices capturing the ranges of all clocks and the differences between all clocks. To access the elements of the matrices, the standard notations for matrices are used, i.e. $D_{i,j}$ is the element in the ith row and j^{th} column. Each element in a DBM determines a clock difference, i.e. $D_{i,j} = n$ represents the inequation $c_i - c_j \leq n$ where $c_i, c_j \in C$. The index 0 is always allocated to the reference clock with which it is possible to determine the pure ranges of all clocks. So, $D_{j,0}$ gives the upper bound of clock c_j, and $D_{0,j}$ the lower bound.

An example for an STS of the timed automaton of Figure 2.5 is illustrated in Figure 2.6. The outer rectangles annotated with $S_0, ..., S_5$ represent the states of the STS consisting of a location vector which is illustrated on the top of the inner rectangles. The inner rectangles represent the zones, which are detailed in the rectangles below as DBMs. The automaton of Figure 2.5 triggers the event e with a period of 20 time units with a jitter of 1.

The STS of Figure 2.6 starts in its initial state S_O consisting of the initial location l^0 and the zone where the clock c_p is set to zero. State S_1 represents the time successor, where the clock can progress up to a value within the interval $[0, 20]$.

The location l_1 of the automaton is reached by taking the discrete transition either from *init* to *releasing* or from *wait* to *releasing*. In both transitions the clock c_p is reset and thus the value of this clock in state S_2 of the STS is $c_p = 0$. Note that location l_1 represents the state called *releasing* of the automaton. Analogously, location l_2 represents the state *wait*.

Right before entering state S_2 from state S_1, the clock c_p is within the interval $[0, 20]$ before it is reset. In contrast to this when state S_2 is reached from state S_5 the clock has the value $[20, 20]$ before it is reset. Such pre-reset valuations are not explicitly visible in this STS representation: The original definition of the symbolic transition systems as introduced in Definition 6 discards the information of reached interval values through the reset operation. The originally computed clock ranges are lost as a result of the reset. Such clock ranges will be relevant for the iterative timing analysis introduced in Chapter 3. Because of this, in Chapter 3 a slightly adapted definition for STSs is introduced, which preserves these pre-reset valuations.

2.3. Modeling of System Architectures

In this section the considered system architectures are described and formalized. System architectures consist of a set of components, which have some syntactic interfaces consisting of input ports and output ports. An interface defines the boundary of interactions between the corresponding component and its environment. A component may be further decomposed into a set of interconnected subcomponents or a behavior model. In literature, connections between subcomponents and the overall component are called *delegations*[BBB+11]. Resources like electronic control units (ECUs) or bus systems refine this general notion of component by offering some properties like a scheduling policy.

Next, the notions of component, interface, and resource are formalized, which are the basic ingredients to model system architectures. Designer of such architectures will use some modeling languages which include more domain specific elements. In this thesis, the UML (Unified Modeling Language) profile MARTE is used to model such system architectures. A relevant subset of the MARTE modeling elements is related to the introduced formal elements. This is the topic in Subsection 2.3.2.

2.3.1. Components and Resources

A component M is a tuple $M = (I, in, out, \mathcal{L})$, where I is the interface of the component consisting of a set of input ports *in* and output-ports *out*. The language $\mathcal{L}$ of a component is defined over its interface. On each port $p \in I$ a set of events Σ_p can be observed. Without loss of generality, for each port it is assumed that only one type of event does

occur such that the alphabet of a component is directly defined through its port set I. This is no restriction, as a port with multiple events can be modeled also by creating one separate port for all events. The input and output port sets can also be determined via two functions defined as follows.

Definition 7 (Directed Interface). *An interface I consists of a set of ports. A directed interface partitions I to input ports and output ports. For this, let $\{in, out\} : I \to I$ be two disjoint functions, such that $in(I) \cap out(I) = \emptyset$ and $I = in(I) \cup out(I)$, where $in(I)$ defines the set of input ports and $out(I)$ the set of output ports.*

As introduced in Section 2.2 the language $\mathcal{L}$ of components are sets of timed words over its interface I. The set of all possible timed words over an interface I is given by I^ω.

When a set of components is considered, projection operations of the corresponding languages are necessary to correctly define the composition operations of these components.

Definition 8 (Restriction of Words). *Let $\sigma = (\rho, \tau)$ be a timed trace over the interface I and let $I' \subseteq I$. The restriction of σ to I', in short $\sigma_{\downarrow_{I'}} = (\rho, \tau)_{\downarrow_{I'}}$, is the trace σ in which tuples containing events from the set $I \backslash I'$ are left out.*

The restriction of a timed language is then defined in a straightforward manner.

Definition 9 (Restriction of Languages). *Let $\mathcal{L}$ be a language over the interface I and let $I' \subseteq I$. The restriction of $\mathcal{L}$ to I' is defined as follows:*

$$\mathcal{L}_{\downarrow_{I'}} := \{\sigma' \mid \exists \sigma \in \mathcal{L}.\ \sigma_{\downarrow_{I'}} = \sigma'\}.$$

Definition 10 (Extension of Languages). *Let $\mathcal{L}$ be a language over the interface I' and let $I' \subseteq I$. The extension of $\mathcal{L}$ to I is defined as follows:*

$$\mathcal{L}_{\uparrow_I} := \{\sigma \mid \sigma \in I^\omega.\ \sigma_{\downarrow_{I'}} \in \mathcal{L}\}.$$

Note that for each possible time sequence a word is given in I^ω. Consider two disjoint alphabets and a timed language which is defined on one of these alphabets. The extension operation is defined in such a way that all possible timed words over the second alphabet are added to the language. The following lemma formalizes this property.

Lemma 1. *Let I_1, I_2 be two port sets with $I_1 \cap I_2 = \emptyset$, and $\mathcal{L}$ be a non-empty language over I_1. Then $(\mathcal{L}_{\uparrow_{I_2}})_{\downarrow_{I_2}} = I_2^\omega$, i. e. the set of all words over I_2.*

The proof of this lemma is straightforward by applying the extension and restriction operations successively.

A *resource* inherits the characteristics of a component and adds further properties. A resource r is modeled by the tuple $r = (M, \mathcal{T}, Sch, \mathcal{R}, \mathcal{A})$, where M is a component as

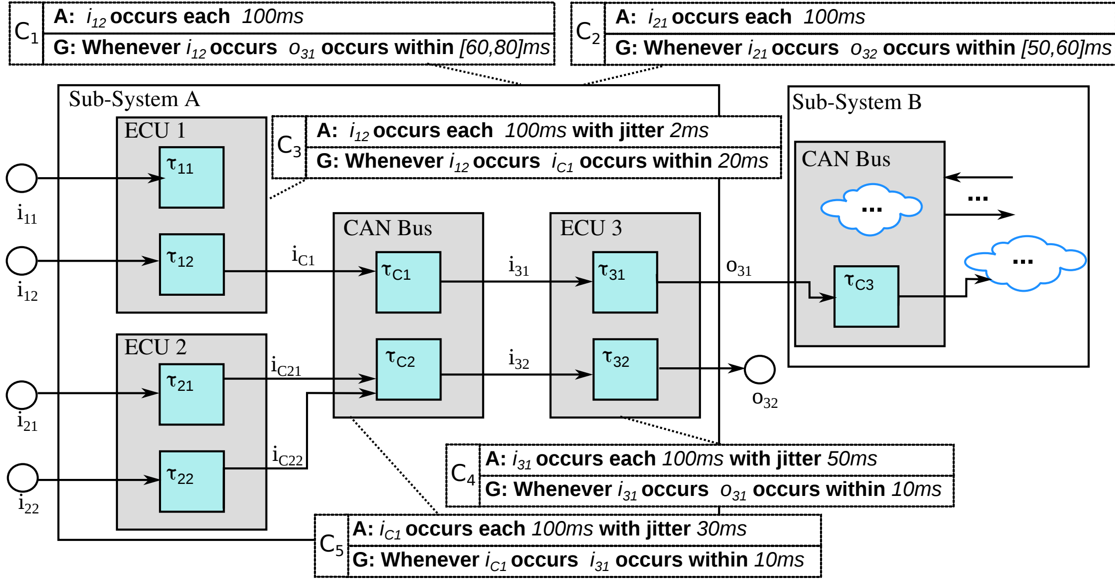


Figure 2.7.: System architectures of interest.

defined above. A mapping $\mathcal{T} : \mathsf{T} \to \mathbb{B}$ determines the set of tasks that is allocated to a resource. For each resource, a scheduling policy *Sch* is given such as *First Come First Served* (FIFO), *Fixed Priority Scheduling* (FPS) with and without preemption, or *Time Division Multiple Access* (TDMA).

Two additional functions provide dynamic book keeping needed to perform scheduling analysis. These functions depend on the state of the resource and contain the following information:

- A ready list $\mathcal{R}$ determining tasks that are released but to which no computation time has been allocated up to now.
- An active task map $\mathcal{A} : \mathsf{T} \to [t_1, t_2]$ with $t_1, t_2 \in \mathbb{N}_{\geq 0}$, which determines the interruption times of tasks. This map is ordered and the first element determines the current running task.

Composing two components results again in a component. This is defined in the following.

Definition 11 (Composition of Components)**.** *Let M_1, M_2 be two components. Then the composition of both is again a component $M = (I, in, out, \mathcal{L})$ with $I = I_1 \cup I_2, out = out_1 \cup out_2, in = I \backslash out$, and $\mathcal{L} = \mathcal{L}^1_{\uparrow_I} \cap \mathcal{L}^2_{\uparrow_I}$.*

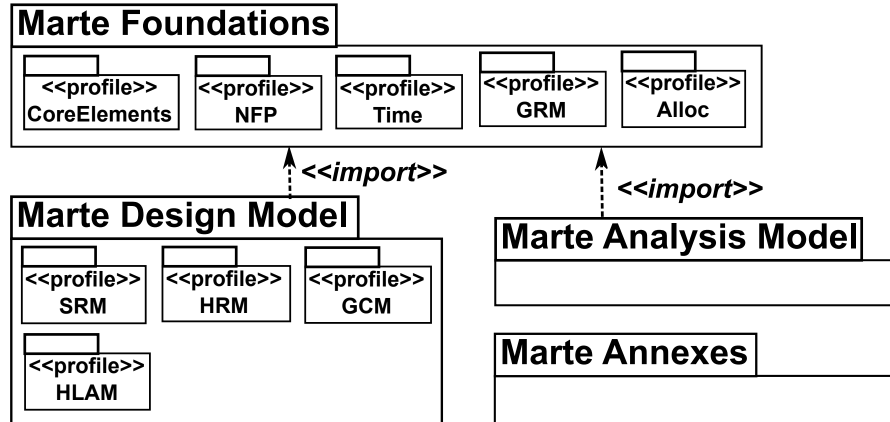


Figure 2.8.: Overview of the MARTE profile.

A system architecture $a = (K, \mathsf{T})$ (also referred to as system) is defined by a set of composed components $K = (I, in, out, \mathcal{L})$ (which is again a component itself), and a task set T.

Consider for example the architecture of Figure 2.7, which consists of two components *Sub-System A* and *Sub-System B*. The interface of component *Sub-System A* consists of the input ports $i_{11}, i_{12}, i_{21}, i_{22}$ and the output ports o_{31}, o_{32}. The *Sub-System A* consists of three ECUs interconnected by a CAN bus. On each resource, two tasks are allocated respectively.

2.3.2. Modeling in MARTE

To specify models in a reusable and interchangeable manner it is desirable to use domain specific modeling standards. To annotate the model with relevant resource and timing properties the OMG MARTE (Modeling and Analysis of Real-Time and Embedded Systems) profile [OMG11] is applied.

The general overview of the MARTE profile is illustrated in Figure 2.8. The *Foundations* package contains elements to describe time and the use of concurrent resources. The *Design Model* package consists of artifacts describing software and hardware features of real-time systems, while the *Analysis Model* package is concerned with annotations to support analysis of system properties. The *Annexes* package contains predefined model libraries. In the following, the packages and elements are detailed, which are of interest for the purposes of this thesis. The interested reader may find more detailed information in the OMG MARTE standard [OMG11].

`GRM` stands for *Generic Resource Modeling* and offers basic concepts to model systems which execute real-time tasks. The *Hardware Resource Modeling* (`HRM`) and *Software*

Resource Modeling (SRM) packages refine the GRM package. The HRM package contains more detailed hardware parameters like cache sizes or frequency information. This level of granularity is not necessary for the early analysis phases considered in this thesis where only *ECUs* and bus systems are considered on a high abstraction level.

From the SRM package the SW_Concurrency sub-package is considered in this thesis. This sub-package includes concepts with which entities competing for the usage of a shared resource can be modeled.

In the following, the specific language artifacts which are applied are described in detail.

Resources To model *ECUs* the element ComputingResource from the GRM package is used. This entity enables to allocate a scheduler to a computing resource. For buses the element CommunicationMedia from the same package is used. Note that also elements from the more specific HRM package could be used. For instance, the element HWBus could be used to model a bus system. Analogously, HWProcessor could be applied to model an *ECU*. These artifacts contain detailed hardware information such as *bandwidth* or *instruction per second* value parameters. However, this thesis targets the analysis of early design stages, where such hardware details are not available. Thus, considering the more abstract modeling elements mentioned above is sufficient for the purpose of this thesis.

Scheduler A *scheduler* is modeled by the concept of Scheduler from the GRM package. A scheduler has the properties *isPreemptable*, *schedulingPolicy* (where a set of predefined policies can be configured), and *processingUnit* with which the corresponding resource can be allocated.

Tasks A *task* is modeled by using the following entities:

- ResourceUsage from the GRM package: Here, the relevant properties are the *execution time*, which can be either a single value corresponding to the worst-case execution time or an interval where the lower bound represents the best-case execution time. Additionally, the property *used resource* is relevant for our purposes. With this, tasks can be allocated to resources.

- SWSchedulableResource from the SW_Concurrency package: The properties *is preemptable*, *resMult*, *priority elements*, and *period elements* are considered in this thesis. The property *priority elements* is used to allocate a priority to the task, *is preemptable* states whether the task could be preempted by another task allocated on the same resource, and *period elements* is used to encode the event streams with which independent tasks are triggered. The value given in the property *period elements* is interpreted as an event model as introduced in the previous section.

When analyzing the dynamic behavior of the modeled resources, the maximum number of possible parallel activations of each task is relevant. This information is entered in the property *resMult*.

- Dependencies of tasks are specified through the usage of the UML *dependency* relations.

End-to-end latency constraints between a set of tasks and timing constraints of single tasks are specified utilizing the RSL-language, which are detailed in the next section.

2.4. Specification of Requirements

Specifications of requirements can be performed in various manner. Early requirements are typically captured in natural language. Unfortunately, natural language tends to be ambiguous which could lead to misunderstandings between the OEM, suppliers, or special departments within a company, and to problems in the integration stage of system parts. For example, interfaces do not fit together as different value types or ranges were expected, or an implementation does not deliver the results which were originally expected.

Thus, a formalization of these early requirements is necessary to omit such problems. For this, many approaches can be found in literature as for example the *Linear Temporal Logic* (LTL) [Pnu77], which extends the propositional logic by a set of temporal modal operators. As these formalisms are quite hard to understand and to use correctly in context, more intuitive graphical formalisms were defined like the *Live Sequence Charts* (LSCs) where the interaction between system components can be specified in an unambiguous way [DH01]. Text-pattern-based formalisms were worked out like the *Requirement Specification Language* (RSL) which will be the topic of Subsection 2.4.2.

Besides the formalization of requirements, it is advisable to distinguish between assumed and guaranteed behavior. To require for example that a component port is triggered periodically does not lie in the responsibility of the component itself but in the responsibility of its context. Such a distinction eases the integration phase of components, as an automatic verification of consistency of ports can be performed. For this purpose the contract-based design paradigm inspired by Bertrand Meyer's programming language Eiffel [Mey92] was worked out. The work of Bertrand Meyer enriched object oriented programming by using pre- and post-conditions. Later works such as [BCF+08, BDH+12, BCN+11] generalized the contract theory and made it applicable for model-based design. In [BCN+11] a meta-theory of contracts is developed and different concrete contract theories are cast into this framework like the one of [BCF+08], benefiting from properties that are already established on the level of the meta-theory. Also in today's programing languages contracts are used. For example a contract definition is included in the *Bean Validation* specification of the Java Enterprise Edition.

Methods of classes are annotated with constraints over the input parameters and return values. Also a restricted form of refinement is applied: Inheriting classes have to refine the annotated contracts of the corresponding root class. For this, the constraints over the return values of the included methods are allowed to be added and be stricter, while constraints on the method parameters are only allowed to be defined on the root class level. As detailed later, this is consistent with the general contract refinement definition. Contracts and all necessary operations on contracts are the topic of the next subsection.

2.4.1. Contract-based Design

The idea of contracts was originally inspired by Bertrand Meyer's programming language Eiffel and its *design by contract* paradigm [Mey92]. Requirements of systems are typically specified as assertions, which are basically properties that may or may not be satisfied by some behavior [BCF+08]. Formally speaking, an assertion defines a set of traces over a set of variables (or ports), i. e. an assertion is a (timed) language. Let M be an implementation of a component, which again defines a set of traces, E a corresponding assertion, and let M and E be defined over the same set of ports. Then M satisfies E if $M \subseteq E$.

Contracts define properties of component behaviors and their environments, thus are assertions for component interfaces. Contracts follow the principle of separation of concerns. More specifically, contracts are pairs consisting of an assumption (A) and a guarantee (G). The assumption specifies how the context of the component, i. e. the environment from the point of view of the component, should behave. Only if the assumption holds, then the component will behave as guaranteed. This kind of specification allows to replace components by more detailed ones, if they allow a more abstract environment, without re-validating the whole system. Thus, the system decomposition can be verified with respect to contracts without the knowledge of the concrete implementation.

Contracts help to explicitly specify assertions about the necessary context of components, such that implicitly assumed behavior leading to integration errors between different development teams can be avoided. Each development team has only to rely on the specification of components of other teams, i. e. these can be treated in a black-box manner, as the internal behavior is not of interest. Especially for OEM-supplier relations this has a great benefit, as the OEM typically does not know details about implementations of components from the suppliers.

Moreover, the usage of contracts can help to decrease the complexity of verifying the implementation against its specification. Such a distinction of assumed and guaranteed behavior helps to perform more compositional analyses rather than holistic analyses. Internal implementation details of other components, on which the component under verification relies to, need not to be considered. For example, consider the system in Figure 2.9, to which a contract is assigned. The system contract states that the input port 'a' is triggered each 50ms, and when it is triggered, the system has to respond by

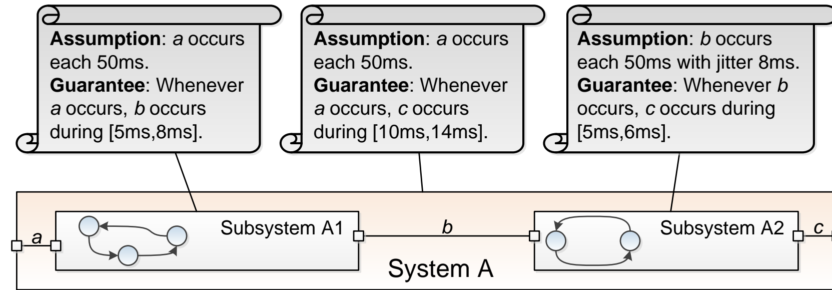


Figure 2.9.: Example for a contract specification.

sending an event on port 'c' within the specific time interval between 10 and $14ms$. The system is decomposed into two subsystems each with one contract, and some internal behavior modeled by, e. g. state machines. Assume that the functionality on subsystem A2 depends on the output of subsystem A1. Further, assume that the subsystems would *not* be annotated with contracts. Thus, to validate the contract of the overall system A, the composed behavior of both subsystems has to be computed, which generally leads to large state spaces. Using contracts for A1 and A2, the computation of the composed behavior of both systems can be omitted and the sub-contracts can be validated locally.

By using contracts a compositional verification technique is enabled. Verifications can be performed locally by explicitly stating the necessary assumptions on the contexts. Besides the verification of the local subcomponents, by using contracts some further proof obligations have to be performed:

- *Satisfaction*: Does the implementation of a component satisfy the contracts of the corresponding component? The implementation may be some kind of a state automaton or program code.

- *Consistency*: Are the interface specifications of all components consistent? For example, is the assumption of the contract of Subsystem A2 of Figure 2.9 consistent with the guarantee of the contract of Subsystem A1?

- *Refinement*: Does the composition of all sub-contracts refine the contract of the overall component? For example, does the composition of the contracts of Subsystem A2 and A1 refine the contract of System A?

For this, the notions of contract-refinement and -composition are needed, which are the topic of the next paragraphs.

Semantics of Contracts

A contract C is a pair (A, G), where A, G are assertions over an alphabet Σ. Formally, the semantics of a contract is defined as

$$[\![C]\!] := [\![A]\!]^{Cmpl} \cup [\![G]\!], \tag{2.6}$$

where $(X)^{Cmpl}$ defines the complement of a set X in some universe $\mathcal{U}$, and $[\![X]\!]$ is defined as the semantic interpretation of X. In our case, $[\![X]\!]$ is given in terms of sets of timed words over some alphabet. The alphabet is given through the interface of the corresponding component.

For the sake of readability, the semantic braces are omitted in the following. These set theoretic notation can also be formulated as logical Boolean formulas in an equivalent manner. Concerning the semantics of contracts the logical notation is in the form of $C = \neg A \vee G$. Whenever needed, either the set theoretic or the logical formula notation is used.

An implementation of a component satisfies its contract, if its behavior is a subset of the behavior defined by the contract.

Definition 12. *Let $C = (A, G)$ be a contract defined over a component interface I and M be an implementation defined over the same interface. The implementation satisfies the contract, in short $M \models C$, if $[\![M]\!] \subseteq [\![C]\!]$.*

The set relation of this definition between implementation and contract can also be re-formulated to $M \cap A \subseteq G$. Thus, if there is some behavior of the implementation and allowed or not constrained by the assumption, it should also be part of the guarantee. If not, the implementation would violate the guaranteed behavior of the corresponding component.

A contract (A, G) is consistent if it is implementable, i.e. it guarantees some behavior, in short $G \neq \emptyset$. Each contract has a unique *maximum* implementation in the sense of language inclusion, which is $M_{max} := \neg A \vee G$. Every possible implementation which satisfies its contract is thus a subset of M_{max}.

Contract Composition

The composition operation on contracts is a necessary operator for component-based designs, as a set of components are connected to build up more complex components. Thereby, connected components interact via their ports.

To correctly create such composition structures, certain composition criteria have to be fulfilled. These composition criteria concern syntactical aspects such as types of values on ports which have to match, or the direction of ports, i.e. output ports have to be connected to input ports, and components must not *share* output ports. Note that value types on ports are not of interest in this thesis. More abstract event triggers are

considered, as the focus here are not functional analyses, but the analysis of timing of event occurrences. Such structural composability aspects are not the focus of this work and will be not detailed in the following. The focus will be rather on composition criteria concerning the semantical aspects of components given by their specifications. More details for composition criteria can also be found in [BCN+11].

The structural composition was already been given in Definition 11. The composition of the interface specifications has to be defined in the following, i. e. the composition on the specification level. In [BCN+11] the criteria for a correct composition were stated. A contract composition operation must be therefore *associative*, i. e. for contracts C_1, C_2, C_3 it must hold that $(C_1 \otimes C_2) \otimes C_3 = C_1 \otimes (C_2 \otimes C_3)$, and *commutative*, i. e. $C_1 \otimes C_2 = C_2 \otimes C_1$. These criteria have to be fulfilled as composing components in different order shall lead to the same result.

The definition of contract composition of [Hun11] is applied in this thesis.

Definition 13 (Composition of Contracts)**.** *Let $C_i = (A^i, G^i)$ for $i = 1, \cdots, n$ be a set of contracts over the interfaces I_i. The composition $C = (A_\otimes, G_\otimes) = C_1 \otimes \cdots \otimes C_n$ over the interface $I = \bigcup_i I_i$ is defined as follows:*

$$A_\otimes = \bigcap_i A^i_{\uparrow_I} \cup \bigcup_i (A^i \cap \neg G^i)_{\uparrow_I},$$
$$G_\otimes = \bigcap_i G^i_{\uparrow_I}.$$

This definition is obtained from the conjunction of the corresponding contracts. This is shown for two contracts in the following.

$$\begin{aligned} C_1 \wedge C_2 = \; & (\neg A_1 \vee G_1) \wedge (\neg A_2 \vee G_2) \\ = \; & \neg(A_1 \vee A_2) \vee (\neg A_1 \wedge G_2) \vee (\neg A_2 \wedge G_1) \vee (G_1 \wedge G_2) \\ = \; & (A_1 \vee A_2) \wedge (A_1 \vee \neg G_2) \wedge (A_2 \vee \neg G_1) \Rightarrow G_1 \wedge G_2 \\ = \; & (A_1 \wedge A_2) \vee (A_1 \wedge \neg G_1) \vee (A_2 \wedge \neg G_2) \Rightarrow G_1 \wedge G_2 \end{aligned}$$

The above composition definition corresponds also to the definitions of [BCF+08, BFM+08, QGP10] but differs from the formulas in [BCN+11], since it does not rely on contract saturation. That is, for a contract C, it is not required that $G = \neg A \cup G$ does hold. Contract saturation is detailed in Section 4.4.

Contract Refinement

An important property of contract-based design is *refinement*. The intention of refining a property is that previously specified behavior gets more concrete. As an example, the order of a set of events on the output of a component is not known in an early stage of a design, as the scheduler is not defined so far. So, all possible orderings are considered. In a later design step, when a fixed schedule has been defined, the output can be refined to a specific order of events. Another example is the allocation of a task to a resource. If this allocation is not determined in an early design phase, a coarse interval for its execution time has to be assumed to keep the flexibility to allocate this task to different resources. The time the allocation is determined, this coarse interval is refined in the sense that it

gets tighter. The refined components have to respect the requirements specified for their abstract counterparts.

A contract C' *refines* another contract C, if the assumption of C' is less restrictive than the one of C, and the overall specified behavior of C' is more restrictive than the behavior permitted by C. In the following, the refinement relation with respect to sets of traces is formalized.

Definition 14 (Refinement)**.** *Let $C = (A, G)$, $C' = (A', G')$ be contracts over the same set of ports I. C' refines C, in short $C' \preceq C$ or $C' \Rightarrow C$ as logical formula, if and only if*

$$[\![A']\!] \supseteq [\![A]\!] \text{ and } [\![C']\!] \subseteq [\![C]\!].$$

The second condition can be simplified as follows.

Lemma 2. *Let $C = (A, G)$, $C' = (A', G')$ be contracts over the same set of ports I. The second condition of the refinement condition of Definition 14 $[\![C']\!] \subseteq [\![C]\!]$ can be simplified to $[\![A]\!] \cap [\![G']\!] \subseteq [\![G]\!]$, if the first condition $[\![A']\!] \supseteq [\![A]\!]$ holds.*

Proof. $\neg A' \vee G' \Rightarrow \neg A \vee G \quad \Leftrightarrow \quad (\neg A' \vee G') \wedge A \Rightarrow G \quad \overset{(A \Rightarrow A')}{\Rightarrow} \quad A \wedge G' \Rightarrow G$ □

The last step follows, as $\neg A' \wedge A = false$, if $A \Rightarrow A'$. From this definition of refinement it can be directly followed that if an implementation M satisfies its contract C and $C \preceq C'$ then M also satisfies C'.

Virtual Integration Condition and Check

When components are decomposed to a set of interconnected subcomponents, it has to be checked whether the contracts of all sub-contracts "fit together", i.e. are composable with respect to Definition 13, and the composition of these sub-contracts *refines* the contract of the overall component. Both proof obligations are summarized in a single condition called the *virtual integration* which was originally introduced in [DHJ+11].

Lemma 3 (Virtual Integration Condition)**.** *Let $C, C_1, ..., C_n$ with $C = (A, G)$ and $C_i = (A_i, G_i)$ for $i = \{1, \cdots, n\}$ and $n \in \mathbb{N}_{>0}$ be contracts. The contracts $C_1, ..., C_n$ can be virtually integrated in the context of C if the following holds:*

i) $A \wedge C_1 \wedge ... \wedge C_n \quad \Rightarrow \quad A_1 \wedge \cdots \wedge A_n,$
ii) $A \wedge G_1 \wedge ... \wedge G_n \quad \Rightarrow \quad G.$

The first condition states that the assumptions of the ith subcomponent must follow from the contracts of the other subcomponents and the overall assumption. The second condition states that the contract of the overall component must follow from the subcontracts of the composition structure. This condition is derived from the composition operation and refinement relation introduced in Subsection 2.4.1. This is proved in the following.

Proof. Let $C = (A, G)$ be a contract of a component, which is decomposed into a set of components, to which sub-contracts are allocated. Let $C' = (A', G')$ be the contract resulting by the composition of this set of sub-contracts according to Definition 13. According to the refinement relation of Definition 14 both a) $A \subseteq A'$, and b) $A \wedge G' \subseteq G$ must hold. The derivation of i) is mainly to substitute $A' = \bigcap_i A^i_{\uparrow_I} \cup \bigcup_i (A^i \cap \neg G^i)_{\uparrow_I}$, and reformulate the expression to $A \cap \bigcap_i (\neg A^i \cup G^i)_{\uparrow_I} \subseteq \bigcap_i A^i_{\uparrow_I}$. The second condition is derived by substituting G' accordingly. □

Observe that the sub-condition *i*) includes negations, which complicate the application of verifications. This is especially a problem for specification formalisms which are not closed under the complementation operation such as timed automata, which shall be applied in this thesis. In Chapter 4 (or more precisely in Section 4.4), this issue will be targeted by inspecting some characteristics of contracts.

2.4.2. Requirement Specification Language

To specify the assumption and guarantee part of contracts natural language could be used, but this has disadvantages: Natural language is ambiguous and inconsistent requirements could be specified. Moreover, an automatic analysis would not be applicable. If simulations or analyses cannot be performed, failures in the system and inconsistencies of its specification may not be detected until late phases of the development process. These failures may lead to delays in the development process and high production costs. Formal languages could solve these problems, but in general are hard to understand and therefore difficult to use.

To cope with the unintuitive specification form of formal languages, pattern-based languages can be used: Here, a set of standard text patterns, which consist of static text elements with fixed semantic interpretations are specified. With such a kind of specification an intuitive usage is given while the usage of automated validations and verifications is still possible. In this work, a specific pattern-language called *Requirement Specification Language* (RSL) [RSRH11, Pro07] is used. The RSL consists of text-pattern for different aspects of a system design like safety, functionality, or real-time. In particular, the focus of this thesis is on the real-time pattern specified in this language.

The basic elements occurring in the text patterns are *events*, *sets of events*, and *time intervals*. Events are signals that occur at a defined single point in time and have no duration, e.g. periodically sent values from sensors or messages sent over a bus system. All names of events in a set of text-pattern have to be used consistently, i. e. same events must have the same identifiers.

A time interval is defined through two points in time. These points in time refer to either timed values or observable events. Using timed values, the end-points of an interval may be *inclusive* or *exclusive*. An inclusive end-point of an interval specifies that this

point in time belongs to the interval, an exclusive end-point indicates that the point in time does not belong to the interval.

If events are used instead of timed values, an interval is defined through two events [*startEvent, endEvent*]. The first occurrence of an event *startEvent* then opens the interval and the corresponding *endEvent* closes the interval.

In the following, the text-pattern from the RSL are introduced, which capture the timing behavior that is of interest for this thesis.

R1-Pattern - Periodic Activation

The R1-Pattern pattern describes the periodic occurrence of an event such as the activation of a task. The occurrence of each event can be delayed by an additional jitter. The patter is given in the following:

event `occurs each` *period* [`with jitter` *jitter*].

The semantics of this pattern is derived form the work in [RSRH11]. Let p be the period, and j the jitter specified in an instance of this pattern.

$$\mathcal{L}_{R1} = \{(\rho, \tau) \mid \exists \tau', \forall i \in \mathbb{N}.\ \tau'_0 \in [0, p] \wedge (\tau'_i \leq \tau_i \leq \tau'_i + j) \wedge (\tau'_{i+1} - \tau'_i = p)\} \qquad (2.7)$$

The additional time sequence τ' is needed to fix the periodical grid, without being affected by the jitter. If τ' would be left out, one would get some over-approximations for the timing behavior. Assume the semantics would have been by using the following condition:

$$\forall i \in \mathbb{N}.\ \tau_0 \in [0, p + j] \wedge (\tau'_{i+1} - \tau'_i \in [p - j, p + j]$$

Assume that $p = 30, j = 5$. Then the sequence $(0, 35, 70, 105, ...)$ would be allowed, which contradicts the original semantics, as the jitter is accumulated.

An example for this pattern could be the specification of a diagnosis task, which shall be executed every 100ms and may jitter 10ms. The pattern would be instantiated as follows:

`diagnosisTaskActivate occurs each 100ms with jitter 10ms`.

This pattern is a characterization of the previously discussed *periodic with jitter* event model of Figure 2.3. Note also that in [RSRH11] the condition $\tau'_0 \in [0, p]$ is missing in the definition of the semantics of this pattern. Without this condition the occurrence of the first event larger than the period would be allowed. This would not reflect the defined behavior of occurrence curves. Because of this, the original condition has been extended in this thesis as illustrated above.

R2-Pattern - Sporadic Activation

The R2-Pattern is a generalization of the R1-Pattern and defined as follows.

event `occurs sporadic with minperiod` *period* [`and maxperiod` *period*] [`and jitter` *jitter*].

This pattern describes the sporadic occurrence of an event. The minimal distance of occurrences is defined by *minperiod*, which can be delayed by an additional jitter. The *maxperiod* specifies the maximal distance between two successive occurrences. Note that if the upper bound of occurrences is not given, the maximum distance between two successive events could be arbitrary thus leading to an unbounded system. This would correspond to a liveness property, for which it is well known that these are hard to analyze. In this work, the usage of this pattern is restricted in such a manner that the upper bound is always specified.

The semantics of this pattern was introduced in [RSRH11]. Let p_{min} be the minperiod, p_{max} the maxperiod, and j the jitter specified in an instantiation of this pattern.

$$\mathcal{L}_{R2} = \{(\rho, \tau) \mid \exists \tau', \forall i \in \mathbb{N}.\ \tau'_0 \in [0, p] \wedge (\tau'_i \leq \tau_i \leq \tau'_i + j) \wedge (p_{min} \leq \tau'_{i+1} - \tau'_i \leq p_{max})\} \tag{2.8}$$

As in the case of the R1-pattern, the additional time sequence τ' is needed to prevent over-approximations occurring. Note that also for this case the condition $\tau'_0 \in [0, p]$ is missing in the original definition of [RSRH11].

An example for this pattern could be the specification of a processing of received data, which must be performed at most every 50ms and at least every 100ms. The instantiated pattern would be as follows:

`dataReceived occurs sporadic with minperiod` 50ms `and maxperiod` 100ms.

R3-Pattern - Delay between Events

The R3-pattern describes the dependency between the occurrence of two events. It is defined as follows.

`Whenever` *event* `occurs,` *event* `occurs [during` *interval*`]`.

The first event of this pattern specifies a trigger, to which always a response shall follow. The optional interval of the pattern specifies that the response event shall occur in the corresponding time bounds after the trigger event was received. In this work, the usage of this pattern is restricted in such a fashion that the interval is always given. If no interval would be specified, a liveness property would be obtained.

The semantics of this pattern over the alphabet $\Sigma = \{e_1, e_2\}$, where e_1 is the triggering event and e_2 the response, is the following.

$$\mathcal{L}_{R3} = \{(\rho, \tau) \mid \forall i \in \mathbb{N}\ \exists j > i.\ (\rho_i = e_1) \Rightarrow (\rho_j = e_2) \wedge (\tau_i + lb) \leq \tau_j \leq (\tau_i + ub)\} \quad (2.9)$$

Note that $lb, ub \in \mathbb{N}$ are the lower and upper bounds of the pattern, where $lb \leq ub$. Thus, the semantics of this pattern allows to receive multiple triggering events before a single response is sent.

An example for this pattern would be the response time of a task, which shall not exceed 10ms. The instantiated pattern would be as follows:

`whenever` ***taskStart*** `occurs,` ***taskEnd*** `occurs during [0ms, 10ms].`

Extensions of R3-Pattern - Delays for Sets of Events

Instead of specifying the distance of single events, also the distance between two event sets can be specified. The pattern is extended as follows.

`Whenever` set$\{event_n$, ..., $event_m\}$ `occurs,` set$\{event_k$, ..., $event_s\}$ `occurs [during` *interval*`].`

The idea is that whenever all events of the first set have been received, all events of the second set shall occur within a specific time interval.

To define the semantics, consider the following instantiated pattern:

`Whenever` set$\{e_1, e_2\}$ `occurs,` set$\{e_3, e_4\}$ `occurs during` *[lb,ub]*.

The language results than in the following.

$$\mathcal{L}_{R3,set} = \{(\rho, \tau) \mid \forall i \in \mathbb{N}\ \exists j, k, l \in \mathbb{N}. i < j < k < l\ \wedge\ (\rho_i = e_n) \wedge (\rho_j = e_m) \wedge n \neq m \Rightarrow (\rho_k = e_p) \wedge (\rho_l = e_q) \wedge p \neq q \wedge (\tau_j + lb) \leq \tau_l \leq (\tau_j + ub)\} \quad (2.10)$$

where $n, m \in \{1, 2\}, p, q \in \{3, 4\}$.

As an example, consider that for a distance control system of a car the two sensor data *speed of rotation* and *distance to front car* have to be received in order to regulate the speed and some feedback to the driver:

```
whenever set{rotationSpeed, distanceFrontCar} occurs, set{regulateSpeed,
            driverFeedback} occurs during [10ms, 50ms].
```

Note that using this extended version of the pattern the problem of receiving multiple events of the same type occurs. For instance, *rotationSpeed* occurs twice while *distance-FrontCar* has not been received so far. There are various techniques in literature to deal with these problems. The most pragmatic way which is applied in this thesis is to ignore multiple occurrences. The more complex solution is to deal with a *bounded* number of multiple events and to service this maximal number.

The following pattern is defined in an analogous manner as the previous one. The difference is that the events shall occur in an ordered way.

`Whenever` $event_n$ `and then` ... `and then` $event_m$ `occurs,` $event_k$ `and then` ... `and then` $event_s$ `occurs [during` ***interval***`].`

The semantics of this pattern is a consecutive extension of expression in Formula 2.9 by replacing the single event e_1 by the conjunctions of the corresponding triggering events, i.e. by the expression $event_n \wedge ... \wedge event_m$, and analogously e_2 by the expression $event_k \wedge ... \wedge event_s$.

Consider again the above example:

`whenever` ***rotationSpeed*** `and then` ***distanceFrontCar*** `occurs,` ***regulateSpeed*** `and then` ***driverFeedback*** `occurs during [10ms, 50ms].`

If the event *distanceFrontCar* would occur before the event *rotationSpeed*, this sequence would not trigger the right hand side.

R4 Pattern - Distance between events

The R4-pattern is closely related to the basic R3-pattern and defined as follows.

`Distance between` $event_1$ `and` $event_2$ `within` ***interval***`.`

The only difference to the R3-Pattern is that there is no order between both events occurring in the pattern. Thus, there are no dedicated triggering and response events. If one of the events occurs, the other has to occur within the defined interval.

Let $n, m \in \{1, 2\}$. The semantics of this pattern over the alphabet $\Sigma = \{e_1, e_2\}$ is the following.

$$\mathcal{L}_{R4} = \{(\rho, \tau) \mid \forall i \in \mathbb{N}\ \exists j > i.\ (\rho_i = e_n) \Rightarrow (\rho_j = e_m) \wedge n \neq m \wedge (\tau_i + lb) \leq \tau_j \leq (\tau_i + ub)\} \tag{2.11}$$

Note that $lb, ub \in \mathbb{N}$ are the lower and upper bounds of the pattern, where $lb \leq ub$. Also here, the semantics allows to receive multiple triggering events before a single response is sent. This can handled analogously to the techniques illustrated for the R3-Pattern. As for R3-Pattern, multiple activations are ignored in this thesis.

As an example, consider the distance of two sensor data:

`Distance between` *rotationSpeed* `and` *distanceFrontCar* `within` $[5, 10]ms$.

Whenever one of the two events *rotationSpeed* or *distanceFrontCar* occurs, the other event has to occur within the defined time interval.

2.5. Summary

In this chapter, all necessary ingredients for a model-based design of real-time systems have been introduced and the underlying semantics were discussed. The specification of tasks and the underlying architecture consisting of inter-related components and resources was formalized and the modeling by using Marte was discussed. Requirements of architectures and their decomposition structures are specified using the pattern-based language called Requirement Specification Language (RSL). The distinction of assumed and guaranteed behaviors was discussed by using the notion of contracts. By using contracts proof obligations arise, including the correctness check of decomposition structures, i. e. the check whether the contract of an overall component follows from the contracts of its parts, and whether all contracts “fit together”. Further, the satisfaction of the implementation of a component or resource has to be checked, i. e. it has to be verified whether the implementation in terms of some automata structure or program code fulfills the corresponding contracts.

3. State-based Timing Analysis

3.1. Motivation

The choice of the scheduling policy of a shared resource has a direct influence on the timings of the allocated tasks and thus on the performance of architectures in terms of response times and end-to-end latencies. To guarantee that all timing constraints are fulfilled, analyses have to be performed, especially when *hard* real-time systems are considered, where violations of deadlines could lead to very high costs or even threats to human life.

Analytical analysis approaches consider the worst-case scenario of a resource which is also called the *critical instance*. This scenario is given, when all tasks allocated to the corresponding resource are activated simultaneously. For the analysis of local resources this results in adequate response times. Unfortunately, these approaches tend to result in pessimistic response times when distributed systems are considered.

Consider the system in the left part of Figure 3.1 consisting of two resources $R1$ and $R2$ on which two tasks are allocated respectively. The terminations of τ_1 and τ_2 do trigger τ_3 and τ_4 respectively. To activate τ_3 and τ_4 (nearly) in parallel, a task on $R1$ has to be interrupted right before its termination. If $R1$ is assumed to be a bus system, this scenario would not occur in real, as interrupts cannot occur on a bus system. Still assuming the worst-case activation scenario of the tasks allocated to $R2$ by the tasks on the bus $R1$ results in pessimistic response times. Note that in real life of course tasks cannot be allocated to bus systems. In this work, tasks are considered on bus systems to model the timing behavior in an abstract manner and handle such resources analogously to ECUs.

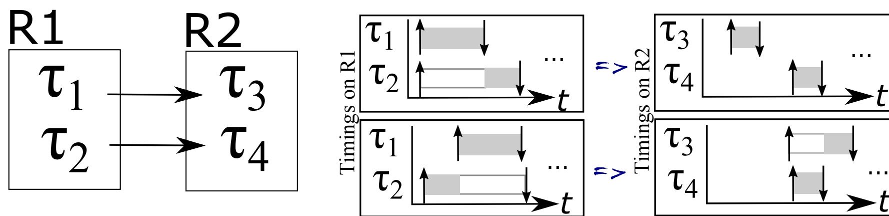


Figure 3.1.: Scenarios of possible timings (right) of the distributed system (left).

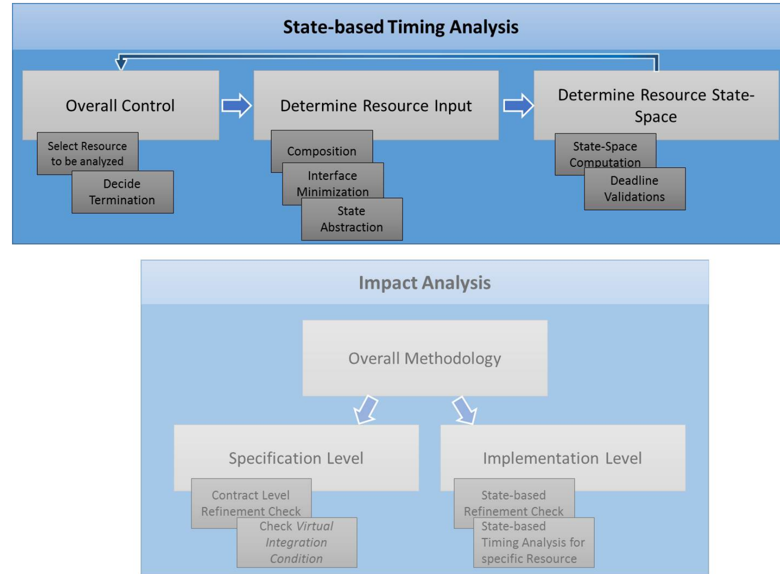


Figure 3.2.: Overview of contributions of thesis.

Consider also that it is not sufficient to restrict the analysis to only the critical instances of the resources on which independent tasks are allocated, and to use these results to compute the state spaces of all dependent resources. This approach results in too optimistic response times. For this, assume that resource $R1$ is an electronic control unit (ECU), and task τ_1 has a higher priority than τ_2, and task τ_4 a higher priority than τ_3. Two timing scenarios for this system are illustrated in Figure 3.1: In the upper part the worst-case timing scenario of tasks on resource $R1$ is illustrated. In the lower part a second scenario is illustrated, where τ_2 is interrupted right before its termination. The resulting timings on resource $R2$ are illustrated in the most right part of the figure. Interestingly, the worst-case response time of task τ_3 is obtained by the second scenario, and not by the one where the critical instance of resource $R1$ is considered.

Thus, the timing behaviors of such distributed systems are state-dependent. To obtain adequate results on the timings of tasks an analysis approach should take the state-dependent behavior into account.

The scope of this chapter is the verification of timing constraints and end-to-end latency constraints for distributed hard real-time systems consisting of a set of resources such as ECUs and bus systems. For this, a state-based approach is introduced which considers all possible task activation and interrupt scenarios of an architecture. Thus, in analogy to model checking methods, the full state space for the analysis approach is considered. A well-known problem with such approaches is the one of the state space explosion.

To handle this problem the state space of an architecture is constructed in an iterative manner. On the interfaces of dependent resources minimization and abstraction methods are applied to keep the resulting state spaces as small as possible. The worked out approach was presented the first time in [GHRS12], and was later refined and extended in subsequent publications [GSHR13a, GSHR13b].

Referring to the overall content of this thesis, which is illustrated in Figure 3.2, this chapter deals with the state-based timing analysis part illustrated in the upper part of the figure. This analysis approach will serve as the foundation of the impact analysis approach introduced in Chapter 4 as illustrated in the lower part of Figure 3.2, i.e. the impact analysis will *use* the state-based timing analysis whenever necessary.

This chapter is organized as follows. First, relevant related works are detailed which address the analysis of hard real-time systems. In Section 3.3 the general analysis approach worked out in this thesis is sketched. The state space of a resource is concretized and a simplified version of the symbolic transition systems is introduced. To realize an iterative analysis, in Section 3.4 the abstraction and composition operations for STSs are presented. Section 3.5 details the computation of a resource STS. In Section 3.6 further abstraction techniques leading to more pessimistic response times while boosting the scalability of the approach are introduced. In Section 3.7 the analysis approach is applied to a driver assistance system case study. Finally, a summary of this chapter is given.

3.2. Related Work

There are two relevant research topics for this thesis, i.e. the *analytical* and the *model checking*-based analysis approaches for timing properties. First, the analytical – which are also referred to as classical – approaches are detailed. Then, the model checking approaches are considered. The last category of related works which is considered here is a combination of both the classical and model checking approaches.

