T U M

I N S T I T U T F Ü R I N F O R M A T I K

Umfassendes Architekturmodell für das Engineering eingebetteter Software-intensiver Systeme

Manfred Broy, Martin Feilkas, Johannes Grünbauer, Alexander Gruler, Alexander Harhurin, Judith Hartmann, Birgit Penzenstadler, Bernhard Schätz, Doris Wild

TUM-I0816
Juni 08

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-06-I0816-0/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

Zusammenfassung

In diesem technischen Bericht wird ein umfassendes Architekturmodell für die Entwicklung von eingebetteten Systemen vorgestellt. Im Vordergrund stehen hier Software-intensive Systeme, so wie sie im Automobil zu finden sind. Das Architekturmodell kann jedoch auch auf alle anderen Branchen übertragen werden, die sich mit der Entwicklung von Software-intensiven eingebetteten Systemen beschäftigen.

Die Grundidee hinter diesem umfassenden Architekturmodell ist eine Software-Entwicklung entlang geeigneter automobilspezifischer Abstraktionsebenen. Damit soll die Grundlage für einen systematischen Software-Entwicklungsprozess und für eine modellbasierte Werkzeugunterstützung geschaffen werden.

Dieser Bericht konzentriert sich auf die Charakterisierung der einzelnen Abstraktionsebenen, deren Ziele, Hauptelemente und Sichten. Die wichtigsten Konzepte werden anhand eines durchgängigen Fallbeispiels veranschaulicht.

Inhaltsverzeichnis

1 Einleitung

Die Entwicklung von Automotive Software war bisher – geprägt durch den klassischen Zulieferprozess – auf eine steuergeräteorientierte Architektur ausgerichtet: eine Hauptfunktion – meist aus mehreren Teilfunktionen bestehend – entsprach vielfach einem Steuergerät, es bestand nur wenig Interaktion zwischen den Hauptfunktionen. Durch den Bedarf für eine wachsende Anzahl von Funktionen (und damit auch der Anzahl der Steuergräte) und die Notwendigkeit der stärkeren Vernetzung von Funktionen (und damit auch der stärkeren Interaktion) kann eine ausschließlich steuergeräteorientierte Architektur der gestiegenen Komplexität hinsichtlich Entwicklungsqualität und -effizienz nicht mehr Rechnung tragen.

Wesentliches Problem ist dabei die Existenz grundsätzlicher Mängel einer solchen Architektur hinsichtlich Aspekten wie

- Frühe Erfassung der funktionalen und domänenspezifisch-architektonischen Abhängigkeiten und damit frühere Kompatibilitäts- und Qualitätssicherung,
- Getrennte Darstellung der funktionalen, logischen, und technischen Anforderungen,
- Erfassung von Abhängigkeiten zwischen den funktionalen, logischen, und technischen Anforderungen und Lösungen sowie Verfolgung von Änderungsauswirkungen,
- Nutzung und Dokumentation von Entscheidungsspielräumen bei der Gruppierung von Funktionen zu fachlichen Komponenten, beispielsweise zur Optimierung der Wiederverwendung,
- Nutzung und Dokumentation von Entscheidungsspielräumen bei der Abbildung von Komponenten zu Hardware- und Softwareeinheiten, beispielsweise zur Speicher- und Laufzeitoptimierung.

Neuere Ansätze wie AUTOSAR [AUT07b] unternehmen hier den ersten Schritt von der Trennung von Softwarekomponente und Steuergerät durch die Einführung einer – jedoch stark software*technisch* geprägten – Komponentenarchitektur.

Für eine substantielle Veränderung ist jedoch ein *einheitliches*, *einfaches*, *klares*, und *umfassendes* Modell für automotive Software notwendig, das die Ebenen

- Nutzungsfunktionsarchitektur
- Logische Komponentenarchitektur
- Technische Implementierungsarchitektur

trennt und in Beziehung setzt. Im Folgenden wird ein solches Modell vorgestellt. Auf jeder Abstraktionsebene werden nach einem einheitlichen Prinzip die drei wesentlichen Elemente des jeweiligen Architekturmodells beschrieben: die Festlegung der *wesentlichen Bausteine*, das zentrale *Strukturierungsprinzip*, und die Darstellung von *Wechselwirkungen*, die zwischen den wesentlichen Bausteinen auftreten können:

Nutzungsebene: Nutzungsfunktionen, Funktion/Subfunktion-Beziehung, Funktionsabhängigkeiten

Logische Architektur: Komponenten, Komponente/Unterkomponente-Beziehung, Interaktionsbeziehungen

Technische Architektur: Hardware-/Softwareeinheiten, Einbettungsbeziehung, Auswirkung der Einbettung

Als zentrales koordinierendes Element der Systementwicklung reicht dabei die Bedeutung eines solchen Architekturmodells von der Grundlage für eine phasenübergreifende Werkzeugunterstützung, über die Strukturierung eines herstellerübergreifenden Zuliefer- und Integrationsprozesses, bis hin zur Klärung der organisatorischen Abstimmungsprozesse, und hat somit neben der fachlichen auch eine strategische Dimension.

Struktur dieses Berichts Im folgenden Kapitel werden zunächst die wichtigsten Begriffe definiert und die Abstraktionsebenen kurz vorgestellt. Die Kapitel 3-5 widmen sich dann der ausführlichen Beschreibung der einzelnen Abstraktionsebenen. Das Papier schließt in Kapitel 6 mit einer kurzen Zusammenfassung des Ansatzes. Die im Folgenden eingeführten Konzepte werden anhand eines durchgängigen Fallbeispiels veranschaulicht. Als Fallbeispiel wurde eine typische Automobilfunktion, das ACC-System (Adaptive Cruise Control) gewählt. Dieses umfasst eine Geschwindigkeitsregelung (Tempomat), eine Abstandsregelung zu Objekten in Fahrtrichtung, sowie eine Unfallwarnung. Die Angaben zum Fallbeispiel sind größtenteils dem Bericht eines Interdisziplinären Projekts [Sch07] entnommen, dessen Ziel die Modellierung des ACC-Systems war. Für eine ausführliche Beschreibung des Fallbeispiels sei auf diesen Bericht verwiesen. Alle relevanten Details werden jedoch an den entsprechenden Stellen im Papier eingeführt.

2 Architekturmodell für Software-intensive Systeme im Automobil - Einführung

In diesem Kapitel werden zunächst einige wesentliche Begriffe geklärt und danach das Architekturmodell im Überblick vorgestellt.

2.1 Begriffsdefinitionen

System Ein System besteht aus Elementen (Komponenten, Subsystemen), die zueinander in Beziehung stehen. Ein System ist durch die Definition von Systemgrenzen von seiner Umwelt weitgehend abgrenzt. Alle Elemente, die nicht innerhalb der Systemgrenze liegen, gehören zur Umwelt. Die Schnittstelle eines Systems gibt die Beziehungen zwischen den Elementen des Systems und den Elementen der Umwelt des Systems an. Im Kontext dieses Berichts werden eingebettete Systeme betracht: Ein eingebettetes System ist ein System, das integraler Bestandteil eines physikalischen System ist und zentrale Aufgaben zur Regelung und Steuerung für die Bereitstellung der Funktionalität des Gesamtsystems übernimmt.

Sicht Sichten zeigen ein System aus einer bestimmten Perspektive. Sie bieten für diese Perspektiven geeignete Modellierungskonzepte an und ermöglichen damit eine genaue Betrachtung und Analyse dieser Perspektiven. Beispiele für solche Perspektiven sind die Verhaltensperspektive oder die Datenperspektive eines Systems.

Abstraktionsebene Eine Abstraktionsebene beschreibt das gesamte System unter einem bestimmten Abstraktionsgrad. Der Abstraktionsgrad ist dabei so gewählt, dass jede Ebene bestimmte Aspekte, die bei der Entwicklung von Software-intensiven Systemen wichtig sind, besonders gut darstellen kann. Jede Ebene beinhaltet mehrere Sichten, die geeignete Modelle für diese Aspekte bieten.

Architekturmodell Ein umfassendes Architekturmodell für ein System beschreibt alle Modellsichten, die die Struktur dieses Systems festlegen und deren Beziehungen untereinander. Die Modellsichten werden entsprechend ihres Abstraktionsgrades in aufeinander aufbauende Abstraktionsebenen zusammengefasst.

2.2 Herausforderungen beim Entwurf von Software-intensiven Systemen im Automobil

Der Entwurf von Software-intensiven Systemen im Automobil unterliegt - wie in [Bro06] und [Gri05] beschrieben - besonderen Herausforderungen.

In der Automobilindustrie basiert ein hoher Anteil wettbewerbsrelevanter Innovationen auf Software. Damit geht einher, dass es auch immer mehr Funktionen gibt, die über ein heterogenes Steuergerätenetzwerk verteilt werden. Durch den hohen Verteilungsgrad der Funktionen wird die Komplexität der Funktionsentwicklung erhöht und zugleich die Möglichkeit, unerwünschte Abhängigkeiten der Funktionen untereinander zu entdecken, enorm erschwert.

