Timing Verifikation von AUTOSAR Softwarearchitekturen

Dissertation
zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)
an der
Fakultät für Elektrotechnik, Informatik und Mathematik
der Universität Paderborn

vorgelegt von
M. Sc. Steffen Beringer

Paderborn,
Juni 2022

Timing Verifikation von AUTOSAR Softwarearchitekturen

Zusammenfassung

Automobilhersteller weltweit befinden sich zur Zeit in einem Wettlauf um die Einführung autonomer Fahrzeuge. Die Innovationen in diesem Bereich werden dabei fast ausschließlich durch die Entwicklung neuer Software realisiert. Dies führt dazu, dass die Software im Automobil immer komplexer wird, diese jedoch in immer kürzeren Abständen auf die Straße gebracht werden soll. Dabei muss die Software stets hohen Qualitätsanforderungen genügen, da sie in vielen Fällen sicherheitskritische Funktionen umsetzt. Dies macht die Validierung dieser Fahrfunktionen zu einer großen Herausforderung.

Des Weiteren gilt für viele Funktionen im Automobil, dass diese harte Echtzeitanforderungen erfüllen müssen, was zusätzlich im gesamten Entwicklungsprozess berücksichtigt werden muss. Die Validierung dieser Anforderungen ist besonders aufwändig, da die Durchführung von Simulationen, wie sie häufig zur Validierung von Steuergerätesoftware angewendet werden, nicht zielführend ist, da diese nicht alle möglichen Randfälle mitbetrachten und somit die Korrektheit der Anforderungen nicht nachweisen können.

Der Einsatz formaler Verifikation für Softwarearchitekturmodelle im Automobil ermöglicht es die Korrektheit und Robustheit von Steuergerätesoftware signifikant zu erhöhen. Dies gilt insbesondere für die Verifikation von Echtzeitanforderungen mittels Timing Verifikation. Existierende Methoden benötigen jedoch für die Durchführung der Timing Verifikation Steuergerätecode, der häufig erst spät im Prozess verfügbar ist oder berücksichtigen den in der Automobilindustrie verbreiteten AUTOSAR-Standard nicht. Des Weiteren ignorieren existierende Ansätze die Komplexität und Fehleranfälligkeit der Herleitung von formalen Echtzeitanforderungen für AUTOSAR

aus Anforderungsdokumenten. Dies führt insgesamt dazu, dass existierende Ansätze erst spät im Entwicklungsprozess angewendet werden können und eine geringe praktische Relevanz aufweisen.

In dieser Arbeit schlagen wir einen neuen Ansatz zur Timing Verifikation von AUTOSAR Softwarearchitekturen vor. Es wird zunächst eine Formalisierung für AUTOSAR Softwarearchitekturen, sowie für AUTOSAR Timing Anforderungen bereitgestellt. Daraufhin stellen wir einen Prozess vor, in dem AUTOSAR Timing Anforderungen zunächst auf Konsistenz geprüft werden und anschließend formal verifiziert werden. Im Gegensatz zu anderen Ansätzen wird kein Quellcode benötigt, sondern ausschließlich das AUTOSAR Architekturmodell verwendet. Die Methode zur Konsistenzprüfung beinhaltet die Transformation der AUTOSAR Timing Anforderungen in logische Constraints, die mithilfe von SMT-Solvern geprüft werden. Des Weiteren werden auf der Basis der logischen Constraints Mechanismen zur Auflösung inkonsistenter Anforderungsmengen zur Verfügung gestellt. Für die Verifikation der Timing Anforderungen wird eine Transformation des formalen AUTOSAR-Modells nach Timed Automata vorgestellt. Anschließend wird die Effizienz unseres Ansatzes mithilfe einer Fallstudie, sowie weiteren synthetischen Modellen bewertet.

Timing verification of AUTOSAR software architectures

Abstract

Automobile manufacturers worldwide are currently in a race to introduce autonomous vehicles. Innovations in this area are being realized almost exclusively through the development of new software. As a result, automotive software is becoming increasingly complex, while release-cycles are shortened. Still it is required to always meet high quality requirements, since in many cases it implements safety-critical functions. This makes validation of these functions a major challenge.

Furthermore, many functions in automobiles must fulfill hard real-time requirements, which must also be taken into account throughout the entire development process. The validation of these requirements is particularly time-consuming because simulations, which are often used to validate electronic control unit (ECU) software, are not effective because they do not consider all possible corner cases and thus cannot prove the correctness of the requirements.

The use of formal verification for automotive software architecture models significantly increases the correctness and robustness of ECU software. This is especially true for the verification of real-time requirements using timing verification. Existing methods require ECU source-code to perform timing verification, which is often not available until late in the process, or do not take into account the AUTOSAR standard that is widely used in the automotive industry. Furthermore, existing approaches ignore the complexity and error-proneness of deriving formal real-time requirements for AUTOSAR from requirements documents. Overall, this means that the existing approaches can only be applied late in the development process and have little practical relevance.

In this thesis, a new approach for timing verification of AUTOSAR software architectures is proposed. First, a formalization for AUTOSAR software architectures as well as for AUTOSAR timing requirements is provided. A process is then presented in which AUTOSAR timing requirements are first checked for consistency and then formally verified. In contrast to other approaches, no source code is required, but only the AUTOSAR software architecture model is used. The consistency checking method involves transforming the AUTOSAR timing requirements into logical constraints that are checked using SMT solvers. Furthermore, mechanisms for resolving inconsistent sets of requirements are provided based on the logical constraints. For the verification of timing requirements, a transformation of the formal AUTOSAR model to Timed Automata is presented. Subsequently, the efficiency of our approach is evaluated with the help of a case study, as well as further synthetic models.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Danksagung

Diese Dissertation und alle in dieser Arbeit erzielten Erkenntnisse sind das Ergebnis einer konstanten Unterstützung einer Vielzahl von Personen über einen langen Zeitraum gewesen. Daher möchte ich mich zu Beginn der Arbeit bei allen bedanken, die in irgendeiner Form hierzu beigetragen haben.

Mein ganz besonderer Dank gilt meiner Betreuerin Prof. Dr. Heike Wehrheim, die mir die Gelegenheit gegeben hat, mich als externer Promotionsstudent in ihrer Fachgruppe zu beteiligen. Vielen Dank für die wissenschaftliche Betreuung meines Themas, für die stets hilfreichen Diskussionen, für die Geduld bei der Betreuung und schließlich für die Begutachtung dieser Arbeit. Ich möchte mich ebenfalls bei der Prüfungskommission bestehend aus Prof. Dr. Gregor Engels, Jun-Prof. Dr. Henning Wachsmuth, Dr. Christian Soltenborn und Dr. Matthias Meyer für die aufgewendete Zeit bedanken.

Ebenfalls möchte ich mich bei allen (ehemaligen) Kollegen der Arbeitsgruppe bedanken, die mich während meiner Zeit an der Universität unterstützt haben: Nils Timm, Dominik Steenken, Galina Besova, Steffen Ziegert, Tobias Isenberg, Marie-Christine Jakobs, Oleg Travkin, Manuel Töws, Felix Pauck, Arnab Sharma, Sven Flake, Jürgen König, Jan Haltermann, Cedric Richter und Elisabeth Schlatt.

Diese Dissertation ist in Zusammenarbeit mit dem Unternehmen dSPACE entstanden. In diesem Zusammenhang möchte ich mich für die uneingeschränkte Unterstützung bei meinen Betreuern Elmar Schmitz und Dr. Matthias Gehrke bedanken. Des Weiteren möchte ich mich bei allen Kollegen meiner Gruppe bei dSPACE für die vielen fruchtbaren Diskussionen bedanken.

Zuletzt möchte ich mich ganz besonders bei meiner Frau Kristin bedanken, die mich in allen Phasen meiner Dissertation unterstützt hat und stets an mich und das Gelingen dieser Arbeit geglaubt hat.

1 Einleitung

Die Komplexität von Steuergerätesoftware und Regelalgorithmen im Automobil nimmt stetig zu, beispielsweise aufgrund der Integration von mehr Komfortfunktionen oder durch komplexere Steuerungen bei Elektrofahrzeugen. Dies hat zur Folge, dass die Entwicklung und der Test dieser Systeme zeitaufwändig ist. Erschwerend kommt hinzu, dass gerade Steuergerätesoftware sicherheitskritisch ist und zudem Echtzeitbedingungen einhalten muss, weshalb es notwendig ist zusätzlich zu den funktionalen Anforderungen auch Echtzeitanforderungen (Timing Anforderungen) zu verifizieren. In diesem Kapitel wird zunächst in Abschnitt 1.1 der Kontext der Arbeit vorgestellt und die Vorteile einer frühzeitigen Timing Verifikation von Steuergerätesoftware genannt. Die spezifischen Problemstellungen, die aus der Entwicklung und Absicherung komplexer Steuergerätearchitekturen entstehen, werden in Abschnitt 1.2 vorgestellt. Abschnitt 1.3 stellt die Lösungsansätze zur frühzeitigen Timing Verifikation vor, die im Rahmen der Arbeit umgesetzt wurden. Abschnitt 1.4 gibt schließlich eine Übersicht über die folgenden Kapitel.

1.1 Motivation

In Zeiten neuer gesellschaftlicher Herausforderungen wie dem Klimawandel, Ressourcenknappheit und dem demografischen Wandel sind es gerade neue Innovationen im Bereich des Automobils, die versprechen einen Beitrag zur Lösung dieser Herausforderungen leisten zu können. Durch die Einführung von elektrischen, autonom fahrenden Fahrzeugen ist es beispielsweise möglich Emissionen zu reduzieren und Ressourcen einzusparen (Tomás et al., 2020, Hannon, 2016). Darüber hinaus erhalten Fahrzeuge weitere Funktionen wie beispielsweise einen Automatischen Bremsassistenten, Automatische Geschwindigkeitsregelung (Adaptive Cruise Control (ACC)), Fahrspurerkennung (Lane-Detection) oder Elektronische Stabilitätskontrolle (Electronic Stability Control (ESP)), welche die Sicherheit beim Fahren erhöhen und die gesellschaftliche Teilnahme älterer Verkehrsteilnehmer ermöglichen (Yurtsever et al., 2020).

komplexer Regelalgorithmen

Diese neuen Innovationen basieren jedoch nicht mehr auf der Weiterentwicklung einzelner mechanischer Bauteile, sondern erfordern die Ausführung *komplexer Regelalgorithmen* verteilt auf eine Vielzahl von Steuergeräten und sind somit abhängig von Software (Broy, 2006). Dies führt dazu, dass die Menge an Software in modernen Autos seit Jahren exponentiell steigt (Broy et al., 2007). Ein modernes Auto wie beispielsweise ein 7er-BMW mit erweiterten Fahrerassistenzfunktionen enthält mehr als 150 Steuergeräte und mehr als 150 Millionen Zeilen Quellcode (Charette, 2021).

sicherheitskritisch

Diese Software gilt es nicht nur zu entwickeln, sondern in besonderem Maße auch abzusichern und zu testen. Denn gerade Software im Automobil ist häufig *sicherheitskritisch* und schwer nach der Auslieferung zu aktualisieren. Werden Absicherungsmaßnahmen nicht konsequent eingesetzt, kann fehlerhafter Steuergerätecode ins Serienfahrzeug gelangen und so Personenschäden durch Verkehrsunfälle verursachen, was letztendlich selbst für große Automobilkonzerne schwere wirtschaftliche Folgen haben kann (Kane et al., 2011, Douglas und Fletcher, 2019).

Darüber hinaus realisiert automotive Software sehr unterschiedliche Funktionen. Für einige Softwarefunktionen, die einen unmittelbaren Einfluss auf das Fahrverhalten des Fahrzeugs und die Sicherheit des Au-

tofahrenden haben, wie beispielsweise die Sicherheitselektronik oder die Software zur Steuerung des Antriebsstrangs und des Fahrwerks, gilt, dass sie nicht nur sicherheitskritisch ist, sondern die korrekte Funktionalität abhängig von der korrekten zeitlichen Ausführung ist und somit zusätzlich harten *Echtzeit*bedingungen genügen muss (Broy et al., 2007). Der Aufwand zur Validierung dieser Funktionen ist besonders hoch, da im gesamten Entwicklungsprozess diese Echtzeitanforderungen mit betrachtet werden müssen und die Verifikation besonders aufwändig ist (AUTOSAR, 2019h).

Echtzeit

Dies führt dazu, dass die Entwicklung und der Test automotiver Software immer teurer wird. Während am Anfang der 1980er Jahre der Kostenanteil für elektronische Komponenten und die Software eines Autos noch bei 10% lag, so waren es 2010 bereits 35%, und für 2030 wird geschätzt, dass dieser Anteil auf 50% weiter steigt (Charette, 2021). Gleichzeitig konkurrieren sowohl Automobilhersteller als auch Zulieferer untereinander und stehen so unter einem starken Kostendruck (Broy et al., 2007).

Im Zuge dieser Entwicklungen ist es zunehmend wichtiger die Entwicklung und den Test von Steuergeräten mithilfe moderner Softwareentwicklungs- und Qualitätssicherungsmethoden zu verbessern und zu beschleunigen. Für diese Methoden müssen integrierte (modellbasierte) Werkzeugumgebungen entwickelt werden, die diese Methoden umsetzen und die in der Automobilindustrie verwendeten Entwicklungsartefakte verarbeiten können (Broy et al., 2010).

Diese Dissertation wurde in Zusammenarbeit mit dem Unternehmen *dSPACE* * durchgeführt. dSPACE ist ein Anbieter von Simulations- und Validierungslösungen für die Entwicklung vernetzter, autonomer und elektrisch angetriebener Fahrzeuge. Das dSPACE Geschäftsfeld *Software-In-The-Loop-Testing* (SIL-Testing) beinhaltet Werkzeuge zum Simulieren und Testen virtueller (software-basierter) Steuergeräte (Kempkes und Walther, 2018). Dies ermöglicht es den Absicherungsprozess für die vollständige *Steuergerätesoftware* ohne die Verfügbarkeit eines Prototypensteuergeräts durchzuführen, indem diese auf handelsüblicher PC-Hardware, in der Cloud oder zusammen mit bereits existierenden Steuergeräten am HIL-Simulator in Verbindung mit einem Umgebungsmodell simuliert

dSPACE *

Software-In-The-Loop-Testing

Steuergerätesoftware

*http://www.dspace.de

werden (Kempkes und Walther, 2018). Dadurch können Integrations- und Systemtests früher im Entwicklungsprozess vorgenommen werden. Fehler können so frühzeitig erkannt und behoben werden und somit kann der gesamte Entwicklungsprozess beschleunigt werden. Ein virtuelles Steuergerät (*V-ECU*) kann dabei aus unterschiedlichen Modellen assembliert werden und somit auch auf verschiedenen Abstraktionsebenen vorliegen (Kempkes und Walther, 2018). Neben der Simulation einzelner Funktionsmodelle auf der Basis von MATLAB®/Simulink (MathWorks, 2022) können V-ECUs insbesondere Steuergeräte simulieren, die auf der Basis des AUTOSAR Standards (AUTOSAR, 2019b) generiert wurden.

V-ECU

Der *AUTOSAR* Standard spielt eine Schlüsselrolle bei der effizienten Entwicklung von Steuergeräten. Der Standard spezifiziert ein Metamodell, das die Modellierung von Steuergerätearchitekturen erlaubt. Des Weiteren wird dort ein einheitliches Austauschformat, sowie eine eigene Entwicklungsmethodik vorgegeben (Kindel und Friedrich, 2009). Das Metamodell von AUTOSAR unterstützt die Wiederverwendung von Softwarekomponenten und ermöglicht es in Zusammenspiel mit der Entwicklungsmethodik, verteilte Softwarearchitekturen zunächst hardwareunabhängig, d.h. beispielsweise ohne die konkrete Zuweisung von Steuergerätekomponenten auf Steuergerätehardware, zu modellieren und erst später um hardware-abhängige Eigenschaften zu erweitern (Kindel und Friedrich, 2009). So können bereits komplexe verteilte Steuergerätearchitekturen modelliert und simuliert werden ohne das finale Wissen über die Hardware zu haben.

AUTOSAR

Neben der Modellierung der Systemarchitektur unterstützt AUTOSAR ebenfalls die Modellierung von *Timing Anforderungen*. Diese ermöglichen es Anforderungen an das Zeitverhalten eines AUTOSAR Modells festzuhalten, indem Teile des Systemmodells mit sogenannten *Timing Events* verknüpft werden, deren zeitliches Auftreten in einem laufenden System mit Constraints belegt werden können. So können Echtzeitanforderungen eines Steuergeräts in der Architektur mit aufgenommen werden und für Analysen herangezogen werden (AUTOSAR, 2019g).

Timing Anforderungen

Timing Events

Software im Automobil wird üblicherweise nach dem *V-Modell* entwickelt, welches einen Prozess mit fest definierten Phasen beinhaltet und ein

V-Modell

Vorgehen beginnend bei der Definition der *Benutzeranforderungen* bis zum Test des Systems beschreibt (Boehm, 1979). Werden für das zu entwickelnde System Anforderungen identifiziert, die das zeitliche Verhalten beschreiben oder einschränken, so müssen diese ebenfalls in allen folgenden Phasen nachgehalten und schließlich auch verifiziert werden. Standards wie die *ISO 26262* (ISO International Organisation for Standardisation, 2018) verlangen dabei explizit die Festlegung des Echtzeitverhaltens der Software (Schäuffele und Zurawka, 2010). Alle im Prozess verwenden Methoden, Sprachen und Werkzeuge müssen das Echtzeitverhalten mit abbilden und die Nachverfolgbarkeit bis zur Echtzeitanforderung sicherstellen können. Dabei gilt, je später im Prozess Fehler in Modellen gefunden werden, desto aufwändiger ist die Korrektur, da möglicherweise alle Modelle der vorherigen Phasen korrigiert werden müssen. Für eine effiziente Systementwicklung ist es daher vorteilhaft, wenn in allen Phasen der Systementwicklung bereits von Anfang an eine *hohe Qualität* der dort erstellten Modelle sichergestellt ist. So werden Fehler früher erkannt und die folgenden Phasen arbeiten mit korrekten Modellen, sodass unnötige Iterationen vermieden werden. Dadurch wird der gesamte Entwicklungsprozess beschleunigt, wodurch die Kosten und das Risiko für die Entwicklung gesenkt werden (Schäuffele und Zurawka, 2010). Auf der Basis dieses Leitparadigmas lassen sich in Verbindung mit den existierenden Technologien konkrete Problemstellungen identifizieren.

Benutzeranforderungen

ISO 26262

Modellqualität

1.2 Problemstellung

Die Technologie der Software-In-The-Loop (SIL)-Tests (Kempkes und Walther, 2018) ermöglicht es Steuergeräte auf der Grundlage der AUTOSAR-Architektur und der kompilierten Verhaltensmodelle zu simulieren. Für die Durchführung einer Simulation müssen jedoch eine Reihe von Modellen erstellt werden. Abbildung 1.1 zeigt eine Übersicht über diese Modelle: Neben der Erstellung der Softwarearchitektur (oben Mitte) muss die vollständige Steuergerätesoftware in das AUTOSAR Modell integriert werden (unten links). Dafür müssen zunächst Verhaltensmodelle - häufig auf der Basis von

MATLAB/Simulink - erzeugt werden (oben rechts). Diese Modelle spiegeln das Verhalten beispielsweise eines Reglers wider, enthalten aber noch keine Modellinformationen über die Implementierung auf einem späteren Steuergerät wie beispielsweise die verwendeten Datentypen der Modellvariablen. Diese Informationen werden im nächsten Schritt annotiert. Erst dann kann aus den Modellen Steuergerätecode generiert werden, der dann in die AUTOSAR Architektur integriert wird. Schließlich muss aus einer vorhandenen AUTOSAR-Architektur heraus ein virtuelles Steuergerät (V-ECU) konfiguriert, generiert und kompiliert werden.

HOHER ZEITAUFWAND

Sowohl die Schritte der Funktionsentwicklung als auch die anschließende Konfiguration und Generierung der V-ECU sind zeitaufwändig . Zudem sind in frühen Entwicklungsphasen die Modelle aus der Funktionsentwicklung noch nicht verfügbar. Aus diesem Grund ist es für eine frühzeitige Validierung von AUTOSAR Softwarearchitekturen hilfreich Methoden zu betrachten, die ausschließlich auf den Artefakten der Softwarearchitektur arbeiten und nicht auf das Vorhandensein von Steuergerätecode angewiesen sind.

SPÄTE VERFÜGBARKEIT

Des Weiteren lassen sich mit dem bisherigen Vorgehen funktionale Anforderungen durch die Simulation des Steuergeräts gut testen. Allerdings ist dieses Vorgehen für Timing Anforderungen ungeeignet, da diese unter allen möglichen Umständen eingehalten werden müssen. Die Validierung von Timing Anforderungen erfordert daher den Einsatz von *Verifikationsmethoden*, die alle möglichen Randfälle mitbetrachten.

VERIFIKATIONSMETHODEN

Eine weitere Herausforderung bei der Validierung von AUTOSAR Timing Anforderungen betrifft bereits die Spezifikation der Anforderungen: Diese werden auf der Basis der Echtzeitanforderungen auf Benutzerebene hergeleitet (siehe orange Artefakte in Abbildung 1.1). Gerade bei großen Anforderungsmengen kann es für Anforderungsentwickler jedoch schwierig sein, den Überblick über die Anforderungen zu behalten, was zu einer schlechten *Anforderungsqualität* führt und beispielsweise Inkonsistenzen nach sich ziehen kann. Inkonsistente Anforderungsmengen sind ein Indikator für Missverständnisse bei der erwarteten Systemfunktionalität. Des Weiteren führt die Verifikation von inkonsistenten Anforderungsmengen zu unnötigen Verifikationsläufen, da die Verifikation einiger Anforderun-

ANFORDERUNGSQUALITÄT

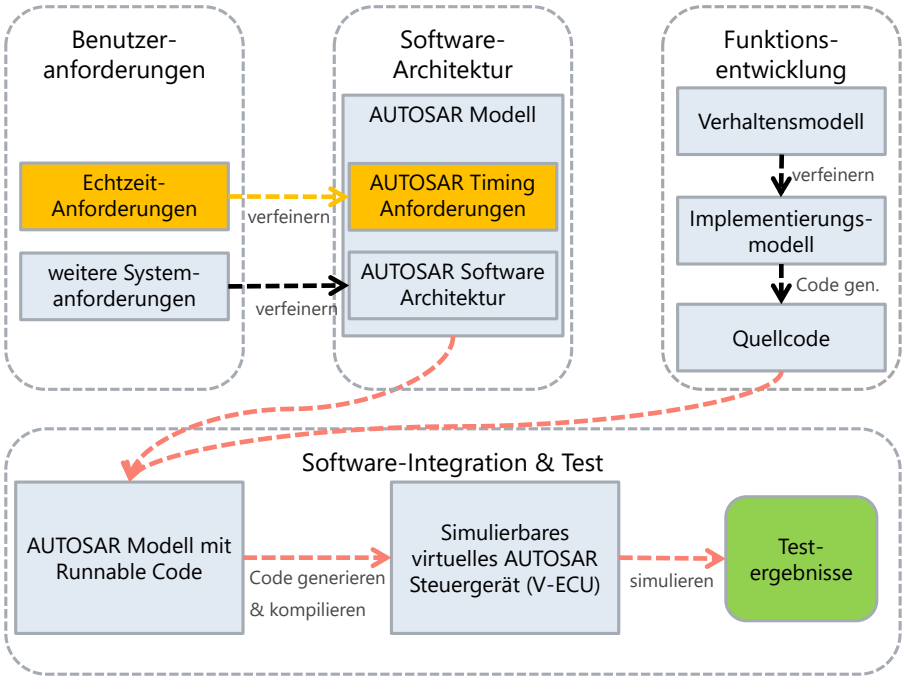

Abbildung 1.1: Herausforderungen bei der Entwicklung von Steuergerätesoftware: die Herleitung formaler AUTOSAR Timing Anforderungen ist aufgrund der großen Lücke hinsichtlich Abstraktion und Formalisierung fehleranfällig (orange); der existierende Prozess benötigt alle Artefakte aus der Funktionsentwicklung (oben rechts), sodass Testergebnisse erst spät vorliegen (grün).

gen unweigerlich fehlschlagen würden, was einen großen Einfluss auf die Effizienz der Entwicklung hat. Daher ist es vorteilhaft, wenn Inkonsistenzen in den Timing Anforderungen frühzeitig erkannt und behoben werden, sodass keine unnötigen Timing Verifikationen durchgeführt werden.

Insgesamt werden die folgenden Anforderungen an die Timing Verifikation von AUTOSAR Softwarearchitekturen gestellt:

1. **Anwendung von Verifikationstechniken für Timing Anforderungen.** Timing Anforderungen lassen sich nicht mithilfe von Simulation verifizieren, da diese in allen möglichen Situationen eingehalten werden müssen. Die Verifikation von Timing Anforderungen erfordert daher den Einsatz von Verifikationsmethoden, die alle möglichen

Randfälle mitbetrachten.

2. **Verifikation von AUTOSAR Timing Anforderungen ohne Steuergerätecode.** Um frühzeitig im Entwicklungsprozess Aussagen über die Korrektheit von AUTOSAR Timing Anforderungen zu erhalten, ist es notwendig eine Verifikationsmethode zu entwickeln, die ausschließlich auf der Grundlage der definierten AUTOSAR Steuergerätearchitektur arbeitet und somit auch ohne Quellcode oder Verhaltensmodelle anwendbar ist.

3. **Frühzeitige Rückmeldung über die Qualität der Timing Anforderungen.** Das Herleiten von Timing Anforderungen auf AUTOSAR-Architekturebene aus Benutzeranforderungen ist fehleranfällig. Daher ist es vorteilhaft, wenn Inkonsistenzen in den Timing Anforderungen frühzeitig erkannt werden.

1.3 Beitrag der Arbeit

Die Arbeit verfolgt das Ziel, den Prozess der Softwareentwicklung in der Automobilindustrie durch Methoden zu verbessern, mit denen Timing Anforderungen einfacher und schneller erstellt und bereits frühzeitig auf Konsistenz- und Korrektheitseigenschaften hin überprüft werden können. Dabei berücksichtigen die Methoden den existierenden AUTOSAR-Standard und lassen sich nahtlos in den Gesamtprozess der automotiven Systementwicklung nach dem V-Modell integrieren (Bundesstelle für Informationstechnik, 2012). Insgesamt werden die folgenden Kernaspekte behandelt:

Verifikation

1. **Erarbeitung einer Methode zur *Verifikation* von AUTOSAR Timing Anforderungen**: Für die Verifikation von AUTOSAR Timing Anforderungen wird eine Transformation des AUTOSAR Modells nach Timed Automata spezifiziert. Diese überführt das Timing Verhalten eines AUTOSAR-Modells in Timed Automata, sodass sich Timing Anforderungen mithilfe existierender Werkzeuge veri-

fizieren lassen. Da es für das AUTOSAR-Metamodell keine formale Verhaltensbeschreibung gibt, muss diese vorab erstellt werden.

2. **Erarbeitung einer Methode zur *Konsistenzanalyse* von AUTOSAR Timing Anforderungen**: Die Methode stellt sicher, dass die zu verifizierenden Timing Anforderungen untereinander keine temporalen Konflikte beinhalten. Hierfür wird eine Transformation der AUTOSAR Timing Constraints nach SMT spezifiziert, sowie Mechanismen zur Identifikation von Ursachen für Inkonsistenzen bereitgestellt. KONSISTENZANALYSE

3. **Erarbeitung von Maßnahmen zur *Auflösung von Inkonsistenzen* für AUTOSAR Timing Constraints**: Für inkonsistente Anforderungsmengen werden relevante Teilmengen berechnet und mithilfe von Graphen visualisiert, um die Ursachen für inkonsistente Anforderungsmengen identifizieren zu können und Möglichkeiten zur Auflösung dieser anbieten zu können. AUFLÖSUNG VON INKONSISTENZEN

4. ***Evaluierung* der Methoden anhand eines praxisnahen Fallbeispiels**: Für die Evaluierung der entwickelten Methoden wird neben der Messung von Laufzeiten für synthetische AUTOSAR-Modelle ebenfalls ein realitätsnahes Beispiel herangezogen, mit dem sich die Anwendbarkeit und der Nutzen bei realen Modellen bewerten lässt. EVALUIERUNG

Abbildung 1.2 zeigt einen Überblick über den Lösungsansatz dieser Arbeit.

1.4 Struktur der Arbeit

Die Arbeit ist wie folgt gegliedert: Zunächst werden in *Kapitel 2* die Grundlagen eingeführt. Es wird der Entwicklungsprozess in der Automobilindustrie, sowie der AUTOSAR Standard vorgestellt. Weiterhin werden Grundlagen von Timing Analysen, sowie Timing Anforderungen vorgestellt. Des Weiteren werden die Formalen Methoden vorgestellt, die im weiteren Verlauf der Arbeit angewandt werden. KAPITEL 2

In *Kapitel 3* beschreiben wir den integrierte Lösungsansatz. Des Weiteren werden die erarbeiteten Methoden in den Entwicklungsprozess eingeord- KAPITEL 3

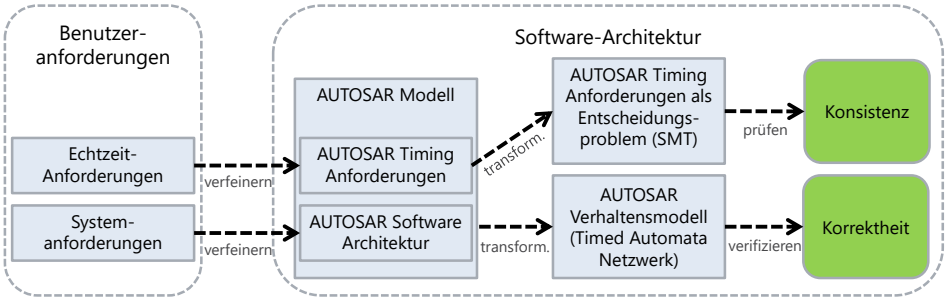

Abbildung 1.2: Lösungsansatz: Die entwickelten Methoden ermöglichen es Anforderungen auf Konsistenz und Korrektheit zu überprüfen. Auf die Artefakte aus der Funktionsentwicklung kann dabei verzichtet werden.

net und alternative Möglichkeiten zur Integration des Ansatzes in den Entwicklungsprozess aufgezeigt. Weiterhin wird in diesem Kapitel das formale Modell des AUTOSAR Metamodells vorgestellt, welches in den darauf folgenden Kapiteln Anwendung findet.

Kapitel 4 In *Kapitel 4* stellen wir das Konzept zur Konsistenzanalyse von AUTOSAR Timing Anforderungen vor. Es wird zunächst die für die Konsistenzanalyse notwendige Transformation der AUTOSAR Timing Anforderungen nach SMT vorgestellt. Anschließend werden Methoden zur Darstellung und Korrektur inkonsistenter Anforderungsmengen vorgestellt.

Kapitel 5 *Kapitel 5* betrachtet die entwickelte Methode zur Timing Analyse von AUTOSAR Timing Anforderungen. Es werden die für die Timing Analyse relevanten Transformationen der AUTOSAR Konstrukte nach Timed Automata vorgestellt. Dies beinhaltet zum einen den timing-relevanten Teil der AUTOSAR Softwarearchitektur als auch die Timing Anforderungen.

Kapitel 6 In *Kapitel 6* werden die zuvor vorgestellten Methoden an einem detaillierten Anwendungsfall angewandt. Das Fallbeispiel wird erläutert und die Ergebnisse werden vorgestellt. Anhand dieser Ergebnisse wird dann diskutiert, in wie weit sich die Methoden für die Anwendung in der Praxis eignen.

Kapitel 7 *Kapitel 7* gibt einen Einblick in die Realisierung der Methoden durch eine prototypischen Werkzeugunterstützung. Es wird die allgemeine Struktur

des entwickelten Analyseframeworks vorgestellt, sowie auf einige spezifische Realisierungsaspekte eingegangen. Weiterhin werden detaillierte Evaluierungsergebnisse vorgestellt, um die praktische Anwendbarkeit der Methode einschätzen zu können.

Kapitel 8 fasst die Ergebnisse zusammen und diskutiert die erzielten Erkenntnisse. Es gibt außerdem einen Ausblick auf offene und zukünftige Problemstellungen und mögliche Lösungsansätze. KAPITEL 8

2 Grundlagen

Die Komplexität von eingebetteten Systemen im Automobil steigt beständig aufgrund der Integration neuer und komplexerer Fahrfunktionen. Insbesondere die *Vernetzung* der einzelnen Assistenzsysteme und die Verwendung einer Vielzahl von Sensoren und die daraus resultierenden zu verarbeitenden Daten in autonomen Fahrzeugen erhöht die Komplexität weiter. Gleichzeitig werden die *Entwicklungszyklen* neuer Modellreihen jedoch immer weiter verkürzt, um *Kosten* zu senken und Produkte schneller auf den Markt zu bringen. Dabei bleiben die grundlegenden Anforderungen an die Sicherheit der Systeme unter Berücksichtigung des Echtzeitverhaltens bestehen.

Vernetzung

Entwicklungszyklen

Kosten

Nur durch die umfassende Anwendung von Maßnahmen zur *Software-Qualitätssicherung* ist es möglich, schneller und frühzeitiger Fehler während der Entwicklung aufzudecken und somit die Softwarequalität auch bei kurzen Entwicklungszyklen zu steigern. Diese Maßnahmen bestehen aus einer einheitlichen Entwicklung anhand eines Prozesses, der auch über Unternehmensgrenzen hinweg etabliert ist, der Anwendung konstruktiver Methoden wie *Domänenspezifischen Sprachen* und Architekturpattern, sowie systematischer statischer und dynamischer Analyseverfahren während aller Entwicklungsphasen.

Software-Qualitätssicherung

Domänenspezifischen Sprachen

Dieses Kapitel beschreibt die Grundlagen dieser Arbeit. Zunächst werden die spezifischen Prozesse und Methoden der Softwareentwicklung und -qualitätssicherung in der Automobilindustrie beschrieben. Dies ist zum einen das verwendete V-Modell und zum anderen der AUTOSAR Standard, sowie die in dieser Arbeit verwendete AUTOSAR Entwicklungsumgebung. Weiterhin wird in diesem Kapitel ein Beispiel eingeführt, dass im weiteren Verlauf der Arbeit zur Verdeutlichung der entwickelten Methoden herangezogen wird. Des Weiteren werden die verschiedenen Ausprägungen von Timing Analysen vorgestellt und eingeordnet. Ebenfalls werden Sprachen zur Spezifikation von Timing Anforderungen vorgestellt und Kriterien zur Beschreibung der Qualität aufgezeigt. Schließlich werden die grundlegenden Definitionen der verwendeten formalen Analysemethoden erläutert und in den Gesamtkontext der Arbeit eingeordnet.

2.1 Automotive Softwareentwicklung

Dieser Abschnitt gibt einen Überblick über die Entwicklung von Steuergerätesoftware. Es wird dabei zunächst das Vorgehensmodell vorgestellt, das den Kernprozess zur Entwicklung von elektronischen Systemen und Software bereitstellt. Darauf aufbauend wird der AUTOSAR Standard eingeführt, der den Prozess durch eine eigene Methodik und ein Metamodell unterstützt. An dieser Stelle wird besonders auf die für diese Arbeit wichtigen Modellelemente zur Spezifikation von Timing Anforderungen eingegangen. Anschließend werden existierende Methoden zur Timing Analyse vorgestellt, sowie Werkzeuge für die Modellierung von AUTOSAR im Allgemeinen und Werkzeuge zur Analyse von Timing Anforderungen im Speziellen.

2.1.1 Vorgehensmodell und Artefakte

Softwarekrise

Komplexe technische Systeme erfordern ein planvolles Vorgehen bei der Entwicklung angefangen bei der Definition von Anforderungen bis zum abschließenden Systemtest. In der sogenannten *Softwarekrise* in den 60er Jahren erkannte man, dass die bisher genutzten Methoden zur Entwicklung

von Software nicht mit der Rechenleistung und Komplexität der verfügbaren Hardware mithalten konnten (Dijkstra, 1972). So waren beispielsweise die Kosten für die Entwicklung des Betriebssystems *OS/360* von IBM um ein vielfaches höher als zunächst geplant (Brooks, 1975).

Seitdem wurden vermehrt Methoden und Technologien entwickelt, die die Entwicklung von Software einfacher und vorhersagbarer machen und somit das Risiko von Fehlentwicklungen und Fehleinschätzungen von Kosten verringern sollten. Neben der Entwicklung neuer Programmierparadigmen entstanden seitdem *Vorgehensmodelle*, die die Entwicklung komplexer Software einfacher und planbarer gestalten sollten, indem für die Entwicklung einzelne fest definierte Schritte oder Phasen festgelegt wurden und in einem Prozess angeordnet wurden. Eine konkrete Definition findet sich in IEEE (1990):

Vorgehensmodelle

Definition 1 (Vorgehensmodell (IEEE, 1990))**.** *Der Prozess, in dem Nutzerbedürfnisse in ein Softwareprodukt umgesetzt werden. Der Prozess umfasst das Übersetzen der Benutzeranforderungen in Softwareanforderungen, die Umwandlung der Softwareanforderungen in ein Design, die Implementierung des Designs in Code, das Testen des Codes und manchmal die Installation und das Überprüfen der Software für den betrieblichen Einsatz. Diese Aktivitäten können sich überschneiden oder iterativ durchgeführt werden.*

Beispiele für Vorgehensmodelle sind das Wasserfallmodell (Royce, 1970), das V-Modell (Boehm, 1979), das Spiralmodell (Boehm, 1988), der Rational Unified Process (Rational Software, 1998), Extreme Programming (Beck und Andres, 2005) oder Scrum (Beck et al., 2001, Cohn, 2010).

Elektronische Systeme und Software in der Automobilindustrie werden heutzutage üblicherweise nach dem *V-Modell* entwickelt. Das V-Modell wurde ursprünglich von Boehm (1979) als sequenzielles Vorgehensmodell entworfen und im Laufe der Zeit weiterentwickelt. Die heute aktuelle Ausprägung ist das *V-Modell® XT* der Bundesstelle für Informationstechnik (2012). Das Gesamtmodell ist in Abbildung 2.1 zu sehen. Es beinhaltet mehrere Schritte und beginnt bei der Analyse von Nutzeranforderungen und endet beim finalen Systemtest und Akzeptanztest. Das Modell kann dabei in einen oberen Teil und einen unteren Teil aufgeteilt werden.

V-Modell

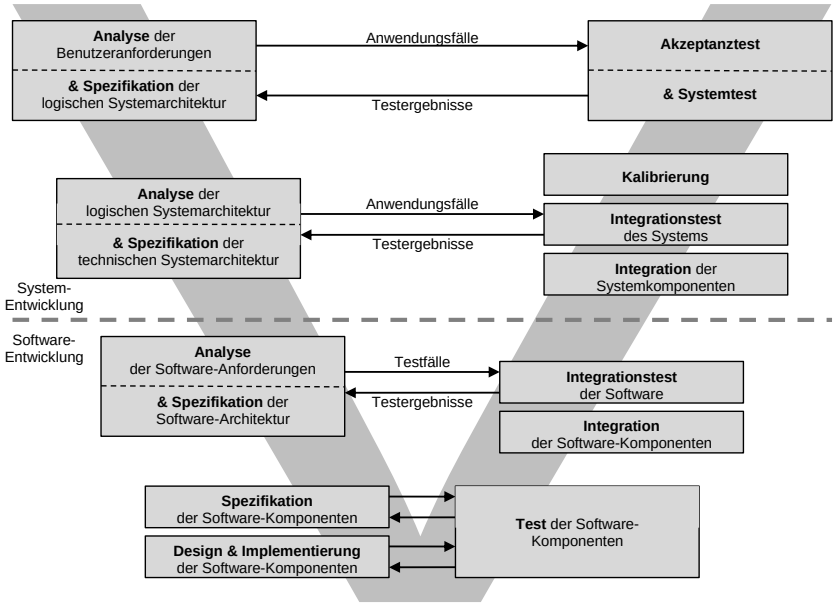

Abbildung 2.1: V-Modell Entwicklungsprozess nach Schäuffele und Zurawka (2010)

Der obere Teil beschäftigt sich mit dem Design und der Analyse der gesamten Systemarchitektur. Dies bedeutet insbesondere, dass diese Schritte unabhängig von jeglichen technischen Realisierungsaspekten betrachtet werden. Danach wird der Prozess im unteren Teil aufgeteilt in die Teilbereiche *Softwareentwicklung*, *Hardwareentwicklung*, *Aktuatorentwicklung* und *Sensorentwicklung* (Schäuffele und Zurawka, 2010). Da sich der entwickelte Ansatz auf die Phasen der Softwareentwicklung bezieht, werden ausschließlich die relevanten Phasen der Softwareentwicklung vorgestellt.

Designphasen
Validierungsphasen

Des Weiteren lässt sich das Modell in einen linken Zweig, welcher die *Designphasen* beinhaltet, in denen das System schrittweise aufgeteilt und verfeinert wird, und einen rechten Zweig, welcher die *Validierungsphasen* beinhaltet, die die jeweilige Phase auf der rechten Seite absichert, aufteilen.

Wenn ein Test in einer Phase auf der rechten Seite fehlschlägt, so wird das Design entsprechend der Phase der linken Seite verbessert. Da jede Design-

phase auf den Ergebnissen der vorherigen Phase beruht, lösen Änderungen in einer Phase normalerweise auch Änderungen in den darauffolgenden Phasen aus. Daher sind Fehler, die erst in späten Testphasen gefunden werden sehr teuer, da sie zu früheren Designphasen gehören und dadurch mehr Änderungen hervorrufen.

Eine Lösung, um das späte Erkennen von Fehlern zu vermeiden ist es, Tests späterer Phasen in frühere Phasen vorzuverlagern. Dies kann beispielsweise durch die Simulation von Hardware, die noch nicht zur Verfügung steht, erreicht werden. Diese Methode wird auch *Test Frontloading* genannt (Stark et al., 2011, Klein et al., 2017). Ein weiterer Schritt das späte Erkennen von Fehlern zu vermeiden ist es, bereits in frühen Designphasen Methoden zur Fehlererkennung anzuwenden. In diesen Phasen sind jedoch nur Softwaremodelle, d.h. Architekturmodelle und Reglermodelle, vorhanden. Daher können nur Validierungstechniken angewendet werden, die ausschließlich auf diesen Modellen beruhen.

Test Frontloading

Da das Ziel dieser Arbeit die Entwicklung von Methoden zur frühzeitigen Absicherung von automotiver Steuergerätesoftware ist, basieren die genutzten Artefakte ausschließlich auf Modellen, die bereits in frühen Phasen der Softwareentwicklung zur Verfügung stehen. Dies sind im Wesentlichen die in der automotiven Softwareentwicklung gebräuchlichen *Architekturmodelle*, welche auf dem AUTOSAR Standard basieren.

Architekturmodelle

2.1.2 AUTOSAR

AUTOSAR* steht für **AUT**omotive **O**pen **S**ystem **AR**chitecture und ist der etablierte Standard für die Entwicklung von Software in der Automobilindustrie. AUTOSAR definiert eine Softwarearchitektur und Interfaces in Form eines Metamodells (siehe dafür Abschnitt *Metamodellhierarchie*) , sowie ein eigenes Dateiformat für den Datenaustausch. Weiterhin definiert der Standard eine eigene Entwicklungsmethodik. Seit 2017 veröffentlicht die AUTOSAR Entwicklungspartnerschaft ebenfalls einen Standard zur Spezifikation dynamisch adaptierbarer Services innerhalb einer Softwarearchitektur im Automobil. Dieser Standard wird *Adaptive Platform* genannt und

Adaptive Platform

*http://www.autosar.org

existiert parallel neben dem bereits existierenden AUTOSAR Standard, der seitdem ebenfalls als *Classic Platform* bezeichnet wird. Im Folgenden werden ausschließlich Konzepte der Classic Platform genauer betrachtet.

Classic Platform

AUTOSAR Schichtenarchitektur

AUTOSAR definiert eine Schichtenarchitektur für Steuergeräte (siehe Abbildung 2.2), die drei verschiedene Softwareschichten beinhaltet (AUTOSAR, 2019d):

Application Layer

- Die Applikationsschicht (*Application Layer*) ist die oberste Schicht (grau). Sie beinhaltet die auszuführende Steuergerätesoftware, was in der Automobilindustrie zumeist die Implementierung der Regleralgorithmen ist. Innerhalb dieser Schicht wird die Software durch eine komponentenbasierte Architektur weiter strukturiert.

Runtime Environment Layer (RTE)

- Die RTE-Schicht (*Runtime Environment Layer (RTE)*) administriert die Kommunikation zwischen Softwarekomponenten der Applikationsschicht untereinander und zwischen Softwarekomponenten der Applikationsschicht und Basissoftwaremodulen der Basissoftwareschicht (orange). Sie stellt somit standardisierte Interfaces für die Software auf Applikationsebene zur Verfügung.

Basic Software Layer

- Die Basissoftwareschicht (*Basic Software Layer*) beinhaltet Module für die Basisfunktionalitäten eines Steuergerätes. Die Basissoftwareschicht ist weiterhin unterteilt in eine Serviceschicht (*Service Layer*) (blau), eine ECU-Abstraktionsschicht (*ECU Abstraction Layer*) (grün) und eine Mikrocontroller-Abstraktionsschicht (*Microcontroller Abstraction Layer, MCAL*) (rot). Der *Service Layer* beinhaltet die wesentlichen Steuergeräteservices wie beispielsweise das Betriebssystem, das Zustandsmanagement, Diagnoseservices, Speicherservices und Kommunikationsservices. Der *ECU Abstraction Layer* realisiert eine Abstraktion der Steuergerätehardware für die oberen Schichten und stellt Module für den Zugriff auf die Hardwareperipherie bereit. Der *MCAL* stellt Treibermodule für das Steuergerät zur Verfügung und greift direkt auf die Hardware

Service Layer

ECU Abstraction Layer

MCAL

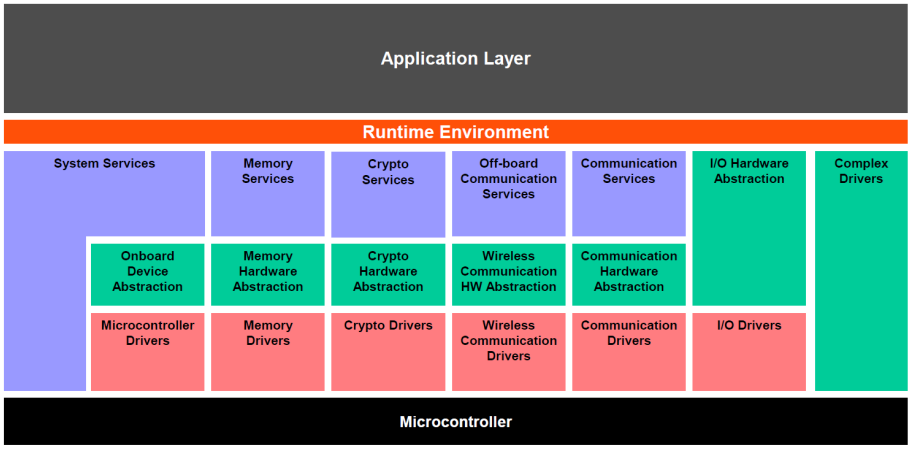

Abbildung 2.2: AUTOSAR Schichtenarchitektur aus AUTOSAR (2019d)

zu. Zusätzlich zu der horizontalen Strukturierung werden die Basissoftwaremodule auf vertikaler Ebene nach Funktionsbereichen eingeteilt. Diese Bereiche teilen sich auf in Systemdienste (*System Services*), Speicherdienste (*Memory Services*), Kryptographiedienste (*Crypto Services*) Kommunikationsdienste (*Communication Services*), I/O-Stack (*I/O Hardware Abstraction*) und komplexe Gerätetreiber (*Complex Device Drivers*). Eine detaillierte Beschreibung aller Module kann in AUTOSAR (2019g) gefunden werden.

AUTOSAR Entwicklungsmethodik

Die AUTOSAR Entwicklungsmethodik beschreibt die wesentlichen Schritte für AUTOSAR Entwicklungsprojekte angefangen bei der Definition einer abstrakten Softwarearchitektur bis zur Generierung der ausführbaren Steuergerätesoftware (Kindel und Friedrich, 2009, AUTOSAR, 2019e). Die Methodik ist dabei in formaler Form als *SPEM*-Modell der Object Management Group (2008) festgehalten. Sie beschreibt Aufgaben, Rollendefinitionen, Werkzeuge und Arbeitsprodukte und setzt sie in Form von Workflows zueinander in Beziehung. Sie ist daher nicht mit einem Entwicklungsprozess zu vergleichen, da beispielsweise keine Restriktionen hinsichtlich der Reihenfolge von Workflows beschrieben werden oder ob und wann Itera-

SPEM

tionen durchgeführt werden. Das Ziel der Methodik ist es, die Entwicklung zu parallelisieren und somit zu verkürzen. Die Entwicklungsmethodik ist in fünf *Entwicklungssichten* aufgeteilt, die sich jeweils mit einer Teilmenge der spezifizierten Workflows beschäftigen: (AUTOSAR, 2019e):

Entwicklungssichten

1. *Virtual Functional Bus* (VFB): In der VFB-Sicht wird die Softwarearchitektur eines Systems modelliert. Die Softwarearchitektur besteht aus den *Softwarekomponenten* der Applikationsschicht, die über Ports miteinander verbunden sind. Für *Ports* können dann wiederum Interfaces definiert werden, beispielsweise Client-Server- oder Sender-Receiver-Interfaces. Bei der Modellierung in dieser Sicht ist es unerheblich auf welchen Steuergeräten die Komponenten später ausgeführt werden. Die Kommunikation der Komponenten untereinander wird durch Verbindungen über den Virtual Functional Bus abstrahiert. Die Spezifikation der Hardwaretopologie und die Verteilung von Softwarekomponenten auf einzelne Steuergeräte kann zu einem späteren Zeitpunkt erfolgen. Eine detaillierte Beschreibung findet sich in AUTOSAR (2019h).