3.2.1. Classical Analytical Approaches

In general, analytical methods are stateless methods, which solve closed form expressions. Thus, such analyses are fast and scale good with the size of the system. Unfortunately, for distributed systems with task dependencies these approaches tend to result in pessimistic response times as illustrated in the introduction of this chapter. These approaches are also not good in analyzing other relevant system properties like event ordering or safety properties, as for this a representation of the concrete system state is necessary.

There are many works addressing the classical scheduling analyses which makes it impossible to give a complete list of all works. The focus here is on the most relevant works for the Fixed Priority Scheduling (FPS) approach, which is in the scope of this thesis. The most common holistic and compositional approaches in literature are presented here.

A good reference book which gives an overview of all relevant scheduling policies with their characteristics and utilization bounds is given in [But05]. In this book, the author introduces the fundamental concepts of hard real-time systems. He presents various algorithms for the scheduling of aperiodic and periodic tasks. Besides this, algorithms for fixed- and dynamic-priority servers are presented. Servers are relevant when periodic and aperiodic tasks are considered together within a system. For all approaches a schedulability analysis and an analysis of the complexity and performance is performed.

The Rate-Monotonic Scheduling (RMS) approach was introduced by Liu and Layland in [LL73]. It is an optimal approach for independent periodic tasks with statically assigned priorities and deadlines. The values of the deadlines are equal to the end times of the corresponding periods. The priorities are assigned according to the periods of the tasks. The more general term of this approach is called Fixed Priority Scheduling, where priorities can be assigned arbitrarily.

Analyses of Single Resources

Let us first focus on techniques for the analysis of single resources, as these are the basis for most of the other approaches. In this thesis, the way to compute response times and interference time intervals are basically performed as introduced in these works.

In [LL73] the authors proposed a schedulability check for RMS which is based on the utilization of a (single) processor. The general result is that for processors with a large task set the utilization factor has to be lower than 70 percent to guarantee schedulability.

Joseph and Pandya worked out in [JP86] an approach for a response time calculus for RMS. For their computation they consider the *critical instances* for all tasks, i. e. the worst-case scenario with the largest interference time caused by higher priority tasks. Such a situation is given, when all tasks are activated simultaneously. It was shown that when a task fulfills its deadline at its critical instant, it also will fulfill its deadline in all other situations. In general, the response time of a task τ_i is computed by taking the sum of the worst-case execution time of the task $wcet_{\tau_i}$ and the interference $I(\tau_i)$ caused by all higher priority tasks.

$$r_{\tau_i} = wcet_{\tau_i} + I(\tau_i). \tag{3.1}$$

In [JP86] the interference interval is computed as follows:

$$I(\tau_i) = \sum_{\tau_j \in hp(\tau_i)} (wcet_{\tau_j} \left\lceil \frac{r_{\tau_i}}{p_{\tau_j}} \right\rceil), \tag{3.2}$$

where $hp(\tau_i)$ is the set of all tasks allocated to the same resource which have a higher priority than τ_i.

The approach was extended by [ABR$^+$86] with timing jitter. As stated in Section 2.1 a jitter specifies that task releases are allowed to deviate with a certain value from their

period. By considering a jitter, the interference of a task increases as events may arrive earlier:

$$I(\tau_i) = \sum_{\tau_j \in hp(\tau_i)} (wcet_{\tau_j} \left\lceil \frac{r_{\tau_i} + j_{\tau_i}}{p_{\tau_j}} \right\rceil). \tag{3.3}$$

Thus, a jitter leads to larger response times due to the larger number of possible preemptions.

Holistic Analyses

When dealing with distributed systems and considering task dependencies, where the termination of a task activates another task, a jitter in the activation of dependent tasks has to be considered. These jitter monotonically increase along the task chains and can even exceed the original period. In literature this is called *jitter propagation*, meaning that the jitter resulting from varying response times is propagated to all dependent tasks and has to be considered in their activations as well.

To deal with activation behaviors of dependent tasks, Tindell introduced in [Tin94] *timing offsets*. These offsets describe the delays resulting from the execution times of the triggering tasks of the considered dependent task. The holistic scheduling analysis was then topic of the work of [TC94] : Fixed-point response time equations extended with offsets were specified for all tasks of a considered distributed system capturing all task dependencies of the architecture.

Palencia et al. also worked on the holistic schedulability analysis in [PGH97] and extended their works in [PH98] to deal with dynamic offsets. Dynamic offsets characterize the possible variations of the response times of the triggering tasks.

Compositional Analyses

Compositional analyses were worked out by many authors to enable more powerful approaches than the holistic ones. This thesis does not directly apply the results of the following works but is inspired by these approaches in such a way, that the worked out approach performs the overall analysis in an iterative manner.

The *Real-Time Calculus* (RTC) was worked out by Thiele at al. in [TCN00]. The authors defined continuous request and delivery curves – also called arrival and service curves – which are mathematical descriptions for the amount of computation requested and delivered by a resource up to a specific time. This algebra enables to determine relations between processor demands and deliveries in terms of input and output curves. Using an upper and a lower arrival curve, event models can be specified. These curves determine the minimum and maximum number of events within a time interval. Analogously, upper and lower service functions determine the bounds on available resource capacity, which may also be defined as number of events. A processing resource thus can

have a set of input arrival curves and a service curve. The resource itself is characterized also by a set of functions relating its input curves to output curves, which are the remaining service and a set of arrival curves for dependent resources. The *Modular Performance Analysis* (MPA) framework is based on this real-time calculus[CKT03], where the RTC was applied to analyze the performance of a distributed architecture consisting of computation and communication resources for hierarchical scheduling.

The compositional scheduling analysis realized by Symtavision is called SymTA/S (Symbolic Timing Analysis for System). Its concept was worked by Kai Richter [Ric04] and was improved and extended in several works, e.g. [RE10]. This work is based on [RE02] where the authors defined interface transformations between heterogeneous event models. More precisely, they define transformations between the event models *periodic, sporadic, periodic with jitter, periodic with burst*, and state whether the transformations are approximative or preserve the accuracy. The main idea behind SymTA/S is to transform event models of a specific type to models of another type whenever needed, i.e. whenever different models are considered between dependent resources. With this, the classical scheduling algorithms for the local analysis of resources can be directly reused, as the inputs of the resources are described as standard event models. This implemented concept is very fast and is able to handle large systems. The extension in [RE10] exploits the non-preemptive characteristic of bus systems, i. e. the tasks of a bus system can never complete their computations (nearly) simultaneously. With this, the classical critical instance cannot occur for dependent resources.

CARTS (Compositional Analysis of Real-Time Systems) is another tool for the compositional real-time scheduling analysis [PLE+11]. The approach is able to handle hierarchically scheduled systems, for which it generates resource interfaces enabling the compositional analysis of timing properties. Schedulability is checked for tasks whose resource usage is bounded by periodic resource models developed by Lee et al. [PLE+11]. Composition is done on the resource model level resulting again in periodic resource models by using abstractions.

3.2.2. Model Checking Approaches

Next, model checking approaches are illustrated. The approach of this thesis is based on the formalism of timed automata, thus relies on the model checking approach.

To realize a state-based scheduling analysis with preemption the first works in this area considered the formalism of stopwatch automata. A stopwatch is a real-valued variable. The derivate can be set to zero, which corresponds to stopping the evolution of the variable, such that the tracing of the exact allocated execution times of tasks is possible. Such an approach has been worked out in [AM02] for preemptive jop-shop scheduling. Unfortunately, for this class of automata the reachability problem is known to be undecidable in general [CL00]. Because of this, we chose the formalism of timed automata for our approach.

State-based analysis with UPPAAL has for example been performed in [DILS09] for preemptive and non-preemptive scheduling policies. All entities like tasks, processors, and schedulers are modeled in terms of timed automata and are combined to an automaton network. If preemption is considered, stopwatches [CL00] are used to change the rate of the corresponding clocks to zero. For this, automaton templates are given, which need to be instantiated in the corresponding context. Tasks enter an error state, whenever their computation times exceed their deadline. The schedulability is then checked with respect to reachability of the error states. As mentioned above, problems in the verification of systems including preemptive scheduling policies can arise (decidability of reachability).

In [NWY99] the authors prove that the checking of schedulability for extended timed automata considering a non-preemptive scheduling policy is decidable. This thesis does directly rely on the results of this work and the work of [FPY02]. Thus these works have to be detailed in the following.

The extended class of automata of [NWY99] is later referred to as *task automata* [FKPY07]. The extended automaton class consists of a regular timed automaton, a mapping of actions to tasks, and a scheduling queue. The guard on a transition specifies all possible arrival times of the task which is triggered by the action of the transition. In later works of the authors [FPY02, FKPY07] locations instead of actions are mapped to a set of tasks, such that a set of tasks can be activated in parallel. When a transition is fired, a task is instantiated, added to the queue and the running task is determined with respect to the considered scheduling policy. The queue further determines the remaining computation time and the relative deadline, which is realized by clock variables. When a task gets running, its remaining computation time clock is reset. Whenever a delay transition is taken, the remaining computation time of the running task and the deadlines of all other tasks are decreased correspondingly. To prove the decidability, task automata are mapped to an ordinary timed automaton network. The schedulability is determined through reachability of bad states. In [FPY02] the authors extend this work in order to deal with preemptive scheduling policies. For this, the authors adapt their task automaton formalism such that clocks may be subtracted by a natural number. Such automata are in general undecidable as shown in [BDFP00]. Based on the work of McMains and Variya [MV94] who characterized a decidable sub-class of such automata, referred to as *suspension automata*, the authors of [FPY02] prove that reachability is preserved, if for all clocks there is a maximal value called the *ceiling*, such that subtractions are performed in bounded zones (each clock is subtracted by a value of the ceiling at a max). The schedulability problem is then again encoded as a reachability problem.

The main contribution of the authors of [FPY02] is the proof that the problem of checking schedulability for preemptive scheduling for extended timed automata is decidable. In analogous to their previous work, the states of such an extended automaton consist of a discrete location, a clock valuation, and a task queue. The locations are annotated by tasks. The general semantics of their extended automata is the same as the semantics of timed automata: Whenever the automaton takes a discrete transition to a location,

an instance of each task with which the location is annotated, is created and added to the task queue. The task queue is a sorted list consisting of instances of tasks together with their remaining computation times and deadlines, which are real valued variables. Delay transitions correspond to either resource idle times when no task is active, or the execution of running tasks. If the automaton takes a delay transition, the remaining computation time of the running task and all deadlines of the other tasks are decreased by the corresponding delay value. The schedulability is encoded as the reachability of bad states: If there exists a state containing a task which violates its deadline but which still has a remaining computation time, the bad state is entered. Results of this work were also implemented in the TIMES tool [AFM+04]. It is based on the UPPAAL DBM library, which was extended with subtraction operations on DBMs. In contrast to our approach, the TIMES tool performs analysis tasks in a holistic manner.

In [KY04, FKPY07] the authors extend their work of [FPY02] and determined the decidability and undecidability class of task automata. Their main result is that when interval computation times of tasks (which is the distinction between best- and worst-case execution times), finishing times of tasks release new task instances, data dependencies, and preemption is considered, the problem for this class of automata is undecidable.

Fortunately, the above mentioned subtraction operation is not needed to deal with preemptive scheduling policies, as it was also demonstrated in [HV06]. The authors of this work use the original timed automaton formalism to model this problem and verify timing properties by using UPPAAL. As a front-end they use sequence diagrams, from which timed automata are derived. The approach is applied on an in-car radio navigation system case study, where end-to-end latency deadlines to change the volume of the radio and the reception of *traffic message channel* (TMC) signals from a radio station are specified. In [PWT+07] these automaton models were reused to apply the approach on a set of benchmark systems and to compare the analysis results to other techniques such as MPA or SymTA/S. Unfortunately, the work of [HV06] illustrates only an instantiation of timed automata for a couple of example architectures to perform the analysis. It does not illustrate a general concept to automatically build the automaton network of a given system architecture and perform the analysis.

The work presented in [BH09] deals with an extension of timed automata. The presented model is called *Interrupt Timed Automata* (ITA) as they define interrupt levels as a control structure. On each level *exactly* one clock is active, where clocks from higher level suspend clocks on all lower levels. Guards on transitions are linear expressions over clocks from the current level and all levels below. Updates are linear expressions only over clocks at levels below. This restricted model of stopwatch-automata has the characteristic that the reachability problem remains decidable. The model is not more expressive than the approaches illustrated above. In contrast to this work, the approach works in a holistic manner. Also, the introduced formalism seems not to be as intuitive as the formalism of timed automata.

In [MC09] a compositional, timed automaton-based analysis technique for preemptive,

hierarchical scheduling strategies was presented. A major difference to this thesis is that they adopt a discrete time formalism instead of dense time in order to deal with preemption.

The work of [CGR12] is not directly related to this thesis, but is interesting to look at. In this work, timed parameterized networks are introduced as an extension of timed automata. Instead of checking reachability properties for a fixed number, a number n of processes are considered for an increasing n. For this, a fully parametric reachability check is worked out. This is realized by transforming the timed automaton network into an input model of SMT (Satisfiability Modulo Theories) solver which is able to handle such parametric systems. The main focus of their work is not the one of this thesis, as they consider parameterized networks and do not apply the concept to the timing properties of resources. Anyway, the work of [CGR12] could be used for future extensions, to extend the approach illustrated in this thesis.

3.2.3. Combination of Analytical and State-based Approaches

Our approach shall combine analytical and state-based approaches by the usage of abstraction techniques on the interfaces of the resources of a given system. For such a combined approach there are several works in literature as introduced in the following.

The authors of [DMS09] consider task network models and map those to timed automata networks, while proving that the mapping preserves the original semantics. The first step is to compute the response times of the tasks in an analytical manner. With the obtained response times, the automaton network is constructed. The obtained network is checked against requirements formalized as Live Sequence Charts [DH01], which are also translated to timed automata and composed to the system network. The properties are verified using Uppaal. In contrast to this thesis, they focus on the transformation of task networks to the formalism of timed automata. The analysis is then performed in a holistic manner.

A combination of timed automata and the RTC is also applied by the authors of [KMY07] and implemented as a tool called *CATS*. The architecture is modeled as in RTC extended by timed automata: The service curves are used to model resources, and arrival curves and timed automata are used to model task arrival pattern. The focus here is on the interfaces of dependent systems. In contrast to our work state-dependent behavior is not preserved on the interfaces.

In [PTCT07] the authors propose a methodology to combine *Event Count Automata* (ECA) with the Real-Time Calculus (RTC). An ECA is a discrete automaton extended by integer valued count variables, which determine the number of received events over time intervals. A state specifies the minimum and maximum number of events which may arrive in every time step when the system is in the corresponding state. The proposed methodology allows to switch between these different modeling formalisms, i. e. processing elements which are state-dependent can be modeled as ECAs, and those which need only

information about the input event streams by RTC functions. To connect resources modeled by different formalisms, transformations between these have been worked out. Arrival curves can be translated to ECAs, and ECAs are able to be translated back whenever needed. As our approach, this work realizes a non-holistic approach. In contrast to our work they focus on the translation of the formalisms for dependent components. With this, state-dependent behavior can only be handled locally for a resource or a small part of the overall architecture. This is because when performing the translation by counting events, the state-dependent behavior is lost for dependent resources. The approach in this thesis tries to preserve the state-dependent behavior for all dependent resources.

In [LPT09] the authors combine the RTC approach with the formalism of timed automata. Thus, in contrast to [PTCT07] the continuous time model is considered. For this, they describe the transformation from event models specified as arrival curves to a network of timed automata, and back. The difficulty in doing so is that the arrival curves are defined in the time interval domain while timed automata are defined in the time domain. With this technique, individual resources may be modeled in an abstract stateless manner by the usage of RTC, or considering state-dependent behavior using timed automata. The distinction of their work to this thesis is the same as illustrated for the work of [PTCT07].

3.2.4. Contribution of this Chapter

What is missing in literature is an appropriate combination of model checking approaches to analyze timing properties in a more compositional manner, without resulting in too pessimistic results. The timing analysis approach of this chapter introduces a new approach, which is interface driven: The state spaces of resources are computed successively. Abstraction techniques on the interfaces of dependent resources are then worked out to reduce the input behavior. With this reduced state space enough behavior shall be kept such that on the one hand the timings of the tasks on dependent resources are not too pessimistic, and on the other hand the overall verification time is reduced. The application of different abstraction techniques on the interfaces of dependent resources has not been worked out in literature so far.

3.3. General Approach

The timing analysis approach worked out in this thesis is based on model checking, where the state spaces of the resources are computed, which encapsulate the relevant timing information for tasks and end-to-end latencies. In contrast to standard model checking, this approach works in an iterative fashion. The interfaces between resources are kept as minimal as possible to boost the scalability of the approach, while the accuracy of the response times is not affected by approximations. In other words, whenever interfaces

between dependent resources are minimized, the end-to-end timing latencies and response times remain exact with respect to the considered system model.

The approach distinguishes parts of the system containing cycles and parts which are cycle-free. The focus for the iterative analysis technique is on cycle-free system parts. In general, cycles in systems lead to mutual dependencies, such that the state spaces cannot be separated and computed in isolation. The solution of our approach to handle parts of the architecture containing such mutual dependencies is to perform the analysis in a classical holistic fashion. Regarding the holistic part, our analysis will not yield any improvements with respect to the state-of-the-art tools. A certain restricted class of feedback loops can be handled also in an iterative fashion, but in general such parts have to be analyzed holistically. Such a restricted class is discussed in the Appendix B.

In this section, first the general concept of the scheduling analysis technique worked out in this thesis is illustrated. In-depth details such as state space characteristics and usage of the elements such as clock variables is detailed thereafter. These details are necessary to understand the subsequent sections.

3.3.1. Iterative Analysis Approach

The timing analysis approach proceeds as follows: In order to build the state space of a resource, its *input behavior* has to be determined. The input behavior defines the *activation times* of all tasks which are allocated to the resource. All state spaces here are mainly represented by symbolic transition systems (STSs). States of the STSs determine ranges of valuations of clock variables and their differences. The states further contain information about released tasks, i.e. whether a task is currently running, is interrupted, or is in the ready queue. The representation of the state space is detailed in the next subsection.

A resource may have multiple sources for its inputs: Independent tasks are triggered by event streams, while dependent tasks are triggered when the tasks on which they depend terminate. Thus, the activation behaviors for all tasks on a resource are given by a set of STSs. In order to determine a *single* input state space for each resource, the product of the corresponding input STSs has to be computed.

To built the state space of a resource, all of its input behaviors have to be available. For resources, on which only independent tasks are allocated, the input is directly given: For every event stream the STS is directly derived as illustrated in Figure 2.6, and the product of all STSs is computed. For instance, this is the case for resource $R1$ of the system illustrated in Figure 3.3. In contrast to this, to compute the state space of a dependent resource such as $R2$ in Figure 3.3, first the state spaces of all resources on which $R2$ depends have to be available. Note that for parts of the considered system consisting of resources with cyclic dependencies a problem arises here: The input of a resource R cannot be determined separately from the behavior of resources from which it depends on, if these resources also depend on R. Thus, such parts of the considered

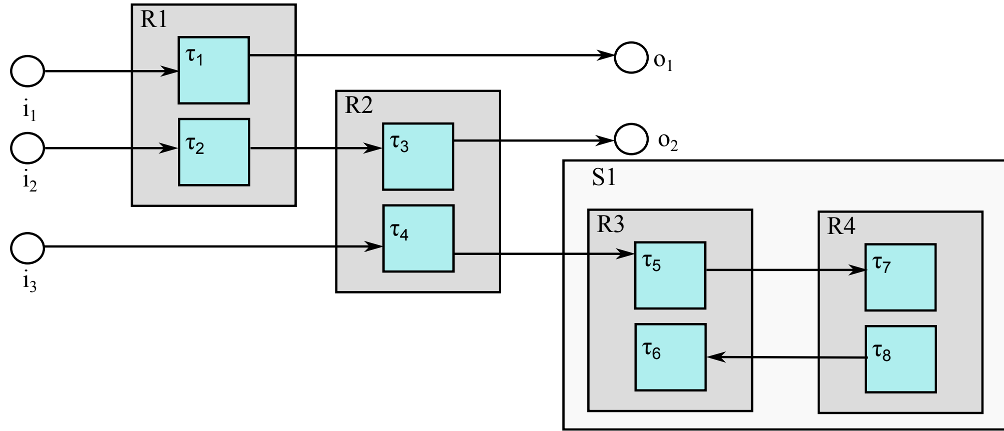


Figure 3.3.: System architecture containing four resources and dependencies.

system architecture have to be treated holistically: Instead of determining the input of a single resource, the input of this whole system part containing such mutual dependencies has to be computed, and based on this input the analysis is performed holistically. As an example consider the part $S1$ of the system illustrated in Figure 3.3: As there is a mutual dependency between the resources $R3$ and $R4$, the part $S1$ has to be analyzed in a holistic manner.

When the input of a resource is determined, the next step is to build the state space of the resource itself. For this, the input state space, the behavior of the scheduling policy, and the execution times and priorities of the allocated tasks are taken into account. The computed state space of a resource is then used to determine the response times of the allocated tasks. Further, the resource STS serves as an input for *dependent* resources, i. e. for resources on which dependent tasks are allocated. In order to keep the interfaces between the resources as small as possible, parts of the state spaces that are not relevant for the input behavior of the dependent resources are abstracted.

Thus, two operations on STSs are needed for such an iterative approach, (i) the *product* computation of a set of STSs, and (ii) the *abstraction* of parts of the state spaces, such that only the relevant information with respect to dependent resources are kept.

Consider the example in Figure 3.3, which consists of four resources where on each resource two tasks are allocated respectively. The task τ_3 on resource `R2` depends on task τ_2 on resource $R1$. Tasks τ_1, τ_2, and τ_4 are activated by event streams, thus the inputs of resource $R1$ are directly given and its state space can be computed. Next, the input of resource $R2$ has to be determined, which depends on both the state space of $R1$ and the event stream activating τ_4. As the detailed timing information of the task τ_1 is not relevant for $R2$, the output of the STS of $R1$ is reduced by abstracting from states

encapsulating information about this task. After this minimization, both STSs can be combined by computing the product. After the computation of the state space of $R2$ the holistic analysis for $S1$ can be started.

The needed operations on STSs and the analysis algorithm are detailed in Section 3.5.

3.3.2. Symbolic Transition Systems of Resources

To encode the problem of a state-based timing analysis, we will use *resource STSs* which are a concretization of the general symbolic transition systems of Definition 6 by defining special variables for the discrete location vector and specific clocks to capture the timing behavior of tasks. In general, we need three types of clock variables to decide *i*) when a task is activated (or instantiated), *ii*) when a task finishes its computation, and *iii*) the response time of an instantiated task. Further, as the computation times of instantiated tasks have to be measured, the states are equipped by some task queues as already mentioned in Section 2.3.

These concretizations will be detailed in the following. Note that the semantics of the general STS is not affected by the following specializations.

Discrete Location Vector

Independent tasks are triggered by event streams. For this, the timed automaton-based representation of Figure 2.3 is considered here. Observing this automaton reveals an initial non-determinism inherent to event streams. In order to capture this non-determinism of independent tasks the discrete part of the state space of each resource has to be represented by 2^n locations for n independent tasks. If further jitter behavior is considered, the number of locations grows to 3^n instead, as a further location is needed for each task as illustrated in the automaton of Figure 2.3. The set of locations is indicated by $L = L_1 \times \cdots \times L_n$, where $L_i = \{l_i^0, l_{i,1}, l_{i,2}\}$ is the location encoding the state of a task τ_i over the index set $I = \{1, ..., n\}$. The initial location where no task has been released so far is indicated by $l^0 = (l_1^0, \cdots, l_n^0) \in L$.

Note that this location vector can also easily be extended to handle more complex task behaviors such as the formalism of function networks [BMS09]. These function networks have state-dependent execution times, such that a more precise modeling of real-time systems can be realized. However, in this work we focus on the location vector introduced above and restrict to the basic task definition.

Besides the set of locations, the state set of a system is defined over clock valuations over a set of clocks C that capture the timing behaviors of the allocated tasks. This is detailed in the following paragraph.

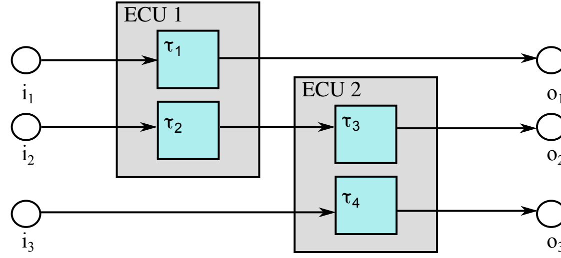


Figure 3.4.: Example architecture with periods $p_{\tau_1} = 60$, $p_{\tau_2} = 5$, $p_{\tau_4} = 60$, and computation times $c_{\tau_1} = 35$, $c_{\tau_2} = 2$, $c_{\tau_3} = 4$, $c_{\tau_4} = 12$, and priorities $pr_{\tau_1} > pr_{\tau_2}$, $pr_{\tau_4} > pr_{\tau_3}$.

Periodic Activation Clock

The first type of clock is necessary for each *independent* task τ. It traces the periodical activation times of such a task. These clocks are referred to as *periodic activation clocks* $c_p(\tau)$ for a task τ.

For *dependent* tasks period activation clocks are not necessary. The activation of such tasks are implicitly given by the termination of the tasks from which they depend on. Consider the example of Figure 3.4: For this architecture, the periodic clocks $c_p(\tau_1), c_p(\tau_2), c_p(\tau_4)$, and computation time clocks $c_c(\tau_2)$, $c_c(\tau_3)$ are needed. The task τ_3 is implicitly activated whenever an instance of task τ_2 terminates.

Computation Time Clock

In scenarios, where a task may be activated before its previous instance has finished its computation, a further clock is needed which measures the time frame from releasing a task up to the finish of its computation. Such a scenario will be called *overlapping activation* in the following. In these scenarios multiple task instances t_i of a task τ may be active at the same point of time. This kind of clock is called *computation time clock* $c_c(\tau)$ in the following and is necessary to be able to determine when a task instance finishes its computation.

When overlapping activations of a task do not occur in the system, a single clock for such a task is sufficient to determine both its periodic activation and the finish of computation of the corresponding instances. For this, consider the scenarios of Figure 3.5: Whenever an instance of a task is activated after its clock c_p reaches the value of the task period, i.e. $c_p == P$ holds, the clock c_p is reset (indicated by the curved brackets), such that the next activation occurs exactly after the specified period value P. After a task has been released, it runs until it finished its computation. It finishes its computation, when enough resource time has been allocated to the task, which is

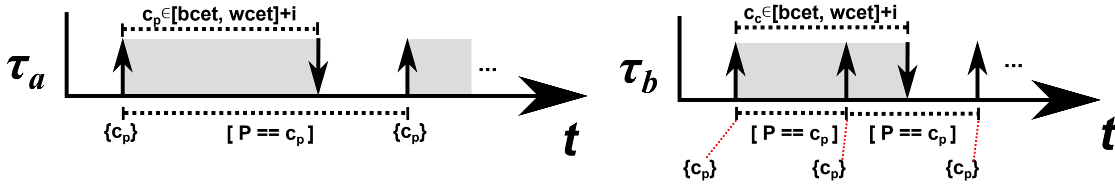


Figure 3.5.: Scenarios with a non-overlapping activation of task τ_a (left) and an overlapping activation of τ_b (right).

determined by the sum of its execution time $[bcet, wcet]$ and a delay interval i caused by interrupts form higher priority tasks. When there is no overlapping activation like in the left scenario of Figure 3.5, the period clock can be used to measure this time frame, as the clock is never reset before an instance terminates. It is assumed that the information about the existence and number of maximal possible overlapping activations is given a priori and not determined by the analysis approach automatically.

If multiple instances can be released as illustrated in the right scenario of Figure 3.5, this situation changes: The clock c_p cannot be used to determine the above mentioned finishing time, as c_p is reset each time an instance is released. Thus, a further clock c_c is needed to trace the computation time of an active instance.

Response Time Clock

Computation time clocks are needed to determine, whether a task instance may finish its computation. If the exact response times of each task instance are of interest, we need a further type of clock, which is referred to as *response time clocks*. Computation time clocks are not able to measure the exact response times, as there are scenarios, where these have to be reset while a task instance is activated and in the ready task set.

Multiple response time clocks are needed, one for each task instance t_i, as the approach relies on using simple *clocks* to realize the timing analysis with preemption. The reason for this is that neither the derivative of a clock can be changed (e.g. to *stop* a clock from progressing), nor subtraction operations can be performed as e.g. done in the work of Krčál and Yi [KY04].

As one separate clock per task instance has to be used, the maximal number of possible parallel activations of one task needs to be known This maximal number is called task instance bound and can always be derived a priori: Considering fixed priority scheduling policies, for an independent task the number is bounded by the maximum interrupt time from all higher priority tasks, and the period value of the task. For instance, consider the task τ_2 in Figure 3.4. The bound results in the following value: $n = \lceil \frac{c_{\tau_1}}{p_{\tau_2}} \rceil + 1 = 8$. Thus, within a computation time of 35 time units of an instance of τ_1, at most eight instances of τ_2 may be activated (each 5 time units).

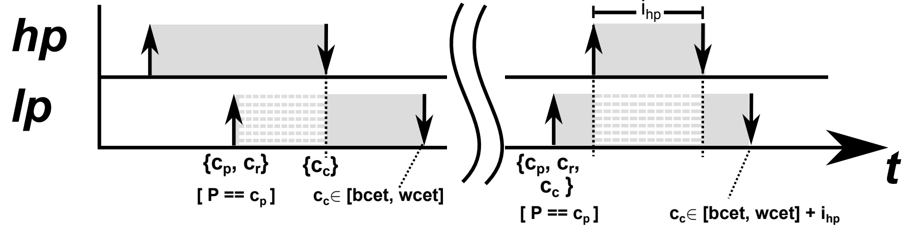


Figure 3.6.: Two tasks hp, lp and the interrupt scenarios.

For dependent tasks this computation is a bit more complex and is based on fixed-point computations. Consider the task τ_3 in the system of Figure 3.4: First, the maximum number of possible terminations of τ_2 caused by a burst have to be determined, which is basically a fixed-point computation. During the computation time of an instance of τ_1, in the worst-case eight instances of τ_2 may be released. Thus, after the termination of τ_1, eight successive terminations of τ_2 occur within 16 time units in such a case. Within this time frame, further three instances of τ_2 are released, as the period of τ_2 is 5 time units. By the time these three instances terminate, one further activation of τ_2 can occur. Thus, within a time frame of 24 time units 12 terminations of τ_2 can occur. This number could already be used as a bound for τ_3. This bound can be further refined as during the time frame of 24 time units instances of τ_3 may terminate. Within this time frame, also the worst-case interruption times of all higher priority tasks have to be taken into account. In this case the execution time of τ_4 has to be considered which is 12 time units. This value has to be subtracted from the above value, i.e. $24 - 12 - 2 = 10$. Thus, in the worst-case from 24 time units only 10 time units are available for instances of τ_3. Within this time frame two instances of τ_3 can be terminated. Thus, the bound can be decreased to a value of 10. Let in the following $inst : \mathsf{T} \to \mathbb{N}$ be the function which determines for each task its maximal number of possible activations.

To measure the time for end-to-end latencies between task chains $\tau_1, \cdots, \tau_n$ also multiple response time clocks $c_r(\tau_1, \tau_n)$ are needed for the same reason as for the multiple response time clocks of task instances. For such a chain, the number of end-to-end latency clocks is bound to $max(inst(\tau_1), \cdots, inst(\tau_n))$.

Putting it togeter

The application of the introduced clocks is illustrated in Figure 3.6, where two tasks hp (high priority) and lp (low priority) are allocated to a single resource with a fixed priority scheduling policy. All clocks in the figure refer to the task lp. The clocks in curved brackets indicate a reset, square brackets indicate clock constraints. The parameter P is the period of lp.

First, consider the left scenario: The task lp is released while an $hp-$task instance is already running, and its period clock and response time clocks are reset. The time hp finishes its computation, the computation clock c_c of lp is reset to be able to determine the finishing time. That is, the task lp terminates, when this clock is within the best- and worst-case computation time. Note that at this point, the response time of the task is contained in the clock c_r.

In the second scenario illustrated in the right part of the figure, the running instance of lp is interrupted by an instance of hp. The termination of lp is then reached, when its clock c_c is in the range of the interval resulting from the sum of the best- and worst-case computation time of lp and the interrupt interval i_{hp}.

Task Lists

To manage the activation of tasks and resource allocations to task instances appropriately, all states of resource STSs are enriched by an active task map and a ready task list.

$\mathcal{A}_s : T \rightarrow I$ is the active task map of state s of a resource STS under analysis, indicating the interruption times $I = [a, b]$ with $a, b \in \mathbb{N}$ of all instantiated active tasks T. The first element determines the currently running task. This map may be empty for some states as for example for the initial state.

The ready task list $\mathcal{R}_s = (t_a, t_b, ..., t_n)$ of state s of a resource STS contains all tasks ready to run, but to which no computation time has been allocated so far. This list preserves the order of task instantiation, i.e. if task t_i was instantiated before t_j of the same type, t_i will be handled before t_j. In other words a newly released task is added into the ready task list and gets not running until the previously instantiated tasks of the same type have finished their computations. Note that if an unbounded system is considered this list is not finite.

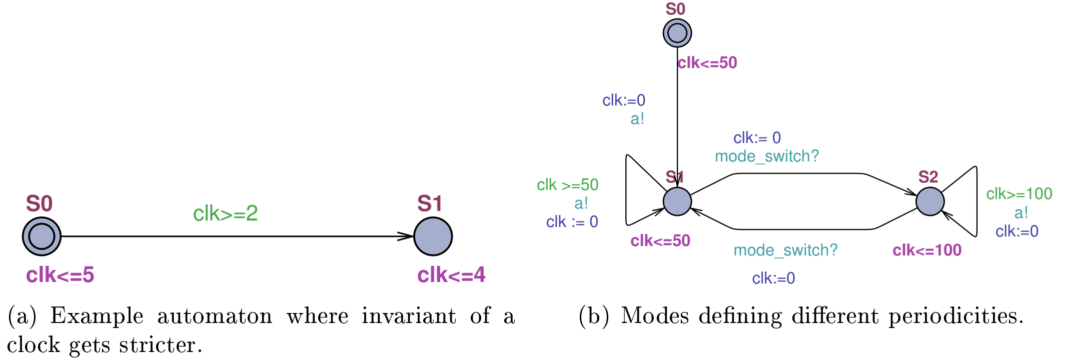
For each task type it holds that at most a single task instance of this type is included in the active task map. All other ready to run instances of this type are inserted in the ready list.

3.3.3. Simplification of Symbolic Transition Systems

To have a more compact representation of the symbolic transition systems where time successors and discrete successors are combined, the definition of the transition relation of the STSs is slightly modified in the following (rf. to Definition 6). This will also solve the problem of preserving the lower clock bounds when a discrete transitions is taken, as it was discussed in Section 2.2.

First, recall the original transition relation definition:

- $\langle l, D \rangle \xrightarrow{\Delta} \langle l, D^{\uparrow} \cap D_{I(l)} \rangle$, where $I(l) = \bigwedge_{j=1}^{n} I_j(l_j)$



(a) Example automaton where invariant of a clock gets stricter.

(b) Modes defining different periodicities.

Figure 3.7.: Example for invariants in automata.

- $\langle l, D\rangle \xrightarrow{\lambda} \langle l', (D \cap D_{\varphi_i} \cap D_{\varphi_j})[\varrho_{\{i,j\}} \to 0] \cap D_{I(l')}\rangle$ where $l' = l[l_{\{i,j\}} \to l'_{\{i,j\}}]$, if there are some $i, j \in \{1, ..., n\}$ with $\lambda \in \Sigma_i \cap \Sigma_j$ such that $(l_i, \lambda!, \varphi_i, \varrho_i, l'_i) \in R_i$ and $(l_j, \lambda?, \varphi_j, \varrho_j, l'_j) \in R_j$.

In the second part of the relation, the intersection between the current zone D and the zones $D_{\varphi_i}, D_{\varphi_j}$ which is induced by the guards of the firing transitions φ_i, φ_j is taken. After resetting the clocks defined by the corresponding transition, the intersection with the reached zone and the zone $D_{I(l')}$ induced by the invariant of the reached location is taken. The last step, i.e. the intersection with the invariant of the reached location, is only relevant for clocks which are not reset. Further, invariants only affect the upper bounds of the corresponding clocks. Thus, this intersection is only relevant for locations, where the upper bound of a clock which is not reset gets stricter.

For this, consider the example of Figure 3.7(a): The invariant constraining the valuation of clock *clk* in state $S1$ is stricter than the one in state $S0$, which affects the reached zone. Note that such automata are not deadlock free: If the control flow remains in state $S0$ within the time interval of $(4, 5]$ the automaton is not able to switch to location $S1$.

In this thesis, timed automata are restricted in such a manner that the last step is not necessary, i.e. the intersection with the invariants of the reached location. This operation can be omitted, if the upper bounds of all clocks never get tighter in any reached target location without being previously reset. This is a valid restriction for the systems which are considered in this thesis: For this, consider the types of clocks used for the analysis approach mentioned above: The bounds of the computation time clocks of instantiated tasks are either fixed values or are increased by the interruption time, thus are monotonically increasing, but do not get stricter. The period clock c_p will never change within a period cycle, thus will also not get stricter. If mode-dependent periods and execution times are considered as in function networks, the corresponding clocks are

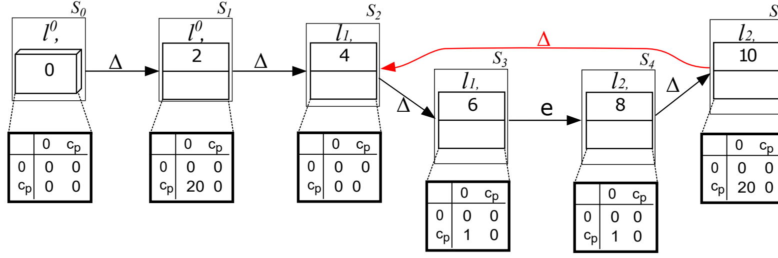


Figure 3.8.: Original STS for an event stream automaton with $p = 20, j = 1$.

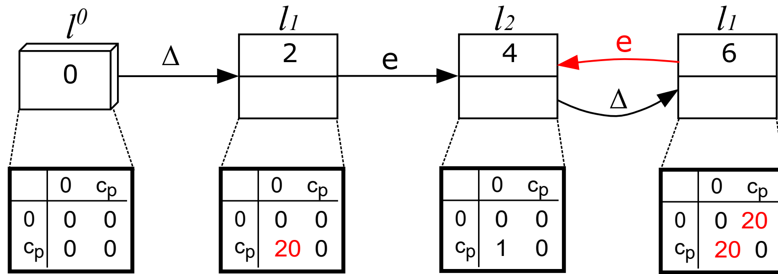


Figure 3.9.: Simplified STS for an event stream automaton with $p = 20, j = 1$.

previously reset to adequately capture the timing properties. For instance, consider the transition from $S2$ to $S1$ in the automaton of Figure 3.7(b): The period is switched from 100 time units to 50 time units (modeled by defining the invariants accordingly). Before the state is entered, the clock determining the periodical activation is reset.

With this restriction the last intersection can be skipped and both transition steps can be combined to a single one as follows:

$$\langle l, D\rangle \xrightarrow{\lambda} \langle l', (D[\varrho_{\{i,j\}} \to 0])^{\uparrow} \cap D_{I(l)} \cap D_{\varphi_i} \cap D_{\varphi_j}\rangle, \tag{3.4}$$

where $I(l) = \bigwedge_{j=1}^{n} I_j(l_j)$ and $l' = l[l_{\{i,j\}} \to l'_{\{i,j\}}]$, if there are some $i, j \in \{1, ..., n\}$ with $\lambda \in \Sigma_i \cap \Sigma_j$ such that $(l_i, \lambda!, \varphi_i, \varrho_i, l'_i) \in R_i$ and $(l_j, \lambda?, \varphi_j, \varrho_j, l'_j) \in R_j$.

Recall the original STS of Figure 3.8. With the new transition relation, the resulting STS is illustrated in Figure 3.9.

Observe that in contrast to the original STS in Figure 3.8 this simplified STS is more compact. Further, the lower bounds of the reached locations are preserved, e.g. in state $\langle l_1, D_6\rangle$ it is known that the value of c_p is exactly 20 time units. This information is crucial for the operations on STSs to realize an iterative analysis as detailed in the next section.

3.4. Operations on Symbolic Transition Systems

As discussed in the previous section, to realize an iterative analysis approach two basic functions on the symbolic transition systems are needed, i.e. the *product* computation of a set of STSs to determine the input behavior of a resource, and the *interface computation* in terms of abstraction functions on the STS such that only the relevant information with respect to dependent resources are kept. The abstraction in this section is defined in such a manner that the activation behavior which is described by the original interface STS remains exact. Thus, the abstraction here is used as a minimization of the state spaces. Later in Section 3.6 further abstraction functions are defined in such a way that after their applications over-approximated STSs with respect to the timing behaviors of the considered tasks are achieved, while resulting in smaller state spaces.

3.4.1. Interface Computation

The iterative analysis approach shall keep the interfaces between dependent resources as small as possible, while preserving the accuracy of the computed response times and end-to-end latencies. To minimize the state space abstractions on the state set of STSs are necessary. In general, an abstraction function defined over a state set S is a surjective function $\alpha : S \to S'$, where $S' \subseteq S$. Abstraction functions should also be defined in such a way that these are *total*:

$$\forall s \in S, \exists s' \in S' : \alpha(s) = s'. \tag{3.5}$$

The state set of our approach consists of zones and discrete location vectors, thus specific abstractions on both parts of the state space are presented. The combination of both abstractions and the application to interface STSs is addressed thereafter.

Abstraction on Locations

As illustrated previously, the state space of a resource consists of a set of discrete locations $L = L_1 \times \cdots \times L_n$ over an index set $I = \{1, \cdots, n\}$ for each resource on which n independent tasks are allocated. For a location $l = (l_1, \cdots, l_n) \in L$ let $l_{|i}$ with $i \in \{1, \cdots, n\}$ be the ith element of the tuple, i.e. $l_i \in L_i$.

For $I' \subseteq I$ let $\alpha_{I'} :\ L \to L'$ be the surjective function mapping the set of locations L over I to the set of locations L' over index set I', where locations within the tuples with indexes not in I' are left out. Let $l \in L$, then $\alpha_{I'}(l) = (l'_{i_1}, \cdots, l'_{i_k})$ for $i_1 < \cdots < i_k \in I'$ and $I' \backslash \{i_1, ..., i_k\} = \emptyset$, such that $l'_z = l_{|z}$ for all $z \in I'$.

For instance, consider the index set $I = \{1, ..., n\}$ and $I' = I \setminus \{i\}$. By the application of this the following is obtained: $\alpha_{I'}(l_1, ..., l_i, ..., l_n) = (l_1, ..., l_{i-1}, l_{i+1}, ..., l_n)$.

As $\alpha_{I'}$ is surjective the inverse function $\alpha_{I'}^{-1}$ maps a location l over I' to a set of locations L over I with $I' \subseteq I$:

$$\alpha_{I'}^{-1}(l') := \{l \in L \mid \alpha_{I'}(l) = l'\}. \tag{3.6}$$

That is $\alpha_{I'}^{-1}$ extends the locations induced by I' to locations containing elements of I, e.g. for $I = I' \cup \{n+1\}$ we get $\alpha_{I'}^{-1}(l_1, ..., l_n) = \{(l_1, ..., l_n, l_{n+1}) | \; l_{n+1} \in L_{n+1}\}$.

As an abbreviation tasks instead of explicit indexes are used in the following, e.g. $\alpha_{\{\tau_i, \tau_j\}} := \alpha_{\{i,j\}}$ or $l_{\tau_i} = l_i$.

Abstraction on Zones

The abstraction on clock zones is basically a projection operation over the contained propositions. Let $C' \subseteq C$ be two clock sets. For a clock constraint $g \in \Phi(C)$ let $g_{|C'}$ be the constraint, where all propositions containing clocks of the set $C \backslash C'$ are removed. For the induced zone $D = \{\nu \mid \nu \models g\}$ of a constraint $g \in \Phi(C)$ the zone projection operation is defined accordingly:

$$D_{|C'} = \{\nu \mid \nu \models g_{|C'}\}. \tag{3.7}$$

Analogously, for a constraint $g \in \Phi(C')$ let $g_{|_C^{-1}}$ be the constraint extended with propositions containing clocks in $C \backslash C'$. Let $D_{|_C^{-1}} = \{\nu \mid \nu \models g_{|_C^{-1}}\}$ be the corresponding induced zone. The new constraints are of the form $0 \leq c \leq \infty$ for all $c \in C \backslash C'$. Thus, the extension operation is defined in such a way that it does not affect the original zone.

Let $g \in \Phi(C')$ be a clock constraint and D its induced zone. If the zone D is extended to C and restricted thereafter again to the original clock set, the original zone is not affected:

$$D = (D_{|_C^{-1}})_{|C'}. \tag{3.8}$$

For instance, consider the sets $C = \{c_1, c_2\}, C' = \{c_2\}$ and the constraint $g = c_1 - c_2 \leq 3 \wedge c_1 \leq 5 \wedge c_2 \leq 1$. Then $g_{|C'} = c_2 \leq 1$. Further, we have $(g_{|C'})_{|_C^{-1}} = c_2 \leq 1 \wedge 0 \leq c_1 \leq \infty$.

Computing the Interface Between Dependent Resources

With the aid of the abstraction functions introduced on clock zones and sets of locations, the abstraction function which minimizes the interface between dependent resources is enabled to be defined. Note that for the sake of readability only dependencies between two resources will be detailed here. This is no restriction to the general case, as all definitions can be extended in a straightforward manner.

First, the following functions need to be defined in order to be able to reference to all relevant parts of a state space. For a dependent task, the function $pre(\tau)$ determines the set of tasks, from which it directly depends. Let $G = (\mathsf{T}, E)$ be a task dependency

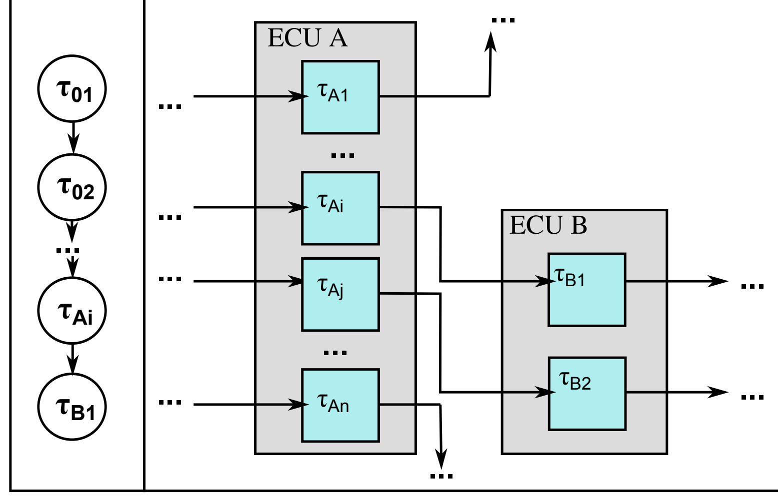


Figure 3.10.: Left: task dependencies of task τ_{B1}; Right: computation of interface between ECU A and ECU B.

graph according to Definition 1. The set of immediate predecessors of a task $\tau \in \mathsf{T}$ is determined by the following function:

$$pre : \begin{cases} \mathsf{T} & \rightarrow & 2^{\mathsf{T}} \\ \tau & \mapsto & \{\tau' \mid (\tau', \tau) \in E\} \end{cases} \tag{3.9}$$

Note that if a task τ has no predecessors in the task dependency graph, the function *pre* will return the empty set.