Darüber hinaus herrscht in der Automobilindustrie ein sehr hoher Kostendruck. Dies hat zur Folge, dass kostengünstige Hardware verwendet wird, die sowohl vom Speicherplatz her knapp bemessen ist, als auch von der Rechengeschwindigkeit her sehr niedrig angesetzt ist. Da ein Teil der Funktionen im Fahrzeug harte Echtzeitanforderungen erfüllen muss, stehen sich hier zwei widersprüchliche Anforderungsklassen gegenüber, die jedoch beide erfüllt werden müssen.

Auch die Themen Wiederverwendung und Variabilität sind in der Automobilindustrie von zentraler Bedeutung. Es gibt einerseits Basisfunktionalitäten, wie die elektronische Ansteuerung des Fensterhebers, die sich in allen Fahrzeugen eines Automobilherstellers befinden, und andererseits variable Funktionalitäten, die sich - abhängig von der gewählten Ausstattungsvariante - in unterschiedlichen Ausbaustufen in den einzelnen Fahrzeugen befinden, wie etwa bei der Einparkhilfe (vorne und/oder hinten, visuell und/oder mit Warntönen, ...). In beiden Fällen möchte man den Wiederverwendungsgrad möglichst hoch halten, d.h. die Funktionalität soll nicht für jede Ausstattungsvariante und jede mögliche Hardwareplattform vollständig neu entworfen bzw. aufwändig geändert werden müssen.

Die genannten Spezifika gelten nicht nur in der Automobilindustrie. Die meisten Herausforderungen beim Entwurf von Software-intensiven Systemen im Automobil lassen sich auch auf

viele andere Branchen übertragen, die eingebettete Systeme verwenden. Hervorzuheben wären hier die Mobiltelefonhersteller. In einem Mobiltelefon befindet sich zwar kein heterogenes Steuergerätenetzwerk, aber dennoch ist die Komplexität aufgrund der extremen Multifunktionalität entsprechend hoch und damit die Gefahr von unerwünschten Wechselwirkungen zwischen Funktionen beträchtlich. Die Systeme müssen auch hier sehr effizient sein, d.h. geringe Codegröße und Einhaltung von Echtzeitbedingungen trotz kostensparender Hardwareressourcen sind in dieser Branche ebenso eine Anforderung.

2.3 Überblick

In Abbildung 1 ist das Architekturmodell für Software-intensive Systeme im Automobil im Überblick dargestellt. Es ist grundsätzlich unterteilt in folgende drei Abstraktionsebenen:

- Nutzungsebene
- Logische Architektur
- Technische Architektur

Der Abstraktionsgrad nimmt dabei von oben nach unten ab.

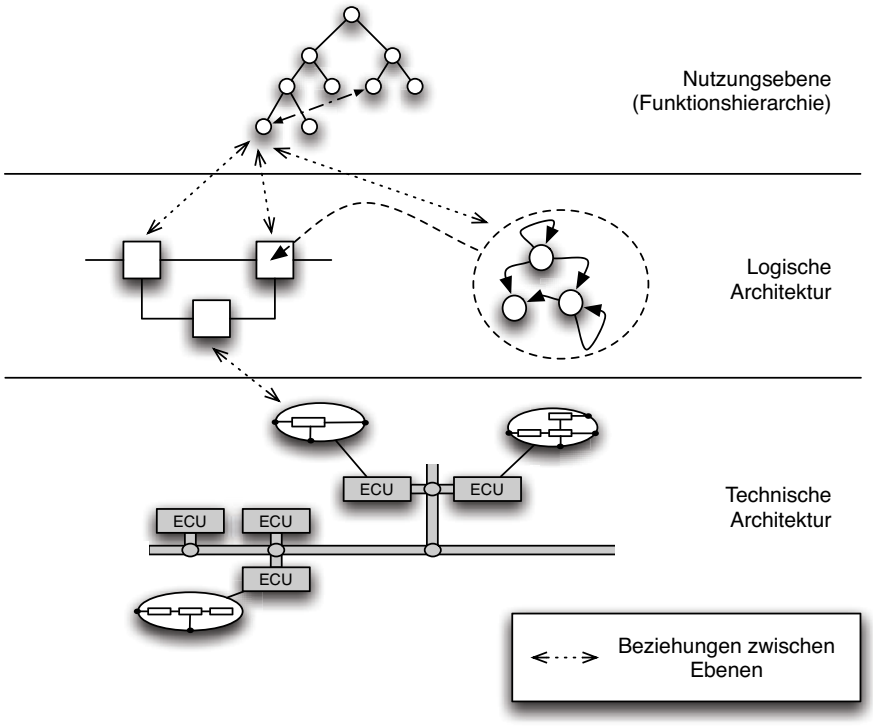

Abbildung 1: Das Ebenenmodell

Die Aufteilung der Ebenen wurde in dieser Form vorgenommen, um den besonderen Herausforderungen bei der Entwicklung von Software für eingebettete Systeme und speziell denen der Automobilindustrie zu begegnen. In jeder Ebene werden geeignete Modelle verwendet, mit denen sich die Aspekte der einzelnen Herausforderungen besonders gut beschreiben lassen.

Die Nutzungsebene bietet Modelle, die erlauben, funktionale Anforderungen zu formalisieren, diese in Form von hierarchischen Beziehungen darzustellen und zusätzlich Abhängigkeiten zwischen diesen funktionalen Anforderungen darzustellen. Sie bietet damit die Grundlage, unerwünschte Wechselwirkungen zwischen Funktionen früh im Entwicklungsprozess zu erkennen. Dadurch und durch den hohen Abstraktionsgrad ist sie auch gut für die Erweiterung um Produktlinienkonzepte geeignet.

Die logische Architektur bietet Modelle, die eine Strukturierung der Funktionalität in fachliche Komponenten erlaubt. Die in der Nutzungsebene formalisierten funktionalen Anforderungen werden durch ein Netzwerk von hierarchischen Komponenten, die jedoch noch unabhängig von der zugrunde liegenden Hardware sind, realisiert. Das Modell des Systems auf der Ebene der logischen Architektur ist ausführbar und simulierbar, und damit einer frühen Architekturverifikation zugängig. Durch die Modularisierung und die Hardwareunabhängigkeit dieser Ebene wird die Komplexität des Modells reduziert und ein hohes Potential für die Wiederverwendung geschaffen.

Die technische Architektur beschreibt schließlich abstrakt die Realisierung, welche aus Hardware und Software besteht. Sie bietet geeignete Modelle, die das Verhalten von Hardware und Software einheitlich beschreiben und erlauben, den Einfluss der verwendeten Hardware auf das Verhalten des gesamten Systems zu beschreiben. Der Abstraktionsgrad ist dabei so gewählt, dass Aussagen über die Einhaltung von Echtzeitanforderungen möglich sind und beim weiteren Übergang von der technischen Architektur zur Implementierung lediglich sofwaretechnische Umformungen (z.B. Middlewareaufrufe), jedoch keine Verhaltensveränderungen mehr stattfinden.

Durch die Trennung von Hardwareabhängigkeit und Hardwareunabhängigkeit im gesamten Modell ist auf jeder Ebene ein hohes Potential für die Wiederverwendung gegeben.

In den folgenden Kapiteln werden die drei Abstraktionsebenen unter folgenden Aspekten beschrieben:

- Kurzbeschreibung und Ziele
- Hauptelemente
- Sichten

Oben genannte Aspekte konzentrieren sich auf die Elemente und deren Beziehungen innerhalb einer Abstraktionsebene. Der Übergang von einer Abstraktionsebene zur nächsten und damit die Beziehungen zwischen Elementen zweier benachbarter Ebenen ist noch Gegenstand der aktuellen Forschung und steht daher in diesem Papier nicht im Vordergrund.

2.4 Verwandte Arbeiten

Die Idee, die Komplexität durch eine systematische Softwareentwicklung für eingebettete Systeme entlang von domänenspezifischen Abstraktionsebenen zu reduzieren und damit die Funktio-

nalität und die Plattform getrennt zu behandeln, ist nicht neu. Innerhalb der Automobildomäne sind hier vor allem drei Ansätze, nämlich AUTOSAR, EAST ADL und mobilSoft, zu nennen.