Softwarekomponenten

Ports

2. *System*: In der System-Sicht wird das Gesamtsystem modelliert. Ein zentraler Teil ist hierbei die Beschreibung welche Softwarekomponenten auf welche Steuergeräte verteilt sind und wie diese wiederum miteinander verbunden sind. Das Ergebnis ist ein *ECU Extract*, das wiederum als Eingabe für die Integration der ECU Software in der ECU-Sicht benötigt wird. Die vollständige Spezifikation ist in AUTOSAR (2019h) beschrieben.

ECU Extract

3. *Software Component*: Auf Softwarekomponenten-Ebene wird die Struktur und das Verhalten der einzelnen Softwarekomponenten der VFB-Ebene definiert. Es werden sogenannte *Runnable Entities* definiert, in denen der ausführbare Reglercode gekapselt ist. Eingabe- und Ausgabedaten werden über Ports und innerhalb einer Softwarekomponente über Inter-Runnable-Variablen festgelegt. Des Weiteren wird festgelegt, wann und in welchen Zyklen Runnables ausgeführt werden sollen (AUTOSAR, 2019f).

Runnable Entities

4. *Basic Software*: In der Basissoftwaresicht werden für einzelne Steuergeräte die *Basissoftwaremodule* konfiguriert und Quellcode der Module im Modell integriert. Zusammen mit dem ECU Extract kann dann in der ECU-Sicht die Integration der ECU Software erfolgen (AUTOSAR, 2019g). Basissoftwaremodule

5. *ECU*: Die ECU-Sicht beschreibt die Sicht auf ein einzelnes Steuergerät. Wenn auf Systemebene die definierten Softwarekomponenten auf Steuergeräte verteilt wurden, alle Basissoftwarekomponenten in der Basissoftware-Sicht konfiguriert wurden und Applikationssoftwarekomponenten der Softwarekomponentensicht fertig spezifiziert wurden, dann kann in dieser Sicht aus den abstrakten Verbindungen der Softwarekomponenten auf dem VFB-Bus eine RTE generiert werden, die je nach Position der Softwarekomponenten interne Speicheraufrufe oder das Verschicken von Nachrichten über einen externen Kommunikationsbus durchführt. Das Ergebnis ist dann die ausführbare Steuergerätesoftware (AUTOSAR, 2019b).

AUTOSAR Metamodell Hierarchie

Neben der Entwicklungsmethodik enthält AUTOSAR eine formale Beschreibung zur Definition von AUTOSAR-konformen Modellen. Diese *Modelle* sind „eine Abstraktion eines realen Systems, das Vorhersagen oder Schlussfolgerungen ermöglicht“(Kühne, 2006). Modelle verfügen nach Stachowiak (1973) über die folgenden Eigenschaften: Modelle

- Abbildungseigenschaft: Ein Modell ist ein Abbild der Wirklichkeit. Eigenschaften, die für das Modell gelten, gelten auch für die Entität in der realen Welt.

- Abstraktions- oder Reduktionseigenschaft: Ein Modell erfasst nur eine Teilmenge von Attributen des Originals. Details, die für das Modell nicht benötigt werden, werden weggelassen.

- Pragmatismus-Eigenschaft: Ein Modell dient einem bestimmten Zweck. Die Erstellung des Modells erfolgt zielorientiert.

Im Kontext der Modellgetriebenen Softwareentwicklung wird die Spezifikation der Modellelemente und Regeln, die benötigt werden um valide Modelle zu erzeugen, auch *Metamodell* genannt. Dieses beinhaltet üblicherweise die Definition einer abstrakten Sytax, (mindestens einer) konkreten Syntax, sowie statischer und dynamischer Semantik (Stahl und Völter, 2006). Die abstrakte Syntax beschreibt dabei die abstrakten Modellierungselemente und ihre Beziehungen zueinander, während die konkrete Syntax die Repräsentation von Modellinstanzen beschreibt, so dass diese von einem Parser akzeptiert werden. Die statische Semantik enthält Regeln, die die Wohlgeformtheit eines Modells zusätzlich beschreiben. Die dynamische Semantik beschreibt das Verhalten des Modells bei der Ausführung (Stahl und Völter, 2006). Eine konkrete Technologie zur Spezifikation von Metamodellen wird von der Object Management Group (2011b) vorgeschlagen. Diese definiert eine Metamodell Hierarchie mit vier Ebenen ($\mathcal{M}_0$ - $\mathcal{M}_3$), wobei jede Ebene die Sprachkonstrukte der darunterliegenden Ebene beschreibt bzw. umgekehrt jede Ebene eine konkrete Instanz des Typmodells auf der darüberliegenden Ebene darstellt. Die Ebene $\mathcal{M}_0$ stellt dann das reale System dar, auf der Ebene $\mathcal{M}_1$ befinden sich die von einem Entwickler erzeugten Modelle, die Ebene $\mathcal{M}_2$ beschreibt die zur Erzeugung von Modellen notwendigen Sprachkonstrukte bzw. das Metamodell und schließlich befindet sich auf Ebene $\mathcal{M}_3$ das Meta-Meta-Modell, dass zur Spezifikation von Metamodellen herangezogen werden kann. Dieses wird auch *Meta Object Facility (MOF)* genannt. Neben der Definition eigener Metamodelle durch die Instanziierung eines MOF-Modells lässt sich auch das UML-Metamodell durch sogenannte *UML-Profile* um Stereotypen, Tagged Values und Constraints erweitern (Stahl und Völter, 2006). Dieser Mechanismus ist einfacher zu handhaben, weil auf das existierende UML-Metamodell zurückgegriffen werden kann. Die Anpassbarkeit ist jedoch nicht so stark gegeben, da beispielsweise keine Entitäten des UML-Metamodells entfernt werden können.

Metamodell

Meta Object Facility (MOF)

UML-Profile

Die Syntax von AUTOSAR bedient sich ebenfalls der Spezifikationen der OMG. Die vollständige Metamodell Hierarchie besteht jedoch aus insgesamt fünf Metaebenen (siehe Abbildung 2.3) (AUTOSAR, 2019c). Die Ebenen $\mathcal{M}_0$ und $\mathcal{M}_1$ sind analog zu den Definitionen der *UML* (Ob-

UML

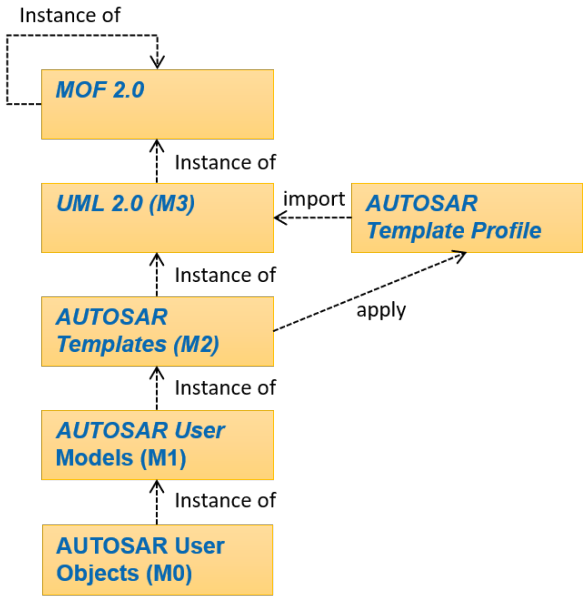

Abbildung 2.3: Metamodellebenen von AUTOSAR (AUTOSAR, 2019c)

ject Management Group, 2015) definiert als Realisation des laufenden AUTOSAR-Systems und als Modelle, die von AUTOSAR-Entwicklern erstellt werden wie beispielsweise Softwarekomponenten, Ports und deren Verbindungen (AUTOSAR, 2019c). Diese Modelle sind Instanzen des AUTOSAR-Metamodells auf $\mathcal{M}_2$. Hier sind die Metamodellelemente wie Ports und Softwarekomponenten spezifiziert. Das AUTOSAR Metamodell wird auch als Menge sogenannter *AUTOSAR Templates* bezeichnet, da das Metamodell nicht ausschließlich eine Instanz des UML Metamodells ist, sondern ebenfalls ein UML Profil, das sogenannte *AUTOSARTemplateProfile* anwendet. Dieses wurde entworfen, um besser Templating Mechanismen umsetzen zu können. Es beinhaltet beispielsweise die Definitionen von *Types*, *Prototypes* und der Assoziation vom Typ *isTypeOf*. Formal ist ein Template auf $\mathcal{M}_2$ daher eine Instanz vom UML 2 Metamodell bei gleichzeitiger Anwendung des AUTOSAR Template Profils (AUTOSAR, 2019e). Das UML Metamodell, sowie das AUTOSAR Template Profil sind daher auf der Ebene $\mathcal{M}_3$ verortet. Das MOF-Metamodell ist dann übergeordnet auf Ebene $\mathcal{M}_4$ zu finden. Im Gegensatz zu anderen domänenspezifischen

AUTOSAR Templates

AUTOSARTemplateProfile

Sprachen definiert AUTOSAR für das Metamodell keine konkrete Syntax für alle Modellelemente. Es werden nur für Modellelemente der Applikationssoftware wie beispielsweise Softwarekomponenten, Ports und Interfaces grafische Elemente in AUTOSAR (2019f) spezifiziert. Ein Beispiel für so eine Softwarearchitektur findet man in Abbildung 2.7. Ebenso existiert für AUTOSAR keine formale statische und dynamische Semantik. Modellierungsrestriktionen, sowie das Verhalten von Modellelementen innerhalb der Softwarearchitektur werden im Standard nur textuell in den Template Dokumenten erfasst.

AUTOSAR Timing Extensions

Eine Teilmenge des AUTOSAR Metamodells beschäftigt sich mit der Annotation von Modellelementen mit Zeiteigenschaften und Zeitanforderungen (AUTOSAR, 2019a). Für jede Entwicklungssicht gibt es eine Menge von AUTOSAR-Elementen, die mit Ereignissen (sogenannten *Timing Description Events*) annotiert werden können. Diese Events sind eine abstrakte Repräsentation eines spezifischen Systemverhaltens, das zur Systemlaufzeit überwacht werden kann, wie beispielsweise das Starten eines Runnable Entity oder die Ankunft eines neuen Datenpakets an einem bestimmten Port. Die Bedeutung von Anforderungen im allgemeinen und der Zusammenhang von Anforderungsartefakten in unterschiedlichen Entwicklungsphasen mit AUTOSAR Timing Constraints werden in Abschnitt 2.2 erläutert. AUTOSAR Timing Extensions ermöglichen nun die Spezifikation von *Timing Constraints*, mit denen zeitliche Abhängigkeiten zwischen zwei oder mehreren Events spezifiziert werden können. Darüber hinaus können komplexere Zeitabhängigkeiten definiert werden, indem Ereignisse durch Ereignisketten (*Timing Description Event Chains*) verkettet werden. Dadurch lässt sich eine Folge von Ereignissen spezifizieren, für die ein Timing Constraint gelten muss. Abbildung 2.5 zeigt das Metamodell für Zeiteigenschaften und Ereignisketten. Eine Übersicht der relevanten Metamodellelemente zeigt Abbildung 2.4: Für die einzelnen AUTOSAR-Sichten gibt es die spezialisierten *TimingExtensions VfbTiming*, *SwcTiming*, *SystemTiming*, *BswModuleTiming* und *EcuTiming*, die jeweils ein für die entsprechende Sicht re-

Timing Description Events

Timing Constraints

Timing Description Event Chains

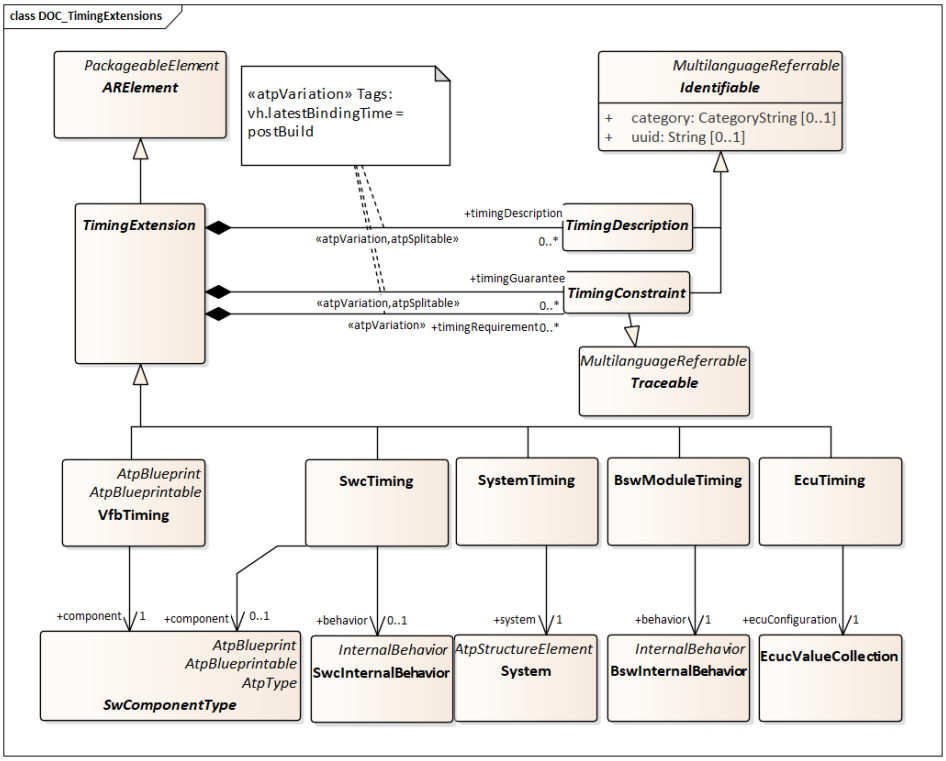

Abbildung 2.4: Metamodell für Timing Extensions aus AUTOSAR (2019a)

levantes Modellelement referenzieren, beispielsweise eine Softwarekomponente (*SwComponentType*) für das VfbTiming. Darüber hinaus hat jede *TimingExtension* eine Menge an *TimingDescriptions*, sowie *TimingConstraints*, die je nach Kontext die Rolle als *TimingGuarantee* oder *TimingRequirement* wahrnehmen.

Für die Definition von Timing Constraints gibt es eine Menge spezialisierter Unterklassen, die es ermöglichen bestimmte Klassen von zeitlichen Abhängigkeiten zu beschreiben. Abbildung 2.6 zeigt die verfügbaren Timing Constraints in AUTOSAR. Im Folgenden werden diese Constraints genauer vorgestellt. Eine vollständige Beschreibung befindet sich in AUTOSAR (2019a):

- *Offset Timing Constraint*: Der Offset Timing Constraint spezifiziert eine minimale und maximale Zeitdifferenz zwischen einem

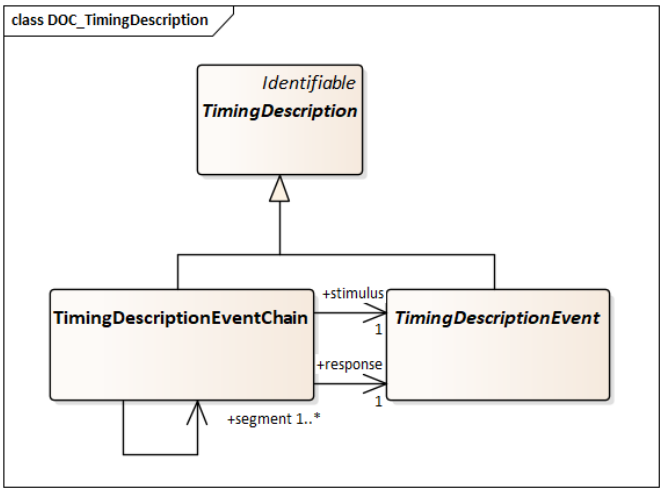

Abbildung 2.5: Metamodell für Timing Descriptions aus AUTOSAR (2019a): Eine Timing Description ist entweder ein einzelnes Timing Event oder eine Verkettung von Timing Events über Trigger- und Response-Assoziationen.

Source-Event und einem Target-Event.

- *Latency Timing Constraint*: Ein Latency Timing Constraint spezifiziert eine minimale und maximale Latenzzeit auf der Basis einer Timing Description Event Chain. Der Constraint wird verwendet, um Anforderungen an die zeitliche Verzögerung zwischen einem Stimulus-Event und allen weiteren Timing Events innerhalb der Ereigniskette bis zu einem Response-Event zu spezifizieren.

- *Synchronization Timing Constraint*: Der Synchronization Timing Constraint spezifiziert eine Menge von Timing Events, die gleichzeitig ausgeführt werden müssen. Zusätzlich kann durch die Spezifikation eines Toleranzwertes der Constraint abgeschwächt werden.

- *Execution Order Constraint*: Der Execution Order Constraint spezifiziert eine Sequenz von ausführbaren Einheiten (*Executable Entities*). Die Ausführung jedes Executable Entity darf dabei erst dann gestartet werden, wenn das vorherige Executable Entity beendet wurde.

Executable Entities

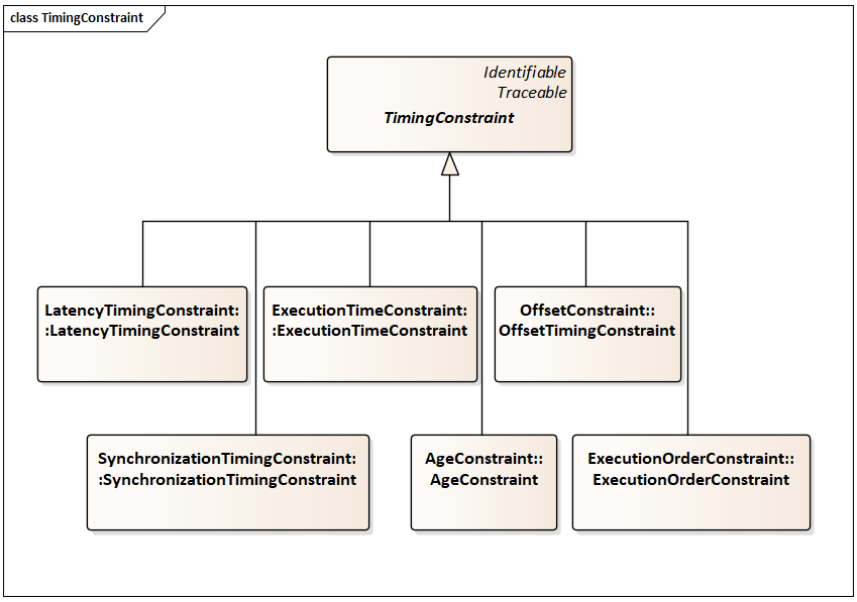

Abbildung 2.6: Verfügbare Timing Constraints in AUTOSAR (AUTOSAR, 2019a)

- *Execution Time Constraint*: Der Execution Time Constraint spezifiziert Restriktionen an die Ausführungszeit eines einzelnen Executable Entity. Im Gegensatz zu den anderen Timing Constraints, wird der Constraint nicht über Restriktionen der Timing Events modelliert.
- *Age Constraint*: Der Age Constraint spezifiziert das maximale Alter (Age) von Daten für einen Port. Als Spezialfall des Latency Timing Constraint kann er verwendet werden ohne dass der Absender der Daten bekannt sein muss.

SystemDesk® als Entwicklungswerkzeug für AUTOSAR

SystemDesk®† ist das Entwicklungswerkzeug für AUTOSAR-Architekturen von dSPACE. Es unterstützt dabei sowohl bei der Modellierung von AUTOSAR-Modellen durch eine übersichtliche Darstellung, als auch bei

†https://www.dspace.com/de/gmb/home/products/sw/system_architecture_software/systemdesk.cfm

der Erstellung von AUTOSAR-Software durch Codegeneratoren. Des Weiteren können sogenannte *Validierungsregeln* Modelle auf Wohlgeformtheit überprüfen. Auf diese Weise lassen sich bereits einfache Syntaxfehler, die beispielsweise durch unvollständige Modelle entstehen, erkennen, da das Werkzeug zur einfachen und intuitiven Modellierung auch das Anlegen von zunächst unvollständigen Modellen erlaubt. Weiterhin gibt es Regeln, die die Kompatibilität zu anderen Werkzeugen überprüfen oder die Generierbarkeit der RTE sicherstellen. Die Validierung von AUTOSAR-Modellen über die syntaktische Korrektheit hinaus ist dagegen mit Validierungsregeln nicht möglich. Eine Validierung des Modellverhaltens kann nur durch die Generierung, Kompilierung und anschließende Simulation eines virtuellen Steuergeräts erfolgen. Die Abbildung 2.7 zeigt die Oberfläche der Entwicklungsumgebung. Grafische Repräsentationen, im Kontext der Entwicklungsumgebung auch einfach *Diagramme* genannt, gibt es für die wichtigsten Elemente des Metamodells. So werden Softwarekomponenten, Ports und Verbindungen, die innerhalb einer Softwarekomposition vorhanden sind, in einem *Composition Diagram* zusammen dargestellt. Des Weiteren können einzelne Softwarekomponenten zusammen mit ihren Ports, Interfaces und Datentypen in einem *Component Diagram* visualisiert werden. Andere Modellelemente werden in einer hierarchischen Baumstruktur, dem *Project Manager* dargestellt und deren Eigenschaften in einem *Eigenschafts-Dialog*.

Validierungsregeln

Das AUTOSAR Blinker-Beispiel

Im Folgenden betrachten wir eine einfache AUTOSAR Softwarearchitektur. Die Architektur verwaltet die rechte und linke Blinkerleuchte eines Fahrzeugs. Diese werden entsprechend den Blinker- und Warnlichtsensoren geschaltet. Die Applikationsschicht beinhaltet mehrere Softwarekomponenten, die wiederum mehrere Runnables mit ausführbarer Software umfassen. Die Beispielarchitektur ist in Abbildung 2.8 zu sehen. Die zwei Softwarekomponenten auf der linken Seite lesen Sensordaten ein und prüfen diese auf Fehler, bevor die Signale an die nächste Softwarekomponente weitergeleitet wird. Die Software-Komponente Indicator Composition

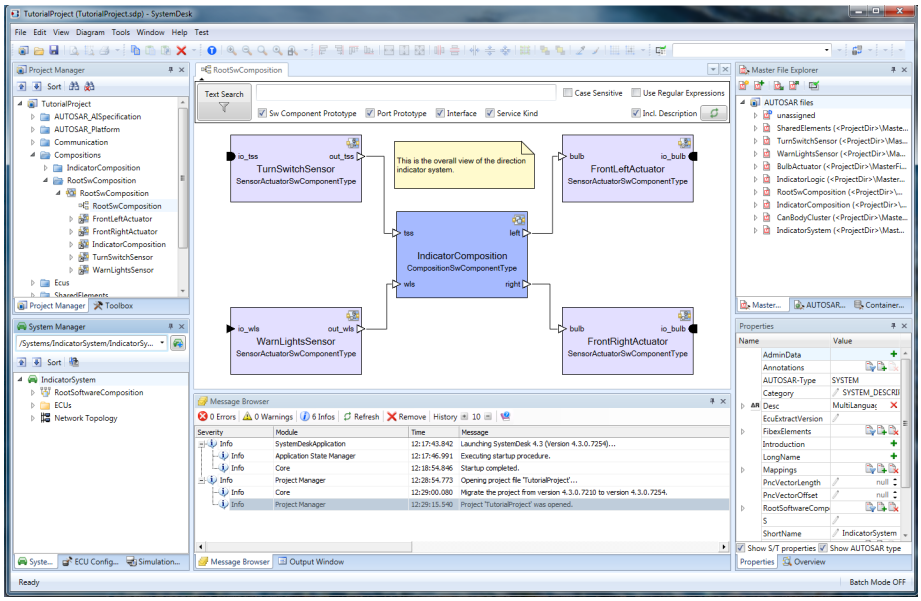

Abbildung 2.7: AUTOSAR Entwicklungsumgebung SystemDesk®

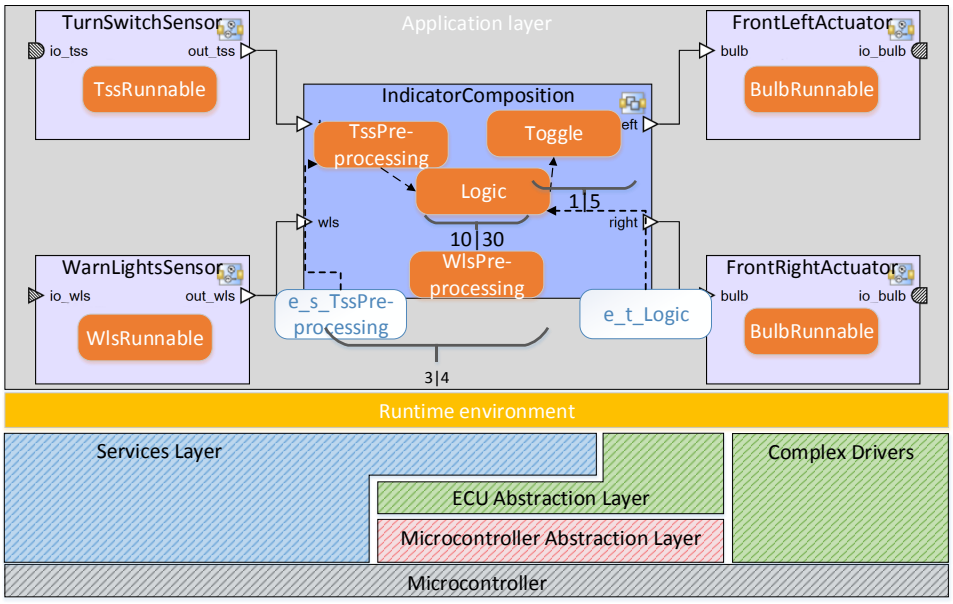

Abbildung 2.8: Beispiel einer AUTOSAR Softwarearchitektur

empfängt diese Sensordaten. Die Komponente beinhaltet mehrere Runnable Entities für die Vorverarbeitung der Signale, sowie die Logik des Systems. Die Aktuator-Softwarekomponenten auf der rechten Seite sind für die Aktivierung der Blinkerlampen zuständig. Darüber hinaus enthält das Beispiel eine Konfiguration der RTE und auf der Ebene der Basissoftwareschicht existiert die Konfiguration des Betriebssystems. Weitere Basissoftwaremodule werden in diesem Beispiel nicht berücksichtigt.

Darüber hinaus wird das Beispiel um die folgenden Timing Events und Timing Constraints erweitert. Das Runnable TssPreprocessing beinhaltet ein Event für den Start $e^{s}_{\text{TssPreprocessing}}$ und für das Runnable Logic wurde ein Event für die Terminierung e^{t}_{Logic} hinzugefügt. Diese Timing Events werden im Beispiel als weiße Boxen dargestellt, die mit Runnables verbunden sind. Weiterhin beinhaltet die Softwarearchitektur drei Timing Constraints: Ein Execution Order Constraint, welcher die Ausführungsreihenfolge der Runnables beschränkt. Dieser Constraint ist als gestrichelte Linie zwischen den Runnables zu sehen; ein Offset Timing Constraint, welcher die Ausführungszeit zwischen den neu hinzugefügten Timing Events beschränkt. Dieser wird durch die eckigen Klammern unter den Timing Events angezeigt. Und schließlich beinhaltet die Softwarearchitektur einen Execution Time Constraint, der die Ausführungszeit des Logic Runnables beschränkt. Tabelle 2.1 zeigt eine Übersicht über die Timing Constraints.

Timing Analyse

Ein wesentliches Merkmal von Steuergerätesoftware ist, dass sie Teil eines größeren Systems mit physikalischen Schnittstellen ist. Über diese Schnittstellen beeinflusst das System seine Umgebung. Viele Funktionen wie beispielsweise das Auslösen eines Airbags müssen zeitlich zusammen mit Ereignissen der Umgebung ausgeführt werden. Das korrekte Verhalten des Systems ist daher nicht nur abhängig von korrekten Funktionsergebnissen, sondern auch vom Zeitpunkt an denen diese Ergebnisse zur Verfügung stehen. Daher wird das System auch als *Echtzeitsystem* bezeichnet (Buttazzo, 2011, Stankovic und Ramamritham, 1988). Für die ingenieurmäßige Entwicklung solcher Systeme ist es daher wichtig Zeitanforderungen und Zeit-

Echtzeitsystem

Tabelle 2.1: Beispiel Timing Constraints

Beschreibung	Timing Constraint
Die Runnables TssPreprocessing, Logic und Toggle müssen nacheinander ausgeführt werden.	$r_{eoc} = \langle \text{TssPreprocessing}, \text{Logic}, \text{Toggle} \rangle$
Die Berechnung des Blinker-Logik muss frühestens nach 3ms und spätestens nach 4 beendet sein, nachdem die Vorverarbeitung des des Blinkersensors gestartet wurde.	$r_{otc} = (e^s_{\text{TssPreprocessing}}, e^t_{\text{Logic}}, 3, 4)$
Die Berechnung des Logic Runnables muss zwischen 10ms und 30ms beendet sein.	$r_{etc_1} = (\text{Logic}, 10, 30)$
Die Berechnung des Toggle Runnables muss zwischen 1ms und 5ms beendet sein.	$r_{etc_2} = (\text{Toggle}, 1, 5)$

eigenschaften in Anforderungsartefakten und Architekturmodellen zu erfassen.

Wie in Abschnitt 2.1.2 bereits beschrieben ermöglicht es AUTOSAR ebenfalls Zeitanforderungen und Zeitgarantien des Echtzeitsystems im Architekturmodell festzuhalten. AUTOSAR liefert aber keine Methoden, um die Korrektheit der Zeitanforderungen zu überprüfen. Für die Analyse des Modells und allen weiteren beteiligten Artefakten wie beispielsweise Reglercode, Binärcode und Hardwarespezifikationen werden daher Timing Analyseverfahren benötigt. Diese Verfahren können auf verschiedenen Ebenen und mit unterschiedlichen Methoden erfolgen (Traub, 2010).

Zum einen können Laufzeiten durch experimentelle Verfahren gemessen werden. Dazu werden in früheren Entwicklungsphasen Simulationsmodelle der Regler, Hardware und der Umgebung verwendet und innerhalb einer *Model-In-The-Loop-Simulation* (MIL-Simulation) simuliert. In späteren Phasen lassen sich dann Reglermodelle zunächst gegen Software austauschen, um diese in einer *Software-In-The-Loop-Simulation* (SIL-Simulation) zu testen und dann schließlich durch reale Hardware auszutauschen, um dann Laufzeittests durchzuführen (Traub, 2010). Die Simulation zusammen mit realer Hardware wird auch als *Hardware-In-The-Loop-Simulation*, (HIL-Simulation) bezeichnet (Schäuffele und Zurawka, 2010). Diese Verfahren können jedoch nicht das gesamte Systemverhalten verifizieren und somit keine kritische Randfälle wie obere Laufzeitschranken (*Worst-Case Execution Times, (WCET)*) finden. Werkzeuge für die Timing Simulation sind beispielsweise *ChronSim* (Anssi et al., 2012) und die *Real-Time Testing Observer Library* (dSPACE GmbH, 2020).

Model-In-The-Loop-Simulation

Software-In-The-Loop-Simulation

Hardware-In-The-Loop-Simulation

Worst-Case Execution Times, (WCET)

Zum anderen können Laufzeiten mithilfe analytischer Verfahren berechnet werden. Der Vorteil dieser Verfahren ist, dass sie den gesamten Zustandsraum des Modells untersuchen und somit auch Worst-Case Execution Times berechnen können. Dies ist notwendig, um Zeitanforderungen wie AUTOSAR Timing Constraints verifizieren zu können.

Diese Methoden unterscheiden sich weiterhin hinsichtlich ihrer Genauigkeit und der verwendeten Modelle. Zhang et al. (2014) unterscheiden zwei verschiedene Ebenen, auf denen Timing Analysen erfolgen:

1. Analyse auf *Taskebene*: Diese Methoden berechnen die Worst-Case Execution Time (WCET) für einen einzelnen Task oder Codeblock und benötigen dafür den auszuführenden Quellcode oder die für die Zielplattform kompilierte und ausführbare Binärdatei, sowie ein Modell der Hardwareplattform, auf denen der Code ausgeführt wird.

2. Analyse auf *Systemebene*: Diese Methoden untersuchen die Laufzeit eines gesamten Echtzeitsystems bestehend aus mehreren in einem Netzwerk verteilten Steuergeräten, auf denen wiederum mehrere Tasks ausgeführt werden. Diese Methoden werden häufig auch

als *System-Level Performance Analysis* bezeichnet. Einige Methoden benötigen jedoch als Grundlage die auf Taskebene berechneten WCETs.

Der in dieser Arbeit entwickelte Ansatz verifiziert das Zeitverhalten eines vollständigen AUTOSAR-Systems und fällt daher in die Kategorie *Analyse auf Systemebene*. Er setzt jedoch voraus, dass die WCETs einzelner Tasks bekannt sind. Dafür lassen sich existierende Werkzeuge wie beispielsweise aiT (Ferdinand und Heckmann, 2004) verwenden oder Schätzungen auf der Grundlage von Expertenwissen durchführen.

2.2 Anforderungsmodelle und Anforderungsqualität

Ein Teilbereich des Systementwicklungsprozesses beschäftigt sich mit der Gewinnung von Anforderungen und der Dokumentation von Anforderungsartefakten. Der Begriff Anforderung wird in IEEE (1990) folgendermaßen definiert:

Definition 2 (Anforderung IEEE (1990), Übersetzung aus Pohl (2008))**.** *Eine Anforderung ist :*

1. *Eine Bedingung oder Eigenschaft, die ein System oder eine Person benötigt, um ein Problem zu lösen oder ein Ziel zu erreichen.*

2. *Eine Bedingung oder Eigenschaft, die ein System oder eine Systemkomponente aufweisen muss, um einen Vertrag zu erfüllen oder einem Standard, einer Spezifikation oder oder einem anderen formell auferlegten Dokument zu genügen.*

3. *Eine dokumentierte Repräsentation einer Bedingung oder Eigenschaft wie in 1 oder 2 definiert*

In diesem Abschnitt wird ein Überblick über Modellierungssprachen für Anforderungen in der Systementwicklung gegeben. Der Fokus liegt dabei

auf Sprachen, die das Modellieren von Echtzeiteigenschaften erlauben. Weiterhin wird *Konsistenz* als ein relevantes Qualitätskriterium für Anforderungsartefakte genauer vorgestellt, da dies im Rahmen der Arbeit Anwendung findet.

Konsistenz

2.2.1 Anforderungsmodellierung in der Systementwicklung

Ein wesentlicher Aspekt bei der Entwicklung von elektronischen Systemen und Software ist das Erheben und Verwalten von Anforderungen. Üblicherweise erfolgt die Spezifikation am Anfang des Entwicklungsprozesses, sie kann aber auch während der Projektdurchführung weiterhin angepasst werden (Pohl, 2008). Schäuffele und Zurawka (2010) beschreiben das Anforderungsmanagement als Unterstützungsprozess für die Systementwicklung mit den Teilaufgaben des Erfassens von Anforderungen und Verfolgens von Anforderungen, während die Erstellung der logischen und technischen Systemarchitektur durch die Analyse der Anforderungen dem Kernprozess des V-Modells zugerechnet wird (siehe Abbildung 2.1).

Die akzeptierten Benutzeranforderungen sind die Basis für alle weiteren Entwicklungsphasen (Schäuffele und Zurawka, 2010). Sie werden zunächst informell als natürlichsprachlicher Text festgehalten, um auch für den Kunden verständlich zu sein. Im weiteren Verlauf werden dann die textuellen Anforderungen weiter verfeinert und formalisiert, beispielsweise mithilfe der *Unified Modeling Language (UML)*. Die UML ermöglicht das beispielhafte Modellieren von Anforderungen in Form von Anwendungsfällen (Use Cases) mithilfe einer grafischen Syntax Object Management Group (2015). Diese beschreiben formal die Interaktion eines Anwenders mit dem System und ermöglichen es Beziehungen zwischen Anwendungsfällen, sowie weitere Bedingungen festzuhalten. Diese Bedingungen können dann formal als OCL-Constraints spezifiziert werden (Object Management Group, 2014).

Unified Modeling Language (UML)

In den Phasen der Systementwicklung wird häufig das UML-Profil *SysML* eingesetzt. Es erweitert die UML über den UML-Profile Mechanismus um Sprachkonstrukte und Diagramme zur Spezifikation von Anforderungen. Diese können dann hierarchisch strukturiert werden. Des Weiteren können Beziehungen der Anforderungen untereinander

SysML

definiert werden, sowie zwischen Anforderungen und weiteren Modellelementen aus der Systemarchitektur (Object Management Group, 2017). Die eigentliche Anforderung ist jedoch weiterhin in textueller Form festgehalten. Für die Modellierung von Anforderungen mit Zeitbezug ist das nicht ausreichend. Daher existieren verschiedene Ansätze, die die Modellierung von Anforderungen mit Zeitbezug in den unterschiedlichen Entwicklungsphasen unterstützen.

2.2.2 Modellierung von Timing Anforderungen

Timing Anforderungen (auch Echtzeitanforderungen genannt) werden ebenso wie andere Anforderungsartefakte auch zunächst natürlichsprachlich beschrieben. Es ist jedoch sehr schwierig natürlichsprachliche Anforderungsartefakte direkt in AUTOSAR Timing Constraints zu überführen, da der Unterschied im Detaillierungsgrad sehr hoch ist. Daher werden die natürlichsprachlich festgehaltenen Timing Anforderungen zunächst in zeitbehaftete Modelle auf funktionaler Systemebene überführt und dort weiter zerlegt. Bei der Spezifikation der technischen Systemarchitektur werden dann die zuvor spezifizierten Timing Anforderungen auf Timing Anforderungen für einzelne Steuergeräte abgebildet, um so die Lücke zwischen textuellen Anforderungen und AUTOSAR Softwarearchitektur zu schließen (AUTOSAR, 2019d). Abbildung 2.9 zeigt die Dekomposition von Funktionen und die Abbildung dieser Teilfunktionen auf Steuergeräte. Die den Funktionen zugehörigen Timing Anforderungen müssen ebenfalls zerlegt werden und auf Timing Anforderungen für Steuergerätefunktionen abgebildet werden.

Für die Modellierung von Timing Anforderungen auf Systemebene werden existierende Modellierungssprachen verwendet, die um Methoden zur Spezifikation von Timing Anforderungen erweitert wurden. So kann beispielsweise das UML-Profil *MARTE* genutzt werden (Object Management Group, 2011a). MARTE erweitert die UML um Sprachkonstrukte zur Modellierung und Analyse von Echtzeitsystemen. Es besteht im Wesentlichen aus drei Hauptpaketen. Das *MARTE Analysis Model* beinhaltet Konzepte zur Validierung und Verifikation, das *MARTE Design Model* beinhaltet Elemen-

MARTE

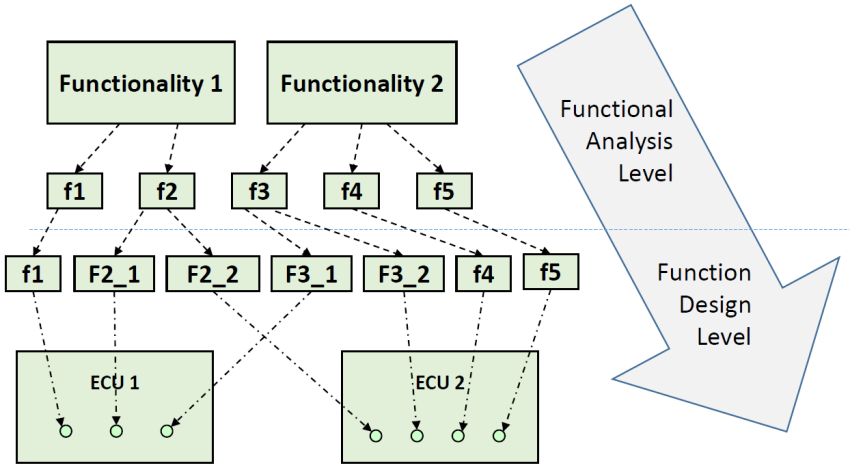

Abbildung 2.9: Dekomposition von Funktionen (AUTOSAR, 2019d)

te zur Modellierung von Echtzeitsystemen und das Paket *MARTE Foundations* beinhaltet generelle Elemente, die innerhalb der anderen Pakete benötigt werden (Ribeiro et al., 2018). Unter anderem ermöglicht das Unterpaket *Time* sehr detailliert die Modellierung von Zeiteigenschaften. Konkrete Zeitinstanzen werden durch Uhren (*Clocks*) festgehalten und Timing Anforderungen über Uhren können dann durch *Clock Constraints* festgelegt werden. Eine Übersicht über die Pakete des Profils findet sich in Abbildung 2.10.

Weiterhin existieren verschiedene Ansätze wie beispielsweise Ribeiro et al. (2018), die SysML und MARTE in einem Metamodell kombinieren. Dieses Metamodell spezifiziert beispielsweise für die Requirements aus SysML spezialisierte Elemente für nichtfunktionale Anforderungen und Timing Anforderungen, die es erlauben für eine Anforderung zusätzlich Zeiteigenschaften wie minimale und maximale Antwortzeiten festzulegen. Ebenfalls gibt es weitere Typen von Beziehungen zwischen Anforderungen. So kann mit der «Synchronized» Beziehung explizit die synchrone Ausführung von Anforderungen modelliert werden.

Eine weitere, in der Automobilindustrie verbreitete, domänenspezifische Sprache ist EAST-ADL (EAST-ADL Association, 2013). Die Sprache definiert vier verschiedene Abstraktionsebenen für die Modellierung von elektrisch / elektronischen Systemen. Angefangen mit der *Fahrzeugebene (Vehicle*

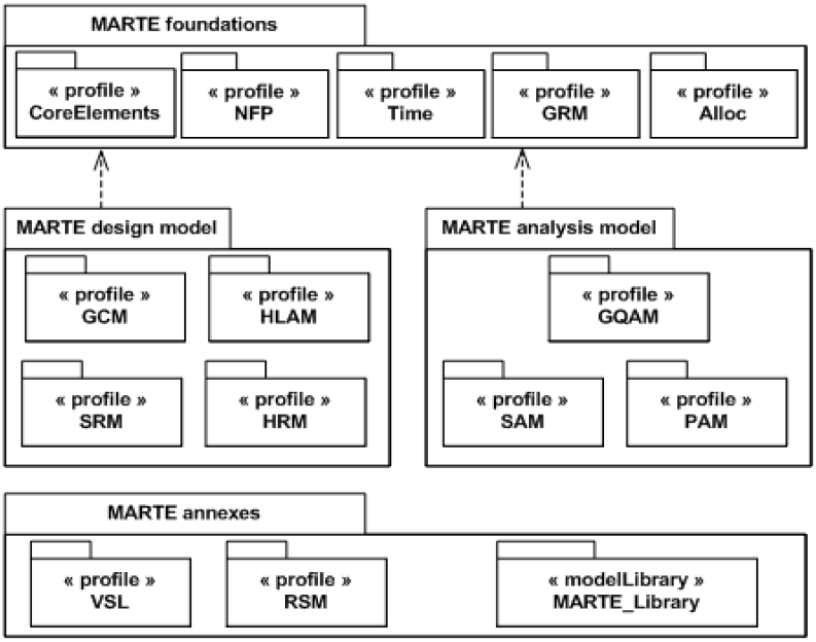

Abbildung 2.10: Architektur des MARTE Profils

Level) werden auf dieser Ebene Featuremodelle erstellt und daraus dann Architekturmodelle, erst auf Analyseebene (*Analysis Level*) als sogenannte *Functional Analysis Architecture* und dann auf Entwurfsebene (*Design Level*) als *Functional Design Architecture* weiter verfeinert. Die Implementierungsebene als unterste Ebene nutzt dann den AUTOSAR-Standard für die Spezifikation der Softwarearchitektur. Auf jeder Ebene können zu den entprechenden Modellen Anforderungen und Timing Constraints definiert und referenziert werden. Timing Constraints lassen sich dabei schon auf einem ähnlichen Abstraktionsnievau wie bei AUTOSAR modellieren, können aber bereits auf Fahrzeugebene angewendet werden. Eine Übersicht über die Architektur von EAST-ADL zeigt die Abbildung 2.11.

2.2.3 Konsistenz von Anforderungsdokumenten

Anforderungsdokumente beinhalten die dokumentierten Anforderungsartefakte für das zu entwickelnde System. Gerade die anfangs in Form des

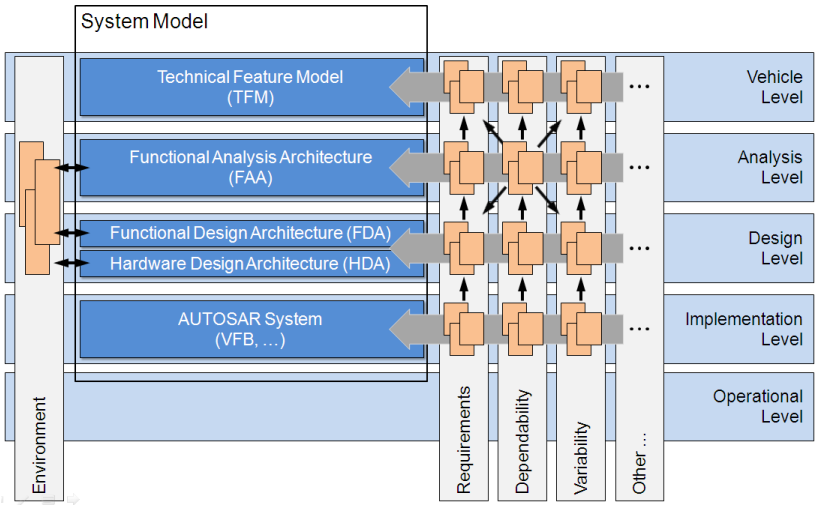

Abbildung 2.11: EAST-ADL Architekturebenen(EAST-ADL Association, 2013)

Pflichtenhefts dokumentierten Anforderungsartefakte sind von zentraler Bedeutung, da sie die Grundlage für alle weiteren Aktivitäten in der Systementwicklung bilden (Pohl, 2008). Ein fehlerhaftes Anforderungsdokument ist daher eine der Hauptursachen für Softwarefehler (Galin, 2004). Eine hohe Qualität der Anforderungsdokumentation und der einzelnen darin enthaltenen Anforderungsartefakte ist daher wichtig, um eine hohe Softwarequalität zu erreichen.

In der Literatur werden verschiedene Qualitätskriterien für Anforderungsdokumente vorgeschlagen, die die Qualität eines Anforderungsdokumentes bewerten. So definiert die IEEE in IEEE (1998) Charakteristiken für „gute "Anforderungen. Dort wird beschrieben, dass Anforderungen korrekt, eindeutig, vollständig, konsistent, überprüfbar, gewichtet, änderbar und rückverfolgbar sein sollen.

Konsistenz

Im Kontext der analytischen Qualitätssicherung mithilfe formaler Methoden ist insbesondere das Qualitätskriterium *Konsistenz* von entscheidender Bedeutung, da die Durchführung laufzeitintensiver Verifikationsmethoden nur auf der Basis einer konsistenten Anforderungsmenge verwertbare Ergebnisse liefern kann.

Konsistenz von Anforderungen wird in IEEE (1998) als „interne Konsis-

tenz "konkretisiert, denn ist eine Anforderungsmenge inkonsistent zu einem externen Spezifikationsdokument, so ist sie nicht korrekt.

Definition 3 (Interne Konsistenz (IEEE, 1998))**.** *Eine Anforderungsmenge ist „intern konsistent ", wenn keine Teilmenge im Konflikt zueinander steht. Ein Konflikt ist dann vorhanden, wenn*

1. *die Eigenschaften der realen Objekte im Konflikt zueinander stehen,*
2. *logische oder temporale Konflikte zwischen zwei oder mehr Aktionen bestehen,*
3. *zwei oder mehr Anforderungen unterschiedliche Bezeichnungen für dasselbe reale Objekt haben.*

Der Fokus dieser Arbeit liegt auf der Konsistenzprüfung von AUTOSAR Timing Constraints. Wir gehen daher davon aus, dass Anforderungen bereits als Timing Constraints formalisiert wurden, sodass Konflikte, die aus 1 und 3 resultieren, bereits aufgelöst wurden. Da die Timing Anforderungen ausschließlich Aussagen über das Zeitverhalten des Systems machen, bezieht sich der Konsistenzbegriff im Verlauf der Arbeit explizit auf temporale Konflikte innerhalb einer Anforderungsmenge.