To determine for all (dependent) tasks $\tau \in \mathsf{T}$ the set of independent tasks, which directly or indirectly lead to a trigger of τ, the following function is defined:

$$init : \begin{cases} \mathsf{T} & \rightarrow & 2^{\mathsf{T}} \\ \tau & \mapsto & \begin{cases} \{\tau\}, \; if \; pre(\tau) = \emptyset, \\ \bigcup_{\tau' \in pre(\tau)} init(\tau'), \quad \text{else.} \end{cases} \end{cases} \tag{3.10}$$

As an example, consider the task dependency graph in the left part of Figure 3.10, where $init(\tau_{01}) = \{\tau_{01}\}$, and $init(\tau_{B1}) = \{\tau_{01}\}$.

Further, the following functions are needed.

- hp_{T} : $\tau \mapsto \mathsf{T}'$ with $\mathsf{T}' \subset \mathsf{T}$ determines for a task $\tau \in \mathsf{T}$ the set of tasks with a higher priority,
- lp : $\mathsf{T}' \mapsto \tau$ with $\mathsf{T}' \subseteq \mathsf{T}$ determines the task τ with the lowest priority within the set T'. For the set of tasks of a resource, this element is always unique.

The index of hp is omitted whenever it is obvious from the context.

Next, the relevant subset of clocks $C' \subseteq C_A$ of an interface STS between a resource A and a resource B is determined. Assume that on resource B tasks are allocated which depend on tasks allocated on A. For this, let $F = \mathsf{T}_A \cap pre(\mathsf{T}_B)$ be the set of tasks on a resource A which trigger some tasks on a dependent resource B. In the example of Figure 3.10 this results in $F = \{\tau_{Ai}, \tau_{Aj}\}$. The subset of relevant clocks is chosen in such a manner that the computation time clocks $c_c(\tau)$ of tasks in the set $\tau \in F$ are preserved. These clocks determine the activation times of the dependent tasks *relatively* to their starting times. Thus, also the information has to be preserved, when these tasks were initially activated. This information is encapsulated in the period clocks of the tasks triggered by event streams, i.e. $c_p(init(\tau))$ for all $\tau \in F$. In this set, clocks of tasks which are allocated on other resources than A, B of the system architecture may be included.

Unfortunately, also the set of period clocks of all higher priority tasks needs to be preserved. This topic is discussed in detail in Subsection 3.6.2.

To sum up, the following set of clocks C' has to be preserved:

$$\forall \tau \in hp(lp(F)).\ \exists c_p(init(\tau)) \in C' \quad \wedge \quad \forall \tau \in F.\ \exists c_c(\tau) \in C'. \tag{3.11}$$

Consider the example in the right part of Figure 3.10 and assume that the task priorities correspond to the ordering in the figure. Then we get

$$C' = \{c_p(\tau_{A1}), \cdots, c_p(\tau_{Aj}), c_c(\tau_{Ai}), c_c(\tau_{Aj}))\}.$$

With this set, the abstraction function is defined that computes the output STS of resource A with the state set S_A for a resource B on which dependent tasks are allocated:

$$\alpha_F : \begin{cases} S_A & \rightarrow \quad S' \\ \langle l, D \rangle & \mapsto \quad \langle \alpha_F(l), D_{|C'} \rangle. \end{cases} \tag{3.12}$$

Note that this definition works fine if the index set for the tasks is globally defined for the complete system architecture. If *local* indexes are defined (for each resource) a reordering and relation function of the indexes would be necessary. Here, we will not detail this discussion and assume that indexes are defined globally.

The inverse function for a state is defined as follows.

$$\alpha_F^{-1} : \langle l, D \rangle \mapsto \{\langle l', D_{|_C^{-1}} \rangle |\ l' \in \alpha_F^{-1}(l)\}. \tag{3.13}$$

This abstraction function induces the following interface STS:

Definition 15. *Let $STS_A = (S, S^0, C, \Sigma, \rightarrow)$ be the STS of a resource A. The interface between two resources A, B where $F = \mathsf{T}_A \cap pre(\mathsf{T}_B)$ and $F \neq \emptyset$ holds, is the STS $STS'_A = (S', S^0_{\alpha_F}, C_F, \Sigma_F, \rightarrow_{\alpha_F})$ over task set F with*

- *$S' = \alpha_F(S)$ the induced set of abstract states,*
- *$S^0_{\alpha_F} = \alpha_F(S^0)$ the initial abstract state,*
- *$C_F \subseteq C$ the clock set as defined in Formula 3.11,*
- *$\Sigma_F \subseteq \Sigma$ the alphabet depending on the task subset F,*
- *$\rightarrow_{\alpha_F} \subseteq S' \times \Sigma_F \times S'$ the abstract transition relation, where $a \xrightarrow{x}_{\alpha_F} b$ iff there exists a $s_1 \in \alpha_F^{-1}(a)$ and $s_2 \in \alpha_F^{-1}(b)$ such that $s_1 \xrightarrow{y} s_2$, where $x = y = \lambda$ if $\lambda \in \Sigma_F$, else $x = \Delta$.*

Each symbol in Σ represents the start or termination of a specific task. In the set Σ_F only those elements of Σ are preserved which represent tasks included in the set F.

The transition relation of a resource STS $R = (S, S^0, C, \Sigma, \rightarrow)$ is always *left-total*, that is for every state $s \in S$ there exists a successor $t \in S$ such that $s \xrightarrow{\lambda} t$ for some $\lambda \in \Sigma$. Thus all words of a resource STS are infinite and there is no deadlock state. The abstraction function preserves this property.

Lemma 4 (Deadlock Free). *Let $STS = (S, S^0, C, \Sigma, \rightarrow)$ be an STS with a left-total transition relation. Then it holds that the transition relation of its abstraction $STS'_A = (S', S^0_{\alpha_F}, C_F, \Sigma_F, \rightarrow_{\alpha_F})$ according to Definition 15 is also left-total.*

Proof. Let $R = (S, S^0, C, \Sigma, \rightarrow)$ be an STS with a left-total transition relation and $R' = (S', S^0_\alpha, C_F, \Sigma_F, \rightarrow_\alpha)$ be the abstracted STS with task set F according to Definition 15. For $s' \in S'$ there exists a $s \in S$ such that $\alpha_F(s) = s'$ (Formula 3.12). As the transition relation of R is total, there is a $t \in S$ such that $s \xrightarrow{\lambda} t$ for some $\lambda \in \Sigma$. As α_F is surjective, there is a $t' \in S'$ with $\alpha(t) = t'$. According to the transition relation of Definition 15 it holds that $s' \xrightarrow{x}_\alpha t'$ with $x \in \Sigma_F$. □

Thus the abstract transition relation is defined in such a way that if there exists a state in the original STS, which has an outgoing transition to another state, there also exists a transition between the abstract representations of both states. With this the abstraction yields an over-approximation of the original STS, while preserving the exact timings for the dependent tasks.

Lemma 5 (Over-Approximation). *Let $STS_A = (S, S^0, C, \Sigma, \rightarrow)$ be an STS with a left-total transition relation. Then it holds that the transition relation of its abstraction $STS'_A = (S', S^0_{\alpha_F}, C_F, \Sigma_F, \rightarrow_{\alpha_F})$ according to Definition 15 is an over-approximation of STS_A.*

To proof that the abstraction leads to an over-approximation, first a simulation relation between STSs has to be defined to be able to determine when an STS has more behavior than another STS.

Definition 16 (Simulation relation). *Given two symbolic transition systems* $STS_1 = (S_1, S_1^0, C_1, \Sigma_1, \rightarrow_1)$ *and* $STS_2 = (S_2, S_2^0, C_2, \Sigma_2, \rightarrow_2)$ *with* $C_1 \subseteq C_2$ *and* $\Sigma_1 \subseteq \Sigma_2$*. A relation* $sim \subseteq S_2 \times S_1$ *is a simulation relation between* STS_1 *and* STS_2*, if for all* $(\langle l_2, D_2\rangle, \langle l_1, D_1\rangle) \in sim$ *the following holds:*

1. $D_1 = D_{2|C_1}$.
2. *For all* $t \in S_2$ *with* $\langle l_2, D_2\rangle \xrightarrow{\lambda}_2 t$ *there exists a* $s \in S_1$ *such that* $\langle l_1, D_1\rangle \xrightarrow{\lambda'}_1 s$ *and* $(t, s) \in sim$*, and* $\lambda = \lambda'$ *if* $\lambda \in \Sigma_1$*, else* $\lambda' = \Delta$.

Two symbolic transition systems STS_1 *and* STS_2 *are simulation equivalent, in short* $STS_2 \preceq STS_1$*, if there exists a simulation relation* $sim \subseteq S_2 \times S_1$ *such that* $(\langle l_2^0, 0_{C_2}\rangle, \langle l_1^0, 0_{C_1}\rangle) \in sim$*.*

It is said that the concrete STS is simulated by the abstract STS. The proof of the over-approximation property is closely related to the proof of the left-total property:

Proof. (Lemma 5) Let $R = (S, S^0, C, \Sigma, \rightarrow)$ be an STS with a total transition relation and $R' = (S', S_\alpha^0, C_F, \Sigma_F, \rightarrow_\alpha)$ be the abstracted STS with task set F according to Definition 15. The simulation relation of the states is given by $sim = \{(s, s') \mid s \in S,\ s' \in S',\ \alpha(s) = s'\}$. Consider an element $(s, s') \in sim$, for which holds that $\alpha(s) = s'$. As R is total, there is a transition $s \xrightarrow{\lambda} t$. Then, two properties hold:

- The abstraction function is surjective, such that there is a $t' \in S'$ with $\alpha(t) = t'$.
- According to the transition relation of Definition 15 it holds that $s' \xrightarrow{\lambda'}_\alpha t'$ with $\lambda = \lambda'$ if $\lambda \in \Sigma_F$, else $\lambda' = \Delta$.

With this we can conclude that $(t, t') \in sim$. Note that this also holds for the initial states of both STSs. □

3.4.2. Composition Function

Let $A_i = (S_i, S_i^0, C_i, \Sigma_i, \rightarrow_i)$ for $i = \{1, ..., n\}$ with $n > 1$ be a set of STSs over pairwise disjoint clock sets C_i. In the following, the product construction $A = A_1 \times ... \times A_n$ is defined, which is itself an STS over the clock set $C = C_1 \cup ... \cup C_n$. For each created state of the product STS it is stored from which input states, i.e. states of the input STSs $A_1, ..., A_n$, it resulted. To this end for each A_i the function ξ_i is introduced that maps each product STS state to a state of A_i.

The initial state of the product is given by

$$\langle l^0, 0_C\rangle := \langle (l_1^0, ..., l_n^0), 0_{C_1} \cup ... \cup 0_{C_n}\rangle, \tag{3.14}$$

where $\langle l_i^0, 0_{C_i}\rangle$ is the initial state of A_i. Note that $\xi_i(\langle l^0, 0_C\rangle) = \langle l_i^0, 0_{C_i}\rangle$ for $i \in \{1, ..., n\}$.

According to Definition 6 the time successor $\langle l, D'\rangle$ of state $\langle l, D\rangle$ is then determined by

$$\langle l, D'\rangle := \langle l, D^{\uparrow} \cap D'_{1|_C^{-1}} \cap ... \cap D'_{n|_C^{-1}}\rangle \tag{3.15}$$

where $\xi_i(\langle l, D'\rangle) = \langle l_i, D'_i\rangle$ for $i \in \{1, ..., n\}$ are the time successors of the input states $\langle l_i, D_i\rangle$. Note that the zones from all STSs are extended to the global clock set C. Note also that always there is a unique time successor because all input states have a unique time successor as illustrated in Section 2.2.

Starting from the computed time successor to compute all possible discrete steps in the product transition system, *each* outgoing transition (with respect to the current location of the corresponding STS) from *each* STS is tried to be *fired*. In fact, this is also done in Definition 6: Whenever the guards of a transition are fulfilled, a discrete transition is enabled and can be fired. This is the case, when the intersection of the zone induced by the guard and the current zone is not empty. The maximal number of discrete successor transitions of a product automaton state is then the sum of the number of discrete successor transitions of each corresponding input state. The discrete successors of a state $\langle l, D\rangle$ of the product STS are given by the following set. The following definition is given for the composition of two components to keep the following discussions and proofs readable. Note that this is no restriction to the general case as it can be easily extended to $n-$components by adapting the constraint in the set definition.

$$\begin{aligned} dSucc(\langle l, D\rangle) := \{ \quad & \langle l[l_{\{i,j\}} \rightarrow l'_{\{i,j\}}], D'\rangle \\ & \mid \exists i, j \in \{1, ..., n\}, i \neq j. \langle l_{\{i,j\}}, D_{\{i,j\}}\rangle \xrightarrow{\lambda} \langle l'_{\{i,j\}}, D'_{\{i,j\}}\rangle \\ & \wedge \ \lambda \in \Sigma_i \cap \Sigma_j \ \wedge \ D' \neq \emptyset\} \end{aligned} \tag{3.16}$$

where $D' = (D \cap \rho^{-1}(D'_i \cap D'_j)_{|_C^{-1}})[\rho(D'_{\{i,j\}}) \rightarrow 0] \cap D'_{i|_C^{-1}} \cap D'_{j|_C^{-1}}$, and $\langle l_i, D_i\rangle = \xi_i(\langle l, D\rangle)$ and $\langle l_j, D_j\rangle = \xi_j(\langle l, D\rangle)$ for all $i, j \in \{1, ..., n\}$. The discrete step is possible, if the resulting zone is not empty. The function $\rho(D_{\{i,...,k\}})$ represents the set of clocks that are reset in the corresponding zones $D_i, ..., D_k$, and $\rho^{-1}(D_{\{i,...,k\}})$ represents the symbolic states *before* the reset operation of the corresponding transitions have been performed. Note that the set of clocks to be reset are originally defined on the input transitions, but this set can also be derived on the target zones.

To proof the correctness of the product construction, the notion of *equivalence* has first to be defined. The equivalence of several STSs is determined through the bisimulation relation defined as follows.

Definition 17 (Bisimulation relation). *Given two symbolic transition systems* $STS_1 = (S_1, S_1^0, C, \Sigma, \rightarrow_1)$ *and* $STS_2 = (S_2, S_2^0, C, \Sigma, \rightarrow_2)$ *over the same clock set* C *and alphabet* Σ. *A relation* $r \subseteq S_1 \times S_2$ *is a bisimulation relation between* STS_1 *and* STS_2, *if for all* $(\langle l_1, D_1\rangle, \langle l_2, D_2\rangle) \in r$ *the following holds:*

1. $D_1 = D_2$.

2. *For all* $s \in S_1$ *with* $\langle l_1, D_1\rangle \xrightarrow{\lambda}_1 s$ *there exists a* $t \in S_2$ *such that* $\langle l_2, D_2\rangle \xrightarrow{\lambda}_2 t$ *and* $(s,t) \in r$.

3. *For all* $t \in S_2$ *with* $\langle l_2, D_2\rangle \xrightarrow{\lambda}_2 t$ *there exists a* $s \in S_1$ *such that* $\langle l_1, D_1\rangle \xrightarrow{\lambda}_1 s$ *and* $(s,t) \in r$.

Two symbolic transition systems STS_1 *and* STS_2 *are bisimulation equivalent, in short* $STS_1 \approx STS_2$, *if there exists a bisimulation relation* $r \subseteq S_1 \times S_2$ *such that* $(\langle l_1^0, 0_C\rangle, \langle l_2^0, 0_C\rangle) \in r$.

The bisimulation relation defines an equivalence relation between symbolic transition systems. Thus, if two STSs are in a bisimulation relation, they satisfy the same properties. The focus of this thesis are properties concerning upper bounds of clocks like task deadlines and end-to-end latency constraints.

Lemma 6 (Satisfaction of Properties)**.** *A transition system* STS *over a clock set* C *fulfills a deadline formula* $\varphi \in \Phi(C)$, *in short* $STS \models \varphi$, *iff for all reachable states* $\langle l, D\rangle$ *of* STS *it holds that* $D \subseteq D_\varphi$.

The proof of this lemma is trivial: If no clock valuation exceeds the allowed bounds defined by the condition, the condition is fulfilled. The bisimulation relation preserves these properties.

Theorem 1. *Let* $STS_1 \approx STS_2$ *be two bisimulation equivalent transition systems and let* $\varphi \in \Phi(C)$. *It holds that* $STS_1 \models \varphi$ *iff* $STS_2 \models \varphi$.

Proof. ($\Rightarrow$) Only one direction of the theorem is shown, as the other direction is performed in an analogous way. Let $STS_1 \approx STS_2$ and a formula $\varphi \in \Phi(C)$ be given. Let $(s_0, ..., s_n)$ be a path of STS_1 with $s_0 = \langle l_1^0, 0_C\rangle$, $s_i \xrightarrow{\lambda_i}_1 s_{i+1}$ for all $i \in \{0, ..., n-1\}$, and $s_n = \langle l_{1,n}, D_{1,n}\rangle$ where $D_{1,n} \nsubseteq D_\varphi$. Then according to Definition 17 (Items 2. and 3.) there is a path $(t_0, ..., t_n)$ in STS_2 with $t_0 = \langle l_2^0, 0_C\rangle$ and $t_i \xrightarrow{\lambda_i}_2 t_{i+1}$ for all $i \in \{0, ..., n-1\}$, where $(s_i, t_i) \in r$ for all $i \in \{0, ..., n\}$ and $t_n = \langle l_{2,n}, D_{2,n}\rangle$. As $(s_n, t_n) \in r$ it holds that $D_{2,n} \nsubseteq D_\varphi$ (Item 3. of Definition 17). $\square$

Theorem 2. *Let* $A_1, ..., A_n$ *be a set of timed automata. Let further* $A = STS(A_1 \times ... \times A_n)$ *be a symbolic transition system from Definition 6 and* $B = STS(A_1) \times ... \times STS(A_n)$ *be the symbolic transition system as defined in this section. Then there exists a bisimulation relation such that* $A \approx B$.

Proof. Let $A = (S, S^0, C, \Sigma, \rightarrow_A)$ be an STS build in terms of the parallel composition Definition 6 of n timed automata and let $B = (X, X^0, C, \Sigma, \rightarrow_B)$ an STS build with respect to the product construction defined above. The bisimulation relation between A and B is inductively defined as follows:

- $(S^0, X^0) \in r$ with $D_{S^0} = D_{X^0}$ (base case),

- If $(s,x) \in r$ with $s \in S, x \in X$ and $s = \langle l_A, D_A \rangle, x = \langle l_B, D_B \rangle$ then $D_A = D_B$ (induction hypothesis). Two cases for the transition relation can occur:
  - s has a *time successor* defined as $s' = \langle l_A, D_A^{\uparrow} \cap D' \rangle$ with $D' = D_{I(l_1) \wedge \ldots \wedge I(l_n)}$, iff (according to the product definition) x has a time successor defined as $x' = \langle l_B, D'' \rangle$ with $D'' = D_B^{\uparrow} \cap D'_{1|_C^{-1}} \cap \ldots \cap D'_{1|_C^{-1}}$. With $D'_j = D_j^{\uparrow} \cap D_{I(l_j)}$ for $j \in \{1, \ldots, n\}$ we directly get $D' = D''$ and set $(s', x') \in r$ (inductive step).
  - For the discrete successors both directions need to be treated explicitly
    1. s has a *discrete successor* defined as $s' = \langle l'_A, D'_A \rangle$ with $D'_A = (D_A \cap D_{\varphi_j})[\rho_j \to 0] \cap D_{I(l'_A)}$ and $l'_A = l_A[l_j \to l'_j]$, iff for a $j \in \{1, \ldots, n\}$ the timed automaton A_j has a transition $(l_j, \lambda_j, \varphi_j, \rho_j, l'_j) \in R_j$. Then, the STS of A_j has a discrete successor $\langle l_j, D_j \rangle \xrightarrow{\lambda_j} \langle l'_j, D'_j \rangle$ with $D'_j = (D_j \cap D_{\varphi_j})[\rho_j \to 0] \cap D_{I(l'_j)}$. Per definition it holds $D_{A|C_j} = D_j$. With the product definition on STSs, the state x has also a discrete successor $\langle l_B[l_j \to l'_j], D'_B \rangle$ with $D'_B = (D_B \cap \rho^{-1}(D'_j)_{|_C^{-1}})[\rho(D'_j) \to 0] \cap D'_{j|_C^{-1}}$. With $D_A = D_B$ and $\rho^{-1}(D'_j) = D_j \cap D_{\varphi_j}$ we get $D'_A = D'_B$.
    2. x has a *discrete successor* defined as $\langle l_B[l_j \to l'_j], D'_B \rangle$ with $D'_B = (D_B \cap \rho^{-1}(D'_j)_{|_C^{-1}})[\rho(D'_j) \to 0] \cap D'_{j|_C^{-1}}$ iff there is an $STS(A_j)$ with $j \in \{1, \ldots, n\}$, which has a discrete successor $\langle l_j, D_j \rangle \xrightarrow{\lambda} \langle l'_j, D'_j \rangle$ with $D'_j = (D_j \cap D_{\varphi_j})[\rho_j \to 0] \cap D_{I(l'_j)}$. Per definition it holds that $D_{B|C_j} = D_j$ and $\rho^{-1}(D'_j) = D_j \cap D_{\varphi_j}$. Then, the corresponding timed automaton A_j has a transition $(l_j, \lambda_j, \varphi_j, \rho_j, l'_j) \in R_j$. If A_j has such a discrete transition, then s has a *discrete successor* defined as $s' = \langle l'_A, D'_A \rangle$ with $D'_A = (D_A \cap D_{\varphi_j})[\rho_j \to 0] \cap D_{I(l'_A)}$ and $l'_A = l_A[l_j \to l'_j]$. As $D_A = D_B$ holds per assumption, we directly get $D'_A = D'_B$.
  - When both 1. and 2. do hold, we set $(s', x') \in r$ (inductive step).

□

The function $\rho^{-1}(D)$ represents the symbolic state *before* the reset operation of the corresponding transition has been performed. Thus, in order to apply the product construction there is a need to store both the pre-reset state and the post-reset state of discrete successors. By using the simplified transition relation of Equation 3.4 this problem is avoided, as always the pre-reset state is stored and the post-reset state is computed when the successor is computed.

3.5. Analysis Algorithm

The operators introduced in the previous section enable the computation of interfaces between resources by applying abstraction and composing multiple STSs, if the inputs of the corresponding tasks are originated from different sources. After the computation of the input behavior of a resource, the computation of the resource STS is enabled. As a result, the resource STS contains the response times of the allocated tasks and end-to-end latencies, which are of interest, i.e. for which constraints are defined. The computation of such a resource STS is the topic of this section.

Note that end-to-end latency constraints are specified by two tasks here. The instantiation of the first task of a latency constraint, which is referred to as the *triggering task*, determines the point in time from which the constraint is assumed to be active, i.e. from where the time is started to be measured. The second task of a latency constraint is called the *tail task*. The termination of the tail task determines the point in time, up to which the latency shall be measured.

Before illustrating the resource computation concept, some important properties of STSs and their states have to be explained. First, analogous to the computation of the product, to be able to determine for each of the resource states from which state of the input transition system – in short “input state” – it resulted, the function ξ is introduced, which maps each resource STS state to an input STS state.

To manage the activation of tasks and resource allocations to task instances appropriately, all states of resource STSs are enriched by maps, lists, and queues. This is detailed in the next paragraphs. The symbol τ is used for task types, and t for task instances. The type of a task instance t is simply determined by $\tau(t)$.

Multiple Task Dependencies In this work it is assumed that independent tasks are triggered from *single* event streams. In contrast to this, *dependent* tasks may depend on a set of other tasks, i.e. the termination of tasks in this set activates the considered dependent task. Two activation types have to be considered, i.e. *synchronous* and *asynchronous* activations. Asynchronous activation means that whenever the considered task receives a trigger from one of the other tasks it depends on, it is activated. Synchronous means that from all tasks, from which the considered task depends on, at least a trigger has to be received to be released.

In Figure 3.11 the difference between both activation types is illustrated for a task τ_3 which depends two tasks τ_1 and τ_2: In the left part of the figure the synchronous activation is illustrated. Task τ_3 is activated, if it receives at least one trigger from both predecessors. Note that a trigger is received if a predecessor terminates. In the right part the asynchronous activation is illustrated. Task τ_3 gets activated, if it receives a trigger from either τ_1 or τ_2.

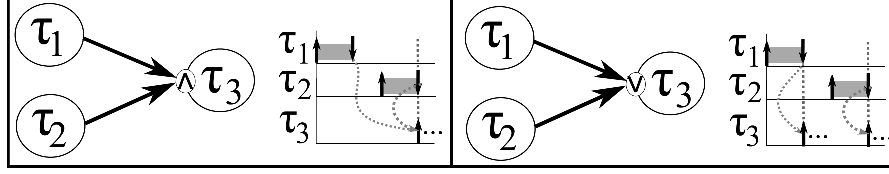


Figure 3.11.: Left: TDG with synchronous activation with resulting activation behavior; Right: TDG with asynchronous activation with resulting activation behavior.

The following discrete function will determine the activation type of a task.

$$act : \mathsf{T} \rightarrow \{sync, async\} \tag{3.17}$$

For independent tasks τ we define $act(\tau) := async$.

For tasks with synchronous activations all states s of a resource have a corresponding map $\mathcal{M}_{s,sync}$. In such a map all occurred triggers which are relevant for corresponding dependent tasks are added, i.e. $\mathcal{M}_{s,sync} : T \rightarrow \mathsf{T} \times ... \times \mathsf{T}$ for all $\tau \in T \subseteq \mathsf{T}$ with $act(\tau) = sync$. If at least a trigger event from each task type on which the corresponding tasks depend on is received, an instance of the corresponding task is activated. By activating such a task, the map is updated appropriately, i.e. an entry from each task, on which the dependent one expects a trigger, is removed from the map. As an example consider that a task τ_n synchronously depends on two tasks τ_1, τ_2. In state s_m three trigger of τ_1 occurred (by terminations of corresponding instances), but no trigger of τ_2 occurred so far. Thus, we have $\mathcal{M}_{s_m,sync}(\tau_n) = (\tau_1, \tau_1, \tau_1)$. By entering state s_{m+1} an instance of τ_2 terminates, and thus generates a trigger for τ_n. An instance of τ_n is generated and the map is updated to $\mathcal{M}_{s_{m+1},sync}(\tau_n) = (\tau_1, \tau_1)$.

For asynchronous activation patterns dependent tasks are triggered by the termination of each task they depend on. Thus, in contrast to the synchronous case, for asynchronous activations a map is not needed. Nevertheless, there is a need to be able to determine the task type from which a dependent instance was activated in order to correctly compute end-to-end timing latencies. As an example consider again the task τ_n which depends on τ_1 and τ_2. An end-to-end latency is defined between the start of τ_1 and the end of τ_n. To be able to determine from which task type an instance was activated in a state s, the function $\mathcal{M}_{s,async} : \bar{T} \rightarrow 2^{\mathsf{T}}$ is defined, where $t \in \bar{T} \subseteq \mathsf{T}$ with $act(t) = async$. Note that the set $\bar{\mathsf{T}}$ is a set of task *instances*. The function is defined recursively: If a task t_n triggers t_{n+1} then $\mathcal{M}_{s,async}(t_{n+1}) := \mathcal{M}_{s,async}(t_n) \cup \{\tau(t_n)\}$.

Consider again the example above: If $\mathcal{M}_{s_m,async}(t_n) = \{t_2\}$ and t_n triggers t_{n+1} by entering state s_{m+1} then $\mathcal{M}_{s_{m+1},async}(t_{n+1}) = \{t_2, t_n\}$. If there is an end-to-end latency constraint between t_1 and t_{n+1}, it is possible to determine that the instance t_{n+1} was not started by an instance of t_1.

Note that for the descriptions of the analysis algorithms the *dot-notation* for both functions $s.\mathcal{M}_{async} := \mathcal{M}_{s,async}$ and $s.\mathcal{M}_{sync} := \mathcal{M}_{s,sync}$ is used.

Controlling Boundedness If the attributes of a task are chosen inappropriately by the system designer, new instances of tasks may be released permanently before previously activated instances finish their computations. The simplest example for this occurs when the period is smaller than the execution time of the task. With such a system configuration the timing analysis approach of this thesis would not terminate and not deliver the requested response times. Thus, there is a need of a mechanism to determine such configurations. As detailed in Subsection 3.3.2 the maximal number of possible parallel activations of all tasks needs to be known a priori. These numbers are relevant to determine the needed number of response time clocks c_r, as one per instance is necessary.

The allocation of indexes to task instances is realized by a ring-buffer. Whenever a new instance of a corresponding task shall be released, an index is requested before. Thus, if no free index is available, the approach is able to determine that the system is unbounded under the assumption that the bound is computed according to Subsection 3.3.2.

For tasks, for which the response time is not relevant, the response time clocks are omitted to keep the state space as small as possible. Note that the engineer specifies which response times are of interest and which are not. To be able to determine the boundedness for instances of such tasks, each state s counts the number of active instances in a corresponding function, i.e. $\mathcal{B}_s : \mathsf{T} \to \mathbb{N}$. If the number exceeds the specified bound, the violation of the boundedness can be determined.

Relating Matrix Indexes As mentioned before, the valuations of all clocks are represented by matrices. Each clock is referenced by a corresponding index. Clocks may be shared between a set of STSs. For example, consider a resource STS: The valuations of some of its clocks are directly determined by the clock valuations of the input states, as for example the clocks with which the periodic activation of independent tasks is determined. As an iterative analysis approach is applied, the indexes of shared clocks in the matrices of different STSs may change. Thus, there is a need to relate the indexes of shared clocks of different STSs. Let $\mathscr{C}$ be the set of all clocks of the considered system architecture. It is assumed that $\mathscr{C}$ is totally ordered, i.e. the set is antisymmetric, transitive, and total. Let $\mathcal{D}$ be the set of all zones over all subsets of $\mathscr{C}$. The clocks in the matrix representation of a zone $D \in \mathcal{D}$ may be arbitrarily ordered. To obtain a matrix which adheres to the order relation of $\mathscr{C}$, the function $\zeta : \mathcal{D} \to \mathcal{D}$ is defined reordering a zone D according to the total order relation. To simplify the readability, further $\tilde{D} := \zeta(D)$ is defined as the short notation.

An example is given in Figure 3.12 for a resource on which two independent tasks are allocated. The matrices of the input STS of the resource consist of the period clocks of the tasks, as illustrated in the left-most matrix in the figure. The period clock of τ_1 has

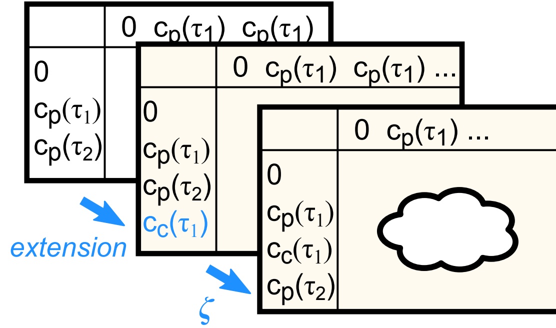


Figure 3.12.: Example for relating indexes of clocks of different matrices.

the index 1, the period clock of τ_2 has the index 2. When considering resources, further clocks as described in previous sections are available. Thus, the matrix is extended by such clocks as illustrated in the center of the Figure 3.12. In the right-most figure the reordering is illustrated.

Note that the usage of this function is restricted to matrices containing the same clocks, which is fine for our purposes. A matrix will be adapted to the clock set of another matrix by the application of the zone projection operations of Subsection 3.4.1, before both matrices are set into relation.

Next, the algorithm which computes the STS of a resource and all necessary operations such as the computation of successors of a resource state is presented.

3.5.1. Main Analysis Algorithm

The general idea to compute the STS of a resource is to extend its input STS by the behaviors of the tasks which are allocated on this resource, and the considered scheduling policy.

In Listing 3.1 the main algorithm for the computation of the STS of a resource $r = (M, \mathcal{T}, Sch, \mathcal{R}, \mathcal{A})$ for a given input STS STS_{in} is illustrated. The algorithm starts by creating the initial symbolic state $s_0 = \langle l^0, C_0 \rangle$ of the resource STS. This state is added to a queue Ψ, which determines the states, for which successors have to be computed. Thus, the algorithm works in a breadth-first manner, where first all successors of a state are computed before the next state is considered. Whenever the successors of a state are computed, this state is removed from the queue. The exit condition of the algorithm is reached, when no states are left in the queue Ψ.

Listing 3.1: Main code *computeResourceSTS(* $STS_{in} = (S_{in}, S^0_{in}, C_{in}, \Sigma_{in}, \rightarrow_{in})$ *).*

```
1 queue Ψ := {⟨l^0, 0_C⟩}
```

```
2  ξ := { (⟨l^0, 0_C⟩, S^0_in) }
3  while( Ψ.size > 0 )
4      ⟨l, D⟩ := Ψ.dequeue(), ⟨l_in, D_in⟩ := ξ(⟨l, D⟩)
5      forall( edge_in := ⟨l_in, D_in⟩ -λ->_in s with s ∈ S_in )
6          set Ψ' := computeSuccessor(⟨l, D⟩, edge_in, ξ)
7          forall(s ∈ Ψ') Ψ.enqueue( s )
8          if ( checkTimings(Ψ') fails) exit
9  if( existsDependentResource() ) mergeBisimStates()
```

In Line 2 the initial state S^0_{in} of the input transition system is set as the corresponding input state of $\langle l^0, 0_C \rangle$.

The possible successors of a state $\langle l, D \rangle$ are mainly determined by the successors of the corresponding state of the input STS. This input state is determined by the map ξ in Line 4. The set of outgoing transitions of the determined input state defines time successors and the set of tasks, which can be released.

Each state which is generated is added to the queue Ψ. For all newly generated states Ψ' it is determined whether a timing constraint $d_t \in \mathbb{N}_{\geq 0}$ of a task t is violated as follows.

$$\forall \langle l, D \rangle \in \Psi', t \in \mathcal{A}_{\langle l,D \rangle} \cup \mathcal{R}_{\langle l,D \rangle} : \; D_{r(t),0} \leq d_t. \tag{3.18}$$

An analogous check is performed for end-to-end latency constraints, i.e. all clocks measuring end-to-end latencies are checked against the corresponding constraint values. If Equation 3.18 or any end-to-end latency constraint is violated, or no state is left in Ψ, the algorithm terminates. Note that for the sake of readability, tasks are used as indexes of matrices: For a task τ the notation $D_{p(\tau)}$ is used to refer to $c_p(\tau)$, i.e. to the clock tracing the periodical activation the task. With $D_{p(\tau),0}, D_{0,p(\tau)}$ it is referred to its upper and lower bound respectively. Analogously, $D_{c(\tau)}$ refers to the computation time clock of τ. For a task instance t with $D_{r(t)}$ it is referred to $c_r(t)$, i.e. the clock measuring the accumulated response time.

After the computation of the resource STS, the state space is minimized by the application of an extended version of the bisimulation relation of Definition 17 in Line 9. The minimization is only performed if there are resources which depend on the output behavior of the current resource to omit unnecessary computations. The relation used for minimization is detailed in Subsection 3.5.5. Note that this step can only be performed after the computation of the complete STS, as all outgoing transitions of all states have to be known in order to determine the equivalence of two states.

Next, the successor computation function is introduced.

3.5.2. Successor Computation

Observing the STS transition relation of Definition 6 reveals that the successor computation consists of two steps, i.e. the computation of the *reachable time successor*

determined by the invariants of the discrete locations, and the discrete step where tasks are triggered. The computation of the reachable time successor affects the upper bounds of a zone, while the discrete step affects the lower bounds.

When considering resources, two sources of timing invariants are given: The first type of invariants stems from the periodical activations of tasks. Whenever the period clock of a task reaches its period value, an instance of the task has to be released. The second type of invariant stems from the accumulated response time of the running task. The time a running task instance is able to finish its computation as enough computation time has been allocated to it, this task shall not continue its computation in a successor state but shall terminate.

In the following, the algorithms computing the *time successor* and computing the effect of taking a *discrete step* are illustrated. Note that for an interval i the functions $i.lb(), i.ub()$ are used to access the lower and upper bound of i.

Time Successor

The computation of a time successor of a resource state $s := \langle l, D\rangle$ is illustrated in Listing 3.2. According to Definition 6 to compute the time successor first the upper time bounds of the current zone are removed in Line 1 by computing $D^{\uparrow}$. As the simplified version of STSs is applied, before the upper bounds are removed, clocks according to Formula 3.4 are reset. This is realized by equipping each state s by a set $\breve{\varrho}$, where all clocks to be reset are added. These clocks are added the time, when s is computed as a successor as detailed later in the *takeDiscreteStep* method. The set $\breve{\varrho}$ is accessed by the dot notation $(s.\breve{\varrho})$.

In Line 2 the resulting zone of this operation is intersected by invariants defined by the input state. The zone D^{in} denotes the zone of the successor state of $\xi(\langle l, D\rangle)$, and C, C_{in} denote the set of clocks of the resource STS and the input STS respectively. The upper bound timing constraints of the input STS are needed in order to prevent missing any task activations, as the input defines the release times of all allocated tasks, or tasks, from which dependencies exist. For the set of all clocks which are defined in both the input STS and the resource STS the upper bounds are set to the ones defined in the input state. If a clock c has the value zero and is reset on the input edge, then this reset is also performed for the resource successor.

Listing 3.2: Listing for method $D' :=$ *invIntersection(* $s := \langle l, D\rangle$, D^{in} *)*.

1 $D' := (D[s.\breve{\varrho} \rightarrow 0])^{\uparrow}$
2 **forall** $(c \in C_{in} \cap C)$ $\tilde{D}'_{c,0} := \tilde{D}^{in}_{c,0}$
3 **if** $(\mathcal{A}_s.size > 0)$
4   task t := $\mathcal{A}_s$. first
5     $D'_{c(t),0} := wcet_{\tau(t)} + \mathcal{A}_s(t).ub()$
6 $D'[\varrho_{unused} \rightarrow 0]$

If there is a running task instance t in the current resource state s, a further invariant is given for the successor to prevent missing the finish of computation of t. Thus, in Line 3 it is checked whether there is a running task by determining the size of the active task map $\mathcal{A}_s$. As the map is sorted, the first entry determines the running task instance t. The worst-case response time is determined in Line 5 by computing the sum of the worst-case execution time of the corresponding task type and the interrupt time, which is determined by the corresponding entry in the active task map, i.e. $\mathcal{A}_s(t)$. The result serves as an upper bound constraint for the computation time clock of t.

In Line 6 all clocks are reset, which are currently not in use, e.g. clocks measuring the allocated computation times of inactive tasks.

Discrete Step

In Listing 3.3 the procedure of the discrete step is illustrated. In contrast to the invariant intersection method where time passes up to the allowed time bounds, the discrete step defines the occurrence of some events in the alphabet of the considered resource, or more specific the release or the termination of task instances. As it was detailed earlier, some tasks may be included in the alphabet of a resource, but are not allocated to this resource. This is for example the case to appropriately handle task dependencies and avoid approximations by abstracting the clocks of these tasks. Thus, it is necessary to distinguish whether such tasks are allocated to the considered resource or not in order to adequately handle the resource task lists. For this, the already introduced function $\mathcal{T} : \mathsf{T} \to \mathcal{B}$ is used. Note that if the input edge defines a time successor the default event will be given, i.e. $\tau = \Delta$. For this, we set $\mathcal{T}(\Delta) := false$.

Listing 3.3: Method $\langle l', D' \rangle$:= *takeDiscreteStep*($\langle l, D \rangle$, $\langle l'_{in}, D'_{in} \rangle$, *taskType* τ, *task* t, *bool* $isTerm$).

```
1  if (𝒯(τ) ∧ isTerm)
2        forall (task t' ∈ 𝒜⟨l,D⟩ with τ(t') ≠ τ)
3              𝒜⟨l',D'⟩(t') := 𝒜⟨l,D⟩(t') + [bcet_τ, wcet_τ]
4        D'_{0,c(t)} := bcet_τ + 𝒜⟨l,D⟩(t).lb()
5        Sch(𝒜⟨l',D'⟩ := 𝒜⟨l,D⟩\{t}, ℛ⟨l',D'⟩)
6        if (not timingRelevant(τ)) ℬ⟨l',D'⟩(τ).decrement
7        ⟨l', D'⟩.freeE2EIndexes(τ)
8  else
9        D' := D̃ ∩ ζ( ( D'_in )|_C^{-1} )
10       if( τ ≠ Δ ∧ 𝒯(τ) ) l'_τ := (l'_in)_τ
11 if( τ ≠ Δ ) instantiateNewTasks(τ, t, ⟨l', D'⟩, ⟨l'_in, D'_in⟩, isTerm)
12 setClocksToBeReset(⟨l', D'⟩.ϱ̆)
```

If the statement in Line 1 evaluates to *true*, i.e. the task t is allocated to the resource and shall terminate its execution in the successor state, the following is performed: First, the execution time of t is accumulated to all interrupted tasks by incrementing their interrupt times in the active task map $\mathcal{A}_{\langle l,D \rangle}$ (Line $2-3$). This is analogous to classical scheduling analysis methods, where response times are computed by a fixed point equation propagating the interruptions of a task. Note that the interrupt times of only those tasks are incremented, which have been fully interrupted by t. All tasks, which were instantiated after the start of t are kept in the ready task list $\mathcal{R}_{\langle l,D \rangle}$. For instance, consider the following scenario in the system of Figure 3.4: Let t_3^1 be the running task instance in a state s_i, which gets interrupted in the successor state s_{i+1} by a new instance t_4. Then, by entering the next successor s_{i+2} a further task instance t_3^2 gets instantiated. The active task map $\mathcal{A}_{s_{i+2}}$ thus includes both tasks $\mathcal{A} = \{t_4 \mapsto [0,0], t_3^1 \mapsto [0,0]\}$, while the ready task list has a single entry, i.e. $\mathcal{R}_{s_{s+2}} = \{t_3^2\}$. In state s_{i+3} task t_4 terminates and the resulting interrupt time for task t_3^1 is put in the active task map: $\mathcal{A}_{s_{i+3}} = \{t_3^1 \mapsto [12,12]\}$.

As t shall finish its computation, the zone of the successor gets the new constraint $c(t) \geq bcet_\tau + \mathcal{A}_{\langle l,D \rangle}(t).lb()$ on Line 4, which corresponds to the minimal amount of computation time needed to terminate the task. This minimal time is the sum of the best-case execution time and the minimal interrupt times in the current state. Note that here only the lower bounds of the clocks have to be handled, as the upper bounds are already handled by the invariant intersection operation.

In Line 5 the task t is removed from the active task map, and the next running task is determined by the usage of the scheduling policy. When performing $\mathcal{A}_{\langle l,D \rangle} \backslash \{t\}$ also the index of the task is released, if a response time clock was used for t. If not, the corresponding variable value in Line 6 is decremented. In Line 7 the indexes for the end-to-end latency clocks are handled, i.e., if t is the *tail task* of the latency constraint (the end point where the end-to-end latency is measured) the corresponding index is freed.

If the statement in Line 1 evaluates to *false*, the successor of the corresponding input state defines either a start of a task or the termination of a task not allocated to the current resource. If the task on the current edge of the input STS is not relevant for the resource, also input edges with the Δ-event are needed to be handled by the algorithm. This event occurs, when previous abstractions of the input STS were performed. Such edges do not instantiate tasks but define timing constraints on the successors of the current state. In all cases, the input successor defines constraints for clocks $C_{in} \cap C$ which have also to be taken into account for the resource successor, as it is performed in Line 9 by taking the intersection of D and D'_{in}. If further the input defines the start of a task allocated to the resource, the position of the corresponding location vector is switched in Line 10.

By starting or terminating tasks, these have also to be instantiated and the resource task lists have to be managed. This is realized by the method *instantiateNewTasks*, which is illustrated in the next paragraph.

In Line 12 all clocks which need to be reset in the successor state $\langle l', D' \rangle$ are collected

by adding the corresponding clocks to a set. As stated before, each state s is equipped by a set $\breve{\varrho}$ to which the following clocks are added in Line 12 :

- $c_p(\tau)$, if an independent task of type τ is instantiated,
- $c_r(t)$, if a task t is instantiated and its timing is relevant, or t is terminating,
- $c_c(\tau)$ if a task t of type τ
  - is instantiated,
  - terminates,
  - or was in the ready list $\mathcal{R}$ and gets running in the successor state.

Further, if a task is instantiated, on which occurrence an end-to-end latency constraint starts, and the corresponding *tail task* is allocated on the resource, the corresponding clock measuring the latency is added to the set $\langle l', D' \rangle.\breve{\varrho}$. Also, when such a *tail task* allocated on the resource finishes its computation, the corresponding clock is added to this set.

Please also refer to Subsection 3.3.2 where this topic of clock resets is detailed and exemplified through a set of scenarios.

Instantiate Tasks

Tasks can either be instantiated by the triggering of an event stream, or the termination of a task they depend on. In the Lines 1−2 of Listing 3.4 this case distinction is performed and the set of tasks S which shall be instantiated is determined. If the task τ_{in} is not a terminating one, then only τ_{in} is added to S. In the other case, if the considered task is a terminating one, the set of triggered tasks is determined in Line 2.

Listing 3.4: Listing *instantiateNewTasks(taskType τ_{in}, task t_{in}, state s', state s'_{in}, bool isTerm)* .

```
1  if(¬isTerm)  S := {τ_in}
2  else  S := τ_in.triggeredTasks
3  forall(taskType τ ∈ S)
4      if( 𝒯(τ) )
5          if(act(τ) = sync)
6              s'.ℳ_sync(τ).push(τ_in)
7              isActive := checkActivation(s'.ℳ_sync, τ)
8          else
9              isActive := true
10         if(isActive)
11             s'.requestE2EClock(τ)
```

```
12              if(timingRelevant(τ)) s'.getFreeIndex(τ)
13              else B_{s'}(τ).increment
14              Sch(A_{s'}, R_{s'}.add( t := createInstance(τ) ) )
15              if(τ_in ≠ τ ∧ act(τ) = async) M_async(t) := {τ_in} ∪ M_async(t_in)
16      else if( τ ∈ R_{s'_in} ∪ A_{s'_in} ) s'.requestE2EClock(τ)
```

All triggered tasks are then iterated: In Line 4 it is checked if the considered triggered task is allocated on the resource. If the triggered task τ depends on other tasks and has a synchronous activation condition, it is checked whether the triggered task can be activated (Line 5). In Line 6 the trigger τ_{in} is added to the set $\mathcal{M}_{sync}(\tau)$ of the successor state s'. Then, it is checked whether in this set there is already an entry of all tasks, from which τ depends on by using the method *checkActivation*. This method is not further described as it involves only technical issues. If the method returns *true*, τ is considered to be activated. In case of an independent task or a task with an asynchronous activation condition such a check has not to be performed. The task is considered to be active directly (Line 9).

If it is determined that the task is ready to be activated (Line 10), it is checked whether an end-to-end latency clock index is needed in Line 11 by calling the method *requestE2EClock*. This method returns an index, if *i*. there is an end-to-end latency constraint starting from the triggered task τ, and *ii*. if the *tail task* is allocated on the current resource. The corresponding clock with the returned index is used to trace the end-to-end latency starting in the successor state s'.