EAST ADL Die EAST ADL (**E**lectronics **A**rchitecture and **S**oftware **T**echnology - **A**rchitecture **D**efinition **L**anguage) ist 2004 im Rahmen des ITEA[1] Projekts EAST EEA [ITE08], bestehend aus Automobilherstellern, Zulieferern, Softwareherstellern und Universitäten, entstanden. Sie ist speziell für die Automobildomäne entworfen worden und beschreibt Software-intensive Elektrik/Elektronik-Systeme im Automobil auf fünf unterschiedlichen Abstraktionsebenen beginnend mit „High Level" - Anforderungen und nutzersichtbaren Features bis hin zu implementierungsnahen Details wie Betriebssystemkonstrukten (z.B. „Tasks") und elektronischer Hardware. Hierbei liegt der Fokus der EAST ADL auf der Beschreibung der strukturellen Aspekte und nicht auf der Beschreibung des Verhaltens. Die Verhaltensbeschreibung stammt zu einem großen Teil aus externen Werkzeugen (wie z.B. Matlab/Simulink [The07] und ASCET [ETA08a]). Die EAST ADL setzt auf UML2 [OMG07] auf und definiert ein UML2-Profil. Die EAST ADL wurde als Ausgangspunkt für die Entwicklung des Industriestandards AUTOSAR [AUT07b] (siehe unten) genutzt.

Bezug zum hier vorgestellten Gesamtarchitekturmodell:
Der in diesem Bericht vorgestellte Ansatz eines Gesamtarchitekturmodells beschreibt die Struktur und das Verhalten des Systems integriert. Die EAST ADL legt den Fokus auf die Struktur. Auch liegt bei der EAST ADL eine formale Grundlage für die Modelle nicht im Vordergrund, sondern vielmehr die Aufteilung der Systementwicklung in Artefakte, die unterschiedlichen Abstraktionsebenen zugeordnet werden können, und die Entwicklung einer einheitlichen Sprache für diese Artefakte. Das Gesamtarchitekturmodell geht hier noch einen Schritt weiter und möchte mit seinen Modellen, die auf einer formalen Grundlage basieren, ein Fundament für ein Werkzeug schaffen, das einen durchgängigen und systematischen Entwicklungsprozess unterstützt. Die Abstraktionsebenen der EAST ADL dienten dem in diesem Bericht vorgestellten Gesamtarchitekturmodell als Basis. So kann von den Inhalten und Zielen her das *Vehicle Feature Model* und die *Functional Analysis Architecture* der EAST ADL als Pendant zur Nutzungsebene gesehen werden, die *Functional Design Architecture* entspricht in etwa der Ebene der logischen Architektur und der Abstraktionsgrad des *Function Instance Models*, des *Platform Models* und des *Allocation Models* entsprechen ungefähr dem Abstraktionsgrad der Modelle, die auf der Ebene der technischen Architektur zu finden sind.

AUTOSAR Die industrielle Entwicklungspartnerschaft AUTOSAR (**AUT**omotive **O**pen **S**ystem **AR**chitecture) wurde 2003 mit dem Ziel gegründet, die wachsende Komplexität bei der Entwicklung von Elektrik/Elektronik-Systemen in Kraftfahrzeugen zu beherrschen [AUT07b]. Dies soll durch die Entwicklung einer standardisierten Softwareinfrastruktur auf Basis einer standardisierten Softwarearchitektur, die eine modulare und weitgehend plattformunabhängige Softwareentwicklung erlaubt, erreicht werden. Die von AUTOSAR vorgeschlagene Softwarearchitektur ist eine Schichtenarchitektur, die auf der untersten Schicht von der Hardware abstrahiert und auf der obersten Schicht Schnittstellen für die Applikation bietet. Durch die Trennung von plattformspezifischen und applikationsspezifischen Softwareanteilen und die

[1] Information Technology for European Advancement, www.itea-office.org

Standardisierung der Schnittstellen ist eine Grundlage für eine plattformunabhängige und modulare Entwicklung von Funktionen geschaffen worden.

Bezug zum hier vorgestellten Gesamtarchitekturmodell:
Das Verhalten der AUTOSAR-Softwarekomponenten ist bereits auf die in der Automobildomäne üblichen Plattformen zugeschnitten. Ein durchgängiger Entwicklungsprozess, der ausgehend von bereits festgelegten Anforderungen als Ergebnis fertige AUTOSAR-Softwarekomponenten liefert, existiert noch nicht. Das in diesem Bericht vorgestellte Gesamtarchitekturmodell dient als Basis für einen durchgängigen Entwicklungsprozess und hat zum Ziel, eine Systematik zur Softwareentwicklung entlang von automobilspezifischen Abstraktionsbenen anzugeben, die erlaubt anforderungskonforme Software für jede mögliche Plattform zu generieren. Als mögliches Ziel ist hier auch die Generierung von AUTOSAR-Softwarekomponenten denkbar.

mobilSoft Innerhalb des Forschungsverbundes „Softwaretechnik für das Automobil der Zukunft - mobilSoft“[2] [Mob07], zu dem sich die beiden bayerischen Automobilhersteller Audi AG und BMW AG, einige namhafte Zulieferer und Forschungseinrichtungen (unter anderem die TU München) zusammengeschlossen haben, wurde als ein Teilprojektergebnis ein Satz von automobilspezifischen Abstraktionsebenen entworfen und in [WFH+06] beschrieben.

Bezug zum hier vorgestellten Gesamtarchitekturmodell:
In mobilSoft wurden vier aufeinander aufbauende Abstraktionsebenen identifiziert, nämlich die Nutzungsebene, die Funktionsebene, die Clusterebene und die Plattformebene. Diese Ebenen dienten dem hier vorgestellten Ansatz als Basis. Die Nutzungsebene entspricht vom Abstraktionsgrad her der in diesem Ansatz vorgestellten Nutzungsebene, die Funktionsebene entspricht der logischen Architektur und die beiden Abstraktionsebenen Clusterebene und Plattformebene wurden in der technischen Architektur zusammengefasst.

Rich Component Model Das in [Dam05] und [DVM+05] beschriebene Rich Component Model (RCM) zielt auf eine frühe Integration von funktionalen und nichtfunktionalen Anforderungen ab. Dem RCM liegt ein Ebenenmodell zugrunde, das aus einer architekturunabhängigen funktionalen Ebene, einer ECU-Ebene und einer Hardwareebene besteht. RCMs erweitern UML2-Komponentenmodelle durch

- Komponentenspezifikationen, die um funktionale und nichtfunktionale „viewpoints“ (z.B. Echtzeit, Sicherheit, Ressourcen, Stromverbrauch, ...) angereichert werden
- Explizite Darstellung der Abhängigkeiten zwischen Zusicherungen („promises“) einer Komponentenspezifikation und den Prämissen („assumptions“), die an die Umgebung gemacht werden
- „Classifiers“ für die Zusicherungen, die die Position der Zusicherungen im Ebenenmodell und deren Vertrauenswürdigkeit angeben.

Die Zusicherungen und Prämissen werden mit Standardformalismen wie z.B. erweiterte, kommunizierende Zustandsmaschinen und Varianten von temporaler Logik spezifiziert.

[2] gefördert vom Bayerischen Wirtschaftsministerium unter IuK 188/001

Bezug zum hier vorgestellten Gesamtarchitekturmodell:
Beim RCM wird ein Konzept präsentiert, das Beziehungen zwischen einzelnen Ebenen anhand der Beziehungen zwischen den jeweiligen Komponenten der Ebenen beschreibt. Der Schwerpunkt liegt nicht in der Beschreibung des Systems unter unterschiedlichen Abstraktionsgraden, sondern in der Integration von nichtfunktionalen und funktionalen Anforderungen. Das in dem vorliegenden Bericht vorgestellte Architekturmodell legt seinen Schwerpunkt auf die Beschreibung des Systemverhaltens in seinem jeweiligen Abstraktionsgrad auf allen Ebenen und konzentriert sich daher auch nur auf die Realisierung der Anforderungen, die direkte Auswirkung auf das Verhalten des Systems haben.

3 Nutzungsebene

3.1 Kurzbeschreibung und Ziele

Die Nutzungsebene liefert eine strukturierte Spezifikation des Systemverhaltens, wie es an der Systemgrenze durch Benutzer wahrgenommen wird. Da mit Benutzer oft der spätere Endkunde gemeint ist, sprechen wir auch von *Kundenfunktionalitäten*, welche auf der Nutzungsebene in Form von Diensten formal beschrieben und hierarchisch strukturiert werden. Grundsätzlich kann ein Benutzer ein Mensch, aber auch ein anderes System sein. Das Hauptaugenmerk dieser Ebene liegt dabei auf der Strukturierung der Funktionalität in verschiedene Nutzungsgruppen. Ausschlaggebend sind hierbei rein funktionale, anwenderbezogene Aspekte.