2.3 Formale Analyse

Formale Methoden sind Modellierungssprachen, die durch eine formale, mathematisch definierte *Semantik* charakterisiert sind (Clarke und Wing, 1996). Für diese Methoden lassen sich Analyseverfahren entwickeln, mit denen sich konkrete Eigenschaften eines Modells formal beweisen lassen. Diese Analyseverfahren sind beispielsweise das sogenannte *Model Checking* (Clarke und Emerson, 1981, Queille und Sifakis, 1982), bei dem der gesamte Zustandsraum eines Modells exploriert wird und auf dessen Erfüllbarkeit hinsichtlich einer Temporallogischen Formel geprüft wird, *Abstrakte Interpretation* (Cousot und Cousot, 1977), bei der die Programmanalyse üblicherweise über eine Datenflussanalyse erfolgt, diese jedoch über die Definition eines Verbandes über-approximiert wird oder *Deduktive Verifikation* (Hoare,

Semantik

Model Checking

Abstrakte Interpretation

Deduktive Verifikation

1969), bei der die Analyse eines Programms über Axiome und Beweisregeln erfolgt. In diesem Abschnitt werden formale Methoden vorgestellt, die im Rahmen der Arbeit zur Anwendung kommen.

2.3.1 Formale Modellierung von Echtzeitsystemen

Durch die Verfügbarkeit von Modellierungstechniken zur Spezifikation von Echtzeitsystemen können komplexe technische Systeme in verschiedenen Entwicklungsphasen und aus unterschiedlichen Sichten beschrieben werden. Für die bisher beschriebenen Sprachen existiert jedoch keine formale Semantik, sodass die automatische Verifikation auf diesen Modellen nicht möglich ist. Erst durch die Anwendung von formalen Sprachen ist es möglich für dynamische Modelle Eigenschaften wie Erreichbarkeit von Systemzuständen oder Deadlockfreiheit, sowie Safety- und Liveness-Eigenschaften zu prüfen. Formale Sprachen zur Spezifikation von Echtzeitsystemen sind beispielsweise *Timed CSP* (Reed und Roscoe, 1986), eine Erweiterung der Sprache *CSP* von Hoare (1978) um Zeit, *Timed Petri-Nets* (Ramchandani, 1973), eine zeitbehaftete Erweiterung von Petrinetzen, *PLC-Automata* (Dierks, 1997), Zeitautomaten (*Timed Automata*) (Alur und Dill, 1994), *Duration Calculus* (Chaochen et al., 1991) oder Timed CCS (Yi, 1991). Wir verwenden in unserem Ansatz *Timed Automata* zur formalen Spezifikation. Für Timed Automata existieren mehrere Werkzeuge zur Modellierung und Verifikation, auf die wir für die Generierung der Modelle in dieser Arbeit zurückgreifen können. Im Folgenden gehen wir näher auf die Spezifikation und Analyse von Timed Automata ein.

Timed Automata

2.3.2 Timed Automata

Timed Automata wurden 1994 von Alur und Dill (1994) eingeführt, um das Verhalten von Echtzeitsystemen zu modellieren. Sie erweitern das Konzept von Büchi Automaten um Uhren mit reellen Zeitwerten. Dadurch lassen sich zeitliche Bedingungen an Zuständen und Kanten annotieren.

Definition 4 (Uhrenconstraints (Clock constraints))**.** *Sei $\mathbb{X}$ eine Menge von Uhren und sei $\Phi(\mathbb{X})$ die Menge der Uhrenconstraints mit Constraints φ wie folgt*

definiert:

$$\varphi ::= x \sim c \mid x - y \sim c \mid \varphi_1 \wedge \varphi_2$$

mit $x, y \in \mathbb{X}, c \in \mathbb{Q}^{\geq 0}$ *und* $\sim \in \{<, >, \leq, \geq\}$

Im Folgenden wird die Spezifikation von Timed Automata auf der Basis von Alur und Dill (1994) vorgestellt. Die Notation ist aus Olderog und Dierks (2008) entnommen und erweitert um die Spezifikation von Broadcast-Kanälen aus Behrmann et al. (2004).

Definition 5 (Timed Automata)**.** *Ein Timed Automaton ist ein Tupel* $\mathcal{A} = (L, B, B^*, X, I, U, E, I_{ini})$ *mit:*

- *einer endlichen Menge an Locations* L*,*
- *einer Menge von Signalen* B*, die mittels Handshake miteinander kommunizieren,*
- *einer Menge von Signalen* B^**, die über Broadcast-Kanäle miteinander kommunizieren,*
- *einer Menge von Uhren* X*,*
- *einer Zuweisung von Invarianten zu Locations* $I : L \to \Phi(X)$*,*
- *einer Abbildung für die Locations, ob diese unmittelbar (urgent) ausgeführt werden müssen (sodass in diesen Locations die Zeit nicht weiterlaufen kann)* $U : L \to \{true, false\}$*,*
- *einer Menge von Kanten gelabelt mit Signalen, einem Guard und einer Menge von Uhren, die zurückgesetzt werden:* $E \subseteq L \times B \cup B^* \times \Phi(X) \times \mathcal{P}(X) \times L$*,*
- *und einer initialen Location* $I_{ini} \in L$*.*

$\Phi(X)$ spezifiziert eine Menge von Uhrenconstraints (beispielsweise $x < 3$, siehe dazu auch Olderog und Dierks (2008)). Eine *Konfiguration* eines Timed Automaton ist dann ein Paar einer Location und einer Wertzuweisung (clock valuation) $\nu : X \to Time$, wobei $Time \in \mathbb{R}^{(\geq 0)}$ reelle Zahlen sind. Wir

Konfiguration

verwenden $\nu \models \varphi$ für einen Uhrenconstraint $\varphi \in \Phi(X)$, falls der Constraint für die jeweilige Wertzuweisung gilt, was folgendermaßen definiert ist:

$$\begin{aligned} \nu \models x \sim c \quad & gdw. & \nu(x) \sim c, \\ \nu \models x - y \sim c \quad & gdw. & \nu(x) - \nu(y) \sim c, \\ \nu \models \varphi_1 \wedge \varphi_2 \quad & gdw. & \nu \models \varphi_1 \text{ und } \nu \models \varphi_2 \end{aligned}$$

Die Abbildung 2.12 zeigt einen Beispielautomaten wie er in UPPAAL spezifiziert wird. Der Automat zeigt eine vereinfachte Variante eines in einen Timed Automaton transformierten *Runnable Entity* mit vier Locations und vier Transitionen. Die Locations *reading* und *writing* sind spezielle *Urgent locations*, erkennbar am Symbol ∪, in denen die Zeit nicht weiterlaufen kann. Sie sind semantisch äquivalent zu einer Location mit einer Uhr und einer Invariante $x \leq 0$ auf dieser Location. Erhält der Automat das *start*-Signal, wird die Transition zur Location *reading* ausgeführt und die Uhr x zurückgesetzt. Über das Signal *bulb_signal* erfolgt die Transition in die *running*-Location. Aufgrund der Invariante und dem Guard der folgenden Transition bleibt der Automat maximal 2 und mindestens 1 Zeiteinheit in dieser Location. Danach erfolgt die Transition in die Location *writing* und *finished* über die jeweiligen Signale.

URGENT LOCATIONS

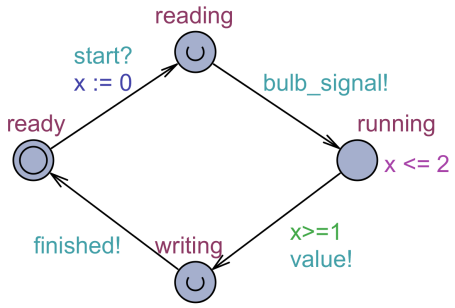

Abbildung 2.12: Beispiel Timed Automaton visualisiert mit UPPAAL

Die operationelle Semantik von Timed Automata wird als gelabeltes Transitionssystem definiert. Transitionssysteme wurden von Keller (1976) eingeführt und zuerst von (Plotkin, 1981) zur Spezifikation von Programmier- und Spezifikationssprachen verwendet. Die Semantikdefini-

tion basiert auf Behrmann et al. (2004) und der Notation aus Olderog und Dierks (2008).

Definition 6 (Semantik für Timed Automata)**.** *Die operationelle Semantik von Timed Automata $\mathcal{A}$ ist definiert als gelabeltes Transitionssystem $\mathcal{T}(\mathcal{A}) = (Conf(\mathcal{A}), \rightarrow, C_{ini})$ mit:*

- $Conf(\mathcal{A}) = \{\langle l, v\rangle \mid l \in L, v : X \rightarrow Time, v \models I(l)\}$,
- *einer initialen Konfiguration* $C_{ini} = \{\langle l_{ini}, v_{ini}\rangle\}$
- *und einer Transitionsrelation* $\rightarrow \subseteq Conf(\mathcal{A}) \times (Time \cup B) \times Conf(\mathcal{A})$ *mit zwei verschiedenen Arten von Transitionen:*
  - *Delay-Transition:* $\langle l, v\rangle \xrightarrow{t} \langle l, v + t\rangle$ *falls* $v + t' \models I(l) \forall t' \in [0, t] \wedge \forall l \in L : U(l) = false$
  - *Aktions-Transition:* $\langle l, v\rangle \longrightarrow \langle l', v'\rangle$ *falls* $(l, \alpha, \varphi, Y, l') \in E$ *mit* $v \models \varphi$ *und* $v' = v[Y := 0]$ *and* $v' \models I(l')$.

Einzelne Timed Automata können zu einem *Netzwerk* zusammengefasst werden. Dabei können die Automaten über zwei verschiedene Wege miteinander kommunizieren: synchron mittels Handshake-Kommunikation (wie beispielsweise auch in der Prozessalgebra CCS (Milner, 1980)) oder über Broadcast-Kanäle. Der Sender in einer Broadcast-Kommunikation kann dann mit allen Timed Automata kommunizieren, die gerade für den Empfang über den Broadcast-Kanal aktiviert sind. Im Folgenden wird die Semantikdefinition aus Behrmann et al. (2004) mit der Notation aus Olderog und Dierks (2008) vorgestellt: Netzwerk

Definition 7 (Semantik von Timed Automata Netzwerken)**.** *Für ein Timed Automata Netzwerk* $\mathcal{N} = \mathcal{A}_i = (L_i, B_i, B_i^*, X_i, I_i, E_i, l_{ini,i})$ *mit* $i = 1, \ldots, n$ *ist die Semantik definiert als Transitionssystem* $\mathcal{T}(\mathcal{N}) = (Conf(\mathcal{N}), \rightarrow, C_{ini})$ *mit:*

- $Conf(\mathcal{N}) = \{\langle \vec{l}, v\rangle \mid l_i \in L_i \wedge v : X \rightarrow Time \wedge v \models \bigwedge_{k=1}^{n} I_k(l_k)\}$
- *einer initialen Konfiguration* $C_{ini} = \{\langle \vec{l_{ini}}, v_{ini}\rangle\} \cap Conf(\mathcal{N})$ *mit* $\vec{l_{ini}} = (l_{ini,i}, \ldots, l_{ini,n})$ *und* $v_{ini}(x) = 0$

- *und einer Transitionsrelation* $\rightarrow = Conf(\mathcal{N}) \times Conf(\mathcal{N})$ *mit drei verschiedenen Arten:*
  - *Eine lokale Transition* $\langle \vec{l}, v \rangle \xrightarrow{a} \langle \vec{l'}, v \rangle$ *für einen Timed Automaton* $i \in \{1 \dots n\}$ *findet statt, wenn es eine Kante* $(l_i, \alpha, \varphi, Y, l'_i) \in E$ *gibt, sodass*
    - $v \models \varphi$
    - $\vec{l'} = l[l_i := l'_i]$,
    - $v' = v[Y := 0]$ *und* $v' \models I_i(l'_i)$
  - *Synchronisationstransition:* $\langle \vec{l}, v \rangle \xrightarrow{t} \langle \vec{l'}, v' \rangle$ *für zwei Timed Automata* $i, j \in \{1, \dots, n\}$ *mit* $i \notin j$ *und einem Kanal* $bin B_i \cap B_j$ *und Kanten* $(l_i, b!, \varphi_i, Y_i, l'_i) \in E_i$ *und* $(l_j, b?, \varphi_j, Y_j, l'_j) \in E_j$
  - *Delay-Transition:* $\langle \vec{l}, v \rangle \xrightarrow{t} \langle \vec{l}, v+t \rangle$ *falls* $v+t' \models \bigwedge_{k=1}^{n} I_k(l_k) \forall t' \in [0, t]$.

Formale Beschreibung von Anforderungen mittels Temporallogiken Die Möglichkeit zur formalen Beschreibung von Systemeigenschaften reaktiver Systeme über Temporallogiken wurde zuerst von Pnueli (1977) mit der *Linear Temporal Logic (LTL)* eingeführt. Diese lässt sich jedoch nur auf Modellen mit diskreter Zeit anwenden. Erweiterte Logiken zur Beschreibung von Echtzeiteigenschaften auf Modellen mit kontinuierlicher Zeit wie bei Timed Automata wurden dann durch die Erweiterung existierender Temporallogiken ermöglicht. Diese sind beispielsweise *Timed Computation Tree Logic (TCTL)* von Alur et al. (1993), die eine Erweiterung der *Computation Tree Logic (CTL)* von (Clarke und Emerson, 1981) ist, sowie *Metric Temporal Logic (MTL)* (Koymans, 1990) und *Timed Propositional Logic (TPTL)* (Alur und Henzinger, 1989) als Erweiterung von LTL. In dieser Arbeit wird eine Teilmenge von TCTL verwendet, die sich mittels UPPAAL überprüfen lässt. Die Syntax und Semantik dieser Logik wird in den Definitionen 8 und 9 beschrieben.

UPPAAL Zur Verifikation von Timed Automata existieren verschiedene Werkzeuge wie beispielsweise *UPPAAL* [‡] (Larsen et al., 1997), Kronos[§] (Yo-

[‡] http://www.uppaal.org
[§] http://www-verimag.imag.fr/DIST-TOOLS/TEMPO/kronos/

vine, 1997) oder HyTech (Henzinger et al., 1997). In dieser Arbeit wird zur Verifikation der AUTOSAR Timing Anforderungen UPPAAL eingesetzt, da das Werkzeug die Spezifikation von Timed Automata, wie in Definition 5 beschrieben, vollständig unterstützt. Abbildung 2.13 zeigt einen Screenshot der grafischen Oberfläche von UPPAAL mit dem Bridge-Crossing Problem aus Levmore und Cook (1981). Die Oberfläche ermöglicht die einfache Definition von Timed Automata und TCTL-Queries und ermöglicht darüber hinaus den Verifikationsprozess zu starten und fehlerhafte Traces in einer Simulation zu inspizieren.

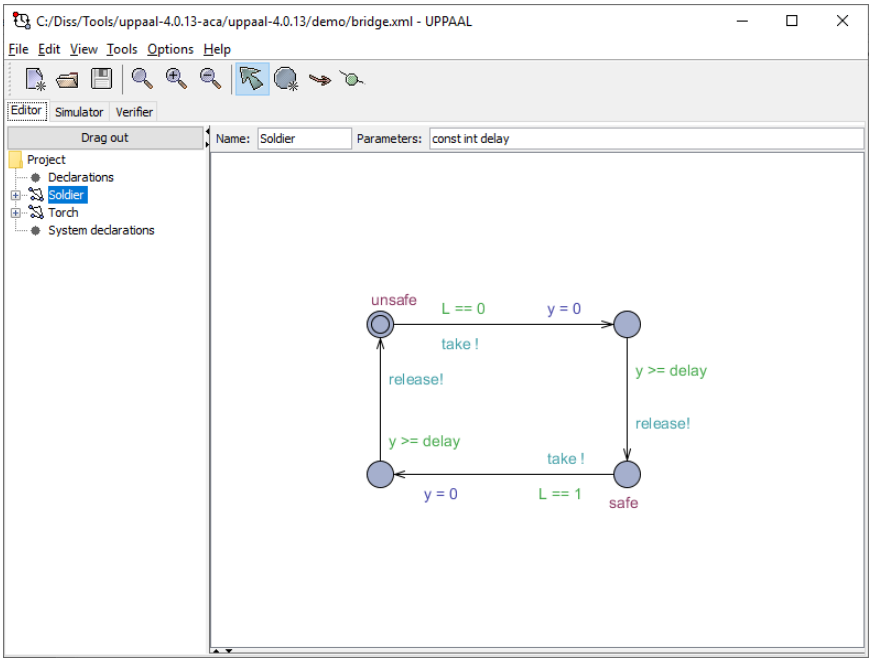

Abbildung 2.13: Screenshot von UPPAAL

UPPAAL verwendet zur Spezifikation von Modellanforderungen eine Teilmenge von TCTL (Behrmann et al., 2004). Die Logik beinhaltet *Basisformeln BF*, *Konfigurationsformeln CF* und *Pfadformeln, PF*, die sich wiederum in existenzielle und universelle Pfadformeln (EPF und APF) aufteilen (Olderog und Dierks, 2008). Im folgenden wird die Syntax beschrieben:

Definition 8 (UPPAAL-TCTL Formeln Olderog und Dierks (2008))**.**

$$
\begin{aligned}
BF &::= \mathcal{A}_i.l \mid \varphi, \\
CF &::= BF \mid \neg CF \mid CF_1 \wedge CF_2, \\
EPF &::= \exists\Diamond CF \mid \exists\Box CF, \\
APF &::= \forall\Box CF \mid \forall\Diamond CF \mid CF_1 \rightarrow CF_2, \\
PF &::= EPF \mid APF.
\end{aligned}
$$

Definition 9 (UPPAAL-TCTL Semantik Olderog und Dierks (2008))**.** *Sei ξ ein Pfad einer Konfiguration mit Start $\langle \vec{l_0}, v_0 \rangle, t_0$ und für einen Wert $t \in Time$ sei $\xi(t) = \{\langle \vec{l}, v \rangle \mid \exists i \in \mathbb{N} : (t_i \leq t \leq t_{i+1} \wedge \vec{l} = \vec{l_i} \wedge v = v_i + t - t_i)\}$ die Menge der Konfigurationen zum Zeitpunkt t. Sei $\models$ eine binäre Erfüllbarkeitsrelation zwischen Konfigurationen $\langle \vec{l_0}, v_0 \rangle, t_0$ eines Timed Automata Netzwerks und Formeln F geschrieben als $\langle \vec{l_0} v_0 \rangle, t_0 \models F$ wird induktiv definiert:*

$$
\begin{aligned}
\langle \vec{l_0}, v_0 \rangle, t_0 \models \mathcal{A}_i.l \quad & gdw. \quad l_{0,i} = l \\
\langle \vec{l_0}, v_0 \rangle, t_0 \models \varphi \quad & gdw. \quad v_0 \models \varphi \\
\langle \vec{l_0}, v_0 \rangle, t_0 \models \neg CF \quad & gdw. \quad \langle \vec{l_0}, v_0 \rangle, t_0 \not\models CF, \\
\langle \vec{l_0}, v_0 \rangle, t_0 \models CF_1 \wedge CF_2 \quad & gdw. \quad \langle \vec{l_0}, v_0 \rangle, t_0 \models CF_1 \text{ und } \langle \vec{l_0}, v_0 \rangle, t_0 \models CF_2 \\
\langle \vec{l_0}, v_0 \rangle, t_0 \models \exists\Diamond CF \quad & gdw. \quad \exists \xi \text{ mit Start bei } \langle \vec{l_0}, v_0 \rangle, t_0 \\
& \quad \exists t \in Time, \langle \vec{l}, v \rangle \in Conf : t_0 \leq t \\
& \quad \wedge \langle \vec{l}, v \rangle \in \xi(t) \wedge \langle \vec{l}, v \rangle, t \models CF, \\
\langle \vec{l_0}, v_0 \rangle, t_0 \models \forall\Box CF \quad & gdw. \quad \forall \xi \text{ mit Start bei } \langle \vec{l_0}, v_0 \rangle, t_0 \\
& \quad \forall t \in Time, \langle \vec{l}, v \rangle \in Conf : t_0 \leq t \\
& \quad \wedge \langle \vec{l}, v \rangle \in \xi(t) \wedge \Rightarrow \langle \vec{l}, v \rangle, t \models CF, \\
\langle \vec{l_0}, v_0 \rangle, t_0 \models \exists\Box CF \quad & gdw. \quad \exists \xi \text{ mit Start bei } \langle \vec{l_0}, v_0 \rangle, t_0 \\
& \quad \forall t \in Time, \langle \vec{l}, v \rangle \in Conf : t_0 \leq t \\
& \quad \wedge \langle \vec{l}, v \rangle \in \xi(t) \wedge \Rightarrow \langle \vec{l}, v \rangle, t \models CF, \\
\langle \vec{l_0}, v_0 \rangle, t_0 \models \forall\Diamond CF \quad & gdw. \quad \forall \xi \text{ mit Start bei } \langle \vec{l_0}, v_0 \rangle, t_0 \\
& \quad \exists t \in Time, \langle \vec{l}, v \rangle \in Conf : t_0 \leq t \\
& \quad \wedge \langle \vec{l}, v \rangle \in \xi(t) \wedge \langle \vec{l}, v \rangle, t \models CF,
\end{aligned}
$$

$\langle \vec{l_0}, v_0 \rangle, t_0 \models CF_1 \rightarrow CF_2$ *gdw.*

$\forall \xi$ *mit Start bei* $\langle \vec{l_0}, v_0 \rangle, t_0$
$\forall t \in Time, \langle \vec{l}, v \rangle \in Conf : t_0 \leq t$
$\wedge \langle \vec{l}, v \rangle \in \xi(t) \wedge \Rightarrow \langle \vec{l}, v \rangle, t \models CF_1$
impliziert $\langle \vec{l}, v \rangle, t \models \forall \Diamond CF_2$.

2.3.3 Satisfiability Modulo Theories

Das *Satisfiability Modulo Theories (SMT) Problem* ist eine Erweiterung des allgemeinen Erfüllbarkeitsproblems für boolsche Formeln (*Boolean Satisfiability Problem (SAT)*) um eine Hindergrundtheorie (Barrett, Clark und Tinelli, Cesare, 2018). Eine *Hintergrundtheorie* lässt sich informell als endliche oder unendliche Menge an Formeln, die durch gemeinsame grammatikalische Regeln charakterisiert sind, beschreiben (Kroening und Strichman, 2008). Beispiele hierfür sind lineare Arithmetik, Bitvektoren, Arrays oder Zeigerlogik. Die in dieser Arbeit verwendete Theorie ist die der *linearen Arithmetik*. Diese ist wie folgt definiert:

Hintergrundtheorie

linearen Arithmetik

Definition 10 (Syntax für SMT Formeln mit linearer Arithmetik)**.** *Die Syntax einer prädikatenlogischen Formel mit linearer Arithmetik ist durch die folgende Grammatik gegeben (Kroening und Strichman, 2008):*

$$
\begin{aligned}
\mathit{formula} &: \mathit{formula} \wedge \mathit{formula} \mid (\mathit{formula}) \mid \mathit{atom} \\
\mathit{atom} &: \mathit{sum}\ \mathit{op}\ \mathit{sum} \\
\mathit{op} &: = \mid \leq \mid < \\
\mathit{sum} &: \mathit{term} \mid \mathit{sum} + \mathit{term} \\
\mathit{term} &: \mathit{identifier} \mid \mathit{constant} \mid \mathit{constant}\ \mathit{identifier}.
\end{aligned}
$$

Formeln bestehen also aus Konjunktionen linearer Constraints. Ein linearer Constraint ist dabei eine (Un-)Gleichung über Terme, die wiederum Konstanten oder Variablen sein können. Die Syntax lässt sich einfach um die Operatoren $>$ und $\geq$ sowie $-$ erweitern, indem die jeweiligen Symbole ne-

giert werden. Als Domäne für die Terme können sowohl rationale Zahlen als auch natürliche Zahlen (Integer) gewählt werden. Formeln mit rationalen Zahlen lassen sich mit dem generellen Simplexalgorithmus und damit in polynomieller Zeit lösen. Für natürliche Zahlen ist das Problem NP-vollständig (Kroening und Strichman, 2008). Ein Beispiel für eine solche Formel ist $\mathcal{F} = a_1 + 1 \leq a_2 \wedge a_2 + 5 \leq a_3$.
Im weiteren Verlauf der Arbeit verwenden wir ein Werkzeug zur Lösung der Erfüllbarkeit von logischen Formeln mit linearer Arithmetik, um die Konsistenz von Timing Anforderungen zu überprüfen.

2.3.4 Maximum Satisfiability & Unsatisfiable Cores

Die zuvor beschriebenen Methoden ermöglichen es logische Formeln auf ihre Erfüllbarkeit hin zu überprüfen. Das bloße Wissen über die Erfüllbarkeit ist jedoch nicht immer ausreichend. Insbesondere, wenn eine Formel nicht erfüllbar ist, ist es wichtig zu wissen *warum* diese nicht erfüllbar ist. Daher gibt es neben dem Erfüllbarkeitsproblem weitere Fragestellungen, die sich mit *Optimierungsaufgaben* für logische Formeln beschäftigen (Bjorner und Phan, 2014). Im Folgenden werden die Probleme der *Maximum Satisfiability* zum Finden einer maximal großen Menge erfüllbarer Formeln und des *Unsat Cores* zum Finden einer minimal großen Menge unerfüllbarer Formeln vorgestellt.

Optimierungsaufgaben

Maximum Satisfiability

Unsat Cores

Maximum Satisfiability

Die Aufgabe des Findens einer maximal großen Menge an erfüllbaren Formeln wird auch als *MaxSMT* bezeichnet und wird in Bjorner und Phan (2014) folgendermaßen definiert:

MaxSMT

Definition 11 (Weighted MaxSMT)**.** *Gegeben sei eine Menge an Formeln $\mathcal{F}$ und numerischen Gewichten $w \in \mathcal{W}$ zusammen mit einer Funktion für jede Formel $c : \mathcal{F} \to W$. Dann ist es die Aufgabe für* Weighted Maximum Satisfiability Solving Modulo Theories (MaxSMT) *eine Teilmenge $I \subseteq \mathcal{F}$ zu finden, sodass:*

1. $\bigwedge_{f \in I} f$ *erfüllbar ist und*

2. $\sum_{f \in I} c(f)$ *maximal für alle Teilmengen von* $\mathcal{F}$ *ist.*

Ist also eine logische Formel nicht erfüllbar, so beschreibt das Problem MaxSMT das Finden einer Teilmenge von Formeln, die eine zuvor festgelegte *Kostenfunktion* maximiert. Falls es keine Prioritäten bei den Formeln gibt, so wird $c(f) = 1$ für alle Formeln gesetzt. Das Finden eines Maximum Satisfiable Sets ist ebenfalls wie das einfache Erfüllbarkeitsproblem NP-vollständig (Garey et al., 1976). Das folgende Beispiel verdeutlicht die Definition von MaxSMT:

Kostenfunktion

Beispiel 1 (Beispiel für MaxSMT)**.** *Sei* $\mathcal{F} := \{f_1, f_2, f_3\}$ *eine Menge von Formeln mit* $f_1 := a < b$, $f_2 := b < c$ *und* $f_3 := c < a$, *wobei a, b und c jeweils Konstanten sind. Die Formelmenge* $\mathcal{F}$ *ist nicht erfüllbar, da es für die Konjunktion der Formeln keine erfüllbare Belegung der Konstanten gibt. Die Teilmengen* $I_1 := \{f_1, f_2\}$, $I_2 := \{f_2, f_3\}$ *und* $I_3 := \{f_3, f_1\}$ *sind hingegen erfüllbar und maximal groß. Eine erfüllbare Belegung für* I_1 *wäre beispielsweise* $a = 1, b = 2, c = 3$. *Daher sind diese Teilmengen Beispiele für eine Lösung von MaxSMT.*

In dieser Arbeit verwenden wir einen Algorithmus zum Lösen von MaxSMT, um eine maximal große Menge an konsistenten Anforderungen zu berechnen.

Minimum Unsatisfiable Cores

Ein Unsatisfiable Core (oder in Kurzform: *Unsat Core*) einer unerfüllbaren logischen Formel in konjunktiver Normalform ist jede unerfüllbare Teilmenge der ursprünglichen Menge an Klauseln (Kroening und Strichman, 2008).

Definition 12 (Unsatisfiable Core)**.** *(Lynce und Silva, 2004) Gegeben sei eine Formel* φ, *UC ist ein Unsatisfiable Core für* φ *genau dann wenn UC eine Formel* φ_c *ist, sodass* φ_c *unerfüllbar ist und* $\varphi_c \subseteq \varphi$.

Unsat Cores können dabei helfen, die Ursache für die Unerfüllbarkeit einer Formel zu finden, indem sie den Fokus auf eine unerfüllbare Teilmenge

des Gesamtproblems legen (Kroening und Strichman, 2008). Für jede Formel existieren jedoch typischerweise mehrere Unsat Cores mit einer unterschiedlichen Anzahl an Klauseln, wobei einige Unsat Cores wiederum Teilmengen anderer sind (Lynce und Silva, 2004). Um möglichst genau den Grund der Unerfüllbarkeit einer Formel zu finden, ist es wichtig, dass ein Unsat Core minimal ist. Dieser Unsat Core wird dann als *Minimal Unsatisfiable Core* bezeichnet:

Definition 13 (Minimal Unsatisfiable Core)**.** *(Lynce und Silva, 2004) Ein Unsatisfiable Core UC für φ ist ein Minimal Unsatisfiable Core genau dann wenn aus dem Entfernen jedes beliebigen Ausdrucks $\omega \in UC$ folgt, dass $UC - \omega$ kein Unsatisfiable Core mehr ist.*

Ein Minimal Unsatisfiable Core einer unerfüllbaren logischen Formel ist also eine Teilmenge von Aussagen, die nicht weiter verkleinert werden kann, ohne dass sie dann erfüllbar wird (Liffiton und Sakallah, 2008). Im Rahmen dieser Arbeit verwenden wir den Minimal Unsatisfiable Core zur Berechnung kleinstmöglicher inkonsistenter Anforderungsmengen und nennen ihn im folgenden einfach *Unsat Core*. Das folgende Beispiel verdeutlicht die Definition:

Beispiel 2 (Beispiel für einen Minimal Unsatisfiable Core)**.** *Sei $\varphi = a < b \wedge b < c \wedge c < d \wedge c < a \wedge d < b$. Die Gesamtformel ist unerfüllbar. Jede beliebige Teilmenge an Ausdrücken, die ebenfalls unerfüllbar ist, ist dann ein Unsatisfiable Core, beispielsweise $\varphi_c = a < b \wedge b < c \wedge c < d \wedge d < b$. φ_c ist jedoch nicht minimal, da durch das Entfernen des Ausdrucks $a < b$ die Formel ebenfalls unerfüllbar ist. Erst aus der Formel $\varphi_{c'} = b < c \wedge c < d \wedge d < b$ kann kein weiterer Ausdruck entfernt werden, ohne dass die Formel dann erfüllbar wird. Somit ist $\varphi_{c'}$ ein Minimal Unsatisfiable Core.*

Sowohl für das Lösen von SMT-Formeln, als auch für die Berechnung von MaxSMT und Unsat Core wird in dieser Arbeit der SMT Solver *Z3*[¶] verwendet (de Moura und Bjørner, 2008). Z3 wird von Microsoft entwickelt und wird dort bereit seit 2007 verwendet. Für das Erstellen von SMT-Formeln existieren Anbindungen an die verschiedensten Programmiersprachen wie

Z3

[¶]https://github.com/Z3Prover/z3

C, OCaml, Java oder .NET. Ebenfalls ist es möglich Formeln in der standardisierten Beschreibungssprache SMT-LIB einzulesen (Barrett, Clark et al., 2017, de Moura und Bjørner, 2008).

2.4 Zusammenfassung

In diesem Kapitel wurden die für diese Arbeit wichtigen Grundlagen vorgestellt. Zunächst wurde der Entwicklungsprozess der Automobilindustrie vorgestellt (siehe Abschnitt 2.1). Anhand dieses Prozesses wird im nächsten Kapitel (Kapitel 3) der entwickelte Ansatz einsortiert und daran dessen Vor- und Nachteile erläutert.

Weiterhin wurde der AUTOSAR-Standard in Abschnitt 2.1 vorgestellt, der eine Entwicklungsmethodik und Metamodell umfasst. Dieses Metamodell ist Gegenstand der im weiteren Verlauf vorgestellten Analysemethoden in Kapitel 4 und 5.

Ebenfalls wurden Anforderungsmodelle, sowie Konsistenz als relevantes Qualitätskriterium für Anforderungen in Abschnitt 2.2 vorgestellt, für das später eine Methode zur Verifikation vorgestellt wird (Kapitel 5).

Es wurden formale Methoden zur Verifikation zeitbehafter Systeme, sowie SMT Grundlagen in Abschnitt 2.3 vorgestellt. Die in den nachfolgenden Kapiteln vorgestellten Ansätze nutzen diese, indem relevante Teile des AUTOSAR-Modells in die jeweiligen Analysemodelle transformiert werden, sodass existierende Werkzeuge für die Verifikation eingesetzt werden können.

3

Frühzeitige Verifikation von AUTOSAR Timing Anforderungen

Eine der wichtigsten Kriterien in der Entwicklung aktueller elektrisch-elektronischer Architekturen ist Timing (AUTOSAR, 2019d). Der Grund dafür ist nicht nur, dass die zu entwickelnden Teilfunktionen Echtzeitanforderungen haben, sondern darüber hinaus auch sicherheitskritisch sind, sodass die Absicherung besonders sorgfältig im Sinne von Sicherheitsstandards wie der *ISO 26262* (ISO International Organisation for Standardisation, 2018) durchgeführt werden muss. Des Weiteren ist die Verifikation von Timing Anforderungen sehr aufwändig. Etablierte Validierungsmethoden wie die Simulation des Steuergeräts unterstützen entweder die Simulation des Zeitverhaltens gar nicht oder nur eingeschränkt. Ebenfalls haben diese Verfahren nur eine eingeschränkte Aussagekraft, da sie kritische Randfälle wie obere Laufzeitschranken nur abschätzen können (Richter, 2005), sodass sie für die Timing Verifikation ungeeignet sind (Wilhelm et al., 2008, Ha-

ISO 26262

mann et al., 2006). Werkzeuge oder Methoden zur statischen Analyse hingegen können aufgrund der Verwendung abstrakter Modelle nur eine *konservative Abschätzung* von Laufzeitschranken berechnen, wie beispielsweise Real-Time Calculus von Thiele et al. (2000), Albers et al. (2008) oder SymTA/S von Feiertag et al. (2008) (Zhang et al., 2014) oder haben *hohe Laufzeiten*, wenn sie beispielsweise auf Model Checking Techniken wie Timed Automata basieren (Perathoner et al., 2009). Des Weiteren erfordern diese Methoden zudem, dass existierende Modelle wie AUTOSAR- oder Bus-Modelle und Quellcode zunächst in für die Analysewerkzeuge notwendigen abstrakten Analysemodelle transformiert werden müssen. Daher werden diese Methoden häufig erst spät im Entwicklungsprozess eingesetzt (Anssi et al., 2012). Werden Fehler beim Validieren der Timing Anforderungen erkannt, so müssen jedoch alle in den vorherigen Entwicklungsphasen erstellten Artefakte überarbeitet werden. Je später Fehler identifiziert werden, desto aufwändiger und teurer wird die Korrektur und somit die gesamte Entwicklung (Schäuffele und Zurawka, 2010). Das Ziel ist es somit Timing Anforderungen so früh wie möglich durch automatisierte Verfahren zu überprüfen und gegen alle verfügbaren Artefakte zu validieren. Somit lassen sich Fehler frühzeitig identifizieren und Entwicklungszyklen können weiter verkürzt werden ohne dabei Defizite bei der Qualitätssicherung einzugehen.

konservative Abschätzung

hohe Laufzeiten

Der im Folgenden vorgestellte Lösungsansatz unterstützt dieses Vorgehen, indem Anforderungen schon vor der eigentlichen Verifikation auf Konsistenz überprüft werden, sodass eine laufzeitintensive Verifikation nur auf konsistenten Daten durchgeführt wird und somit unnötige Wiederholungen der Verifikation aufgrund von inkonsistenten Anforderungsmengen vermieden werden. Des Weiteren lässt sich die Verifikation ohne die Verwendung von Quellcode durchführen, was dazu führt, dass auch die Timing Verifikation bereits in Entwicklungsphasen durchgeführt werden kann, in denen noch kein Quellcode vorhanden ist.

Dieses Kapitel stellt eine Übersicht über die entwickelte Methode zur frühzeitigen Timing Verifikation von AUTOSAR Timing Anforderungen vor. Es wird zunächst das Konzept der integrierten Konsistenzprüfung und Timing Verifikation vorgestellt. Danach werden die für die Methode benötigten AUTOSAR Modellelemente formal definiert. Im Anschluss wird

diskutiert, an welchen Stellen des AUTOSAR-basierten Softwareentwicklungsprozesses die Methode zum Einsatz kommen kann und welche Vor- und Nachteile daraus resultieren. Dieses Kapitel basiert teilweise aus den bereits veröffentlichten Arbeiten Beringer und Wehrheim (2016) und Beringer und Wehrheim (2020).

3.1 Konsistenzprüfung und Timing Verifikation

Verschiedene Klassen von Timing Anforderungen ermöglichen eine einfache Verwendung für verschiedene Teile der Softwarearchitektur in unterschiedlichen Entwicklungsphasen. Nichtsdestotrotz kann es schwierig werden den Überblick über alle Anforderungsartefakte zu behalten, gerade wenn diese im Laufe der Zeit in Menge und Komplexität zunehmen, sodass Inkonsistenzen auftreten können. Eine inkonsistente Anforderungsmenge kann ein Anzeichen für ein Missverständnis der erwarteten Systemfunktionalität durch den Softwareentwickler, den Systemarchitekten oder weitere Stakeholder sein. Sie kann aber auch durch eine fehlerhafte Modellierung oder eine fehlerhafte Interpretation der natürlichsprachlich beschriebenen Anforderungen sein. Darüber hinaus führen inkonsistente Anforderungen immer zu fehlerhaften Verifikationsläufen, wenn Anforderungen formal gegen die Systemarchitektur geprüft werden, da die Verifikation unvermeidlich für einige Anforderungen fehlschlägt.

Daher ist es vorteilhaft, Inkonsistenzen in Anforderungsmengen bereits vorab zu erkennen. Dazu schlagen wir eine Methode zur Unterstützung der frühzeitigen Validierung der AUTOSAR Softwarearchitektur vor, indem zunächst der Prozess der Spezifikation von AUTOSAR Timing Anforderungen durch eine Konsistenzanalyse vereinfacht wird und im Anschluss daran mittels Timing Verifikation auf Korrektheit hinsichtlich der spezifizierten Anforderungen überprüft wird. Dies umfasst insgesamt die folgenden vier Schritte: Im ersten Schritt wird eine vorgegebene Menge an AUTOSAR Timing Anforderungen auf *Konsistenz überprüft*, indem diese in logische Formeln transformiert werden. Dies ermöglicht es die Timing An-

Konsistenz überprüft

forderungen zu überprüfen bevor weitere Verifikationsaufgaben angestoßen werden und unterstützt einen Requirements Engineer bei der Fehlerbehebung. Ist die Menge der Anforderungen inkonsistent, werden in einem zweiten Schritt mögliche *Ursachen für Inkonsistenzen* identifiziert. Dazu werden Teilmengen ermittelt, die zur Auflösung der Inkonsistenz in Betracht gezogen werden sollten, und dem Anwender vorgeschlagen. Dies beinhaltet Teilmengen, die für sich gesehen bereits inkonsistent sind. Des Weiteren werden größtmögliche konsistente Anforderungsmengen ermittelt, sodass ein Requirements Engineer durch die Betrachtung der Teilmenge im Zusammenhang mit den restlichen Anforderungen Inkonsistenzen ausfindig machen und auflösen kann. Unterstützt wird er ebenfalls durch eine geeignete grafische Visualisierung dieser Teilmengen. Ein Requirements Engineer hat dann im dritten Schritt die Möglichkeit auf Basis der erzeugten Hinweise durch *Modifikation* der Teilmengen eine neue Anforderungsmenge zu erzeugen. Diese kann dann erneut auf Konsistenz geprüft werden. Ist die Menge der Anforderungen schließlich konsistent, kann sie in einem vierten Schritt verifiziert werden. Für die *Verifikation* werden dann sowohl die Anforderungen als auch die Softwarearchitektur nach Timed Automata transformiert und anschließend mittels UPPAAL verifiziert.

Ursachen für Inkonsistenzen

Modifikation

Verifikation

Eine Übersicht über die vier Schritte und ihre Ausführungsreihenfolge, sowie ein Verweis auf den entsprechenden Abschnitt der Arbeit, ist in Abbildung 3.1 dargestellt. Für eine gegebene Menge an Anforderungsartefakten $\mathcal{R}$ und eine gegebenes Softwarearchitekturmodell $\mathcal{M}$ ist die Ausführungsreihenfolge wie folgt:

1. Prüfen der Konsistenz der Anforderungsmenge $\mathcal{R}$. Dieser Schritt wird in Abschnitt 4.1 vorgestellt.

2. Wenn die Gesamtmenge der Anforderungsartefakte nicht konsistent ist, werden die Ursachen für Inkonsistenz identifiziert, ansonsten weiter nach Schritt 4. Dieser Schritt wird in Abschnitt 4.2.1 vorgestellt.

3. Überarbeiten der Anforderungsartefakte auf der Grundlage der Hinweise aus dem vorherigen Schritt. Danach kann erneut bei Schritt 1 begonnen werden und die Schritte 1-3 werden so lange wiederholt

bis die Anforderungsmenge konsistent ist. Dieser Schritt wird in Abschnitt 4.2.2 vorgestellt.

4. Durchführen der Verifikation auf der Anforderungsmenge und dem Architekturmodell $(\mathcal{M}, \mathcal{R})$. Dies wird in Kapitel 5 gezeigt.

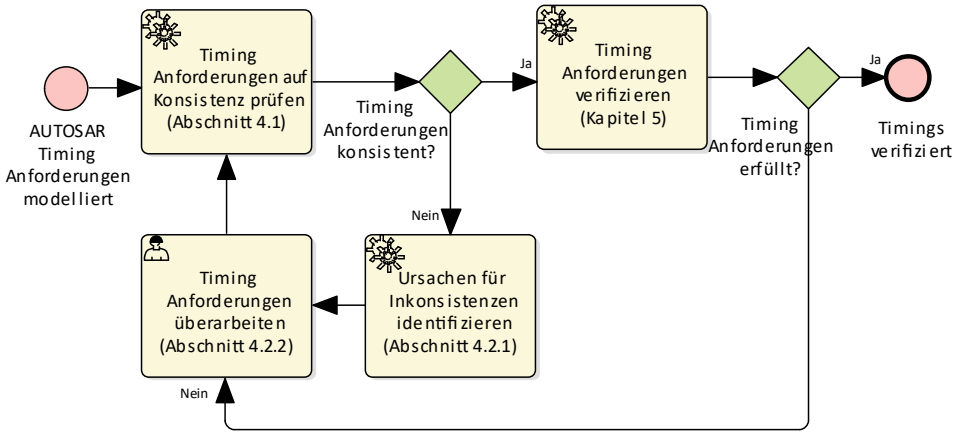

Abbildung 3.1: Analyseprozess als BPMN-Prozess

3.2 Zeitaspekte innerhalb einer AUTOSAR Softwarearchitektur

Die Spezifikation von AUTOSAR gibt zwar über das Metamodell eine formale Syntax vor, die formale Semantik des Modells wird allerdings nur in textuellen Spezifikationen festgehalten, sodass semantische Analysen des Timing Verhaltens auf AUTOSAR-Modellen nicht durchgeführt werden können. Üblicherweise wird für die Validierung von AUTOSAR ein simulationsbasierter Ansatz gewählt, bei dem sowohl für das AUTOSAR-Modell als auch für die Verhaltensmodelle aus Simulink Code generiert, kompiliert und im Anschluss als virtuelles Steuergerät im Verbund mit einem Umgebungsmodell simuliert wird. Diese Möglichkeit der Generierung virtueller Steuergeräte auf der Basis von AUTOSAR-Modellen mithilfe

von Codegenerierungs- und Simulationswerkzeugen wie Targetlink®und SystemDesk®ist zur Verwendung als Grundlage für eine formale Semantik zur Verifikation von Timing Anforderungen jedoch nicht zielführend, da der Code und somit das Verhalten dieser Steuergeräte in der Simulation zur Absicherung funktionaler Eigenschaften vorgesehen ist und Zeitaspekte wie beispielsweise spezifizierte Worst- und Best-Case Execution Times nicht berücksichtigt. Daher ist es notwendig eine eigene formale Semantik auf Basis der textuellen Spezifikationen herzuleiten, die alle für eine Timing Analyse notwendigen Metamodell-Elemente abbildet.

Eine vollständige Formalisierung des AUTOSAR Metamodells ist jedoch aufgrund der Größe des Metamodells und der Spezifikation nicht so einfach möglich. Allerdings haben viele Modellelemente keinen Einfluss auf das zeitliche Verhalten des Systems oder es existieren viele spezialisierte Klassen, die jedoch alle ein ähnliches zeitliches Verhalten haben.Wir beschränken uns daher im Folgenden auf Elemente, die einen großen Einfluss auf das Laufzeitverhalten des Systems haben und geben anschließend eine vereinfachte Formalisierung des Metamodells vor. Die formale Beschreibung von AUTOSAR teilt sich im Folgenden in die formale Spezifikation des AUTOSAR Architekturmodells und der AUTOSAR Timing Anforderungen auf.

3.2.1 Formale Spezifikation der AUTOSAR Softwarearchitektur

In diesem Abschnitt wird eine formale Verhaltensspezifikation erarbeitet, die das Zeitverhalten eines AUTOSAR Steuergeräts nachbildet, sodass sich auf dieser Basis Analyseverfahren entwickeln lassen. Für die formale Spezifikation der AUTOSAR Softwarearchitektur werden auf allen Ebenen der AUTOSAR-Schichtenarchitektur timing-relevante Elemente identifiziert und im formalen Modell festgehalten. Auf dieser Grundlage lassen sich dann Timing Analysen für fast allen Sichten der AUTOSAR-Entwicklungsmethodik erstellen.

Formale Eigenschaften der Softwarearchitektur

Wir definieren für jede AUTOSAR Softwareschicht eine Teilmenge der verfügbaren Modellelemente, die für das Laufzeitverhalten relevant ist und für die wir dann ein formales Modell angeben.

Timing auf Applikationsebene

Für die Modellierung des Timing Verhaltens auf Applikationsebene ist es notwendig das Verhalten von *Runnables*, Variablenzugriffen (*Variable Accesses*) und deren Verbindungen (*Assembly Connections*) abzubilden. Wir abstrahieren vom Konzept der Softwarekomponenten und Ports, da es für das Timing nicht relevant ist, ob zwei Runnables in verschiedenen Softwarekomponenten über Ports miteinander kommunizieren oder direkt in einer einzigen Softwarekomponente über sogenannte Inter-Runnable-Variablen, solange beide Runnables auf demselben Steuergerät laufen. Weiterhin gibt es Modellelemente, mit denen der Resourcenverbrauch von Runnables modelliert werden kann. Mithilfe dieser Elemente werden dann Worst-Case Execution Times und Best-Case Execution Times festgelegt.

Timing auf RTE Ebene

Die RTE-Ebene ist eine standardisierte Schnittstelle für die Software der Anwendungsschicht und ist verantwortlich für das Triggern der Runnables, so wie sie im Betriebssystem spezifiziert sind. Das Betriebssystem verfügt über einen *Scheduler* und überwacht die Ausführung von Resourcen durch OSTasks. Aus diesem Grund müssen Runnables auf OSTasks abgebildet werden, um die Ausführungsreihenfolge der Runnables festzulegen. Dies geschieht in der RTE-Konfiguration mithilfe des sogenannten *RTEEventToTaskMapping*. Dieses Mapping bildet Events, die das Triggern eines Runnables darstellen, auf OSTasks ab.

RTEEventToTaskMapping

Timing auf Basissoftwareebene

Auf Basissoftwareebene spezifiziert AUTOSAR Module, die für jedes Steuergerät einzeln spezifiziert werden. Am wichtigsten für das Timing Verhalten sind Module, die Einfluss auf die Ausführungsreihenfolge der Runnables haben. Dies ist im Wesentlichen das AUTOSAR Betriebssystem, das auf dem OSEK-Standard* basiert.