If the response time of τ is relevant a clock index is requested in Line 12 which traces the response time of the instantiated task starting in the successor state. If not, the corresponding value in the map $\mathcal{B}_{s'}$ of the successor state is incremented. In Line 14 a new instance of τ is created and added to the ready task list $\mathcal{R}_{s'}$. The scheduling policy is then applied to determine the next running task. Note that if a task instance t is moved from the ready map to the active map, i.e. a previously released task which did not get computation time so far, its computation time clock $c_c(t)$ is added to the reset set $\check{\varrho}$ of the successor in method *takeDiscreteStep* as illustrated before.

If τ is not independent and activated by an event stream as indicated in Line 15 by the condition $\tau_{in} \neq \tau$, and the activation condition of τ is asynchronous, then the function $\mathcal{M}_{async}(t)$ is extended by the triggering task τ_{in} and all tasks, which activated the triggering task instance t_{in}. As already stated this is needed to correctly evaluate end-to-end latencies.

If the triggered task τ is not allocated on the resource, it is determined whether the trigger leads to a task instance activation in the target input state s'_{in} in Line 16. If an instance is activated in s'_{in}, it is checked whether an end-to-end latency clock is needed analogously to the case in Line 11, and if so, an index is requested.

Successor Computation

The previously introduced methods to compute the time successor and the discrete step are called from the surrounding method *computeSuccessor* illustrated in Listing 3.5, which computes a successor of a resource state $s = \langle l, D \rangle$ with respect to its corresponding input state s_{in} and an outgoing edge of s_{in}.

Listing 3.5: Method $\Psi := computeSuccessor(\ s := \langle l, D \rangle,\ s_{in} \xrightarrow{\lambda}_{in} \langle l'_{in}, D'_{in} \rangle,\ \xi\ )$.

```
1  D' := invIntersection(⟨l,D⟩, D'_{in})
2  if(A_s.size = 0 ∧ \tilde{D}' ∩ ζ( ( D'_{in} )_{|_C^{-1}}) ≠ ∅)
3       bool isTerm := A_{s_{in}}.first.isTerm(⟨l'_{in}, D'_{in}⟩)
4       s_2 := takeDiscreteStep(⟨l,D'⟩, ⟨l'_{in}, D'_{in}⟩, λ, A_{s_{in}}.first, isTerm)
5       s.addSuccessor(λ, s_2, ξ)
6  else if( A_s.size > 0 )
7       task t := A_s.first
8       [bcrt_t, wcrt_t] := [bcet_{τ(t)}, wcet_{τ(t)}] + A_s(t)
9       handlePossibleCases([bcrt_t, wcrt_t], λ, ⟨l,D'⟩, ⟨l'_{in}, D'_{in}⟩ )
```

First, the reachable time successor is determined for state s with respect to the considered successor of the corresponding input state s_{in} by calling *invIntersection* of Listing 3.2. This operation leads to the zone D'.

In Line 2 it is determined whether there exists a running task in the current state s by checking the size of the active task map, and whether the discrete step defined in the input successor leads to a valid zone by taking the intersection of D' and D'_{in}. If this intersection leads to an empty set, there exists no clock valuation which can fulfill all clock constraints defined in this resulting zone.

Thus, if the input successor cannot be fired and – taking the *else*-part of Line 6 into account – there is no running task, this input edge is skipped and the next input edge is considered in Listing 3.1. Otherwise, if the condition in Line 2 evaluates to true, the successor resource state s_2 is built by calling the method *takeDiscreteStep* of Listing 3.3. Note that in Line 2 actually the projection operation $D'_{in|C'}$ with $C' = C_{in} \cap C$ has to be performed in general, but as the input is abstracted in such a manner that it only preserves all relevant clocks, this operation can be omitted here.

In Line 3 the running task instance in the input state is determined by looking up the first entry of the active task map. Then it is checked whether this task is terminating in the target state, or defines a task start on the input edge. To determine whether the event t on an input edge defines the termination of a task instance or the triggering of an independent task the method $t.isTerm(s)$ is called, which returns *true* if t terminates in the (target) state s, and *false* if the edge defines a task instantiation of an independent task. Note that $\mathcal{A}_{s_{in}}.first$ may be NULL in the case that there is no running task instance in the input state. This is the case if the input defines an event stream behavior. The variable is set to *false* in this case.

If the computed successor does not exist yet, it is added to the graph by building an edge from the current state to the computed successor, and $\xi(\langle l', D'\rangle) = \langle l'_i, D'_i\rangle$ is set. Further, the created successor is added to the state set Ψ' (not shown in the code), indicating that the successors of this state have to be computed.

If the condition in Line 2 evaluates to false, but there is a running task (Line 6) in the current resource state, it has to be determined whether this task can terminate *before* the discrete step from the input graph is taken. For this, first the response time in Line 8 is computed by taking the *sum* of its execution time and its interrupt time. The *handlePossibleCases* method then determines the possible successors.

Handle Possible Cases

The termination of a running task is determined in the code fragment of Listing 3.5 by calling the *handlePossibleCases* method, which operates as follows:

1. If the running task t of type τ cannot terminate, i.e. $D'_{c(\tau),0} < bcrt_\tau$, the intersection between the zone of the state $s := \langle l, D'\rangle$ and the zone of the input state $\langle l'_{in}, D'_{in}\rangle$ is computed, and a new task instance is released, if the input edge defines a task start. For this, $s_2 := \texttt{takeDiscreteStep}(\langle l, D'\rangle, \langle l'_{in}, D'_{in}\rangle, \lambda, \mathcal{A}_{s_{in}}.first, isTerm)$ is called, where $isTerm := \mathcal{A}_{s_{in}}.first.isTerm(\langle l'_{in}, D'_{in}\rangle)$. Two cases can occur here:
   a) If this intersection is equal to the empty set, then the discrete step cannot be taken. The intersection would result in an invalid zone, where there exists no clock valuations which fulfill all clock constraints.

      Thus, the considered input edge cannot be fired in the current resource state and is skipped. This means that the running task has first to finish its computation before the input edge can be taken. The program flow is returned to the main algorithm then, where the next input edge is considered.
   b) Otherwise, if the intersection is not empty, a new instance is released if the input edge defines a task start in the current resource, i.e. if $\lambda \neq \Delta$ and λ triggers an independent task allocated to the resource, or the input edge defines a terminating task triggering another task allocated to the resource. To release new tasks, the method $instantiateNewTasks(\lambda, \mathcal{A}_{s_{in}}.first, \langle l, D'\rangle, s'_{in}, isTerm)$ is called. At least the new edge is added by calling s.**addSuccessor**(λ, s_2, ξ), where ξ is extended by $\xi \cup \{(s_2, \langle l'_{in}, D'_{in}\rangle)\}$.
2. If a running task instance t has to terminate as $D_{0,c(t)} = wcrt_t$ holds, then $s_2 :=\texttt{takeDiscreteStep}(\langle l, D'\rangle, -, \tau(t), t, true)$ is called. Note that the target input state is not relevant in this case, as the input successor will define the potential successors of s_2. Again, the new edge is added by calling s.**addSuccessor**($\tau(t)$, s_2, ξ), where ξ is extended by $\xi \cup \{(s_2, s_{in})\}$. At least, the recursion *computeSucces-*

$sor(s_2, s_{in} \xrightarrow{\lambda}_{in} \langle l'_{in}, D'_{in} \rangle, \xi)$ is performed. Thus, the successor of the input is tried to be fired at the computed successor s_2.

If $D'_{c(\tau),0} \geq bcrt_\tau \wedge D'_{0,c(\tau)} \leq wcrt_\tau$ holds then both orderings are possible, i.e. the task instance *may* terminate before the input is fired, or vice versa first the input is considered and the task instance terminates in the next successor. To realize this, both 1. and 2. are executed in this order.

3.5.3. Completeness and Soundness of Algorithm

In this section it is shown that the scheduling analysis algorithm introduced in the previous subsections is *correct* with respect to determined timing violations. Correctness refers to the general terms of algorithmic correctness, i.e. that the algorithm is sound and complete.

Theorem 3. *The timing analysis algorithm is sound and complete. If a timing violation is determined, it is a valid one (soundness), and if there exists a timing violation, the algorithm will find a path to a state where timing requirements are violated (completeness).*

The input of a resource on which only independent tasks are allocated is correct by the Theorem 2. The activations of these are represented by timed automata, which result in non-complex STSs as illustrated in Figure 3.9. As the single STSs are trivially correct, only the product of these has to be computed. So, the more general case where resources contain dependent tasks is considered in the following proof.

Proof. (Soundness) Assume that the algorithm terminates and returns a wrong answer, i.e. it determines that a task t violates its timing constraint d, but in real it does not. Thus, the algorithm has found a trace prefix $\sigma = \sigma_1, ..., \sigma_n$ $(n > 1)$ with a corresponding state sequence $s = s_0, ..., s_n$ leading to a state $s_n = \langle l, D \rangle$, where it holds that $D_{r(t),0} > d_t$, (compare Equation 3.18).

Assume that t is a task getting instantiated in state s_i, $i \in [1, ..., n-2]$. The theorem is proofed by performing the following case distinction on the status of the instantiated task.

Case 1: Consider first the case that task t does not get interrupted by any task after instantiation in state s_i. When the task is instantiated, both clocks $c_c(t), c_r(t)$ are reset, and not reset in the successors up to state s_n. As all clocks progress with the same rate, both clocks have always the same valuation in all successor states of s_i. When computing the zones of all states $s_i, ..., s_n$, always the intersection with the upper bound constraint $wcet_{r(t)} + \mathcal{A}_{s_j}.ub$ where $i \leq j \leq n$ is performed in method *invIntersection* (Line 5). As t is never interrupted, $\mathcal{A}_{s_j}.ub$ equals always to zero. Thus, if $D_{r(t),0} > d_t$ holds in state s_n it must also hold that $wcet_{r(t)} > d_t$, thus the counter example is valid.

Case 2: Assume again that the task t gets the status of *running* after instantiation. As in the first case both clocks $c_c(t), c_r(t)$ are reset, and not reset in the successors up to

state s_n. Further, assume that in the following transitions a set of higher priority tasks $hp(t)_1, ..., hp(t)_m$ with $m \geq 1$ are instantiated, thus interrupting the execution of t. Let the priorities of the $hp-$tasks be determined by their index. Let s_j with $j > i$ be the first successor where t is interrupted by $hp(t)_m$.

Assume that there is a sequence of *hp*-task activations with increasing priority, i.e. $s_i \xrightarrow{hp(t)_m} s_{i+1} \xrightarrow{hp(t)_{m-1}} \ldots \xrightarrow{hp(t)_1} s_{i+m}$, in such a way that no task terminates before the next one is activated. This is the case if the activation time is less than the *bcet* of the current running task. Thus, when computing the successor of state s_k with $k \in [i+1, ..., i+m-1]$ first the reachable time successor in *invIntersection* is determined. Then in method *handlePossibleCases* it must hold that $c_c(hp(t)_j) < bcrt_{\tau(hp(t)_j)}$ for $1 \leq j \leq m$ to create the above state sequence. Note that *bcrt* corresponds to the *bcet* here, as the hp-tasks were not preempted before the next higher priority task is instantiated. From state s_{i+1} up to state s_{i+m} at most $ts = \Sigma_{j=1}^{m} bcet_{\tau(hp(t)_j)} - \varepsilon$, $\varepsilon \rightarrow 0$ time units pass.

Case 2a: As each clock is incremented with the same rate, $c_{r(t)}$ is also incremented with exactly this amount of time (ts). Thus, if the timing constraint violation occurs in state s_{i+m}, it must be correct.

Case 2b: Assume the timing constraint violation does not occur after the activation of $hp(t)_1$. Then, each time an *hp*-task $hp(t)_j$ terminates in state s, in Line 3 of method *takeDiscreteStep* the interrupt time of each interrupted task in the $\mathcal{A}_s$ is incremented by the execution time of the terminating task $hp(t)_j$. Thus, when all hp-tasks terminate, and the timing constraint of t is not violated so far, t gets running again in the reached state s_x with $i + m < x < n$. When the successor of s_x is computed the upper bound of the computation time clock of t is determined in *invIntersection* (Line 5) by the sum of the *wcet* of the task and the upper bound of the accumulated response times determined in $\mathcal{A}_{s_x}$. If the upper bound clock valuation violates the timing constraint, it is a valid violation as all interrupts were accumulated appropriately.

Case 3: Assume there are already higher priority tasks $hp(t)_1, ..., hp(t)_m$ in the active task map $\mathcal{A}_{s_i}$ when t is instantiated. Further, assume that the running task has already run $[lb_1, ub_1]$ time units. First, the task t is added to the ready list $\mathcal{R}_{s_{i+1}}$. Both clocks $c_c(t), c_r(t)$ are reset, but in contrast to the previous cases only $c_r(t)$ is not reset in the successors up to state s_n. The computation time clock $c_c(t)$ is reset up to the state, where t gets the status *running.* Thus, whenever a successor is computed, (among other clocks) the clock $c_r(t)$ keeps progressing and tracing the time since t was instantiated. As already illustrated above, this progress takes place in the *invIntersection* method at Line 2 and 5, and in method *takeDiscreteStep* in Line 4 if an *hp*-task is terminating, or in Line 9 if a time successor is taken or a new task is released. The time, t gets running, it is determined by the clock $c_c(t)$ whether its computation can be finished or not. The clock $c_r(t)$ traces the time since task instantiation, i.e. has progressed as follows: $c_r(t) := \Sigma_{j=2}^{m}([bcet_{\tau(hp(t)_j)}, wcet_{\tau(hp(t)_j)}]) + [bcet_{\tau(hp(t)_1)}, wcet_{\tau(hp(t)_1)}] - [lb_1, ub_1]$. Thus, again if the timing constraint is violated, this violation is valid.

□

Proof. (Completeness) Assume that there exists a state of the system under analysis, for which the timing requirements of a task t is violated, but not found by the algorithm. Let $\varphi_0, \cdots \varphi_n$ be a task activation sequence leading to a schedule where an activated task instance t misses its timing constraint. It has to be shown that no matter what this activation sequence concretely looks like, the algorithm detects a timing constraint violation caused by such a trace.

A similar case distinction as above is performed.

Case 1: Assume that t is activated, set to running and not interrupted in the subsequent schedule. In such a case a timing constraint violation can only occur if $wcet_{\tau(t)} > d_t$. As long as t does not finish its computation it must hold that $c_c(t) < wcet_{\tau(t)}$ as constrained in the *handlePossibleCases* method. By determining the reached time successor for each state where t is running, the constraint $c_c(t) \leq wcrt_{\tau(t)}$ is added in Line 5 of method *invIntersection*. As t is never interrupted in this case, it holds that $wcrt_{\tau(t)} = wcet_{\tau(t)}$. Thus, $wcet$ of t is never exceeded but is reached when t finishes its computation. Therefore, if $wcet_{\tau(t)} > d_t$ holds, it will be finally determined by the main algorithm.

Case 2: The second cause of a timing constraint may be that t is interrupted before it terminates. Let t get the status *running* after its instantiation. The time t is instantiated, its clocks $c_r(t)$ and $c_c(t)$ are reset and not reset again until t finishes its computation. An input of a higher priority task may interrupt t in a successor state, if t has not finished its computation up to this point, i.e. if $c_c(t) < wcet_{\tau(t)}$ holds (cf. method *handlePossibleCases*). Both clocks of the task $c_r(t)$ and $c_c(t)$ continue to progress, even if t is interrupted. Whenever a higher priority task terminates in a state s_x, its interrupt time for t is traced in the active task map $\mathcal{A}_{s_x}$. With this, the clock $c_c(t)$ has to progress to the value of the sum of its worst-case execution time and the sum of the interrupt times to be able to finish its computations (cf. *handlePossibleCases* method). As the clock $c_r(t)$ progresses equally with $c_c(t)$, timing constraint violations caused by interrupts will be determined by the algorithm in the main method.

Case 3: The third case of a timing constraint violation of a task may occur when the task t is inserted to the ready list. The only major difference to the second case is that the clock $c_c(t)$ is reset when the task gets running, while $c_r(t)$ starts to progress the time as soon as the task is instantiated. Thus, the task t finishes its computation, whenever the valuation for $c_c(t)$ is within the interval $[bcet, wcet]$ (if it is not interrupted after it gets the status running). As $c_r(t)$ traces the time from instantiation of t, all interrupt times are included in the valuation of this clock. Similar to the proof of the soundness, $c_r(t)$ valuates to the following: $c_r(t) := \Sigma_{j=1}^{m}([bcet_{\tau(hp(t)_j)}, wcet_{\tau(hp(t)_j)}]) \;\; - [lb_1, ub_1]$, where $hp(t)_2, ..., hp(t)_m$ do fully interrupt t, and t is instantiated during the execution of $hp(t)_1$. Again, possible timing constraint violations are detected by the main algorithm.

□

3.5.4. Termination of Algorithm

Next, it is shown that if the algorithm computing a resource STS always terminates if it is called for *finite* input STSs. For this, it is shown that only a finite number of states are generated, and that the recursion *handlePossibleCases* is called finitely.

Proof. The recursion is entered, when a task t_0 finishes its computation. Hereby, a transition $s \xrightarrow{end(t_0)} s'$ is created. If $\mathcal{A}_{s'}.size = 0$, the *if* part of Listing 3.5 is entered, the successor according to the input state is computed, and the algorithm terminates after instantiating such tasks, which are triggered and for which the activation condition is met according to the algorithm in Listing 3.4.

Else, there exists at least one task $t \in \mathcal{A}_{s'}$, which could also terminate its computation and trigger another task allocated to the resource. If the execution time of a task would be equal to zero, the algorithm would infinitely continue to terminate and start new task instances through the defined recursion. Thus, it has to be shown that time always increases when terminating tasks, as finally the discrete step defined by the edge of the input state has to be taken before the running task can terminate. This is directly given under the assumption that the execution times of tasks are always greater than zero. The response times of preempted tasks are always increased when the running task t_0 terminates: $\forall t \in \mathcal{A}_s, \tau(t) \neq \tau(t_0) : \mathcal{A}_{s'}(t) = \mathcal{A}_s(t) + [bcet_{\tau(t_0)}, wcet_{\tau(t_0)}]$ (see Line 3 in Listing 3.3). When such a preempted task gets active in the next recursion steps, this increased response time is used to compute the next discrete successor. With this the proof can be concluded: $\forall t \in \mathcal{A}_s \cap \mathcal{A}_{s'}, \tau(t) \neq \tau(t_0) : \mathcal{A}_s(t) < \mathcal{A}_{s'}(t)$. □

Next, it is proofed that the algorithm computes only a finite set of symbolic states, i.e. always terminates.

Proof. As there are only a finite set of discrete locations, the single possible source of infiniteness are the clock valuations. An infinite symbolic state set would be given, if clocks without a ceiling would be considered. Thus, we have to show that all clocks are bounded. When computing the time successor, always the invariant intersection function is called: Under the assumption that all clocks of the input STS have a ceiling, it holds that there is always a ceiling for all clocks tracing the periodical activation, i.e. $c_p(\tau) \leq p_\tau$. Also the clock $c_c(t)$ for the running task instance t is constrained in Line 5 of Listing 3.2 by the sum of the worst-case computation time and the upper bound of the corresponding interrupt time. For all other tasks t in the active task map and the ready list the corresponding clocks $c_c(t)$ are not constrained. Nevertheless, whenever the algorithm of Listing 3.5 is finished, the corresponding response time clocks are checked against the deadlines, i.e. whether $c_r(t) \leq d_t$, thus are also always bounded. If for a task t there is no response time clock, there is an instance counter variable $\mathcal{B}(t)$. If this counter exceeds the specified value, an instantiated task will not be able to finish its

computation in the allowed time frame. Note that this holds under the assumption, that the value for $\mathcal{B}(t)$ is chosen appropriately and not too restrictive. □

3.5.5. Minimization through Untimed Bisimulation, Timed Simulation Relation

To minimize computed STSs a new relation is introduced in this subsection, which is a combination of a bisimulation and a simulation relation. With this relation the number of states of a computed resource STS can be reduced without loosing any accuracy with respect to the timing behaviors of the tasks.

The idea is to merge paths which have on the one hand equivalent discrete behaviors, i.e. paths on which the same sequences of events are produced. In the context of scheduling this corresponds to merging of paths which have the same ordering of task activations and terminations. On the other hand paths which shall be merged must have a related timing behavior: If all clock zones of a path p_1 are a subset or equal to the clock zones of a path p_2, and p_2 has an equivalent discrete behavior as p_1, then p_1 is obsolete. All possible timing behaviors of p_1 are already covered by p_2. Thus, the behavior represented by path p_1 is a subset of the one of p_2. Note that this relation is defined in such a manner that paths with equivalent timings are also merged. This relation is called *untimed bisimulation, timed simulation* (UBTS) and is defined as follows.

Definition 18 (UBTS relation)**.** *Let two symbolic transition systems $STS_1 = (S_1, S_1^0, C, \Sigma, \rightarrow_1)$ and $STS_2 = (S_2, S_2^0, C, \Sigma, \rightarrow_2)$ over the same clock set C and alphabet Σ be given. A relation $r_{ubts} \subseteq S_1 \times S_2$ is an untimed bisimulation, timed simulation relation between STS_1 and STS_2, if for all $(\langle l_2, D_2\rangle, \langle l_1, D_1\rangle) \in r_{ubts}$ the following holds:*

1. $D_2 \subseteq D_1$.
2. *$\langle l_1, D_1\rangle$ is visited infinitely if and only if $\langle l_2, D_2\rangle$ is visited infinitely.*
3. *For all $s \in S_1$ with $\langle l_1, D_1\rangle \xrightarrow{\lambda}_1 s$ there exists a $t \in S_2$ such that $\langle l_2, D_2\rangle \xrightarrow{\lambda}_2 t$ and $(s, t) \in r_{ubts}$.*
4. *For all $t \in S_2$ with $\langle l_2, D_2\rangle \xrightarrow{\lambda}_2 t$ there exists a $s \in S_1$ such that $\langle l_1, D_1\rangle \xrightarrow{\lambda}_1 s$ and $(s, t) \in r_{ubts}$.*

For $(s_2, s_1) \in r_{ubts}$ we also write $s_2 \preccurlyeq s_1$.

Let $r_{ubts} \subseteq S \times S$ be an *untimed bisimulation, timed simulation* relation between the state set of a single transition system STS. This relation is a preorder (reflexive and transitive), i.e. if $(s_1, s_2), (s_2, s_3) \in r_{ubts}$ then $(s_1, s_3) \in r_{ubts}$, but not symmetric because of the first condition of Definition 18. When successive states are in such a relation (i.e. different paths of an STS) these can be merged. The reduced STS satisfies the same timing constraints as the original one. Unnecessary computations of redundant states

e.g. when computing the state spaces of dependent resources can be omitted. This reduces the overall analysis time. To prove this, we need first to define the term *maximal element* as follows.

Definition 19 (Maximal Element). *Let $r_{ubts} \subseteq S \times S$ be an untimed bisimulation, timed simulation relation between the state set of a transition system $STS = (S, S^0, C, \Sigma, \rightarrow)$. An element $s \in S$ is called maximal, if there is no $s' \in S$ with $s' \neq s$ such that $s \preccurlyeq s'$.*

The relation may further induce equivalence classes of states defined as follows.

Definition 20 (Equivalence Class). *Let $r_{ubts} \subseteq S \times S$ be an untimed bisimulation, timed simulation relation between the state set of a transition system $STS = (S, S^0, C, \Sigma, \rightarrow)$. Two states $s, s' \in S$ with $s \neq s'$ are equivalent, if $s \preccurlyeq s'$ and $s' \preccurlyeq s$. The set of all equivalence classes induced by r_{ubts} is denoted by χ_{ubts}. A representative of a class $c \in \chi_{ubts}$ is given by $[c]$.*

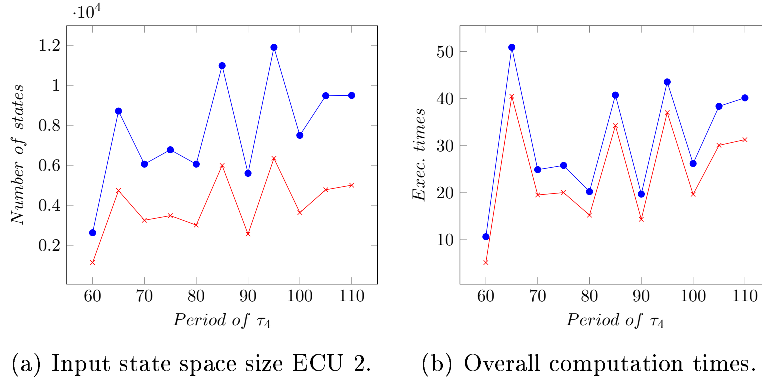
Lemma 7. *Let S be the state set of an $STS = (S, S^0, C, \Sigma, \rightarrow)$, $r_{ubts} \subseteq S \times S$ be an untimed bisimulation, timed simulation relation, and φ a timing constraint. If $s \in S$ is maximal with respect to r_{ubts}, and $s \models \varphi$, then for all $s' \in S$ with $s' \preccurlyeq s$ it holds $s' \models \varphi$.*

Lemma 8. *Let S be the state set of an $STS = (S, S^0, C, \Sigma, \rightarrow)$, $r_{ubts} \subseteq S \times S$ be an untimed bisimulation, timed simulation relation, and φ a timing constraint. For all $c \in \chi_{ubts}$ it holds that if $[c] \models \varphi$ then for all $s \in c$ it holds that $s \models \varphi$.*

Both lemmas do hold, as the relation is transitive. With both lemmas it is allowed to correctly reduce the state space of the original STS, where only the maximal states and the representatives of every equivalence class are left. The complexity of determining the simulation equivalent states is $n(n-1)$, where n is the number of states, as each state has to be compared with all other states to determine the related states.

Note that this minimization does not necessarily lead to smaller state spaces of dependent resources, as no information is abstracted. Nevertheless, the computation time of the STSs of dependent resources is reduced, as less input states have to be considered, for which resource states have to be computed. Thus, with this technique the multiple computations of the same resource states can be omitted.

The effect of this minimization technique is illustrated by applying it to the architecture which is illustrated in Figure 3.4. The minimization is applied to the input STS of $ECU2$. Multiple scenarios are considered by varying the period p_{τ_4} of task τ_4. The result is illustrated in the left part of Figure 3.13. For instance, in the scenario where the period of τ_4 is 85 time units, the size is reduced from 10982 to 5991 states, which is a significant saving. The resulting overall computation times for each scenario is illustrated in the right part of the figure. The overall analysis time is positively affected, as less input states have to be considered for the dependent resource: While approximately 41 seconds were needed in the original case to analyze the architecture with $p_{\tau_4} = 85$, this time could be decreased to approximately 34 seconds.



(a) Input state space size ECU 2. (b) Overall computation times.

Figure 3.13.: Left: Resulting state spaces of architecture in Figure 3.4; Right: Overall computation times – Blue curves: Without UBTS minimization; Red curves: With UBTS minimization.

Over-approximation	$\forall$-properties	$M' \models \varphi \;\Rightarrow\; M \models \varphi$
$M \preccurlyeq M'$	$\exists$-properties	$M' \not\models \varphi \;\Rightarrow\; M \not\models \varphi$
Under-approximation	$\exists$-properties	$M' \models \varphi \;\Rightarrow\; M \models \varphi$
$M' \preccurlyeq M$	$\forall$-properties	$M' \not\models \varphi \;\Rightarrow\; M \not\models \varphi$

Table 3.1.: Preservation of universal and existential properties considering approximations.

These computation times have been determined by executing the scenario at least five times and taking the average computation time. All tests were performed on the same machine (standard laptop with 2GHz and 6GB main memory).

3.6. Abstraction Techniques

In the context of scheduling analysis specific abstraction functions are defined in the following. There are two approaches for abstractions in model checking, i.e. *over-* and *under*-approximations. Basically, for over-approximation extra behavior is added which is not part of the original model. In contrast to this, for under-approximations actual behavior is removed to focus on a part of the overall behavior. Both approaches target to reduce the size of a given state space while preserving different properties.

In Table 3.1 the preservation relation when using over- and under-approximations is illustrated. Thereby, it is distinguished between *universal* and *existential* properties. Universal properties are such properties which shall hold for every reachable system

state, whereas to fulfill an existential property it is sufficient to find a reachable system state, where this property holds. Over-approximations preserve universal properties, as the original behavior is at least contained in the over-approximation. Thus, if the over-approximated model fulfills such a property, one can conclude that the original model also fulfills this property. This is not true for existential properties, as the abstract state fulfilling such a property must not be part of the original model. Analogously, under-approximations preserve existential properties, but not universal ones.

In our case, task constraints and end-to-end latency constraints correspond to universal properties. Thus, an abstraction technique which leads to an over-approximated behavior has to be defined. In contrast to the previous minimization techniques, such an abstraction may of course also lead to spurious counter-examples, which do not occur in the real model. Nevertheless, the scalability could be improved by such techniques.

Next, the effects of clock resets are analyzed. Based on the results of the Subsection 3.6.1 the effects of abstracting specific clocks of resource STSs are determined and an abstraction function yielding over-approximated STSs is derived. In Subsection 3.6.3 the timed simulation relation between the states of STSs is defined. On the basis of this relation an abstraction function which also results in over-approximated STSs is worked out. In 3.6.5 it is discussed how under-approximated STSs can help for our analysis task approach. The last Subsection 3.6.6 will introduce an approximation for scenarios, where event bursts may occur.

3.6.1. Clock Resets and Duration Clocks

In this section the effect of clock resets is analyzed. Clock resets lead to points of discontinuities in the evolution of the clock valuations. Assume the following sequence of clock valuations for a clock c is given: $(c = 0), (c \in [2, 3]), (c \in [4, 4])$. When an intermediate clock reset is added resulting in the sequence $(c = 0), (c \in [2, 3]), (c := 0), (c \in [1, 2])$ the following problem arises: We do not know how to combine the intervals $[2, 3]$ and $[1, 2]$ *correctly* to obtain the original behavior. As there are no further constraints we can combine these arbitrarily, e. g. we are able to generate the concrete time sequence $(c = 0), (c = 2), (c := 0), (c = 1)$ which was not possible in the original symbolic sequence. Thus, by adding clock resets over-approximated behavior will be obtained.

This observation is helpful to see that two clocks cannot be replaced by a single one without loosing accuracy. The following trace set characterizes the behavior of a task τ with an execution time in the range of $[5, 10]$ time units, triggered by an event stream with a period of 20 time units. Note that the symbol τ signalizes the start of the task and $\overline{\tau}$ the end of computation of the task.

$$(\tau, \{c_p \in [0, 20], c_c = 0\}),\ (\overline{\tau}, \{c_p \in [5, 10], c_c \in [5, 10]\}),\ [(\tau, \{c_p \in [20, 20], c_c = 0\}),$$
$$(\overline{\tau}, \{c_p \in [5, 10], c_c \in [5, 10]\})]^{\omega}.$$

Intuitively, for tasks which depend on τ do only need the timing information about the termination times $\overline{\tau}$ rather than the start times of the task. Thus, one could try to determine the termination times by adding a *duration* clock c_d, which traces the time from the triggering of τ up to the end of its computation. Every time τ terminates, this clock is then reset. Thus, the following extended trace set would be obtained:

$$(\tau, \{c_p \in [0, 20], c_c = 0, c_d = [0, 20]\}),\ (\overline{\tau}, \{c_p \in [5, 10], c_c \in [5, 10], c_d = [5, 30]\}),$$
$$[(\tau, \{c_p \in [20, 20], c_c = 0, c_d = [10, 15]\}),\ (\overline{\tau}, \{c_p \in [5, 10], c_c \in [5, 10], c_d = [15, 25]\})]^\omega.$$

To get rid of unnecessary information such as the starting time of τ and thus to save some states, the next step is to abstract from the clocks c_p and c_c such that we get the following abstracted trace set:

$$(\overline{\tau}, \{c_d = [5, 30]\}),\ [(\delta, \{c_d = [10, 15]\}),\ (\overline{\tau}, \{c_d = [15, 25]\})]^\omega.$$

Analogously to the case of inserting resets, we do not know how to "combine" the concrete clock valuations of this abstract trace. Besides the original concrete traces, we get further concrete traces like for example $\sigma = (\overline{\tau}, 5), (\overline{\tau}, 15), (\overline{\tau}, 15)...$ This trace does not occur in the original case. The major problem of this approach is that the original activation period decreases, in the example from 20 to 15 time units, which means that the activation load gets much larger than in the original case. Thus, this abstraction leads to too coarse models for our analysis approach and will not find further application in the following.

3.6.2. Clocks of Interface STSs

In Subsection 3.4.1 a set of clocks was derived which have to be preserved for the computation of interfaces between dependent resources. Here, the effects which occur when such clocks are abstracted will be illustrated.

To illustrate the effect when period clocks of higher priority tasks are abstracted in the context of the interface computation, consider again the architecture of Figure 3.4 with adapted task properties as follows: $p_{\tau_1} = 20$, $p_{\tau_2} = 10$, $p_{\tau_4} = 20$, $c_{\tau_1} = 11$, $c_{\tau_2} = 1$, and $p_{\tau_4} = 15$.

Assume the state space of `ECU1` has already been computed and as a next step the interface to resource `ECU2` has to be determined.

A finite sequence of states of the resource STS of `ECU1` is illustrated in the top part of Figure 3.14. In the figure the focus is only on the clock zones without considering the discrete locations, which are not relevant for this scenario. The first row and column of the matrices represent the reference clock, the second the period clock $c_p(\tau_1)$ of τ_1, and the third the period clock $c_p(\tau_2)$ of τ_2. Note that all other clocks are left out for the purpose of readability. The clocks within the curved brackets below the matrices indicate the reset of these clocks after reaching the corresponding state.

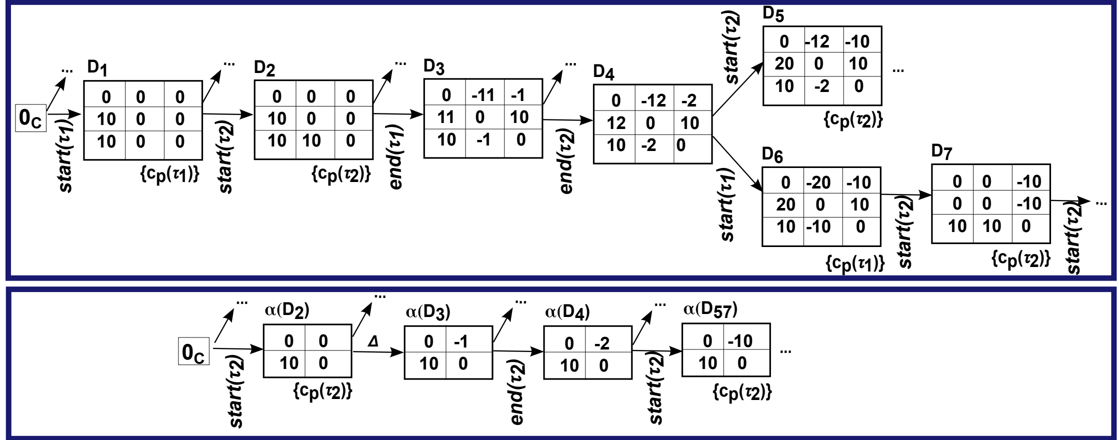


Figure 3.14.: Example trace - Top: an output path of a resource; Bottom: computed interface.

The finite sequence illustrated in the top part of Figure 3.14 starts by triggering an event *start*(τ_1) which starts an instance of τ_1 within the time interval of $[0, 10]$. Note that in this scenario τ_1 cannot start after the time value of 10 although its period is 20. This is because of the ordering of the events in this scenario: If an instance of τ_1 starts *before* an instance of τ_2, this must happen before the clocks reach the period value of τ_2, which is 10 in this example. Note also that of course there is a path in which τ_2 can also arrive before τ_1, but this is not illustrated in the figure. In such a case τ_1 could arrive in a time interval of $[0, 20]$.

After the start of τ_1 in state D_1 an instance of τ_2 is started also within the time interval of $[0, 10]$ in state D_2. In D_2 the relation between both clocks is determined in the zone by the constraints $c_p(\tau_2) - c_p(\tau_1) \leq 10$ and $c_p(\tau_1) - c_p(\tau_2) \leq 0$, which means that the distance of the second event to the first one amounts to $[0, 10]$. Thus, the distance of both events may be 10 time units at a max, or may occur simultaneously.

In state D_3 task τ_1 ends its computation after enough resource time has been allocated to it, i.e. 11 time units after it started. This duration corresponds to the tasks execution time as it was not interrupted by any other task. Reaching the state D_4 by ending the computation of τ_2 one can see that this happens between 2 and 10 time units after the instance started its computation. The DBM of the state D_4 gives the relation between both clocks which is $c_p(\tau_1) \geq c_p(\tau_2) + 2 \wedge c_p(\tau_1) \leq c_p(\tau_2) + 10$, i.e. the value of $c_p(\tau_1)$ is at least 2 time units larger (first part of the condition) and up to 10 time units larger than the value of $c_p(\tau_2)$ (second part of the condition).

Let us assume that in state D_1 task τ_1 arrives at $t = 0$. In the upper left part of Figure 3.15 the scheduling for the situation up to the point, where state D_4 is reached,

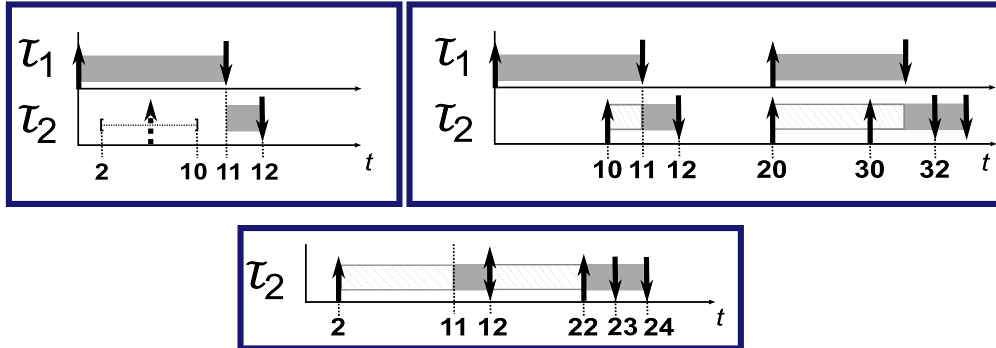


Figure 3.15.: Event sequence.

is illustrated. If τ_1 starts by entering the successor D_6, the relation between both clocks is refined to $c_p(\tau_2) = c_p(\tau_1) - 10$, i. e. the value of $c_p(\tau_2)$ is exactly 10 time units smaller than $c_p(\tau_1)$. With this, the valuation of the clock $c_p(\tau_2)$ in state D_4 must have been exactly 2 time units to reach state D_6. This situation is illustrated in the upper right part of Figure 3.15.

If the period clock of task τ_1 is abstracted as the specific timing behavior of this task is not relevant for the dependent resource `ECU2`, some over-approximations would be obtained, which is illustrated in the following. In the lower part of Figure 3.14 the abstraction of the trace, which is illustrated in the upper part of the figure, is depicted. In the DBMs only the reference clock and $c_p(\tau_2)$ are left. The problem of this sequence is that there is no relation kept between the intervals of states D_3 and D_4 and can be combined arbitrarily. An example of a combination is illustrated in the lower part of Figure 3.15 which would not occur in real: After the start of τ_2 at time 2 it finishes its computation at time 12, at which time an instance of a new instance is also activated. The next activation occurs at time 22. The instances end at times 23 and 24. What happens here is that both traces of Figure 3.14 are put together and mixed, such that besides the traces which occur in real further traces are produced. This results in an over-approximated behavior. This extra behavior leads to activations of dependent tasks which occur in too dense time points, and thus increase the load of the successive resource. An increase of the load of course means that the end-to-end latencies increase such that more pessimistic results are obtained.

If the successive dependent resource has enough spare time, such an increasing of response times must not occur. For this, consider Figure 3.16, in which the abstracted trace of Figure 3.15 is extended by the behavior of the tasks on the dependent resource $ECU2$. Assume that τ_4 has an execution time of 15 time units, and τ_3 an execution time of 2 time units. In the worst-case, τ_4 decides to start its computation when τ_2 terminates

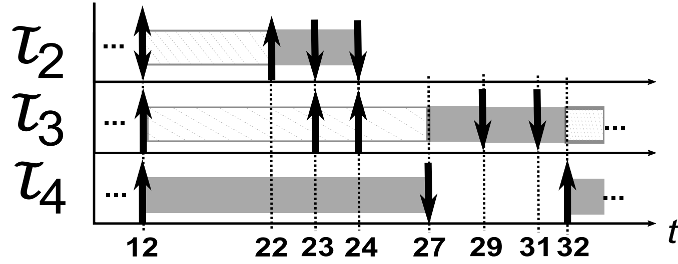


Figure 3.16.: Effect of abstraction.

and thus triggers an instance of task τ_3, such that τ_3 cannot start its computation. Considering the timing behavior of τ_2, further instances of τ_3 are triggered at times 23 and 24. Note that in the not-abstracted case these activations would appear later, i.e. at times 32 and 33. When τ_4 finishes its computation at time 27, 5 time units are left to complete three instances of τ_3, before an instance of τ_4 is instantiated again. Two instances can be completed, but the third is interrupted by an activation of τ_4. If the execution time of τ_4 would have been 14, the third instance of τ_3 could have been finished with a response time, which is within the real best- and worst-case response times.

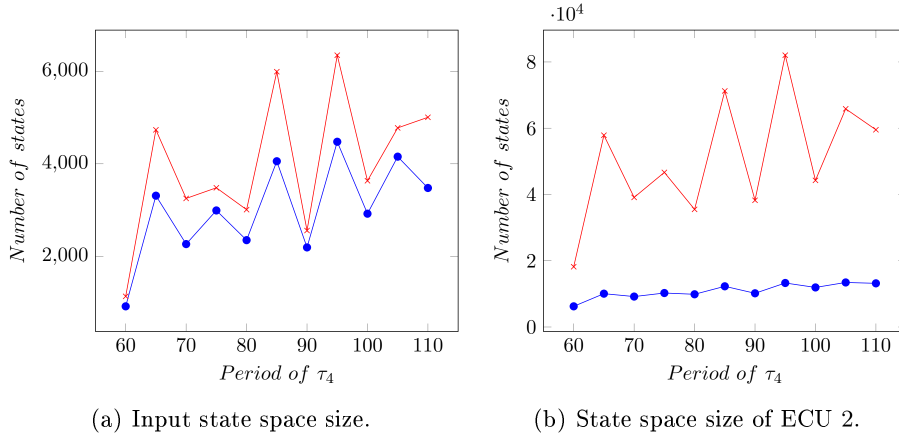
The effect of the clock abstraction is illustrated by applying it to the architecture which is illustrated in Figure 3.4. In this example, the clocks of the task τ_1 are abstracted for the input STS of $ECU2$. As before, multiple scenarios are considered by varying the period p_{τ_4} of task τ_4. The result is illustrated in Figure 3.17. In the left part the resulting input state space size of $ECU2$ is illustrated. Although no significant reduction of the state space is obtained here, the resulting resource state sizes of $ECU2$ are much smaller. This is illustrated in the right part of the figure. For the scenario where the period of τ_4 is 85 time units the size of the state space is reduced from 71246 states to 12281 states, and from 38246 states to 10172 states.

3.6.3. Abstraction through Simulation Relation

Simulation relations have been widely used in literature to provide approximations to language inclusion in polynomial time. Here, an appropriate simulation relation on STSs is defined to derive over-approximated STSs. For the definition of a timed simulation relation first the refinement of zones has to be defined.

Definition 21 (Zone Refinement)**.** *Let D, D' be clock zones over a clock set C. D' refines D, if for every $\nu \in D'$ it holds that $\nu \in D$. This corresponds to the classical set inclusion, i.e. $D' \subseteq D$.*

Note that a zone D over a clock set C satisfies a clock constraint g, iff $D \subseteq D_g$. The refinement operation therefore preserves the satisfaction of clock constraints.



(a) Input state space size. (b) State space size of ECU 2.

Figure 3.17.: Resulting state spaces of architecture in Figure 3.4 – Blue curves (marked with dots): Number of states with abstraction of hp clocks; Red curves (marked with x): Without abstraction.

Definition 22 (Timed Simulation Relation). *The timed simulation relation* $r_{ts} \subseteq S_2 \times S_1$ *is defined between two symbolic transition systems* $STS_1 = (S_1, S_1^0, C, \Sigma, \rightarrow_1)$ *and* $STS_2 = (S_2, S_2^0, C, \Sigma, \rightarrow_2)$ *over the same clock set* C *and alphabet* Σ*. For all* $(\langle l_2, D_2 \rangle, \langle l_1, D_1 \rangle) \in r_{ts}$ *the following holds:*

1. $D_2 \subseteq D_1$,
2. $\langle l_1, D_1 \rangle$ *is visited infinitely if and only if* $\langle l_2, D_2 \rangle$ *is visited infinitely,*
3. *For all* $t \in S_2$ *with* $\langle l_2, D_2 \rangle \xrightarrow{\lambda}_2 t$ *there exists a* $s \in S_1$ *such that* $\langle l_1, D_1 \rangle \xrightarrow{\lambda'}_1 s$ *and* $(t, s) \in r_{ts}$*, and* $\lambda = \lambda'$.

For $(s_2, s_1) \in r_{ts}$ *we also write* $s_2 \preccurlyeq s_1$.

The proof that this abstraction leads to an over-approximated STS is according to the simulation relation of Definition 16.

Analogous to the *untimed bisimulation, timed simulation* relation of Subsection 3.5.5 the timed simulation relation defines a preorder. Let $r_{ts} \subseteq S \times S$ be a timed simulation relation between the state space of a single transition system STS. If $(s_1, s_2), (s_2, s_3) \in r_{ts}$ then $(s_1, s_3) \in r_{ts}$.

The maximal elements of this relation are defined analogously:

Definition 23 (Maximal Element). *Let* $r_{ts} \subseteq S \times S$ *be a timed simulation relation between the state spaces of a single transition system* STS*. An element* $s \in S$ *is called maximal, if there is no* $s' \in S$ *with* $s' \neq s$ *such that* $s \preccurlyeq s'$.

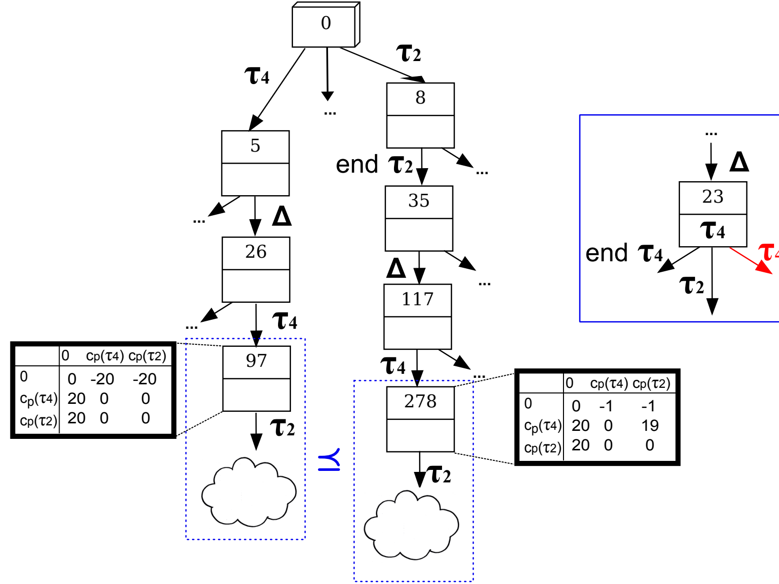


Figure 3.18.: Two paths of an input STS of a resource, where $97 \preccurlyeq 278$; Blue box (on top right): State of resource STS created if input states 97 and 278 were merged.