Auf der Nutzungsebene wird für das zu spezifizierende Gesamtsystem die Systemgrenze festgelegt. Dies umfasst die Definition der Schnittstelle zur (externen) Umgebung (Fahrer, Straße etc.) sowie die Identifikation und Definition der Schnittstelle zu umliegenden Systemen, mit denen das zu entwickelnde System interagiert. Das Verhalten des Gesamtsystems wird dann aus *Blackbox-Sicht* spezifiziert, d. h. es wird der Nachrichtenaustausch an der identifizierten Systemgrenze, also zwischen Gesamtsystem und Umwelt, festgelegt. Dabei wird das Verhalten des Gesamtsystems in Form eines Dienstmodells strukturiert erfasst. Dieses setzt sich zusammen aus einer *Hierarchie von Diensten* und *Querbeziehungen* zwischen diesen. Dienste liefern eine *formale Spezifikation* einer Teilfunktionalität des Systems (einer Kundenfunktion) und können mittels geeigneter Notationstechniken, etwa durch generalisierte MSCs, Tabellen, I/O-Automaten oder Assumption/Guarantee-Spezifikationen repräsentiert werden. Querbeziehungen beschreiben, wie die modular definierten Dienste zusammenspielen bzw. sich gegenseitig beeinflussen, um das gewünschte Gesamtverhalten zu erbringen. Besteht also eine Querbeziehung zwischen Diensten, so ergibt sich das Gesamtverhalten nicht allein aus den modularen Spezifikationen der einzelnen Dienste, sondern nur unter zusätzlicher Berücksichtigung der Querbeziehung. Die Querbeziehungen machen somit die Interaktion der Dienste explizit und definieren das (über die modularen Spezifikationen hinausgehende) Interaktionsverhalten. Es gibt eine Vielzahl methodisch relevanter Querbeziehungen, z. B. *aktiviert/deaktiviert*, *enables/disables* oder *modifies*. Auf Basis der Querbeziehungen kann ein Abhängigkeitsmodell abgeleitet werden, das eine genauere Analyse der Abhängigkeiten zwischen einzelnen Subfunktionalitäten ermöglicht.

Abbildung 2 liefert einen schematischen Überblick über das Dienstmodell des ACC-Systems. Die Funktionalität des Gesamtsystems (`ACC`) lässt sich unterteilen in Teilfunktionalitäten zur

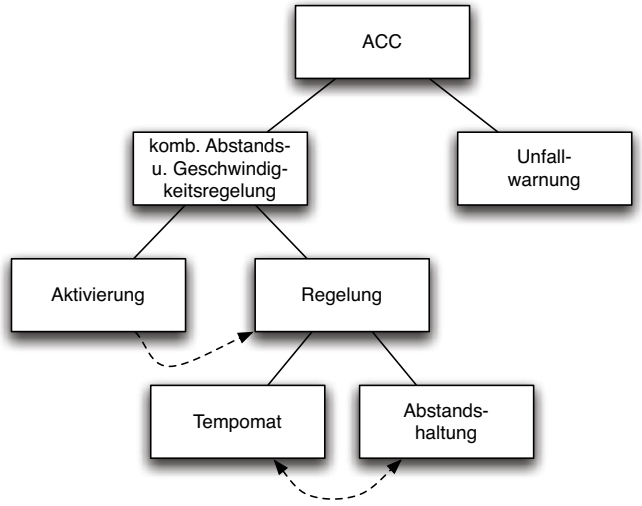

Abbildung 2: Dienstmodell des ACC-Systems

eigentlichen Regelung der Fahrzeuggeschwindigkeit (`Kombinierte Abstands- und Geschwindigkeitsregelung`) sowie zur Ausgabe einer Warnung bei Unfallgefahr (`Unfallwarnung`). Die Abstands- und Geschwindigkeitsregelung lässt sich weiter untergliedern in eine Funktionalität, die die Aktivierung des Systems regelt (`Aktivierung`), und die eigentliche Regelung, sofern das System aktiv ist (`Regelung`). Diese beinhaltet eine herkömmliche Tempomatfunktion (`Tempomat`) sowie eine Abstandsregelung, wenn ein Objekt in Fahrtrichtung erkannt ist (`Abstandshaltung`). Die einzelnen Teilfunktionalitäten beeinflussen sich gegenseitig, wie in der Abbildung durch gestrichelte Querbeziehungen dargestellt.

Die Zerlegung in Teildienste stellt keine Systemdekomposition in interagierende (kommunizierende) Einheiten dar, sondern lediglich eine hierarchische Strukturierung der Funktionalität, die das Gesamtsystem anbietet. Ein Teildienst stellt somit einen Ausschnitt aus dem Gesamtverhalten dar. Die Systemgrenze ändert sich bei der Zerlegung in Teildienste nicht: auch Teildienste spezifizieren nur das Verhalten, wie es an der Grenze des Gesamtsystems beobachtbar ist. Insbesondere stellt die syntaktische Schnittstelle des Teildienstes eine Projektion aus der Gesamtschnittstelle dar. Wie bereits erwähnt, ist die Strukturierung in Teildienste geprägt durch anwenderbezogene Aspekte. Die einzelnen Teildienste formalisieren verschiedene Möglichkeiten der Nutzung des Systems, z. B. durch verschiedene Anwender (vgl. UseCases). Von fachlich-strukturellen Aspekten (vor allem Kommunikation zwischen Teilkomponenten, vgl. Kapitel 4) wird auf dieser Ebene abstrahiert. Auf der Ebene der Logischen Architektur werden die Dienste und Querbeziehungen dann durch ein Netzwerk an Komponenten, die durch geeignete Kommunikationsbeziehungen miteinander verbunden sind, umgesetzt.

Obwohl die vollständige Definition des Gesamtsystemverhaltens wünschenswert ist, erlauben wir auf dieser Ebene auch *partielle* Verhaltensdefinitionen, da hier nur bekannte Anforderungen an das Gesamtsystemverhalten abgebildet werden. Einzelne Dienste legen das Verhalten nicht notwendigerweise für alle möglichen Eingabefolgen fest. Die Totalisierung erfolgt spätestens

in der darauf folgenden Ebene, der *logischen Architektur*, bei dem Übergang von Diensten zu (totalen) Komponenten.

Ausgangssituation Den Ausgangspunkt für die Festlegung der Nutzungsebene bildet ein Satz von Anforderungen an das Systemverhalten. Diese Anforderungen können in unterschiedlicher Form vorliegen, beispielsweise als textuelle Dokumentation einer Menge an einzelnen Anforderungen (z. B. Doors-Dokument oder rein textuelle Beschreibung in einem Lastenheft) oder als Sammlung von Use Cases[3].

Die Nutzungsebene bildet das Modell für die funktionalen Anforderungen. Geht man von dem idealisierten Fall aus, dass alle (funktionalen) Anforderungen an das System bereits informell erfasst sind, so werden diese beim Übergang zur Nutzungsebene durch Dienste und deren Querbeziehungen umgesetzt. Die Umsetzung erfolgt jedoch nicht durch eine Abbildung einer Anforderung auf genau einen Dienst, vielmehr stehen die ausgearbeiteten Anforderungen und die Dienste in einer $n : m$ Beziehung. Eine Anforderung kann durch einen oder mehrere Dienste umgesetzt werden und ein Dienst kann eine oder mehrere Anforderungen realisieren. Darüber hinaus ermöglicht die Formalisierung der Anforderungen auf der Nutzungsebene die Analyse bestehender Anforderungen und somit das Erkennen und Auflösen von Inkonsistenzen und fehlenden Anforderungen.

Ziele Zusammengefasst ergeben sich auf der Nutzungsebene folgende Ziele:

- Formale Beschreibung und funktionale Strukturierung des Black-Box-Verhaltens des Gesamtsystems;
- Konsolidierung der funktionalen Anforderungen durch eine formale Präzisierung der Anforderungen, die das Systemverhalten aus Nutzungssicht beschreiben;
- Erkennen und Auflösen von Widersprüchen in den funktionalen Anforderungen;
- Komplexitätsreduktion durch eine hierarchische Strukturierung der Funktionalität aus Nutzersicht;
- Besseres Verständnis der funktionalen Zusammenhänge durch formalisierte Erfassung und Analyse der Wechselwirkungen zwischen verschiedenen Funktionalitäten („Teildienste“);
- Separation of Concerns, d. h. eine geeignete Aufteilung der Systemfunktionalität in Teilfunktionalitäten, die von unterschiedlichen Entwicklern umgesetzt werden können.

3.2 Verwandte Arbeiten

Verwandte Ansätze, die sich wie die Nutzungsebene mit der Modellierung der Systemfunktionalität auf einem relativ hohen Abstraktionsniveau beschäftigen, finden sich in unterschiedlichen

[3] Ein Use-Case-Diagramm stellt Beziehungen zwischen Akteuren und Anwendungsfällen aus Sicht der Akteure/Stakeholder dar. Use-Cases sind vorwiegend ein Hilfsmittel zur Anforderungsermittlung und dienen weniger der Verhaltensbeschreibung und dem Systemdesign.

Bereichen. Wir gehen im Folgenden auf relevante Ansätze insbesondere aus den Bereichen der modellbasierten Entwicklung, des formalen Requirements Engineerings und der Feature Modellierung ein.

Focus Die Grundlage für die in dieser Arbeit vorgestellte Dienstmodellierung ist die FOCUS-Theorie, die in [BS01] von Broy und Stølen eingeführt wurde. Sie wurde ursprünglich zur logischen Beschreibung von verteilten Systemen entwickelt. FOCUS eignet sich jedoch nicht nur für die Spezifikation (totaler) Komponentensysteme, sondern auch zur Beschreibung partiellen Verhaltens (genannt Dienste [Bro05]). Das zentrale Konzept zur Systembeschreibung mit FOCUS ist der Nachrichtenstrom.