Für diese Modellelemente geben wir ein vereinfachtes formales AUTOSAR Architekturmodell wie folgt an:

Definition 14 (AUTOSAR architecture)**.** *Die vereinfachte formale* AUTOSAR Softwarearchitektur $AR = (R, C, VA, T, TRM, p)$ *besteht aus*

1. *einer Menge an Variablenzugriffen (Variable Acess Elemente) VA,*
2. *einer Menge von RunnableEntities*

   $$R \subseteq \{(VA_{read}, VA_{write}, wcet, bcet) \mid VA_{read} \subseteq VA, VA_{write} \subseteq VA, bcet \leq wcet\}$$

   mit VA_{read} einer Menge an Lesezugriffen (Variable Read Accesses), VA_{write} einer Menge an Schreibzugriffen (Variable Write Accesses) mit $VA_{read} \cap VA_{write} = \emptyset$, $wcet \in \mathbb{N}$ der Worst-Case- und $bcet \in \mathbb{N}$ der Best-Case-Execution Time,
3. *einer Menge an AssemblyConnections $C \subseteq \{(left, right) \mid left \in VA, right \in VA\}$, die zwei Variablemzugrifsselemente miteinander verbinden,*
4. *einer Menge periodisch getriggerter Tasks T mit Periode p und*
5. *einem Task-Runnable-Mapping $TRM : R \to T$, welches Runnables auf Betriebssystemtasks abbildet.*

Dieses formale Modell einer AUTOSAR Softwarearchitektur wird im weiteren Verlauf in Abschnitt 5.1 in ein Netzwerk aus Timed Automata transformiert, um das Timing Verhalten formal abzubilden, auf dessen Basis spezifizierte Timing Anforderungen verifiziert werden können.

*http://osek-vdx.org

Formales Modell der AUTOSAR Timing Extensions

Sei $E = \{e_1 \dots e_n\}$ die Menge der *Timing Description Events* und $R = \{re_1, \dots, re_m\}$ die Menge der *Runnable Entities*. Sei weiterhin die Menge aller Timing Constraints definiert als $\mathcal{R} = \{r_1, \dots, r_k\}$. Die in Abschnitt 2.1.2 vorgestellten AUTOSAR Timing Extensions werden dann folgendermaßen formalisiert:

Offset Timing Constraint Der Offset Timing Constraint $r_{otc} = (e_s, e_t, min, max)$ beschränkt die Zeit zwischen einem Source Event $e_s \in E$ und einem Target Event $e_t \in E$ durch die Definition eines minimalen und maximalen Offsets $min, max \in \mathbb{R}$ zwischen diesen beiden Events. Die Summe aller Offset Timing Constraints der Anforderungsmenge $\mathcal{R}$ bezeichnen wir als $\mathcal{R}_{otc} \subseteq \mathcal{R}$.

Latency Timing Constraint Der Latency Timing Constraint $r_{ltc} = (C, min, max)$ beschreibt eine minimale und maximale Latenz im Bereich einer Timing Chain C. Die Timing Chain wird dabei abgebildet auf eine geordnete Sequenz von Timing Events, $C = \langle e_1, \dots, e_n \rangle$, $e_i \in E$ für alle $1 \leq i \leq n$, die in der spezifizierten Zeitspanne auftreten müssen. Die Summe aller Latency Timing Constraints der Anforderungsmenge $\mathcal{R}$ bezeichnen wir als $\mathcal{R}_{ltc} \subseteq \mathcal{R}$.

Synchronization Timing Constraint Der Synchronization Timing Constraint $r_{stc} = (S, t), S \subseteq E$ spezifiziert eine Menge von Timing Events, die simultan innerhalb eines festgelegt Toleranzwertes t auftreten müssen. Die Summe aller Synchronization Timing Constraints der Anforderungsmenge $\mathcal{R}$ bezeichnen wir als $\mathcal{R}_{stc} \subseteq \mathcal{R}$.

Execution Order Constraint Der Execution Order Constraint $r_{eoc} = \langle re_i, \dots, re_j \rangle$ beschränkt die Ausführungsreihenfolge von Runnable Entities. Für jeden Execution Order Constraint $r_{eoc} \in \mathcal{R}_{eoc}$ mit einer geordneten Menge von Runnable Entities $\langle re_1 \dots re_n \rangle$ darf ein nachfolgendes Runnable Entity re_{i+1} nur dann ausgeführt werden, wenn das Runnable Entitie re_i ausgeführt ist. Die Summe aller Execution Order Constraints der Anforderungsmenge $\mathcal{R}$ bezeichnen wir als $\mathcal{R}_{eoc} \subseteq \mathcal{R}$.

Execution Time Constraint Der Execution Time Constraint $r_{etc} = (re, min, max)$ beschränkt nicht das Auftreten von Timing Events, sondern die Ausführungszeit eines Runnable Entities re auf eine minimale Dauer min und eine maximale Dauer max. Runnables können dabei sowohl Runnable Enitites auf der Applikationssebene oder Basissoftwaremodule auf Basissoftwareebene sein. Die Summe aller Execution Time Constraints der Anforderungsmenge $\mathcal{R}$ bezeichnen wir als $\mathcal{R}_{etc} \subseteq \mathcal{R}$.

Die formale Beschreibung der AUTOSAR Timing Extensions wird als Basis für die in den folgenden Kapiteln beschriebenen Transformationen nach SMT (Abschnitt 4.1) und Timed Automata (Abschnitt 5.2) verwendet.

3.3 Integration der Methode in bestehende Entwicklungsprozesse

Im Grundlagenkapitel wurde das V-Modell als aktueller Entwicklungsprozess der Automobilindustrie sowie die Anwendung der AUTOSAR-Methodik innerhalb dieses Prozesses beschrieben. Des Weiteren wurde nun eine Methode zur Verifikation von AUTOSAR Timing Constraints vorgestellt. Diese Methode arbeitet auf der Basis von AUTOSAR-Artefakten und beinhaltet eine Konsistenzprüfung und Verifikation von AUTOSAR Timing Constraints.

Es gibt nun unterschiedliche Möglichkeiten, zu welchem Zeitpunkt oder in welcher Prozessphase Konsistenz- und Timing Analyse durchgeführt werden können. Eine Möglichkeit ist es die Analysen durchzuführen, nachdem die Softwarekomponenten entwickelt und integriert wurden. Zu diesem Zeitpunkt lassen sich dann auch bereits Simulationsmodelle der integrierten Software erstellen und funktionale Anforderungen testen. Ebenfalls lassen sich dann Worst-Case Execution Times aus einer Taskanalyse innerhalb der Timing Analyse verwenden, sodass die Analyse hierdurch die Laufzeiten der Zielhardware widerspiegelt. Der Nachteil an diesem Vorgehen ist, dass zunächst erste Implementierungen für alle Softwarekomponenten existieren müssen und zusätzlich WCET-Analysen

durchgeführt werden müssen (siehe Abschnitt 5.2). Dies ist jedoch erst sehr spät im Entwicklungsprozess der Fall. Für diese WCET-Analysen ist es ebenfalls neben dem zusätzlichen Zeit- und Kostenaufwand notwendig, die Zielhardware der Software bereits zu kennen. Daher ist dieses Vorgehen zumindest aus der Sicht einer frühzeitigen Absicherung der Timing Anforderungen eher unvorteilhaft.

Eine weitere Möglichkeit ist es, die Timing Analyse bereits durchzuführen, sobald die ersten formalisierten Timing Anforderungen und das Architekturmodell vorliegen. Eine Analyse lässt sich so bereits durchführen ohne dass Implementierungen für Softwarekomponenten vorliegen. So kann die erste Timing Analyse bereits sehr früh durchgeführt werden. Eine erste Einschätzung der Laufzeit einzelner Runnable Entities muss dann allerdings von Experten vorgenommen werden. Noch früher lässt sich beispielsweise eine erste Konsistenzanalyse durchführen. Hierfür werden weder Quellcode, noch Details des AUTOSAR-Modells benötigt, sondern lediglich die AUTOSAR Timing Constraints. Daher wird auch im Sinne einer frühzeitigen Absicherung der Timings vorgeschlagen, bei der Entwicklung bereits frühzeitig die Modellierung von Timing Constraints durchzuführen, um die anschließende Konsistenzanalyse einsetzen zu können.

Abbildung 3.2 zeigt die Integration an zwei verschiedenen Stellen im Entwicklungsprozess. Auf der linken Seite wird die Konsistenzanalyse und Verifikation auf der Basis von Laufzeitschätzungen für einzelne Runnables vorgenommen. Ist die Anforderungsmenge konsistent und werden die festgelegten Timing Constraints eingehalten, kann das Ergebnis als Timing Budget für die Implementierung der Softwarekomponenten herangezogen werden. Auf der rechten Seite wird die Konsistenzanalyse und Verifikation erst nach der Implementierung gemacht. In diesem Fall kann auf zuvor durchgeführte Taskanalysen zurückgegriffen und somit die Laufzeiten von Runnable Entities festgelegt werden. Führt die Verifikation zu dem Ergebnis, dass nicht alle Timing Constraints eingehalten werden können, muss die Implementierung der Runnable Entities angepasst werden.

Im Kontext der AUTOSAR-Entwicklungsmethodik ist es für die Verifikation notwendig mindestens eine Softwarearchitektur auf VFB-Ebene zu modellieren, alle Softwarekomponenten mit den enthaltenen Runnables zu

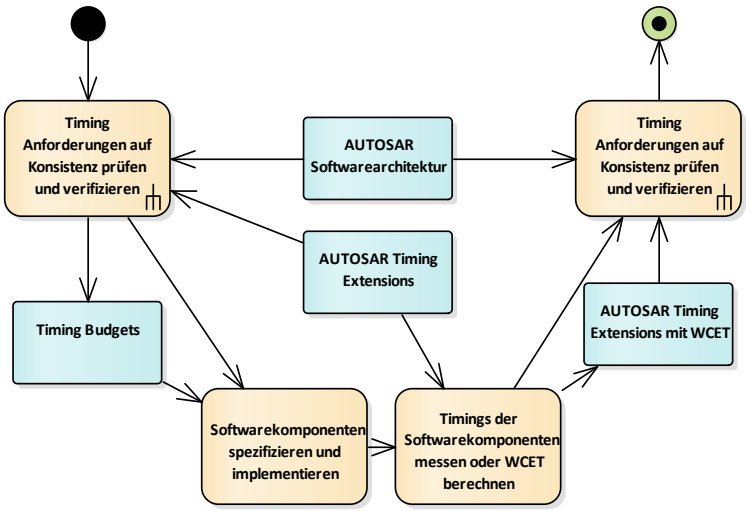

Abbildung 3.2: Integration der Methode in den Entwicklungsprozess

modellieren, sowie eine Konfiguration der RTE zu erstellen und die Basissoftwaremodule des Betriebssystems zu konfigurieren.

3.4 Zusammenfassung

In diesem Abschnitt wurde ein Konzept zur Verifikation von Timing Anforderungen vorgestellt. Dazu wurde zunächst in Abschnitt 3.1 ein Prozess definiert, der die einzelnen Schritte beschreibt. Weiterhin wurden die Zeitaspekte einer AUTOSAR Softwarearchitektur und der Timing Anforderungen betrachtet und relevante Teile in Abschnitt 3.2 formalisiert. Des Weiteren wurde in Abschnitt 3.3 vorgestellt wie sich das Konzept im Rahmen des etablierten Softwareentwicklungsprozesses verhält und welche Vor- und Nachteile dieser Ansatz hat. Das Konzept besteht im Kern aus zwei Teilen: einer Methode zur Konsistenzprüfung der Timing Anforderungen mittels SMT-Formeln und einer Methode zur Verifikation mittels Timed Automata. Diese beiden Methoden werden in den folgenden beiden Kapiteln detaillierter betrachtet.

4

Konsistenzanalyse von AUTOSAR Timing Anforderungen

Während der Entwicklung eingebetteter automotiver Softwaresysteme werden Timing Anforderungen auf Benutzerebene nach und nach weiter verfeinert bis sie schließlich als AUTOSAR Timing Anforderungen die zeitlichen Constraints im Kontext der AUTOSAR Softwarearchitektur beschreiben. Bevor diese Anforderungen nun verifiziert werden, werden sie hinsichtlich ihrer Konsistenz überprüft, da eine konsistente Anforderungsmenge Grundvoraussetzung dafür ist, dass alle Anforderungen erfüllbar sind.

In diesem Kapitel wird die Methode zur Konsistenzanalyse von AUTOSAR Timing Anforderungen vorgestellt. Diese repräsentiert den ersten Schritt im Gesamprozess aus Abschnitt 3.1. Die Analyse wird dabei ausschließlich auf den Timing Anforderungen durchgeführt und nicht auf dem dazugehörigen Architekturmodell. Dadurch ist es möglich, bereits vor der Verfügbarkeit eines vollständigen Architekturmodells, die Konsistenz der

Anforderungen zu überprüfen. Die Konsistenzanalyse wird durchgeführt, indem alle Timing Anforderungen in *SMT-Formeln* mit linearer Arithmetik transformiert werden und auf Erfüllbarkeit hin untersucht werden. Dieser logische Ansatz wurde gewählt, da er die zeitlichen Beschränkungen, die durch die Timing Anforderungen entstehen, intuitiv beschreibt. Ebenfalls ist durch die breite Verfügbarkeit effizienter Werkzeuge zum Lösen von SMT-Formeln zu erwarten, dass auch große Anforderungsmengen mit diesem Ansatz geprüft werden können.

SMT-Formeln

Zunächst wird die Transformation der in Abschnitt 3.2.1 vorgestellten formal beschriebenen AUTOSAR Timing Anforderungen in eine SMT-Formel vorgestellt und am Blinkerbeispiel aus Abschnitt 2.1.2 erläutert. Danach werden Verfahren zur Visualisierung inkonsistenter Anforderungsmengen beschrieben, um eine einfache Korrektur zu ermöglichen. Dieses Kapitel basiert zum Großteil aus den bereits veröffentlichten Arbeiten in Beringer und Wehrheim (2020).

Wie in Definition 3 beschrieben, ist eine Anforderungsmenge genau dann konsistent, wenn keine Teilmenge daraus in Konflikt zueinander steht. Das bedeutet auch, dass die Anforderungsmenge konsistent ist, solange ein Modell oder System existiert, dass alle Anforderungen erfüllen kann.

Definition 15 (Konsistenz)**.** *Sei $\mathcal{R} = \{r_1, \ldots, r_n\}$ die Menge der Timing Anforderungen in einem AUTOSAR Modell und sei $\mathcal{M}$ die Menge aller Modelle. Dann ist $\mathcal{R}$ konsistent gdw. $\exists \mathcal{M} \in \mathcal{M} : \forall r_i \in \mathcal{R} : \mathcal{M} \models r_i$.*

In unserem Fall besteht die Menge aus allen Modellen, die mithilfe des AUTOSAR Metamodells definiert werden können und die Anforderungen sind definiert als AUTOSAR Timing Constraints. Daher ist die Menge an AUTOSAR Timing Constraints genau dann konsistent, wenn es ein AUTOSAR Modell gibt, dass alle durch die Timing Anforderungen erzeugten Restriktionen erfüllt. Da die Timing Constraints jedoch ausschließlich das zeitliche Verhalten des Modells beschränken, ist es ausreichend, wenn ausschließlich die Timing Events für eine Analyse betrachtet werden. Das Finden einer konsistenten Anforderungsmenge wird daher algorithmisch reduziert auf das Finden einer validen Ausführungsreihenfolge von Timing Events.

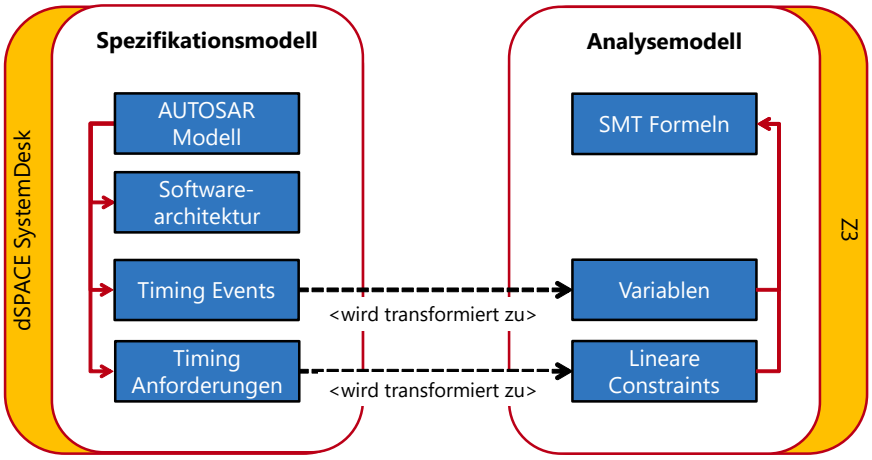

Abbildung 4.1: Transformation von AUTOSAR Modellen nach SMT-Formeln

4.1 Transformation der AUTOSAR Timing Anforderungen nach SMT

In diesem Abschnitt beschreiben wir die Transformation von AUTOSAR-Timing Anforderungen in logische Formeln für einen SMT Solver. Die SMT-Formeln spiegeln die durch die Timing Anforderungen zuvor spezifizierten zeitlichen Beschränkungen wider, sodass die Erfüllbarkeit der Formeln mit der Existenz einer gültigen Ordnung von Timing Events, die alle Timing Requirements erfüllt, äquivalent ist.

Das abstrakte Konzept ist in Abbildung 4.1 beschrieben. Das AUTOSAR-Modell enthält das Systemmodell, die Timing Events, die über Assoziationen mit dem Systemmodell verknüpft sind, und die Timing Constraints. Die Timing Events werden in SMT-Variablen transformiert und die Timing Constraints in lineare Constraints über die generierten Variablen. Das Systemmodell für die Konsistenzanalyse wird *nicht* benötigt. Die Transformationen wurden automatisiert und werden direkt auf das AUTOSAR-Modell in SystemDesk angewendet.

Einige Timing Constraints basieren nicht ausschließlich auf Timing Events, sondern auch auf Runnable Entities, zum Beispiel der Execution Order Constraint. Um Inkonsistenzen zwischen diesen Timing Constraints

und Timing Constraints auf Basis von Timing Events zu erkennen, werden Timing Constraints auf der Grundlage von Runnable Entities so transformiert, dass sie ebenfalls auf Timing Events basieren. Dies ist möglich, da AUTOSAR Timing Events zur Verfügung stellt, die im Zusammenhang mit Runnable Entities stehen, wie beispielsweise das Starten oder die Terminierung eines Runnable Entities. Daher gehen wir im Folgenden davon aus, dass für jedes Runnable Entity $re \in \mathit{RunnableEntities}$ entsprechende Timing Events e^s_{re} und e^t_{re} existieren, die Ereignisse für den Ausführungsbeginn und die Terminierung des Runnable Entities darstellen. Einige Timing Constraints schränken auch das Auftreten zwischen Timing Events ein, sodass eine untere und obere Schranken selbst durch das Auftreten eines anderen Timing Events definiert werden kann. Dennoch können alle Beschränkungen für Timing Events als lineare Constraints definiert werden. Deswegen kann die Existenz eines gültigen Modells überprüft werden, indem nach einer gültigen Belegung $t : E \to \mathbb{R}_{\geq 0}$ von Timing Events mittels eines SMT-Solvers mit linearer Arithmetik gesucht wird.

Für ein gegebenes AUTOSAR Modell mit Timing Events $E = \{e_1 \dots e_n\}$ und Timing Constraints $\mathcal{R} = (r_1, \dots, r_m)$ wird eine SMT-Formel $\mathcal{F} = \bigwedge_{i=1}^{n} f_{r_i}$ sukzessive konstruiert. Jeder Term in $\mathcal{F}$ ist entweder ein konstanter Wert, wie beispielsweise ein Toleranzwert oder Minimum-/Maximum, oder eine Variable, die einem Timing Event e entspricht und eine erfüllbare Zuweisung in den Zeitbereich $t : E \to \mathbb{R}_{\geq 0}$ benötigt. Im Folgenden werden die Transformationen für jeden Typ von Timing Constraint genauer dargestellt.

Offset Timing Constraint Ein Offset Timing Constraint $r_{otc} = (e_s, e_t, min, max)$ beschränkt die Zeit zwischen dem Auftreten eines Source Timing Events e_s und einem Target Timing Event e_t. Daher werden für jeden Offset Timing Constraint $r_{otc} \in \mathcal{R}_{otc}$ zwei Ausdrücke zur SMT-Formel wie folgt hinzugefügt:

$$f_{r_{otc}} = e_s + max \geq e_t \wedge e_s + min \leq e_t. \tag{4.1}$$

Latency Timing Constraint Um zu verifizieren, dass ein Latency Timing Constraint nicht mit anderen Timing Constraints im Konflikt steht, beispielsweise mit einem Offset Timing Constraint, muss ein gültiges Auftreten von Timing Events vorliegen, wobei eine Sequenz von Timing Events vorliegen muss, die nicht den Maximalwert des Latency Timing Constraints überschreitet. Daher wird für jeden $r_{ltc} \in \mathcal{R}_{ltc}$ mit einer Event Chain $C = \langle e_1, \ldots, e_n \rangle$ die SMT-Formel wie folgt erweitert:

$$f_{r_{ltc}} = \forall i, 1 \leq i \leq n-1 : e_i \leq e_{i+1} \wedge min \leq e_n - e_1 \leq max. \tag{4.2}$$

Synchronization Timing Constraint Ein Synchronization Timing Constraint $r_{stc} = (S, tolerance), S \subseteq E$ spezifiziert eine Menge von Timing Events, die gleichzeitig mit einem Toleranzwert auftreten müssen. Für jeden Synchronization Timing Constraint $r_{stc} \in \mathcal{R}_{stc}$ wird daher der Toleranzwert für jede Kombination von Timing Events geprüft. Die SMT-Formel wird daher folgendermaßen erweitert:

$$f_{r_{stc}} = \forall e, e' \in S : e - e' \leq tolerance. \tag{4.3}$$

Execution Order Constraint Für jeden Execution Order Constraint $r_{eoc} \in \mathcal{R}_{eoc}$ mit einer Sequenz von Runnable Entities $\langle re_1, \ldots, re_n \rangle$ gilt, dass ein nachfolgendes Runnable Entity re_{i+1} nur dann ausgeführt werden darf, wenn das vorherige Runnable Entity re_i terminiert ist. Daher gilt $\forall i, 1 \leq i < n : re_i \leq re_{i+1}$. Da unser Ansatz daruf beruht eine gültige Zuweisung basierend auf den Timing Events zu finden, müssen wir zunächst eine Modellierungsalternative finden, die äquivalent ist, aber stattdessen auf Timing Events anstelle von Runnable Entities basiert. Daher modellieren wir die beschränkte Ausführung der Runnable Entities durch Timing Events, die das Starten und die Terminierung eines entsprechenden Runnable Entities repräsentieren und beschränken diese Timing Events so, dass das Startevent eines nachfolgenden Runnable Entities später erfolgen muss als das Terminierungsevent des vorherigen Runnable Entites. Sei $e^s_{re_i}$ und $e^t_{re_i}$ das Startevent und das Terminierungsevent eines Runnable Entities

re_i. Dann erhalten wir die folgende SMT-Formel:

$$\forall i, 1 \leq i \leq n-1 : e^s_{re_i} \leq e^t_{re_i} \wedge e^t_{re_i} \leq e^s_{re_{i+1}}. \tag{4.4}$$

AUTOSAR ermöglicht auch die Anwendung von Execution Order Constraints auf Basissoftwaremodule. Die Benennung von Timing Events ist dann anders, die vorgestellte allgemeine Transformation kann aber analog angewendet werden.

Execution Time Constraint Ein Execution Time Constraint beschreibt eine Beschränkung der Ausführungszeit auf der Basis lediglich eines AUTOSAR-Elements. Nichtsdestotrotz entspricht eine Beschränkung der Ausführungszeit eines Runnable Entities einem Offset Timing Constraint zwischen dem Ausführungsstart und der Terminierung des assoziierten Runnable Entities. Daher entspricht ein Execution Time Constraint einem Offset Timing Constraint $r_{etc} = (r, min, max) = r_{otc}$ mit $r_{otc} = (e_{r_s}, e_{r_t}, min, max)$, wobei e_{r_s} und e_{r_t} die Timing Events sind, die für den Start bzw. die Terminierung des Runnable Entities stehen.

Die Transformation von AUTOSAR Timing Requirements nach SMT wurde beispielhaft für das Blinkerbeispiel aus Abschnitt 2.1.2 durchgeführt. Für den Execution Order Constraint r_{eoc} gilt, dass die Start-Events jedes Runnables vor dem jeweiligen Terminierungs-Event stattfinden müssen und das Start-Event für das Runnable Logic vor dem Terminierungs-Event des Runnables TssPreprocessing stattfinden muss. Analog dazu muss das Terminierungs-Event des Runnables Logic vor dem Start-Event des Runnables Toggle auftreten. In diesem Zusammenhang erzeugt unser Ansatz fünf Ungleichungen, die die Zeitvorgaben wie beschrieben einschränken. Für den Offset Timing Constraint r_{otc} werden zwei Ungleichungen generiert, die den minimalen und maximalen Offset für die entsprechenden Timing Events einschränken. Schließlich werden für die Execution Time Constraints r_{etc} und r_{etc_2} jeweils zwei Klauseln generiert, die das Timing für den Start und die Terminierung des Logic Runnables und Toggle Runnables einschränken. Die generierten Ungleichungen für die einzelnen Timing Constraints sind in Tabelle 4.1 zusammengefasst.

In diesem Beispiel ist die Anforderungsmenge nicht konsistent, da die generierte Menge an Klauseln unerfüllbar ist. So kann beispielsweise für die Menge $\{f_1, f_2, f_4, f_7, f_8\}$ keine erfüllbare Belegung gefunden werden. So gilt für f_1, f_2, f_4 dass diese in einer zeitlichen Abfolge auftreten müssen: $e^s_{\text{TssPreprocessing}} \leq e^t_{\text{TssPreprocessing}} \leq e^s_{\text{Logic}} \leq e^t_{\text{Logic}}$. Die Klausel $f_7 = e^s_{\text{TssPreprocessing}} + 4 \geq e^t_{\text{Logic}}$ beschränkt den zeitlichen Abstand zwischen dem ersten Event $e^s_{\text{TssPreprocessing}}$ und dem letzten Event e^t_{Logic} auf *maximal* 4 Zeiteinheiten. Eine mögliche Belegung für diese Klauseln wäre $e^s_{\text{TssPreprocessing}} := 1, e^t_{\text{TssPreprocessing}} := 2, e^s_{\text{Logic}} := 3, e^t_{\text{Logic}} := 4$. $f_8 = e^s_{\text{Logic}} + 10 \leq e^t_{\text{Logic}}$, beschränkt jedoch den Zeitabstand zwischen den beiden Events e^s_{Logic} und e^t_{Logic} auf *mindestens* 10 Zeiteinheiten, sodass dadurch die vorgeschlagene Belegung ungültig ist, da durch die Belegung $e^s_{\text{Logic}} := 3, e^t_{\text{Logic}} := 4$ der minimale Zeitabstand nicht eingehalten wird. Eine alternative Belegung mit beispielsweise $e^s_{\text{Logic}} := 3, e^t_{\text{Logic}} := 13$ macht zwar f_8 erfüllbar, jedoch wird dadurch in jedem Fall die Klausel f_7 verletzt, da nun der minimale Zeitabstand zwischen dem ersten Event $e^s_{\text{TssPreprocessing}} = 1$ und dem letzten Event $e^t_{\text{Logic}} = 13$ größer als 4 ist. Die Klauseln sind somit unerfüllbar und die Menge der Timing Constraints ist inkonsistent.

Die vorgestellte Methode ermöglicht es, zeitliche Inkonsistenzen in Anforderungsmengen zu erkennen. Das zuvor gezeigte Beispiel zeigt jedoch, dass es schon bei wenigen Anforderungen schwierig ist, die Gründe für die Inkonsistenz nachzuvollziehen. Dies erschwert eine Interpretation der Ergebnisse und eine effiziente Korrektur der Anforderungen ist somit nicht möglich. Daher werden im Folgenden Verfahren vorgestellt, die die Identifikation von Ursachen für Inkonsistenzen unterstützen, um die Korrektur der Anforderungen zu beschleunigen.

4.2 Verfahren zur Korrektur inkonsistenter Anforderungsmengen

In diesem Abschnitt werden Methoden zur Korrektur inkonsistenter Anforderungsmengen vorgestellt. Dazu betrachten wir zuerst formale Methoden, die sich mit der Berechnung von für die Auflösung von Inkonsistenzen re-

Tabelle 4.1: Beispieltransformationen

Anforderung	Generierte Formeln
$r_{eoc} =$ $\langle$TssPreprocessing, Logic, Toggle$\rangle$	$f_1 = e^s_{\text{TssPreprocessing}} \leq e^t_{\text{TssPreprocessing}}$ $f_2 = e^s_{\text{Logic}} \leq e^t_{\text{Logic}}$ $f_3 = e^s_{\text{Toggle}} \leq e^t_{\text{Toggle}}$ $f_4 = e^t_{\text{TssPreprocessing}} \leq e^s_{\text{Logic}}$ $f_5 = e^t_{\text{Logic}} \leq e^s_{\text{Toggle}}$
$r_{otc} = (e^s_{\text{TssPreprocessing}}, e^t_{\text{Logic}}, 3, 4)$	$f_6 = e^s_{\text{TssPreprocessing}} + 3 \leq e^t_{\text{Logic}}$ $f_7 = e^s_{\text{TssPreprocessing}} + 4 \geq e^t_{\text{Logic}}$
$r_{etc} = (\text{Logic}, 10, 30)$	$f_8 = e^s_{\text{Logic}} + 10 \leq e^t_{\text{Logic}}$ $f_9 = e^s_{\text{Logic}} + 30 \geq e^t_{\text{Logic}}$
$r_{etc_2} = (\text{Toggle}, 1, 5)$	$f_{10} = e^s_{\text{Toggle}} + 1 \leq e^t_{\text{Toggle}}$ $f_{11} = e^s_{\text{Toggle}} + 5 \geq e^t_{\text{Toggle}}$

levanten Teilmengen der zuvor generierten SMT-Formel befassen. Dies ist die Berechnung der maximal großen Menge an konsistenten Anforderungen mit Hilfe eines Algorithmus zum Lösen von MaxSMT und der Berechnung einer minimal großen Menge inkonsistenter Anforderungen mit Hilfe eines Algorithmus zur Berechnung des Unsat Core. Danach stellen wir eine Visualisierungsmethode vor, die die zuvor berechneten Teilmengen in den Gesamtkontext der SMT-Formel einbezieht und somit zusätzlich eine intuitive Möglichkeit zur Auflösung von Inkonsistenzen darstellt.

4.2.1 Identifikation von Ursachen für Inkonsistenzen

Für die Korrektur einer inkonsistenten Anforderungsmenge ist es notwendig, dass einige der enthaltenen Anforderungen geändert oder aus der Menge entfernt werden müssen. Die Identifikation dieser Anforderungen ist jedoch bei größeren Mengen schwierig. Daher ist es notwendig einem Requirements Engineer Hinweise zu geben, welche Anforderungen möglicher-

weise die Inkonsistenz auslösen oder zumindest die Menge der ursächlichen Anforderungen einschränken. Dies können zum einen Anforderungen sein, die außerhalb einer maximal großen konsistenten Teilmenge liegen. Ist die konsistente Teilmenge hinreichend groß und damit die Differenzmenge klein, so kann durch das Ändern oder Entfernen dieser Teilmenge die Konsistenz sichergestellt werden. Zum anderen kann durch die Identifikation minimaler unerfüllbarer Teilmengen eine kleine inkonsistente Teilmenge identifiziert werden, bei der ein Requirements Engineer durch das Löschen oder Ändern einer der in dieser Teilmenge enthaltenen Anforderungen ebenfalls die Konsistenz sicherstellen kann.

Die im Folgenden vorgestellten Methoden verwenden die Lösungen von MaxSMT zur Identifikation maximal großer Teilmengen und Unsat Core zur Identifikation minimaler inkonsistenter Teilmengen, die auf der Grundlage der generierten SMT-Formel $\mathcal{F}$ berechnet wurden. Um eine Rückverfolgbarkeit von SMT-Klauseln zu den Timing Anforderungen zu haben, speichern wir zusätzlich eine Abbildung $w : \mathcal{F} \rightarrow \mathcal{R}$, die jede SMT-Formel f auf die entsprechende Timing Anforderung aus $\mathcal{R}$ abbildet.

MaxSMT

Die Lösung von MaxSMT für die generierte SMT-Formel ermöglicht es, die maximal große Menge an Klauseln $\mathcal{I}$ zu finden, die erfüllbar sind. Diese lassen sich dann in einem nächsten Schritt auf die zugehörigen Timing Anforderungen abbilden und somit als konsistent identifizieren. Mithilfe dieser Informationen kann ein Requirements Engineer Fehler in der Anforderungsmenge korrigieren. Dies geschieht dadurch, dass er Timing Anforderungen, deren zugehörige SMT-Klauseln nicht in der MaxSMT Menge enthalten sind, entweder aus der Anforderungsmenge entfernt oder bearbeitet bzw. abschwächt, indem er größere Bereiche für die Timing Constraints wählt.

Eine MaxSMT Teilmenge für die aus der Beispieltransformation in Tabelle 4.1 erstellten SMT-Formeln ist $\mathcal{I} = \mathcal{R} \setminus \{f_7\}$. Der zu f_7 gehörige Timing Constraint ist r_{etc}. Ein Requirements Engineer bekommt somit die Information, dass alle Anforderungen ohne $w(f_7) = r_{etc}$ konsistent sind. Die zu

betrachtenden Anforderungen wurden somit von vier auf eine reduziert. Er kann jetzt prüfen inwieweit der Execution Time Constraint korrekt modelliert wurde und entscheiden, ob die darin festgelegte minimale Ausführungszeit fehlerhaft ist und diese aus dem Constraint entfernen.

Unsat Core

Die Anwendung des Minimal Unsat Core ermöglicht es für eine SMT-Formel eine Teilmenge $UC \subseteq F$ zu finden, die unerfüllbar ist. Durch die Abbildung auf die Timing Constraints lässt sich somit die Menge an Timing Constraints bestimmen, die inkonsistent sind. Im Idealfall besteht diese Menge nicht aus allen Timing Anforderungen, sodass ein Requirements Engineer während der Identifikation und Korrektur nicht alle Timing Anforderungen betrachten muss.

Beide Methoden können unabhängig voneinander oder auch in Kombination angewendet werden, indem beispielsweise die Ergebnismenge $\mathcal{F} \setminus \mathcal{I}$ als Eingabe für die Berechnung des Unsat Cores genommen wird.

In unserem Beispiel aus Tabelle 4.1 ist ein Unsat Core $UC = \{f_1, f_4, f_7, f_8\}$. Für diese Teilmenge existiert keine erfüllbare Ordnung der Timing Events $e^s_{\text{TssPreprocessing}}$, $e^t_{\text{TssPreprocessing}}$, e^s_{Logic} und e^t_{Logic}. Da das Entfernen einer Klausel aus UC die Formel erfüllbar machen würde, ist der gefundene Unsat Core ebenfalls ein Minimal Unsat Core. Da keine Klausel aus UC zur Timing Anforderung r_{etc_2} gehört, kann diese Timing Anforderung nicht für die Inkonsistenz verantwortlich sein. Auf diese Weise können wir hier den Unsat Core verwenden, um die Anforderungsartefakte einzuschränken, die wir während der Korrektur betrachten müssen.

4.2.2 Visualisierung von (In-)Konsistenz

Ein entscheidender Punkt für die Identifikation von Ursachen für Inkonsistenzen ist eine adäquate Ergebnisvisualisierung der generierten MaxSMT und Unsat Core Klauseln. In diesem Zusammenhang stellen wir eine graphbasierte Visualisierung von MaxSMT und Unsat Core vor, wobei die Knoten

im Graphen die zeitlichen Beziehungen (d.h. die SMT-Klauseln) zwischen den Events darstellen.

Definition 16 (Ergebnisgraph)**.** *Gegeben sei eine Menge an Timing Events E und eine Menge an transformierten Anforderungsformeln $\mathcal{F}$, dann ist der Ergebnisgraph $G = (G_V, G_E, c_{min}, c_{max})$ folgendermaßen konstruiert:*

- *$G_V = E$ ist die Menge an Knoten, wobei jeder Knoten jeweils ein Timing Event darstellt,*
- *$G_E \subseteq G_V \times G_V$ ist die Menge der gerichteten Kanten, wobei gilt: $(e, e') \in G_E$ wenn die Menge $\mathcal{F}$ die Formel beinhaltet: $e \leq e'$ or $e + min \leq e'$ or $e + max \geq e'$ für beliebige $min, max \in \mathbb{R}$,*
- *$c_{min} : G_E \to 2^{\mathbb{R}}$ ist die Beschriftungsfunktion für die minimale Zeitdistanz, wobei $min \in c_{min}(e, e')$ wenn die Menge $\mathcal{F}$ eine Formel $e + min \leq e'$ beinhaltet und*
- *$c_{max} : G_E \to 2^{\mathbb{R}}$ ist die Beschriftungsfunktion für die maximale Zeitdistanz, wobei $max \in c_{max}(e, e')$ wenn die Menge $\mathcal{F}$ die Formel $e + max \geq e'$ beinhaltet.*

Für MaxSMT und Unsat Core werden zwei Ergebnisgraphen separat visualisiert. Im MaxSMT Ergebnisgraphen G_{max} werden die Knoten grün dargestellt, wenn die korrespondierenden Klauseln, die die entsprechenden Timing Beziehungen zwischen den Timing Events darstellen, in der MaxSMT-Teilmenge enthalten sind, ansonsten werden sie rot dargestellt. Im Unsat Core Ergebnisgraphen G_{uc} werden Knoten grün dargestellt, wenn sie nicht zum Unsat Core UC gehören, ansonsten werden sie rot dargestellt.

Abbildung 4.2 zeigt die grafische Darstellung der (generierten) Ereignisse und zeitlichen Beziehungen zwischen ihnen unter Verwendung der Beispieltransformationen aus Tabelle 4.1. Timing Events, die zu erfüllbaren Timing Anforderungen gehören, sind in grün dargestellt, während Timing Events, die zu nicht erfüllbaren Timing Anforderungen gehören, in rot dargestellt werden. Anhand dieses Beispiels lassen sich sehr deutlich die Vorteile des grafischen Ansatzes zeigen: Im Gegensatz zur textuellen Darstellung kann ein Anwender sofort erkennen, dass

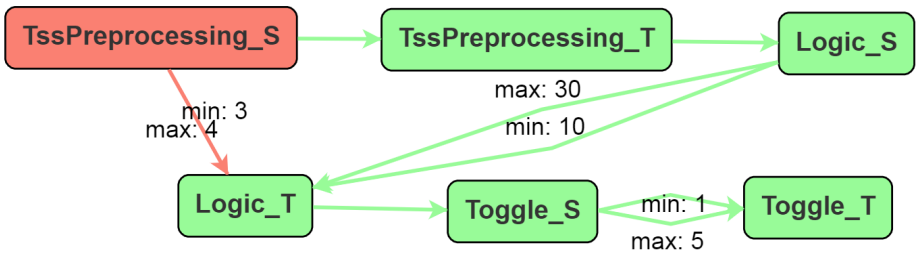

Abbildung 4.2: Generierter Ergebnisgraph für MaxSMT

- das Timing Event $e^s_{\text{TssPreprocessing}}$ das einzige Timing Event ist, dass innerhalb der MaxSMT Menge nicht vorhanden ist,
- eine Timing Relation zwischen $e^s_{\text{TssPreprocessing}}$ und e^t_{Logic} nicht in MaxSMT vorhanden ist und somit vermutlich die Inkonsistenz erzeugt und
- mehrere Pfade von $e^s_{\text{TssPreprocessing}}$ nach e^t_{Logic} existieren, die vermutlich im Konflikt zueinander stehen.

Weiterhin kann ein Anwender aus diesen Erkenntnissen bereits folgende Schlussfolgerungen ziehen bzw. Handlungsalternativen ableiten:

- Da die Timing Relation zwischen $e^s_{\text{TssPreprocessing}}$ und e^t_{Logic} durch einen einzelnen Offset Timing Constraint erzeugt wurde, würde durch das Entfernen dieses Constraints die Menge konsistent werden.
- Da mehrere Pfade von $e^s_{\text{TssPreprocessing}}$ nach e^t_{Logic} existieren, kann es auch eine Lösung sein, den anderen (längeren) Pfad genauer zu betrachten. Das Entfernen oder Unterbrechen des Pfades würde die Menge ebenfalls konsistent machen.
- Schließlich lässt sich anhand der Beschriftung beider Pfade der erforderliche Zeitabstand zwischen den Timing Events ablesen. Dies lässt vermuten, dass durch das Abschwächen der Timing Relation bzw. das Vergrößern der Zeitbeschränkungen auf einem oder beiden Pfaden eine konsistente Anforderungsmenge gebildet werden kann.

Abbildung 4.3 zeigt den Unsat Core Ergebnisgraph G_{uc}. Im Gegensatz zu G_{max} zeigt dieser Graph, dass die Timing Constraints mit den Timing Events

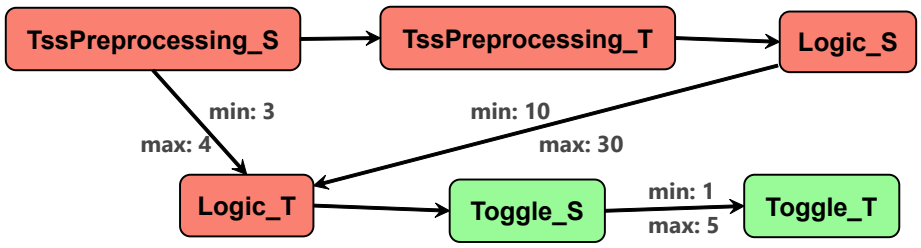

Abbildung 4.3: Generierter Ergebnisgraph für Unsat Core

$e^s_{\text{TssPreprocessing}}$, $e^t_{\text{TssPreprocessing}}$, e^s_{Logic} und e^t_{Logic} aufgrund ihrer Abhängigkeiten in Konflikt zueinander stehen. An dieser Darstellung lässt sich erkennen, dass die Inkonsistenz durch das Entfernen eines beliebigen Constraints innerhalb des rot markierten Pfades aufzulösen ist.

4.3 Stand der Technik

Für die automatisierte Konsistenzprüfung von AUTOSAR-basierten Timing Anforderungen existieren nach aktuellem Stand keine direkten verwandten Arbeiten. Es gibt jedoch Arbeiten, die Timing Anforderungen mithilfe anderer Sprachen beschreiben. Weiterhin gibt es Arbeiten, die sich mit dem Begriff der Modellkonsistenz im Allgemeinen und im Besonderen für UML-basierte Sprachen beschäftigen. Da Anforderungen zunächst nur in natürlichsprachlicher Form vorliegen, gibt es darüber hinaus Arbeiten, die sich auf sprachlicher Ebene mit dem Begriff der Konsistenz auseinandersetzen. Im Folgenden werden die wichtigsten Arbeiten vorgestellt:

Sprachen zur Spezifikation von Zeitbeschränkungen Es existieren mehrere Methoden und Sprachen zur formalen Spezifikation von Zeitbeschränkungen, z.B. die *Clock Constraint Specification Language (CCSL)* (André, 2009, Mallet und Simone, 2015), das von der EAST-ADL Association (2013) für Anwendungsfälle im Automobilbereich übernommen wurde, *Timed Story Scenario Diagrams (TSSD)* (Klein und Giese, 2007), *Universal Pattern Language (UPL)* (Teige et al., 2016), *Structured Assertion Language*

for Temporal Logic (SALT) (Bauer et al., 2006) oder Echtzeit-Mustersprachen (Gruhn und Laue, 2006). Für die Verifikation definieren die Ansätze in der Regel Transformationen nach Logiken auf niedrigen Ebenen wie TLTL, MTL oder TCTL oder in Observer, die auf ausführbarem C-Code basieren. In unserem Ansatz haben wir uns auf den weit verbreiteten Standard AUTOSAR und die integrierten AUTOSAR Timing Constraints konzentriert.

Modellkonsistenz Methoden, die die Konsistenz von Modellierungsartefakten überprüfen, z.B. UML-Modelle, werden in Rasch und Wehrheim (2003), Seifert et al. (2005), Kotb und Katayama (2005), Simmonds und Bastarrica (2005), Kalibatiene et al. (2013), Derrick et al. (2002), Abdelhalim et al. (2011) oder für SysML State Machines in Jacobs und Simpson (2017) vorgestellt. Üblicherweise wird Konsistenz als eine Eigenschaft zwischen verschiedenen Modelldiagrammsichten, verschiedenen Modellversionen (Engels et al., 2002) oder Modellen auf verschiedenen Abstraktionsebenen behandelt (Mens et al., 2005, Engels et al., 2001). Darüber hinaus wird Konsistenz auch zwischen verschiedenen Modelltypen betrachtet, wenn diese semantische Abhängigkeiten haben. So wird beispielsweise in Engels et al. (2008) die Konsistenz zwischen Business Process Models und Web Services betrachtet, da die Business Processes auf Web Services abgebilet werden, beide aber unterschiedliche Reihenfolgeabhängigkeiten haben können. Diese Ansätze unterscheiden sich von unserem Ansatz, da wir nur an der Konsistenz einer Menge von Anforderungen interessiert sind, wobei diese insgesamt in einem einzigen Modell erfasst wird, welches immer dem AUTOSAR-Metamodell entspricht. Viele Arten von Inkonsistenzen, zum Beispiel syntaktische Inkonsistenzen, treten daher nicht auf, da die syntaktische Konsistenz durch das Modellierungswerkzeug sichergestellt wird. Weitere Regeln, die die statische Semantik von AUTOSAR beschreiben, sind jedoch textuell definiert und werden nur teilweise überprüft.

In Post et al. (2011b) und Post et al. (2011a) werden Systemanforderungen in der Echtzeitlogik Duration Calculus (Chaochen et al., 1991) formuliert. Die Verifikation der Konsistenzeigenschaft erfolgt durch die Transformation jeder Anforderung in einen sogenannten Phase Event Automaton (PEA)

und anschließend in einen UPPAAL-verifizierbaren zeitgesteuerten Automaten.

Die Verifikation von zeitgesteuerten Automaten-basierten Echtzeitsystemen wird z.B. in Kim et al. (2015) vorgestellt, wo ein neuartiges Analyseframework gezeigt wird, das symbolische und statistische Modellüberprüfung in UPPAAL kombiniert, um so die Verifikationslaufzeit zu beschleunigen. Ein ähnlicher Ansatz für die Konsistenzprüfung von Zeitanforderungen wird in Toennemann et al. (2018) vorgestellt. Im Gegensatz zu unserem Ansatz verwenden die Autoren Timing Anforderungen und ein Systementwurfsmodell zur Konsistenzprüfung. Dahingegen schlagen wir einen mehrstufigen Ansatz vor, bei dem wir zunächst nur die Anforderungsmenge selbst für die Konsistenzprüfung verwenden und erst dann die Systemarchitektur zu Verifikationszwecken verwenden.

Konsistenz natürlichsprachlicher Anforderungsartefakte Darüber hinaus existieren Methoden, die Inkonsistenzen in natürlichsprachlichen oder semi-formalen Anforderungsartefakten identifizieren: In Mahmud et al. (2016) werden strukturierte Anforderungsspezifikationen in SAT-Formeln transformiert, um durch die Identifizierung von Antonymen logische Inkonsistenzen zu finden. Im Rahmen des Product Line Engineering überprüfen Mendonça et al. (2009) Featuremodelle durch Transformation in Aussagenlogik. Pittke et al. (2014) definieren linguistische Konsistenzbedingungen für strategische Anforderungsartefakte (sogenannte *Goal-Modelle*). Diese Konsistenzbedingungen beinhalten sowohl syntaktische als auch semantische Bedingungen, die beispielsweise Konsistenz unter der Berücksichtigung von Homonymen und Synonymen sicherstellen.

4.4 Zusammenfassung

In diesem Kapitel wurde eine Methode zur Konsistenzprüfung von AUTOSAR Timing Anforderungen vorgestellt. Diese Methode ermöglicht es bereits vor der eigentlichen Timing Verifikation die Qualität der Timing An-

forderungen zu überprüfen und somit zu verbessern. Besonders die Berechnung der maximal großen Menge an erfüllbaren Anforderungen (MaxSMT) vereinfacht die Identifikation fehlerhafter Modellelemente. Weiterhin wurde eine Methode zur graphbasierten Visualisierung von MaxSMT und Unsat Core vorgestellt, die das Finden von Fehlern weiter vereinfacht.

Wenn eine Anforderungsmenge keine Inkonsistenzen mehr beinhaltet, so kann sie danach auf Korrektheit überprüft werden. Diese Timing Verifikation für AUTOSAR Timing Anforderungen wird im nächsten Kapitel vorgestellt.

5

Timing Verifikation von AUTOSAR Softwarearchitekturen

Wurde eine Anforderungsmenge als konsistent identifiziert, können potenziell alle darin enthaltenen Timing Anforderungen für eine Softwarearchitektur erfüllt sein. Dies muss nun in einem nächsten Schritt in einer Timing Analyse verifiziert werden. Da mit diesem Ansatz auch sicherheitskritische Echtzeitsysteme geprüft werden sollen, ist es wichtig auch worst-case Latenzen zwischen Timing Events zu finden, sodass hierfür ein analytischer Ansatz gewählt werden muss.