Lemma 9. *Let S be the states of an STS and φ a timing constraint. If $s \in S$ is maximal and $s \models \varphi$, then for all $s' \in S$ with $s' \preccurlyeq s$ it holds $s' \models \varphi$.*

The lemma holds, as the relation is transitive. The resulting abstraction is thus an over-approximation of the original STS.

In Figure 3.18 the application of this abstraction is illustrated for an input STS of a resource. The states 97 and 278 are determined to be in a simulation relation: The range of $c_p(\tau_4)$ in state 278 is $[1, 20]$, whereas in state 97 it has the exact value of 20. Thus, the abstraction reduces the state space by deleting state 97 and all its successors, and by changing the edge $26 \xrightarrow{\tau_4} 97$ to $26 \xrightarrow{\tau_4} 278$.

Besides the original trace $(\tau_4, t_1)(\Delta, t_2)(\tau_4, t_3)$... with $t_3 \in [20, 20]$ we get further traces as the range of time frame t_3 increases to $[1, 20]$ in state 278. Thus, the second instance of τ_4 may arrive at times $[1, 20)$ which were not possible in the original trace, increasing the load of the resource. This is indicated in the figure by the resource state 23, where the output transition activating an instance of τ_4 is possible while an instance of τ_4 is already active.

The complexity of determining the simulation equivalent states is $n(n-1)$, where n is the number of states. Each state has to be compared with all other states. Some comparisons are performed twice: For all pairs of states s_1, s_2, we check both $s_1 \preccurlyeq s_2$

and $s_2 \preccurlyeq s_1$. For many cases, the complexity is reduced to $n(n-1)/2$ in the best-case. For this, the already analyzed pairs are added to a hash table. Corresponding state pairs are added to this table, if the check was already successful for the first comparison, or if the zones are completely disjoint or not in a subset relation, such that comparing the other direction would not make any sense.

3.6.4. Effects of Over-Approximations for Iterative Analysis Approach

So far, techniques to generate over-approximations of the original state spaces were presented. Generally, over-approximations are an effective approach to minimize a given state space, such that the complexity to traverse through this state space and perform some analysis to evaluate given properties – i.e. timing deadlines in our case – on the reached states is minimized. Thereby, the over-approximation is necessary to give safe results.

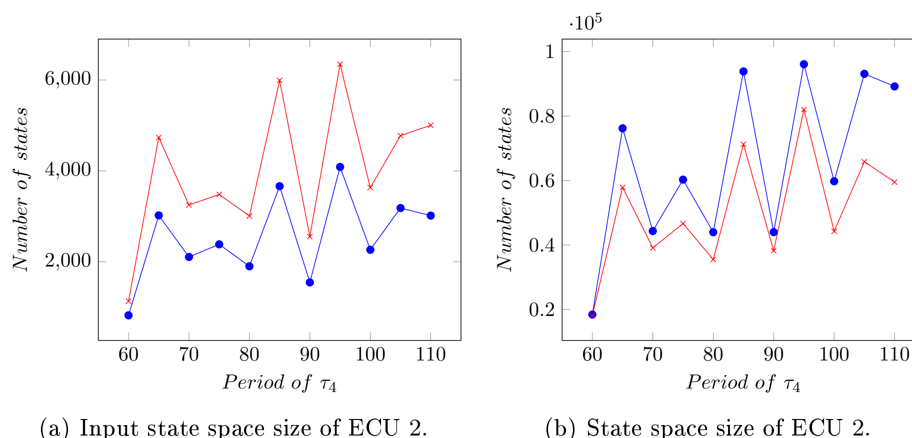
The abstraction techniques illustrated in the previous subsections were implemented and applied to our iterative analysis approach: For each computed interface STS an over-approximation is computed, which then is used as an input for dependent resources. Interestingly, this approach leads to larger state spaces for dependent resources. The reason for this is that when performing over-approximations, more behavior is allowed to occur which also increases the load of the resources as illustrated before. As more behavior is allowed in the input STS, more activation situations have to be handled within the dependent resource state spaces, leading to increased state spaces and also longer analyses times.

The effect of the timed simulation relation abstraction is illustrated by applying it to the architecture which is illustrated in Figure 3.4. In specific, the timed simulation relation is applied to the input state space of $ECU2$. The result is illustrated in Figure 3.19. Although the input state spaces for every scenario are reduced as illustrated in the left part of the figure, the resulting numbers of the states of the resource STSs are always above the ones without applying the abstraction on the input STSs. For instance, in the scenario where the period of τ_4 is 85 time units, the state space increases from 71246 states to 93869 states.

To result in smaller state spaces for dependent resources, under-approximation techniques would be more suitable. Unfortunately, such techniques lead to too optimistic and unsafe verification results. In the next subsection, the benefits of such approaches in our context is discussed.

3.6.5. Testing: Abstraction through Under-Approximation

Using the concept of depth-first search through the state space of a resource a path which produces a schedule with events at fixed time intervals which repeat infinitively often in the same order can be determined. By reducing the state space of the resource to only



(a) Input state space size of ECU 2.

(b) State space size of ECU 2.

Figure 3.19.: Resulting state spaces of architecture in Figure 3.4 – Blue curves (marked with dots): Number of states with timed simulation relation abstraction; Red curves (marked with x): Without abstraction.

this single state sequence, an under-approximation of the original state space is obtained. This single state sequence is then used to compute the state space of dependent resources.

There are various alternatives to select an appropriate path. The easiest and fastest approach is to select an arbitrary trace. A more appropriate solutions might be to select the path which results in the most dense activations with respect to a dependent task on the target resource. In such a trace the load of the dependent resource is higher, which could cause larger response times. Another intuitive solution is to select the path, where the activation times of all tasks on the dependent resource are as close as possible. This solution is related to the critical instance in classical schedulability analysis, where it is assumed that the activations of all tasks might occur simultaneously. But this solution must not necessarily result in the worst-case latency times, as a path could be selected, where some tasks terminate within their best-case execution- or response times.

The advantage of the under-approximation approach is that one can get results really fast. Unfortunately, this concept is closely related to testing and thus a path could be chosen which does not include a timing constraint violation, although there might exist paths which do violate some constraints. This fact holds independently from the selection approach of a path mentioned above.

To solve this problem, this approach can be extended in such a manner that all paths of a resource are tried iteratively instead of only a single path. That is, if the first path does not find any timing constraint violations, the second path is chosen, and so on. If all paths are chosen at the end, all possible behaviors are covered and the answer of the verification is sound in the sense that the system does not violate its timing constraints.

Of course, for systems which do not violate their timing constraints this approach does not yield faster analysis results as all possible behaviors are considered iteratively. But for systems violating timing constraints this approach can help in finding counter examples in a much faster way.

Considering the implementation of the approach, the selection of an arbitrary path is realized and applied in the case study of Subsection 3.7.2.

3.6.6. Abstraction for Event Bursts

A new abstraction technique is introduced, which enables the minimization of the output STS of a resource when event bursts occur. Consider the architecture in Figure 3.4 where two tasks τ_1, τ_2 are allocated on an $ECU1$. Task τ_1 has higher priority, and has a computation time of 35 time units. The period of τ_2 is $p_{\tau_2} = 5$ time units. When τ_1 starts its computation, it is possible that eight instances of τ_2 can be activated before the active instance of τ_1 finishes its computation. This scenario is illustrated in the left part of Figure 3.20. In this state sequence, an instance of τ_1 gets instantiated in the first state D_i, such that the corresponding computation time clock c_c of this running instance evolves over the sequence. The first instance of τ_2 is activated in the same time instance as τ_1 in state D_{i+1}. All other instances of τ_2 are activated according to the period of the task, i.e. after each 5 time units. Whenever an instance of τ_2 is activated, its period clock has to be reset, which is indicated through the clocks in the curly brackets on the left hand of the corresponding states. At the end of the sequence, the instance of τ_1 terminates as its clock reaches the needed computation time.

Considering resources on which tasks are allocated that depend on τ_2 nothing relevant happens in this state sequence, as dependent resources do only need the termination times of the instances of τ_2. The only relevant information of this sequence is the number of instances of τ_2 which occur until the active instance of τ_1 finishes its computation. Unfortunately, this state sequence cannot simply be merged to a single state: If done so, the successor of state D_i, where $c_c(\tau_1) = 0 \wedge c_p(\tau_2) = 0$ holds, would be the state D_{i+8}, where $c_c(\tau_1) = 35 \wedge c_p(\tau_2) = 5$ holds. In the semantics of timed automata this is a problem as all clocks have to be progress in the same rate. Thus, such a merging of states would produce a point of discontinuity.

The reason for this point of discontinuity is that the period clock $c_p(\tau_2)$ is always reset on every occurrence of τ_2. As this clock is kept for the output interface, these states are not merged. To get rid of such state sequences, a new clock is introduced for all tasks where such event bursts can occur. These clocks are called *abstract period clock* and denoted as c_{ap}. These clocks do not affect the state space, but trace of the time that passes from the first activation of a burst. The idea of this clock is illustrated in the center of Figure 3.20. When the first instance of task τ_2 of the example gets activated in state D_{i+1}, the clock $c_{ap}(\tau_2)$ is reset. Then, instead of resetting it on every occurrence of a further activation, the clock is let progressed until an instance of τ_2 gets the status

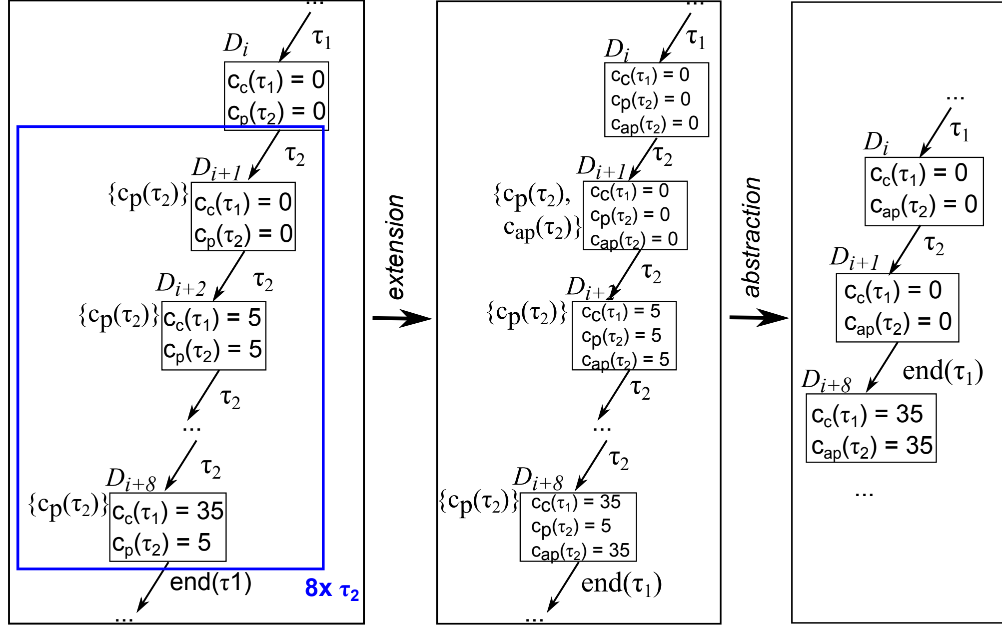


Figure 3.20.: Merging states by the application of the *abstract period clock*.

running, which is the successor of state D_{i+8} in the example.

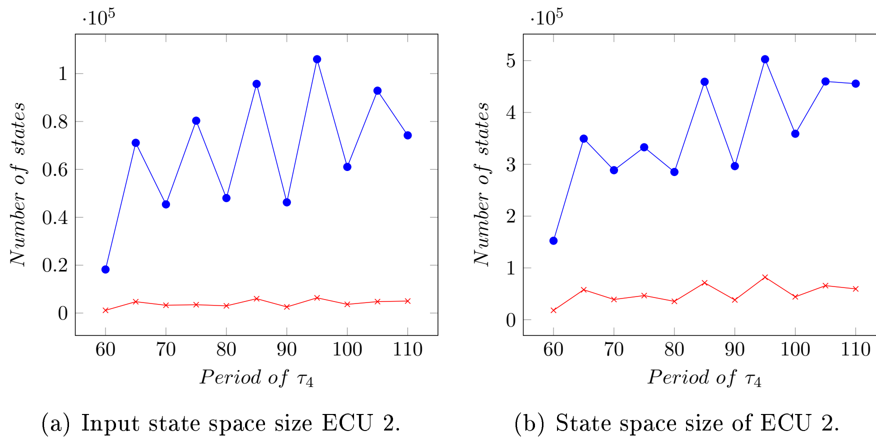
When computing the output of the resource, the period clock of each task is abstracted, which has been extended by an abstract period clock. Hence, paths with activation bursts do collapse in a single state, as it is illustrated in the right part of Figure 3.20. As there is no reset of a relevant clock in the output, and all clocks progress with the same rate, all intermediate states can be merged. Thus, no point of discontinuity is produced by this approach. The exact number of the instances of τ_2 are preserved, as these are contained in the ready task list of the last state of this sequence.

Let T_r be the task set of a resource r. The clock set C of a resource is extended with a set of *abstracted period clocks* (in short c_{ap}), such that the following holds:

$$\forall \tau \in \mathsf{T}_r, hp(\tau) \neq \emptyset \wedge instances(\tau) > 1,\ \exists c_{ap}(\tau) : c_{ap}(\tau) \in C \tag{3.19}$$

Note that T is the task set allocated to the considered resource.

This abstraction is restricted to such cases, where only local timing constraints of task are specified. If end-to-end latency constraints are specified, we need to keep the information of the activation of each instance. Thus, for dependent resources the states as described above cannot be merged, as the exact activation behavior would not be available then.



(a) Input state space size ECU 2.

(b) State space size of ECU 2.

Figure 3.21.: Resulting state spaces of architecture in Figure 3.4 – Blue curves (marked with dots): Number of states without abstraction; Red curves (marked with x): Number of states with burst abstraction.

The effect of the burst abstraction is illustrated by applying it to the architecture illustrated in Figure 3.4. The result is illustrated in Figure 3.21. The significant reduction of the number of states is obtained for both the input of the resource $ECU2$ and the resource state space itself. For instance, in the scenario where the period of τ_4 is 85 time units, the input state space is reduced from 95698 states to 5991 states, and the resource state space from 459210 states to 71246 states.

3.7. Case Study: Driver Assistance System

3.7.1. Overview

The introduced concepts are evaluated on an industrial case study of a lane-keeping-support system (LKS) in the following. This case study was originally introduced in [WTH+14]. The LKS is a part of a driver assistance system which is also detailed here. A previous version of the presented evaluation was already published in [GWB15].

The functional structure of the driver assistance system (DAS) mainly consists of an extended lane-keeping-support (LKS) system described accompanied by an adaptive cruise control system. The LKS is a driver assistant system that helps the driver to keep his car on the lane during a trip. A scenario in which the LKS is of particular importance is a tired or distracted driver loosening the grip of the steering wheel which leads to the car bearing away from the current lane. As a car unintentionally leaving its lane may lead to a hazardous situation, the LKS aims to prevent this from happening. So in order

to keep the car on the lane the system evaluates whether the driver leaves the current lane unintentionally by using cameras recognizing road markings, and monitoring driver actions. The LKS system decides to intervene by steering the car back to the lane if necessary via a correction of the steering angle or via a short braking intervention on either side of the car. These two intervention options do not mutually exclude each other, they are more like variants depending on the actual variant configuration of the car that the LKS is integrated in.

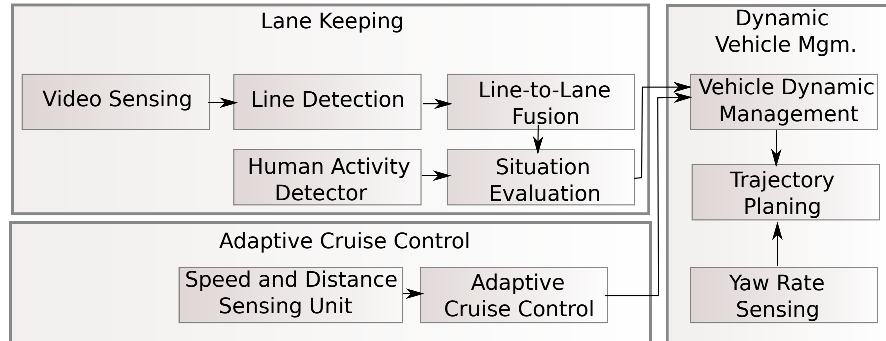


Figure 3.22.: Functional structure of the driver assistance system.

The overall functional structure of the DAS is illustrated in Figure 3.22 and its corresponding component structure in Figure 3.23. It consists of three major blocks, i.e. the lane keeping, the adaptive cruise control, and the dynamic vehicle management functionality.

The functional chain for the lane keeping starts at the video sensing unit (f_{vsu}), which receives an uncompressed two mega-pixel video stream from a camera with a period of $80ms$ and pre-processes this data. This data is then passed to the line detection unit (f_{ldu}). The line detection unit uses the video stream to extract line information which may indicate lanes of a street. From there, rather small data amounts are passed down the functional chain. The line-to-lane-fusion function (f_{l2l}) then extracts the line data from the line detection unit and extracts information about the own and the neighboring lanes relevant for the car. The situation evaluation (f_{seu}) assesses the current situation based on the passed lane data, the current steering angle together with other relevant human machine interface data received from the human activity detector function (f_{had}), and outputs the (un)intended trajectory as indicated by the sensor data to the vehicle dynamic management function. Additionally, the situation evaluation forwards data about whether the trajectory is intended or unintended based on certain thresholds in the sensor data analysis.

An adaptive cruise control (ACC) consists of two major functions, i.e. the speed and distance sensing unit receiving data from speed and radar sensors, and an adaptive cruise control functionality which keeps the currently set speed, which is determined by

the driver. The ACC slows the vehicle down in case it is gaining on another vehicle ahead and has thereby to ensure a fixed minimum distance to the vehicle ahead. The vehicle dynamic management receives the desired and actual speed from the ACC.

The trajectory planning function is the controlling part that compares the trajectory from the situation evaluation with the current trajectory of the car, which is analyzed by using a yaw-rate sensor. Based on the offset between these two trajectories and the information about the intention of the driver it decides when and how to intervene. After the trajectory planning the corresponding braking and steering actuators are triggered.

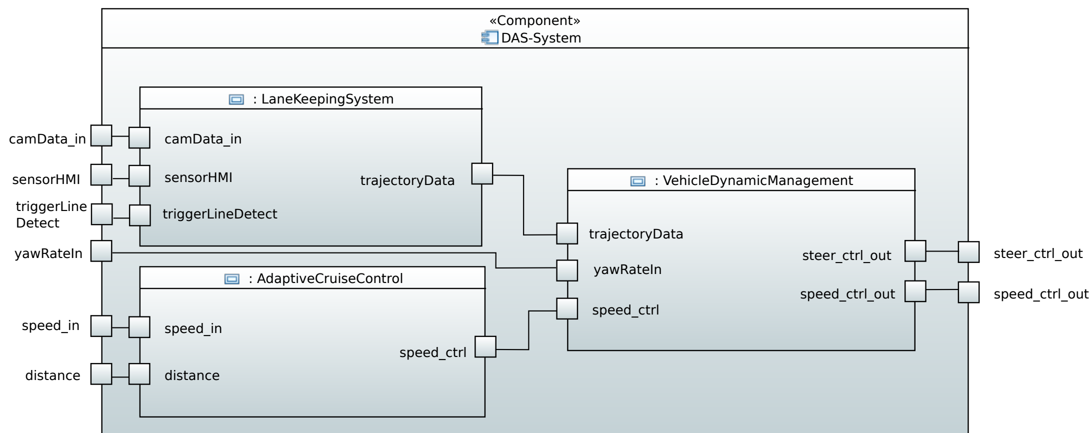


Figure 3.23.: High level component structure of the overall driver assistance system.

The driver assistance system is decomposed according to the three functional clusters, i.e. into the three subsystems *lane keeping system* (LKS), *adaptive cruise control* (ACC) and *vehicle dynamic management system* (VDS). The LKS has two input ports, namely camera data *camData_in* which is necessary to determine the lane markings, and *sensorHMI* which is the human interface data including the current steering wheel angle. This angle is indicated by the position of the steering wheel and determines the position of the car with respect to the lane and the angles of the wheels itself. From this input it computes an angle for the wheels which shall be set and outputs this on the port *trajectoryData*.

The ACC uses its inputs *speed_in* and *distance* – indicating the current speed of the car and the speed which is should be maintained, and the distance to a front car respectively – to compute the new speed of the car. The result of the computation is delivered at the output port called *speed_ctrl*.

The VDS combines and adjusts the control values of the ACC and the LKS and sets the corresponding actuator values *steer_ctr_out* and *speed_ctrl_out*.

In the following, the LKS is detailed and the analysis approach is evaluated on this subsystem.

3.7.2. Lane-Keeping-Support System

The decomposition of the lane keeping system is illustrated in Figure 3.24. It consists of two computing resources *LaneDetection* and *SituationEvaluation* interconnected by a communication resource *CANBUS*. To all resources a scheduler is allocated. For all resources, the fixed-priority policy is considered (preemptive for the computation resources and non-preemptive for the communication resource).

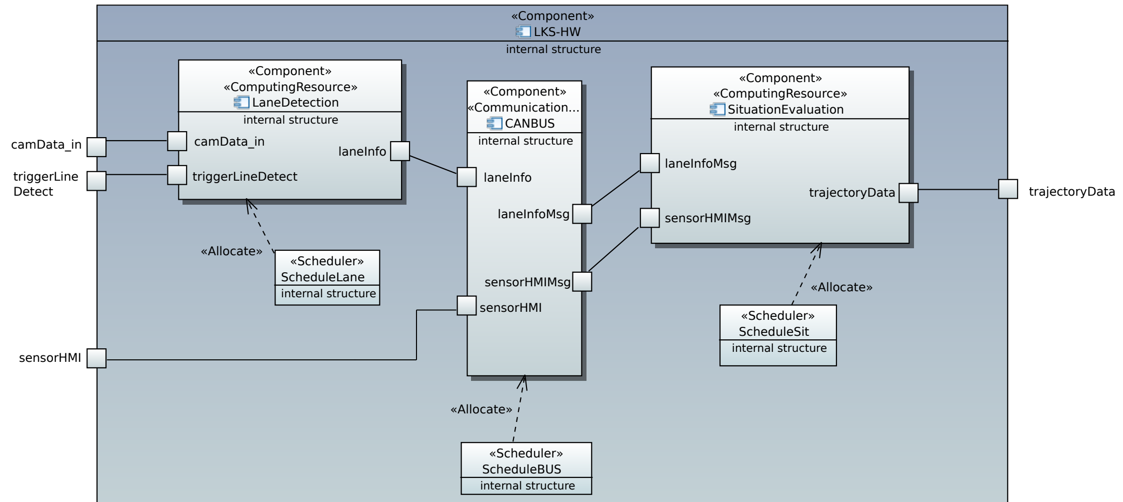


Figure 3.24.: Decomposition of the lane keeping assistance system.

The functional view of the MARTE model of the LKS containing the allocated tasks of the LKS is illustrated in Figure 3.25. As the resource names suggest, the *VideoSensing*, *LineDetection*, and *LineToLaneFusion* tasks are allocated to the *LaneDetection* resource, the two communication tasks *ComTask1* and *ComTask2* to the *CANBUS*, and the *HumanActivityDetector* and the *SituationEvaluation* to the *SituationEvaluation* resource.

In addition to the functional specification of Figure 3.22 some further technical details are added: The communication between the *LineDetection* function and the *LineToLaneFusion* function is realized through a shared buffer, as the activation periods are different. The *LineDetection* writes its processed data into this shared memory, from which the *LineToLaneFusion* reads the latest data to perform its computation.

The data of the *LineToLaneFusion* is wrapped into message and send via the bus to the *SituationEvaluation*. As described above, the *SituationEvaluation* function gets triggering data from the functions *HumanActivityDetector* and indirectly the *LineToLane Fusion* via *ComTask1*. This activation is considered to be synchronous, as the situation evaluation needs both information to work appropriately.

The task *HumanActivityDetector* gets its needed sensory input via the bus system. Note that the sensors are not modeled explicitly, as these are not relevant for our analysis

function name	*vs*	*ld*	*l2l*	*m1*	*m2*	*had*	*seu*
period	80	-	20	-	20	-	-
exec.time	3	30	2	2	2	5	5
priority	0	1	2	0	1	0	1

Table 3.2.: Scheduling-relevant information for tasks and messages in milliseconds.

task. It is assumed that the time to fetch the relevant data of tasks which need sensory input (here *VideoSensing* and *ComTask2*) are included in their computation times.

All scheduling-relevant information for the tasks and messages are displayed in Table 3.2. In the table the task names are abbreviated as follows: *VideoSensing (vs), LineDetection (ld), LineToLaneFusion (l2l), ComTask1 (m1), ComTask2 (m2), HumanActivityDetector (had), SituationEvaluation (seu).*

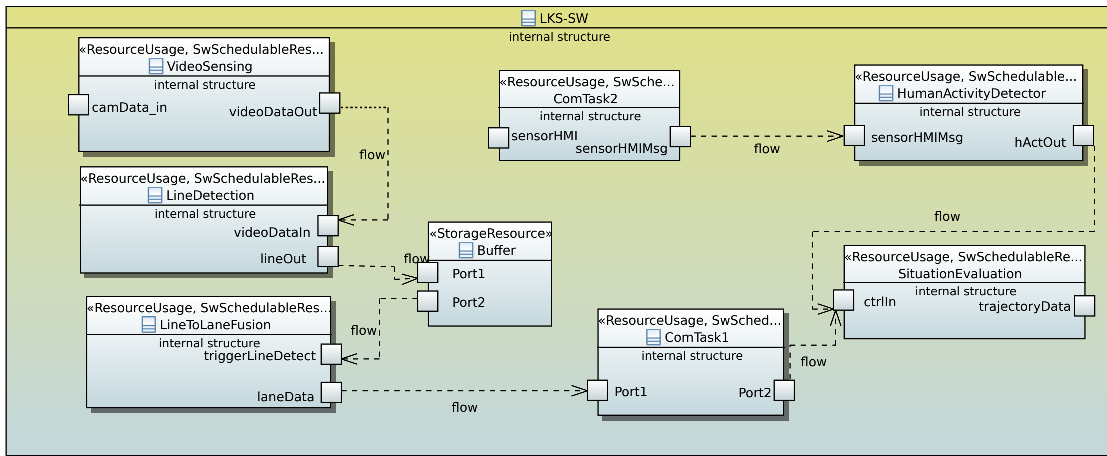


Figure 3.25.: Functional view of the lane keeping assistance system.

We are interested in the response times of the tasks *LineToLaneFusion* and *SituationEvaluation*, and the latency between these tasks. Further, the following latency constraint shall be evaluated.

```
whenever triggerLineDetect occurs, trajectoryData occurs during [0ms,
50ms].
```

The event *triggerLineDetect* triggers the task *LineToLaneFusion (l2l)* according to its period illustrated in Table 3.2, i.e. the event is fired every 20 time unit. The event *trajectoryData* is generated, whenever task *SituationEvaluation (seu)* terminates.

Timing of task/latency	(a) plain	(b) hp-clocks	(c) burst	(d) testing
$LineToLaneFusion$	$[2, 35]$	$[2, 35]$	$[2, 35]$	$[2, 35]$
$SituationEvaluation$	$[5, 13]$	$[5, 13]$	$[5, 13]$	$[5, 5]$
$LineToLaneFusion \rightarrow SituationEvaluation$	$[9, 47]$	$[8, 47]$	-	$[16, 25]$
Number of states *CANBUS* → *SituationEvaluation*	6.588	3.471	823	6.588
Number of states *SituationEvaluation*	73.568	52.328	4.317	68

Table 3.3.: Timing Analysis results in milliseconds for (a) analysis without abstraction, (b) using abstraction of period clocks, (c) using burst abstraction, and (d) applying testing.

3.7.3. Evaluation Results

The analysis results are listed in Table 3.3. The fist column (a) contains the exact timings for the tasks *LineToLaneFusion* and *SituationEvaluation* and the end-to-end response time between both tasks. For this, the implementation of our iterative analysis approach without the application of any abstraction was applied. This analysis was the slowest in contrast to the ones where abstractions were applied. This corresponds to the number of states which were computed and also listed in the lower part table: The size of the state space of the interface STS between *CANBUS* and *SituationEvaluation* consists of 6.588 states, while the resulting state space of resource *SituationEvaluation* consists of 73.568 states.

The second column (b) gives the timing results by applying the period clock abstraction technique introduced in Subsection 3.6.2. In particular, the period clock of task *VideoSensing* was abstracted. The effect is that the best-case response time of the end-to-end latency is lower than the original one, thus resulting in larger timing interval. In contrast to this, the state spaces of both the interface STS and the STS of resource *SituationEvaluation* get much smaller, resulting in a faster analysis.

The third column (c) lists the timings when the burst abstraction of Subsection 3.6.6 is applied: This approach results in the exact response times for both tasks, but – as it was described previously – end-to-end response times cannot be determined when this abstraction is used. As an end-to-end clock is not used to determine this latency, the state spaces get much smaller.

Finally, the testing approach sketched in Subsection 3.6.5 is applied. In particular, the algorithm selects just a single trace from the interface STS between *CANBUS* and *SituationEvaluation*. This trace is used as the single input of the resource *SituationEvaluation*. As illustrated in the table, this results in 68 states for the resource STS. The resulting timings are illustrated in the last column (d). Here, we can see that the input results in a path of the resource STS, where the response time of task *SituationEvaluation* is exactly $5ms$. The interesting part here is the end-to-end latency of $[16, 25]ms$. This is exactly what we expected, i.e. neither the best-case nor the worst-case timings were found. The determined execution time frame gives no guarantee that the timing constraint always

holds but represents possible behavior. This result can only give a hint to what latencies can be expected. Still, it is a usable approach if e.g. the original analysis does not terminate because of a too large state space. The testing approach then can be used to iteratively validate such an end-to-end latency.

3.7.4. Observation on Scalability

When performing the verifications on the introduced case study by using our prototype implementation, we determined some characteristics which influence the run times of the verification.

First, the scalability of the approach heavily depends on the applied abstractions. If a detailed verification is performed where no abstractions are applied, the verification is able to handle only small systems like the holistic model-checking approaches in the literature. Second, major aspects of the systems under analysis which negatively influenced the scalability of the approach were the following:

- Increasing number of independent tasks: The more independent tasks were defined, the larger the state spaces got. The reason for this effect is the initial non-determinism of the initial triggering of tasks, which is defined by a time interval. With this, the product construction leads to large state spaces, as all possible activation scenarios are constructed. When the initial non-determinism is deactivated, the resulting state spaces were much smaller. As an example, for two tasks only two possible activation scenarios could occur after deactivation.
- The difference between the characteristics of tasks: If the periods and the execution times of two (or more) tasks are similar, the resulting state spaces are much smaller than in scenarios, where these characteristics are diverging. The reason for this is that in more heterogeneous scenarios more states are necessary to keep track of the individual states of the tasks.
- The length of task chains: As more clocks are involved in such scenarios, which leads to an increasing state space.

The challenge here is to find the appropriate abstraction level for the verification task, such that on the one hand the result is usable (i.e. not too pessimistic) and on the other hand delivers results in an acceptable amount of time.

3.8. Summary

In this chapter our state-based scheduling analysis technique for distributed real-time systems was presented. The state space of the entire system architecture is defined by symbolic transitions systems (STS) and is constructed in an iterative manner. For this,

two main operations on STSs were introduced, namely the product construction and the abstraction of clocks and locations of an STS of interface STSs. It was shown that the product construction preserves timing properties and thus can be adequately used to successively build the sate space of the whole system architecture. Based on an input STS of a resource describing the activation times of the allocated tasks, an algorithm which builds the STS of this resource was described. Based on the state space of a resource, response times are determined. This iterative analysis enables checking end-to-end constraints in a more efficient manner than a holistic analysis as only the relevant parts of the state space are kept, while preserving all interleavings and task dependencies. Note that by *enabling analysis in a more efficient manner* we refer to the reduced state spaces with which the analysis concept has to deal in contrast to holistic approaches. We did not compare the introduced approach with state-of-the-art tools as the prototype implementation of the concept is not suited and optimized for such purposes.

Thereafter, abstraction techniques in the context of the presented iterative analysis were introduced. First, the effect of replacing two clocks by a single one was discussed. The abstraction of clocks of interface STS was illustrated and the resulting over-approximated STS was explained. The timed simulation relation was then introduced to apply an approximation technique, which tends to yield smaller state spaces for holistic analysis approaches or local state spaces, but affects in larger state spaces for dependent resources especially for iterative analysis approaches such as ours. As a last technique the abstraction for burst-scenarios was introduced. For all abstraction techniques appropriate examples were studied and their effects have been demonstrated.

Last, the applicability of the approach on a driver assistance system case study was demonstrated. For this, the lane-keeping-support subsystem part was detailed. In particular, the hardware and task architecture in terms of a MARTE model was introduced, and the timing analysis was performed on this model. We compared the response times and state spaces of our basic approach and the results of the applications of the abstraction techniques. The positive effects on the sizes of the number of states could been demonstrated, which boosts the scalability of state-based analysis approaches.

4. Contract-based Impact Analysis

4.1. Motivation

The design of a system typically consists of several specification, implementation, and analysis steps. Within these steps the specification, the architectural realization, and implementations are created to fulfill and realize the corresponding requirements. During the development process of such a system, business needs could change, or initially assumed aspects or goals could get refined, affecting the requirements of the system under design. Thus, the derived specification and the designed architecture typically are subject of changes. To give only some examples: Adaptations of the architecture of an already existing and analyzed system could be the removal of some resources due to cost savings, such that the tasks on this resource have to be re-allocated to the remaining resources. Another source of changes are incremental development cycles, where a previous version of a system is reused to include new features. Also technical issues could cause changes: If during the integration phase errors occur as interfaces do not fit together, the specifications of individual components must be modified, which also could affect the corresponding implementations.

As stated before, there are several proof obligations in safety-critical systems to assure the correct service and to prevent failures which could lead to critical situations. When changes occur, all verification steps must be repeated to guarantee that the changes do not violate any requirement. The impact analysis introduced in this chapter targets such re-verifications by determining the affected parts of the system by a change, thus preventing to perform all verification steps from scratch. With this technique it is possible to keep re-verifications local in certain cases. Suppose that the specification of a system component shall be replaced. If the newly adapted specification *refines* the previous specification, only the internals of the new component itself have to be re-verified without the need of a repetition of a complete integration check of this component with its context.

Refer again to the overall content of this thesis illustrated in Figure 4.1: The content of this chapter introduces the impact analysis methodology together with its basic techniques. If some parts of the architecture need to be re-verified on implementation level, the impact analysis will use the state-based timing analysis approach introduced in Chapter 3.

This chapter is organized as follows. At first, related works on verification tools for contract specifications and on approaches reducing the effort of performing re-verifications are presented. In Section 4.3 the overall concept of the impact analysis approach is intro-

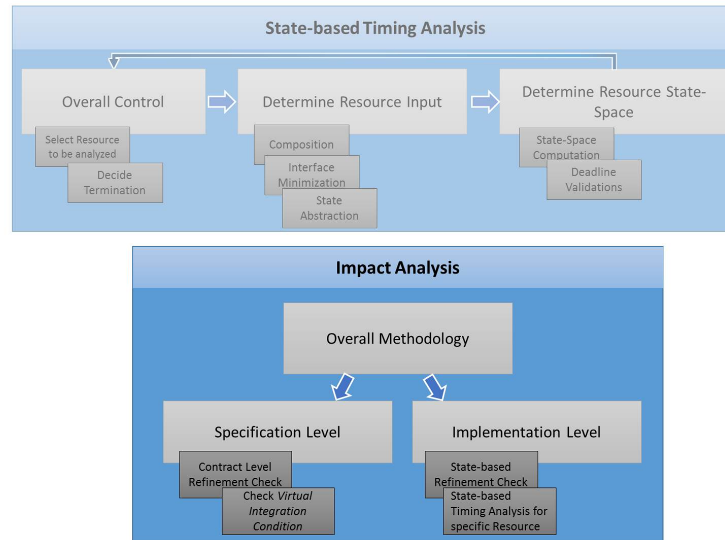


Figure 4.1.: Overview of contributions of thesis.

duced. It integrates two verification techniques, one on the specification and the other on the implementation level of a system design. The techniques on both level are detailed in the subsequent Sections 4.4 and 4.5. The introduced approach is evaluated in Section 4.6. Finally, all results of this chapter are concluded in Section 4.7.

4.2. Related Work

The theoretical foundations and practical applications of the contract-based specification method were elaborated in [BCF+08, DHJ+11] as introduced in Chapter 2. Here, two aspects of related works are detailed. The first aspect deals with tool support for the automatic verification of contract specifications. The second aspect is the reduction of re-verifications when changes of the system under design occur.

4.2.1. Tool Support for Verification of Contract Specifications

OCRA (Othello Contract Refinement Analysis) is a tool to check contract refinement and consistency [CDT13]. The applied architectures are closely related to the architectures considered in this thesis: Components are decomposed to a set of interconnected subcomponents, and each component is annotated with corresponding contract specifications. Contracts are specified by the use of a temporal logic called *Othello* (Object Temporal Hybrid expressions Linear-time temporal Logic) [CRST13], which is a human readable

language. The semantics of this language is given in terms of an extended version of linear time temporal logic (LTL) called HRELTL (Hybrid Linear Temporal Logic with Regular Expressions). In contrast to this thesis, they use an SMT solver to check such relations. The backend engine of OCRA is *NuSMV3* which itself is based on the SAT solver *MathSAT*. MathSAT only considers quantifier free formulas, while NuSMV3 extends this to the logic of LTL. The tool is also able to verify implementations against contracts. In contrast to this thesis the approach is restricted to discrete-time LTL contracts and finite state machines. The main result of the authors concerning the applicability of their approach is that their tool scales well with the number of decomposition structures. The reason for this lies in the nature of contracts, as the verification steps are performed locally within structures.

The focus of the authors of [BMSH10] are modal input/output automata (MIO). With this model, two types of behaviors can be described, i.e. allowed and required behaviors, modeled as *may* and *must* transitions respectively. Further, as in timed automata, input, output, and internal actions are distinguished. In their work, modalities are applied to refinement and compatibility, as detailed in the following.

A MIO A_1 *strongly modally refines* another MIO A_2 which is defined over the same signature, if there is a strong modal refinement relation between A_1 and A_2, where A_1 can simulate all must transitions of A_2 immediately, and whenever A_1 can perform a may transition, A_2 can simulate this with a corresponding may transition. *Strong modal compatibility* between two MIOs A_1, A_2 requires that if A_1 can send an output action – either by a may or a must transition – which is in the input alphabet of A_2, then A_2 *must* immediately be able to receive the message and is not allowed e.g. to take internal transitions before.

Weak modal refinement does allow internal actions before must transitions are fired, i.e. if the more abstract MIO is able to fire a must transition with an output action, the refining MIO is allowed to fire a finite set of internal must transitions, but has finally to simulate the must transition of the abstract MIO. With this notion of refinement, more flexibility is allowed by e.g. the modeling of internal computations during the refinement process. *The weak modal compatibility* is defined accordingly, i.e. where internal must-transitions are allowed before the synchronization has to be performed. The weak modal compatibility allows a more loosely coupling of interfaces.

This approach has also been implemented in the MIO Workbench [BML11], which is an Eclipse-based editor for modal I/O automata equipped with a verification back-end. The verification back-end is able to check strong and weak modal compatibility and refinement. This work is related to this thesis, as it also deals with automata-based verification of properties like composition and refinement. In contrast to this thesis, the authors considered only untimed systems and did not consider contracts explicitly.

In [AdADS+06] a tool called TICC (Tool for Interface Compatibility and Composition) was introduced. It checks the compatibility of component interfaces, where untimed transition systems with input and output actions and discrete variables encoding the

states are considered. Components are modeled based on game theory, where components are represented by automata with input and/or output actions on transitions. The semantics of such automata is given by the well-known two-player games, where the input player represents the environment, and the output player represents the component itself.

The authors later extended this work in [DLL$^+$10b] for dense time systems by introducing the timed input/output automata (TIOA), which are based on a timed version of the game semantics, and provided also a tool called ECDAR (Environment for Compositional Design and Analysis of Real-Time Systems) [DLL$^+$10a]. In contrast to standard timed automata, the transitions are differentiated between input and output transitions. Edges controlled by the environment are annotated by receiving events, while edges controlled by the automaton itself are annotated by sending events.

The designer can create automata for both the implementation and specification of a system. After the creation, refinement and consistency checks between all automata can be performed through the integrated game engine, which is based on Uppaal-Tiga. For this, appropriate queries are defined to enable the verification engine. The consistency check verifies whether there exists an implementation which fulfills its corresponding specification. Thus, a strategy for the output player is searched, which prevents the specification to reach a bad state. An example to verify the refinement: Let the implementations in terms of TIOA $N_1, ..., N_3$, and an overall system specification automaton S be given. To check the refinement, the query *refinement : (N1 || N2 || N3)* $\leq S$ is created. In the sense of compositional design, all implementations could also get sub-specifications, which are again TIOA. Then, the refinement for each implementation against its specification would be checked, and at least the correct composition of all sub-specifications with the overall specification would be verified. The approach is a bit unintuitive, as both the specification and implementation have to be created in terms of TIOA. Further, the authors did not detail the scalability of their approach. They only give a hint that the scalability gets better, if sub-specifications are created, instead of performing a holistic analysis against the overall specification [DLL$^+$10a].

In [CGP03] the authors state that the definition of appropriate assumptions for component contracts is a complex task, and thus they claim that this task has to be automated. For this, they present an algorithm called L^* which iteratively learns assumptions of components and their behavior. This learning is based on refinement. They consider finite discrete traces produced by labeled state transition systems. The paper is restricted to cases where the system is composed of two components M_1, M_2. The goal is to decompose the system property $C = (true, G)$ to sub-contracts $C1 = (true, A)$ and $C2 = (A, G)$, i.e. to find an appropriate assumption A which is guaranteed from component M_1 (and thus contract $C1$). The algorithm iteratively computes an intermediate assumption A_i consisting of a set of finite words. This A_i is then used as the context for M_2, and it is checked whether M_2 is able to fulfill the property G. If not, the assumption is too weak such that further behavior has to be removed. To do so, the counter-example trace returned from the used model checker is applied. If otherwise the found assumption A_j

is strong enough for M_2 to guarantee G, is checked in a second step whether A_j is not too strong for the behavior of the predecessor component M_1. As in the first step, if A_j is too strong, then A_j needs to be weaken by adding further behavior via the usage of the returned counterexample. In the other case, it holds that the property G holds for $M_1||M_2$. The authors extend the learning algorithm (among other things) to a chain of components in [PGB+08]. The basic techniques to check refinements of contracts are also applied in this thesis. In contrast to this thesis, the authors focus on the automatic derivation of appropriate component assumptions, while in this work the focus is on the application of the virtual integration of components into their context.

4.2.2. Impact Analysis

In the work of [KHM+96] the system behavior is given as a set of functions, which may depend on each other. The work is based on regression testing, where a set of tests are defined for each function. If a test fails, the error is corrected and all tests are re-run. Also previously successfully tested functions are re-tested to ensure, whether the changes did not affect these functions. The authors applied this idea to formal verification and called the approach *regression verification*. The first part of the work is to apply an algorithm called *localization reduction* which abstracts a model P relative to a given property φ. That is, the algorithm determines the part of P which is relevant for the corresponding property φ, and from this it builds a reduced model P', which is smaller than P and thus computationally simpler. The basic idea of this reduction is the application of the language inclusion of automata theory, where an automaton P' is a reduction of P, if $\mathcal{L}(P) \subseteq \mathcal{L}(P')$. Further, P' is obtained from φ and P, such that it is correct by construction, i.e. $\mathcal{L}(P') \subseteq \varphi$ holds. The second part targets the problem of storing the models: The reduced models are saved by the use of hash functions. With these functions only the hash value of the reduced models are stored. If changes occur, the corresponding (new and old) hash values are compared to determine whether the verification result is affected.

The result of their work is that the computations of reduced models together with the computations and the checks of the hash values is less complex than the original checks determining whether a system satisfies a given property. Moreover, the storage of models is omitted, as only the corresponding hash values are stored. The approach has been implemented in a verification engine called *COSPAN*. The idea of the work of [KHM+96] is closely related to this thesis. The authors do only check a part of a given model. In contrast to this thesis, this is realized by building a new abstract model which is sufficient to re-check a given property. Our approach does not need to calculate an abstract model as it is able to determine indirectly affected parts of the given architecture, which then are re-checked. Further, the authors realize a holistic analysis, whereas the impact analysis on the specification level of this thesis operates in a compositional manner.

The approach of [BLN+13] targets the CEGAR-based (Counterexample-Guided Ab-

straction Refinement) verification approaches: In the basic CEGAR approach first an appropriate abstraction is searched, which is detailed enough to not cause a spurious counterexample – i.e. a trace produced by the abstract model which violates the property but is not in the trace set of the original model – and coarse enough such that the verification of the abstraction is simpler than the original model. This level of abstraction is called *precision information*. The CEGAR approach starts from a coarse grained model (e.g. where all information is abstracted) and refines the model until the property is verified and no spurious counterexample could be found. Within this refinement process constraints to the abstract state space are derived, which prevent that the spurious counterexamples can be produced. When changes in the program occur, the whole process has to be repeated. The authors target this problem by storing the precision results from CEGAR and make these available, if a repetition of the verification has to be performed. With the loaded precision information the initial abstract model is generated. This approach helps to significantly reduce the number of refinement steps of CEGAR, which is a costly operation according to the authors. The authors extended an existing verification framework to realize their approaches and used a Linux device driver to evaluate their approach. The general idea is related to our approach: Already computed results are stored and loaded whenever re-checks are necessary. In contrast to our approach, the focus of [BLN+13] is on the minimization of applying refinement steps to determine an adequate level of abstraction after changes are applied. Thus, the approach targets the re-computation of the abstract model and then to perform a holistic analysis on the basis of this model.

The work of Beyer and Wendler [BW13] deals with different strategies to reuse verification results, which are the conditional model checking, the verification witnesses, and the precision reuse already illustrated above. The conditional model checking approach targets to use partial verification results of previous verification runs, in cases where the verification tool could not fully complete the verification because of e.g. memory exhaustion. The original approach was introduced in [BHKW12]. The idea of this technique is to perform the verification iteratively. For this, it is specified which parts of the system should be analyzed. This is realized by constraining the input of the verification tool. The verification tool analogously delivers a condition which describes the part of the system, which has been analyzed. This output condition then serves as an input condition for the next verification run, which omits to re-verify the already analyzed part. Using this approach different verification engines could be combined by calling them sequentially or combining them by partitioning. In the former case the output condition of the first engine servers as an input condition of the second engine. In the latter case, the state space is partitioned and each verification engine is started with its input condition represented by the corresponding partition. This approach is closely related to our timing analysis approach of Chapter 3, where we iterate through the resources of a given architecture and use the verification results of a resource as an input for the verification of a dependent resource. In the context of the impact analysis methodology we will extend

this approach in this chapter to re-use already analyzed resources by storing and loading the state spaces. In contrast to [BW13] we focus on timing properties, whereas the work of [BW13] targets the verification of functional properties.