Modellbasierte Entwicklung Bei der modellbasierten Entwicklung werden die unterschiedlichen Aspekte eines Softwaresystems durch Modelle beschrieben. Eine wichtige und grundlegende Arbeit dabei ist die generative Software-Entwicklung, wie von Czarnecki et al. in [CE00] eingeführt. Charakteristisch für die generative Programmierung ist die automatische Erzeugung von Programmcode mit Hilfe eines Generators. Grundlage hierfür ist die Abstraktion häufig vorkommender Programmkonstrukte in formalen Modellen. Folglich ist dieser Ansatz – wie auch andere modellbasierte Ansätze (z. B. von Schätz [Sch06]) – im Unterschied zu dem hier vorgestellten Dienstmodell auf einem deutlich niedrigeren Abstraktionsgrad angesiedelt. Bekannte modellbasierte Ansätze sind daher zur Erstellung einer formalen Anforderungsspezifikation nicht geeignet.

Ein weiterer interessanter Ansatz ist in der Arbeit von Schätz [Sch05] zu finden. In diesem Ansatz werden logische (Design-)Komponenten aus Funktionen gebildet, die funktionale Anforderungen spezifizieren. Genau wie Dienste in unserem Ansatz, müssen diese Funktionen nicht notwendigerweise total sein. Im Gegensatz zu unserer Arbeit dürfen diese Funktionen keine gemeinsamen Output-Ports haben. Demzufolge ist es nicht möglich, das Verhalten an einem Port unter Berücksichtigung mehrerer Use Cases zu spezifizieren. Unser Ansatz hat diese Einschränkung nicht und ist deswegen für die Spezifikation von Anforderungen aus verschiedenen Sichten besser geeignet.

Modellbasiertes Requirements Engineering Modellbasierte Requirements Engineering Ansätze gewinnen in letzter Zeit zunehmend an Bedeutung. Ein Vertreter ist die modellbasierte Anforderungsanalyse und Systemdefinition mit dem Requirements Management-Werkzeug AUTORAID [Aut04]. In AUTORAID wird ein strukturierter Modellierungsansatz definiert, der die zielorientierte Erarbeitung und Abstimmung der verschiedenen Anforderungen mithilfe grundlegender Systemmodellierungssichten unterstützt. Mit Hilfe der entsprechenden iterativen Überprüfung und Konsolidierung entwickelter Anforderungs- und Designmodelle wird mit diesem Ansatz eine integrierte Anforderungs- und Systemspezifikation realisiert [GS07]. AUTORAID ist in das Spezifikationswerkzeug AUTOFOCUS integriert und nutzt dessen formal fundierte Systemsichten und grafische Beschreibungstechniken. Im Gegensatz zu dem hier vorgestellten Ansatz, werden Anforderungen in AutoRAID textuell erfasst und mit Design-Artefakten verknüpft. Dabei werden funktionale wie auch nicht-funktionale Anforderungen berücksichtigt. Im

Gegensatz dazu konzentriert sich die Nutzungsebene auf die *designunabhängige* Strukturierung, Formalisierung und Analyse der *funktionalen* Anforderungen aus Nutzersicht.

Formales Requirements Engineering Die in dieser Arbeit vorgestellten Konzepte zielen darauf ab, funktionale Anforderungen zu formalisieren und die Lücke zwischen den Anforderungen und dem Systementwurf zu schließen. Der Strukturierung von Anforderungen und dem Übergang von Anforderungen zu einer System-Architektur wurde in den letzten Jahren immer mehr Aufmerksamkeit gewidmet. Ein wichtiges Paradigma des modernen Requirements-Engineerings ist die zielorientierte Vorgehensweise, d. h. ausgehend von den Absichten der Stakeholder die Anforderungen abzuleiten. In [vL03] beschreibt van Lamsweerde den Übergang von zielorientierten Anforderungsspezifikationen über funktionale Spezifikationen zu abstrakten Architekturen und deren Verfeinerungen gemäß nicht-funktionaler Anforderungen. Auch im Ansatz von Medvidovic et al. [MGEB03] wird versucht, die Lücke zwischen Anforderungen und Design mit Hilfe weiterer Modelle zu schließen. Diese Herangehensweisen verwenden jedoch jeweils sehr unterschiedliche Modelle für die verschiedenen Abstraktionsebenen. Die Modelltransformation zwischen diesen Modellen ist nicht formal festgelegt und die formale Durchgängigkeit geht verloren. Im Gegensatz dazu ermöglicht der in dieser Arbeit vorgestellte Ansatz, funktionale Anforderungen mit Hilfe von Diensten formal zu modellieren und zu strukturieren. Das verwendete Dienstmodell setzt dabei auf der selben theoretischen Grundlage (FOCUS) wie die nachfolgende logische Architektur auf. Dies schafft die nötigen Voraussetzungen und erleichtert einen formalen Übergang von Anforderungsspezifikationen zur Architektur.

Feature Modellierung Mit FODA [KCH+90] und FORM [KKL+98] wurde eine Methode eingeführt, die sich beim Requirements Engineering an Features orientiert. Hier werden die Features eines Systems grafisch mittels Kompositions- und Querbeziehungen strukturiert. Einige weiterführende Ansätze beschäftigen sich mit der Formalisierung von Feature Bäumen mittels Grammatiken [BO92, CHE05] oder Prädikatenlogik [SZW05]. In all diesen Ansätzen liegt der Fokus jedoch auf der Formalisierung von Beziehungen zwischen den einzelnen Features. Beziehungen spielen auch auf der Nutzungsebene eine bedeutende Rolle, jedoch liegt der Schwerpunkt zunächst auf der formale Spezifikation des Verhaltens der eigentlichen Features. Dies wird von keinem der existierenden Ansätze ausreichend behandelt. "As a consequence, these approaches focus on the analysis of dependencies, however abstracting away from the causes for these dependencies" [Sch06].

Dienstbeziehungen Beziehungen zwischen Diensten werden bereits in [DGP+04] eingeführt. Hier wird bei der Definition der Dienste bereits der Zusammenhang zu anderen Diensten beschrieben. Diese Beziehungen sind jedoch rein informell und besitzen keinerlei formale Fundierung. Weiterhin wird keine Unterscheidung zwischen Querbeziehungen und hierarchischen Beziehungen getroffen.

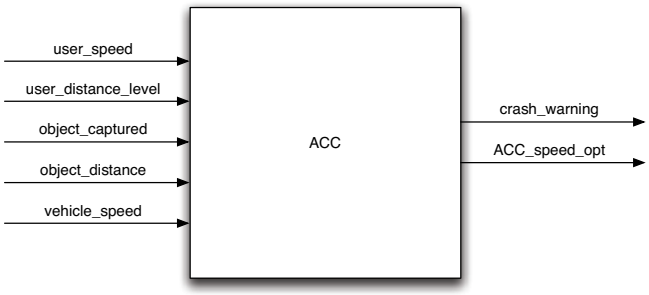

Abbildung 3: Syntaktische Schnittstelle des ACC-Systems

3.3 Hauptelemente

Die Hauptmodellierungselemente der Nutzungsebene sind Dienste, hierarchische Beziehungen, sowie Querbeziehungen. Die einzelnen Elemente werden im Folgenden beschrieben.

Dienste Das Hauptelement der Nutzungsebene sind *Dienste*. Das Konzept der Dienste ist in [Bro05] formal definiert. Ein Dienst entspricht einem Anwendungsfall und definiert eine Menge von Mustern von Interaktionen zwischen System und Umgebung. Jeder Dienst liefert eine formale Spezifikation einer (Teil-)Funktionalität (Kundenfunktion), die das System anbietet. Das Verhalten eines Dienstes wird hierbei durch die Spezifikation des Nachrichtenaustauschs mit der Umwelt festgelegt. Jeder Dienst wird daher beschrieben durch seine syntaktische Schnittstelle und seine semantische Schnittstelle. Die *syntaktische Schnittstelle* eines Dienstes ($I \blacktriangleright O$) umfasst eine Menge von getypten Eingabeports I und getypten Ausgabeports O, über die ein Dienst mit seiner Umwelt kommunizieren kann.

Die syntaktische Schnittstelle des ACC-Systems ist in Abbildung 3 dargestellt. Formal gilt:

$$I_{ACC} = \{user\_speed, user\_distance\_level, object\_captured, object\_distance, vehicle\_speed\}$$
$$O_{ACC} = \{crash\_warning, ACC\_speed\_opt\}$$

Der Benutzer kann das System durch Einstellen einer gültigen Wunschgeschwindigkeit und einer gültigen Abstandsstufe aktivieren (Ports `user_speed` und `user_distance_level`). Ferner empfängt das System von seinen Sensoren, ob ein Objekt in Fahrtrichtung erfasst ist (`object_detected`) und wie weit es entfernt ist (`object_distance`). Ein weiterer Sensor meldet die aktuelle Geschwindigkeit des eigenen Fahrzeugs (`vehicle_speed`). Auf Basis dieser Eingaben gibt das System die optimale Geschwindigkeit (`ACC_speed_opt`) und eine Warnung bei Unfallgefahr (`crash_warning`) aus.