Dieses Kapitel stellt die Methode zur Timing Verifikation von AUTOSAR Softwarearchitekturen vor. Dazu werden sowohl die AUTOSAR Softwarearchitektur als auch die im Modell enthaltenen AUTOSAR Timing Constraints nach Timed Automata transformiert. Dadurch, dass der Ansatz ausschließlich das AUTOSAR Architekturmodell betrachtet und die einzelnen Reglerfunktionen nicht berücksichtigt, kann die Timing Analyse bereits in frühen Entwicklungsphasen erfolgen. Sobald Reglermodelle und

deren Laufzeiten auf der zu verwendenden Hardwareplattform zur Verfügung stehen, können diese in das Analysemodell eingefügt werden. Somit kann die Analyse iterativ erweitert werden und die Analyseergebnisse werden exakter.

Zunächst wird die Transformation des formalen Architekturmodells beschrieben, das zuvor in Abschnitt 3.2.1 eingeführt wurde. Danach wird die Transformation der Timing Requirements aus Abschnitt 3.2.1 vorgestellt und im Anschluss in Abschnitt 5.3 mit existierenden Verfahren verglichen. Der Inhalt basiert zum Großteil aus den bereits veröffentlichten Arbeiten in Beringer und Wehrheim (2016).

5.1 Transformation des AUTOSAR Architekturmodells nach Timed Automata

Während AUTOSAR eine formale Syntax als OMG Metamodell vorgibt, ist die Semantik der Modelle nur in Form von natürlichsprachlicher Spezifikationen beschrieben. Um Timing Anforderungen formal verifizieren zu können, ist es jedoch notwendig eine formale Semantik für relevante Metamodellelemente abzuleiten. Wir verwenden für unsere Methode Timed Automata, da diese das zeitliche Verhalten der Modellelemente formal beschreiben können.

Für die Verifikation von Timing Anforderungen in AUTOSAR wurde eine Abbildung von AUTOSAR-Modellen auf Timed Automata modelliert, wobei das AUTOSAR-Modell sowohl die Softwarearchitektur also auch die Timing Anforderungen enthält. Die AUTOSAR Architektur wird in ein Netzwerk von Timed Automata transformiert, wobei jede Timing Anforderung in einen Testautomaten und eine TCTL-Query transformiert wird (siehe Abbildung 5.1). Im resultierenden Gesamtnetzwerk kommunizieren die Testautomaten mit den Architekturautomaten über Broadcast-Kanäle. So können die Testautomaten die Architekturautomaten nicht blockieren und das Verhalten der Softwarearchitektur kann nicht durch das Erstellen von Testautomaten beeinflusst werden.

Für ein gegebenes AUTOSAR-Modell $AR = (R, AC, VA, T, TRM)$ wird ein

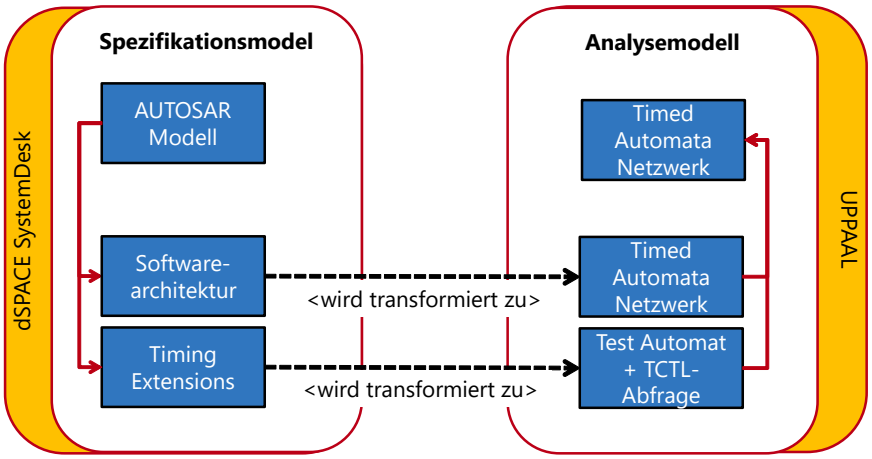

Abbildung 5.1: Transformation des AUTOSAR Modells in ein Netzwerk aus Zeitautomaten und TCTL-Abfragen

Netzwerk aus Timed Automata $\mathcal{N} = (\mathcal{A}_1 \mid\mid .. \mid\mid \mathcal{A}_n)$ erstellt. Im Folgenden werden die Transformationen der einzelnen Modellelemente genauer vorgestellt:

RunnableEntities RunnableEntities (Runnables) repräsentieren Codefragmente, die in die Softwarearchitektur integriert werden. Das Triggern wird durch die RTE-Ebene gesteuert. Des Weiteren haben Runnables Zugriff auf einen vordefinierten Satz von Variablen. Variablen mit Lesezugriff werden direkt beim Start des Runnables gelesen, während Schreibzugriffe vor der Terminierung ausgeführt werden*. Die Ausführung des Runnable Codes benötigt Zeit. Diese wird durch die Best-Case Execution Time und Worst-Case Execution Time angegeben.

Für jedes RunnableEntity im Spezifikationsmodell wird ein Timed Automaton generiert, der sowohl die Datenzugriffe als auch das Laufzeitverhalten berücksichtigt. Die Datenzugriffe werden dabei entweder über sogenannte Inter-Runnable-Variablen bei der Kommunikation innerhalb einer Softwarekomponente dargestellt oder über Ports bei der Kommunikation zwischen Softwarekomponenten. Für jede Kommunikationsbeziehung

*In AUTOSAR wird dies auch als impliziter Lese-/Schreibzugriff beschrieben

werden Variablenzugriffe in Form von Locations und Transitionen erzeugt, die das Senden und Empfangen von Daten repräsentieren.

Für jedes RunnableEntity $re \in R$ einer Softwarekomponente mit $re = (VA_{read}, VA_{write}, wcet, bcet)$ wird ein Timed Automaton $\mathcal{A} = (L, B, B^*, X, I, U, E, I_{ini})$ generiert. Sei $VA_{read} = \{re\_VA_{read_1}, \ldots, re\_VA_{read_n}\}$ die Menge der Schreibzugriffe (VA_{write} analog). Im Folgenden nutzen wir eine beliebige Ordnung von 1 bis n dieser Mengen.

- Locations: $L = \{re\_ready\_loc, re\_running\_loc\} \cup \{re\_Va_{read}\_loc \mid Va_{read} \in VA_{read}\}$
  $\cup \{re\_Va_{write}\_loc \mid Va_{write} \in VA_{write}\}$,
- Handshake Kommunikation: $B = \{re\_start, r\_finished\}$,
- Broadcast Kommunikation: $B^* = \{re\_va_{read} \mid Va_{read} \in VA_{read}\}$
  $\cup \{re\_va_{write} \mid Va_{write} \in VA_{write}\}$,
- Uhren: $X = \{x\}$,
- Invarianten: $I(re\_running) = \{x <= wcet\}$,
- Urgency: $\forall Va \in VA_{read} \cup VA_{write} : U(r\_Va) = true$,
  $U(r\_ready) = false$, $U(re\_running) = false$,
- Kanten: $E = \{(re\_ready, re\_start?, \emptyset, \{x\}, r\_Va_{read_1})$,
  $(re\_Va_{read_n}, re\_va_{read_n}!, \emptyset, \emptyset, re\_running)\} \cup$
  $\{(re\_Va_{read_j}, re\_va_{read_j}!, \emptyset, \emptyset, re\_Va_{read_{j+1}}) | 1 \leq j \leq |VA_{read}| - 1\} \cup$
  $\{(re\_Va_{write_j}, re\_va_{write_j}!, \emptyset, \emptyset, re\_Va_{write_{j+1}}) | 1 \leq j \leq |VA_{write}| - 1\} \cup$
  $\{(re\_running, re\_va_{write_1}!, \{x \geq bcet\}, re\_Va_{write_1})\} \cup$
  $\{(re\_Va_{write_n}, re\_finished!, \emptyset, \emptyset, re\_ready)\}$ und
- Initiale Location: $I_{ini} = re\_ready\_loc$.

Der generierte Timed Automaton besteht mindestens aus den Locations *re_ready_loc* und *re_running_loc* (mit dem Namen des Runnables vorangestellt). Der Automat befindet sich in der Location *re_ready_loc*, wenn das Runnable gerade nicht ausgeführt wird und ansonsten in der Location *re_running_loc*. Initial befindet sich der Automat im Zustand *re_ready_loc*.

Jeder implizite Variablenzugriff des RunnableEntity wird ebenfalls als Location repräsentiert. Die Identifikation des Zugriffs erfolgt über Signale an den Transitionen. Diese Signale werden nicht nur für die Synchronisierung verwendet, sondern auch, falls vorhanden, für vorhandene Testautomaten, die aus den AUTOSAR Timing Anforderungen generiert werden (siehe Abschnitt 5.2) und die den Datenfluss in der Softwarearchitektur erkennen müssen. Daher werden die Channels als *Broadcast* Channels definiert. Die in Abschnitt 5.2 vorgestellten Testautomaten müssen dieses Verhalten berücksichtigen.

Abbildung 5.2 veranschaulicht ein transformiertes RunnableEntity mit einem eingehenden Variablenzugriff (Lesezugriff) und einem ausgehenden Zugriff (Schreibzugriff). Es zeigt das RunnableEntity *TssPreprocessing*, welches sich innerhalb der Softwarekomponente *IndicatorLogic* befindet (siehe auch Abbildung 2.8) und die Rohdaten des Blinkersensorwertes *tss_value* einliest, vorverarbeitet und die Resultate nach *tss_status* schreibt. Weiterhin werden *wcet* und *bcet* als *5ms* bzw. *2ms* angenommen. Der Automat simuliert das Verhalten eines RunnableEntities in der AUTOSAR Architektur, indem es durch das Signal *Tss_Preprocessing_start* gestartet werden kann, und das Laufzeitverhalten durch die Zeitbeschränkungen mittels einer Invariante im Zustand TssPreprocessing_running_loc und einem Guard an der darauf folgenden Transition.

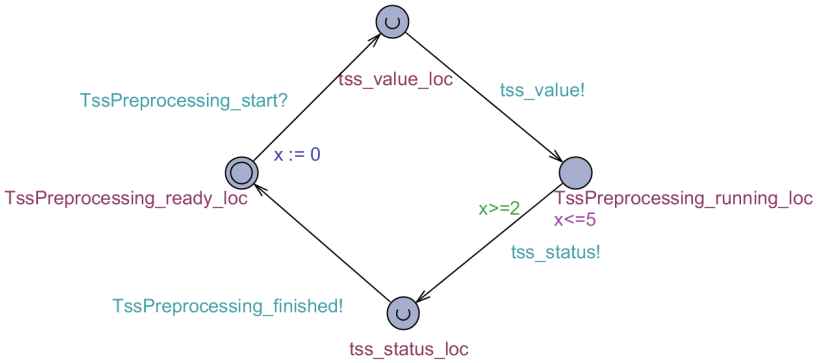

Abbildung 5.2: Beispiel für die Repräsentation des RunnableEntities *Tss_Preprocessing* als Timed Automaton

AssemblyConnections AssemblyConnections $C = (left, right)$ verbinden in AUTOSAR Ports verschiedener Softwarekomponenten. In unserem Modell abstrahieren wir jedoch von Softwarekomponenten und Ports, da dies die Timing Analyse nicht beeinflusst. Die Connections stellen daher Schreib- und Lesezugriffe von Variablenzugriffselementen dar. Für jede AssemblyConnection wird ein Timed Automaton generiert, der den Datenfluss zwischen den Runnables über die Ports beschreibt. Der Automat enthält eine Location, sowie für die Variablenzugriffe *left* und *right* jeweils eine Transition, welche die Verbindung der Runnables in der Softwarearchitektur darstellen. Somit erhalten wir für jede AssemblyConnection $C = (left, right)$ einen Timed Automaton $\mathcal{A} = (L, B, B^*, X, I, U, E, I_{ini})$ mit:

- Locations: $L = \{ac\_start\}$,
- Signale: $B = \{left, right\}, B^* = \{\}$,
- Uhren: $X = \emptyset$,
- Invarianten $I : \emptyset$,
- Urgency: $U(ac\_start) = false$,
- Kanten: $E = \{(ac\_start, left?, \emptyset, \emptyset, ac\_start), (ac\_start, right?, \emptyset, \emptyset, ac\_start)\}$ und
- Initialer Location: $I_{ini} = ac\_start$.

Task Runnable Mapping Für die korrekte Ausführungsreihenfolge der Runnables im Analysemodell wird für jeden OSTask ein Timed Automaton $\mathcal{A}$ generiert. Dieser Automat triggert die enthaltenen Runnables des OSTasks in der spezifizierten Reihenfolge. Der Automat sendet das *Start*-Signal an die einzelnen Runnables. Danach wird das Runnable in die *running*-Location gesetzt und es verlässt die *running*-Location, wenn der Runnable-Automat das *finish*-Signal zurück an den Runnable-Mapping-Automaten zurücksendet. Da zwischen Start und Stopp keine Zeit vergeht, werden die entsprechenden Locations als *urgent* markiert.

Sei T die Menge aller OSTasks und für jede OSTask $t \in T$, sei $R_t = \{r \in R \mid TRM(r) = t\}$ die Menge aller *RunnableEntities*, die vom OSTask t getriggert werden. Wiederum nehmen wir eine beliebige Reihenfolge der Menge R_t an, wobei wir die Indizes 1 to n verwenden. Dann existiert für jeden OSTask $t \in T$, ein Timed Automaton $\mathcal{A}$ im Analysemodell mit:

- Locations: $L = \{t\_ready, t\_running\} \cup \{t\_r\_start, t\_r\_stopped \mid r \in R_t\}$
- Signale: $B = \{t\_run, t\_processed\} \cup \{t\_r\_start, t\_r\_finished \mid r \in R_t\}, B^* = \{\}$,
- Uhren: $X = \{x\}$
- Invarianten: $I(t\_running) = \{x == 0\}$,
- Urgency: $\forall r \in R_t : U(t\_r\_finished) = true, U(t\_running) = true$,
- Kanten: $E = \{(t\_ready, t\_run?, \emptyset, \emptyset, t\_running)$,
  $(t\_running, t\_r_1\_start!, \emptyset, \emptyset, t\_r_1\_running)$,
  $(t\_r_n\_stopped, t\_processed!, \emptyset, \emptyset, t\_processed)$,
  $\cup \{(t\_r\_stopped, t\_r\_start!, \emptyset, \emptyset, t\_r\_running)$,
  $(t\_r\_running, t\_r\_finished?, \emptyset, \emptyset, t\_r\_stopped) \mid r \in R_t\}$ und
- Initialer Location: $I_{ini} = t\_ready$.

Abbildung 5.3 zeigt beispielhaft ein Task Runnable Mapping als Timed Automaton.

Tasks Jedes AUTOSAR-basierte Steuergerät enthält ein AUTOSAR-konformes OSEK Betriebssystem, das die Ausführung von *OsTasks* auf dem Steuergerät koordiniert. OsTasks haben die Zustände *suspended*, *ready* und *running*. Zusätzlich sind sie durch eine Zykluszeit p definiert, mit der sie periodisch aufgerufen werden. Ein Task ist im Zustand *ready*, wenn er durch den Betriebssystemscheduler zur Ausführung ausgewählt werden kann. Wenn der Scheduler den Task zur Ausführung auswählt, wird der Task in den Zustand *running* gesetzt. Nach der Terminierung, aber vor Ablauf der Zykluszeit, wird der Task in den Zustand *suspended* gesetzt.

Für jeden OsTask $t \in T$ wird ein Timed Automaton wie folgt $\mathcal{A}$ generiert:

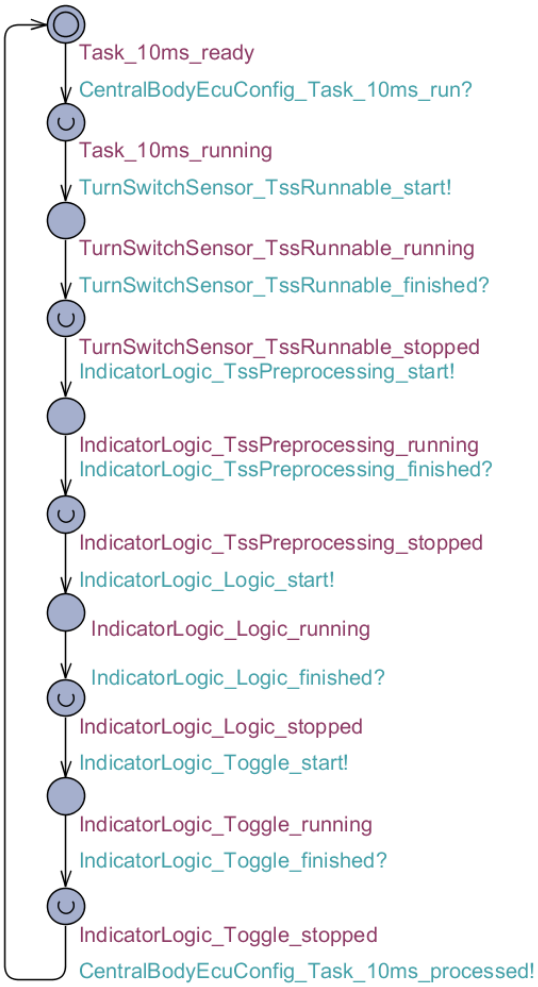

Abbildung 5.3: Beispiel für das Task Runnable Mapping als Timed Automaton

- Locations: $L = \{ready, starting, running, terminating, suspended\}$,
- Handshake Kommunikation: $B = \{t\_startTask, t\_run, t\_processed, t\_terminateTask, t\_isNotReady\}$,
- Broadcast Kommunikation: $B^* = \{\}$, $X = \{x\}$,
- Invarianten: $I(running) = \{x <= p\}$, $I(suspended) = \{x <= p\}$,
- Urgency: $U(ready) = false$, $U(starting) = true$, $U(running) = false$,

$U(terminating) = true, U(suspended) = false,$

- Kanten: $E = \{(ready, t\_startTask?, \emptyset, \emptyset, starting),$
  $(starting, t\_run!, \emptyset, \emptyset, running),$
  $(running, t\_processed?, \emptyset, \emptyset, terminating),$
  $(terminating, t\_terminateTask!, \emptyset, \emptyset, suspended),$
  $(suspended, \varphi, \{x == p\}, \{x\}, ready),$
  $(suspended, t\_isNotReady!, \emptyset, suspended)\}$ und
- Initialer Location: $I_{ini} = ready$.

Das Verhalten eines OsTasks wird modelliert durch die Generierung von Locations für *ready*, *running* und *suspended* und zusätzlich (urgent)-Locations für das Senden und Empfangen mehrerer Signale zur Synchronisation mit dem RunnableToTask-Mapping-Automaten. Der OsTask startet in der Ready-Location und kann dann durch den Task Scheduler getriggert werden. Durch den Empfang des Signals *startTask* bekommt das *EventToTaskMapping* das Signal den OsTask auszuführen und setzt diesen in den Zustand *running*. Danach wird das EventToTaskMapping ausgeführt, d.h. es werden alle RunnableEntities ausgeführt und danach das Signal *processed* empfangen und das Signal *terminateTask* wird an den Scheduler gesendet. Der OsTask verbleibt dann solange im Zustand *suspended* bis die Periode des OsTasks abgelaufen ist. In der Zwischenzeit synchronisiert sich der Automat nur über das Signal *isNotReady* mit dem Scheduler. Danach wird der OsTask zurück in den Zustand *ready* gesetzt und kann erneut durch den Scheduler ausgewählt werden. Abbildung 5.4 zeigt beispielhaft die Repräsentation eines AUTOSAR OSTasks als Timed Automaton.

5.2 Transformation der Timing Anforderungen nach Timed Automata

Neben der Transformation des Architekturmodells müssen ebenfalls die Timing Anforderungen von AUTOSAR nach Timed Automata transformiert werden. Für die Transformation der einzelnen Timing Anforderungen wird

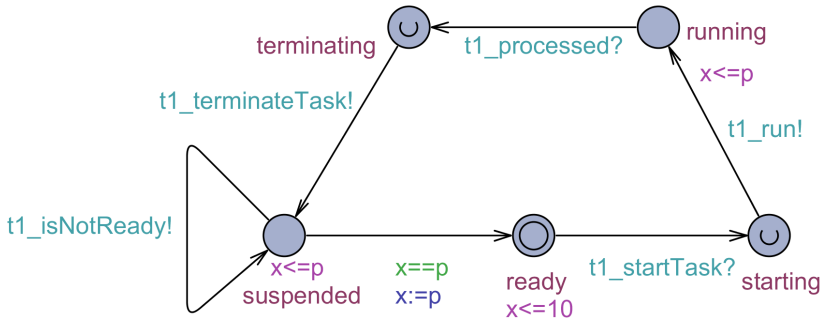

Abbildung 5.4: Beispiel für die Repräsentation eines OSTask als Timed Automaton

jeweils ein Testautomat generiert, sowie eine TCTL-Abfrage, die die Timing Anforderung auf dem Testautomaten prüft.

Latency Timing Constraint Bei der Transformation des Latency Timing Constraints wird die Event Chain $C = \langle e_1, \ldots, e_n \rangle$ in einen Testautomaten transformiert, der die Event Chain als Kette von Locations modelliert. Zwischen jeder Location wird eine Transition generiert, die das entsprechende Signal des in der Event Chain modellierten Events empfängt. Die Verifikation der definierten Latenz wird durch eine Uhr erreicht, die die Zeit während des Durchlaufens der Event Chain misst und am Ende der Event Chain zurückgesetzt wird. Die maximale Latenz wird dann durch eine TCTL-Abfrage überprüft, die den maximalen Uhrenwert des Testautomaten überprüft. Daher wird für jeden Latency Timing Constraint $le \in \mathcal{R}_{ltc}$ ein Timed Automaton $\mathcal{A}$ wie folgt generiert:

- Locations: $L = \{lc\_e | e \in C\}$,
- Signale: $B^* = \{e | e \in C\}$, Uhren: $X = \{x\}$,
- Invarianten $I(lc\_e_1) = \{x \leq 1\}$,
- Kanten: $E = \{(lc\_e_j, e_j?, \emptyset, \emptyset, lc\_e_{j+1}) | 1 \leq j < n - 1\}$
  $\cup \{lc\_e_n, e_n, \emptyset, \{x\}, lc\_e_1 \cup \{(lc\_e_1, e_1?, \emptyset, \{x\}, lc\_e_1)\}$,
- Initiale Location: $I_{ini} = lc\_e_1$.

In der ersten Location lc_e_1 (also *bevor* das erste Event empfangen wurde) setzt der Automat in regelmäßigen Schritten die Uhr zurück (implementiert durch die Selbsttransition und Invariante auf lc_1), sodass der Wert der Uhr nur dann 1 überschreitet, wenn das erste Ereignis empfangen wird. Es ist zu beachten, dass nach der Definition von B^* die generierten Signale eine Broadcast-Kommunikation verwenden. Zusätzlich wird die TCTL-Abfrage $\phi = AG(x < maximum)$ generiert. AG bedeutet, dass die Eigenschaft auf allen Pfaden immer gültig sein muss.

Abbildung 5.5 zeigt einen Latency Timing Constraint Automaton, der die Zeit vom Start-Event, wenn der Blinkersensor die Rohdaten erhält, bis zum Ende-Event, wenn der Aktuator, der die Blinkerleuchten ansteuert, das Signal erhält, misst.

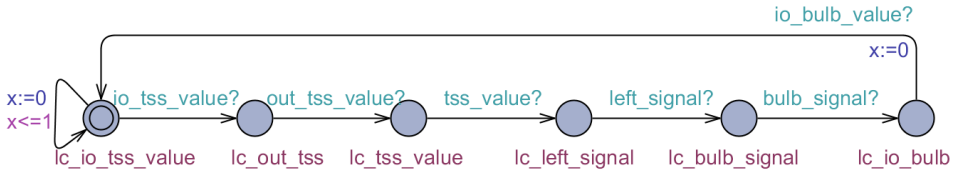

Abbildung 5.5: Timed automaton eines Latency Timing Constraints

Execution Order Constraint Anforderungen an die geordnete Ausführung von RunnableEntities werden durch den Execution Order Constraint erfasst. Ein Execution Order Constraint $r_{eoc} = \langle re_i, \ldots, re_j \rangle, re_i \in Re$ wird durch eine geordnete Abfolge einer Teilmenge der verfügbaren RunnableEntities definiert, für die die Ausführungsreihenfolge angegeben ist.

Für jeden ExecutionOrderConstraint r_{eoc} wird ein Timed Automaton wie folgt generiert:

- Locations: $L = \{re_i\_EOC\_started, re_i\_EOC\_finished \mid 1 \leq i \leq n\}$ $\cup \{init, error\}$,
- Broadcast Kommunikation: $B^* = \{re_i\_EOC\_start, re_i\_EOC\_finished \mid i = 1, \ldots, n\}$,
- Handshake Kommunikation: $B = \{\}$,

- Uhren: $X = \{\}$,
- Invarianten I ist *true* für alle Locations,
- Urgency: $U(re_n\_EOC\_finished) = true$,
- $E = \{init, re_1\_EOC\_start?, \emptyset, \emptyset, re_1\_EOC\_started\} \cup$
  $\{re_i\_EOC\_started, re_i\_EOC\_finished?, \emptyset, \emptyset, re_i\_EOC\_finished \mid i = 1, \ldots, n\} \cup$
  $\{re_i\_EOC\_finished, re_{i+1}\_EOC\_start?, \emptyset, \emptyset, re_{i+1}\_EOC\_started \mid i = 1, \ldots, n\} \cup$
  $\{re_n\_EOC\_finished, \tau, \emptyset, \emptyset, init\}$
- $I_{ini} = init$.

Man beachte, dass gemäß der Definition von B^* die generierten Signale Broadcast Kommunikation verwenden. Darüber hinaus wird für jede Location $l \in L$ eine TCTL-Abfrage $\phi_l = AF(l)$ und eine weitere TCTL-Abfrage $\phi_e = AG\ not\ error$ generiert. Diese Eigenschaft setzt voraus, dass auf allen Pfaden des Systemverhaltens jede Location irgendwann besucht wird (d.h. die Events werden in der angegebenen Reihenfolge empfangen). Eine falsche Reihenfolge würde zudem den Automaten in den *error*-Zustand setzen, was mit ϕ_e geprüft wird. Abbildung 5.6 zeigt beispielhaft den Automaten r_{eoc} aus Tabelle 4.1, der die Reihenfolge der Runnables *TssPreprocessing*, *Logic* und *Toggle* beschränkt.

Synchronization Timing Constraint Der Synchronization Timing Constraint wird verwendet, um Anforderungen an die Synchronizität von Timing Events zu stellen. Er wird definiert durch eine Menge an Events e_i und einem Toleranzwert, der die maximale zeitliche Abweichung zwischen dem ersten Auftreten und dem letzten Auftreten eines Events der Menge angibt. Ein Synchronization Timing Constraint ist erfüllt, wenn $\forall e_i, e_j \in S : |t_{e_i} - t_{e_j}| \leq tolerance$, wobei t_i der Zeitpunkt ist, zudem e auftritt.

Daher wird für jeden Synchronization Timing Constraint r_{stc} ein Timed Automaton wie folgt generiert:

- Locations: $L = \{sc\_init\}$,

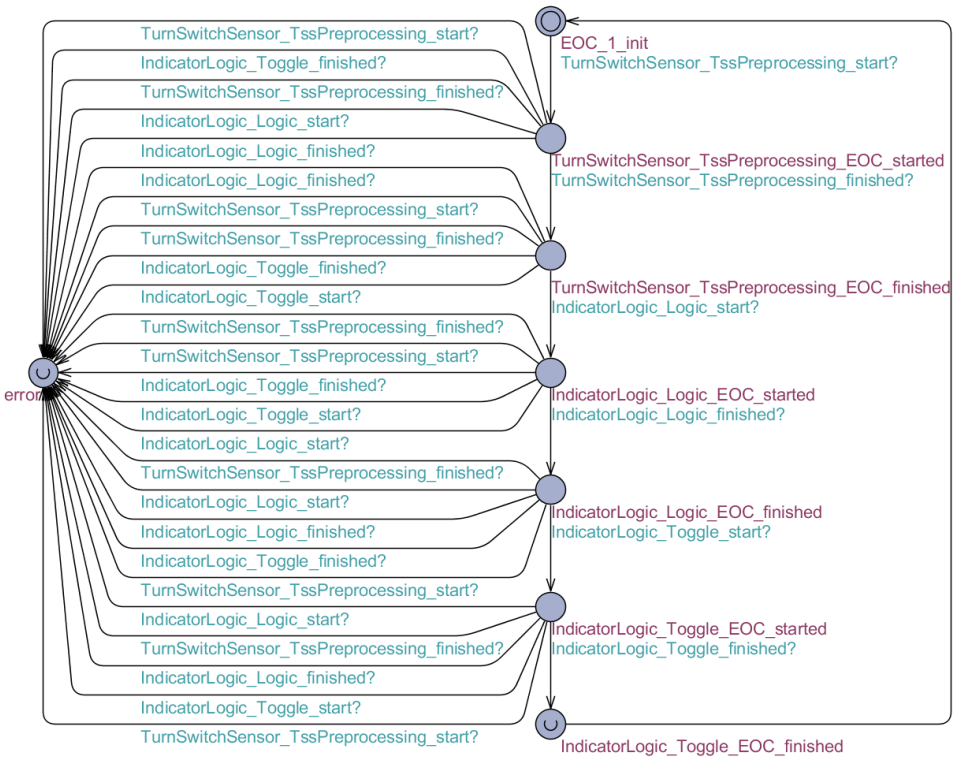

Abbildung 5.6: Timed Automaton für eoc_1

- Signale: $B^* = \{e | e \in S\} B = \{\}$,
- Uhren: $X = \{x\}$,
- Invarianten: $I = \{\emptyset\}$,
- Urgency: U ist *false* für alle Locations,
- Kanten: $E = \{sc\_init, e?, \emptyset, \emptyset, sc\_init\}$, $I_{ini} = \{sc\_init\}$.

Auch hier verwenden die Signale wieder Broadcast-Kommunikation. Weiterhin werden für die generierten Transitionen Funktionen spezifiziert, die jedes Mal ausgeführt werden, wenn die Transition schaltet. Für jede Transition $e_i \in E$ wird die entsprechende Funktion *e_i_receiving* ausgeführt. Zusätzlich werden für jeden Automaten lokale Deklarationen definiert, wie sie in Listing 5.1 beschrieben sind. Die lokalen Deklarationen von UPPAAL

werden verwendet, um zu speichern, welche Signale bereits vom Automaten empfangen wurden. Die für jedes Event generierte Funktion speichert in einer Variable *e_i_received*, ob das Event empfangen wurde. Die Funktionen *isRunning* und *isCompleted* werden in jeder Funktion *e_i_receiving* aufgerufen, um das erste Auftreten eines Events und das Auftreten des letzten Events festzuhalten. Sowohl beim ersten Auftreten eines Events als auch beim Auftritt des letzten Events wird die Uhr des Automaten zurückgesetzt. Wird das erste Event empfangen, wird zudem der Automat in den Zustand *running* gesetzt, indem die boolsche Variable *running* gesetzt wird. Nach dem Auftreten des letzten Events wird die Variable dann wieder zurückgesetzt. Sobald also die *running*-Variable gesetzt wurde und die Uhr zurückgesetzt wurde, läuft die Zeit solange weiter bis alle Events empfangen wurden und wird erst dann wieder zurückgesetzt. Wird nun, solange der Automat im Zustand *running* ist, die Uhrzeit durch den maximalen Toleranzwert begrenzt, so spiegelt dies die Semantik des Synchronization Timing Constraints wieder.

Listing 5.1: Lokale Deklarationen in UPPAAL

```
clock x;
bool e_i_received = false;
bool running = false;

void e_i_receiving() {
        isRunning();
        e_i_received = true;
        isCompleted();
}

void isRunning(){
        if(!running){
                x=0;
                running=true;
        }
}

void isCompleted(){
        if (e_1_received && .. e_n_received){
                e_i_received = false;
                x=0;
                running=false;
        }
}
```

Daher wird für den Synchronization Timing Constraint eine TCTL-Abfrage wie folgt generiert:$AG(running \implies x \leq tolerance)$. Abbildung 5.7 zeigt beispielhaft die Transformation eines Synchronization Timing Constraints, das die synchrone Ausführung der Runnables des linken und rechten Blinkeraktuators erfordert. Analog zum Automaten werden die erforderlichen lokalen Deklarationen generiert, d.h. zwei Flags *Bulb1_received* und *Bulb2_received*, sowie die Funktionen *Bulb1_receiving* und *Bulb2_receiving*.

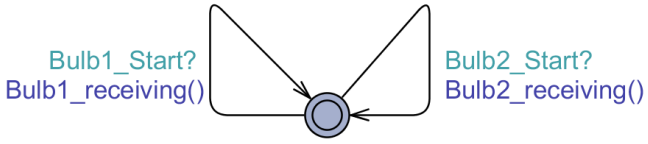

Abbildung 5.7: Beispiel eines Synchronization Timing Constraints für die Synchronizität der Blinkerlampen

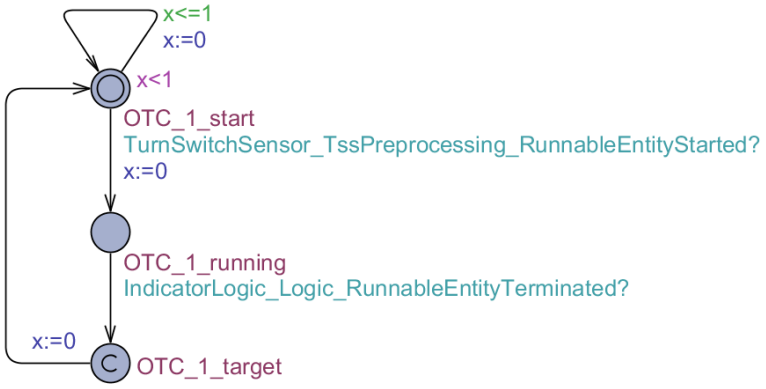

Abbildung 5.8: Timed Automaton für r_{otc}

Offset Timing Constraint Der Offset Timing Constraint wird für die Transformation als ein Spezialfall des Latency Timing Constraints behandelt, bei dem lediglich zwei Timing Events beteiligt sind, ein Source-Event und ein Target-Event. Dementsprechend wird für jeden Offset Timing Constraint $r_{otc} = (e_s, e_t, min, max)$ ein Timed Automaton analog zu einem Latency Timing Constraint mit = $r_{ltc} = (\langle e_s, e_t \rangle, min, max)$ generiert mit den TCTL-Queries $\phi^1 = AG(x \leq max)$ und $\phi^2 = AG(not\ OTC\_start\ implies\ x \geq min)$. Ein Beispiel für einen Offset Timing Constraint ist r_{otc} aus dem Blinkerbeispiel in Abschnitt 2.1.2. Dieser wird in Abbildung 5.8 gezeigt. Die generierten TCTL-Queries sind $\phi_{r_{otc^1}} = AG(x \leq 4)$ und $\phi_{r_{otc^2}} = AG(not\ OTC\_1\_start\ implies\ x \geq 3)$.

Execution Time Constraint Der Execution Time Constraint gibt die minimale und maximale Laufzeit für ein RunnableEntity an. Für diese Analyse auf Taskebene existieren bereits Methoden wie in Abschnitt beschrieben. Im

Folgenden wird daher für die Verifikation der Execution Time Constraint nicht weiter betrachtet und auf die existierenden Methoden zur Analyse auf Taskebene hingewiesen.

Für die Verifikation der AUTOSAR-Architektur werden alle generierten Automaten $\mathcal{A}_i = (L_i, B_i, X_i, I_i, E_i, I_{i_{ini}})$ zu einem Netzwerk von Timed Automata $\mathcal{N} = (\mathcal{A}_1 \mid\mid .. \mid\mid \mathcal{A}_n)$ verbunden. Dann ist ein *TimingConstraint* T mit TCTL-Formel ϕ_T für das Modell gültig, genau dann wenn $(\mathcal{N} \mid\mid T) \models \phi_T$, d.h. der Automat für ein einzelnes Timing Constraint wird mit dem Netzwerk der Timed Automata verbunden, das die Softwarearchitektur repräsentiert. Es wird dabei immer nur ein Testautomat angebunden, um Seiteneffekte, die durch das Hinzufügen weiterer Testautomaten entstehen können, zu vermeiden. Dieses Netzwerk wird dann anhand der für den jeweiligen Testautomaten spezifizierten TCTL-Formel überprüft. Wird kein expliziter Testautomat für einen Timing Constraint erzeugt, wie beim Execution Time Constraint, so wird das erzeugte Netzwerk $\mathcal{N}$ auf Deadlockfreiheit überprüft.

Die Transformation der AUTOSAR Softwarearchitektur und der Timing Requirements nach Timed Automata wurde beispielhaft für das Blinkerbeispiel aus Abschnitt 2.1.2 durchgeführt. Aus Kapitel 4 wissen wir bereits, dass die Anforderungsmenge inkonsistent ist. Somit ist zu erwarten, dass ohne Änderung der Anforderungen mindestens eine Anforderung nicht erfüllbar ist.

Der generierte Execution Order Constraint für r_{eoc_1} wurde bereits in Abbildung 5.6 vorgestellt. Die generierte TCTL-Query ist $\phi_{r_{eoc_1}} = \mathcal{A}G\ not\ error$. Der Offset Timing Constraint r_{otc} ist in Abbildung 5.8. Die generierten TCTL-Queries sind $\phi_{r_{otc^1}} = \mathcal{A}G(x \leq 4)$ und $\phi_{r_{otc^2}} = \mathcal{A}G(not\ OTC\_1\_start\ implies\ x \geq 3)$.

Für das Blinkerbeispiel verwenden wir beispielhaft Worst-Case und Best-Case Execution Times aus Tabelle 5.1 für die Runnables. Aus diesen Werten lässt sich erkennen, dass die Execution Time Constraints r_{etc_1} und r_{etc_2} erfüllt werden. Der Execution Order Constraint wird durch die korrekte Ausführungsreihenfolge der Runnables ebenfalls abgebildet. Der Offset Timing Constraint ist jedoch nicht erfüllt, da zwischen dem Auftreten des Start-Events und des Target-Events mehr als 4ms vergehen.

Die generierten Timed Automata Netzwerke werden in UPPAAL verifi-

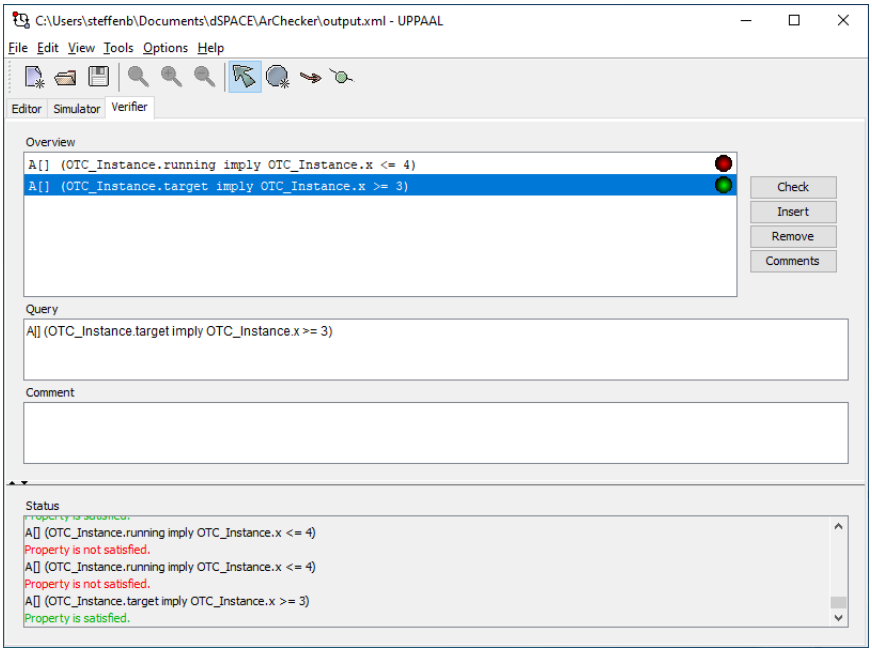

Abbildung 5.9: Der von der Softwarearchitektur nicht erfüllte Offset Timing Constraint r_{otc} in der UPPAAL GUI

ziert. Werden Timing Constraints nicht von der Softwarearchitektur erfüllt, können diese in der grafischen Oberfläche von UPPAAL genauer betrachtet werden. Abbildung 5.9 zeig beispielhaft das Werkzeug mit den generierten TCTL-Anfragen für $\phi_{r_{otc^1}}$. Es zeigt in diesem Fall, dass die erste TCTL-Anfrage des Offset Timing Constraints $\phi_{r_{otc^1}}$ nicht erfüllt ist. UPPAAL bietet darüber hinaus verschiedene Möglichkeiten an fehlerhafte Pfade zu generieren, anzuzeigen und zu analysieren.

5.3 Stand der Technik

Der hier vorgestellte Ansatz verfolgt die Timing Verifikation von AUTOSAR Timing Constraints durch die Transformation nach Timed Automata. Darüber hinaus gibt es weitere Ansätze, die Timing Verifikation auf Systemebene mit anderen Methoden und Frameworks vorschlagen, sowie Ansätze, die ebenfalls auf Timed Automata als Analysemethode basieren, jedoch

Tabelle 5.1: Execution Times der RunnableEntities

Runnable	BCET	WCET
TssRunnable	1	1
WlsRunnable	1	1
TssPreprocessing	1	3
WlsPreprocessing	1	3
Logic	10	10
Toggle	1	5
BulbRunnable	1	1

nicht das Konzept der AUTOSAR Timing Constraints im Fokus haben. Die relevantesten Arbeiten werden im folgenden vorgestellt.

Timing Verifikation mit Timed Automata Die Anwendung von Timed Automata zur Verifikation verteilter eingebetteter Systemarchitekturen wird in der Literatur häufig vorgeschlagen. So werden beispielsweise in Hendriks und Verhoef (2006) UML-Sequenzdiagramme mit zusätzlichen Zeitannotationen nach Timed Automata transformiert, um Worst-Case Response Times zu berechnen. Neuere Arbeiten verwenden Timed Automata zur Verifikation von ROS[†]-basierten Roboteranwendungen (Halder et al., 2017), zur Verifikation von BPEL-Anforderungen (Gao et al., 2021), SystemC Designmodellen (Herber et al., 2015) oder zur Verifikation des RabbitMQ Protokolls (Li et al., 2022).

Ein ähnlicher Ansatz, der in Neumann et al. (2012) beschrieben wird, verwendet ebenfalls Timed Automata für die Analyse von AUTOSAR-Architekturen. Hier werden für die Analyse ebenfalls Automaten für das AUTOSAR-Systemmodell, insbesondere auch für Runnables, Tasks und das Task-Runnable-Mapping mithilfe eines Template-basierten Mechanismus erstellt. Die generierten Automaten unterscheiden sich von unserem Ansatz aufgrund der unterschiedlichen Herangehensweise, da wir bei der Definition der Transformationen auf das zuvor spezifizierte formale Modell zurückgreifen und wir zur Interpretation der Semantik von AUTO-

[†]Robot Operating System

SAR ebenfalls auf den zur Verfügung stehenden Quellcode zurückgreifen konnten. Zudem lag der Fokus unserer Arbeit auf der Verifikation der Timing Anforderungen, während Neumann modellinhärente Timing Fehler erkennt. Im Gegensatz zu unserem Ansatz unterscheiden sich die generierten Automaten für RunnableEntities insofern, dass Variablenzugriffe immer durchgeführt werden können, wenn ein RunnableEntity im Zustand *running* ist und somit ähnlich zu expliziter Kommunikation in AUTOSAR sind, während wir bei uns das Lesen und Schreiben der Variablen bei impliziter Kommunikation vor- bzw. nachgelagert ist und somit ebenfalls die Semantik der impliziten Datenkommunikation, die durch die RTE orchestriert wird, unterstützt, wodurch unser Ansatz etwas exakter die Zeitpunkte der Ausführung dieser Kommunikation beschreibt. Bei der Generierung von OSTasks werden in unserem Ansatz keine Committed Locations erzeugt, sodass in unserem Fall die Automaten nicht in einen Fehlerzustand laufen können, was wiederum näher an der Semantik der AUTOSAR-Spezifikation ist. Zudem verzichten wir bei der Generierung des Task-Runnable Mappings im Gegensatz zum dort gezeigten Automatentemplate auf die Implementierung verschiedener Trigger-Zeiten, um die Komplexität der Automaten klein zu halten. Darüber hinaus verzichten wir vollständig auf die Generierung von Softwarekomponenten und Ports, da diese keinen Einfluss auf die Laufzeit des Systemmodells haben. Hier verzichten wir auf existierende Sturkturelemente des AUTOSAR-Standards, sodass unser Ansatz näher an der Semantik des generierten Quellcodes ist. Das so erstellte Automatennetzwerk wird dann in UPPAAL auf Deadlockfreiheit geprüft. Im Gegensatz zu dem hier vorgestellten Ansatz ermöglichen die Transformationen die Erkennung allgemeiner Timing Fehler, die sich aufgrund einer fehlerhaften Konfiguration des Systemmodells ergeben und mithilfe der Verifikation auf Deadlockfreiheit erkannt werden. Es werden aber keine Transformationen der AUTOSAR Timing Constraints durchgeführt, was für die Analyse von Timing Anforderungen jedoch notwendig ist. Des Weiteren wird für die Automaten keine Broadcast-Kommunikation verwendet, die für eine Verknüpfung mit den hier vorgestellten Test-Automaten der Timing Anforderungen benötigt wird. Weiterhin beinhalten die Automaten zusätzliche Fehlerzustände

mit denen die Deadlockfreiheit des Systemmodells geprüft wird, die wiederum in unserem Ansatz nicht benötigt werden. Schließlich wird kein formales Modell definiert, welches als Grundlage für die Generierung der Automaten aus den gezeigten Templates herangezogen wird.

In Neumann und Giese (2013) wird eine erweiterte Definition von Timed Automata vorgestellt, die eine kompositionelle Timing Verifikation ermöglicht, und auf ein AUTOSAR-Beispielmodell angewendet. Im Gegensatz zu unserer Arbeit werden die AUTOSAR-Transformationen hier auf Softwarekomponentenebene durchgeführt, wohingegen wir in unserem Ansatz von Softwarekomponenten weitgehend abstrahieren. Des Weiteren wurden die Automaten, welche die Implementierungen der Softwarekomponenten und das Zusammenspiel der Runnables darstellen, nicht vollständig automatisiert erzeugt. Stattdessen werden Automaten lediglich teilweise aus den bestehenden AUTOSAR Interfaces generiert.

Weitere Ansätze, die ebenfalls Timed Automata verwenden, schlagen die Konstruktion von Testautomaten (oder Szenario-Automaten) für die Spezifikation von Anforderungen vor Gehrke et al. (2006), berücksichtigen aber ebenfalls nicht die AUTOSAR Timing Extensions. In der in Scheickl und Ainhauser (2012) vorgestellten Arbeit wird eine Werkzeugunterstützung für die Verifikation von AUTOSAR Timing Anforderungen vorgestellt. Die Anforderungen werden verifiziert, indem sie mit spezifizierten Timing Garantien verglichen werden. Für diesen Ansatz müssen jedoch Timing Garantien spezifiziert werden, was in unserem Ansatz nicht notwendig ist.

Timing Verifikation für Echtzeitsysteme Es gibt verschiedene Methoden für die Analyse von Timing Anforderungen. Neben der Modellierung und Verifikation von zeitbehafteten Systemen mittels Timed Automata, existieren Methoden basierend auf der allgemeinen Scheduling-Analyse (Liu und Layland, 1973). In den Arbeiten von Richter (2005) und Feiertag et al. (2008) wird ein kompositioneller Scheduling Ansatz vorgestellt, der auf traditioneller Scheduling-Theorie für Echtzeitsysteme basiert und auch prototypisch für AUTOSAR implementiert wurde (Rhandor, 2012). Der Ansatz geht davon aus, dass Signale nur eingeschränkt an Komponenten ankommen können, beispielsweise mit einer fixen Frequenz und maximalen

Jitter. Diese Ankünfte werden in sogenannten Ereignisfunktionen spezifiziert. Wenn die ankommenden Signale nicht mit den vorgegebenen Modellen übereinstimmen, wird die Zeitanalyse unpräzise (Perathoner et al., 2009).

Real-Time Calculus (Thiele et al., 2000, Albers et al., 2008) ist ein Framework für die Performanceanalyse von Echtzeitsystemen, das auf dem Netzwerk-Kalkül basiert (Perathoner et al., 2009). Durch die Spezifikation eines Event Stream Modells kann ein Signalfluss durch ein System analysiert werden. Dies ist ein generischerer Rahmen als der von Richter (2005). Beide Methoden abstrahieren von den konkreten Metamodellelementen der AUTOSAR Softwarearchitektur. Darüber hinaus wenden wir unsere Methode direkt auf die AUTOSAR-Timing-Extensions an, während andere Methoden die Anwendung diesen Aspekt nur ansatzweise betrachten.

Eine weitere Methode zur Timing Verifikation von Echtzeitsystemen auf Basis von Architekturmodellen ist beispielsweise Petriu und Woodside (2004). Dort wird das UML-Profile for Schedulability, Performance and Time als Basis genommen und zur Timing Verifikation nach Timed Petri Nets transformiert.