The verification witnesses of [BW13] targets the re-verification of *witnesses*, which could be positive or negative. In the paper the re-verification is performed by using a counterexample returned from the verification program, thus checking if it is still valid in the changed program. For this, the generated counterexample has to be machine-readable. The result of the authors is that the reuse of verification results can save time and memory needed by the used model checker. They claim that a standardized format to represent the verification results is needed in order to combine verification tools effectively. Storing witnesses is not in focus of our work, but is an interesting approach to extend our impact analysis methodology: If a given system violates a property, a trace leading to a bad state is stored. After changes are applied to the system, the validity of the stored trace could be checked before the actual verification is triggered. If the system does still exhibit this trace, the verification could be saved, as the system still does not satisfy the property.

The authors of [OGR14] have investigated the benefits and limitations of a localized change management process. Similar to this thesis, contracts are used to identify design elements, which are affected by a change. The authors focus on the change management process, where a set of standard tools like Doors are integrated. Changes on the model entities within these tools are witnessed by their prototype implementation, which then triggers the needed V&V activities. In contrast to our work, they focus on the change management process and on functional safety properties, while the focus of this work is on the analysis. The approach of [OGR14] could be seen a preceding process which triggers our approach and delivers a list of changed artifacts of the underlying system architecture.

In the previous subsection the contract verification tool OCRA was mentioned. This tool is also a related work for the impact analysis part, as a feature of OCRA is the reuse of components and their implementations by a library functionality [CDT13]. If an implementation is reused from the library, model checking against its context contract can be omitted by checking whether the contracts, which the implementation fulfills, refine the contract of the new context. This is closely related to our impact analysis approach: If the contract of the implementation changes, either a re-verification of the component itself is necessary, or it is sufficient to do the refinement check between the old and the new contract. In contrast to our work, the approach is restricted to discrete-time LTL contracts.

4.2.3. Contribution of this Chapter

As discussed in the last sections, automatic re-verification has been the target of a couple of works in the literature. What is missing is an integrated overall impact analysis

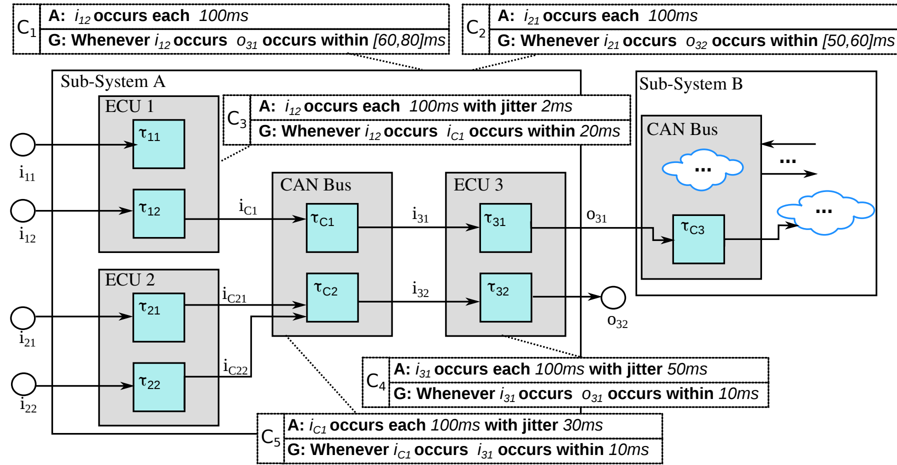


Figure 4.2.: System architectures of interest.

methodology for dense time systems and the application of the contract theory. This methodology shall be intuitive to use for system designer without the need to learn new formalisms or in-depth details in verification techniques.

Our approach applies the contract theory and standard component-based design techniques such that a system can be captured in an intuitive manner. By using contracts our impact analysis is able to operate in a compositional manner, as parts of the system annotated by context assumptions and guarantees to its context can be treated completely in isolation. Our approach does not need to calculate an abstract model to verify timing properties. It is able to determine indirectly affected parts of the given architecture, which then are re-checked. Further, the timing analysis approach of Chapter 3 is integrated in the impact analysis methodology, such that besides specification aspects also implementation aspects can be verified by using the same formalism.

4.3. Impact Analysis Methodology

Every time a change in the design of a system occurs or requirements are changed during the development process of a system, a re-verification of its properties has to be performed to ensure correct functionality. To minimize the effort of re-verifications, it is desirable to reuse previous analysis results, which still do hold after performing changes to the design. For this, parts of the overall design have to be determined, which were not affected by the changes. This section targets the minimization of such re-verifications. Thereby, the focus is on timing properties. For this, the state-based timing analysis technique introduced in Chapter 3 is extended by an impact analysis approach. Two abstraction

levels are defined, which could be affected by changes, which are the specification level on the one hand, and the implementation level on the other hand.

On the *specification level* a *virtual integration checking* technique is introduced through the concept of reachability analysis of timed automata. Suppose that the specification of a system component in terms of assume/guarantee style contracts shall be replaced. If the new contract of the adapted specification *refines* the previous contract, the internals of the new component itself need to be checked. Thus, the effects of such changes are kept local, and only a part of the overall architecture has to be re-verified. Of course, if the contract of a component is completely replaced, it has also be checked whether the new specification is consistent with its context.

If resources are not annotated by contracts, on the *implementation level* state transition systems describe the behavioral interfaces between dependent resources. For such cases, an appropriate refinement relation between these state transition systems on the interfaces of resources is defined. With this notion of refinement, re-verifications of parts of the system on implementation level can be avoided.

Both *complete* specifications, which fully describe the interfaces between all components, and *incomplete* specifications, where the interface descriptions of only a subset of components is given, are considered in this thesis. In a design process incomplete specifications are quite typical as some properties of parts of a system may be obtained in later design steps. A complete interface specification is given for e.g. the task chain $\tau_{12}, \tau_{C1}, \tau_{31}$ in the architecture of Figure 4.2: The assumptions of the contracts of *ECU1*, *CAN Bus*, and *ECU3* describe the activation behaviors of the corresponding tasks τ_{12}, τ_{C1}, and τ_{31}. That is, the local contracts C_3, C_4 and C_5 can be verified locally without considering any behavior of the systems from their contexts. If the system specification is *complete* the virtual integration checking technique allows us to determine the impact of changes which may occur during the development process of a system without the consideration of any implementation detail.

For incomplete specifications a refinement check of the behavioral interfaces of the resources, which are determined by the iterative timing analysis technique illustrated in Chapter 3, is performed. An example of an *incomplete* specification is the task chain $\tau_{21}, \tau_{C2}, \tau_{32}$ in the same architecture of Figure 4.2. For this task chain only an end-to-end latency constraint is specified. To analyze the end-to-end latency constraint for such cases, the state-based analysis approach of Chapter 3 is applied. If changes in the architectures occur, the approach is able to handle these appropriately. For this, during the iterative analysis the verification results are stored and can be reused in later design steps.

Generally, in a design process both approaches are exploited in combination: When a component is replaced, the new component has to be integrated in the existing context, i.e. the impact on the specification level has to be determined. In a second step, the impact on the implementation level as described above has to be checked and the necessary re-verifications have to be performed. By applying the introduced impact analysis

approach such re-verification steps can be reduced on both the specification level and on the implementation level.

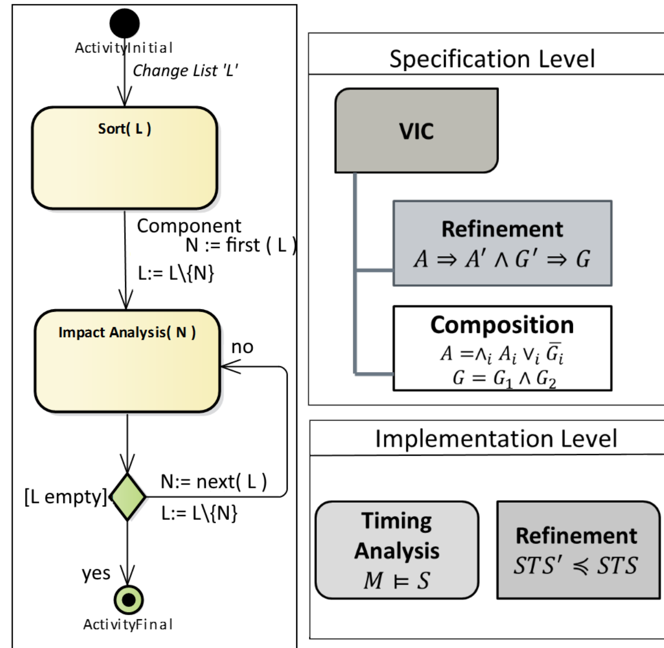


Figure 4.3.: Impact analysis methodology - Left: General control flow; Right: Integration of verification techniques.

The general control flow of the impact analysis methodology is illustrated in the left part of Figure 4.3. As the input a (possibly empty) list of all changed components of the system under design is passed to the process. This list contains all components, which were either structurally changed or their contracts were changed. The list is empty, if the system is analyzed the first time.

First, the components in the list L are put into an order by a procedure called *Sort*. The algorithm puts the components first in a top-down order, i.e. components on a higher abstraction level are treated first. Note that a component consists of either resources or is composed of subcomponents. These subcomponents are again composed of a set of components or resources. It is assumed that a component does not consists of a mixture of resources and composed subcomponents. On the resource level, the order to handle the resources is determined by the task dependencies. For instance, resource R_i contains the task t_i and is affected by a change. The resource R_j contains the task t_j which (directly or indirectly) depends on t_i. Then, first R_i has to be handled before R_j. Note that for parts of the considered system consisting of resources with cyclic dependencies no order can be derived. As stated in Chapter 3 these parts have to be treated in a

holistic manner.

For the contract level, an order is not relevant, as the explicitly specified assumptions allow us to perform verifications locally.

The logic called *Impact Analysis* then has the general control over the applied techniques illustrated in the right part of Figure 4.3, which are on the specification level the virtual integration check (VIC) consisting of the refinement and the composition check, and on the implementation level the iterative timing analysis and the refinement check on computed state spaces. The control flow of this procedure is detailed in Figure 4.4.

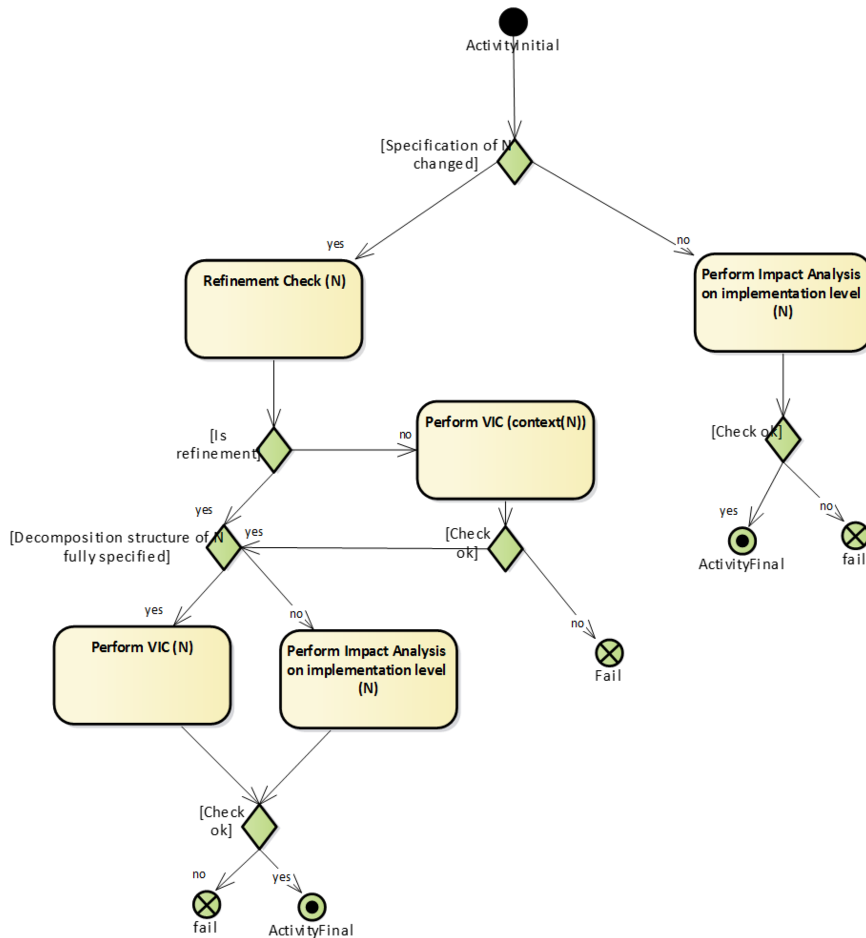


Figure 4.4.: Impact Analysis procedure for a component N.

Let N be the first selected component by the *Impact Analysis* algorithm. If the specification of N is changed, the left branch is taken, else the right one. It is assumed that if

the specification is not changed, the component N is either a resource or a composition of resources. In such a case the changes on N concern the implementation aspects, and the procedure *Perform Impact Analysis on Implementation Level* is called, which consists of the timing analysis and the refinement check on state transition systems as described above. Within this procedure, the timing analysis is performed for the directly affected components. Thereafter, it is checked whether the new computed state spaces of the affected resources refine the previous state spaces (computed before the corresponding change occurred). If the new state spaces do refine the previous ones, then the dependent resources do not need to be verified. The techniques on this level are detailed and addressed in Section 4.5.

If a change concerns the specification then the left branch is taken, where it is first checked, whether the new contract of N refines its old contract. If the refinement holds (and thus the guard *Is refinement* in the figure evaluates to true), the context of N is not affected, and only the internals of this component have to be verified. If thereby the decomposition structure of N is completely specified by contracts, the virtual integration check of N and its decomposition structure has to be verified. That is, it is checked, whether the contracts of the decomposition structure do fit to the new overall contract of N. Otherwise, if the decomposition structure has not a complete specification, the impact analysis on the implementation level is triggered, which was already described above. Note that if the decomposition structure is not complete, it is assumed that this structure corresponds to a set of resources. This is no restriction to the general case, but facilitates the description of the approach. Note that if only the guaranteed end-to-end latency of N is changed, only a look-up in the state spaces of the *last* resources has to be performed to verify whether the new latency guarantees are still fulfilled. Otherwise, if the assumptions like task activation times are changed, the timing analysis has to be repeated for affected resources.

Otherwise, if the change of N is not a refinement, the composition to its dependent resources could be affected such that the virtual condition check has to be performed for its context, which is obtained by the method *context(N)* in the figure. Note that if for the component no context is defined, the method *context(N)* returns *nil* and the VIC is skipped. If this check does not fail, further the internals of the component need to be checked as already described.

For the overall component of N (obtained by the method N' :=*context(N)*) the VIC has only to be performed once. If there are several subcomponents of N' with changed contracts in a non-refinement manner, the logic would call the same VIC several times, which is not necessary. To prevent this repetition of the VIC, a flag is set for a component N', when the VIC for N' is called the first time.

The applied analysis techniques of this branch are detailed in the next section.

As illustrated in the left part of Figure 4.3 the *Impact Analysis* procedure operates until all components in the change list are handled.

Note that if an analysis like the VIC determines that a property is not fulfilled, the process terminates in a *fail* state, which indicates that the process is terminated (with an appropriate error message). In this case, the changes violate some integration aspects or the implementation is not able to fulfill the new guarantees under the new assumptions.

In the next sections, all referred techniques will be illustrated. Note that the introduced impact analysis approach was previously worked out and published in [GHSR14].

4.4. Impact Analysis on Specification Level

As previously stated, the specification of a system is subject to change. At early stages of the design process details are not known such as the complete scope of the functionality or architectural decisions which may affect certain performance aspects. Some requirements may arise during the development process which were not considered at first. During the design process such details get more and more concrete, which typically lead to changes and refinements of existing requirements.

In this thesis requirements are specified by the usage of assume/guarantee style contracts. If the contract of a component N is changed and the new contract *refines* the previous one, only the internals of N need to be re-verified as its context is only affected in a positive manner, which means that all properties still do hold. If N is decomposed in subcomponents with local contracts, it has to be checked whether the new component contract is still fulfilled by the composition of the contracts of all subcomponents. As an example, consider the following contract (A_K, G_K) of a component K:

(A_K) $sensor_{in}$ `occurs each` $10ms$
(G_K) `Whenever` $sensor_{in}$ `occurs,` $actuator_{out}$ `occurs during` $[0, 50]ms$

Suppose the guarantee is changed as follows:
(G'_K) `Whenever` $sensor_{in}$ `occurs,` $actuator_{out}$ `occurs during` $[0, 30]ms$

The component K with its new contract (A_K, G'_K) will deliver its output signal to its dependent components not later than $30ms$ anymore after activation. As the dependent resources are already able to handle the time interval $[0, 50]$, they are also able to handle the new interval. Thus, there is no need to analyze this property. On the other hand, the internal implementation of K has now to realize this stricter timing latency. It has to be checked, whether there exist scenarios, in which the timing latency of $30ms$ cannot be met.

Otherwise, if the new contract is not a refinement of the old one, an additional consistency check with all dependent parts of the system has to be performed. Thus, it has to be re-verified whether the component can be virtually integrated to its intended context.

Such a virtual integration check operates on the specification level of a system without the need to consider all implementation details at once.

In the case where a part of the system is changed, and the system is *fully* specified, the introduced approach is able to keep the effects of changes local such that only a part of the whole architecture has to be re-verified.

For such scenarios a timed automaton-based approach is introduced in this section, with which it is possible to automatically validate *virtual integration* aspects. This integration encapsulates the correct composition of components with respect to their contract specifications, and the correct refinement of a higher level contract. The corresponding condition for this was introduced in Subsection 2.4.1. Next, relevant properties of contracts will be inspected, which are *directedness* and *saturation*. From these properties the simplification of the original virtual integration condition will be derived, which enables the usage of timed automata to realize an appropriate verification approach.

4.4.1. Simplifying the Virtual Integration Condition

In Subsection 2.4.1 the virtual integration condition was introduced. This condition summarizes the composition and refinement checks of components, which have to be applied when components are decomposed to a set of interconnected subcomponents to guarantee correct interaction and the consistency of the system specification. Unfortunately, this condition includes negations. As a timed automaton approach shall be applied in this thesis to verify this condition, and timed automata are not closed under complementation, this section deals with the simplification of this property by using some characteristics of contracts.

Saturated Contracts

In Definition 12 it was already stated that the implementation M of a component satisfies a contract $C = (A, G)$ of that component, if $[\![M]\!] \subseteq [\![\neg A]\!] \cup [\![G]\!]$. From this, the maximal possible allowed behavior of a component can be directly derived, which corresponds to the following:

$$[\![M_{max}]\!] = [\![\neg A]\!] \cup [\![G]\!]. \tag{4.1}$$

This was the topic of [BCN+11] where the term *saturated* was defined for contracts.

Definition 24 (Saturated Contracts)**.** *Let $C = (A, G)$ be a contract, where A, G are assertions over the interface I with $I(A) \subseteq I(G)$. C is saturated, if $[\![G]\!] \supseteq [\![\neg A]\!]$.*

Note that for an assertion E the function $I(E)$ determines the set of ports, over which it is defined. A contract is saturated, if the behavior which is not captured by the environment assumption is included in the guarantee of the component. In [BCF+08] this property is called *canonical*, as the maximal allowed behavior M_{max} is unique. Each

contract can be transformed to a saturated one (canonical form) by adding all traces of $[\![\neg A]\!]$ to the guarantee.

For instance, consider a component with an input port *in* and an output port *out*. Let the component compute the square root of the values received on the input port, i.e. $out = \sqrt{in}$. The component is further annotated by a contract, where its assumption states that the values on the input port shall be greater than or equal to zero. The contract is saturated, if (besides the allowed input values) the guarantee defines also a (default) behavior for negative input values, e.g. $out = NaN$ if $in < 0$.

This property of contracts simplifies the virtual integration condition. If all sub-contracts C_i of a composite are saturated, the first condition of the virtual integration condition (cf. Lemma 3) simplifies to

$$A \wedge G_1 \wedge \cdots \wedge G_n \Rightarrow A_1 \cdots A_n.$$

This is exactly what we want to achieve, i.e. to get rid of the negation part of this constraint. Unfortunately, if the sub-contracts are not in saturated form, we have first to transform the guarantees, i.e. we have to extend the guarantee parts with the negated form of the corresponding assumptions. Effectively, we did not achieve anything, as the negation part has still to be taken into account when performing verifications. We only shifted the terms including negations into the guarantee part.

Receptiveness and Directed Contracts

Contracts are typically a black-box specification formalism and restrict the observable behavior of components. Specifications restrict the port valuations of components without distinguishing between input and output ports of the corresponding component. Thereby, input ports should be controlled by the environment while the valuations of the output ports should depend on the behavior of the component itself. During the design stage problems could arise if this distinction is not performed in a consistent fashion, such that a guarantee is specified, which restricts possible behavior of the input ports, or analogously the assumptions restrict the components output ports. Such contracts define faulty specifications, as the component is not responsible for the input behavior, and the environment is not responsible for the component behavior.

As an example let *requestData* be an output port and *dataIn* be an input port of a component. Consider the following two assumptions captured by an $R3$ pattern (cf. Subsection 2.4.2):

- A_1 : `Whenever requestData occurs, dataIn occurs during [5,10]`
- A_2 : `Whenever dataIn occurs, requestData occurs during [5,10]`

A_1 is *receptive* on the output port *requestData* as it allows occurrences of events on that port on arbitrary points in time. In contrast to this, A_2 is not receptive as it would make restrictions to the component behavior.

To distinguish the responsibilities of the assumptions and guarantees, in [BCF+08, BBB+11] the definition of *receptiveness* was introduced, which is illustrated in the following.

Definition 25 (Receptiveness). *Let I be a directed interface, $\Phi \subseteq \mathcal{L}(I) = (I^\omega \times \mathcal{T})$ a set of traces over I, and $I' \subseteq I$. Φ is called receptive on I', if it is non-empty and for all traces $\sigma \in \Phi$ and for all $n \in \mathbb{N}$ the following holds:*

$$\forall \rho \in \mathcal{L}(I).\ pre(\rho, n) = pre(\sigma, n) \ \Rightarrow\ \exists \sigma' \in \Phi.\ pre(\sigma', n) = pre(\rho, n) \ \wedge\ \sigma'_{\downarrow_{I'}} = \rho_{\downarrow_{I'}}.$$

This condition states that if an arbitrary trace ρ in the (complete) trace set $\mathcal{L}(I)$ is considered, any trace σ in Φ with an identical prefix can be arbitrarily extended with respect to the ports of I'. Thus, the trace set states no restrictions to the ports of I'.

In the following, an example for a non-receptive contract will be given to ease the understanding of this property. Consider a component with two input ports i_1, i_2 and an output port o. The guarantee of the component restricts input ports in such a way that always i_1 should occur before an occurrence of i_2. Let $\sigma \in \mathcal{L}(G)$ with $pre(\sigma, 1) = o$ and let $\sigma' \in \mathcal{L}(I)$ with $pre(\sigma', 2) = o \cdot i_2$. There exists a prefix ($n = 1$) such that the left part of the implication of Definition 25 is true, but the right can never be true as there is no trace in $\mathcal{L}(G)$ which can be extended in a way like σ'.

By applying the receptiveness property to the assumption and guarantee parts of contracts, one can adequately distinguish the responsibilities of the input and output behaviors of a component. Such contracts are called *directed*.

Definition 26 (Directed Contract). *A contract $C = (A, G)$ over an interface I is directed, if A is receptive on $out(I)$ and G is receptive on $in(I)$.*

Note that in the following the notions $in(I)$ and $out(I)$ will be abbreviated to in and out respectively, if the port set I is clear from the context.

For directed contracts the following corollary is derived:

Corollary 1. *Let $C = (A, G)$ be a directed contract over an interface $I = in \cup out$. If $in \subseteq I(A)$ then $\mathcal{L}(in) = [\![A_{\downarrow_{in}}]\!] \cup [\![\neg A_{\downarrow_{in}}]\!]$ (and analogously for the guarantee part).*

If an assumption restricts all input ports of a corresponding component, all possible words over the input ports is obtained by taking the union of the set of traces described by the assumption and its negation.

For a directed contract C it holds that the guarantee does not restrict the language given by the assumption, and vice versa the assumption does not restrict the set of words defined by the guarantee. Thus, the following properties are obtained:

$$A_{\downarrow_{in}} \cap G_{\downarrow_{in}} = A_{\downarrow_{in}} \quad \text{and} \quad A_{\downarrow_{out}} \cap G_{\downarrow_{out}} = G_{\downarrow_{out}}. \tag{4.2}$$

To simplify the virtual integration condition, assume that only such architectures are considered, where the guarantees are pairwise receptive on each other, i.e. no guarantee restricts the language of the other guarantees.

Further, the following corollary is needed.

Corollary 2. *Let A, B be assertions over the interfaces Σ_A, Σ_B respectively. Then like in propositional logic the following holds regardless of the cut set between Σ_A and Σ_B :*

$$A_{\uparrow \Sigma_B} \cup (A_{\uparrow \Sigma_B} \cap B_{\uparrow \Sigma_A}) \Leftrightarrow A_{\uparrow \Sigma_B}.$$

Proof. The trivial case for this is given, when both interfaces are equal, i.e. $\Sigma_A = \Sigma_B$. Let $X = \Sigma_B \backslash \Sigma_A$ and $Y = \Sigma_A \backslash \Sigma_B$ non-empty port-sets. Then according to Lemma 1 $(A_{\uparrow X})_{\downarrow X} \supseteq B_{\downarrow X}$ holds. Analogously, we get $(B_{\uparrow Y})_{\downarrow Y} \supseteq A_{\downarrow Y}$, but $(B_{\uparrow Y})_{\downarrow Y} \cap A_{\downarrow Y} = A_{\downarrow Y}$. □

With these ingredients, the following theorem is derived.

Theorem 4. *Let $C = (A, G)$ be a directed contract over an interface I, and $C_i = (A_i, G_i)$ with $i \in \{1, ..., n\}$ a set of directed sub-contracts over the interfaces I_i with pairwise receptive guarantees. Then the first condition of the VIC of Lemma 3 can be simplified as follows.*

$$A \wedge G_1 \wedge ... \wedge G_n \Rightarrow \ A_1 \wedge \cdots \wedge A_n.$$

The proof is performed for $n = 2$ which is no restriction to the general case but is easier to handle.

Proof. Let $C = (A, G)$ be a directed contract over I, $C_1 = (A_1, G_1), C_2 = (A_2, G_2)$ directed sub-contracts with pairwise receptive guarantees over I_1 and I_2 respectively. According to Lemma 3 the following proof obligation is obtained:

$$(A \wedge \neg A_1 \wedge \neg A_2) \vee (A \wedge \neg A_1 \wedge G_2) \vee (A \wedge \neg A_2 \wedge G_1) \vee (A \wedge G_1 \wedge G_2) \Rightarrow A_1 \wedge A_2.$$

In order to reduce the left part to only the expression $(A \wedge G_1 \wedge G_2)$, it has to be shown that all the other sub-expressions are subsets of this expression. Let $I_{asmp} = I(A) \cup I(A_1) \cup I(A_2)$, $I_{grnt} = I(G_1) \cup I(G_2)$, and $I_{cmpl} = I_{asmp} \cup I_{grnt}$.

Let us first focus on the first sub-expression $(A \wedge \neg A_1 \wedge \neg A_2)$. The subset relation does only hold for the interface I_{asmp}, and not for I_{cmpl}, i.e. $(A_{\uparrow I_{asmp}} \cap \neg A_{1 \uparrow I_{asmp}} \neg A_{2 \uparrow I_{asmp}}) \subseteq A_{\uparrow I_{asmp}} \cap G_{1 \uparrow I_{asmp}} \cap G_{2 \uparrow I_{asmp}}$: Because of the receptivity property it holds that $A_{\uparrow I_{asmp}} = A_{\uparrow I_{asmp}} \cap G_{1 \uparrow I_{asmp}} \cap G_{2 \uparrow I_{asmp}}$. According Corollary 2 it also holds that $A_{\uparrow I_{asmp}} \supseteq A_{\uparrow I_{asmp}} \cap \neg A_{1 \uparrow I_{asmp}} \cap \neg A_{2 \uparrow I_{asmp}}$. From this, it can be followed that the above expression does hold for I_{asmp}. To conclude the proof for this expression, we have further to inspect the words on I_{grnt}: As A is receptive on I_{grnt} it holds that $A_{\uparrow I_{grnt}} = \mathcal{L}(I_{grnt})$. But this also holds for the RHS of the verification condition, i.e. $A_{1 \uparrow I_{grnt}} = A_{2 \uparrow I_{grnt}} = \mathcal{L}(I_{grnt})$. Because of this, it is valid to simplify the verification condition by leaving out the term $A \wedge \neg A_1 \wedge \neg A_2$.

Secondly, consider $(A \wedge \neg A_i \wedge G_j)$ for $i, j \in \{1, 2\}, i \neq j$. As the guarantees are pairwise disjoint, and $\neg A_i \wedge G_j \subseteq G_j$ holds, we can directly derive that $(A \wedge \neg A_i \wedge G_j) \subseteq (A \wedge G_1 \wedge G_2)$ holds. This concludes the proof.

$\square$

Next, the relation between directed and saturated contracts are inspected.

Relation between Saturated and Directed Contracts

Originally, the goal was to show that *directed* contracts are also *saturated*, which would have allowed to apply all formulas that are valid for saturated contracts also for directed contracts without any further architectural assumptions mentioned in the previous section. Unfortunately, it turned out that none of the properties implies the other.

For this, let us consider a directed contract $C = (A, G)$ over the interface I. If C is also saturated, its guarantee would include the negated assumption, i.e. $G = \neg A_{\uparrow_{out(I)}} \cup G'_{\uparrow_{in(I)}}$, where G' is the part of the guarantee without the assumption part. Indeed, the part $A_{\uparrow_{out(I)}}$ does not restrict the output ports, but possibly could add non-allowed behavior – with respect to the guarantee – to these ports. Note that this non-allowed behavior is consistent with the semantics of the contracts, as $(\mathit{false} \Rightarrow \mathit{false})$ is true.

To illustrate the above reasoning, counter examples for both implications are given. Note that the following assertions are specified by LTL expressions. Let $I = \{i, o\}$ be an interface, where i is an input port, and o is an output port.

Consider the assumption $A : \square(o = 2)$ and the guarantee $G : \square(o = 2 \Rightarrow i = 0)$. Clearly, since $\neg A \subseteq G$ holds, the contract is saturated. However, it is easy to see that neither A is receptive on the output port, nor G is receptive on the input port.

For the second implication, consider the assumption $A : \square(i = 0)$ and the guarantee $G : \square(o = 2)$. A is receptive on the output port and G is receptive on the input port. Consider a trace satisfying the condition $i = 1 \wedge o = 3$. The trace is contained in the language $\neg A$ but not in G. Thus, the contract is not saturated.

4.4.2. Timed Automaton-based Analysis Approach

In this section a technique to check the refinement and virtual integration condition of contracts as introduced in Subsection 2.4.1 is presented. For this, all contracts of the corresponding system under analysis are translated to UPPAAL timed automata [LPY97], and a reachability check of states violating any properties [ABL98] is performed. This technique is inspired by so called observer-based analyses in literature: Properties are expressed by employing the same formalism as the system. The system is then extended by the resulting component. This new component is also called observer or monitor in literature. Such a component passively observes the relevant behavior of the system and detects faulty behavior with respect to the specified property.

The introduced checking technique was initially introduced in [GWG11] and [GWO14]. For the analysis approach, it is assumed that the specification of a system or a component consists of either a single contract or a set of directed contracts.

From Contracts to Timed Automata Networks

To validate the condition of Theorem 4 trigger and observer automata are derived from the corresponding parts of the contracts of the overall component and all of its subcomponents. For all parts of the contracts appearing in the left part of the implication of the formula of Theorem 4 trigger automata are derived, which produce the behavior as specified in their assertions. All assertions of the right part of the formula serve as passive observer automata, which react to the trigger automata.

Let $C = (A, G)$ be a contract of an overall component and let $C_i = (A_i, G_i)$ for $i \in \{1, ..., n\}$ with $n \in \mathbb{N}_{>0}$ be a set of contracts of the subcomponents. C will be referred to as global contract, and the C_i to as local contracts in the following. To check the condition of Theorem 4, a timed automaton O_{A_i} is derived out of each local assumption A_i for $i \in \{1, ..., n\}$ serving as passive observer. Further, a timed automaton O_G is derived out of the global guarantee G. The transitions of each O_{A_i} for $i \in \{1, ..., n\}$ and O_G are annotated with receiving events and clock constraints in such a way that the observers accept the set of timed traces which are element of $[\![A_i]\!]$ and $[\![G]\!]$. For all traces which are not element of $[\![A_i]\!]$ (or $[\![G]\!]$) the corresponding observer enters a bad state.

Further, an automaton T_A for the global assumption A and an automaton T_i with $i \in \{1, ..., n\}$ for each local guarantee is derived. These automata serve as trigger for the observer automata. The transitions of T_A are annotated with sending events and timing constraints, such that T_A produces all traces that are element of $[\![A]\!]$. The automata for the local guarantees consist of both receiving and sending events (see next section for details and examples) and are called *transceiver automata* in the following.

From all automata the automaton network

$$S = T_A \parallel T_1 \parallel \ldots \parallel T_n \parallel O_{A_1} \parallel \ldots \parallel O_{A_n} \parallel O_G$$

is built.

If one of the trigger automata produces a sequence which is not an element of $[\![A_1 \wedge ... \wedge A_n]\!]$ – and therefore the subset inclusion property of the local assumptions is violated – or not an element of $[\![G]\!]$ – where the subset property of the global guarantee is violated – a corresponding observer will enter a bad state. Then, S has to be checked against the following UPPAAL query, which is a timed computation tree logic (TCTL) formula:

$$R_1 = A\Box(\neg\, O_{A_1}.bad \wedge ... \wedge \neg O_{A_n}.bad \wedge \neg O_G.bad) \quad (4.3)$$

This formula states that a bad state of any of the observer is never reached.

To save some states the bad states and all the transitions to the bad states of the observer automata can also be omitted. Then, instead of checking the reachability of a bad state, it is checked whether the automaton network S is deadlock free:

$$R_2 = A\Box(\neg\ deadlock). \tag{4.4}$$

Further, it has to be verified, whether the transceiver automata are ever triggered, i.e. finally leave their initial states. So, if T is a transceiver automaton and s_0 is its initial state, the following property is checked:

$$R_2 = \ A\Diamond(\neg\ T.s_0). \tag{4.5}$$

This property states that the automaton T has finally reach some states, which are different from its initial state.

If the automaton network satisfies Formula 4.3, i. e. when $S \models R$, it has been shown that the system architecture annotated with the local contracts can be virtually integrated in the context of the system with the global contract. In other words, the set of local contracts refines the global contract.

From RSL to Timed Automata

To enable an automatic virtual integration check for contracts, where the assumptions and guarantees are specified by the usage of RSL patterns as introduced in Subsection 2.4.2, these pattern have to be transformed to a formal model. In this work, these patterns are transformed to timed automata. The resulting automaton network is then checked against the properties specified in the previous subsection.

The *R1*-pattern specifying the periodic occurrence (with some jitter) of an event is typically used as a trigger to the rest of the system. For example, event streams defining the activation of independent tasks can be specified with the aid of this pattern.

Considering a jitter less than or equal to the period timed automata are derived which are illustrated in Figure 4.5: The left automaton specifies the trigger, the center the observer, and the right the observer automaton pattern containing an explicit *bad* state. According to standard event streams as introduced in the real-time calculus [TCN00] the initial event is fired within the interval $[0, period]$, and each successive event is fired every *period* time unit.

Note that there is an event called *on* which is fired from the trigger automaton and consumed by the observer. This event is necessary to synchronize both automata such that the duration in states with respect to the period and the jitter can be distinguished. By firing the event *on* the trigger indicates that it is ready to generate the first event of the represented event stream (after possibly a delay of *jitter* time units). If these automata would be left asynchronous, the observer would not know when the trigger switches to the state *releasing*. After the trigger has switched to its *releasing* state, it

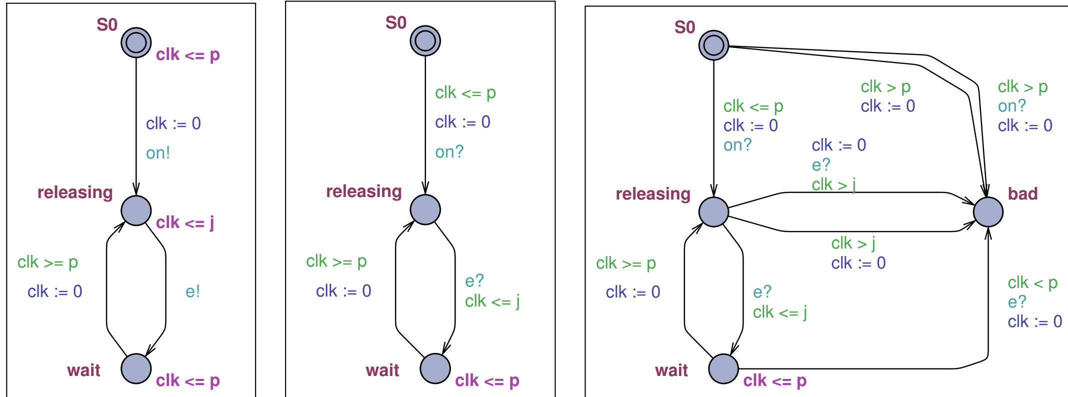


Figure 4.5.: Automata for *R1*-pattern – Left: Trigger automaton; Center: Observer automaton; Right: Observer with explicit bad state.

can immediately take the transition to its state *wait* firing an event *e*. As the observer is still in its initial state, it misses this event. This would even lead to a deadlock, as there does not exist any automata which are able to consume the event *e*.

In contrast to the trigger automaton, the initial state of the observer has no invariant. An invariant enforces the corresponding automaton to leave its state after the defined upper bound of time. As an observer is a passive entity, which shall only react on behavior generated by trigger automata, this should not happen. If the initial state of the observer would have an invariant with an upper bound less than the period, the observer could force the trigger to take the transition to the state *releasing*. An exception of the usage of invariants for observer is given, if the automaton does neither produce nor consume an event on an outgoing transition of a state and shall leave this state in a specific point of time. This is the case for state *wait*.

The right automaton of Figure 4.5 is the extended version of the already explained observer automaton with respect to a bad state. The bad state is entered, whenever the trigger does either fire the event *e* too late with respect to the period (state *S0*) or the jitter (state *releasing*), does not fire the event at all, or does fire the event too early with respect to the period (state *wait*). In those faulty cases, the observer automaton in the center of Figure 4.5 would produce a deadlock as it defines no behavior for such cases.

The edges to the bad state occur twice, i. e. with and without the annotation of the corresponding event. This is because the automaton shall be prevented from getting stuck. For instance if an event is received by the time the jitter exceeds its allowed value and the automaton has not switched to the bad state yet, the edge annotated with the corresponding event to the bad state is fired. From the state *wait* the automaton can switch to the bad state, if the event arrives too early, and from the state *releasing* the

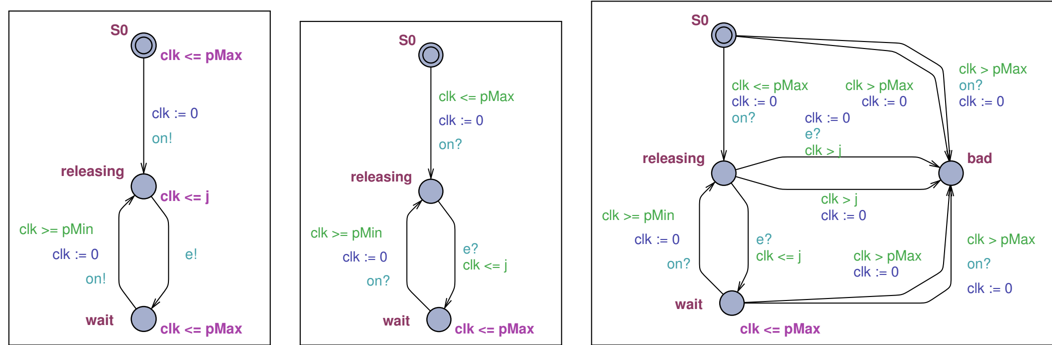


Figure 4.6.: Automata for *R2*-pattern – Left: Trigger automaton; Center: Observer automaton; Right: Observer with explicit bad state.

bad state is reached, when the jitter exceeds its allowed value.

Lemma 10. *The illustrated observer (trigger) automaton accepts (generates) exactly the traces specified by the semantics of the* R1*-pattern in Subsection 2.4.2.*

To proof this, an analysis of the observer is performed in the following.

Proof. The first state has to be left within $[0, p]$ time units. Otherwise the outgoing edge from state *S0* to *releasing* cannot be fired as the clock is not reset in state *S0*. Thus, if the synchronization event *on* (and with this the event *e*) is received too late, the automaton does not accept the trace and deadlocks. In state *releasing* the input event *e* must then be received within $[0, j]$ time units. Thus, the first event may occur within the interval $[0, p+j]$ by switching to the state *wait*. The transition to *releasing* is fired whenever the clock reaches the period value. In the state *releasing* the next event has to occur within the time interval $[0, j]$ after entering this state. Thus, this results in event distances within the interval $[p-j, p+j]$. This exactly corresponds to the defined semantics of the pattern. □

The proof of the automata of all other pattern introduced in the following is performed in a straightforward manner, and thus left out in the following.

The trigger automaton for the sporadic *R2*-pattern is illustrated in the left part of Figure 4.6 and is build similar to the automaton of the *R1*-pattern. It is assumed that the jitter is always less than or equal to the *minPeriod* value. The first event is fired within the interval $[0, maxPeriod]$. Note that in the figure $pMin$ refers to $minPeriod$ and $pMax$ refers to *maxPeriod*. In contrast to the automaton of the *R1*-pattern each successive event is fired within the time interval $[minPeriod, maxPeriod]$, and possibly delayed by some jitter. The *maxPeriod* parameter has always to be defined in order to

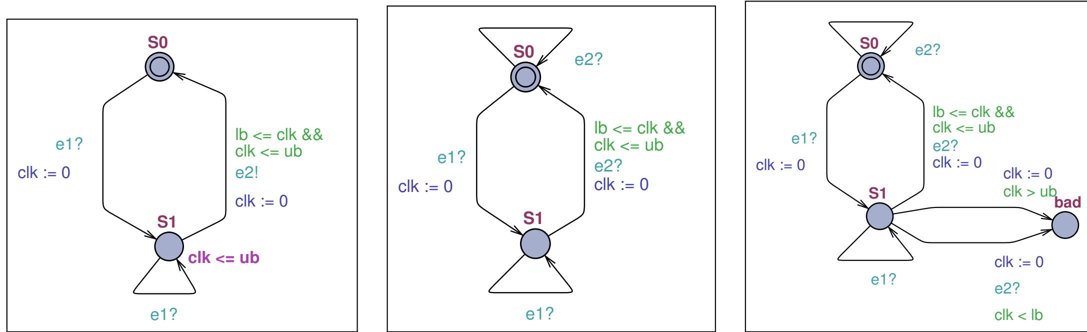


Figure 4.7.: Automata for *R3*-pattern – Left: Transceiver automaton; Center: Observer automaton; Right: Observer with explicit bad state.

have a finite state space. If not, some *liveness* properties would be specified requiring that *finally* an event should be fired. This would require that an event could be sent at arbitrary time in the future. Thus, a verification must wait infinitely long to evaluate such a property. In the sense of timed automata this would mean that the trigger automaton of Figure 4.6 could remain in state S_0 and *wait* forever as S_0 has no invariant which could enforce the automaton to finally leave the state. Thus, one would obtain an unbounded automaton model. To bound this time frame, the $maxPeriod$ parameter has always to be defined, which translates to invariants in the states S_0 and *wait*.

The *R3*-pattern is used to specify a latency caused by, e.g., computations of some components, or when an inputs is needed from the environment, combined with a timeout. The first part of the *R3*-pattern (*whenever X happens*) defines a triggering part. When the event specified in this part of the pattern is received e.g. from the environment or an other contract, the component is responsible to fulfill the second part of the pattern, i.e. to send the response within the specified timing bounds. Thus, in contrast to a purely trigger automaton the *R3*-pattern is translated to a transceiver automaton, which is illustrated in the left part of Figure 4.7: Whenever an event from the environment with respect to the assertion is received, the automaton sends an output event after the delay interval $[lb, ub]$ where $lb, ub \in \mathbb{N}$. For this, the initial state consists of an output transition annotated by the receiving event $e1$. On this edge the clock is reset to appropriately determine the passage of time until the response $e2$ is sent.

In this thesis, only an *iterative* activation of all automata are considered, i. e. simultaneous activations are not considered. Each further activation is ignored. This is realized by equipping state S_1 with a self loop which is annotated by a corresponding receiving event. To deal with multiple activations, explicit indexes must be allocated to the observable events. Also, the number of maximal simultaneous activations needs to be

Figure 4.8.: Transceiver automata for *R3*-pattern with sets of events– Left: Sorted set of events (“*and then*”); Right: Unsorted set of events.

known a priori, such that a corresponding number of automata can be instantiated. This problem is similar to the one of the activation bounds of tasks of the iterative timing analysis addressed in Subsection 3.3.2.

The corresponding observer automaton for the *R3*-pattern is depicted in the center and the right part of Figure 4.7: Whenever an event is received from the environment, the automaton checks whether an event $e2$ is sent as a response. If such an event is sent too early or too late, the automaton in the right part of the Figure 4.7 switches to its bad state, and the one in the center causes a deadlock. The transition $S_1 \rightarrow_{clk<lb} bad$ in the automaton in the right part of the figure is skipped, if the lower bound is set to zero ($lb = 0$), as the constraint $clk < 0$ is never true for a clock. Thus, such a transition is not necessary and can be left out.

The extension of this pattern to multiple events is illustrated in Figure 4.8. Note that only the transceiver automata without an explicit bad state are illustrated. The observer automata and the automata with bad states are constructed analogously. The approach is illustrated for two events respectively. The approach extends in a straightforward manner, if more events are considered. The automata structures get the more unreadable the more events are considered, as demonstrated later.

In the left part of the figure the ordered event set is considered by instantiating the following pattern type:

Whenever $e11$ and then $e12$ occurs, $e21$ and then $e22$ occurs during $[lb, ub]$.

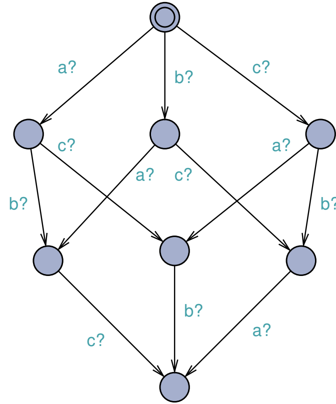


Figure 4.9.: Concept of structure for unsorted set of events within an *R3*-pattern.

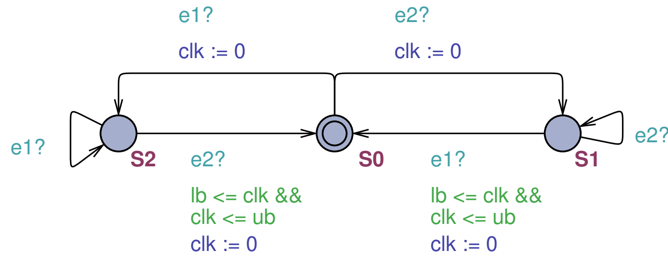


Figure 4.10.: Observer automaton for *R4*-pattern.

In contrast to the automaton of Figure 4.7 intermediate states are introduced. Whenever an event of the left part of the expression is received (here $e11$ and $e12$) the clock is reset on the corresponding outgoing states. The time the last event of the right part of the expression shall be received, the outgoing transition is annotated by the corresponding time interval. All states are equipped by self loops to ignore multiple activations.