Jedem Port wird dabei ein eindeutiger (logischer) Datentyp zugeordnet, wie z. B. natürliche Zahlen, Mengen, Aufzählungen etc., der festlegt welche Nachrichten über den entsprechenden Port ausgetauscht werden können.

In unserem Fallbeispiel kann am Port `user_distance_level` eine Abstandsstufe (`nah`, `mittel`

oder `fern`) oder nichts (ϵ) eingestellt werden:

$$type(user\_distance\_level) = \{nah, weit, fern\} \cup \{\epsilon\}.$$

Am Port `vehicle_speed` liegt die eigene Geschwindigkeit als ganzzahliger Wert im Bereich 0 bis 250 an:

$$type(vehicle\_speed) = \{0, \ldots, 250\}.$$

Die *semantische Schnittstelle* des Dienstes bestimmt das Verhalten des Dienstes. Sie gibt an, welche Ströme von Nachrichten verarbeitet werden können und welche Ausgaben abhängig von welchen Eingaben auftreten. Der Dienstespezifikation liegt hierbei ein diskretes Zeitmodell zugrunde. Es gibt eine Reihe von Spezifikationstechniken, um das I/O-Verhalten des Dienstes zu beschreiben, etwa MSCs, I/O-Automaten oder Assumption/Guarantee-Spezifikationen (siehe [BS01]).

MSCs und Automaten sind weit verbreitete Techniken zur Spezifikation von Verhalten. Um einen Eindruck zu vermitteln, wie eine Assumption/Guarantee-Spezifikation gemeint ist, geben wir beispielhaft die Assumption/Guarantee-Spezifikation des Teildienstes `Geschwindigkeits-regelung` an. Eine Assumption ist hierbei ein Prädikat, das festlegt, welche Inputströme vom System verarbeitet werden können. Eine Guarantee ist ein Prädikat über Inputs und Outputs, das gültige Tupel von Input- und Outputströmen charakterisiert. Um einen Eindruck von Assumption-Guarantee-Spezifikationen zu vermitteln ist in Abbildung 4 die A/G-Spezifikation des Teildienstes `Tempomat` aufgeführt.

$$A_{Tempomat}(x) \equiv \forall t : x[user\_speed](t) \in \{40, ..., 220\}$$
$$G_{Tempomat}(x, y) \equiv \forall t : y[ACC\_speed\_opt](t + 1) = x[user\_speed](t)$$

Abbildung 4: A/G-Spezifikation des Dienstes

Der Dienst `Tempomat` ist nur definiert, wenn die eingestellte Wunschgeschwindigkeit zwischen 40 und 220 km/h liegt. In diesem Fall gibt der Dienst die eingestellte Wunschgeschwindigkeit im nächsten Takt als optimale Geschwindigkeit aus.

Dieses Beispiel zeigt gut das modulare Vorgehen bei der Erstellung der Gesamtspezifikation. Die angeführte Teil-Spezifikation beschreibt hierbei nur das modulare Verhalten des Teildienstes `Tempomat`. Im Gesamtkontext muss dieses Verhalten nicht immer gültig sein, es wird z. B. durch die Abstandsregelung ersetzt, sobald ein Objekt in Fahrtrichtung erkannt wird. Diese Abhängigkeiten fließen jedoch nicht in die modulare Spezifikation ein, sondern werden explizit modelliert und erst bei der Kombination berücksichtigt. Dieses Vorgehen unterstützt eine höhere Wiederverwendung einzelner Teildienste: Der Dienst `Tempomat` kann ohne Anpassungen auch in einem System ohne `Abstandshaltung` verbaut werden.

Hierarchische Beziehungen Die modular spezifizierten Dienste werden hierarchisch zu größeren „zusammengesetzten“ Diensten zusammengefasst, bis man die Spezifikation des Gesamtsystems erhält (siehe auch [GHH07, HH08]). Die syntaktische Schnittstelle sowie das Verhalten der

zusammengesetzten Dienste ergibt sich dabei als Kombination der modularen Spezifikationen (allerdings unter Berücksichtigung der Querbeziehungen).

Die Schnittstelle des Dienstes `Regelung` ist folglich die Vereinigung der Schnittstellen von `Tempomat` und `Abstandshaltung`. Das Verhalten des Dienstes `Regelung` muss nicht explizit spezifiziert werden, sondern ist implizit als Kombination der Teildienste gegeben (unter Berücksichtigung der Querbeziehungen). Analog dazu vereinigt der Dienst `Kombinierte Abstands- und Geschwindigkeitsregelung` die Dienste `Aktivierung` und `Regelung` (syntaktisch und semantisch). Der Dienst `ACC` umfasst die Schnittstellen sämtlicher Teildienste. Die eigentliche Funktionalität von `ACC` ist nur in den Blättern des Baumes (`Aktivierung`, `Tempomat`, `Abstandshaltung` und `Unfallwarnung`) sowie in den Querbeziehungen explizit spezifiziert.

Im Umkehrschluss folgt daraus, dass, wie bereits erwähnt, jeder Teildienst eine Projektion aus dem Gesamtverhalten darstellt. Insbesondere stellt die Schnittstelle des Teildienstes eine Projektion aus der Gesamtschnittstelle da: Jeder Teildienst entspricht einer Teilschnittstelle des Gesamtsystems.

Querbeziehungen Im Dienstmodell können zusätzlich zu Teildienstbeziehungen noch Abhängigkeiten zwischen Diensten erfasst werden. Dabei beschränken wir uns auf *funktionale Abhängigkeiten*, die vom Benutzer wahrgenommen werden, d. h. auf Abhängigkeiten, die das I/O-Verhalten der einzelnen Dienste an der Gesamtsystemschnittstelle beeinflussen, und abstrahieren noch davon, wie diese konkret umgesetzt werden. Daher werden diese Beziehungen auch *Verhaltensbeziehungen* genannt. Verhaltensbeziehungen treten dabei zwischen Diensten auf, die in keiner Teildienst-Beziehung zueinander stehen. Innerhalb der Nutzungsebene kann die Verhaltensabstraktion wiederum in verschiedenen Detaillierungsstufen beschrieben werden.

Abbildung 2 enthält neben der hierarchischen Strukturierung weitere Querbeziehungen. So beeinflussen sich die Dienste `Tempomat` und `Abstandshaltung` gegenseitig: je nachdem, ob ein Objekt in Fahrtrichtung erkannt wurde, ist entweder der eine oder der andere Dienst aktiv. Der Dienst `Aktivierung`, der die Aktivierung des Systems durch den Benutzer und die Korrektheit der eingestellten Werte überprüft, beeinflusst (genauer `enabled` bzw. `disabled`) die zum Dienst `Regelung` zusammengefasste Geschwindigkeits- und Abstandsregelung.

3.4 Sichten

Die Nutzungsebene umfasst verschiedene Sichten auf das System: eine Struktursicht, eine Schnittstellen- bzw. Datensicht, eine Verhaltenssicht, eine Ablaufsicht sowie eine Abhängigkeitssicht. Die einzelnen Sichten werden im Folgenden kurz vorgestellt.

Struktursicht Auf der Nutzungsebene wird die Systemfunktionalität hierarchisch in Teildienste strukturiert. Die einzelnen Teildienste beschreiben jeweils einen Teilaspekt des I/O-Verhaltens an der Gesamtsystemschnittstelle. Diese Teilverhalten werden bei der hierarchischen Kombination zu zusammengesetzten Diensten überlagert (unter Berücksichtigung der Querbeziehungen).

Schnittstellensicht/Datensicht Jeder Dienst (atomar und zusammengesetzt) hat eine syntaktische Schnittstelle, die eine Menge von getypten Ein- bzw. Ausgabeports umfasst. Die syntaktische Schnittstelle eines zusammengesetzten Dienstes ist dabei implizit durch die Vereinigung der syntaktischen Schnittstellen der Teildienste gegeben. Als Datentypen werden den Ports dabei logische Typen zugeordnet, wie Natürliche Zahlen oder Mengen.

Verhaltenssicht Das Verhalten jedes Dienstes ist durch seine *semantische Schnittstelle* definiert. Da ein Dienst partiell sein kann, legt die semantische Schnittstelle zum Einen fest, welche Folgen von Eingaben verarbeitet werden können. Zum Anderen definiert die semantische Schnittstelle eine Abbildung zwischen gültigen Eingaben und Ausgaben. Als Spezifikationstechniken bieten sich z. B. MSCs, I/O-Automaten oder Assumption/Guarantee-Spezifikationen (siehe [BS01]) an.