Timing Verifikation im Kontext von MechatronicUML Neben AUTOSAR existieren andere Entwicklungsmethoden für vernetzte elektrische / elektronische Architekturen im Automobil wie beispielsweise AutoFOCUS (Aravantinos et al., 2015), Amalthea (Becker, 2021) oder MechatronicUML (Becker et al., 2014). Diese ermöglichen ebenfalls die Verifikation von Anforderungen auf Architekturebene. Alle Methoden haben jedoch unterschiedliche domänenspezifische Sprachen und betrachten verschiedene Aspekte des Systems auf unterschiedlichen Abstraktionsebenen.

MechatronicUML

MechatronicUML ist eine Erweiterung der UML und unterstützt den modellbasierten Entwurf mechatronischer Syteme bereits auf Systemebene, indem es eine formale Sprache sowohl zur Spezifikation von Anforderungen als auch zur Spezifikation einer plattformunabhängigen Softwarearchitektur zur Verfügung stellt (Dziwok et al., 2016). Anforderungen werden zunächst in Form von *Modal Sequence Diagrams* (MSDs) spezifiziert (Holtmann et al., 2016). Diese können dann auf ihre Konsistenz geprüft werden

Modal Sequence Diagrams

(Holtmann et al., 2016). Auf der Basis der MSD-Spezifikation wird dann eine plattform-unabhängige komponentenbasierte Softwarearchitektur erstellt (Dziwok et al., 2016). Komponenten der Softwarearchitektur können dann über Ports miteinander kommunizieren. Die Interaktion der Softwarekomponenten wird in MechatronicUML dann formal in *Real-Time Coordination Protocols* festgehalten. Das Verhalten der an den Interaktionen beteiligten Rollen, die den Ports zugewiesen sind, wird dann mittels erweiterter UML Zustandsdiagramme, sog. *Real-Time State Charts* (RTSC) beschrieben (Dziwok et al., 2016). Aus diesen Modellen lässt sich daraufhin ebenfalls das Verhalten der Softwarekomponenten ableiten, welches dann durch den Anwender verfeinert werden kann (Dziwok et al., 2016). Sowohl die Real-Time Coordination Protocols als auch das Verhalten der Softwarekomponenten können einzeln verifiziert werden, sodass auch große Systeme mit diesem kompositionallen Ansatz verifiziert werden können (Becker et al., 2014). Für die Verifikation definiert MechatronicUML eine Abbildung der Real-Time State Charts auf Timed Automata (Becker et al., 2014). Sowohl der hier vorgestellte Ansatz als auch MechatronicUML ermöglichen eine frühe Timing Verifikation von Anforderungen auf Modellebene, verwenden für die Timing Verifikation Zeitautomaten und nutzen UPPAAL als Werkzeug zur Verifikation. Während in MechatronicUML sowohl das Verhalten der Softwarekomponenten als auch der Interaktionen mittels RTSCs beschrieben werden, abstrahieren wir in unserem Ansatz vom konkreten Verhalten der Komponenten und betrachten lediglich die Worst-Case und Best-Case Laufzeiten von Runnables, sowie deren Datenaustausch, betrachten aber dafür in unserer Verhaltensbeschreibung Steuergeräte-spezifische Konfigurationen wie das Task-Runnable-Mapping, welches in MechatronicUML nicht vorhanden ist, da sich die Spezifikation ausschließlich auf plattform-unabhängige Eigenschaften bezieht. Hierdurch lässt sich bereits eine Timing Verifikation mit plattfom-spezifischen Modellelementen durchfürhen bevor Verhaltensmodelle für Komponenten vorhanden sind. Weiterhin können in MechatronicUML zur Beschreibung von Timing Anforderungen auf Architekturebene die existierenden Real-Time Coordination Protocols mit TCTL-Formeln annotiert werden (Eckardt et al., 2013). In unserem Ansatz hingegen verwenden wir die AUTOSAR Timing Extensions, die es

Real-Time State Charts

erlauben auf abstrakterer Ebene das Zeitverhalten des Systems zu beschreiben und transformieren diese dann in Zeitautomaten sowie TCTL-Formeln.

5.4 Zusammenfassung

In diesem Kapitel wurde eine Methode zur Timing Verifikation von AUTOSAR Timing Anforderungen vorgestellt. Mittels dieser Methode ist es möglich, Timing Anforderungen frühzeitig und ohne Zugriff auf Quellcode zu überprüfen. Es werden nur Timing-Annotationen in Form von WCETs und BCETs für Runnables benötigt. Diese müssen mithilfe von Expertenwissen zunächst konservativ abgeschätzt werden, oder es müssen obere Schranken für die Ausführungszeiten beispielsweise mittels statischer Codeanalyseverfahren berechnet werden. Für die Verifikation der AUTOSAR Architektur können dann existierende Werkzeuge für die Verifikation von Timed Automata wie beispielsweise UPPAAL oder Kronos genutzt werden. In den folgenden Kapiteln wird die praktische Anwendbarkeit der Methode evaluiert. Dafür wird zunächst die Methode an einem Fallbeispiel ausgeführt und sowohl die Effizienz und Effektivität evaluiert. Anschließend werden Laufzeitanalysen anhand verschiedener weiterer Beispielmodelle und generierter Modelle durchgeführt.

6

Fallstudie: Fault-Tolerant Fuel-Rate Controller

Die zuvor vorgestellten Methoden wurden mit dem Ziel entworfen, den Entwicklungsprozess zu beschleunigen, indem frühzeitig Timing Fehler innerhalb der AUTOSAR Timing Anforderungen und der Softwarearchitektur gefunden werden. Zur Bewertung der praktischen Anwendbarkeit wird in diesem Kapitel die Methode anhand einer konkreten Fallstudie evaluiert. Das Ziel der Evaluierung ist es zum einen festzustellen, inwieweit und an welcher Stelle Fehler identifiziert werden können, und zum anderen, um eine erste Abschätzung über die Laufzeiten an einem realitätsnahen System zu erhalten.

Zunächst wird der Aufbau des Modells genauer erläutert und auf Limitierungen eingegangen. Des Weiteren wird ein Überblick über die im Modell enthaltenen Timing Anforderungen gegeben. Danach werden die Evaluierungsergebnisse gezeigt und anhand der Ergebnisse die praktische Anwendbarkeit des Ansatzes diskutiert.

6.1 Aufbau des Modells

Signalvorverarbeitung
Kombiinstruments
Kraftstoffmenge

Das Modell beinhaltet einen fehlertoleranten Regler zur Steuerung der Kraftstoffeinspritzung eines Verbrennungsmotors. Der Regler beinhaltet eine *Signalvorverarbeitung*, die fehlerhafte Sensordaten erkennt und korrigieren kann, Funktionen zur Steuerung eines *Kombiinstruments* zur Anzeige relevanter Fahrparameter, sowie den eigentlichen Regler zur Berechnung der *Kraftstoffmenge*. Alle Funktionen sind zusammen als AUTOSAR-Steuergerät realisiert. Dieses beinhaltet auf Applikationsebene vier verschiedene Softwarekomponenten. Des Weiteren beinhaltet das Modell die Spezifikation von RTE, Betriebssystem und weiteren Basissoftwarekomponenten wie das Zustandsmanagement (EcuM) und Speicherdienste (NvRAM). Zur Absicherung von Zeitanforderungen enthält die Architektur eine Reihe von AUTOSAR Timing Constraints, wovon sich die meisten auf Zeiteigenschaften innerhalb der VFB-Sicht beziehen.

6.1.1 Softwarearchitektur

Die AUTOSAR-Softwarearchitektur des Steuergeräts besteht auf Applikationsebene aus insgesamt vier Softwarekomponenten. Die Softwarekomponente FuelsysSensors erhält die Rohdaten der Sensoren aus dem Motormodell (RpRawSensors). Dies sind beispielsweise die aktuelle Geschwindigkeit und die Gaspedalstellung. Die Softwarekomponente erkennt fehlerhafte oder fehlende Sensordaten, korrigiert diese bei Bedarf und gibt als Ergebnis die korrigierten Sensorwerte am Port PpCorrectedSensors, sowie einen berechneten Kraftstoffmodus - berechnet aus dem Motorzustand und dem Vorhandensein von Sensorfehlern - am Port PpFuelMode aus. Die Softwarekomponente FuelsysController erhält die korrigierten Sensordaten (RPCorrectedSensors), sowie den Kraftstoffmodus (RPCorrectedSensors) und berechnet die benötigte Kraftstoffmenge (PpFuelRate). Die Komponente FuelsysCombi erhält ebenfalls die korrigierten Sensorwerte (RpCombiSensors), sowie die berechnete Kraftstoffmenge aus dem FuelsysController und zeigt diese Werte, sowie weitere berechnete

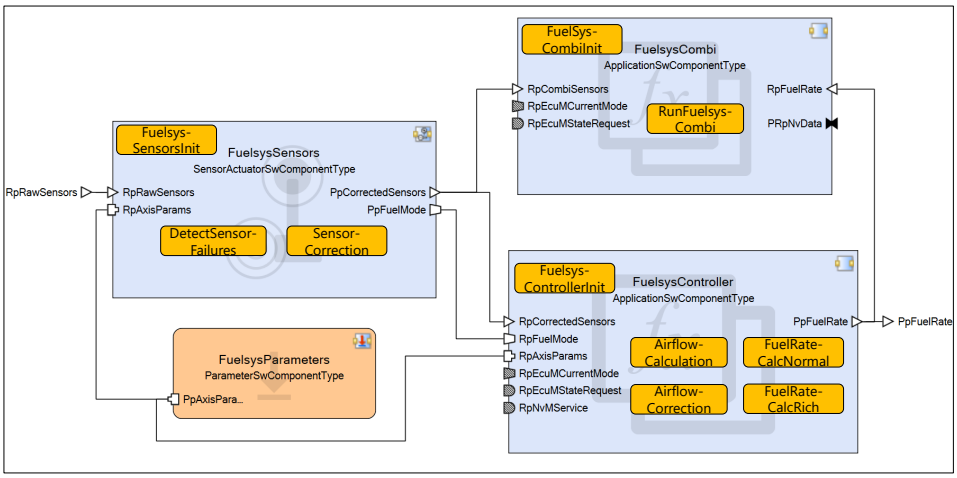

Abbildung 6.1: AUTOSAR Softwarearchitektur des Steuergeräts (Applikationsebene)

Tabelle 6.1: Komplexität der Softwarearchitektur

Anzahl der Softwarekomponenten	10
Anzahl der Applikationssoftwarekomponenten	6
Anzahl der Runnables	16
Anzahl der Tasks	6

Werte wie akkumulierte Fahrzeit, den geschätzten Kraftstoffverbrauch und die geschätzte Restfahrzeit an. Die Softwarekomponente FuelsysCalParams stellt Kalibrierungsparameterwerte für die Komponenten FuelsysSensors und FuelsysController zur Verfügung, mit denen der Regler weiter kalibriert werden kann. Die Softwarekomponenten enthalten insgesamt 16 Runnables, wovon 10 auf die Applikationssoftwarekomponenten des Steuergeräts und 6 auf Basissoftwaremodule entfallen. Diese sind auf insgesamt 6 Tasks mit unterschiedlichen Zykluszeiten verteilt. Alle Softwarekomponenten werden auf Applikationsebene zusammen mit den darin enthaltenen Runnables im Composition Diagramm in Abbildung 6.1 gezeigt. Die Tabelle 6.1 enthält eine Übersicht über die Anzahl der für die Analysemethoden relevanten Modellelemente und somit über die Modellkomplexität.

6.1.2 Timing Constraints

Das Modell enthält neben der Softwarearchitektur eine Reihe von Timing Constraints. Diese sollen sicherstellen, dass das AUTOSAR System die speziellen Akzeptanzkriterien an die Latenz und Synchronizität einhält. So darf beispielsweise zwischen dem Eingang der Sensordaten und dem Senden der neuen Kraftstoffeinspritzrate nur eine bestimmte Zeitspanne liegen, um rechtzeitig eine optimale Rate an den Motor senden zu können, sodass das Reaktionsverhalten den Erwartungen des Fahrers bei minimalem Kraftstoffverbrauch entspricht. Auch soll die Darstellung der Fahrparameter im Kombiinstrument in etwa synchron mit den realen Fahrzeugparametern sein. Insgesamt enthält das Modell 29 verschiedene Timing Constraints. Diese werden in Tabelle 6.2 gezeigt. Dabei wird für jede Anforderung eine textuelle Beschreibung und die dazugehörige formale Darstellung gezeigt. Entsprechend der Namensgebung in der AUTOSAR Architektur ist die natürlichsprachliche Anforderungsbeschreibung ebenfalls auf Englisch spezifiziert.

Tabelle 6.2: Timing Constraints

Beschreibung	Timing Constraint
The execution time of runnable RunFuelsysCombi (rfc) in component FuelsysCombi must be between 5ms and 15ms.	$r_{etc_1} = (\text{rfc}, 5, 15)$
The execution time of runnable FuelsysCombiInit (fci) in component FuelsysCombi must be between 1ms and 10ms.	$r_{etc_2} = (\text{fci}, 1, 10)$
The execution time of runnable AirflowCalculation (acl) in component FuelsysController must be between 1ms and 5ms.	$r_{etc_3} = (\text{acl}, 1, 5)$
The execution time of runnable AirflowCorrection (aco) in component FuelsysController must be between 1ms and 5ms.	$r_{etc_4} = (\text{aco}, 1, 5)$

Tabelle 6.2: Fortsetzung Timing Constraints

Beschreibung	Timing Constraint
The execution time of runnable FuelRateCalcNormal (frcn) in component FuelsysController must be between 1ms and 3ms.	$r_{etc_5} = (\text{frcn}, 1, 3)$
The execution time of runnable FuelRateCalcRich (frcr) in component FuelsysController must be between 1ms and 3ms.	$r_{etc_6} = (\text{frcr}, 1, 3)$
The execution time of runnable FuelRateControllerInit (frci) in component FuelsysController must be between 1ms and 6ms.	$r_{etc_7} = (\text{frci}, 1, 6)$
The execution time of runnable FuelsysSensorsInit (fsi) in component FuelsysSensors must be between 1ms and 6ms.	$r_{etc_8} = (\text{fsi}, 1, 6)$
The execution time of runnable DetectSensorFailures (dsf) in component FuelsysSensors must be between 1ms and 3ms.	$r_{etc_9} = (\text{dsf}, 1, 3)$
The execution time of runnable SensorCorrection (sco) in component FuelsysSensors must be between 1ms and 3ms.	$r_{etc_{10}} = (\text{sco}, 1, 3)$
The execution time of BSW executable EcuM_MainFunction (EcuM) of ECU Controller must be between 0ms and 2ms.	$r_{etc_{11}} = (\text{EcuM}, 0, 2)$
The execution time of BSW executable NvBlockDescriptor_StoreCyclic (NvM) in ECU Controller must be between 0ms and 3ms.	$r_{etc_{12}} = (\text{NvM}, 0, 3)$
The runnables DetectSensorFailures (dsf) and SensorCorrection (sco) in component FuelsysSensors must be executed in order.	$r_{eoc_1} = \langle \text{dsf}, \text{sco} \rangle$

Tabelle 6.2: Fortsetzung Timing Constraints

Beschreibung	Timing Constraint
The runnables AirflowCalculation (acl) and FuelRateCalcNormal (frcn) in component FuelsysController must be executed in order.	$r_{eoc_2} = \langle \text{acl}, \text{frcn} \rangle$
The runnables AirflowCalculation (acl) and FuelRateCalcRich (frcr) in component FuelsysController must be executed in order.	$r_{eoc_3} = \langle \text{acl}, \text{frcr} \rangle$
The runnables AirflowCorrection (aco) and FuelRateCalcRich (frcr) in component FuelsysController must be executed in order.	$r_{eoc_4} = \langle \text{aco}, \text{frcr} \rangle$
The runnables AirflowCorrection (aco) and FuelRateCalcNormal (frcn) in component FuelsysController must be executed in order.	$r_{eoc_5} = \langle \text{aco}, \text{frcn} \rangle$
The runnable FuelRateCalcNormal (frcn) in component FuelsysController must be completed between 3ms and 10ms after the runnable AirflowCalculation (acl) in component FuelsysController has been started.	$r_{otc_1} = (e^s_{\text{acl}}, e^t_{\text{frcn}}, 3, 10)$
The runnable FuelRateCalcRich (frcr) in component FuelsysController must be completed between 2ms and 6ms after the runnable AirflowCalculation (acl) in component FuelsysController has been started.	$r_{otc_2} = (e^s_{acl}, e^t_{frcr}, 2, 6)$
The runnable RunFuelsysCombi (rfc) in component FuelsysCombi must be completed in at most 100ms and at leat 1ms after the data element Sensors of port PpCorrectedSensors (ppcs) in component FuelsysSensors has been updated.	$r_{otc_3} = (e_{ppcs\_sensors}, e^t_{rfc}, 1, 100)$

Tabelle 6.2: Fortsetzung Timing Constraints

Beschreibung	Timing Constraint
The data element Sensors (se) must be updated synchronously at port RpCorrectedSensors (rpcs) in component FuelsysController and RpCombiSensors (rpcombis) in component FuelsysCombi with a tolerance value of 100ms.	$r_{stc_1} = (\{e_{rpcs\_se}, e_{rpcombis\_se}\}, 100)$
The data element Sensors (se) of port RpCombiSensors (rpcombis) and FuelRate (fr) of port RpFuelRate (rpfr) in component FuelsysCombi shall be updated synchronously with a maximum tolerance value of 100ms.	$r_{stc_2} = (\{e_{rpcombis\_se}, e_{rpfr\_fr}\}, 100)$
The termination of runnables RunFuelSysCombi (rfc) in component FuelsysCombi and FuelRateCalcNormal (frcn) and FuelRateCalcRich (frcr) in component FuelsysController must happen synchronously with a tolerance value of 3ms.	$r_{stc_3} = (\{e^t_{rfc}, e^t_{frcn}, e^t_{frcr}\}, 3)$
The start of the runnables RunFuelSysCombi (rfc) in component FuelSysCombi and AirflowCalculation (acl) in component FuelsysController must happen synchronously with a tolerance value of 3ms.	$r_{stc_4} = (\{e^s_{rfc}, e^s_{alc}\}, 3)$
The data element Sensors (se) of port PpCorrectedSensors (ppcs) and fuelMode (fm) of port PpFuelMode (ppfm) in FuelsysSensors must be updated synchronously with a tolerance value of 2ms.	$r_{stc_5} = (\{e_{ppcs\_se}, e_{ppfm\_fm}\}, 2)$

Tabelle 6.2: Fortsetzung Timing Constraints

Beschreibung	Timing Constraint
The data element Sensors (se) of port RpCorrectedSensors (rpcs) and fuelMode (fm) of port RpFuelMode (rpfm) in FuelsysController must be updated synchronously with a tolerance value of 1ms.	$r_{stc_6} = (\{e_{rpcs\_se}, e_{rpfm\_fm}\}, 1)$
When values at port RpCorrectedSensors (rpcs) in component FuelsysController are received then the new fuel rate value at port PpFuelRate (ppfr) in component FuelSysController shall be updated within 25ms.	$r_{ltc_1} = (\langle e_{rpcs\_se}, e_{ppcs\_se}\rangle, 0, 25)$
When data element throttle (t) at port RpRawSensors in component FuelSysSensors is received then the new corrected sensor values (se) at port PpCorrectedSensors (ppcs) in component FuelsysSensors shall be updated within at most 10ms.	$r_{ltc_2} = (\langle e_{rprs\_t}, e_{ppcs\_se}\rangle, 0, 10)$
When data element speed (sp) at port RpRawSensors (rprs) in component FuelSysSensors are received then the new fuel rate value at port PpFuelRate (ppfr) in component FuelSysController shall be available within 1 and 50ms.	$r_{ltc_3} = (\langle e_{rprs\_sp}, e_{ppcs\_se}, e_{rpcs\_se}, e_{ppfr\_fr}\rangle, 1, 50)$

6.1.3 Limitierungen des Modells

Das Ziel dieser Modellevaluierung ist es, die praktische Anwendbarkeit der Analysemethoden bewerten zu können. Dafür ist es notwendig, dass das Modell hinsichtlich Aufbau und Komplexität Modellen entspricht, wie sie bei dSPACE Kunden bzw. AUTOSAR-Endnutzern vorhanden sind. Diesen Kriterien kommt das Modell sehr nahe, da es auf der Grundlage von Erfahrungen mit Kundenmodellen im Support und in Entwicklungsabteilungen

bei dSPACE entstanden ist. Nichtsdestotrotz ist es kein reales, von dSPACE Kunden erstelltes Modell, sodass die Modell- und Strukturkomplexität von Kundenmodellen abweichen kann.

6.2 Ergebnisse

In diesem Abschnitt werden die Ergebnisse der Konsistenzanalyse und Timing Verifikation des Anwendungsfalls präsentiert, sowie die Laufzeiteffizienz der Methoden vorgestellt und begründet. Dafür werden sowohl die Laufzeiten für alle benötigten Schritte vorgestellt, als auch beispielhaft Teile der generierten Modelle.

6.2.1 Testaufbau

Der verwendete SMT-Solver ist Z3 in Version 4.6.0-x64. Für die Timing Verifikation wurde UPPAAL Version 4.0.13 verwendet. Die Ausführung und Laufzeitmessung der Testszenarien wurde auf einem Windows 10 Professional System mit Intel Core i7 4800MQ mit 32GB Arbeitsspeicher durchgeführt. UPPAAL wurde mit BFS, konservativer Zustandsraumreduktion und DBM Zustandsrepräsentation ausgeführt. Die Laufzeit wurde in Sekunden gemessen. Aufgrund des aktuelleren Testaufbaus sind die Laufzeiten nicht mit denen aus Beringer und Wehrheim (2016) und Beringer und Wehrheim (2020) vergleichbar. Die in den Ergebnissen verwendeten Symbole werden in Tabelle 6.3 erklärt.

6.2.2 Ergebnisse der Konsistenzanalyse

Mithilfe der Konsistenzanalyse konnte für das vorliegende Fallbeispiel gezeigt werden, dass die Anforderungsmenge konsistent ist. Dafür wurden aus den Timing Constraints insgesamt 120 Formeln erzeugt, die von Z3 geprüft wurden und erfüllbar sind. Das Ergebnis und die Abhängigkeiten der einzelnen Timing Constraints untereinander lässt sich am einfachsten durch die Visualisierung des Ergebnisgraphen erkennen. Dieser besteht aus mehreren unverbundene Teilgraphen. Diese werden in den Abbildungen 6.2,

Tabelle 6.3: Beschreibung der Symbole

Symbol	Beschreibung
$T(t)$	Laufzeit für die Transformation nach SMT in Sekunden
$T(smt)$	Laufzeit zum Lösen der SMT-Formel in Sekunden
$T(msat)$	Laufzeit für MaxSMT und Unsat Core in Sekunden
$T(tv)$	Laufzeit für die Modelltransformation nach Timed Automata
$T(tv_r)$	Laufzeit für die Transformation der Timing Anforderung nach Timed Automata
$T(tv_m)$	Laufzeit für die Transformation des AUTOSAR Systemmodells nach Timed Automata
$T(v)$	Laufzeit für die Timing Verifikation in Sekunden
ratio	Laufzeitverhältnis in Prozent $\frac{T(t)+T(smt)+T(msat)}{T(tv)+T(v)} * 100$

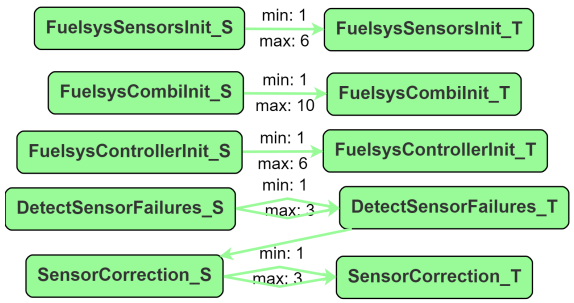

Abbildung 6.2: Teilgraph $G'_1 \subset G$ des Ergebnisgraphen für die Timing Anforderungen etc_1, etc_2, etc_7, etc_8, etc_9, etc_{10} und eoc_1

6.3, 6.4 und 6.5 dargestellt. Da die Anforderungsmenge konsistent ist, sind die Ergebnisgraphen G_{max} und G_{uc} identisch. Der Teilgraph G'_1 enthält überwiegend Execution Time Constraints, die für die Gesamtanalyse zu Offset Timing Constraints auf der Basis von Start- und Terminierungs-Events transformiert wurden. Die einzelnen Constraints haben keine weiteren Abhängigkeiten zu anderen Constraints und sind somit für die Konsistenz der Gesamtmenge unkritisch.

Der Teilgraph G'_2 visualisiert die Reihenfolgebeziehungen zwischen den Runnables der Komponente FuelsysController, die sich sowohl aus den Execution Order Constraints als auch aus den Offset Timing Constraints ergeben. Der Graph zeigt, dass die einzelnen Anforderungen sehr stark mitein-

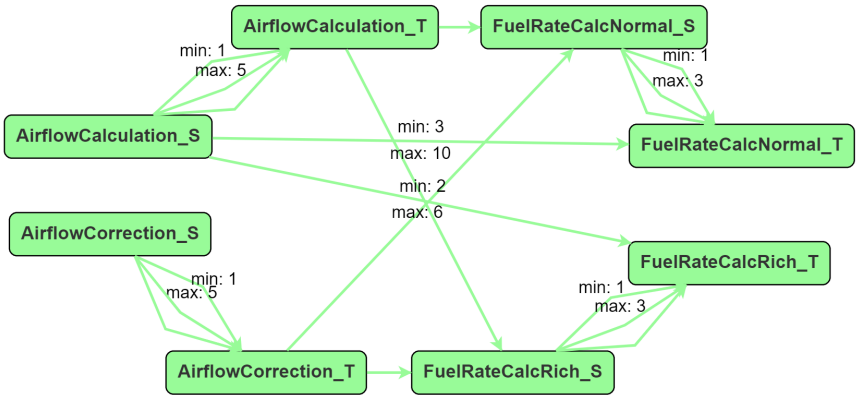

Abbildung 6.3: Teilgraph $G'_2 \subset G$ des Ergebnisgraphen für die Timing Anforderungen etc_3, etc_4, etc_5, etc_6, eoc_2, eoc_3, eoc_4, eoc_5, otc_1 und otc_2

Tabelle 6.4: Laufzeiten für die Transformation Konsistenzanalyse bestehend aus der Transformation des AUTOSAR Modells nach SMT $T(t)$, sowie das Lösen der SMT-Formel $T(smt)$ und die Berechnung von Unsat Core und MaxSMT $T(msat)$

$T(t)$	7,3
$T(smt)$	0,04
$T(msat)$	0,04

ander verwoben sind. Es lässt sich aber auch im Graph sehr schnell erkennen, dass es möglich ist die einzelnen Events in eine zeitliche Reihenfolge zu bringen. Der Teilgraph G'_3 enthält überwiegend Synchronization Timing Constraints und Latency Timing Constraints. Durch die Größe des Graphs und die Anzahl der Kanten lässt sich erkennen, dass die zeitlichen Abhängigkeiten zwischen den einzelnen Timing Anforderungen hoch ist. Dies zeigt, dass eine manuelle Konsistenzprüfung für dieses Modell vermutlich fehleranfällig wäre oder sehr lange dauern würde. Die automatisierte Überprüfung hingegen ist mit insgesamt 7,3 Sekunden sehr schnell durchgeführt. Dabei entfällt die größte Zeit auf die Transformation der Timing Anforderungen nach SMT. Die Zeit zum Lösen der Ungleichungen in Z3 hingegen ist sehr gering. Dies liegt auch daran, dass die Anzahl der Formeln mit 120 sehr überschaubar ist. Ein Überblick über die Laufzeiten der Konsistenzanalyse findet sich in Tabelle 6.4.

Abbildung 6.4: Teilgraph $G'_3 \subset G$ des Ergebnisgraphen für die Timing Anforderungen stc_1, stc_2, stc_4, stc_5, stc_6, ltc_1, ltc_2, ltc_3

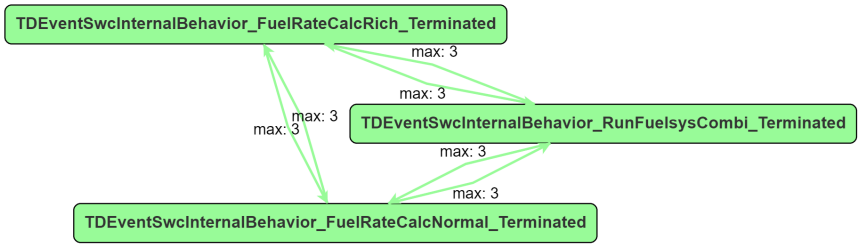

Abbildung 6.5: Teilgraph $G'_4 \subset G$ des Ergebnisgraphen für die Timing Anforderung stc_3

6.2.3 Ergebnisse der Timing Verifikation

Ein Ergebnis der Timing Verifikation ist zunächst, dass mithilfe des generierten Automatennetzwerks nicht nur die spezifizierten Timing Anforderungen geprüft werden können, sondern auch grundsätzliche Modellierungsfehler im AUTOSAR-Modell erkannt werden können, die durch eine inkorrekte Verwendung zeitbehafteter Modellelemente entstehen. Dies kann beispielsweise eine fehlerhafte Zuweisung von Runnables zu Tasks sein. In diesem Fall können Runnables identifiziert werden, deren Worst-Case Execution Time größer ist als die spezifizierte

Zykluszeit des zugewiesenen OSTasks. Des Weiteren können Tasks auf dem Steuergerät identifiziert werden, die nicht innerhalb der angegebenen Zykluszeit ausgeführt werden können, da die Worst-Case Execution Times von Runnables auf anderen Tasks zu hoch sind. Diese Fehler können im Rahmen der Verifikation dadurch erkannt werden, dass das Modell nicht deadlockfrei ist. Für das vorliegende Modell konnten keine Fehler identifiziert werden.

Das generierte Timed Automata Netzwerk des Systemmodells besteht aus insgesamt 65 Timed Automata, für jede Timing Anforderung wurde zudem ein Test-Automat erzeugt, der zusammen mit dem Systemmodell verifiziert wird.

Alle Timing Anforderungen wurden sequenziell verifiziert und sind erfüllbar. Die Abbildung 6.6 zeigt beispielhaft den generierten Test-Timed Automaton für den Latency Timing Constraint ltc_3. Dieser Test-Automat prüft, ob das Timing von der Ankunft neuer Sensordaten bis zur Bereitstellung neuer Werte für die Kraftstoffmenge eingehalten wird, indem die maximale Zeit für das durchlaufen der spezifizierten Ereigniskette geprüft wird. Abbildung 6.7 zeigt den Test-Automaton für den Execution Order Constraint eoc_3. Dieser stellt sicher, dass vor der Ausführung des Runnables zur Korrektur der Sensorwerte zunächst das Runnable zur Erkennung der Fehler ausgeführt wird. Dies wird sichergestellt, indem bei einer fehlerhaften Ausführungsreihenfolge ein unerwartetes Event empfangen wird und im Automat eine Transition zu einem Fehlerzustand getriggert wird. Eine Übersicht über die Laufzeiten für alle Timing Anforderungen findet sich in Tabelle 6.5.

Die Ergebnisse der Tabelle zeigen ebenfalls die Laufzeiten der einzelnen Schritte. Diese sind je nach verwendeter AUTOSAR Timing Extension sehr unterschiedlich. So können Offset Timing Constraints mit durchschnittlich 6 Sekunden am schnellsten verifiziert werden. Für die Verifikation eines Execution Order Constraints werden bereits durchschnittlich 150 Sekunden benötigt und für Synchronization Timing Constraints 102 Sekunden. Für die Latency Timing Constraints werden sogar 502 Sekunden benötigt. Diese Laufzeitunterschiede lassen sich ebenfalls in der Abbildung 6.9 gut erkennen, die die Mittelwerte für die Laufzeiten der einzelnen Timing Anforderungs Typen in Beziehung zueinander setzt. Die Transformationen der

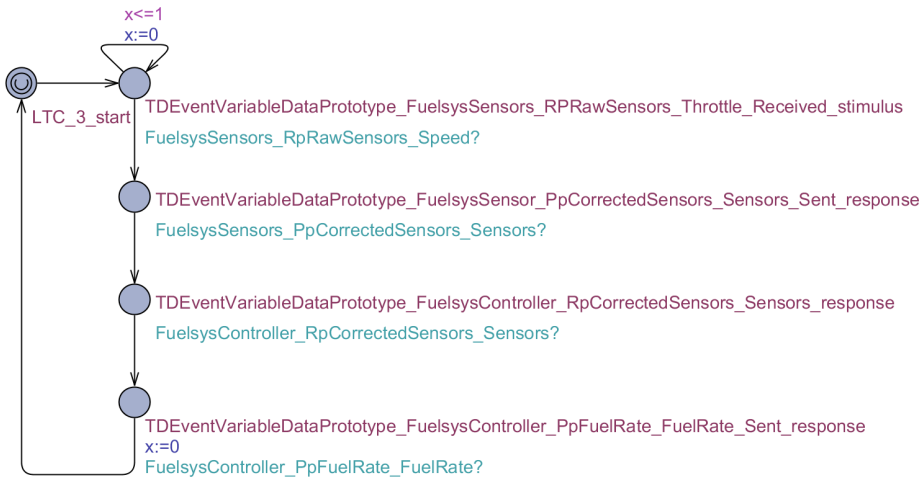

Abbildung 6.6: Timed Automaton für ltc_3

AUTOSAR Timing Anforderungen ($T(tv_r)$) und des Systemmodells nach Timed Automata ($T(tv_m)$) dauern für jede Timing Anforderung mit ungefähr 1 Sekunde und 22 Sekunden in etwa gleich lange. Werden mehrere Anforderungen gleichzeitig verifiziert, ist es zudem nur notwendig die Architekturtransformation einmalig durchzuführen, was die Gesamtlaufzeit verkürzt. Der gesamte Verifikationsdurchlauf dauert zusammen mit der Zeit zur Transformation insgesamt 3273,8 Sekunden. Eine grafische Darstellung der Ergebnisse findet sich in Abbildung 6.8.

6.3 Diskussion

Der gezeigte Anwendungsfall gibt einen Einblick in die praktische Anwendbarkeit der Methode und deren Nutzen. Es konnte gezeigt werden, dass sowohl die Konsistenzanalyse als auch die Timing Verifikation anwendbar waren und das System alle Anforderungen erfüllt. Die durchgeführte Konsistenzanalyse konnte die Konsistenz der Anforderungsmenge bereits vor der Verifikation sicherstellen, sodass sich eine Korrektur der Timing Anforderungen erübrigte. Dies lässt allerdings auch die Fragestellung offen, inwieweit erkannte Inkonsistenzen dazu beitragen

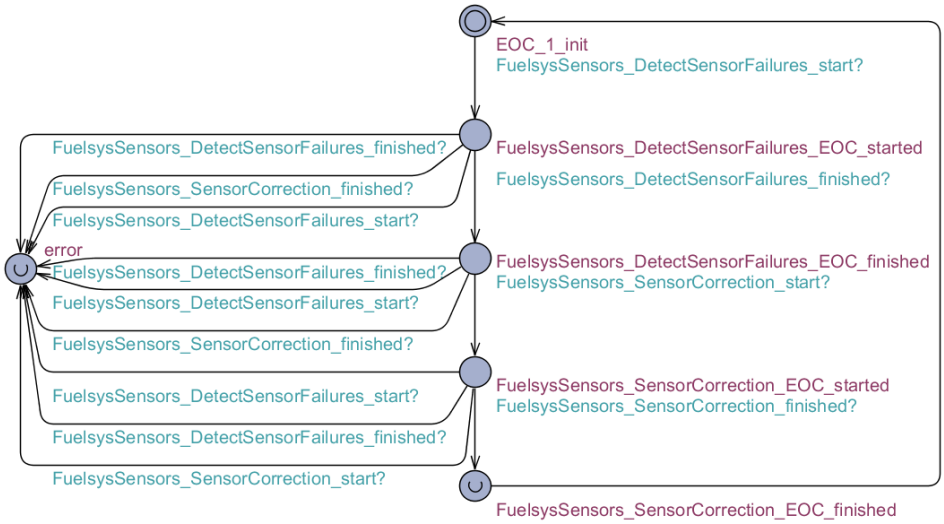

Abbildung 6.7: Timed Automaton für eoc_1

würden, die Gesamtlaufzeit der Methode zu verkürzen, indem beispielsweise inkonsistente Teilmengen aus der Gesamtmenge herausgenommen werden und somit nicht mehr verifiziert werden müssen oder indem davon ausgegangen wird, dass ohne die Anwendung der Konsistenzanalyse unerfüllbare Timing Anforderungen innerhalb der inkonsistenten Teilmengen mehrfach verifiziert werden müssen, da diese ohne eine Konsistenzanalyse vorab nicht erkannt worden wären. Nichtsdestotrotz kann argumentiert werden, dass in diesem Fall die Durchführung der Konsistenzanalyse einen Mehrwert für das Verständnis der Anforderungsmenge geschaffen hat. Denn ein Requirements Engineer kann so die Abhängigkeiten der Anforderungen im Ergebnisgraph einsehen und ist so dazu befähigt bei der Änderung oder Erweiterung der Anforderungen die Konsistenz im Blick zu behalten. Gleichzeitig ist der Zeitaufwand für die Durchführung der Analyse vernachlässigbar.

Die Timing Verifikation konnte ebenfalls für die beschriebenen Anforderungen und das Systemmodell durchgeführt werden. Das Ergebnis zeigt, dass das Systemmodell alle spezifizierten Timing Constraints erfüllt.

Ein Ergebnis der gemessenen Laufzeiten ist, dass die Timing Verifikation wesentlich mehr Zeit in Anspruch nimmt als die Konsistenzanalyse. Nur

Tabelle 6.5: Ergebnisse und Laufzeiten der Timing Verifikation

Timing Anforderung	$T(tv_r)$	$T(tv_m)$	$T(tv)$	$T(v)$	Ergebnis
$r_{eoc_1} = \langle \mathrm{dsf}, \mathrm{sco} \rangle$	0,93	21,1	22,1	98,4	SAT
$r_{eoc_2} = \langle \mathrm{acl}, \mathrm{frcn} \rangle$	0,94	20,9	22,0	163,9	SAT
$r_{eoc_3} = \langle \mathrm{acl}, \mathrm{frcr} \rangle$	0,95	21,1	22,2	164,1	SAT
$r_{eoc_4} = \langle \mathrm{aco}, \mathrm{frcr} \rangle$	0,93	21,0	22,0	164,6	SAT
$r_{eoc_5} = \langle \mathrm{aco}, \mathrm{frcn} \rangle$	0,95	20,9	22,0	159,7	SAT
$r_{otc_1} = (e^s_{\mathrm{acl}}, e^t_{\mathrm{frcn}}, 3, 10)$	0,94	20,8	21,8	5,5	SAT
$r_{otc_2} = (e^s_{acl}, e^t_{frcr}, 2, 6)$	0,91	20,9	21,9	5,4	SAT
$r_{otc_3} = (e_{ppcs\_sensors}, e^t_{rfc}, 1, 100)$	1,01	21,0	22,1	7,1	SAT
$r_{stc_1} = (\{e_{rpcs\_se}, e_{rpcombis\_se}\}, 100)$	1,14	23,2	24,5	107,4	SAT
$r_{stc_2} = (\{e_{rpcombis\_se}, e_{rpfr\_fr}\}, 100)$	0,97	20,7	21,9	146,6	SAT
$r_{stc_3} = (\{e^t_{rfc}, e^t_{frcn}, e^t_{frcr}\}, 3)$	1,03	20,8	21,9	96,5	SAT
$r_{stc_4} = (\{e^s_{rfc}, e^s_{alc}\}, 3)$	0,98	20,7	21,8	92,9	SAT
$r_{stc_5} = (\{e_{ppcs\_se}, e_{ppfm\_fm}\}, 2)$	1,0	20,7	21,8	86,0	SAT
$r_{stc_6} = (\{e_{rpcs\_se}, e_{rpfm\_fm}\}, 1)$	1,06	22,3	23,6	87,2	SAT
$r_{ltc_1} = (\langle e_{rpcs\_se}, e_{ppcs\_se} \rangle, 0, 25)$	1,12	23,3	24,6	601,2	SAT
$r_{ltc_2} = (\langle e_{rprs\_t}, e_{ppcs\_se} \rangle, 0, 10)$	1,13	20,9	22,1	35,3	SAT
r_{ltc_3}	1,51	21,2	22,9	870,8	SAT
$\sum$	17,5	361,5	381,2	2892,6	-

0,22% der Gesamtlaufzeit wird für die Konsistenzanalyse benötigt. Dies liegt daran, dass die Anzahl der generierten SMT-Formeln überschaubar ist. Der größte Anteil wird für die Verifikation des generierten Timed Automata Netzwerks in UPPAAL benötigt. Da die Anzahl der Clocks einen sehr großen Einfluss auf die Laufzeit hat, wird davon ausgegangen, dass die Laufzeiten bei größeren Modellen weiter steigt.

Durch die Beschränkung auf den Anwendungsfall sind die Ergebnisse nicht auf beliebige andere Systeme übertragbar. Zum einen können Systemmodelle hinsichtlich der Modellkomplexität voneinander abweichen, was die Laufzeiten der einzelnen Schritte stark beeinflussen kann. Des Weiteren kann auch die Menge und Komplexität der Timing Anforderungen variieren.

Die Vermutung liegt nahe, dass gerade bei einer großen Anforderungs-

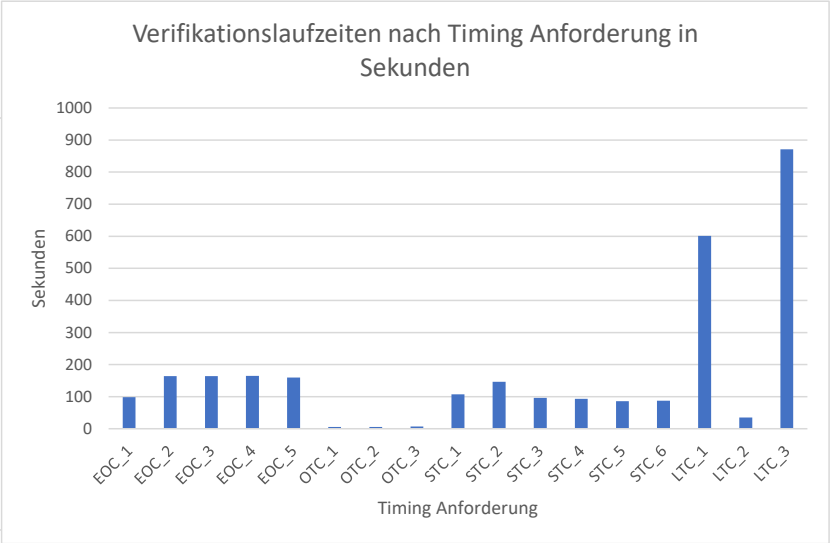

Abbildung 6.8: Laufzeit der Verifikation der Timing Anforderungen

menge die Wahrscheinlichkeit für enthaltene Inkonsistenzen besonders hoch ist und somit häufiger Fehler gefunden werden.

6.4 Zusammenfassung

In diesem Kapitel wurde anhand eines konkreten Anwendungsfalls gezeigt, wie sich Konsistenzanalyse und Timing Verifikation zusammen zur frühzeitigen Timing Analyse von AUTOSAR Softwarearchitekturen einsetzen lassen. Zunächst wurde in Abschnitt 6.1 der Aufbau der AUTOSAR Softwarearchitektur, sowie die Timing Anforderungen vorgestellt. Anschließend wurden in Abschnitt 6.2 die Evaluierungsergebnisse vorgestellt. Da die Menge der Timing Anforderungen keine Inkonsistenzen enthielt, konnte die Konsistenzanalyse nicht zur Verkürzung der Verifikationszeiten beitragen. Nichtsdestotrotz war der Aufwand für die Konsistenzanalyse im Vergleich zur Timing Verifikation vernachlässigbar. Alle Ergebnisse

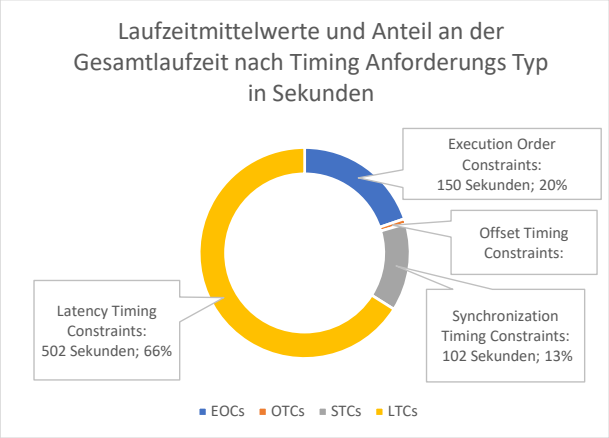

Abbildung 6.9: Laufzeitverhältnis der einzelnen Schritte

und Laufzeiten geben jedoch nur die konkrete Sicht auf diesen einen Anwendungsfall wieder. Dabei können in anderen Modellen sowohl die Komplexität des AUTOSAR-Systemmodells als auch die Menge und Art der Timing Constraints wesentlich voneinander abweichen, was auch die Laufzeiten beeinflussen kann. Um herauszufinden inwieweit die einzelnen Methoden skalierbar sind und sich somit eine Kombination aus Laufzeitgründen als sinnvoll ergibt, wird im nächsten Kapitel genauer betrachtet.

7

Werkzeugunterstützung und Evaluierung

Dieses Kapitel beinhaltet die wesentlichen Ergebnisse der praktischen Anwendung der entwickelten Methode. Zunächst werden einige spezifische Details des prototypisch entwickelten Werkzeugs vorgestellt wie beispielsweise die Softwarearchitektur und die Integration in die existierende Werkzeuglandschaft. Danach werden die mit diesem Werkzeug ermittelten Laufzeitergebnisse für eine Reihe von Beispielszenarien vorgestellt.

7.1 Prototypische Werkzeugunterstützung

Zur Durchführung der Analysen und zur Generierung von Evaluierungsergebnissen wurde in dieser Arbeit eine prototypische Werkzeugunterstützung durchgeführt. Das Werkzeug verwendet die bereits erwähnten Werkzeuge Z3 und UPPAAL zur Analyse der jeweiligen generierten Analysemodelle, die aus den AUTOSAR Modellen erstellt wurden. Der Zugriff auf die AUTOSAR Modelle erfolgt über das Werkzeug SystemDesk.

7.2 Realisierung der Modelltransformationen

Für die Realisierung der Transformationen existieren verschiedene Ansätze. So gibt es zur Spezifikation von Modell-zu-Modell-Transformationen (M2M) etablierte Sprachen und Frameworks wie eMoflon::IBEX (Weidmann et al., 2019) oder Viatra 3 (Bergmann et al., 2015), die auf den Grundlagen von Triple-Graph-Grammatiken (TGGs) von Schürr (1994) basieren, relationale Ansätze wie QVT der Object Management Group (2016) oder hybride Ansätze wie die ATL Transformation Language (Martínez et al., 2017), die eine Spezifikation der Transformationsregeln sowohl imperativ als auch deklarativ erlauben. Neuere Methoden beispielsweise von Anjorin et al. (2022) und Weidmann und Anjorin (2021) betrachten die Beschreibung bidirektionaler Transformationen mittels TGGs. Mithilfe
BX Bidirektionaler Transformation (*BX*) lässt sich die Konsistenz (Inter-Modell Konsistenz) zwischen zwei oder mehreren verschiedenen Artefakten überprüfen und wiederherstellen, indem die Beziehungen zwischen Modellen verschiedener Metamodelle so festgehalten und mithilfe eines Werkzeugs ausgeführt werden, dass Transformationen sowohl vorwärts (von einem Ursprungsmodell in ein Zielmodell) als auch rückwärts (vom Zielmodell zurück in das Ursprungsmodell) transformiert und synchronisiert werden können (Anjorin et al., 2020a). Die Beschreibung der Beziehungen werden in Form einer Konsistenzrelation, die durch eine TGG realisiert wird und deren Sprache alle konsistenten Modellpaare beinhaltet, festgehalten (Anjorin et al., 2020a). So kann beispielsweise die Nachverfolgbarkeit von Anforderungen in Architekturmodellen gewartet werden, auch wenn sich Modelle sowohl im Anforderungswerkzeug als auch im Architekturwerkzeug ändern (Anjorin et al., 2022). Durch die Verknüpfung von TGGs mit Optimierungsmethoden können auch komplexere Konsistenzrelationen, die weitere Domänenbedingungen beinhalten wie beispielsweise die Berücksichtigung von Multiplizitäten, realisiert (Weidmann und Anjorin, 2021) oder sogar Optimierungsprobleme beschrieben und Initiallösungen bestimmt werden (Anjorin et al., 2020b).