In the right part of the Figure 4.8 the resulting transceiver for the pattern with unsorted events is considered. In contrast to the other automaton, the outgoing edges are reproduced in such a manner that all event orderings are considered. The size of the structure gets unreadable for more events. In Figure 4.9 the plain structure for three events of only one part of the pattern (e.g. `Whenever set` $\{a, b, c\}$ `occurs,...`) is illustrated. The size of such a part is determined by 2^n where n is the number of tasks.

The usage of the *R4*-pattern is restricted in such a manner that the pattern only contains either input or output ports, but not both. Without this restriction, the receptiveness property of Definition 25 would be violated. For instance, consider that the

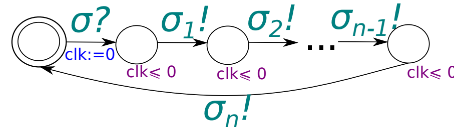


Figure 4.11.: Glue automaton reproducing an event σ for multiple observer automata.

distance between events on an input and an output port is specified within the guarantee part of a contract. If an event on the output port occurs before an event in the input port, the guarantee would restrict the timing of this input. When specifying some delay between input and output ports, one has to use the *R3*-pattern instead.

Because of this restriction, there is only the need of an observer automaton for the *R4*-pattern, which is illustrated in Figure 4.10. The automaton consists of two symmetric parts, which are closely related to the automaton of the *R3*-pattern. The parts reflect the corresponding event ordering. If the event $e2$ is received before the event $e1$, the transition from $S0$ to $S1$ is fired, and the rest operates analogous to the automaton obtained from the *R3*-pattern. The transition from $S0$ to $S2$ reflects the other case, when $e1$ is received before $e2$.

Events which are received by a set of automata instead of only a single one need to be duplicated. This is actually realized by some *glue logic* consisting of small automata that multiply events. Let σ be an event which is received by a set of automata $A_1, ..., A_n$. First, such an event is duplicated n-times and each copy is renamed appropriately. Here the index of the receiving automaton is added as a postfix to the original event name, i.e. $\sigma_1, ..., \sigma_n$. Then, the transitions of the automata are adapted accordingly, i.e. the event σ on the transitions on automaton A_i is replaced by event σ_i. Then, a further automaton, called *glue logic* is added to the automaton network, which receives the original σ event from the triggering automaton, and triggers the duplicated events successively as illustrated in Figure 4.11. The automaton has a clock variable *clk* which is reset whenever the initial state is left. Except of the initial state all states of the glue automaton are annotated with the invariant $clk \leq 0$, which means that no passage of time is allowed in such a state and thus has to be left immediately. With this, all automata receive a copy of event σ in the same time instance. Note that such a glue automaton is created for each event which needs to be duplicated. As no time passes in the intermediate states, the original semantics of the automaton network is not changed.

4.5. Impact Analysis on Implementation Level

During the design process changes affecting the architecture of a system occur, such as adding a new task on an existing resource, the merge of two tasks in a single one,

or the change of the implementation of some components. If such changes occur, already performed analyses have to be repeated, increasing the time needed to verify the functionality and properties of the design, and thus increasing the time to market.

It is required to perform an impact analysis, when changing or maintaining software. With such a technique the designer is able to determine the additional amount of work required to implement a change, i.e. to adapt the context of the changed part of the system. Moreover, an impact analysis helps to identify verifications and test cases which should be re-executed to ensure that the change was implemented correctly [Leh11].

The timing analysis approach introduced in Chapter 3 works in an iterative manner rather than a holistic. Thus, this analysis approach enables to automatically determine whether interfaces of resources are affected through changes. If an output interface STS is recomputed as changes occurred on the corresponding resource, the check introduced in the following subsection is able to determine whether it refines the "old" interface computed in a previous verification step. In such a case a re-verification of resources depending on this interface STS can be omitted.

In the next subsection, the logical flow of the impact analysis approach on the implementation level is introduced. It is demonstrated in which cases a complete re-verification of a component is necessary, a refinement check is performed, and when some of the verification steps can be omitted. Thereafter, the advantages of the presented approach are discussed when further abstraction techniques are applied on the interfaces of resources.

4.5.1. Combining State-based Analysis with a Refinement Checking Technique

The concept of the impact analysis on the implementation level of a system design is illustrated in Figure 4.12 in terms of a UML activity diagram. The activity diagram details and extends the more abstract flow which was introduced in the right branch of Figure 4.4 where the overall impact analysis methodology was illustrated. The approach detailed here was originally published in [GKR15].

Each resource has a status flag for its resource state space called *outputIsConsistent*, which is initially set to *false*. Whenever non-refinement changes of the state space of a resource R occur, this flag is set to *false*. Otherwise, if changes occur leading to a refined state space, the flag is set to *true*. The intention of this flag is to inform resources which depend on R that the corresponding interface STS to R did change in a refined or non-refined manner. With this information the approach is able to determine whether the state spaces of these dependent resources have to be recomputed or can be skipped.

Thus, the approach illustrated in Figure 4.12 starts by checking this property, i.e. whether some inputs of the currently considered resource have changed by calling the method *checkInputStatus*. This method evaluates the input flags *outputIsConsistent* of all resources on which the current resource depends on as mentioned above. Note that for resources on which only independent tasks are allocated, this check is skipped and

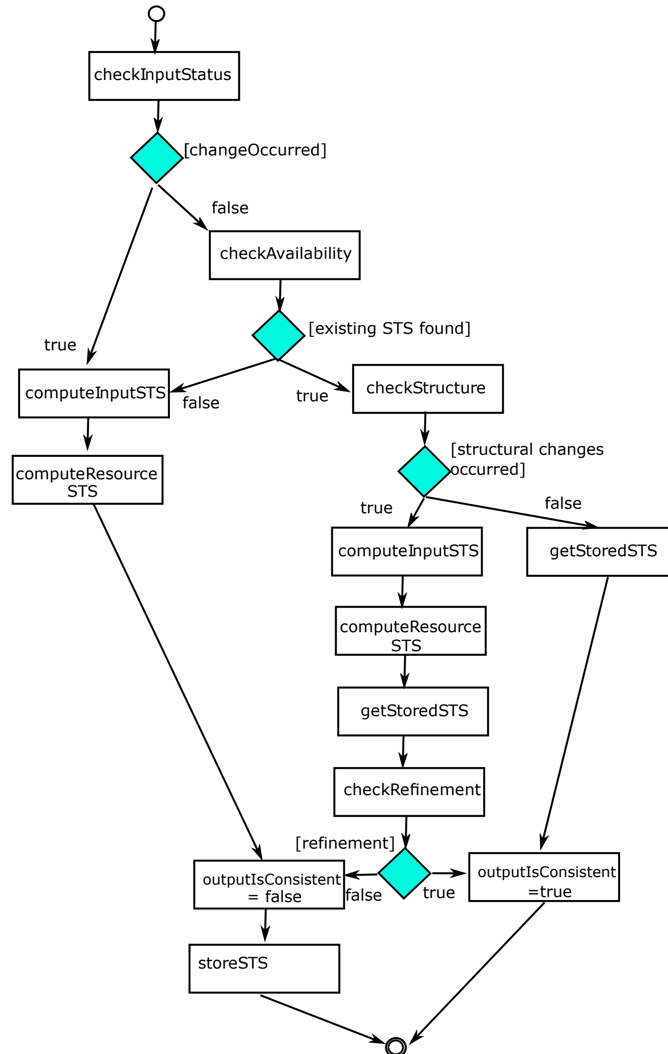


Figure 4.12.: Methodology of the Impact Analysis (timing analysis combined with refinement check).

the right branch is taken. Changes of properties of independent tasks are determined by the method *checkStructure* detailed later. Considering resources with dependent tasks, if changes occurred, *changeOccurred* evaluates to *true*. In this case, the computation of the input STS (*computeInputSTS*) followed by the computation of the resource state space itself (*computeResourceSTS*) is performed as usual. As the resource STS is newly computed, the flag *outputIsConsistent* is set to *false* to signalize dependent resources that this interface has been changed. Last, the resource STS is stored appropriately.

Otherwise, if the output STSs of all resources, from which the current resource depends on, did not change, *checkOccurred* evaluates to *false*. In this case it is checked whether a computed STS of this resource already exists from previous verification steps, in which the resource STS was saved (*checkAvailability*). If not, the left branch is taken and the STSs have to be computed as described above. If there exists an already computed STS, it is checked whether structural changes have occurred (*checkStructure*), i.e. changes concerning the scheduling policy of the resource, the number and type of the allocated tasks, and their properties which are priorities, execution times, and the activation behaviors of independent tasks determined by their period and jitter values. If these properties did not change, the resource STS will also be not affected. Thus, the existing STS is restored by loading it from the file system (*getStoredSTS*). The timing constraints are checked and the flag *outputIsConsistent* is set to *true* which indicates that nothing did change on the output interface.

Otherwise, if some changes on the structure of the resource occurred, *checkStructure* will return *false*. In this case, the input and the resource STSs of the current resource have to be recomputed. Then, the previously computed resource STS has to be loaded, and a refinement check between both resource STSs has to be performed (*checkRefinement*). If the refinement check evaluates to *true*, *outputIsConsistent* is also set to *true* indicating that the resource STS changed in such a manner that all relevant properties are still preserved. Otherwise, the flag is set to *false*. Note that before the recomputation of the resource STS is performed, the input STS of the resource has also to be recomputed because if properties of independent tasks changed, the input STS would also be affected.

To store and load computed state spaces, the corresponding resources must be referenceable. This is realized by allocating unique identifiers to both the architecture and the corresponding resources. The tool then manages a file directory: For each architecture a subdirectory is created, containing subfolders for each resource, in which the computed state spaces and structural aspects are stored. Details for this technique can be found in Appendix A.

After the impact analysis of a resource N has been performed, all dependent resources are triggered to be also verified, if N has set its *outputIsConsistent* flag to false. If not, the next resource which has been changed is considered.

4.5.2. Refinement through Simulation Relation

In this section, a refinement relation between state transition systems will be defined. This refinement basically is based on the timed simulation relation between the states of the corresponding STSs of Definition 22. It will be used to determine, whether some recomputations have to be performed or not. If the input transition system $STS_{in,R}$ of a resource R is changed in such a manner that the new input transition system $STS'_{in,R}$ *refines* the original one, than a recomputation of the STS of R (and all of its successors) can be omitted.

A simulation relation is a rather general concept and not that restrictive like a bisimulation relation. Because of this, more STSs can be found which are in simulation relation than in bisimulation relation. With this, the potential to omit re-verifications is much higher. Of course, a simulation relation does not preserve all characteristics of the original STS, but for our analysis the relevant properties are preserved as shown in the following.

Definition 27 (STS Refinement)**.** *Given two symbolic transition systems* $STS_1 = (S_1, S_1^0, C, \Sigma, \rightarrow_1)$ *and* $STS_2 = (S_2, S_2^0, C, \Sigma, \rightarrow_2)$ *over the same clock set* C *and alphabet* Σ. STS_1 refines STS_2, *if it* simulates *it, in short* $STS_1 \precsim STS_2$. STS_1 *simulates* STS_2, *if a simulation relation* $r \subseteq S_1 \times S_2$ *according to Definition 22 exists such that* $(S_1^0, S_2^0) \in r$.

For the timing analysis, we are interested in properties concerning upper bounds of clocks. A transition system STS over a clock set C satisfies a timing constraint $\varphi \in \Phi(C)$, in short $STS \models \varphi$, if and only if for all reachable states $\langle l, D\rangle$ of STS it holds that $D \subseteq D_\varphi$. The simulation relation preserves these properties:

Theorem 5. *Let* $\varphi \in \Phi(C)$ *be a clock constraint, and* STS_1, STS_2 *be transition systems with* $STS_2 \precsim STS_1$. *It holds that if* $STS_1 \models \varphi$ *then* $STS_2 \models \varphi$.

The proof of this theorem is given through the transitivity of the subset relation of zones: when for all $\langle l_1, D_1\rangle \in S_1$ $D_1 \subseteq D_\varphi$ holds, and for all $\langle l_2, D_2\rangle \in S_2$ a $\langle l_1, D_1\rangle \in S_1$ exists such that $(\langle l_2, D_2\rangle, \langle l_1, D_1\rangle) \in r$, then also $D_2 \subseteq D_\varphi$ holds.

4.5.3. Combining Impact Analysis with Abstractions

Generally an impact analysis is useful in combination with analysis techniques that involve abstractions. This is also a typical scenario for analytic techniques such as in [RRE03]. These techniques are based on the assumption that every interface behavior can be characterized by event streams. To obtain event streams for the outputs of a resource, the actual task behavior is generally over-approximated.

Hence, changes on the behavior of a particular resource might indeed have an impact on the already computed *exact* state space representing its output behavior, but might not have an impact on the over-approximated output behavior of the resource. This can be exploited by our impact analysis. This is because over-approximated behaviors accept more traces than the original behavior. Let M_R correspond to the original exact output behavior of a resource R, and let $M'_R := \alpha(M_R)$ be an over-approximated behavior resulted by applying an abstraction function α on M_R, such that $M_R \precsim M'_R$ holds. If now some changes on R are preformed, it is more likely that the newly recomputed behavior M''_R of the resource refines the abstract behavior M'_R than the more restrictive original behavior M_R. Thus, if M''_R does not refine M_R, but $M''_R \precsim M'_R$ does holds, computations of dependent resources can again be omitted. Considering more abstract behaviors on interfaces can thus boost the minimization of re-verification needs.

A_1	(A_{11}) *triggerLineDetect* `occurs each` $20ms$
	(A_{12}) *sensorHMI* `occurs each` $20ms$
G_1	`Whenever` set$\{triggerLineDetect, sensorHMI\}$ `occurs`, *steer_ctrl_out* `occurs during` $[0, 100]ms$

Table 4.1.: Contract $C_1 = (A_1, G_1)$ of the driver assistance system.

A_2	(A_{21}) *speed_in* `occurs each` $20ms$
	(A_{22}) *distance* `occurs each` $20ms$
G_2	`Whenever` set$\{speed_in, distance\}$ `occurs`, *speed_ctrl_out* `occurs during` $[0, 100]ms$

Table 4.2.: Contract $C_2 = (A_2, G_2)$ of the driver assistance system.

Nevertheless, the degree of abstraction is also crucial for dependent resources such that suitable verification results can be obtained. Event streams are considered as the maximal abstraction of the timing behavior of a task, as these only contain information about best- and worst-case response-times without any information in which cases the corresponding response times occurs. For instance, a task could have a large response time when it is interrupted by a high priority task which is allocated on the same resource, and a small response time when no interrupts occur. This abstraction is ideal for impact analysis approaches, but it leads to very pessimistic timing results as also illustrated in previous sections.

An abstraction indeed might affect the schedulability of a depending resource and hence may cause false negative results. On the other hand, suitable abstraction techniques may pave the way to omit re-verifications.

In Subsection 4.6.3 the impact analysis approach will be adapted in such a case that it is combined with an abstraction technique in order to demonstrate the potential of this approach.

4.6. Evaluation and Case Studies

4.6.1. Contract-Level of the Driver Assistance System

The impact analysis approach is evaluated by applying it to the driver assistance system (DAS), which consists of the lane-keeping-support (LKS) system already introduced in Section 3.7 accompanied by an adaptive cruise control system.

Two contracts $C_1 = (A_{11} \wedge A_{12}, G_1)$ and $C_2 = (A_{21} \wedge A_{22}, G_2)$ are annotated to the overall driver assistance system, to which its implementation has to adhere. Both contracts

A_{ACC}	*speed_in* `occurs each` $20ms$ `with jitter` $2ms$
G_{ACC}	`Whenever` set{*speed_in*, *distance*} `occurs,` *speed_ctrl* `occurs during` $[0, 50]ms$

Table 4.3.: Contract $C_{ACC} = (A_{ACC}, G_{ACC})$ of the adaptive cruise control subsystem.

are defined in the Tables 4.1 and 4.2 respectively. The assumptions of both contacts specify the trigger times for the input events *triggerLineDetect*, *sensorHMI*, *speed_in* and *distance*, which are $20ms$ for all events. For this, the $R1-$Pattern is used.

The guarantee parts of both contracts specify end-to-end latency constraints by the usage of the $R3-$Pattern:

- The guarantee part of the C_1 contract specifies that whenever both trigger events *triggerLineDetect* and *sensorHMI* occur (in an arbitrary order), the steer control signal *steer_ctrl_out* determining the angle of the front wheels of the car has to be computed within the time interval of $[0, 100]ms$. Both input signals are needed to compute an appropriate angle.

- The guarantee part of the C_2 contract specifies that whenever both sensor inputs *speed_in* and *distance* occur, the control signal *speed_ctrl_out* which is responsible for the speed of the car has to be computed within the time interval of $[0, 100]ms$ and the corresponding value has to be available at the output port of the component.

Next, the contracts of the subsystems ACC, LKS, and VDS will be detailed.

In Table 4.3 the contract for the adaptive cruise control subsystem is illustrated. The assumption A_{ACC} specifies the occurrence of the trigger event *speed_in*. It relaxes the assumption A_{21} of system contract C_2 by adding an occurrence jitter of $2ms$. The assumption of the occurrence of the *distance* event is the same as specified by the assumption A_{22}, and thus not repeated in Table 4.3.

The guarantee G_{ACC} specifies an end-to-end latency constraint between the input triggers of *speed_in* and *distance*, and the computed set point of the speed of the car at the output port *speed_ctrl*. Whenever both trigger occur, the output shall be available within the time interval of $[0, 50]ms$.

The guarantee part of the contract of the lane-keeping-support system (LKS) is illustrated in Table 4.4. The assumptions concerning the occurrence of the events *sensorHMI* and *triggerLineDetect* of the contract annotated to the LKS subsystem are the same as specified by the assumptions A_{11} and A_{12}. The guarantee part specifies that whenever the trigger *triggerLineDetect* occurs, the output *trajectoryData* has to be delivered by the implementation of the LKS within the time interval of $[0, 50]ms$.

A_{LKS}	($A_{LKS,1}$) $triggerLineDetect$ `occurs each` $20ms$
	($A_{LKS,2}$) $sensorHMI$ `occurs each` $20ms$
G_{LKS}	`Whenever` $triggerLineDetect$ `occurs`, $trajectoryData$ `occurs during` $[0, 50]ms$

Table 4.4.: Contract C_{LKS} of the lane-keeping-support subsystem.

A_{VDM}	($A_{VDM,1}$) $yawRateIn$ `occurs each` $20ms$
	($A_{VDM,2}$) `Distance between set{`$trajectoryData$, $speed_ctrl$, $yawRateIn$`} within` $[0, 70]ms$
G_{VDM}	`Whenever` set{$trajectoryData$, $speed_ctrl$, $yawRateIn$} `occurs`, set{$steer_ctrl_out$, $speed_ctrl_out$} `occurs during` $[0, 20]ms$

Table 4.5.: Contract $C_{VDM} = (A_{VDM,1} \wedge A_{VDM,2}, G_{VDM})$ of the vehicle dynamic management system.

The contract of the vehicle dynamic management system (VDS) is illustrated in Table 4.5. The assumption consists of two parts: $A_{VDM,1}$ specifies the triggering condition of the input event $yawRateIn$ by the usage of the $R1-$Pattern, which is $20ms$. The second part $A_{VDM,2}$ specifies the allowed timing distance of all input events $speed_ctrl$, $trajectoryData$, and $yawRateIn$ by the usage of the $R4-$Pattern. It specifies that the timing distance of all events is $70ms$ at most, i.e. whenever one of the input events is received, the other input events have to be received within the interval of $[0, 70]ms$.

The guarantee of C_{VDM} specifies the computation latency of the VDS: Whenever all input trigger $speed_ctrl$, $trajectoryData$, and $yawRateIn$ occur, the control values on the output ports $steer_ctrl_out$ and $speed_ctrl_out$ have to be computed within the time interval of $[0, 20]ms$.

For all these contracts the corresponding timed automata were generated by the implementation of our concept. In Appendix C a subset of the generated automata are illustrated. For the generated automaton network the reachability of a deadlock was checked by the usage of the UPPAAL model checker. First the local assumptions are verified. In a second step the global guarantees are verified. This is a valid split, as the virtual integration condition also

According to the simplified virtual integration condition, to check whether the local assumptions A_{ACC} and $A_{VDM,2}$ do hold, the following expression needs to be evaluated:

$$A_{11} \wedge A_{12} \wedge A_{21} \wedge A_{22} \wedge A_{VDM,1} \wedge G_{ACC} \wedge G_{LKS} \Rightarrow A_{ACC} \wedge A_{VDM,2} \quad (4.6)$$

Note that the assumption specifying the trigger of the yaw rate of assumption $A_{VDM,1}$ is left out in the right part of this implication expression. This is because the overall

A'_{LKS}	($A'_{LKS,1}$) $sensorHMI$ `occurs each` $20ms$ with jitter $2ms$
	($A_{LKS,2}$) $triggerLineDetect$ `occurs each` $20ms$
G'_{LKS}	`Whenever` $triggerLineDetect$ `occurs,` $trajectoryData$ `occurs during` $[0, 48]ms$

Table 4.6.: Adapted contract C_{LKS} of the lane-keeping-support system.

system has the identical assumption to its corresponding input port $yawRateIn$. Thus, to keep the description short, the $A_{VDM,1}$ is considered as being the overall system assumption on the yaw rate and added to the left part of this implication relation.

The corresponding automata of the expressions A_{11}, A_{12} A_{21}, A_{22} and $A_{VDM,1}$ serve as triggering automata as described in the previous section, while the automata of G_{ACC} and G_{LKS} serve as both observer and trigger. The automata of A_{ACC} and $A_{VDM,2}$ are pure observer automata. The verification engine for this sub-property took approximately an hour, while exploring 1.206.322 states, with the result that the property holds, i.e. no deadlock occurs and thus both local assumptions do hold.

After the verification of the local assumptions, the end-to-end latency constraints specified in the guarantees G_1 and G_2 were analyzed successively by the following sub-expressions:

$$A_{11} \wedge A_{12} \wedge A_{21} \wedge A_{22} \wedge A_{VDM,1} \wedge G_{ACC} \wedge G_{LKS} \wedge G_{VDM} \Rightarrow G_i,\ i \in \{1, 2\} \qquad (4.7)$$

The verification of the guarantees took more than one day, while the engine explored 15.312.033 states respectively. This is because the state space of the automaton G_{VDM} which produces events for the guarantee observer is really large. The verification results that both properties do hold.

4.6.2. Impact Analysis on the Driver Assistance System

To apply the impact analysis approach, two change scenarios have been worked out. First, the assumption of the triggering condition for the input $sensorHMI$ of the contract of the LKS is changed by adding a timing jitter as specified in Table 4.6. In a second step, the guarantee is restricted such that the computed result needs to be available at the output port within the time interval of $[0, 48]ms$. These changes are performed successively, i.e. first the change of the assumption is applied and the re-verifications are performed, and then in a second step the guarantee is changed and again the verification steps are called.

For both change scenarios, on specification level the approach checks whether the new contract refines the old contract. Thus, the impact approach first verifies that $A_{LKS} \Rightarrow A'_{LKS}$ holds. In a second step, the expression $A_{LKS} \wedge G'_{LKS} \Rightarrow G_{LKS}$ is

evaluated. The verification of both properties took only a couple of seconds. For the first property 26 states were explored. To verify the second property 24 states were explored. As on the specification level the new contract refines the old one, there is no need to repeat the full verification of the virtual integration condition. This saves much verification time like demonstrated in the results above. However, the verification on implementation level has to be repeated in both cases, which is illustrated in the following.

First, consider the change of the assumption: The verification starts by the computation of the resource STS for the *LaneDetection*. The *checkStructure* step determines that the structural aspects of this resource did not change such that the recomputation of this resource STS is omitted. The original state space is reconstructed by the *getStoredSTS* step. In the next step, the *CANBUS* is considered. The input status of the input STS of resource *LaneDetection* is evaluated to true and thus it is correctly determined that the interface from *LaneDetection* to *CANBUS* has indeed not changed. But the next step (*checkStructure*) determines that the trigger condition of the input *sensorHMI* has been changed, which leads to a re-verification of the input of the resource *CANBUS*. Thereafter, the resource STS itself is recomputed. After this step, the refinement check between both STSs is performed which determines that the interface to the resource *SituationEvaluation* is affected in a non-refined manner. As a result, also the STS of *SituationEvaluation* is recomputed.

By changing the guarantee the approach does not recompute any STS, as the *checkStructure* step determines that no changes were performed on the resources *LaneDetect CANBUS* and thus, the input of resource *SituationEvaluation* is determined to be not affected. The STS of the resource *SituationEvaluation* is loaded and the new guarantee is checked by iterating through the state space. Note that to compute this state space 14*sec.* were originally needed, while the time needed to load the STS took less than one second. This approach can be further optimized by also storing all computed response times of all resources. Then the loading process and the iteration through the STS could also be omitted.

In the following, the saved amount of re-verification times on the implementation level will be further analyzed by the usage of more dedicated examples.

4.6.3. Evaluation of Refinement on Implementation-Level

In this subsection, the methodology is evaluated by the usage of the three test systems illustrated in Figure 4.13. Tasks with no input edge are considered as to be independent, i.e. triggered by event streams. The scheduling policies of each ECU is fixed priority with interruption, and the policy of the CAN bus is also fixed priority but (of course) without interruption. The parameters of the tasks are detailed in the table of Figure 4.14, where p is the period of a task, *exec.* is the execution time which may be a single value or an interval, if $bcet \neq wcet$, and pr is the priority of a task.

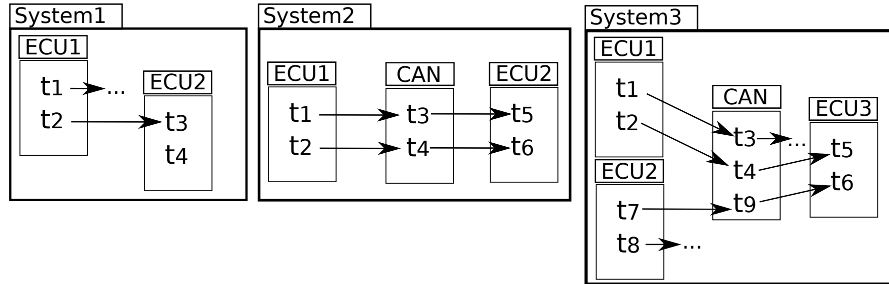


Figure 4.13.: Test systems.

	System 1			System 2			System 3		
	p	*exec.*	*pr*	*p*	*exec.*	*pr*	*p*	*exec.*	*pr*
t1	60	35	1	55	[4,10]	1	50	[4,10]	1
t2	5	2	2	50	[3,7]	2	50	[3,7]	2
t3	-	4	2	-	[2,6]	2	-	[2,6]	2
t4	60	12	1	-	[4,5]	1	-	[4,5]	1
t5				-	[3,5]	2	-	[3,5]	2
t6					[2,3]	1	-	[2,3]	1
t7							50	[3,5]	1
t8							50	[3,5]	2
t9							-	[4,5]	3

Figure 4.14.: Task parameters.

In the scope of the evaluation the execution times needed for an analysis of each resource are compared against the sum of the execution times needed to store and load a corresponding state space and check the refinement of the state spaces of the resources. The idea is to demonstrate that the analyses times of resources are in general much larger than the execution times needed to store and to load the state spaces, and to check whether the loaded state spaces are a refinement of newly computed ones in cases where changes occurred.

Note that all execution times were measured on the same machine to preserve comparability. Each check has been performed five times. The execution times illustrated here are obtained by computing the average times of all measurements.

The measured execution times are illustrated in the table of Figure 4.15. As an example: To built the state space and analyze the timings of $ECU2$ of $System2$ the iterative timing analysis implementation needed $6,75$ seconds. In contrast to this, the refinement check of the existing state space and the newly computed one of $ECU2$ took only $0,015$ seconds. The cell *Sum* is the sum of the cells *Refinement*, *Load* and *Store* and is used to compare the execution times needed to perform these three steps against the plain

		ECU1	ECU2	CAN	ECU3
System 1	Analysis	0,16	6,33		
	Refinement	0,00016	0,0138		
	Load	0,06	1,29		
	Save	0,04	1,4		
	Sum	0,10016	2,7038		
System 2	Analysis	0,09	6,75	0,58	
	Refinement	0,0001	0,015	0,013	
	Load	0,04	0,92	0,17	
	Save	0,03	1,07	0,17	
	Sum	0,0701	2,005	0,353	
System 3	Analysis	0,01	0,03	15,31	77,52
	Refinement	0,0001	0,0001	0,12	0,31
	Load	0,0001	0,009	3,61	10,81
	Save	0,0001	0,0001	3,78	11,4
	Sum	0,0003	0,0092	7,51	22,52

Figure 4.15.: Measured average computation times.

verification time.

In Figure 4.16 the relations between the analysis times for three examples are visualized. The result of the evaluation is that the larger the state space of a resource is and therefore needed time for verification of that resource, the larger the difference will be between the verification time and the computation times needed to load, save, and check the refinement of the old and new state spaces. Thus, for larger systems our refinement methodology is a real gain for our analysis approach. Note that of course, if the refinement check fails, i.e. the new state space of a resource is not a refinement, just performing the plain verification without the refinement check would have run faster. But fortunately, the computation times needed to perform the refinement check are not that large. Actually, the complexity of the refinement check is polynomial, i.e. $n(n-1)$ where n is the number of states: All states have to be compared with each other, while comparing a state with itself can be skipped.

Evaluation of Refinement on Implementation-Level combined with Abstraction

In this subsection we will demonstrate the potential of our impact analysis concept on the implementation level by combining it with an abstraction technique. Note that this part is not a fully worked out and implemented concept, as it shall only demonstrate the potential of the approach and motivate for further research in this area.

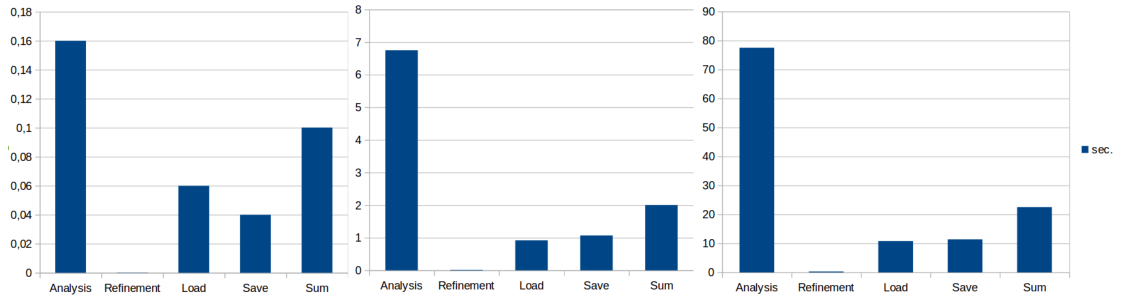


Figure 4.16.: Visualization of some computation times from Table 4.15: a) $ECU1$ in $System1$ (left), b) $ECU2$ on $System2$ (center), and c) $ECU3$ on $System3$.

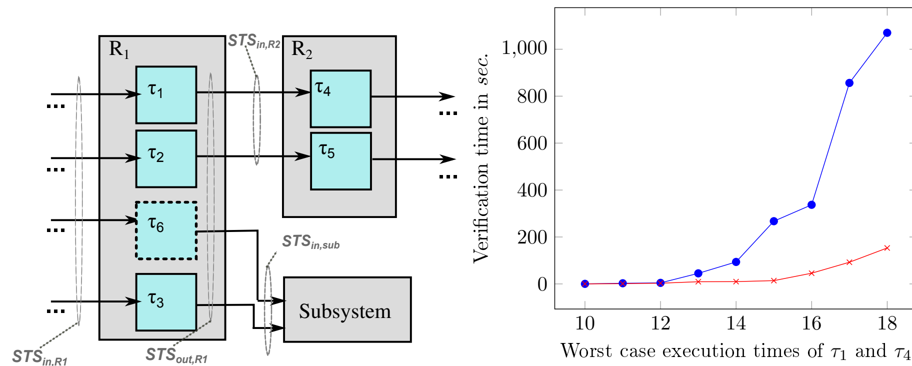


Figure 4.17.: Left: Adapted architecture; Right: Comparison of a re-computation of the STS of R_2 (blue curve, marked with dots) and a refinement check on the output of R_1 (red curve, marked with x).

Consider the architecture in Figure 4.17 consists of two resources R_1 and R_2, and an unspecified subsystem. Initially, three tasks are allocated on resource R_1, namely τ_1, τ_2, τ_3, whereas on R_2 the two tasks τ_4 and τ_5 are allocated. Task τ_4 depends on the output of τ_1, and τ_5 depends on τ_2.

The task τ_6, which is depicted in dashed lines, is added in a later design step. This task has a low priority and does not affect the input behavior of R_2.

After adding τ_6 to R_1 the original concept would proceed as follows: After the computation of the input of R_1 the state space of R_1 (in the following referred to as STS_{R1}) would be re-computed. Then, the refinement relation between the newly computed STS_{R1} and the previous one would be performed. This $checkRefinement$ method would return *false* by setting the *outputConsistent* flag appropriately indicating all dependent resources that its input and therefore its state space have to be re-computed.

As the priority of the added task is lower than τ_1 and τ_2 both tasks are not affected by the change and thus, the input behavior of R_2 actually does not change. We now slightly adapt the introduced concept of the impact analysis on the implementation level (cf. Figure 4.12) as follows: The refinement check is not performed on the state space of R_1 but on the input state space of R_2. As detailed in Chapter 3 this input state space is obtained by abstracting STS_{R1} from tasks which are not relevant for R_2. The refinement check on this newly computed input state space and the previous one (which has of course to be stored instead of R_1 itself) would return *true*, thus preventing the state space of R_2 from a re-computation.

To demonstrate the savings of the verification times, consider the following specification of the scenario in the left part of Figure 4.17 as follows:

- $p_{\tau_1} = p_{\tau_3} = 30,\ p_{\tau_2} = 10.$
- $bcet_{\tau_1} = bcet_{\tau_4} = 10,\ wcet_{\tau_1} = wcet_{\tau_4} \in \{10, ..., 18\},\ c_{\tau_2} = c_{\tau_5} = 2, c_{\tau_3} = c_{\tau_6} = 1.$
- $pr_{\tau_1} > pr_{\tau_2} > pr_{\tau_6} > pr_{\tau_3}, pr_{\tau_3} > pr_{\tau_4}.$

We compare the time needed to re-compute the state space of resource R_2 when τ_6 is added to R_1 with the time needed to check the refinement relation of the input of resource R_2 after the change occurs. In our scenario, we incrementally increase the worst case computation time of both τ_1 and τ_4 simultaneously. A larger *wcet* will increase the state spaces of the resources, thus leading to larger computation times.

In the right part of Figure 4.17 the computation times for both the re-verification of resource R_2 (blue curve with the dot-marks) and the check of the refinement relation (red cure with the x-marks) are illustrated.

As a result we observe that, besides the shorter computation times of the refinement relation, the refinement relation check scales much better for larger state spaces. By combining the refinement check with an abstraction technique (in this case the minimization operation on the interfaces of resources) further analysis time can be saved.

4.7. Summary

In this chapter the extension of the state-based timing analysis approach by an impact analysis approach was illustrated. The impact analysis operates on two different levels of a system design, i.e. on the specification level and on the implementation level. On specification level a new *virtual integration checking* technique through the concept of reachability analysis of timed automata was introduced. If our system specification is *complete* this technique allows us to determine the impact of changes which may occur during the development process of a system without the consideration of any implementation detail. To enable this technique, the original virtual integration condition was adapted by considering a certain class of contracts which are called *directed*.

The impact analysis approach is also defined on the implementation level, which is used when incomplete specifications are considered, or when the implementation of a component changes. To this end, an appropriate refinement relation between state transition systems has been defined. With this notion of refinement, it is able to avoid time consuming re-verifications of parts of the system on implementation level. The implementation of the impact analysis approach was illustrated. The approach was evaluated by measuring the computation times needed to perform the full verification, the storage and load of state spaces, and the computation of the refinement check. A comparison of these execution times was performed. The result is that the larger a system is and the less number of components are affected in a non-refinement manner, the more execution times can be saved needed to perform re-verifications.

These two concepts – i.e. the impact analysis on specification and implementation level – are typically exploited in combination. Changing a component implicates the integration of this component into its context, and the verification of its implementation to the new components local specification. The methodology combining all techniques into one single impact analysis approach was also illustrated in this chapter.

5. Summary and Outlook

In the first chapter the addressed problems in this thesis in the context of hard real-time, safety-critical systems were illustrated.

In Chapter 2 the necessary fundamentals of model-based design of real-time systems and the underlying execution semantics were introduced. The functionality of such systems is given by tasks which are characterized by best- and worst-case execution times. A scheduler determines the order of task executions, which are activated periodically. Requirements of the underlying system architecture are specified by the usage of contracts with which assumed and guaranteed behavior of a component can be distinguished. The parts of the contracts are specified by using the pattern-based Requirement Specification Language. The real-time pattern of this language were described and the formal semantics were given. Proof obligations of correctness checks of decomposition structures were illustrated, which include the composition and refinement check on the level of contracts.

In Chapter 3 a new approach to analyze timing constraints of systems consisting of several computation units and bus systems was introduced. Analogously to model checking, the approach determines the reachability of states of resources, where timing constraints of tasks or end-to-end latency constraints are violated. In contrast to such related works, the presented approach works in an iterative fashion, which enables minimization and abstraction operations on the interfaces of dependent resources. The approach is based on the UPPAAL DBM library. This library implements the finite clock zone representations of valuations of real-valued variables. Using such variables, computation times of tasks are measured. On top of the basic operations of the DBM library, in Section 3.4 higher level functions were defined and implemented. In detail, in Subsection 3.4.1 abstraction operations on zones and locations were defined to determine the minimal interface between two dependent resources. It was shown that the obtained abstracted interface STS is an over-approximation of the original STS and preserves deadlock-freeness. In Subsection 3.4.2 the composition of a set of STSs was presented. It was shown in Theorem 2 that the STS which is obtained by this composition operation has an equivalent behavior to the one which is obtained by first composing a set of timed automata and then building the resulting STS. The main algorithm building iteratively the state space for a given architecture using these operations was presented in Section 3.5.

Further, abstraction techniques were introduced which allow to abstract specific parts of the computed state spaces of computation resources. In detail, in Subsection 3.5.5 the *untimed bisimulation, timed simulation* relation was introduced, where paths are merged

which have on the one hand equivalent discrete behaviors and on the other hand have a related timing behavior in terms of subset relations of clock zones. The application of this technique did not result in much smaller state spaces of dependent resources, but did positively affect the overall analysis times (cf. Figure 3.13). An abstraction technique using a new timed simulation relation was introduced in Subsection 3.6.3 leading to over-approximated STSs. The effects of such over-approximations on the iterative analysis approach were analyzed in Subsection 3.6.4 revealing an interesting effect: Although the state spaces of the interfaces could be reduced for the applied test cases, the resulting numbers of states of dependent resources were always above the numbers of states where the abstractions on the input STSs were not applied. In Subsection 3.6.6 an abstraction technique for interfaces where event bursts occur was presented, leading to significant state space reductions. The usage of this abstraction thereby is restricted to scenarios, where end-to-end latencies are not of interest.

The applicability of the state-based approach was studied on a driver assistance system case study in Section 3.7, which was modeled by the usage of the modeling framework MARTE. The focus on the timing analysis was on the detailed subsystem which is responsible for an automatic lane-keeping of a vehicle. Response times and state spaces of the basic approach and the results of the applications of the abstraction techniques were compared and analyzed. The positive effects on the sizes of the number of states could been demonstrated.

In Chapter 4 the extension of the state-based timing analysis by an impact analysis was illustrated. The overall methodology was illustrated in Section 4.3. The impact analysis operates on two different levels of a system design, i.e. on the specification level and on the implementation level. On specification level a new *virtual integration checking* technique through the concept of reachability analysis of timed automata was introduced in Subsection 4.4.2 deriving timed automaton networks from sets of contracts which are specified using the *Requirement Specification Language* (RSL). To enable such a timed automaton approach, Theorem 4 was derived. This theorem uses the *directedness* and *receptiveness* properties of contracts, resulting in a simplified virtual integration condition, where expressions with negations do not occur.

On the implementation level an appropriate refinement relation between state transition systems was defined in Subsection 4.5.2. With this notion of refinement, it is able to avoid time consuming re-verifications of parts of the system on implementation level. The implementation of the impact analysis approach was illustrated. The approach was evaluated by measuring the computation times needed to perform the full satisfaction verification on the one hand, and the sum of the storage and load of state spaces and the computation of the refinement check on the other hand. A comparison of these execution times was performed, with the result that for larger systems the refinement methodology can help to save much analysis times and thus to help to decrease the time to market of a system.

There are many research areas for future works. This work assumes a dense time semantics of the timing behavior of tasks. The iterative analysis approach can be combined by considering also the discrete semantics. Parts of an architecture, which have been modeled in more detail and are closer to the final implementation then the rest of the architecture may be analyzed by the usage of a discrete time back-end, while the rest is analyzed by considering the dense time back-end.

Future work in this area could also investigate new abstraction techniques which could further boost the scalability of the approach. The explicit computation of the state spaces of resources in an iterative manner enables the analysis of dedicated abstraction operations, e.g. by merging successive states or leaving out some traces. An interesting direction is to find those paths of the state spaces of resources, which yield end-to-end worst-case behaviors. Analogously to classical schedulability analyses where the critical instance leads to worst-case timing behavior, such instance could be determined for distributed systems.

Considering the impact analysis approach, so far only the *iterative* activation of timed automata was considered, i. e. simultaneous activations are not considered and ignored. To deal with multiple simultaneous activations, explicit indexes must be allocated to observable events as illustrated in this work.

In this thesis, pattern considering the real-time behavior were considered. Thus, the approach could be extended by considering further aspects of systems like functional and safety properties specified by text pattern. This would realize a powerful multi-aspect analysis approach.

On the implementation level, alternative refinement relation definitions could be investigated with which a greater set of refining STSs could be determined. Also, the effects of first abstracting a resource STS before performing a refinement check - such it was illustrated in Subsection 4.6.3 - could be analyzed, especially for new abstraction techniques worked out in future researches.

A. Tool Support

The implementation of our methodology consists of three tools: the model converter, the state-based analysis tool, and the contract verification tool.

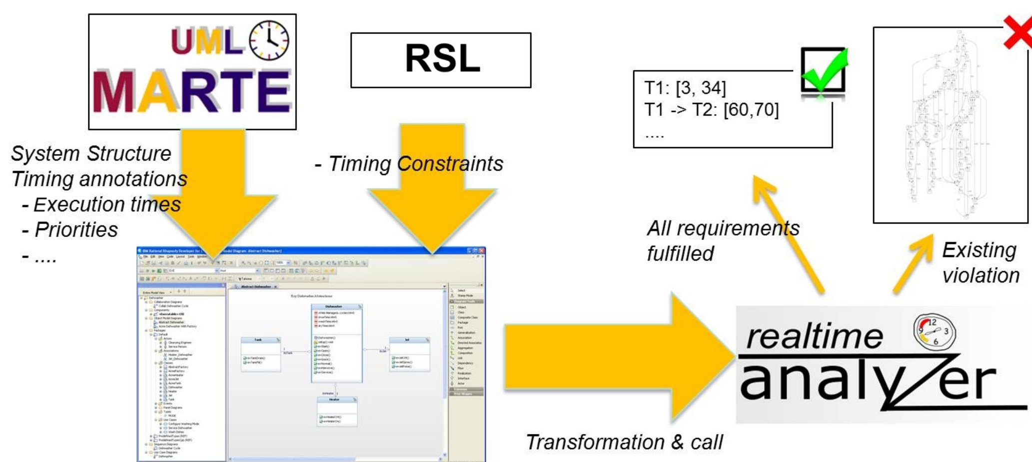


Figure A.1.: Modeling in MARTE and enabling the connection to the analysis approach.

Our approach is illustrated in Figure A.1. To create MARTE models the tool Papyrus was used, which is provided as an Eclipse plugin. After the specification of a system model, the relevant model information has to be extracted and transformed to the input format of our analysis verification back-end *realtimeAnalyzer*. For this, the Eclipse Modeling Framework (EMF) is used. With EMF, we are able to access all model data through Java classes, and generate the necessary data structure for the verification back-end. Thereby, the process is straightforward: We access the model elements in a breadth first search manner and generate the *realtimeAnalyzer* model successively. We will not go into further detail here, as the implementation involves only technical issues. For detailed information to the EMF please refer to [SBPM09]. Support for the transformation of relevant Papyrus model information to the input format of *realtimeAnalyzer* was implemented in the bachelor's thesis of [Fif14].

The state-based timing analysis tool *realtimeAnalyzer* is implemented in C++ and is available for Linux operating systems. The UPPAAL DBM library [1] is used as the basis

[1] http://people.cs.aau.dk/ adavid/UDBM/

of our approach, i.e. we make use of the clock zone implementation with all necessary basic operations such as zone intersection or reset of dedicated clocks.

To ensure the quality of the analysis results, all over the implementation code assertions were included to check basic constraints and to prevent the tool of a faulty usage. Whenever an assertion is violated, the program will automatically terminate with an appropriate error message. To parse the input files, an implementation based on the Spirit parser library [2] was generated. Thereby, Spirit is part of the Boost-C++ libraries, thus is needed to compile *realtimeAnalyzer*.

```
tayfun@ultra5:~/workspaceC/RTANA/build$ ./rtanalyzer --help

Usage of realtimeAnalyzer: ./rtanax [options] <inputSystem>

  rtAnalyzer will generate the following files:
   - counterExample.txt: If some deadlines are not met, RTANA will give a
                         counter example leading to failure state.
   - responseTimes.txt:  Prints the response times of all tasks in this file.
   - <name>STS<number>:  In case you called RTANA with option '-p' the graphs
                         of all CS will be printed in a file with a
                         corresponding name.

  Valid options are the following:
   -b, --burst          : Use the burst abstraction technique.
   -c, --criterion      : Criterion for counter-example generation.
                                 0: Shortest counter-example
                                 1: Counter-example with maximum lateness
                                 2: Counter-example with minimum lateness
                                 3: Arbitrary counter-example
   -e                   : Generate NO counter-example (RTANA will work faster
                          in case of a deadline violation).
   -h, --hp             : Abstract all informations about hp tasks.
   -i                   : Deactivate abstraction on initial non-determinism
                          of highest priority tasks (phasing abstraction).
   -m                   : Perform minimization on STS of resources
                          without dependencies.
   -p, --printsts       : Print all computed STSs in separate files.
   -u, --ubts           : Deactivate untimed bi-simulation, timed simulation
                          minimization.
   -v, --verbose        : Print status information about current computation.


tayfun@ultra5:~/workspaceC/RTANA/build$ 
```

Figure A.2.: Parameters of the *realtimeAnalyzer* tool.

There is a set of parameters (cf. Figure A.2) with which the tool can be called. The most implemented options are the following:

- *-b, –burst*: If this parameter is set, the burst abstraction introduced in Subsection

[2] http://boost-spirit.com/home/

3.6.6 will be applied. If end-to-end latency constraints are defined in the system which shall be analyzed, the program will give an error message that the abstraction is not applicable for this system.

- *-c, –criterion*: The user is able to determine the type of counter-example which is generated by the tool. The user can select between the shortest counter-example, a counter-example with maximum lateness, a counter-example with minimum lateness, and an arbitrary counter-example. Note that the arbitrary counter-example is set as default. Note that this feature has been implemented in the context of the bachelor's thesis of [Koo14].