Ablaufsicht (Interaktionssicht). Die Ablaufsicht beschreibt das Verhalten des Gesamtsystems, indem exemplarische Systemabläufe explizit aufgelistet werden. Sie liefert eine Menge von möglichen Systemläufen, für die die Dienste der Nutzungsebene spezifiziert sind. Ein Systemlauf ist dabei eine Abbildung einer Folge von Eingaben auf eine Folge von Ausgaben und stellt somit die Interaktion mit der Umwelt dar. Da die Dienste partiell sein können, handelt es sich bei der Menge um eine Teilmenge aller möglichen Abläufe.

Abhängigkeitssicht. In dieser Sicht wird nicht das Ein-/Ausgabeverhalten eines Systems beschrieben, sondern die gegenseitige Beeinflussung von Diensten. Diese Abhängigkeiten zwischen Diensten werden durch formal definierte Querbeziehungen beschrieben. Dadurch können über eine statische Abhängigkeitsanalyse versteckte Beeinflussungen (unerwünschte Feature Interaction) bereits auf der Nutzungsebene erkannt werden.

3.5 Gegenstand aktueller Forschung und offene Fragen

Die Nutzungsebene ist in vielen Ansätzen zur Architekturmodellierung gar nicht oder nur skizzenhaft vorhanden. Die Nutzungssicht ist jedoch von zentraler Bedeutung. Folgende weitergehende Fragen sind zu klären, um die Nutzungssicht für die Praxis einsetzbar zu machen:

- Ein wichtiger Vorteil der Formalisierung von Nutzungsanforderungen ist die Möglichkeit, eine Spezifikation bereits in einer sehr frühen Phase der Entwicklung automatisch auf ihre Konsistenz zu überprüfen und Abhängigkeiten (gewünschte und unerwünschte) zu erkennen. Um das zu erreichen, müssen Methoden zur Überprüfung der *Dienstevertäglichkeit* bereit gestellt werden. Zwei Dienste sind verträglich, falls sie in keinem Konflikt zueinander stehen.
- Ein weiterer aktueller Forschungsschwerpunkt ist die Analyse der Abhängigkeiten zwischen Diensten.
- Mit den in diesem Kapitel dargestellten Methoden können nur einzelne Produkte spezifiziert werden. Jedoch ist es oft sinnvoll und effizient eine ganze Familie von Produkten zu entwickeln. Um das zu ermöglichen, muss der dargestellte Ansatz um *Produktlinien-Methoden* erweitert werden (d. h. Begriffe wie Variabilität, Varianten, variantenabhängige Beziehungen müssen in unsere Spezifikationstechnik eingeführt werden).

- Eine weitere wichtige aber noch in Teilen offene Frage ist der *Übergang von der Nutzungsebene zu der logischen Architektur*. Dieser Übergang ist ein essentieller Bestandteil eines formalen modell-basierten Softwareentwicklungsprozesses.

4 Logische Architektur

4.1 Kurzbeschreibung und Ziele

Die logische Architektur schließt sich im Ebenenmodell direkt an die Nutzungsebene an. Sie stellt eine Realisierung der in der Nutzungsebene definierten Dienste (samt Querbeziehungen) durch ein Netz von kommunizierenden *logischen Komponenten* dar. Dabei werden die einzelnen Kundenfunktionen, wie sie in der Nutzungsebene spezifiziert wurden, in ein verhaltensäquivalentes Netz aus logischen Komponenten zerlegt. Vor diesem Hintergrund kann die logische Architektur als eine um abstrakte Kommunikationsbeziehungen angereicherte Konkretisierung der Modelle der Nutzungsebene betrachtet werden. Im Vergleich zur Nutzungsebene steht in der logischen Architektur nicht mehr die Formalisierung der an der Systemgrenze beobachtbaren Funktionalität im Vordergrund, sondern vielmehr die Strukturierung bzw. Aufteilung des Systems in logische, kommunizierende Einheiten, deren Gesamtverhalten das in der Nutzungsebene festgelegte Verhalten realisiert.

Die Strukturierung erfolgt dabei anhand verschiedenster Kriterien wie z.B. einer Aufteilung gemäß einer fachlichen Struktur des Produkts, anhand der organisatorischen Struktur im Unternehmen (Teamstruktur, vgl. Conway's Law) oder anhand anderer nicht-funktionalen Anforderungen (z.B. einer redundanten Auslegung einer Komponente eines sicherheitskritischen Systems aufgrund gesetzlicher Bestimmungen). Zu erwähnen ist jedoch, dass die Strukturierung in logische Komponenten prinzipiell unabhängig von der hierarchischen Zerlegung (Strukturierung) der Nutzungsfunktionalität, wie sie in der Nutzungsebene vorgenommen wurde, ist. In jedem Fall stellen die so strukturierten Komponenten modulare Einheiten dar, die per se von unterschiedlichen Entwicklern getrennt voneinander implementiert werden können, und nach Vorgabe des Modells der logische Architektur danach wieder zum gewünschten Gesamtsystem zusammengesetzt werden können.

Eine logische Komponente ist im weitesten Sinne eine Einheit, die an der Erbringung einer oder mehrerer Dienste der Nutzungsebene beteiligt ist. Im Allgemeinen besteht eine n-m-Beziehung zwischen Diensten aus der Nutzungsebene und Komponenten der logischen Architektur, die diese Dienste erbringen. Eine Komponente hat ähnlich wie ein Dienst ein syntaktisches und semantisches Interface, wodurch sowohl die syntaktische Schnittstelle als auch das Verhalten einer Komponente klar festgelegt sind. Komponenten können miteinander durch getypte, gerichtete Kanäle verbunden werden. Die Kommunikation zwischen Komponenten erfolgt ausschließlich über derartige Kanäle.

Ziele Im Vergleich zur Nutzungssicht werden in der logischen Architektur erstmals strukturelle Informationen modelliert, die eine logische Aufteilung des Systems in kommunizierende Komponenten beschreibt. Weiterhin ist das Verhalten einer Komponente im Vergleich zum

Verhalten eines Dienstes *total*. Das heißt, dass im Gegensatz zu einem Dienst gefordert wird, dass für jede Eingabe Systemreaktionen vorgesehen sind. Die logische Architektur stellt diejenige Abstraktionsebene der Ebenenhierarchie dar, in der das Verhalten des Systems *spätestens* simulierbar ist (d.h. dessen Ablauf anhand des auftretenden Datenflusses ausführbar ist).

Durch die totale Definition des Verhaltens stellt die logische Architektur ein Modell dar, das die Gesamtfunktionalität des Systems vollständig beschreibt, ohne jedoch technische Implementierungsentscheidungen vorwegzunehmen. Die logische Architektur ist damit in der Ebenenhierarchie das erste (funktional vollständige) Modell, das eine implementierungsneutrale Darstellung der funktionalen und logischen Anforderungen vornimmt.

Neben der bereits erwähnten Nutzung des Modells der logischen Architektur zur Simulation des Verhaltens (einschließlich des Kommunikationsflusses) vor der Erstellung einer Implementierung ist ein weiteres wesentliches Ziel der Gruppierung und Strukturierung von Funktionen zu fachlichen Komponenten die Optimierung der Wiederverwendung von bereits vorhandenen Komponenten. Dies bezieht sich auf eigens erstellte Komponenten, zugekaufte COTS[4]-Software, vor allem aber auch auf von Zulieferern erstellte Komponenten. Ziel auf Ebene der logischen Architektur ist es also auch, durch die explizite Modellierung von Strukturinformationen ein geeignetes Modell zu liefern, um — ggf. mit Hilfe von verhaltensinvarianten Umstrukturierungsmaßnahmen — die Erhöhung des Grads der Wiederverwendung bereits bestehender Komponenten zu maximieren.