Für die Transformationen von AUTOSAR nach SMT, sowie AUTOSAR nach Timed Automata wurde ein imperativer Ansatz gewählt. Die Trans-

formationsregeln sind direkt im Framework festgehalten, das auf dem Objektmodell von SystemDesk arbeitet, und generieren als Zielartefakte ein Z3 und ein Timed Automata Objektmodell. Der wesentliche Grund für diese Entscheidung ist, dass die entwickelte Methode möglichst nahtlos in SystemDesk integriert werden sollte, sodass ein Anwender keine zusätzlichen Schritte durchführen muss oder zusätzliche Werkzeuge nutzen muss. So entfällt die Einbindung eines passenden BX-Werkzeugs, was zusätzliche Komplexität in die Werkzeugkette gebracht hätte, da viele existierende Werkzeuge wie eMofolon zum einen nur für das Eclipse-Framework zur Verfügung stehen, das jedoch nicht direkt in SystemDesk integriert werden kann und somit weitere Transformationsschritte für das AUTOSAR-Modell in Form von In- und Exports angefallen wären, und zum anderen das vollständige AUTOSAR Metamodell nicht für das Eclipse-Modeling-Framework als EMF Ecore-Modell zur Verfügung stand, was für diese Frameworks eine Voraussetzung ist. Ebenfalls existieren keine Metamodelle für SMT und Timed Automata.

Der Nachteil der direkten Implementierung der Transformationsregeln ist jedoch, dass die Transformationen im Gegensatz zu einer Realisierung mittels BX-Werkzeug nur unidirektional erfolgen. Dies führt dazu, dass nach Änderungen des AUTOSAR-Modells das jeweilige Analysemodell immer neu erstellt werden muss. Dies kann vernachlässigt werden, da die Generierung der Analysemodelle keine hohen Laufzeiten aufweisen. Ebenfalls ist die Rückverfolgbarkeit der Elemente der Analysemodelle nach AUTOSAR nicht gegeben. Dies führt zu zusätzlichem Aufwand bei der Interpretation der Analyseergebnisse, da sowohl die Ergebnisse aus UPPAAL als auch die Ausgaben des SMT-Solvers und der Visualisierung manuell auf die entsprechenden AUTOSAR Timing Constraints und die restlichen AUTOSAR Modellelemente übertragen werden müssen. Diesem Problem hätte durch die Verwendung von BX-Methoden Rechnung getragen werden können, wenn diese ebenfalls die Ergebnisse der Analysen in den jeweiligen Metamodellen berücksichtigt würden.

7.2.1 Softwarearchitektur

Die Implementierung erfolgte unter der Verwendung der von SystemDesk zur Verfügung gestellten Automatisierungsschnittstelle zur Erstellung und Modifikation von AUTOSAR Modellen und den bereits vorgestellten Werkzeugen bzw. Frameworks UPPAAL und Z3. Das entwickelte Framework besteht aus insgesamt sechs Komponenten:

Modellverifikation

1. ARVerifier: Die Komponente liefert die Schnittstelle zur AUTOSAR *Modellverifikation* und beinhaltet alle Funktionen zur Transformation von AUTOSAR Modellen zu Timed Automata. Zur Verifikation der Timed Automata verwendet die Komponente UPPAAL. Da UPPAAL keine offenen Schnittstellen zur Generierung und Verifikation zur Verfügung stellt, wurde ein Mechanismus implementiert, der aus dem Timed Automata Modell ein für UPPAAL lesbares XML-Dokument erzeugt. Die Steuerung des Model Checkers erfolgt über Kommandozeilenaufrufe.

Konsistenzanalyse

2. ARConsistencyChecker: Die Komponente liefert die Schnittstelle zur *Konsistenzanalyse* von AUTOSAR Timing Anforderungen und beinhaltet alle Funktionen zur Transformation von AUTOSAR Timing Anforderungen nach SMT. Zur Generierung der SMT-Formel verwendet die Komponente die Schnittstelle von Z3, um ein Z3-kompatibles Objektmodell in C# zu erzeugen und dieses anschließend mit Z3 zu lösen. Im Gegensatz zur ARVerifier Komponente ist es somit nicht notwendig, das Objektmodell vorab zu exportieren.

Evaluierung

3. ARVerifierTests: Die Komponente beinhaltet Funktionen zur *Evaluierung* des Analyseframeworks. Die Komponente verwendet die Analysekomponenten ARVerifier und ARConsistencyChecker und die Komponente ARGenerate zur Generierung von AUTOSAR Modellen.

synthetischen AUTOSAR Modellen

4. ARGenerate: Die Komponente bietet eine Schnittstelle zur Generierung von *synthetischen AUTOSAR Modellen* und AUTOSAR Timing Anforderungen.

5. ARGraph: Die Komponente beinhaltet Funktionalitäten zur Visualisierung der *Abhängigkeitsgraphen* in Form einer eigenständigen Webapplikation. Die Komponente verwendet dafür die API der ARConsistencyChecker Komponente.

   Abhängigkeitsgraphen

6. ARVerifierExtra: Die Komponente beinhaltet eine Visualisierungskomponente, die es ermögicht die Analysekomponenten über die *grafische Oberfläche* von SystemDesk zu starten. Dies wird über die Implementierung eines Plugins für SystemDesk realisiert. Die Komponente verwendet dafür die Schnittstellen der Analysekomponenten ARVerifier und ARConsistencyChecker.

   grafische Oberfläche

Abbildung 7.1 gibt eine Übersicht über die entwickelten und eingebundenen Komponenten. Die Komponenten in gelb sind die beiden AUTOSAR Analysekomponenten. Komponenten in lila beinhalten Funktionalität zur Darstellung von Ergebnissen und zur Einbindung des Frameworks in SystemDesk. Komponenten in rot beinhalten Funktionalität zum Test und zur Evaluierung des Frameworks.

7.3 Evaluierung

In diesem Abschnitt wird die Laufzeit des Ansatzes anhand weiterer Beispielmodelle und generierter Testszenarien evaluiert. Es werden dabei zunächst die Rahmenbedingungen für die Testszenarien beschrieben. Dazu gehört die Beschreibung des Generierungsprozesses für die Testszenarien, sowie spezifische Eigenschaften der Beispielmodelle. Danach werden die Laufzeitergebnisse für die Konsistenzanalyse und Timing Analyse vorgestellt. Anhand dieser Ergebnisse wird dann argumentiert, ob und wie die Methoden kombiniert werden können und ob dies in der Praxis sinnvoll ist. Die hier vorgestellten Evaluierungsergebnisse wurden teilweise bereits in Beringer und Wehrheim (2016) und Beringer und Wehrheim (2020) veröffentlicht.

Die Laufzeitanalyse der Fallstudie wurde mit demselben Testaufbau realisiert, der auch im vorherigen Kapitel für die Evaluierung der Methode

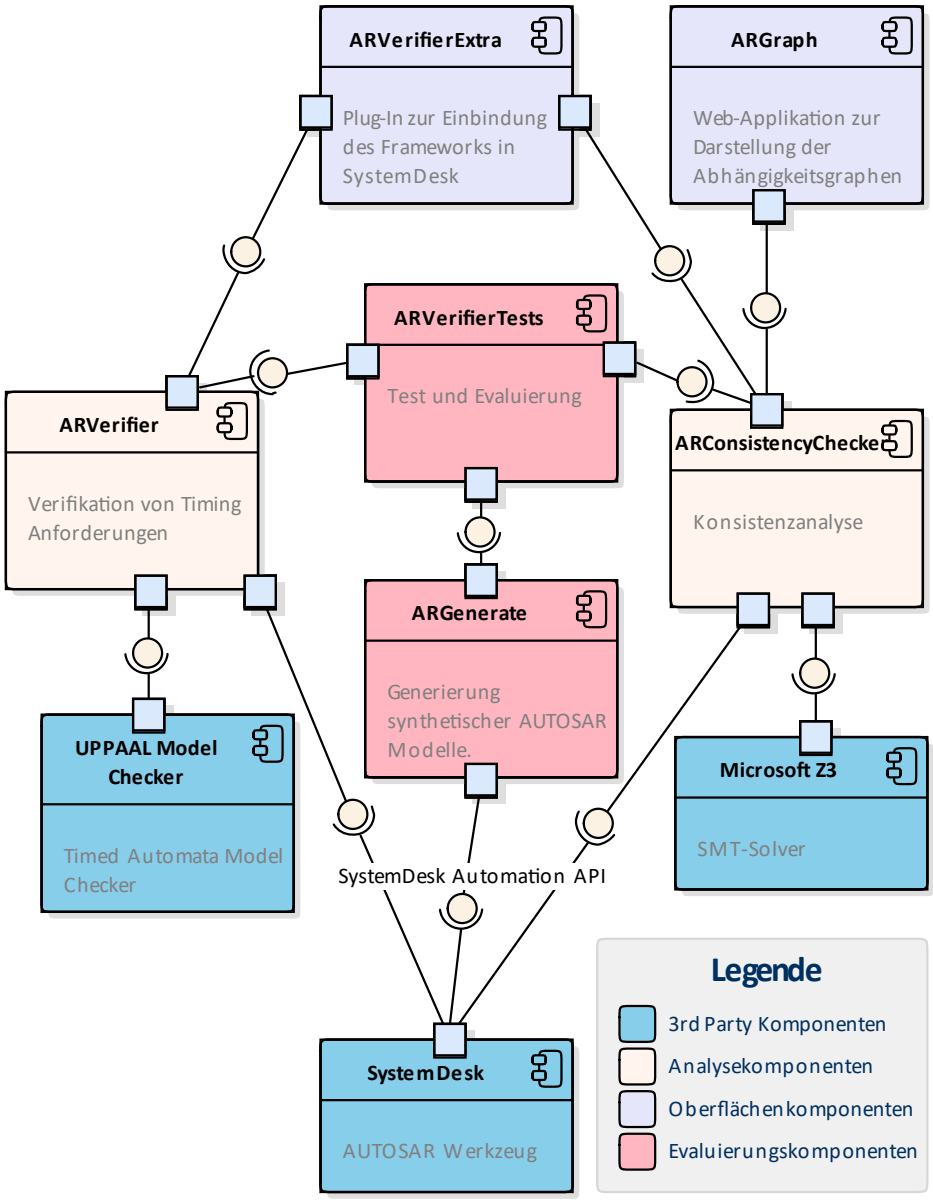

Abbildung 7.1: Komponentenansicht des Analyseframeworks

anhand einer Fallstudie verwendet wurde. Dieser Testaufbau wird in Abschnitt 6.2.1 vorgestellt. Die ebenfalls in diesem Kapitel verwendeten Metriken sind in Tabelle 6.3 zu finden.

7.3.1 Testszenarien

Zunächst wird die Laufzeit für die Transformation der Timing Anforderungen nach SMT, das Lösen der SMT-Formel mittels Z3 und das Finden einer MaxSMT Lösung für jeden Timing Constraint Typ separat und anschließend für eine Kombination aller Timing Constraint Typen gemessen. Auf diese Weise soll für eine vorgegebene Menge an Anforderungen sichergestellt werden, dass die Laufzeiten für alle Timing Constraint Typen in praktischen Anwendungsfällen angemessen sind. Für die Messungen wurden Timing Constraints generiert. Offset Timing Constraints wurden so generiert, dass diese einen minimalen und maximalen Offset zwischen 1ms und 10ms beinhalten. Für die Execution Order Constraints wurden jeweils 5 Runnables aus der Softwarearchitektur ausgewählt und eine beliebige Execution Order gewählt und für jeden Synchronization Timing Constraint wurden 3 Timing Events mit einem zufälligen Toleranzwert gewählt. Die Latency Timing Constraints bestehen aus einer zufällig generierten Timing Event Chain bestehend aus 5 Timing Events, sowie zufälligen Minimal- und Maximalwerten. Eine Übersicht über alle generierten Testszenarien findet sich in Tabelle 7.2. Die Testszenarien 1-10 verwenden nur eine kleine Menge von jeweils 10 Timing Constraints, weil ansonsten die Laufzeit für die Timing Verifikation zu groß ist. Da die Laufzeit ebenfalls für das Berechnen einer MaxSMT Lösung mit betrachtet werden soll, sind die generierten Anforderungsmengen so gewählt, dass diese unerfüllbar sind.

Um die Laufzeiten der Konsistenzanalyse mit Laufzeiten der Timing Verifikation zu vergleichen, wurden mehrere Softwarearchitekturen erstellt, auf deren Basis die Verifikationslaufzeiten gemessen werden. Das Modell M_1 ist sehr klein und beinhaltet nur eine begrenzte Anzahl an Softwarekomponenten, Runnables und Tasks. Das Modell M_2 dagegen ist komplexer. Tabelle 7.1 zeigt eine Übersicht über die Komplexität der Modelle. Da die Anforderungsmenge so gewählt ist, dass sie inkonsistent ist, schlägt die

Timing Verifikation für mindestens eine Timing Anforderung fehl.

Zusätzlich zum vollständigen Vergleich der Laufzeiten beider Methoden mit einer geringen Anzahl an Timing Anforderungen, wird in den Testszenarien 11-18 die Konsistenzanalyse mit einer größeren Menge an Anforderungen durchgeführt, wobei für Szenarien 11-14 eine große inkonsistente Anforderungsmenge generiert wird und für die Szenarien 15-18 eine konsistente Menge. Auf diese Weise lässt sich herausfinden inwieweit die Konsistenz der Anforderungsmenge Einfluss auf die Laufzeit hat. In diesen Szenarien können jedoch keine Messungen für die Timing Verifikation durchgeführt werden, da diese für alle Anforderungen zu groß wäre. Zusätzlich werden in den Testszenarien 12-14 erfüllbare Anforderungsmengen generiert.

7.3.2 Ergebnisse

Tabelle 7.3 zeigt die Laufzeitmessungen für die einzelnen Berechnungsschritte für die Timing Anforderungen und für die Modelle $\mathcal{M}_1$ und $\mathcal{M}_2$. Die Tabelle zeigt, dass die Transformation der AUTOSAR Elemente nach Z3-SMT lange dauert ($T(t)$), und damit einen großen Anteil der Laufzeit der Konsistenzanalyse hat. Dies liegt teilweise daran, dass auf die AUTOSAR Elemente über das Automatisierungsinterface von SystemDesk zugegriffen wird, welches aus Technologiegründen weniger effizient ist als ein direkter Zugriff. Des Weiteren wurden noch keine Optimierungen des Transformationsalgorithmus vorgenommen. Nichtsdestotrotz ist die Laufzeit im Vergleich zur Timing Verifikation akzeptabel. Die Messungen für $T(t)$, $T(smt)$ und $T(msat)$ sind ähnlich für alle Testszenarien mit den gleichen Timing Anforderungen, auch auf verschiedenen Architekturmodellen

Tabelle 7.1: Verwendete Modelle zur Laufzeitmessung

	$\mathcal{M}_1$	$\mathcal{M}_2$
Software Components	5	8
Runnable Entities	10	80
Tasks	5	8

Tabelle 7.2: Testszenarien für die Laufzeitverifikation

Id	$\mathcal{M}$	$\lvert\mathcal{R}_{otc}\rvert$	$\lvert\mathcal{R}_{eoc}\rvert$	$\lvert\mathcal{R}_{stc}\rvert$	$\lvert\mathcal{R}_{ltc}\rvert$	SAT
1	$\mathcal{M}_1$	10	0	0	0	unsat
2	$\mathcal{M}_1$	0	10	0	0	unsat
3	$\mathcal{M}_1$	0	0	10	0	unsat
4	$\mathcal{M}_1$	0	0	0	10	unsat
5	$\mathcal{M}_1$	10	10	10	10	unsat
6	$\mathcal{M}_2$	10	0	0	0	unsat
7	$\mathcal{M}_2$	0	10	0	0	unsat
8	$\mathcal{M}_2$	0	0	10	0	unsat
9	$\mathcal{M}_2$	0	0	0	10	unsat
10	$\mathcal{M}_2$	10	10	10	10	unsat
11	$\mathcal{M}_2$	100	0	0	0	unsat
12	$\mathcal{M}_2$	0	100	0	0	unsat
13	$\mathcal{M}_2$	0	0	100	0	unsat
14	$\mathcal{M}_2$	0	0	0	100	unsat
15	$\mathcal{M}_2$	100	0	0	0	sat
16	$\mathcal{M}_2$	0	100	0	0	sat
17	$\mathcal{M}_2$	0	0	100	0	sat
18	$\mathcal{M}_2$	0	0	0	100	sat

(beispielsweise Testszenario 1 und 6). Dies ist offensichtlich, da die Konsistenzanalyse unabhängig von der verwendeten Softwarearchitektur ist. Die Modelltransformation nach Timed Automata und die Verifikationslaufzeit ist offensichtlich sehr stark abhängig von der Größe und Komplexität der gesamten AUTOSAR Softwarearchitektur. So benötigen die Transformation und Verifikation der Offset Timing Constraints (Szenario 1), Execution Order Constraints (Szenario 2) und Synchronization Timing Constraints (Szenario 3) auf $\mathcal{M}_1$ mit maximal 17,2 bzw. 8 Sekunden sehr wenig Zeit. Dem gegenüber wird für die Transformation und Verifikation einer äquivalenten Anforderungsmenge auf $\mathcal{M}_2$ wesentlich mehr Zeit benötigt. So benötigen die Testszenarien 6-8 ungefähr bereits mindestens eine Minute für die Transformation, während die Verifikation der Execution Order Constraints (Szenario 7) bereits mehr als 10 Minuten und die Verifikaiton der Synchronization Timing Constraints (Szenario 8) mehr als 20 Minuten

Tabelle 7.3: Laufzeiten für die Transformation der Anforderungen nach SMT, SMT-Solving, MaxSMT Berechnung, Transformation nach Timed Automata und Verifikation mit UPPAAL

Id	$T(t)$	$T(smt)$	$T(msat)$	$T(tv)$	$T(v)$	*ratio*
1	5,4	0.06	0.04	13	3.0	34.4
2	10.9	0.02	0.07	17.2	5.0	49.5
3	8.1	0.04	0.04	15.6	8.0	34.6
4	6.5	0.02	0.04	18.1	208.1	2.9
5	27.1	0.1	0.13	28.2	219.0	11.1
6	5.4	0.06	0.04	67.5	88.9	3.5
7	10.9	0.02	0.07	65.7	864.4	1.2
8	8.1	0.04	0.04	72.9	1229	0.6
9	22.7	0.04	0.04	73,3	862.7	2.6
10	27.1	0.02	0.9	82	3031	0.9
11	31.9	0.1	1.3	80	-	-
12	252	0.42	209	571	-	-
13	52.7	0.56	339	328	-	-
14	77,2	0.4	2.4	108,8	-	-
15	33.3	0.1	0.4	81	-	-
16	219	0.39	154	567	-	-
17	57.1	0.54	335	331	-	-
18	80,0	0.1	0.4	109.8	-	-

benötigt. Lediglich die Verifikation der Latency Timing Constraints ist bereits auf dem kleineren Modell $\mathcal{M}_1$ (Szenario 4) mit 208 Sekunden sehr zeitaufwändig. Die Analyse kann allerdings auch auf dem großen Modell $\mathcal{M}_2$ (Szenario 9) mit 862 Sekunden noch in akzeptabler Zeit durchgeführt werden. Die Laufzeitunterschiede der Timing Anforderungen auf den beiden Modellen $\mathcal{M}_1$ und $\mathcal{M}_2$ lassen sich ebenfalls in den Diagrammen 7.2 und 7.3 erkennen.

Schließlich wurden zusätzlich die Laufzeiten für das Blinkerbeispiel aus Kapitel 2 gemessen. Für die Konsistenzanalyse, Transformation und SMT-Solving werden 4,5 bzw. 0.1 Sekunden benötigt, während die Verifikation nur 1,4 Sekunden benötigt.

Die Ergebnisse der Testszenarien 11 bis 18 zeigen, wie unser Ansatz mit mehr Timing Anforderungen skaliert. In jedem Test wurden die Timing An-

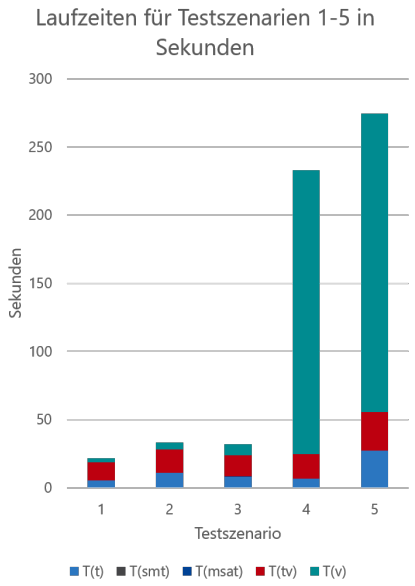

Abbildung 7.2: Laufzeiten der Testszenarien mit $\mathcal{M}_1$ (Szenarien 1-5)

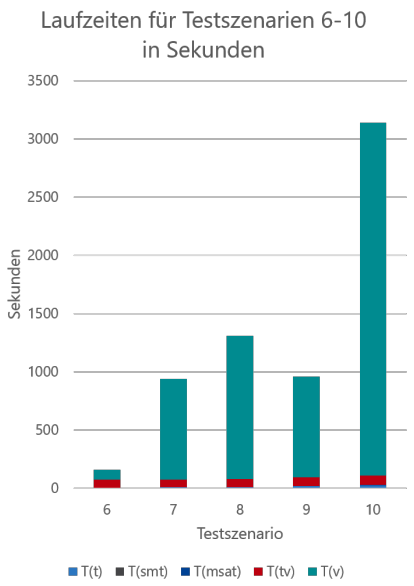

Abbildung 7.3: Laufzeiten der Testszenarien mit $\mathcal{M}_2$ (Szenarien 6-10)

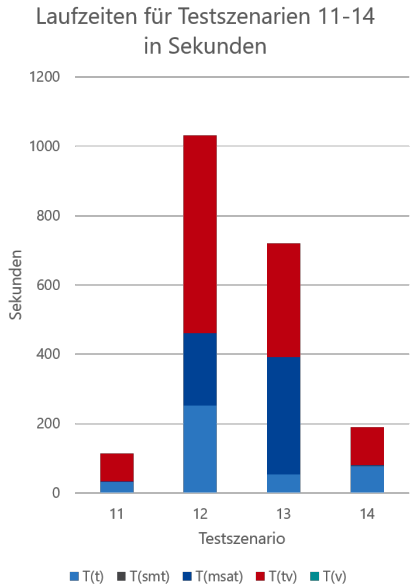

Abbildung 7.4: Laufzeiten der Testszenarien 11 - 14

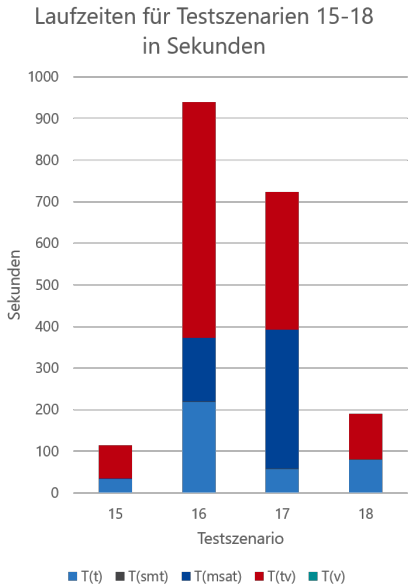

Abbildung 7.5: Laufzeiten der Testszenarien 15 - 18

forderungen von 10 auf 100 erhöht und dabei sowohl inkonsistente (Szenarien 11-14, Abbildung 7.4) als auch konsistente Anforderungsmengen (Szenarien 15-18, Abbildung 7.5) erzeugt. Die Ergebnisse zeigen, dass das Lösen der SMT-Formeln mit Z3 noch immer akzeptabel ist, während jedoch das Finden einer MaxSMT-Lösung wesentlich langsamer für Execution Order Constraints mit 209 und 154 Sekunden (Szenarien 12 und 14) und Synchronization Timing Constraints mit 339 und 335 Sekunden (Szenarien 13 und 17) wird. Aufgrund der hohen Laufzeiten für die Timing Verifikation, haben wir an dieser Stelle auf die Laufzeitmessung für die Timing Verifikation verzichtet. Aufgrund der Tatsache, dass wir jeden Timing Constraint als Testautomaten separat in die Softwarearchitektur integrieren und verifizieren, gehen wir davon aus, dass die Laufzeit linear mit der Anzahl der Timing Constraints steigt. Deswegen würde sich geschätzt hieraus beispielsweise für die Verifikation von 100 Synchronization Timing Constrains (Szenario 13) eine Laufzeit von ca. 8,5 Stunden ergeben.

Ein wesentliches Ergebnis der Laufzeitanalyse ist das Laufzeitverhältnis von Konsistenzanalyse und Timing Verifikation. Während in den Testszenarien 1,2 und 3, in denen ein kleineres Modell verwendet wurde, ungefähr 30% bis 50% der Laufzeit für die Konsistenzanalyse benötigt wurde, sind es in den Testszenarien 6 bis 10 lediglich 0,6% bis 3,5%. Dies zeigt, dass es bei komplexeren Softwarearchitekturen sinnvoll ist, Anforderungen vorab auf Konsistenz zu prüfen. Hingegen kann bei kleineren Softwarearchitekturen auf die Konsistenzanalyse verzichtet werden, da sie auch bei fehlerhaften Anforderungsmengen nur wenig Laufzeit einspart. Nichtsdestotrotz halten wir die Durchführung einer Konsistenzanalyse vor der eigentlichen Timing Verifikation für sinnvoll, da wir davon ausgehen, dass die meisten Modelle in der Praxis sehr groß sind und dass auch die Menge der Anforderungen wesentlich größer ist. Dieses Ergebnis stützt daher den im Konzept in Kapitel 3 vorgestellten Prozess.

Schließlich ist die Effizienz des Gesamtprozesses auch abhängig davon wie schwierig es ist die Timing Anforderungen zu modellieren und wie gut die Kompetenzen des Requirements Engineers sind, da dies wesentliche Einflussfaktoren für die Erzeugung fehlerhafter Anforderungsmengen sind. Wenn die Wahrscheinlichkeit gering ist, dass die erzeugten Anforde-

rungen inkonsistent sind, dann kann es günstiger sein auf die Konsistenzprüfung zu verzichten, da es in jedem Fall die Laufzeit des Gesamtprozesses verlängert.

7.4 Zusammenfassung

In diesem Kapitel wurden zum einen in Abschnitt 7.1 relevante technische Merkmale des Realisierungsansatzes vorgestellt. Insbesondere wurde die Softwarearchitektur vorgestellt, sowie Eigenschaften der Transformation für die AUTOSAR Modelle. Zum Anderen wurde in Abschnitt 7.3 eine detaillierte Evaluierung des Ansatzes hinsichtlich der Laufzeiten der einzelnen Bestandteile durchgeführt. Es konnte gezeigt werden, dass der Ansatz für die evaluierten Beispielmodelle praktisch anwendbar ist und die Konsistenzanalyse einen signifikanten Mehrwert für den Gesamtprozess darstellt.

8 Zusammenfassung und Ausblick

In der vorliegenden Arbeit wurde eine Methode zur Timing Verifikation von AUTOSAR Softwarearchitekturen vorgestellt. In diesem Kapitel wird das Konzept zunächst in Abschnitt 8.1 zusammengefasst. Abschnitt 8.2 diskutiert die wesentlichen Erkenntnisse und Entwurfsentscheidungen, indem diese hinsichtlich den erzielten Ergebnissen, Aufwand und Erweiterbarkeit mit alternativen Lösungsansätzen verglichen werden. Schließlich werden weiterführende Fragestellungen und Lösungsansätze vorgestellt, die in Abschnitt 8.3 einen Ausblick über zukünftige Entwicklungen in diesem Themengebiet geben.

8.1 Zusammenfassung

In dieser Arbeit wurde eine Methode zur Timing Verifikation von AUTOSAR Softwarearchitekturen vorgestellt und evaluiert. Das Ziel der Methode ist es durch das Erkennen von Fehlern in AUTOSAR Timing Anforderungen und in der AUTOSAR Softwarearchitektur frühzeitig

eine hohe Qualität der genannten Artefakte zu erzielen und damit den Entwicklungsprozess von Steuergeräten zu beschleunigen (siehe Kapitel 1). Dafür wurden zunächst in Kapitel 2 die Grundlagen der automotiven Steuergeräteentwicklung, die Bedeutung von Timing Anforderungen und deren Manifestation auf verschiedenen Abstraktionsebenen in unterschiedlichen Modellierungssprachen, sowie die zur Analyse angewendeten Techniken besprochen. Ebenfalls wurde ein Beispielmodell vorgestellt, das im weiteren Verlauf zur Demonstration des vorgestellten Ansatzes herangezogen wurde.

In Kapitel 3 wurde dann der Ansatz der kombinierten Konsistenzprüfung und Timing Verifikation vorgestellt (siehe Abschnitt 3.1) und es wurden Möglichkeiten diskutiert, wie sich der Ansatz in den bestehenden Entwicklungsprozess eingliedern lässt (siehe Abschnitt 3.3). Ebenfalls wurde an dieser Stelle das formale Modell von AUTOSAR vorgestellt, das sowohl für die Konsistenzanalyse als auch für die Timing Verifikation verwendet wird (siehe Abschnitt 3.2).

Der erste Schritt des Ansatzes (siehe auch Abbildung 3.1) besteht aus der Konsistenzprüfung der AUTOSAR Timing Anforderungen. Dieser wurde in Kapitel 4 vorgestellt. Dafür werden die AUTOSAR Timing Anforderungen in SMT-Formeln transformiert und mit dem SMT-Solver Z3 geprüft (siehe Abschnitt 4.1). Werden Anforderungsmengen als inkonsistent identifiziert können in einem zweiten Schritt Verfahren angewendet werden, mit denen die Inkonsistenzen aufgelöst werden können, indem für die Timing Anforderungen der Unsatisfiable Core und Maximum Satisfiability berechnet und visualisiert werden. Dieser Schritt wurde in Abschnitt 4.2 vorgestellt. Die Methode ermöglicht es schon eine frühzeitige Rückmeldung über die Qualität der Timing Anforderungen zu erhalten und diese zu verbessern. Somit erfüllt der Ansatz die in Kapitel 1 festgelegte *Anforderung 3*.

Anforderung 3

In einem dritten Schritt kann dann die Anforderungsmenge auf der Basis der Hinweise aus dem vorherigen Schritt überarbeitet werden. Schließlich kann die konsistente Anforderungsmenge dann im vierten Schritt verifiziert werden (siehe Kapitel 5). Dafür wird das AUTOSAR Softwarearchitekturmodell (siehe Abschnitt 5.1) , sowie die darin enthaltenen AUTOSAR Timing Anforderungen (siehe Abschnitt 5.2) nach Timed Automata transfor-

miert und im Anschluss mit UPPAAL verifiziert. Die Verifikation mit UPPAAL exploriert vollständig den Zustandsraum des Modells. Somit werden auch alle Randfälle bei der Timing Verifikation betrachtet, womit der Ansatz die anfangs festgelegte *Anforderung 1* erfüllt. Ebenfalls wird für die Verifikation ausschließlich das AUTOSAR-Modell benötigt, sodass *Anforderung 2* ebenfalls erfüllt ist.

Anforderung 1

Anforderung 2

Der Ansatz wurde dann anhand eines praktischen Anwendungsfalls in Kapitel 6 evaluiert. Dieser Anwendungsfall besteht aus einem realitätsnahen Modell mit einer größeren Menge an Timing Anforderungen. Das Modell, sowie die Anforderungen wurden zunächst vorgestellt (siehe Abschnitt 6.1) und im Anschluss die Evaluierungsergebnisse präsentiert (siehe Abschnitt 6.2). Die Ergebnisse des Anwendungsfalls zeigen, dass bei hinreichend großen Anforderungsmengen eine Kombination aus Konsistenzanalyse und Timing Verifikation die Effizienz der Verifikation gesteigert werden kann. Diese Erkenntnis wurde durch weitere Evaluierungen anhand verschiedener synthetischer Testszenarien in Kapitel 7 bestätigt (siehe Abschnitt 7.3). Darüber hinaus wurde in diesem Kapitel eine Übersicht über die realisierte prototypische Werkzeugunterstützung gegeben (siehe Abschnitt 7.1).

8.2 Diskussion

Dieser Abschnitt diskutiert und bewertet die wesentlichen Entwurfsentscheidungen, die aufgrund der gesteckten Rahmenbedingungen und der im Laufe der Arbeit gewonnenen Erkenntnisse gemacht worden sind.

8.2.1 Einfluss der Automobilindustrie

Diese Dissertation ist das Ergebnis eines industriell geprägten Promotionsvorhabens. Alle erarbeiteten Methoden lehnen sich daher an den konkreten Vorgaben der Automobilindustrie und Werkzeughersteller an. Dies manifestiert sich beispielsweise in der Anforderung den bereits existierenden AUTOSAR-Standard zu verwenden, sowie die Methoden für existierende Werkzeuge zu realisieren. Im Folgenden werden die Aspekte vorgestellt,

die wesentlich von den existierenden Vorgaben getrieben waren, sowie die daraus resultierenden Konsequenzen:

- Die in dieser Arbeit vorgestellte Methode zur Timing Analyse ist in erster Linie für die Analyse von AUTOSAR Softwarearchitekturen entworfen, da dies der vorherrschende Standard in der Automobilindustrie ist. Neben dem Metamodell zur Beschreibung der Timing Anforderungen, ist es insbesondere das Metamodell zur hardwareunabhängigen komponentenbasierten Architekturmodellierung, was in dieser Arbeit verwendet wird. Diese Modellstrukturen lassen sich in ähnlicher Form auch in anderen Domänenspezifischen Sprachen wiederfinden. Dies betrifft beispielsweise die Architekturmodellierungssprache *Architecture Analysis & Design Language* (AS-2C Architecture Analysis and Design Language, 2017), die überwiegend in der Entwicklung von Systemen der Luftfahrt eingesetzt wird oder *EAST-ADL* (EAST-ADL Association, 2013), die in der Systementwicklung eingesetzten Sprache. Daher gehen wir davon aus, dass insbesondere für das Konzept zur Timing Verifikation von AUTOSAR eine *Übertragbarkeit* auch auf andere Domänenspezifische Sprachen gegeben ist, solange sie die Konzepte von komponentenbasierten Softwarearchitekturen, die Verteilung verschiedener Softwaremodule auf ausführbare Tasks und deren Verteilung auf Hardware, sowie die Spezifikation von Timing Anforderungen unterstützen.

Architecture Analysis & Design Language

Übertragbarkeit

- Aufgrund der Verwendung des AUTOSAR Standards können eine Vielzahl der Automobilhersteller und Zulieferer ihre existierende Modelle ohne die Notwendigkeit zusätzlicher Transformationen für den vorgestellten Ansatz nutzen. Dies steigert die *praktische Relevanz* des Ansatzes erheblich. Darüber hinaus ist zusätzlich durch die Realisierung der Methoden innerhalb der bereits existierenden Werkzeuglandschaft sichergestellt, dass sich Modellierungs- und Qualitätssicherungsaktivitäten, die im V-Modell vor- bzw. nachgelagert sind, nahtlos integrieren lassen. So kann beispielsweise ein AUTOSAR-Modell nach der Timing Analyse direkt aus dem Werkzeug heraus für die Simulation kompiliert und für simulative

praktische Relevanz

Testfahrten verwendet werden.

- Für die Beschreibung der für die Transformation nach Timed Automata und SMT relevanten Modellelemente wurde in Kapitel 3 ein formales Modell in Definition 14 spezifiziert. Dies vereinfachte die Definition der Transformationen, da es ausschließlich die für das Zeitverhalten von AUTOSAR relevanten Metamodellelemente beinhaltet. Das formale Modell, sowie die darauf spezifizierten Transformationen nach SMT und Timed Automata realisieren Syntax- und Semantikdefinitionen für AUTOSAR, die zusätzlich neben der textuellen Semantik der AUTOSAR Spezifikation und der durch den generierten Quellcode eines AUTOSAR-Codegenerators realisierten Semantik stehen. Dabei bilden alle Semantiken unterschiedliche Teilaspekte von AUTOSAR ab:

  - Die textuelle Semantik beschreibt das Verhalten von AUTOSAR in einem für Anwender des AUTOSAR-Standards verständlichen, aber mehrdeutigen und vor allem größtenteils informellen Format. So wird ebenfalls das zeitliche Verhalten des Steuergerätes, sowie die Timing Anforderungen textuell spezifiziert.
  - AUTOSAR Codegeneratoren erzeugen zusammen mit dem in den einzelnen Runnables definierten Code ausführbaren Quellcode für eine konkrete Hardwareplattform. Je nach Zielplattform (Simulation als virtuelles Steuergerät oder Implementierung in Hardware) bildet diese Semantik das vollständige Verhalten des Steuergeräts ab und beschreibt auch die exakten Ausführungszeiten.
  - Die in dieser Arbeit spezifizierte Semantik mittels Timed Automata basiert auf dem formalen Modell aus Definition 14 und definiert ein hardwareunabhängiges und zeitbehaftetes Verhalten von AUTOSAR mittels Timed Automata, indem das Verhalten der im Standard spezifizierten Timing Anforderungen im Kontext der AUTOSAR-Steuergerätearchitektur definiert wird ohne dabei zu detailliert hardwarespezifische Aspekte zu betrachten.

– Die SMT-Semantik basiert wiederum ausschließlich auf den AUTOSAR Timing Extensions und formalisiert die zeitlichen Abhängigkeiten von Timing Events im Kontext der Anforderungen mithilfe von logischen Formeln.

Abbildung 8.1 zeigt die Modelle, sowie deren Beziehungen untereinander. Insgesamt ergeben sich aufgrund der Einführung eines weiteren formalen Modells, sowie der Semantik über Timed Automata und SMT für AUTOSAR folgende Problemstellungen:

1. **Korrektheit der Semantiken.** Da für AUTOSAR keine formale Semantik definiert ist, können wir die Korrektheit der Transformationen in Kapitel 5 nicht beweisen und somit ebenfalls die semantische Äquivalenz des Timed Automata Netzwerks und der SMT-Formel zur AUTOSAR Semantik nicht sicherstellen (siehe auch Abbildung 8.1 (1)). Dieses Problem trifft ebenfalls auf den generierten Quellcode (siehe auch Abbildung 8.1 (2)) zu und kann beispielsweise dadurch gelöst werden, indem neben der Syntax ebenfalls das Verhalten im AUTOSAR Standard formal festgehalten wird, sodass hierauf aufbauend die Verhaltensäquivalenz von Standard und Timed Automata Netzwerk bewiesen werden kann. Ein anderer Lösungsansatz wäre bei der Beschreibung der Semantik unseres Ansatzes die Semantik des generierten Quellcodes als Grundlage zu nehmen. So ließen sich beispielsweise die funktionalen Aspekte des Quellcodes auf die Generierung des Timed Automata Netzwerks abbilden und es würden ausschließlich die spezifizierten Timings der Runnables aus der Spezifikation genommen werden. Durch diesen Ansatz ließe sich das Problem der mangelnden Formalisierung des Standards abschwächen. Umgekehrt ließe sich auch ein Grundgerüst des Quellcodes auf Basis des Timed Automata Modells erzeugen, welches dann um fehlende Teile aus dem AUTOSAR Metamodell ergänzt würde (siehe auch Abbildung 8.1 (3)). Durch die Spezifikation zweier unterschiedlicher Semantiken für die

jeweils unterschiedlichen Teilaufgaben der Konsistenzprüfung und Timing Verifikation entsteht weiterhin das Problem, dass Aussagen die auf der Basis einer Semantik gemacht wurden, nicht auch gleichzeitig für die andere Semantik bewiesen sind. Letztendlich bilden jedoch die Anforderungs-Timed-Automata durch die verwendeten Signale in den Transitionen genau die gleiche Restriktionen wie die SMT-Formeln mithilfe der Variablen hinsichtlich des möglichen zeitlichen Auftretens der Timing Events ab, sodass wenn eine SMT-Formel unerfüllbar ist, ebenfalls mindestens zwei Anforderungs-Timed-Automata untereinander inkompatible Restriktionen beschreiben, sodass mindestens ein Anforderungs-Timed-Automaton nicht den Anforderungen des Systemmodells genügt, wobei das verwendete Systemmodell irrelevant ist.

2. **Konsistenz zwischen Modellen.** Ein weiteres Problem bei der Verwendung verschiedener Modelle mit sich überschneidenden Elementen ist es, dass Änderungen in einem Modell nicht automatisiert in die anderen Modelle übertragen werden. Hierdurch kann es passieren, dass Inkonsistenzen zwischen den verschiedenen Modellen auftreten, sodass Aussagen, die auf der Basis eines Modells gemacht werden, nicht mehr für andere Modelle gelten, da beispielsweise Teilmodelle nicht aktualisiert wurden. Dieses Problem existierte bereits vorher, wird aber durch das Einführen einer weiteren Syntax und Semantik weiter verschärft, denn sobald das AUTOSAR Modell geändert wird, müssen sowohl die Codegenerierung als auch die Generierung des formalen Modells erneut erfolgen. Ein möglicher Lösungsansatz hierfür ist es zwischen den Modellen anstelle imperativer Transformationsregeln bidirektionale Transformationen (BX) zu spezifizieren, die diese Konsistenz sicherstellen können (siehe auch Abschnitt 7.2). Dafür ist es notwendig, sowohl für das formale Modell in Definition 14 als auch für den ausführbaren Code zunächst Metamodelle zu spezifizieren, um diese dann mittels BX zu verknüpfen.

3. **Aussagekraft der Ergebnisse.** Aufgrund der fehlenden formalen Spezifikation des AUTOSAR Standards und der nicht vorhandenen semantischen Relation der Analysemodelle sowohl zur Codesemantik als auch zum AUTOSAR-Standard lassen sich Aussagen hinsichtlich der Korrektheit von Zeiteigenschaften nicht formal auf die anderen Modelle übertragen, was insbesondere bei der Übertragbarkeit der Analyseergebnisse auf die Semantik des finalen Steuergerätes problematisch ist. Diese Problematik betrifft jedoch alle Ansätze, die AUTOSAR als Grundlage für Analysen verwenden. Zusätzlich ist es aufgrund des fehlenden Hardwarebezugs unseres Modells nur auf der Basis von Laufzeitschätzungen für die Runnables möglich die finale Semantik des Steuergeräts anzunähern. Durch die Evaluierung der Transformationen im Werkzeug und Anwendung innerhalb eines größeren Anwendungsfalls konnten wir jedoch sicherstellen, dass die Ergebnisse der Analysemodelle mit erwarteten Ergebnissen übereinstimmten. Zukünftig können jedoch weitere Arbeiten aus Punkt 1 und 2 dazu beitragen, die Semantik der Modelle näher zusammenzubringen und somit die Aussagekraft der Ergebnisse zu verbessern.

- Die notwendige Bindung der Methoden an die existierenden Werkzeuge hat jedoch zu Herausforderungen geführt, die wesentliche *Entwurfsentscheidungen* beeinflusst haben. Die in Kapitel 4 und 5 vorgestellten Methoden haben beide als wesentlichen Kern die Transformation eines als Spezifikationsmodell verwendeten AUTOSAR-Modells in ein Analysemodell. Hier hätte es sich angeboten nach den Grundlagen aus Stahl und Völter (2006) existierende Technologien zur Modelltransformation anzuwenden. Allerdings war es technisch schwierig etablierte Technologien und Sprachen zur Metamodellierung oder zur Spezifikation und Ausführung von Modelltransformationen wie beispielsweise QVT, QVT Operational (Object Management Group, 2016) oder eMoflon::IBEX (Weidmann et al., 2019) ein-

Entwurfsentscheidungen

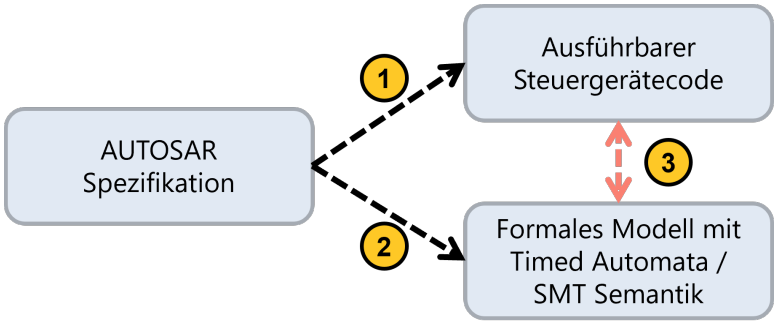

Abbildung 8.1: Darstellung der verschiedenen Modelle, deren Beziehungen untereinander und daraus resultierende Problemstellungen: Generierung von ausführbarem Steuergerätecode (1) aus AUTOSAR, dem formalen Modell aus AUTOSAR (2), sowie die Schwierigkeit die Semantiken zusammenzubringen, insbesondere die Semantik des formalen Modells und des ausführbaren Codes (3), um Verhaltensäquivalenz sicherzustellen.

zubinden, da diese ausschließlich in Form von Eclipse-Plugins* vorliegen und eng an die Datenstrukturen von Eclipse gekoppelt sind, während die vorhandenen Werkzeuge auf anderen Frameworks basieren. Eine Integration hätte die Implementierung aufwändiger gemacht und eventuell für einen Anwender dazu geführt, dass dieser mehrere Werkzeuge hätte nutzen müssen und zusätzliche manuelle Schritte zur Durchführung notwendig gewesen wären. Aus diesen Gründen wurden alle Modelltransformationen wie in Abschnitt 7.2 beschrieben imperativ implementiert.

8.2.2 Weitere Entwurfsentscheidungen

Eine große Herausforderung während der Erarbeitung dieser Dissertation war es, dass die gestellten Anforderungen an die wissenschaftliche Problemstellung und den Lösungsansatz zunächst sehr abstrakt waren, dafür aber bereits konkrete Restriktionen aufgrund der industriellen Anwendbarkeit bestanden. Der Startpunkt für die Arbeit war die Vision den aufwändigen und fehleranfälligen Modellierungsprozess von der technischen Systemarchitektur bis zu einer vollständig simulierbaren

*https://www.eclipse.org/

AUTOSAR-Softwarearchitektur in SystemDesk mithilfe statischer Analysen so zu unterstützen, dass Modellierungsfehler frühzeitig erkannt werden und somit das laufzeitintensive Kompilieren des Simulationsmodells möglichst selten durchgeführt werden muss. Dabei wurde identifiziert, dass die Timing-Eigenschaften von AUTOSAR nicht im Prozess berücksichtigt wurden und auch nicht in der Simulation Berücksichtigung fanden, was jedoch Voraussetzung für ein korrektes System ist. Daher entschieden wir uns diesen Aspekt weiter zu betrachten. Für die Analyse der Timing-Eigenschaften wurde inkrementell ein formales Modell von AUTOSAR entwickelt und es wurde daraufhin evaluiert, in wie weit sich das Zeitverhalten mit existierenden formalen Sprachen mit Zeitbezug nachbilden lässt. Wir entschieden uns für die Transformation nach Timed Automata, da wir hier auf eine gute Werkzeugunterstützung setzen konnten. Daraufhin wurde eine allgemeine Transformation spezifiziert und implementiert. Nachdem sich herausgestellt hatte, dass die Verifikation einiger Timing Anforderungen hohe Laufzeiten hatte und ebenfalls die Modellierung der Timing Anforderungen im Werkzeug aufgrund fehlender Modellierungshilfen fehleranfällig war, entschieden wir uns dazu die Timing Anforderungen genauer und unabhängig von der Softwarearchitektur zu betrachten und die Modellierung zu unterstützen. Die Ideen waren hier Qualitätskriterien für Timing Anforderungen bereits vor der Verifikation überprüfen zu können und domänenspezifische Sprachen für die Modellierung einzusetzen. Für Timing Anforderungen existierten aber schon eine Vielzahl von Ansätzen für domänenspezifische Sprachen, wie beispielsweise musterbasierte Sprachen für temporallogische Formeln (Dwyer et al., 1999, Konrad und Betty, 2005), grafische Modellierungssprachen für verschiedene spezifische Domänen wie beispielsweise *PPSL* für Business Prozesse (Förster et al., 2007) oder *zeitbasierte Story Szenario Diagramme* für Graphtransformationssysteme (Klein und Giese, 2006). Mit der *ARText Timing Language* existierte ebenfalls bereits eine XText basierte domänenspezifische Sprache für AUTOSAR Timing Anforderungen (Scheickl und Ainhauser, 2012). Daher richteten wir den Fokus auf Qualitätskriterien, die bei der Modellierung von Timing Anforderungen beachtet werden müssen. Als Basis dafür wurden die allgemein gültigen Kriterien

für Anforderungen der IEEE (IEEE, 1998) herangezogen und es wurden Kriterien identifiziert, die sich bezogen auf die Timing Anforderungen automatisiert bewerten lassen. Einige Eigenschaften wie Eindeutigkeit sind bereits durch das formale Metamodell von AUTOSAR gegeben. Ebenfalls wird die syntaktische Korrektheit durch das Metamodell von AUTOSAR bestimmt, was zusätzlich mithilfe der SystemDesk Validierungsregeln überprüft werden kann. Wir entschieden daher die Konsistenz der Anforderungen näher zu betrachten und modellierten zeitliche Abhängigkeiten von AUTOSAR Timing Anforderungen zunächst beispielhaft als SMTLIB Modell, um dann daraus eine vollständige Transformation herzuleiten. Die Entscheidung beruhte darauf, dass sich die temporalen Abhängigkeiten einfach als lineare Ungleichungen darstellen ließen, sodass diese Art der formalen Modellierung nahe lag. Prinzipiell wäre es ebenfalls möglich gewesen diese Eigenschaften über Timed Automata darzustellen.