- *-e*: If some deadlines are violated, no counter-examples are generated if this option is set. The tool will work a bit faster in case of a deadline violation, as no output has to be generated.

- *-h, –hp*: Abstract all information about tasks with higher priorities than relevant for dependent resources. This technique was illustrated in Subsection 3.6.2.

- *-i*: With this parameter an abstraction on the initial non-determinism of the task with the highest priority is performed, i.e. it is assumed that the highest priority task is activated at time point null. This option is related to testing techniques.

- *-m*: In Subsection 3.5.1 it was stated that the bisimulation minimization of Subsection 3.3.3 is only applied for resource STSs, which have dependent resources. By setting this parameter, the minimization is also performed for resource STSs which from which no resource depends on.

- *-p, –printsts*: The computed state spaces of all systems will be printed into separate files.

- *-u, –ubts*: Deactivate the untimed bisimulation, timed simulation minimization presented in Subsection 3.3.3.

- *-v, –verbose*: Some debug information is printed to the console. This option could be helpful if new features are implemented and debugging must be performed.

When the tool is called with a model, a set of files are generated:

- *counterExample.txt*: If some deadlines are not met, *realtimeAnalyzer* will generate a counter example trace into this file leading to failure state according to the defined criterion.

- *responseTimes.txt*: All computed end-to-end latency times and local response times will be printed to this file.

- *<name>STS<number>.txt*: The computed state space of a resource with the name <name> will be printed into such a file, if the option *-p* is set.

Name	Size
analysis	1 item
DAS	6 items
CAN1.ecu	578 bytes
CAN1.sts	3.0 MB
LaneDetect.ecu	764 bytes
LaneDetect.sts	43.8 KB
Situation.ecu	607 bytes
Situation.sts	69.6 MB

Figure A.3.: Generated file structure of a system architecture.

In Figure A.3 the file structure generated by the algorithm after an analysis has been performed is illustrated. This file structure resulted by the application of the lane-keeping-support system (LKS) which was introduced in Section 3.7. For all resources two kinds of files are generated, i.e. an *.ecu* and an *.sts* file. The former contains the structural information about the resource, i.e. the allocated tasks, the task parameters, and the scheduling policy. The latter contains the computed STS of the corresponding resource.

The tool checking the virtual integration condition is implemented as an Eclipse plugin. Contracts from the model are extracted and translated to timed automata. Besides all automata, event reproduction via glue automata as mentioned in this thesis are generated automatically. Also the necessary queries are generated. Thereafter, the verification back-end of UPPAAL is called automatically. At last, the verification result is shown in the console.

B. Handling Architectures including Restricted Loop Structures

So far, systems with no feedback loops were considered. If loops are contained a holistic analysis has to be performed in general as the state spaces of the dependent resources cannot be separated. For a restricted class of feedback-loops the iterative analysis approach can still be applied. For this, assume the following scenario: Let a resource with $i = n + m$ tasks be given. The n-highest priority tasks of a resource are either independent or depend on a set of task, where no feedback-loop is included (simple dependency chains). The m-lower priority tasks depend on tasks, which are triggered by the some of the n-higher priority tasks, thus leading to a feedback-loop. For loop structures of this type the following is performed: First, the partial state space of the resource consisting of the behavior of the n-highest priority tasks is computed. This is possible and leads to correct timings for these tasks, as none of the m-lower priority tasks can lead to interferences (cannot interrupt the other tasks). The rest of the state space of the resource is computed, whenever the input behavior of the m-lower priority tasks has been computed.

What was so far not shown is the general procedure of the *realtimeAnalyzer* tool. This is illustrated in Figure B.1. As an input the parser delivers a set of resources with allocated tasks, for which the corresponding STSs have to be computed. This set is stored in a queue q. The algorithm terminates, if q is empty and thus all STSs have been computed, or a deadline is violated.

In constraint $c2$ it is determined whether the input behaviors of all allocated tasks of the currently considered resource have already been determined. If so, the input is computed by the application of the composition and interface computation operations described in Subsection 3.4. The resource STS is then computed and the next resource is considered. For now, let us skip the constraint $c5$ and assume that it evaluates to *false*.

If the constraint $c2$ evaluates to *false*, it is checked in constraint $c3$ whether a partial computation can be performed, i.e. the method determines the set of tasks which do not depend on other tasks or are interrupted by such tasks as described above. With this, the above described restricted feedback-loops can be handled. If a partial computation is possible, the flag *partial* is set to *true* and the partial state space is computed. The constraint $c5$ evaluates to *true* for this case and thus the resource is put again back to the queue. The rest of the state space of this resource is then computed later, when the missing input behaviors have been determined. Thereby, the resource is also included into the set p which is detailed later.

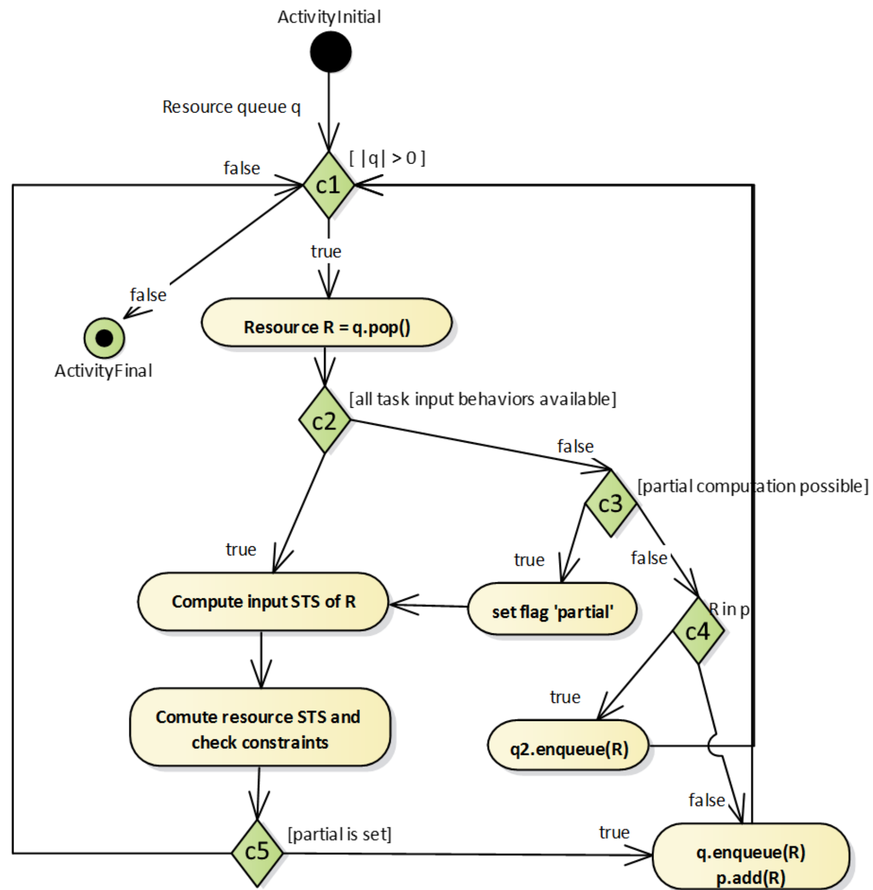


Figure B.1.: General procedure of the timing analysis approach.

If no partial computation can be performed ($c3$ evaluates to false), it is checked whether this resource has already been considered before (constraint $c4$). For this, the state set p is used. This set is empty at the beginning and is filled with resources for which a partial state space cannot be computed. Thus, if a resource is considered the first time, $c4$ evaluates to false and the resource is put back to the working queue q and included into p to annotate it as "‘already considered"’. If a resource is contained in p and the constraint $c4$ therefore evaluates to true, the STS of the corresponding resource is tried to be computed the second time, which is still not possible as the input behaviors are incomplete. For these cases the algorithm infers that there might be more complex loop structures included in the architecture. Thus, such a resource is added in a second queue $q2$. All states which are added in this queue are handled, when the iterative analysis finishes. Thus, the resources in $q2$ are then handled in a holistic manner.

C. Generated Timed Automata

In Subsection 4.6.1 the evaluation of the impact analysis approach on the specification level was performed by the application of a driver assistance system. Here, a subset of the generated timed automata for the text-pattern occurring in the assume/guarantee parts of the contracts are illustrated. All generated automata which are structurally equal to those illustrated here are not shown. The details about the construction and the functionality of the automata are not described here. For this, please refer to Subsection 4.4

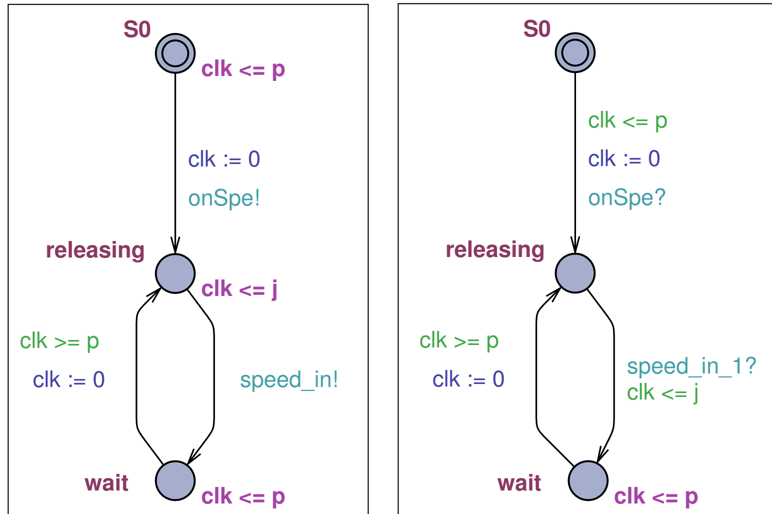


Figure C.1.: Left: Generated trigger automaton for overall system assumption $A21$; Right: Observer automaton of assumption A_{ACC}.

In the left part of Figure C.1 the trigger automaton which results from the assumption $A21$ of the overall driver assistance system is illustrated. The generated event *speed_in* is observed by the automaton in the right part of the figure, which results from the assumption part of the ACC system. This automaton receives the event *speed_in_1* as the original event *speed_in* is replicated by a glue automaton.

A further automaton, which observes the event *speed_in* is the guarantee automaton in the left part of Figure C.2. This automaton is obtained from the guarantee $G2$ of the

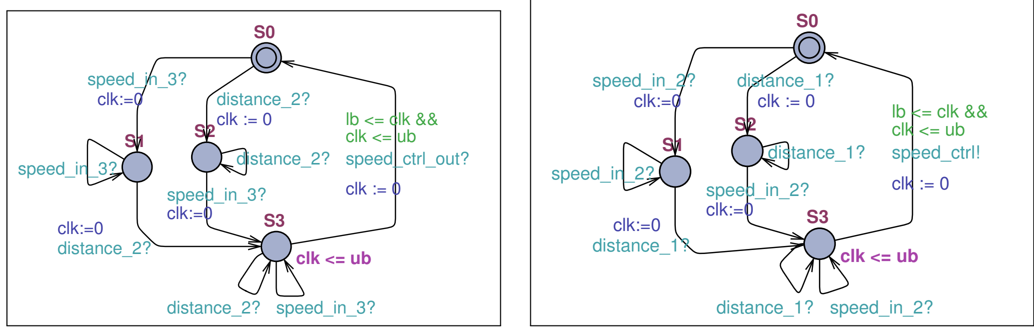


Figure C.2.: Left: Generated observer automaton for overall system guarantee $G2$; Right: Generated transceiver automaton of guarantee G_{ACC}.

overall system. It further receives the input event *distance*, for which there is a similar trigger automaton as the one for the event *speed_in*, but which is not illustrated here. The automaton checks the end-to-end timing latency constraint from receiving both events *speed_in* and *distance* up to the point in time where the event *speed_ctrl_out* is received.

In the right part of Figure C.2 the transceiver automaton for the guarantee G_{ACC} of the ACC system is illustrated. It is triggered by the reception of both events *speed_in* and *distance* and generated the output event *speed_ctrl* after the specified interval.

The observer automaton of the assumption A_{VDS} of the VDS system is illustrated in Figure C.3. Besides the event *speed_ctrl* this event receives the events *trajectoryData* *yawRate* and checks whether the time distance all events are received are within the specified timing bounds. Thereby, the automaton allows that the events may be received in any order.

The transceiver automaton of Figure C.4 illustrates the guarantee G_{VDS} of the VDS system. The events *speed_ctrl*, *trajectoryData*, and *yawRate* may be received in any order. The time all three events are received, the automaton sends both events *steer_ctrl_out* and *speed_ctrl_out* within the specified timing bounds. The latter event is received by the observer automaton of the overall system guarantee $G2$ which then checks, whether the end-to-end latency is violated or the event is received within the allowed bounds.

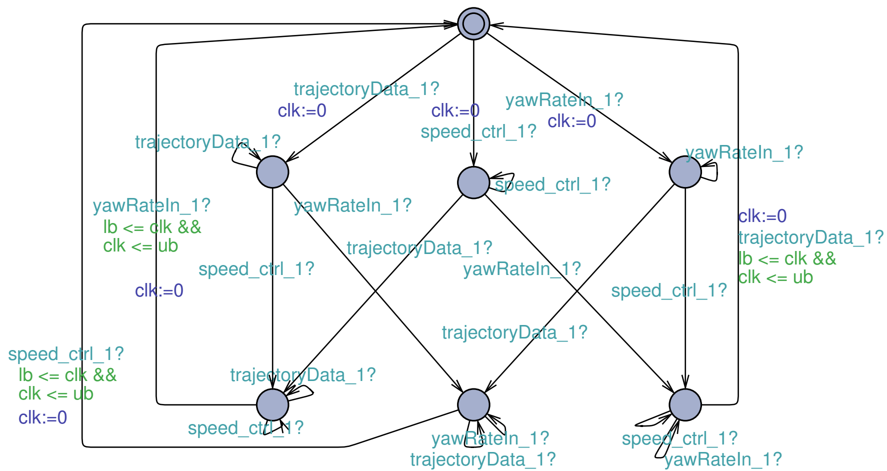


Figure C.3.: Observer automaton of assumption A_{VDS}.

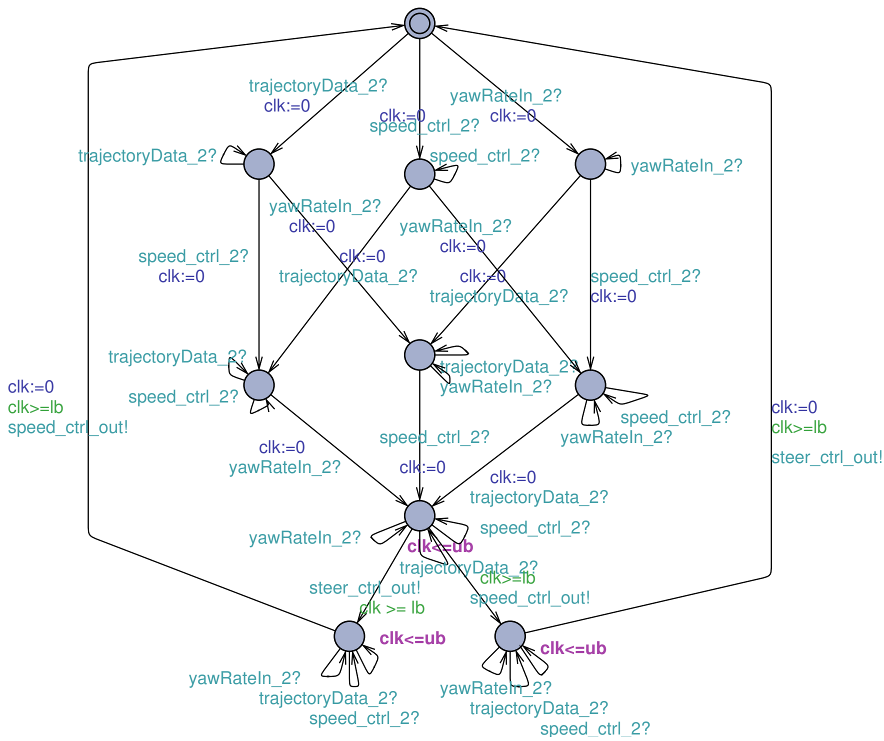


Figure C.4.: Transceiver automaton of guarantee G_{VDS}.

List of Figures

1.1. General concept of the model-based design approach. 12
1.2. Considered system architectures with assume/guarantee style contracts. . 13
1.3. Contributions of this thesis. 15

2.1. E/E-Architecture of typical medium-sized cars. 22
2.2. General classes of scheduling algorithms [Mar06]. 23
2.3. Periodical activation of independent tasks. 25
2.4. Event streams with burst behavior. 27
2.5. Event stream automaton with a period $p = 20$ and a jitter $j = 1$. 32
2.6. STS for an event stream automaton with $p = 20, j = 1$. 32
2.7. System architectures of interest. 35
2.8. Overview of the MARTE profile. 36
2.9. Example for a contract specification. 40

3.1. Scenarios of possible timings (right) of the distributed system (left). . . . 51
3.2. Overview of contributions of thesis. 52
3.3. System architecture containing four resources and dependencies. 62
3.4. Example architecture with periods $p_{\tau_1} = 60$, $p_{\tau_2} = 5$, $p_{\tau_4} = 60$, and computation times $c_{\tau_1} = 35$, $c_{\tau_2} = 2$, $c_{\tau_3} = 4$, $c_{\tau_4} = 12$, and priorities $pr_{\tau_1} > pr_{\tau_2}$, $pr_{\tau_4} > pr_{\tau_3}$. 64
3.5. Scenarios with a non-overlapping activation of task τ_a (left) and an overlapping activation of τ_b (right). 65
3.6. Two tasks hp, lp and the interrupt scenarios. 66
3.7. Example for invariants in automata. 68
3.8. Original STS for an event stream automaton with $p = 20, j = 1$. 69
3.9. Simplified STS for an event stream automaton with $p = 20, j = 1$. 69
3.10. Left: task dependencies of task τ_{B1}; Right: computation of interface between ECU A and ECU B. 72
3.11. Left: TDG with synchronous activation with resulting activation behavior; Right: TDG with asynchronous activation with resulting activation behavior. 80
3.12. Example for relating indexes of clocks of different matrices. 82

3.13. Left: Resulting state spaces of architecture in Figure 3.4; Right: Overall computation times – Blue curves: Without UBTS minimization; Red curves: With UBTS minimization. 97
3.14. Example trace - Top: an output path of a resource; Bottom: computed interface. 100
3.15. Event sequence. 101
3.16. Effect of abstraction. 102
3.17. Resulting state spaces of architecture in Figure 3.4 – Blue curves (marked with dots): Number of states with abstraction of hp clocks; Red curves (marked with x): Without abstraction. 103
3.18. Two paths of an input STS of a resource, where $97 \preccurlyeq 278$; Blue box (on top right): State of resource STS created if input states 97 and 278 were merged. 104
3.19. Resulting state spaces of architecture in Figure 3.4 – Blue curves (marked with dots): Number of states with timed simulation relation abstraction; Red curves (marked with x): Without abstraction. 106
3.20. Merging states by the application of the *abstract period clock*. 108
3.21. Resulting state spaces of architecture in Figure 3.4 – Blue curves (marked with dots): Number of states without abstraction; Red curves (marked with x): Number of states with burst abstraction. 109
3.22. Functional structure of the driver assistance system. 110
3.23. High level component structure of the overall driver assistance system. . . 111
3.24. Decomposition of the lane keeping assistance system. 112
3.25. Functional view of the lane keeping assistance system. 113

4.1. Overview of contributions of thesis. 118
4.2. System architectures of interest. 124
4.3. Impact analysis methodology - Left: General control flow; Right: Integration of verification techniques. 126
4.4. Impact Analysis procedure for a component N. 127
4.5. Automata for *R1*-pattern – Left: Trigger automaton; Center: Observer automaton; Right: Observer with explicit bad state. 137
4.6. Automata for *R2*-pattern – Left: Trigger automaton; Center: Observer automaton; Right: Observer with explicit bad state. 138
4.7. Automata for *R3*-pattern – Left: Transceiver automaton; Center: Observer automaton; Right: Observer with explicit bad state. 139
4.8. Transceiver automata for *R3*-pattern with sets of events– Left: Sorted set of events (“*and then*”); Right: Unsorted set of events. 140
4.9. Concept of structure for unsorted set of events within an *R3*-pattern. . . . 141
4.10. Observer automaton for *R4*-pattern. 141
4.11. Glue automaton reproducing an event σ for multiple observer automata. . 142

4.12. Methodology of the Impact Analysis (timing analysis combined with refinement check) . . . 144
4.13. Test systems. . . . 152
4.14. Task parameters. . . . 152
4.15. Measured average computation times. . . . 153
4.16. Visualization of some computation times from Table 4.15: a) $ECU1$ in $System1$ (left), b) $ECU2$ on $System2$ (center), and c) $ECU3$ on $System3$. 154
4.17. Left: Adapted architecture; Right: Comparison of a re-computation of the STS of R_2 (blue curve, marked with dots) and a refinement check on the output of R_1 (red curve, marked with x). . . . 154

A.1. Modeling in MARTE and enabling the connection to the analysis approach. 161
A.2. Parameters of the *realtimeAnalyzer* tool. . . . 162
A.3. Generated file structure of a system architecture. . . . 164

B.1. General procedure of the timing analysis approach. . . . 166

C.1. Left: Generated trigger automaton for overall system assumption $A21$; Right: Observer automaton of assumption A_{ACC}. . . . 169
C.2. Left: Generated observer automaton for overall system guarantee $G2$; Right: Generated transceiver automaton of guarantee G_{ACC}. . . . 170
C.3. Observer automaton of assumption A_{VDS}. . . . 171
C.4. Transceiver automaton of guarantee G_{VDS}. . . . 171

List of Tables

3.1. Preservation of universal and existential properties considering approximations. 97
3.2. Scheduling-relevant information for tasks and messages in milliseconds. . . 113
3.3. Timing Analysis results in milliseconds for (a) analysis without abstraction, (b) using abstraction of period clocks, (c) using burst abstraction, and (d) applying testing. 114

4.1. Contract $C_1 = (A_1, G_1)$ of the driver assistance system. 147
4.2. Contract $C_2 = (A_2, G_2)$ of the driver assistance system. 147
4.3. Contract $C_{ACC} = (A_{ACC}, G_{ACC})$ of the adaptive cruise control subsystem. 148
4.4. Contract C_{LKS} of the lane-keeping-support subsystem. 149
4.5. Contract $C_{VDM} = (A_{VDM,1} \wedge A_{VDM,2}, G_{VDM})$ of the vehicle dynamic management system. 149
4.6. Adapted contract C_{LKS} of the lane-keeping-support system. 150

Index

$\omega-$Language, 27

Abstract Period Clock, 107
Active Task Map, 35, 67
Assertion, 39
Asynchronous Activation, 79

Büchi Automaton, 27, 29
Bisimulation Relation, 76
Bus, 37
  CAN, 21
  LIN, 21
  MOST, 21

Clock Constraint, 29
  Extension, 71
  Projection, 71
Clock Valuation, 29
Clock Zone, 30
  Boundedness, 31
  Canonical Form, 31
  Extension, 71
  Normalization, 31
  Projection, 71
  Zone Refinement, 102
Complete Specification, 125
Completeness, 91
Component, 33
Computation Time Clock, 64
Contract, 39, 41
  Assumption, 39
  Canonical, 130
  Composition, 41
  Conjunction, 42
  Consistency, 40
  Directedness, 132
  Guarantee, 39
  Maximum Implementation, 41
  Receptivity, 132
  Refinement, 42
  Satisfaction, 40, 41
  Saturation, 130
  Virtual Integration Condition, 43
Correctness, 91
Critical Instance, 54

Deadline, 23, 24
  Hard Deadline, 24
  Soft Deadline, 24
Deadlock Free, 74, 136
Delegation, 33
Difference Bound Matrix, 32
Duration Clock, 99

Event Model, 25
  Event Stream, 25
  Jitter, 25
  Occurrence Function, 25
  Period, 25
Existential Property, 97

Fixed Priority Scheduling, 54

Incomplete Specification, 125
Interface, 33
  Directed Interface, 34

Port, 33
Invariant, 29

Language, 27
Liveness Property, 46
Location Vector, 63

Observer Automaton, 134, 135

Periodic Activation Clock, 64

Ready Task List, 35, 67
Reference Clock, 32
Refinement, 40
Requirement, 38
Resource, 33, 34, 37
Bus System, 33
Electronic Control Unit, 33
Response Time Clock, 65

Satisfaction, 77
Scheduling, 23
Dynamic, 24
Feasibility, 23
FIFO, 35
FPS, 35
Preemption, 24
RMS, 54
Scheduler, 37
Scheduling Policy, 23
Static, 24
TDMA, 35
Simulation Relation, 74
Soundness, 91
Synchronous Activation, 79
System Architecture, 36

Task, 23, 37
Best-Case Execution Time, 24
Deadline, 23, 24
Hard Deadline, 24
Immediate Predecessor, 25
Independent Task, 25
Instance, 79
Instance Bound, 65, 81
Overlapping Activation, 64
Predecessor, 25
Priority, 24
Release Time, 24
Response Time, 25
Soft Deadline, 24
Tail Task, 79
Task Dependency Graph, 25
Worst-Case Execution Time, 24
TCTL, 135
Timed Automaton, 28
Resource STS, 63
Symbolic Transition System, 30
Timed Transition System, 29
Timed Language, 28
Extension, 34
Restriction, 34
Timed Safety Automaton, 29
Timed Simulation Relation, 103
Timed Word, 28
Prefix, 28
Restriction, 34
Trace, 28
Transceiver Automaton, 135
Trigger Automaton, 135

UBTS Relation, 95
Universal Property, 97

Nomenclature

$\alpha : S \to S'$	An abstraction function computing an abstracted state $s' \in S'$ from a concrete state $s \in S$
$\mathcal{A}_s$	The active task map of a state s
$act : \mathsf{T} \to \{sync, async\}$	The function determining the activation type of a task which may be synchronous or asynchronous
$\mathcal{B}_s : \mathsf{T} \to \mathbb{N}$	The Function determining active task instances of a specific type in a state s
$bcet$	Shortage for *best-case execution time* of a task
$\mathscr{C}$	The set of all clocks
$C = (A, G)$	A contract C consisting on an assumption A and a guarantee G
$c_c(\tau)$	The computation time clock of a task type τ
$c_r(\tau)$	The clock tracing the periodical activation of a task type τ
$c_r(t)$	The response time clock of a task instance t
c_τ	The execution time of a task type τ (used when $bcet = wcet$)
$D \in \mathcal{D}$	A clock zone in the set of all zones over a clock set $\mathscr{C}$
$D_{\vert C}, D_{\vert_C^{-1}}$	The projection and extension operations for a clock zone D wrt. clock variables in C
Σ	An alphabet consisting of a set of symbols
Σ^ω	The set of all (timed) words over alphabet Σ
ECU	Abbreviation for *Electronic Computation Unit*
η^+, η^-	Upper and lower occurrence functions characterizing event streams
$I = in \cup out$	The interface of a component consisting of input and output ports
$init : \mathsf{T} \to 2^{\mathsf{T}}$	The recursive function determining the set of independent tasks triggering a given task
$inst : T \to \mathcal{N}$	The function determining maximal number of activations of a task
hp	Shortage for *higher priority* (task)
$\mathcal{L}$	Language over some finite alphabet Σ
$\mathcal{L}_{\downarrow\Sigma}, \mathcal{L}_{\uparrow\Sigma}$	The restriction and extension operations for a timed language $\mathcal{L}$ wrt. symbols in Σ
L	A set of discrete locations
lp	Shortage for *lower priority* (task)
$\mathcal{M}_{s,async}$	The activation function of a state for tasks with asynchronous activation
$\mathcal{M}_{s,sync}$	The trigger map of a state for tasks with synchronous activation

$max(i, \cdots, j)$	The function determining maximum value of a given set of numbers
$\mathbb{N}, \mathbb{N}_{>0}$	Set of the natural numbers with and without value 0
φ	A clock constraint
$\Phi(C)$	The set of all clock constraints over clock set C
pr_τ	The priority of a task type τ
$pre : \mathsf{T} \to 2^{\mathsf{T}}$	The function determining set of immediate predecessors of a task
ρ	An infinite sequence of symbols, or untimed word over some alphabet Σ
$\mathbb{Q}_{\geq 0}, \mathbb{Q}_{>0}$	Set of non-negative rational numbers with and without value 0
$r, \approx$	Bisimulation relation
$\mathbb{R}_{\geq 0}, \mathbb{R}_{>0}$	Set of non-negative real numbers with and without value 0
$r_{ts}, \preccurlyeq$	Timed simulation relation
$r_{ubts}, \preceqq$	Untimed bisimulation, timed simulation relation
$\mathcal{R}_s$	The ready task list of a state s
STS	Shortage for Symbolic Transition System
$\tau \in \mathsf{T}$	A task type in a set of task types
σ	A timed trace or word
τ	An infinite sequence of time values in $\mathbb{R}^+$
Sch	Abbreviation for scheduling policy
ν	The clock valuation function
$wcet$	Shortage for *worst-case execution time* of a task
ξ	A map which determines the relation between states of two or more STSs
χ_r	The set of equivalence classes induced by relation r
ζ	The function to reorder zones

Bibliography

[ABL98] Luca Aceto, Augusto Burgueño, and Kim Larsen. Model checking via reachability testing for timed automata. In Bernhard Steffen, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *Lecture Notes in Computer Science (LNCS)*, pages 263–280. Springer Berlin / Heidelberg, 1998. 10.1007/BFb0054177.

[ABR+86] N. C. Audsley, A. Burns, M. F. Richardson, K. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority preemptive scheduling. *The Computer Journal*, 29 (5):390 – 395, 01/1986.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, April 1994.

[AdADS+06] B.Thomas Adler, Luca de Alfaro, LeandroDias Da Silva, Marco Faella, Axel Legay, Vishwanath Raman, and Pritam Roy. Ticc: A tool for interface compatibility and composition. In Thomas Ball and RobertB. Jones, editors, *Computer Aided Verification*, volume 4144 of *Lecture Notes in Computer Science*, pages 59–62. Springer Berlin Heidelberg, 2006.

[AFM+04] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Times: A tool for schedulability analysis and code generation of real-time systems. In Kim G. Larsen and Peter Niebert, editors, *Formal Modeling and Analysis of Timed Systems*, volume 2791 of *Lecture Notes in Computer Science*, pages 60–72. Springer Berlin Heidelberg, 2004.

[AM02] Yasmina Abdeddaim and Oded Maler. Preemptive job-shop scheduling using stopwatch automata. In *in TACAS '02: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 113–126. Springer, 2002.

[BBB+11] A. Baumgart, E. Böde, M. Büker, W. Damm, G. Ehmen, T. Gezgin, S. Henkler, H. Hungar, B. Josko, M. Oertel, T. Peikenkamp, P. Reinkemeier, I. Stierand, and R. Weber. Aritechture modeling. Technical report, OFFIS, November 2011.

[BCF+08] Albert Benveniste, Benoît Caillaud, Alberto Ferrari, Leonardo Mangeruca, Roberto Passerone, and Christos Sofronis. Multiple viewpoint contract-based specification and design. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, volume 5382 of *Lecture Notes in Computer Science*, pages 200–225. Springer Berlin Heidelberg, 2008.

[BCN+11] A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone, J.B. Raclet, P. Reinkemeier, A. Sangiovanni-Vincentelli, W. Damm, T. Henzinger, and K. Larsen. Contracts for systems design. *Inria Research Report No.8147*, March 2011.

[BDFP00] Patricia Bouyer, Catherine Dufourd, Emmanuel Fleury, and Antoine Petit. Are timed automata updatable? In E. Allen Emerson and Aravinda P. Sistla, editors, *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 464–479. Springer Berlin Heidelberg, 2000.

[BDH+12] Sebastian Bauer, Alexandre David, Rolf Hennicker, Kim Guldstrand Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. Moving from specifications to contracts in component-based design. In Juan de Lara and Andrea Zisman, editors, *Fundamental Approaches to Software Engineering*, volume 7212 of *Lecture Notes in Computer Science*, pages 43–58. Springer Berlin Heidelberg, 2012.

[BFM+08] Luca Benvenuti, Alberto Ferrari, Leonardo Mangeruca, Emanuele Mazzi, Roberto Passerone, and Christos Sofronis. A contract-based formalism for the specification of heterogeneous systems (invited). In *FDL*, pages 142–147, 2008.

[BH09] Beatrice Bérard and Serge Haddad. Interrupt Timed Automata. In Luca de Alfaro, editor, *Foundations of Software Science and Computational Structures*, volume 5504 of *Lecture Notes in Computer Science*, pages 197–211. Springer Berlin / Heidelberg, 2009.

[BHKW12] Dirk Beyer, Thomas A. Henzinger, M. Erkan Keremoglu, and Philipp Wendler. Conditional model checking: A technique to pass information between verifiers. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 57:1–57:11, New York, NY, USA, 2012. ACM.

[BLN+13] Dirk Beyer, Stefan Löwe, Evgeny Novikov, Andreas Stahlbauer, and Philipp Wendler. Precision reuse for efficient regression verification. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 389–399, New York, NY, USA, 2013. ACM.

[BML11] SebastianS. Bauer, Philip Mayer, and Axel Legay. Mio workbench: A tool for compositional design with modal input/output interfaces. In Tevfik Bultan and Pao-Ann Hsiung, editors, *Automated Technology for Verification and Analysis*, volume 6996 of *Lecture Notes in Computer Science*, pages 418–421. Springer Berlin Heidelberg, 2011.

[BMS09] Matthias Büker, Alexander Metzner, and Ingo Stierand. Testing real-time task networks with functional extensions using model-checking. In *Proceedings of the 14th IEEE international conference on Emerging technologies & factory automation*, ETFA'09, pages 564–573, Piscataway, NJ, USA, 2009. IEEE Press.

[BMSH10] Sebastian Bauer, Philip Mayer, Andreas Schroeder, and Rolf Hennicker. On weak modal compatibility, refinement, and the mio workbench. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *Lecture Notes in Computer Science*, pages 175–189. Springer Berlin Heidelberg, 2010.

[But05] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications.* Springer Verlag, University of Pavia, Italy, 2005.

[BW13] Dirk Beyer and Philipp Wendler. Reuse of verification results: Consitional model checking, precision reuse, and verification witnesses. In Ezio Bartocci and C.R. Ramakrishnan, editors, *Model Checking Software*, volume 7976 of *Lecture Notes in Computer Science*, pages 1–17. Springer Berlin Heidelberg, 2013.

[BY04] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In W. Reisig and G. Rozenberg, editors, *Lecture Notes on Concurrency and Petri Nets*, LNCS vol. 3098. Springer–Verlag, 2004.

[CDT13] Alessandro Cimatti, Michele Dorigatti, and Stefano Tonetta. Ocra: A tool for checking the refinement of temporal contracts. In *Proceedings of the 28th International Conference on Automation Software Engineering (ASE)*, pages 702–705. IEEE computer society, 2013.

[CGP03] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Păsăreanu. Learning assumptions for compositional verification. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'03, pages 331–346, Berlin, Heidelberg, 2003. Springer-Verlag.

[CGR12] Alessandro Carioni, Silvio Ghilardi, and Silvio Ranise. MCMT in the land of parametrized timed automata. In Markus Aderhold, Serge Autexier, and Heiko Mantel, editors, *Proceedings of the 6th International Verificatrion Workshop (VERIFY)*, volume 3 of *EPiC Series*, pages 47–64, 2012.

[CKT03] S. Chakraborty, S. Künzli, and L. Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *Proceedings of 6th Design, Automation and Test in Europe (DATE)*, pages 190–195. IEEE, 2003.

[CL00] F. Cassez and K. Larsen. The impressive power of stopwatches. In *Proceedings of the 11th International Conference on Concurrency Theory (CONCUR)*, Lecture Notes in Computer Science, pages 138–152. Springer, 2000.

[CRST13] Alessandro Cimatti, Marco Roveri, Angelo Susi, and Stefano Tonetta. Validation of requirements for hybrid systems: A formal approach. *ACM Trans. Softw. Eng. Methodol.*, 21(4):22:1–22:34, February 2013.

[DH01] Werner Damm and David Harel. LSCs: Breathing life into Message Sequence Charts. *Formal Methods in System Design*, 19(1):45–80, July 2001.

[DHJ+11] W. Damm, H. Hungar, B. Josko, T. Peikenkamp, and I. Stierand. Using contract-based component specifications for virtual integration testing and architecture design. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1–6, March 2011.

[Dil90] David Dill. Timing assumptions and verification of finite-state concurrent systems. In Joseph Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 197–212. Springer Berlin Heidelberg, 1990.

[DILS09] A. David, J. Illum, K.G. Larsen, and A. Skou. Model-based framework for schedulability analysis using uppaal 4.1. In G. Nicolescu and P.J. Mosterman, editors, *Model-Based Design for Embedded Systems*, 2009.

[DLL+10a] A. David, K. G. Larsen, A. Legay, U. Nyman, and A. Wasowski. ECDAR: an environment for compositional design and analysis of real time systems. In *Automated Technology for Verification and Analysis - 8th International Symposium, ATVA 2010, Singapore, September 21-24, 2010. Proceedings*, pages 365–370, 2010.

[DLL+10b] A. David, K. G. Larsen, A. Legay, U. Nyman, and A. Wasowski. Timed I/O automata: a complete specification theory for real-time systems. In *Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control (HSCC)*, pages 91–100, 2010.

[DMS09] H. Dierks, A. Metzner, and I. Stierand. Efficient model-checking for real-time task networks. In *Proceedings of the 6th International Conference on Embedded Software and Systems*. IEEE Computer Society, May 2009.

[Fif14] Conrad Fifelsky. Spezifikation und transformation eines echtzeit-systems von marte zu rtana. Bachelor's thesis (Studienarbeit), University of Oldenburg, 2014.

[FKPY07] Elena Fersman, Pavel Krčál, Paul Pettersson, and Wang Yi. Task automata: Schedulability, decidability and undecidability. *International Journal of Information and Computation*, 205(8):1149–1172, August 2007.

[FPY02] E. Fersman, P. Pettersson, and W. Yi. Timed automata with asynchronous processes: schedulability and decidability. In *Proceedings of TACAS*. Springer, 2002.

[GHRS12] T. Gezgin, S. Henkler, A. Rettberg, and I. Stierand. Abstraction techniques for compositional state-based scheduling analysis. In *Brazilian Symposium on Computing System Engineering, Workshop of Embedded Systems*, SBESC, Natal, Brazil, 2012.

[GHSR14] T. Gezgin, S. Henkler, I. Stierand, and A. Rettberg. Impact analysis for timing requirements on real-time systems. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2014 IEEE 20th International Conference on*, pages 1–10, Aug 2014.

[GKR15] T. Gezgin, B. Koopmann, and A. Rettberg. Combining an iterative state-based timing analysis with a refinement checking technique. In *Proceedings of the* 5th *IFIP International Embedded Systems Symposium (IESS 2015)*, November 2015.

[GSHR13a] Tayfun Gezgin, Ingo Stierand, Stefan Henkler, and Achim Rettberg. Contract-based compositional scheduling analysis for evolving systems. In *Proceedings of the 4th IFIP International Embedded Systems Symposium*, IESS, Paderborn, Germany, june 2013.

[GSHR13b] Tayfun Gezgin, Ingo Stierand, Stefan Henkler, and Achim Rettberg. State–based scheduling analysis for distributed real–time systems. *Journal on Design Automation for Embedded Systems*, pages 1–18, July 2013.

[GWB15] Tayfun Gezgin, Raphael Weber, and Matthias Bueker. State-based real-time analysis for function networks and marte. In *Inproceedings of the* 18th *IEEE International Symposium on Real-Time Distributed Computing (ISORC)*, pages 158–165, April 2015.

[GWG11] Tayfun Gezgin, Raphael Weber, and Maurice Girod. A refinement checking technique for contract–based architecture designs. In *Fourth International Workshop on Model Based Architecting and Construction of Embedded Systems (ACES-MB)*, October 2011.

[GWO14] Tayfun Gezgin, Raphael Weber, and Markus Oertel. Multi-aspect virtual integration approach for real-time and safety properties. In *International Workshop on Design and Implementation of Formal Tools and Systems*, October 2014.

[HNSY92] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. In *Proceedings of the Seventh Annual Symposium on Logic in Computer Science (LICS)*, pages 394–406. IEEE Computer Society, 1992.

[Hun11] Hardi Hungar. Components and contracts: A semantical foundation for compositional refinement. Technical report, OFFIS, 2011.

[HV06] M. Hendriks and M. Verhoef. Timed automata based analysis of embedded system architectures. In *Workshop on Parallel and Distributed Processing Symposium*, April 2006.

[JP86] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.

[KHM+96] R.P. Kurshan, R. H. Hardin, K. L. McMillan, J. A. Reeds, and N. J. A. Sloane. Efficient regression verification. In *IEE Proc. WODES'96*, pages 147–150, January 1996.

[KMY07] P. Krčál, L. Mokrushin, and W. Yi. A tool for compositional analysis of timed systems by abstraction. In E.B. Johnsen, O. Owe, and G. Schneider, editors, *Nordic Workshop on Programming Theory*, 2007.

[Koo14] Bjoern Koopmann. Generierung von gegenbeispielen für eine zustandsbasierte scheduling-analyse. Bachelor's thesis (Studienarbeit), University of Oldenburg, 2014.

[KY04] Pavel Krčál and Wang Yi. Decidable and undecidable problems in schedulability analysis using timed automata. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 236–250. Springer Berlin Heidelberg, 2004.

[Leh11] Steffen Lehnert. A review of software change impact analysis. *Ilmenau University of Technology, Tech. Rep*, 2011.

[LL73] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, 1973.

[LLPY97] K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Efficient verification of real-time systems: Compact data structure and state-space reduction. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, RTSS '97, pages 14–, Washington, DC, USA, 1997. IEEE Computer Society.

[LPT09] Kai Lampka, Simon Perathoner, and Lothar Thiele. Analytic real-time analysis and timed automata: a hybrid method for analyzing embedded real-time systems. In *Proceedings of the seventh ACM international conference on Embedded software*, EMSOFT, pages 107–116, New York, NY, USA, 2009. ACM.

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

[Mar06] Peter Marwedel. *Embedded System Design*. Springer Verlag, University of Dortmund, Germany, 2006.

[MC09] Georgiana Macariu and Vladimir Cretu. Model-based analysis of contract-based real-time scheduling. In *Proceedings of the seventh IFIP International Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS)*, pages 227–239, 2009.

[Mey92] Bertrand Meyer. Applying "Design by Contract". *Journal on Computer*, 25(10):40–51, October 1992.

[MV94] Jennifer McManis and Pravin Varaiya. Suspension automata: A decidable class of hybrid automata. In *Proceedings of the 6th International Conference on Computer Aided Verification*, CAV, pages 105–117, London, UK, 1994. Springer-Verlag.

[NWY99] C. Norström, A. Wall, and Wang Yi. Timed automata as task models for event-driven systems. In *Proceedings of the sixth International Workshop on Real-Time Computing Systems and Applications (RTCSA)*, pages 182–189. IEEE Computer Society, December 1999.

[OGR14] M. Oertel, S. Gerwinn, and A. Rettberg. Simulative evaluation of contract-based change management. In *Proceedings of the International Conference on Industrial Informatics (INDIN)*, 2014.

[OMG11] Object Management Group. UML Profile For MARTE: Modeling And Analysis Of Real-time Embedded Systems, June 2011.

[PGB+08] CorinaS. Păsăreanu, Dimitra Giannakopoulou, MihaelaGheorghiu Bobaru, JamiesonM. Cobleigh, and Howard Barringer. Learning to divide and conquer: applying the l* algorithm to automate assume-guarantee reasoning. *Formal Methods in System Design*, 32(3):175–205, 2008.

[PGH97] J.C. Palencia, J.J. Gutierrez, and M.G. Harbour. On the schedulability analysis for distributed hard real-time systems. In *Proceedings of the ninth Euromicro Workshop on Real-Time Systems*, pages 136 –143, June 1997.

[PH98] J.C. Palencia and M.G. Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS)*, pages 26–37. IEEE Computer Society, December 1998.

[PLE+11] Linh T. X. Phan, Jaewoo Lee, Arvind Easwaran, Vinay Ramaswamy, Sanjian Chen, Insup Lee, and Oleg Sokolsky. Carts: A tool for compositional analysis of real-time systems. *SIGBED Rev.*, 8(1):62–63, March 2011.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th annual symposium on Foundations of Computer Science*, pages 46–57. IEEE Computer Society, 1977.

[Pro07] Project SPEEDS: WP.2.1 Partners. SPEEDS Meta-model Behavioural Semantics — Complement do D.2.1.c. Technical report, The SPEEDS consortium, 2007.

[PTCT07] Linh T.X. Phan, Lothar Thiele, Samarjit Chakraborty, and P.S. Thiagarajan. Composing functional and state-based performance models for analyzing heterogeneous real-time systems. In *Proceedings of the 28th Real-Time Systems Symposium*, RTSS, pages 343 – 352. IEEE, December 2007.

[PWT+07] S. Perathoner, E. Wandeler, L. Thiele, A. Hamann, S. Schliecker, R. Henia, R. Racu, R. Ernst, and M. Harbour. Influence of different system abstractions on the performance analysis of distributed real-time systems. In *Proceedings of the 7th ACM & IEEE int. conference on Embedded software*, EMSOFT, pages 193–202, 2007.

[QGP10] S. Quinton, S. Graf, and R. Passerone. Contract-based reasoning for component systems with complex interactions. Technical Report TR-2010-12, Verimag Research Report, 2010.

[RE02] K. Richter and R. Ernst. Event model interfaces for heterogeneous system analysis. In *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, pages 506–513, 2002.

[RE10] J. Rox and R. Ernst. Exploiting inter-event stream correlations between output event streams of non-preemptively scheduled tasks. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE, Leuven, Belgium, 2010.

[Ric04] Kai Richter. *Compositional Scheduling Analysis Using Standard Event Models*. PhD thesis, Technical University of Braunschweig, Braunschweig, Germany, 2004.

[RRE03] K. Richter, R. Racu, and R. Ernst. Scheduling analysis integration for heterogeneous multiprocessor SoC. In *Procedings of the 24th Real-Time Systems Symposium (RTSS)*, pages 236–245. IEEE Computer Society, December 2003.

[RSRH11] Philipp Reinkemeier, Ingo Stierand, Philip Rehkop, and Stefan Henkler. A pattern-based requirement specification language: Mapping automotive specific timing requirements. In *Software Engineering 2011 - Workshopband*, LNI. GI, 2011.

[SBPM09] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, 2nd edition, 2009.

[SSL+14] Youcheng Sun, Romain Soulat, Giuseppe Lipari, Étienne André, and Laurent Fribourg. Parametric schedulability analysis of fixed priority real-time distributed systems. In Cyrille Artho and Peter Csaba Ölveczky, editors, *Formal Techniques for Safety-Critical Systems*, volume 419 of *Communications in Computer and Information Science*, pages 212–228. Springer International Publishing, 2014.

[TC94] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Journal of Microprocessing and Microprogramming*, 40:117–134, April 1994.

[TCN00] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *Proceedings of the International Symposium on Circuits and Systems (ISCAS)*, volume 4, pages 101–104. IEEE Computer Society, May 2000.

[Tin94] Ken Tindell. Adding Time-Offsets to Schedulability Analysis. Technical report, University of York, Department of Computer Science, England, 1994.

[WTH+14] Raphael Weber, Eike Thaden, Stefan Henkler, Jens Höfflinger, and Steffen Prochnow. Design space exploration for an industrial lane-keeping-support case study. In *Proceedings of DATE Conf. University Booth*, 2014.