8.3 Ausblick

Im Folgenden wird ein Ausblick auf zukünftige Themen, die sich dem hier vorgestellten Ansatz anschließen können, gegeben. Dabei wird ein Blick sowohl auf zukünftige vertiefende Arbeiten der einzelnen Beiträge als auch auf mögliche Forschungsfragen geworfen, die sich mit der Erweiterung des Gesamtprozesses beschäftigen.

Verfeinerung und Erweiterung des formalen Modells für AUTOSAR Das in Kapitel 3 vorgestellte formale Modell der Softwarearchitektur ist das Resultat eines anwendungsbezogenen Ansatzes zur Definition einer formalen Semantik für AUTOSAR. Der Ansatz bildet dabei das Timing-Verhalten der Softwarearchitektur sowohl auf Applikationsebene als auch auf RTE-Ebene und Basissoftwareebene ab. Der Fokus der Arbeit lag dabei auf der korrekten Nachbildung der Timings für die wesentlichen Basissoftwaremodule wie das Betriebssystem, da dies den größten Einfluss auf das Laufzeitverhalten hat. Darüber hinaus existieren weitere Basisssoftwaremodule verteilt auf mehreren horizontalen und vertikalen Ebenen (siehe auch

Abschnitt 2.1.2 und Abbildung 2.2). Für diese Module könnte es sich für eine detaillierte Analyse anbieten, diese ebenfalls als spezialisierte Timed Automata zu modellieren, um das Zeitverhalten weiter zu verfeinern. Des Weiteren ist AUTOSAR ein Standard, der über Jahre hinweg von einer großen Mitgliederanzahl sukzessive erweitert wurde und immer noch erweitert wird, was dazu führt, dass das existierende Meta-Modell mittlerweile sehr groß ist und sich ständig verändert. Die letzte große Erweiterung ist dabei die Einführung der sogenannten *Adaptive Platform*[†]. Diese ermöglicht als Kernkonzept die Definition von Services, die nur noch lose gekoppelt sind und dynamisch zur Laufzeit des Systems gestartet werden können. Dies macht die Software flexibler und einfacher adaptierbar, allerdings wird das Laufzeitverhalten durch die komplexere Architektur schwieriger vorhersehbar, wodurch Timing Analysen wesentlich komplexer werden. Wenn zukünftig auch mit Adaptive AUTOSAR sicherheitskritische Echtzeitanwendungen realisiert werden sollen, muss dieser Aspekt in zukünftigen Arbeiten betrachtet werden.

Adaptive Platform

Generierung laufzeitoptimierter AUTOSAR Architekturen Die Erweiterung der Konsistenzanalyse um Unterstützungsverfahren zur Identifikation von Fehlerursachen für inkonsistente Anforderungsmengen ist aus unserer Sicht ein vielversprechendes Instrument, um temporale Fehler in Timing Anforderungen schneller zu beheben und die praktische Anwendbarkeit der Methode zu steigern. Ein ähnlicher Ansatz für die anschließende Timing-Analyse wäre ebenfalls eine sinnvolle Ergänzung und ein Ansatz für zukünftige Arbeiten. Denn ist eine Timing Anforderung durch das System nicht erfüllbar, so stellt sich die Frage, welche möglichen Ursachen dies hat und welche Möglichkeiten zur Auflösung existieren. Ein einfaches Beispiel hierfür ist ein Latency Timing Constraint, der Events aus mehreren Runnables referenziert, die aufgrund eines falsch konfigurierten Task-Runnable-Mappings jedoch größere Latenzen verursachen als notwendig. Durch die Änderung der Reihenfolge der Runnables im Task kann das Modell dann der Anforderung genügen. Liegen die Timing An-

[†]https://www.autosar.org/standards/adaptive-platform/

forderungen und die AUTOSAR-Architektur auf Applikationsebene vor, so ließe sich das Task-Runnable-Mapping auch automatisiert erzeugen. Ein möglicher Lösungsansatz hierfür ist, dass sowohl Timing Anforderungen als auch die Architektur in ein SMT-Analysemodell integriert werden. Die Laufzeiten von Runnables und deren Datenverbindungen müssten dann als zusätzliche Constraints in das Modell aufgenommen werden. Auf der Basis einer erfüllbaren Variablenbelegung ließe sich dann ein Task-Runnable-Mapping, sowie die OSTasks synthetisieren. Ähnliche Ansätze, die laufzeitoptimierte Systeme erzeugen sind beispielsweise Zhu et al. (2012), Long et al. (2009) oder Wozniak (2013). Diese modellieren das Problem als lineares Programm oder nutzen genetische Algorithmen. Keine der Arbeiten berücksichtigt allerdings die Abhängigkeiten von Timing Anforderungen.

Automatisierte Extraktion und Verfeinerung von Timing Anforderungen aus Simulationen Die ersten Timing Anforderungen werden üblicherweise bereits auf Benutzerebene spezifiziert (siehe auch Abbildung 2.1). Diese werden beispielsweise aus Interviews, Workshops oder aus Änderungswünschen, sowie sonstigen Rückmeldungen der Benutzer aus dem Anwendungsfeld gewonnen und müssen ebenfalls technische und gesetzliche Vorgaben berücksichtigen (Schäuffele und Zurawka, 2010). Doch gerade für die Entwicklung autonomer Fahrzeugfunktionen ist es häufig schwierig konkrete Anforderungen an das Timing Verhalten festzuhalten. Neue Regularien wie beispielsweise zur Absicherung automatischer Spurhalteassistenzsysteme (United Nations, 2020) fordern, dass das System in allen Situationen Unfälle vermeidet, in denen dies auch ein geübter menschlicher Fahrer könnte. Für diese Anforderung ist es zunächst schwierig konkrete End-To-End Timings zu extrahieren, die dann im Rahmen des Entwicklungsprozesses verfeinert werden können. Eine Möglichkeit zur Erhebung konkreter Zeitschranken ist es eine Gesamtfahrzeugsimulation mit der prototypischen Fahrfunktion in unterschiedlichen Szenarien mit verschiedenen Worst-Case Timings durchzuführen und mit einem idealisierten Fahrermodell zu vergleichen. Sobald für eine bestimmte Laufzeit und mindestens einem Szenario für die Fahrfunktion

eine Unfallsituation vorliegt, beim idealisierten Fahrermodell aber nicht, kann diese Laufzeit als Grundlage für ein maximales End-To-End Timing genommen werden. Ähnliche Konzepte werden beispielsweise auch zur Erhebung von Anforderungen an die Genauigkeit von Sensorkomponenten autonomer Fahrzeuge verwendet (Philipp et al., 2021).

Automatisierte Formalisierung natürlichsprachlicher Timing Anforderungen Die entwickelte Methode zur Konsistenzanalyse der Timing Anforderungen realisiert eine analytische Herangehensweise um die Qualität von Anforderungsartefakten zu erhöhen. Ein weiterer Ansatz, um bereits frühzeitig die Qualität von Anforderungsartefakten zu gewährleisten wäre es ebenfalls konstruktive Ansätze zu betrachten. Diese unterstützen einen Requirements Engineer bereits bei der Transformation von Anforderungsdokumenten nach AUTOSAR. beispielsweise durch die automatische Extraktion von AUTOSAR Timing Anforderungen aus textuellen Anforderungsdokumenten mittels linguistischer Verfahren oder aus existierenden SysML Anforderungsmodellen. Hier existieren Ansätze für allgemeine Anforderungsartefakte, die auf formalisierten textuellen Dokumenten arbeiten wie Holtmann et al. (2011). Diese können als Grundlage verwendet werden.

8.4 Schlussbemerkung

Durch die Anwendung formaler Methoden zur Verifikation von Timing Anforderungen bereits auf AUTOSAR-Architekturebene können Timing Fehler in Anforderungsartefakten frühzeitig erkannt werden und so zu einer effizienteren und kostengünstigeren Entwicklung komplexer Steuergerätesoftware beitragen. Eine große Hürde in der industriellen Praxis ist dabei die einfache - d.h. auch ohne Expertenwissen mögliche - Anwendbarkeit, sowie die einfache Integration in bestehende Entwicklungsprozesse und Werkzeuge, die bisher noch nicht in allen Facetten verfügbar war. Der vorgestellte Ansatz überwindet diese Herausforderungen durch die Anwendung des etablierten AUTOSAR-Standards und durch die Bereitstellung

von Mechanismen zur effizienten Identifikation und Korrektur von inkonsistenten Anforderungsmengen. Im späteren Verlauf des Entwicklungsprozesses lässt sich der Ansatz sowohl mit code-basierten formalen Methoden zur Identifikation von Task-Laufzeiten als auch mit zeitabhängigen Simulationen kombinieren, was den praktischen Nutzen des Ansatzes weiter steigert.

Literaturverzeichnis

Islam Abdelhalim, Steven Schneider, und Helen Treharne. Towards a Practical Approach to Check UML/fUML Models Consistency Using CSP. In Shengchao Qin und Zongyan Qiu, Hrsg., *Proceedings of the 13th International Conference on Formal Engineering Methods, ICFEM 2011*, Lecture Notes in Computer Science, Seiten 33–48, Berlin, Heidelberg, 2011. Springer. ISBN 978-3-642-24558-9.

Karsten Albers, Frank Bodmann, und Frank Slomka. Advanced Hierarchical Event-Stream Model. In Zdenek Hanzalek, Hrsg., *Proceedings of the 20th Euromicro Conference on Real-Time Systems, ECRTS 2008*, Prag, 2008. IEEE Computer Society. ISBN 978-0-7695-3298-1.

R. Alur und T. A. Henzinger. A really temporal logic. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, Seiten 164–169. IEEE Computer Society, 1989. ISBN 0-8186-1982-1.

R. Alur, C. Courcoubetis, und D. Dill. Model-Checking in Dense Real-Time. *Information and Computation*, 104(1):2–34, 1993. ISSN 08905401.

Rajeev Alur und David Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

Charles André. *Syntax and Semantics of the Clock Constraint Specifcation Language (CCSL).* Dissertation, Inria Institute, Sophia Antipolis, 2009.

Anthony Anjorin, Thomas Buchmann, Bernhard Westfechtel, Zinovy Diskin, Hsiang-Shang Ko, Romina Eramo, Georg Hinkel, Leila Samimi-Dehkordi, und Albert Zündorf. Benchmarking bidirectional transformations: theory, implementation, application, and assessment. *Software & Systems Modeling*, 19(3):647–691, 2020a. ISSN 1619-1366.

Anthony Anjorin, Nils Weidmann, Robin Oppermann, Lars Fritsche, und Andy Schürr. Automating test schedule generation with domain-specific languages. In Eugene Syriani und Houari Sahraoui, Hrsg., *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, Seiten 320–331, New York, NY, USA, 2020b. ACM. ISBN 9781450370196.

Anthony Anjorin, Nils Weidmann, und Katharina Artic. Bidirectional Transformations in Practice: An Automotive Perspective on Traceability Maintenance (Short Paper). In Catherine Dubois und Julien Cohen, Hrsg., *STAF 2022 Workshop Proceedings: 10th International Workshop on Bidirectional Transformations (BX 2022), 2nd International Workshop on Foundations and Practice of Visual Modeling (FPVM 2022) and 2nd International Workshop on MDE for Smart IoT Systems (MeSS 2022) (co-located with Software Technologies: Applications and Foundations federation of conferences (STAF 2022)), Nantes, France, July 5-8, 2022*, volume 3250 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2022. URL http://ceur-ws.org/Vol-3250/bxpaper3.pdf.

Saoussen Anssi, Karsten Albers, Matthias Dörfel, und Sebastien Gerard. chronVAL/chronSIM: A Tool Suite for Timing Verification of Automotive Applications. In Joseph Sifakis, Hrsg., *Proceedings of the 6th European Congress on Embedded Real Time Software And Systems, ERTS 2012*, 2012.

Vincent Aravantinos, Sebastian Voss, Sabine Teufl, Florian Hölzl, und Bernhard Schätz. AutoFOCUS 3: Tooling Concepts for Seamless, Model-based Development of Embedded Systems Architecting of Cyber-physical and Embedded Systems and 1st International Workshop on UML Consistency Rules (ACES-MB 2015 & WUCOR 2015) co-located with ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2015), Ottawa, Canada, September 28, 2015. In *Proceedings of the 8th International Workshop on Model-based Architecting of Cyber-Physical and Embedded Systems*, Seiten 19–26, 2015. URL http://ceur-ws.org/Vol-1508/paper4.pdf.

AS-2C Architecture Analysis and Design Language. Architecture Analysis & Design Language (AADL), 2017. URL https://www.sae.org/standards/content/as5506c/.

AUTOSAR. Specification of Timing Extensions, 2019a. URL https://www.autosar.org/fileadmin/user_upload/standards/classic/19-11/AUTOSAR_TPS_TimingExtensions.pdf.

AUTOSAR. System Template, 2019b. URL https://www.autosar.org/fileadmin/user_upload/standards/classic/19-11/AUTOSAR_TPS_SystemTemplate.pdf.

AUTOSAR. Software Component Template, 2019c. URL https://www.autosar.org/fileadmin/user_upload/standards/classic/19-11/AUTOSAR_TPS_SoftwareComponentTemplate.pdf.

AUTOSAR. Recommended Methods and Practices for Timing Analysis and Design within the AUTOSAR Development Process, 2019d. URL https://www.autosar.org/fileadmin/user_upload/standards/classic/19-11/AUTOSAR_TR_TimingAnalysis.pdf.

AUTOSAR. Methodology, 2019e. URL https://www.autosar.org/fileadmin/user_upload/standards/classic/19-11/AUTOSAR_TR_Methodology.pdf.

AUTOSAR. AUTOSAR Virtual Functional Bus, 2019f. URL https://www.autosar.org/fileadmin/user_upload/standards/classic/19-11/AUTOSAR_EXP_VFB.pdf.

AUTOSAR. ECU Configuration, 2019g. URL https://www.autosar.org/fileadmin/user_upload/standards/classic/19-11/AUTOSAR_TPS_ECUConfiguration.pdf.

AUTOSAR. Generic Structure Template, 2019h. URL https://www.autosar.org/fileadmin/user_upload/standards/classic/19-11/AUTOSAR_TPS_GenericStructureTemplate.pdf.

Barrett, Clark und Tinelli, Cesare. Satisfiability Modulo Theories. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, und Roderick Bloem, Hrsg., *Handbook of Model Checking*, Seiten 305–343. Springer, Cham, 2018. ISBN 978-3-319-10575-8.

Barrett, Clark, Fontaine, Pascal, Stump, Aaron, und Tinelli, Cesare. The SMT-LIB Standard, 2017.

Andreas Bauer, Martin Leucker, und Jonathan Streit. SALT - Structured Assertion Language for Temporal Logic. In Zhiming Liu und Jifeng He, Hrsg., *Formal Methods and Software Engineering, ICFEM 2006*, Lecture Notes in Computer Science, Berlin, Heidelberg, 2006. Springer.

Kent Beck und Cynthia Andres. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Boston, MA, 2nd ed. Aufl., 2005. ISBN 9780321278654.

Kent Beck, Mike Beedle, Arie Van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Dave Thomas, und Jeff Sutherland. Manifesto for agile software development, 2001.

Jan Steffen Becker. Model Checking Amalthea with Spin. In Sebastian Götz, Lukas Linsbauer, Ina Schaefer, und Andreas Wortmann, Hrsg., *Proceedings of the Software Engineering 2021 Satellite Events, Braunschweig/Virtual, Germany, February 22 - 26, 2021*, volume 2814 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2021. URL http://ceur-ws.org/Vol-2814/paper-A3-1.pdf.

Steffen Becker, Stefan Dziwok, Christopher Gerking, Christian Heinzemann, Wilhelm Schäfer, Matthias Meyer, und Uwe Pohlmann. The MechatronicUML method: model-driven software engineering of self-adaptive mechatronic systems. In Pankaj Jalote, Lionel Briand, und André van der Hoek, Hrsg., *Companion Proceedings of the 36th International Conference on Software Engineering, ICSE Companion 2014*, Seiten 614–615, New York, New York, USA, 2014. ACM. ISBN 978-1-4503-2768-8.

Gerd Behrmann, Re David, und Kim G. Larsen. A Tutorial on UPPAAL. In Marco Bernardo und Flavio Corradini, Hrsg., *Formal Methods for the Design of Real-Time Systems*, Seiten 200–236. Springer, 2004.

Gábor Bergmann, István Dávid, Ábel Hegedüs, Ákos Horváth, István Ráth, Zoltán Ujhelyi, und Dániel Varró. Viatra 3: A Reactive Model Transformation Platform. In Dimitris Kolovos und Manuel Wimmer, Hrsg., *Theory and Practice of Model Transformations*, volume 9152 of *Lecture Notes in Computer Science*, Seiten 101–110. Springer International Publishing, Cham, 2015. ISBN 978-3-319-21154-1.

Steffen Beringer und Heike Wehrheim. Verification of AUTOSAR Software Architectures with Timed Automata. In Maurice H. ter Beek, Stefania Gnesi, und Alexander Knapp, Hrsg., *Proceedings of the Joint 21th International Workshop on Formal Methods for Industrial Critical Systems and 16th International Workshop on Automated Verification of Critical Systems, FMICS-AVoCS 2016*, Lecture Notes in Computer Science, Seiten 189–204, Berlin, Heidelberg, 2016. Springer. ISBN 978-3-319-45942-4.

Steffen Beringer und Heike Wehrheim. Consistency Analysis of AUTOSAR Timing Requirements. In Marten van Sinderen, Hans-Georg Fill, und Leszek A. Maciaszek, Hrsg., *Proceedings of the 15th International Conference on Software Technologies, ICSOFT 2020*, Seiten 15–26. SciTePress, 2020. ISBN 978-989-758-443-5.

Nikolaj Bjorner und Anh-Dung Phan. vZ - Maximal Satisfaction with Z3. In Temur Kutsia und Andrei Voronkov, Hrsg., *6th International Symposium*

on Symbolic Computation in Software Science 2014, EPiC Series in Computing, Seiten 1–9. EasyChair, 2014.

Barry W. Boehm. Guidelines for Verifying and Validating Software Requirements and Design Specifications. In P. A. Samet, Hrsg., *Proceedings of the European Conference on Applied Information Technology of the International Federation for Information Processing, Euro IFIP 1979*, Seiten 711–719. North Holland, 1979.

Barry W. Boehm. A Spiral Model of Software Development and Enhancement. *Computer*, 21(5):61–72, 1988. ISSN 0018-9162. URL https://doi.org/10.1109/2.59.

Frederick P. Brooks. *The mythical man-month: Essays on software engineering*. Addison-Wesley, Reading, Mass., 1975. ISBN 978-0-201-00650-6.

Manfred Broy. Challenges in automotive software engineering. In Leon J. Osterweil, Dieter Rombach, und Mary Lou Soffa, Hrsg., *Proceedings of the 28th international conference on Software engineering*, Seiten 33–42, New York, NY, USA, 2006. ACM. ISBN 1595933751.

Manfred Broy, Ingolf H. Kruger, Alexander Pretschner, und Christian Salzmann. Engineering Automotive Software. *Proceedings of the IEEE*, 95 (2):356–373, 2007.

Manfred Broy, Martin Feilkas, Markus Herrmannsdoerfer, Stefano Merenda, und Daniel Ratiu. Seamless Model-Based Development: From Isolated Tools to Integrated Model Engineering Environments. *Proceedings of the IEEE*, 98(4):526–545, 2010.

Bundesstelle für Informationstechnik. V-Modell XT, 2012. URL http://ftp.tu-clausthal.de/pub/institute/informatik/v-modell-xt/Releases/1.4/V-Modell-XT-Gesamt.pdf.

Giorgio C. Buttazzo. *Hard Real-Time Computing Systems*, volume 24. Springer, Boston, MA, 2011. ISBN 978-1-4614-0675-4.

Zhou Chaochen, C.A.R. Hoare, und Anders P. Ravn. A Calculus of Durations. *Information Processing Letters*, 40(5):269–276, 1991. ISSN 00200190.

Robert N. Charette. How Software Is Eating the Car. *IEEE Spectrum*, 2021, 2021. URL https://spectrum.ieee.org/software-eating-car.

Edmund M. Clarke und E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, Hrsg., *Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, Seiten 52–71, Berlin, Heidelberg, 1981. Springer. ISBN 3-540-11212-X.

Edmund M. Clarke und Jeannette M. Wing. Formal methods. *ACM Computing Surveys*, 28(4):626–643, 1996. ISSN 0360-0300.

Mike Cohn. *Agile Softwareentwicklung: Mit Scrum zum Erfolg!* Addison-Wesley, München and Boston, Mass. [u.a.], 2010. ISBN 978-3827329875.

Patrick Cousot und Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, und Ravi Sethi, Hrsg., *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '77*, Seiten 238–252, New York, New York, USA, 1977. ACM.

Leonardo de Moura und Nikolaj Bjørner. Z3: An Efficient SMT Solver. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, C. R. Ramakrishnan, und Jakob Rehof, Hrsg., *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, Seiten 337–340. Springer, Berlin, Heidelberg, 2008. ISBN 978-3-540-78799-0.

John Derrick, David Akehurst, und Eerke Boiten. A framework for UML consistency. In Ludwik Kuzniarz, Gianna Reggio, Jean-Louis Sourrouille, und Zbigniew Huzar, Hrsg., *UML 2002 Workshop on Consistency Problems in UML-based Software Development*, Lecture Notes in Computer Science, Seiten 182–196, Berlin, Heidelberg, 2002. Springer. ISBN 3-540-44254-5.

Henning Dierks. PLC-automata: A new class of implementable real-time automata. In Miquel Bertran und Teodor Rus, Hrsg., *Transformation-Based Reactive Systems Development*, volume 1231 of *Lecture Notes in Computer Science*, Seiten 111–125, Berlin, Heidelberg, 1997. Springer. ISBN 978-3-540-63010-4.

Edsger W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, 1972. ISSN 0001-0782.

Danielle Douglas und Michael A. Fletcher. Toyota reaches $1.2 billion settlement to end probe of accelerator problems. *Washington Post*,

2019, 2019. URL https://www.washingtonpost.com/business/economy/toyota-reaches-12-billion-settlement-to-end-criminal-probe/2014/03/19/5738a3c4-af69-11e3-9627-c6502id6d572_story.html.

dSPACE GmbH. Real-Time Testing Observer Library, 2020. URL https://www.dspace.com/de/gmb/home/products/sw/test_automation_software/rtt_observer_lib.cfm.

Matthew B. Dwyer, George S. Avrunin, und James C. Corbett. Patterns in property specifications for finite-state verification. In Barry Boehm, David Garlan, und Jeff Kramer, Hrsg., *Proceedings of the 21st ICSE 1999 International Conference on Software Engineering*. ACM, 1999.

Stefan Dziwok, Uwe Pohlmann, Goran Piskachev, David Schubert, Sebastian Thiele, und Christopher Gerking. The MechatronicUML Design Method: Process and Language for Platform-Independent Modeling, 2016.

EAST-ADL Association. EAST-ADL Domain Model Specification, 2013. URL http://www.east-adl.info/Specification/V2.1.12/EAST-ADL-Specification_V2.1.12.pdf.

Tobias Eckardt, Christian Heinzemann, Stefan Henkler, Martin Hirsch, Claudia Priesterjahn, und Wilhelm Schäfer. Modeling and verifying dynamic communication structures based on graph transformations. *Computer Science - Research and Development*, 28(1):3–22, 2013. ISSN 1865-2034.

Gregor Engels, Reiko Heckel, und Jochen Malte Küster. Rule-Based Specification of Behavioral Consistency Based on the UML Meta-model. In Martin Gogolla und Cris Kobryn, Hrsg., *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts and Tools, UML 2001*, volume 2185 of *Lecture Notes in Computer Science*, Seiten 272–286, Berlin, Heidelberg, 2001. Springer. ISBN 978-3-540-42667-7.

Gregor Engels, Jochen M. Küster, Reiko Heckel, und Luuk Groenewegen. Towards consistency-preserving model evolution. In Mikio Aoyama, Katsuro Inoue, und Vaclav Rajlich, Hrsg., *Proceedings of the international workshop on Principles of software evolution - IWPSE '02*, Seite 129, New York, New York, USA, 2002. ACM. ISBN 1581135459.

Gregor Engels, Baris Güldali, Christian Soltenborn, und Heike Wehrheim. Assuring Consistency of Business Process Models and Web Services Using Visual Contracts. In Andy Schürr, Manfred Nagl, und Albert Zündorf,

Hrsg., *Applications of Graph Transformations with Industrial Relevance*, volume 5088 of *Lecture Notes in Computer Science*, Seiten 17–31. Springer, Berlin, Heidelberg, 2008. ISBN 978-3-540-89019-5.

Nico Feiertag, Kai Richter, Johan Nordlander, und Jan Jonsson. A Compositional Framework for End-to-End Path Delay Calculation of Automotive Systems under Different Path Semantics. In *IEEE Real-Time Systems Symposium 2008*, volume 29, 2008. URL https://symtavision.com/downloads/Paper/Fei08_A_Compositional_Framework_for_End-to-End_Path_Delay_Calculation_RTSS08_Paper.pdf.

Christian Ferdinand und Reinhold Heckmann. aiT: Worst-Case Execution Time Prediction by Static Program Analysis. In Renè Jacquart, Hrsg., *Building the Information Society*, volume 156 of *IFIP International Federation for Information Processing*, Seiten 377–383. Springer, Boston, MA, 2004. ISBN 978-1-4020-8156-9.

Alexander Förster, Gregor Engels, Tim Schattkowsky, und Ragnhild Van der Straeten. Verification of Business Process Quality Constraints Based on Visual Process Patterns. In *1st IEEE Int. Symposium on Theoretical Aspects of Software Engineering*, 2007.

Daniel Galin. *Software quality assurance: From theory to implementation / Daniel Galin*. Pearson, Harlow, 2004. ISBN 978-0201709452.

Honghao Gao, Yida Zhang, Huaikou Miao, Ramón J. Durán Barroso, und Xiaoxian Yang. SDTIOA: Modeling the Timed Privacy Requirements of IoT Service Composition: A User Interaction Perspective for Automatic Transformation from BPEL to Timed Automata. *Mobile Networks and Applications*, 26(6):2272–2297, 2021. ISSN 1383-469X.

M. R. Garey, D. S. Johnson, und L. Stockmeyer. Some simplified NP-complete graph problems. *Theoretical Computer Science*, 1(3):237–267, 1976.

Matthias Gehrke, Petra Nawratil, Oliver Niggemann, Wilhelm Schäfer, und Martin Hirsch. Scenario-Based Verification of Automotive Software Systems. In Holger Giese, Bernhard Rumpe, und Bernhard Schätz, Hrsg., *Dagstuhl-Workshop MBEES*, Daghstuhl-Workshop MBEES, Seiten 35–42. TU Braunschweig, Institut für Software Systems Engineering, 2006.

Volker Gruhn und Ralf Laue. Patterns for Timed Property Specifications. In *Electronic Notes in Theoretical Computer Science*, volume 153, Seiten 117–133, 2006.

Raju Halder, Jose Proenca, Nuno Macedo, und Andre Santos. Formal Verification of ROS-Based Robotic Applications Using Timed-Automata. In *2017 IEEE/ACM 5th International FME Workshop on Formal Methods in Software Engineering (FormaliSE)*, Seiten 44–50. IEEE, 2017. ISBN 978-1-5386-0422-9.

Arne Hamann, Razvan Racu, und Rolf Ernst. Formal Methods for Automotive Platform Analysis and Optimization. In *Proc. Future Trends in Automotive Electronics and Tool Integration Workshop (DATE Conference)*, Munich, Germany, 2006.

Eric Hannon. An integrated perspective on the future of mobility, 2016. URL https://www.mckinsey.com/business-functions/sustainability/our-insights/an-integrated-perspective-on-the-future-of-mobility.

Martijn Hendriks und Marcel Verhoef. Timed Automata Based Analysis of Embedded System Architectures. In *20th IEEE International Parallel and Distributed Processing Symposium*, 2006.

Thomas A. Henzinger, Pei-Hsin Ho, und Howard Wong-Toi. HYTECH: a model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer*, 1(1-2):110–122, 1997. ISSN 1433-2779.

Paula Herber, Marcel Pockrandt, und Sabine Glesner. STATE - A SystemC to Timed Automata Transformation Engine. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, Seiten 1074–1077. IEEE, 2015. ISBN 978-1-4799-8937-9.

C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978. ISSN 0001-0782.

Jörg Holtmann, Jan Meyer, und Jan von Detten. Automatic Validation and Correction of Formalized, Textual Requirements. In *Proceedings of the IEEE 4th International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2011*, volume 4. IEEE Computer Society, 2011.

Jörg Holtmann, Ruslan Bernijazov, Matthias Meyer, David Schmelter, und Christian Tschirner. Integrated and iterative systems engineering and software requirements engineering for technical systems. *Journal of Software: Evolution and Process*, 28(9):722–743, 2016. ISSN 20477473.

IEEE. IEEE Standard Glossary of Software Engineering Terminology, 1990.

IEEE. IEEE 830-1998 Recommended Practice for Software Requirements Specifications, 1998.

ISO International Organisation for Standardisation. ISO 26262: Road Vehicles - Functional Safety, 2018. URL https://www.iso.org/standard/68383.html.

Jaco Jacobs und Andrew Simpson. On the formal interpretation and behavioural consistency checking of SysML blocks. *Software & Systems Modeling*, 16(4):1145–1178, 2017. ISSN 1619-1366.

Diana Kalibatiene, Olegas Vasilecas, und Ruta Dubauskaite. Rule Based Approach for Ensuring Consistency in Different UML Models. In Stanislaw Wrycza, Hrsg., *Information Systems 2013*, LNBIP, Seiten 1–16. Springer, 2013. ISBN 978-3-642-40854-0.

Sean Kane, Ellen Liberman, Tony DiViesti, und Melanie MacDonald. An Examination of the National Highway Traffic Safety Administration and the National Aeronautics and Space Administration Engineering Safety Center Assessment and Technical Evaluation of Toyota Electronic Throttle Control (ETC) Systems and Unintended Acceleration, 2011.

Robert M. Keller. Formal verification of parallel programs. *Communications of the ACM*, 19(7):371–384, 1976. ISSN 0001-0782.

Barbara Kempkes und Sven Walther. Testen ohne reale ECUs: Bausatz für virtuelle Steuergeräte: Absicherung per Software-in-the-Loop (SiL). *Automobil Elektronik*, 2018(03-04/2018):50–53, 2018. URL https://www.automobil-elektronik.de/wp-content/uploads/sites/7/2018/04/AEL_03_04_2018_Internet.pdf.

Jin Hyun Kim, Kim G. Larsen, Brian Nielsen, Marius Mikucionis, und Peter Olsen. Formal Analysis and Testing of Real-Time Automotive Systems Using UPPAAL Tools. In *Formal Methods for Industrial Critical Systems (FMICS) 2015*, volume 9128 of *Lecture Notes in Computer Science*, Seiten 47–61. Springer, 2015.

Olaf Kindel und Mario Friedrich. *Softwareentwicklung mit Autosar: Grundlagen, Engineering, Management in der Praxis*. dpunkt, Heidelberg, 2009. ISBN 978-3898645638.

Florian Klein und Holger Giese. *Integrated Visual Specification of Structural and Temporal Properties*. Dissertation, Paderborn, Paderborn, 2006.

Florian Klein und Holger Giese. Joint Structural and Temporal Property Specification Using Timed Story Scenario Diagrams. In *FASE 2007*, volume 4422 of *Lecture Notes in Computer Science*, Seiten 185–199. Springer, 2007.

Serge Klein, Rene Savelsberg, Feihong Xia, Daniel Guse, Jakob Andert, Torsten Blochwitz, Claudia Bellanger, Stefan Walter, Steffen Beringer, Janek Jochheim, und Nicolas Amringer. Engine in the Loop: Closed Loop Test Bench Control with Real-Time Simulation. *SAE International Journal of Commercial Vehicles*, 10(1), 2017. ISSN 1946-3928.

Sascha Konrad und Cheng H.C. Betty. Real-time Specification Patterns. In *Proceedings of the International Conference on Software Engineering, 2005*, 2005.

Yasser Kotb und Takuya Katayama. Consistency Checking of UML Model Diagrams Using the XML Semantics Approach. In Allan Ellis und Tatsuya Hagino, Hrsg., *Proceedings of the 14th international conference on World Wide Web, WWW 2005*, Seiten 982–983. ACM, 2005. ISBN 1-59593-051-5.

Ron Koymans. Specifying real-time properties with metric temporal logic. In *Real-Time Systems*, volume 2, Seiten 255–299, 1990.

Daniel Kroening und Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer, Berlin, Heidelberg, 2008. ISBN 978-3-662-50496-3.

Thomas Kühne. Matters of (Meta-) Modeling. *Software & Systems Modeling*, 5(4):369–385, 2006. ISSN 1619-1366.

Kim G. Larsen, Paul Pettersson, und Wang Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997. ISSN 1433-2779.

Saul X. Levmore und Elizabeth Early Cook. *Super strategies for puzzles and games*. Doubleday, Garden City N.Y., 1st ed. Aufl., 1981. ISBN 038517165X.

Ran Li, Jiaqi Yin, Huibiao Zhu, und Phan Cong Vinh. Verification of RabbitMQ with Kerberos Using Timed Automata. *Mobile Networks and Applications*, 27(5):2049–2067, 2022. ISSN 1383-469X.

Mark H. Liffiton und Karem A. Sakallah. Algorithms for Computing Minimal Unsatisfiable Subsets of Constraints. *Journal of Automated Reasoning*, 40(1):1–33, 2008. ISSN 0168-7433.

C. L. Liu und James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1): 46–61, 1973. ISSN 0004-5411.

Rongshen Long, Hong Li, Wei Peng, Yi Zhang, und Minde Zhao. An Approach to Optimize Intra-ECU Communication Based on Mapping of AUTOSAR Runnable Entities. In Tianzhou Chen und Dimitrios N. Serpanos, Hrsg., *International Conference on Embedded Software and Systems*, Seiten 138–143. IEEE Computer Society, 2009. ISBN 978-0-7695-3678-1.

Inês Lynce und João P. Marques Silva. On Computing Minimum Unsatisfiable Cores of Satisfiability Testing. In *SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing, 10-13 May 2004, Vancouver, BC, Canada, Online Proceedings*, 2004.

Nesredin Mahmud, Cristina Seceleanu, und Oscar Ljungkrantz. ReSA Tool: Structured Requirements Specification and SAT-based Consistency-checking. In Maria Ganzha, Leszek A. Maciaszek, und Marcin Paprzycki, Hrsg., *FedCSIS 2016*, Seiten 1737–1746. IEEE Computer Society, 2016.

Frédéric Mallet und Robert de Simone. Correctness issues on MARTE/CCSL constraints. *Science of Computer Programming*, 106:78–92, 2015.

Salvador Martínez, Massimo Tisi, und Rémi Douence. Reactive model transformation with ATL. *Science of Computer Programming*, 136(4):1–16, 2017.

MathWorks. MATLAB/Simulink, 2022. URL https://www.mathworks.com/products/simulink.html.

Marcilio Mendonça, Andrzej Wasowski, und Krzysztof Czarnecki. SAT-based Analysis of Feature Models is Easy. In Dirk Muthig und John D. McGregor, Hrsg., *SPLC 2009*, Seiten 231–240. ACM, 2009.

Tom Mens, Ragnhild Van der Straeten, und Jocelyn Simmonds. A Framework for Managing Consistency of Evolving UML Models. In Hongji Yang, Hrsg., *Software Evolution with UML and XML*, Seiten 1–30. IGI Global, 2005. ISBN 9781591404620.

Robin Milner. *A calculus of communicating systems*. Lecture Notes in Computer Science. Springer, 1980. ISBN 3-540-10235-3.

Stefan Neumann und Holger Giese. Scalable Real-Time Compatibility for Embedded Components using Language-Progressive TIOA. In *Proceedings of the 16th IEEE Computer Society Symposium on Object/Component/Service-Oriented real-Time Distributed Computing (ISORC)*. IEEE Computer Society, 2013. ISBN 978-1-4799-2111-9.

Stefan Neumann, Norman Kluge, und Sebastian Wätzoldt. Automatic transformation of abstract AUTOSAR architectures to timed automata. In Iulian Ober, Hrsg., *Proceedings of the 5th International Workshop on Model Based Architecting and Construction of Embedded Systems - ACES-MB '12*, Seiten 55–60, New York, New York, USA, 2012. ACM Press. ISBN 9781450318006.

Object Management Group. Software & Systems Process Engineering Meta-Model Specification, 2008. URL https://www.omg.org/spec/SPEM/2.0/PDF.

Object Management Group. UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, 2011a. URL https://www.omg.org/omgmarte/Documents/Specifications/08-06-09.pdf.

Object Management Group. Unified Modeling Language (OMG UML) Superstructure, 2011b. URL https://www.omg.org/spec/UML/2.4.1/Superstructure/PDF.

Object Management Group. Object Constraint Language, 2014. URL http://www.omg.org/spec/OCL/2.4/.

Object Management Group. Unified Modeling Language (OMG UML), 2015. URL https://www.omg.org/spec/UML/2.5/PDF.

Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/-Transformation Specification, 2016.

Object Management Group. OMG Systems Modeling Language, 2017.

Ernst-Rüdiger Olderog und Henning Dierks. *Real-Time Systems: Formal Specification and Automatic Verification*. Cambridge University Press, Cambridge, 2008. ISBN 9780511619953.

Simon Perathoner, Ernesto Wandeler, Lothar Thiele, Arne Hamann, Simon Schliecker, Rafik Henia, Razvan Racu, Rolf Ernst, und Michael González Harbour. Influence of different abstractions on the performance analysis of distributed hard real-time systems. *Design Automation for Embedded Systems*, 13(1-2):27–49, 2009. ISSN 0929-5585.

Dorin B. Petriu und Murray Woodside. A Metamodel for Generating Performance Models from UML Designs. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard

Weikum, Thomas Baar, Alfred Strohmeier, Ana Moreira, und Stephen J. Mellor, Hrsg., *<<UML>> 2004 - The Unified Modeling Language. Modelling Languages and Applications*, volume 3273 of *Lecture Notes in Computer Science*, Seiten 41–53. Springer, Berlin, Heidelberg, 2004. ISBN 978-3-540-23307-7.

Robin Philipp, Hedan Qian, Lukas Hartjen, Fabian Schuldt, und Falk Howar. Simulation-Based Elicitation of Accuracy Requirements for the Environmental Perception of Autonomous Vehicles. In Tiziana Margaria und Bernhard Steffen, Hrsg., *Leveraging Applications of Formal Methods, Verification and Validation*, volume 13036 of *Lecture Notes in Computer Science*, Seiten 129–145. Springer, Cham, 2021. ISBN 978-3-030-89158-9.

Fabian Pittke, Benjamin Nagel, Gregor Engels, und Jan Mendling. Linguistic Consistency of Goal Models. In Wil van der Aalst, John Mylopoulos, Michael Rosemann, Michael J. Shaw, Clemens Szyperski, Ilia Bider, Khaled Gaaloul, John Krogstie, Selmin Nurcan, Henderik A. Proper, Rainer Schmidt, und Pnina Soffer, Hrsg., *Enterprise, Business-Process and Information Systems Modeling*, volume 175 of *Lecture Notes in Business Information Processing*, Seiten 393–407. Springer, Berlin, Heidelberg, 2014. ISBN 978-3-662-43744-5.

Gordon D. Plotkin. A structural approach to operational semantics. In *DAIMI-FN 19*, volume 60-61, Seiten 17–139, Aarhus University, 1981.

Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*, Seiten 46–57. IEEE Computer Society, 1977.

Klaus Pohl. *Requirements Engineering: Grundlagen, Prinzipien, Techniken*. Dpunkt-Verl., Heidelberg, 2., korrigierte aufl. Aufl., 2008. ISBN 978-3-89864-550-8.

Amalinda Post, Jochen Hoenicke, und Andreas Podelski. rt-inconsistency: a new property for real-time requirements. In Dimitra Giannakopoulou und Fernando Orejas, Hrsg., *14th International Conference on Fundamental Approaches to Software Engineering, FASE 2011*, Lecture Notes in Computer Science, Berlin, Heidelberg, 2011a. Springer.

Amalinda Post, Andreas Podelski, und Jochen Hoenicke. Vacuous real-time requirements. In *19th IEEE International Requirements Engineering Conference, RE 2011*, Seiten 153–162. IEEE Computer Society, 2011b.

Jean Pierre Queille und Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In Mariangiola Dezani-Ciancaglini und Ugo Montanari, Hrsg., *Proceedings of the 5th International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, Seiten 337–351, Berlin, Heidelberg, 1982. Springer. ISBN 978-3-540-11494-9.

Chander Ramchandani. *Analysis of Asynchronous Concurrent Systems by Timed Petri Nets*. Dissertation, Massachusetts Institute of Technology, Massachusetts, 1973.

Holger Rasch und Heike Wehrheim. Checking Consistency in UML Diagramms. In Elie Najm, Uwe Nestmann, und Perdita Stevens, Hrsg., *Formal Methods for Open Object-Based Distributed Systems, 6th IFIP WG 6.1 International Conference, FMOODS 2003*, Lecture Notes in Computer Science, Seiten 229–243, Berlin, Heidelberg, 2003. Springer. ISBN 3-540-20491-1.

Rational Software. Rational Unified Process: Best Practices for Software Development Teams. *White Paper, IEEE Annals of the History of Computing*, TP026B, Rev 11/01, 1998.

G. M. Reed und A. W. Roscoe. A timed model for communicating sequential processes. In G. Goos, J. Hartmanis, D. Barstow, W. Brauer, P. Brinch Hansen, D. Gries, D. Luckham, C. Moler, A. Pnueli, G. Seegmüller, J. Stoer, N. Wirth, und Laurent Kott, Hrsg., *Proceedings of the 13th International Colloquium on Automata, Languages and Programming, ICALP 1986*, volume 226 of *Lecture Notes in Computer Science*, Seiten 314–323, Berlin, Heidelberg, 1986. Springer. ISBN 978-3-540-16761-7.

Othmane Rhandor. *Ein Framework zur Zeitanalyse von AUTOSAR Steuergeräten*. Masterarbeit, Universität Paderborn, Paderborn, 2012.

Fabíola Gonçalves C. Ribeiro, Carlos E. Pereira, Achim Rettberg, und Michel S. Soares. Model-based requirements specification of real-time systems with UML, SysML and MARTE. *Software & Systems Modeling*, 17(1): 343–361, 2018. ISSN 1619-1366.

Kai Richter. *Compositional Scheduling Analysis Using Standard Event Models: The SymTA/S Approach*. Dissertation, Braunschweig, 2005.

Winston W. Royce. Managing the development of large software systems: concepts and techniques. *IEEE WESTCON, Los Angeles*, Seiten 1–9, 1970.

Jörg Schäuffele und Thomas Zurawka. *Automotive Software Engineering: Grundlagen, Prozesse, Methoden und Werkzeuge effizient einsetzen*. Vieweg+Teubner, 4 Aufl., 2010.

Oliver Scheickl und Christoph Ainhauser. Tool Support for Seamless System Development based on AUTOSAR Timing Extensions. In *Embedded Real-Time Software and Systems 2012*, 2012.

Andy Schürr. Specification of graph translators with triple graph grammars. In Ernst W. Mayr, Gunther Schmidt, Gottfried Tinhofer, (None), (None), und (None), Hrsg., *Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science, WG 1994*, volume 903 of *Lecture Notes in Computer Science*, Seiten 151–163, Berlin, Heidelberg, 1994. Springer. ISBN 978-3-540-59071-2.

Tilman Seifert, Florian Jug, und Günther Rackl. Automated Quality Assurance for UML Models. In Armin B. Cremers, Rainer Manthey, Peter Martini, und Volker Steinhage, Hrsg., *Informatik 2005 - Informatik LIVE!/2*, LNI, Seiten 496–500. GI, 2005. ISBN 3-88579-397-0.

Jocelyn Simmonds und M. Cecilia Bastarrica. A tool for automatic UML model consistency checking. In David Redmiles, Tom Ellman, und Andrea Zisman, Hrsg., *ASE 2005*, Seite 431. ACM, 2005. ISBN 1-58113-993-4.

Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer, Wien, New York, 1973. ISBN 978-3211811061.

Thomas Stahl und Markus Völter. *Model-driven software development: Technology, engineering, management / Thomas Stahl and Markus Völter, with Jorn Bettin, Arno Haase and Simon Helsen ; foreword by Krzysztof Czarnecki ; translated by Bettina von Stockfleth*. John Wiley, Chichester, 2006. ISBN 978-0-470-02570-3.

John A. Stankovic und Krithi Ramamritham. *Hard real-time systems*. IEEE Computer Society, Washington, DC, USA, 1988. ISBN 978-0-8186-0819-3.

R. Stark, H. Hayka, J. H. Israel, M. Kim, P. Müller, und U. Völlinger. Virtuelle Produktentstehung in der Automobilindustrie. *Informatik-Spektrum*, 34(1):20–28, 2011. ISSN 0170-6012.

Tino Teige, Bienmüller Tom, und Hans Jürgen Holberg. Universal Pattern: Formalization, Testing, Coverage, Verification and Test Case Generation for Safety-Critical Requirements. In Ralf Wimmer, Hrsg., *19. GI/ITG/GMM-Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, MBMV 2016*. Albert-Ludwigs-Universität Freiburg, 2016. ISBN 978-3-00-052380-9.

Lothar Thiele, Samarjit Chakraborty, und Martin Naedele. Real-Time Calculus for Scheduling Hard Real-Time Systems. In *Int. Symposium on Circuits and Systems ISCAS 2000*, 2000.

Jan Toennemann, Andreas Rausch, Falk Howar, und Benjamin Cool. Checking Consistency of Real-Time Requirements on Distributed Automotive Control Software Early in the Development Process Using UPPAAL. In *FMICS 2018*, volume 11119 of *Lecture Notes in Computer Science*, Seiten 67–82. Springer, 2018.

Ricardo F. Tomás, Paulo Fernandes, Eloisa Macedo, Jorge M. Bandeira, und Margarida C. Coelho. Assessing the emission impacts of autonomous vehicles on metropolitan freeways. *Transportation Research Procedia*, 47(4): 617–624, 2020. ISSN 23521465.

Matthias Traub. *Durchgängige Timing-Bewertung von Vernetzungsarchitekturen und Gateway-Systemen in Kraftfahrzeugen*. Dissertation, Karlsruher Institut für Technologie, Karlsruhe, 2010. URL https://publikationen.bibliothek.kit.edu/1000020379.

United Nations. Proposal for a new UN Regulation on uniform provisions concerning the approval of vehicles with regards to Automated Lane Keeping System, 2020. URL https://undocs.org/ECE/TRANS/WP.29/2020/81.

Nils Weidmann und Anthony Anjorin. Schema Compliant Consistency Management via Triple Graph Grammars and Integer Linear Programming. *Formal Aspects of Computing*, 33(6):1115–1145, 2021. ISSN 0934-5043.

Nils Weidmann, Anthony Anjorin, Lars Fritsche, Gergely Varró, Andy Schürr, und Erhan Leblebici. Incremental Bidirectional Model Transformation with eMoflon::IBeX. In James Cheney und Hsiang-ShangEditors Ko, Hrsg., *Proceedings of the 8th International Workshop on Bidirectional Transformations co-located with the Philadelphia Logic Week, Bx@PLW 2019*, Seiten 45–55. CEUR-WS.org, 2019.

Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, und Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):1–53, 2008. ISSN 1539-9087.

Ernest Wozniak. An Optimization Approach for the Synthesis of AUTOSAR Architectures: 10 - 13 Sept. 2013, Cagliari, Italy. In Carla Seatzu, Hrsg., *2013 IEEE 18th Conference on Emerging Technologies & Factory Automation (ETFA)*. IEEE Computer Society, 2013. ISBN 978-1-4799-0864-6.

Wang Yi. CCS + time = an interleaving model for real time systems. In Javier Leach Albert, Burkhard Monien, und Mario Rodríguez Artalejo, Hrsg., *Proceedings of the 18th International Colloquium on Automata, Languages and Programming, ICALP 1991*, volume 510 of *Lecture Notes in Computer Science*, Seiten 217–228, Berlin, Heidelberg, 1991. Springer. ISBN 978-3-540-54233-9.

Sergio Yovine. KRONOS: a verification tool for real-time systems. *International Journal on Software Tools for Technology Transfer*, 1(1-2):123–133, 1997. ISSN 1433-2779.

Ekim Yurtsever, Jacob Lambert, Alexander Carballo, und Kazuya Takeda. A Survey of Autonomous Driving: Common Practices and Emerging Technologies. *IEEE Access*, 8:58443–58469, 2020.

Licong Zhang, Reinhard Schneider, Alejandro Masrur, Martin Becker, Martin Geier, und Samarjit Chakraborty. Timing challenges in automotive software architectures. In Pankaj Jalote, Lionel Briand, und André van der Hoek, Hrsg., *Companion Proceedings of the 36th International Conference on Software Engineering, ICSE Companion 2014*, Seiten 606–607, New York, New York, USA, 2014. ACM. ISBN 978-1-4503-2768-8.

Qi Zhu, Haibo Zeng, Wei Zheng, Marco Di Natale, und Alberto Sangiovanni-Vincentelli. Optimization of task allocation and priority assignment in hard real-time distributed systems. *ACM Transactions on Embedded Computing Systems*, 11(4):1–30, 2012. ISSN 1539-9087.