4.2 Verwandte Arbeiten

Logische Architekturen im industriellen Kontext Die Modellierung einer logischen Architektur im obigen Sinne wird eher selten im industriellen Kontext vorgenommen: Architekturen (im Sinne einer Vernetzung von logischen, kommunizierenden Komponenten) beinhalten meist konkrete Komponenten, die bereits die jeweilige Implementierung darstellen und keine (echte) Abstraktion mehr davon sind. Bei Kommunikationskanälen wird ebenfalls meistens bereits die konkrete Implementierung vorweggenommen: Anstelle von logischen Kanälen findet man im Modell bereits Angaben über Verbindungsart, Bandbreite, Protokoll, etc. Insgesamt sind echte logische Architekturen, die aus tatsächlich logischen Komponenten und Verbindungen bestehen, im heutigen industriellen Entwicklungskontext kaum anzutreffen. Dementsprechend lassen auch die in der industriellen Praxis meist verwendeten Werkzeuge bereits eine zu detaillierte Modellierung zu. Da diese Werkzeuge jedoch durch diszipliniertes Anwenden ebenfalls eine Modellierung einer abstrakten, konzeptuellen Architektur zulassen, benennen und beschreiben wir im Folgenden doch die wichtigsten Vertreter:

MathWorks Simulink *Stateflow* und *ControlDesign* Simulink [The08a] ist eine domänenübergreifende Umgebung zur Simulation und Modellierung von dynamischen und eingebetteten Systemen.
Stateflow[The08b] und *ControlDesign*[The08c] stellen beide Erweiterungen von Simulink dar. *Stateflow* erweitert Simulink um eine Entwicklungsumgebung zur Erstellung von Zustandsmaschinen und Datenflussdiagrammen. Es werden dabei Sprachelemente zur Verfügung gestellt,

[4]Component Off-the-Shelf

die das Modellieren eingebetteter Systeme einschließlich Kontrollzuständen, Überwachungs- und Moduslogiken erlauben. Hierarchische und paralle Zustände samt Zustandsübergängen können in graphischer Form dargestellt werden. Weiterhin erlaubt *Stateflow* die Spezifikation mittels Wahrheitstabellen, Temporaler Logik und das Generieren von C-Code aus einem *Stateflow* Modell.
ControlDesign ergänzt Simulink und *Stateflow* und bietet eine verbesserte Modellierung des Kontrollflusses mit diversen Analyse- und Steuermöglichkeiten der Ablauflogik.

ASCET-SE und ASCET-MD In der Automobilindustrie werden häufig ASCET-Produkte zur Modellierung einer konzeptuellen, logischen Architektur verwendet. ASCET-SE [ETA08b] und ASCET-MD [ETA08c] stellen dabei die Speziallösungen in den Bereichen *Software Engineering* und *Modellierung und Design* dar und sollen die Entwicklung von Embedded Software erleichtern. Dazu gehört die Möglichkeit der Generierung von sicherem, optimiertem und effizientem C-Code für den jeweiligen Mikrocontroller des Produktionssteuergeräts ebenso wie die Konfiguration eines OSEK-Betriebssystems oder compilerspezifischer Optimierungsfunktionen. Im Werkzeug ASCET-MD können Modellkomponenten auf physikalischer Ebene anhand von Blockdiagrammen, Zustandsautomaten, Konditionstabellen, Booleschen Tabellen oder textuell in den konkreten Programmiersprachen ESDL oder C spezifiziert werden. Die zu Grunde liegende objektbasierte Modellarchitektur ermöglicht weiterhin das flexible Kombinieren von Modellkomponenten dieser unterschiedlichen Notationen und den hierarchischen Aufbau von Modellen. Somit kann eine Architektur kommunizierender Komponenten spezifiziert werden.

AutoFOCUS2 Zur modellbasierten Entwicklung verteilter, eingebetteter Systeme wurde am Lehrstuhl für Software & Systems Engineering das prototypische CASE-Tool AutoFocus2 [HSSS96] auf der Grundlage formaler Techniken entwickelt. AutoFocus2 basiert auf der Focus-Theorie (vgl. [BS01]) und wurde seit 1995 in mehreren Iterationen kontinuierlich entwickelt und erweitert. Es ist das Werkzeug, mit dem in der vorliegenden Arbeit auf der Ebene der logischen Architektur modelliert wird.
AutoFocus2 unterstützt die Systementwicklung mit integrierten, im wesentlichen graphischen Beschreibungstechniken, mit deren Hilfe sowohl unterschiedliche Sichten (Systemstruktursicht, Verhaltenssicht, Datensicht) als auch verschiedene Abstraktionsebenen eines Systems beschrieben werden. Abb. 5 zeigt einen beispielhaften Screenshot der Systemstruktursicht. Um konsistente und vollständige Beschreibungen sicherzustellen, bietet AutoFocus2 die Möglichkeit, Konsistenzbedingungen zu formulieren und Systembeschreibungen daraufhin zu überprüfen. Aus ausführbaren Spezifikationen können Prototypen des entwickelten Systems erzeugt werden und in einer Simulationsumgebung ausgeführt und visualisiert werden. Zur formalen Verifikation von Systemeigenschaften verfügt AutoFocus2 über Anbindungen an Modellprüfungswerkzeuge [BHS99].

Architecture Design Das Software Engineering Institute der Carnegie Mellon University (SEI) beschäftigt sich seit geraumer Zeit in vielen Arbeiten mit dem Entwurf von Architekturen. Dabei wird allerdings oft nicht unterschieden zwischen logischer und technischer Architektur.

Im Buch “Software architecture in practice” [BCK98] geben Bass et al. eine umfassende allgemeine Einführung in Softwarearchitektur, in der sie unter anderem auf einige ihrer Ansätze wie ABAS [KK99], ABD [BBC+00] und ADD [WBB+06] verweisen. Dabei sind alle drei Ansätze methodischer Art und beschreiben ein bestimmtes Vorgehen im Entwicklungsprozess, ohne jedoch konkrete Modellierungskonzepte vorzuschreiben. Im Gegensatz dazu liegt der Schwerpunkt in diesem Bericht genau auf den Modellierungskonzepten des vorgestellten Architekturmodells.
ABAS [KK99] steht für Attribute-Based Architectural Styles und überträgt die Idee von Design Patterns aus der Objektorientierung auf Architektur. Ein Architectural Style wird dabei mit Hilfe von Komponententypen und einer Topologie unter ihnen beschrieben; dazu werden den Styles über ein Modell die spezifischen Qualitätsattribute zugeordnet.
ABD [BBC+00] ist die Abkürzung für Architecture Based Design und beschreibt eine Methode, die mit Hilfe von funktionalen, Qualitäts-, und Business-Anforderungen eine konzeptuelle Software Architektur entwirft und dafür auf Software Templates zurückgreift.
ADD [WBB+06] steht für Attribute-Driven Design und beschreibt einen Designprozess für Architektur, der Qualitätsanforderungen in den Mittelpunkt stellt. Input sind funktionale Anforderungen, Design Constraints, Qualitätsanforderungen sowie sonstige Anforderungen oder Beschränkungen, die sich beispielsweise aufgrund von Legacy Systemen ergeben. Daraus werden Software Elemente, Rollen und Verantwortlichkeiten, und Eigenschaften und Beziehungen abgeleitet.
Die vorliegende Arbeit stellt Modellierungskonzepte vor, die in Kombination mit den methodischen Anleitungen der zitierten Werke des SEI angewandt werden können.

“4+1” View Model Die “Architectural Blueprints” [Kru95] beschreiben ein Modell für die Architektur Software-intensiver Systeme, basierend auf der Nutzung von mehreren konkurrierenden Sichten. Die Sichten erlauben die Berücksichtigung der Anliegen unterschiedlicher Stakeholder wie Endnutzer, Entwickler, Projektmanager usw. und die getrennte Behandlung von funktionalen und nicht-funktionalen Anforderungen. Die Sichten sind Logical View, Development View, Process View und Physical View und werden durch einen Szenario-getriebenen, iterativen Entwicklungsprozess entworfen.
Die Ebene der logischen Architektur in der vorliegenden Arbeit kann mit dem Development View von Kruchten verglichen werden.

4.3 Hauptelemente

Die Hauptbestandteile der logischen Architektur sind logische Komponenten, deren Ports mittels (Kommunikations-)Kanälen mit anderen Komponenten verbunden sind. Komponenten beschreiben ein totales Verhalten. Der Ausführungen des Verhaltens von Komponenten und der Kommunikation zwischen Komponenten liegt ein Zeitmodell mit einem systemweiten, kleinsten Taktbegriff zugrunde liegt. Im Folgenden werden die Hauptelemente jeweils genauer beschrieben.

Ports Ports sin die direkten “Berührungspunkte” einer Komponente zu ihrer Umwelt. Eine Komponente kann ausschließlich über ihre Ports bzw. die damit verbundenen Kanäle mit

ihrer Umwelt kommunizieren. Jeder Port besitzt sowohl einen Namen als auch einen Datentyp. Prinzipiell wird die Menge der Ports einer Komponente in zwei Teilmengen, die Menge der Eingabeports und die Menge der Ausgabeports, unterteilt. Ein Eingabeport kann jeweils nur mit einem Kanal verbunden sein, während mit einem Ausgabeport beliebig viele Kanäle verbunden sein können. In Abbildung 5 sind die Eingabeports durch weiße Kreise an den linken Rändern der Komponenten dargestellt. Der Ausgabeport *o_dst* (dargestellt durch einen schwarzen Kreis) ist ein Beispiel für einen Ausgabeport, der durch verschiedene Kanäle mit mehreren, unterschiedlichen Eingabeports verbunden ist.

Kanäle Kanäle spiegeln die direkte Kommunikation zwischen Komponenten wieder. Ein Kanal stellt eine abstrakte Form der Kommunikation dar, indem er zwei Kommunikationspartner (Ports von Komponenten) miteinander verbindet. Kanäle selbst sind typisiert, wobei der Typ eines Kanals mit den Typen der beteiligten Ports kompatibel sein muss[5]. Von der Information, wie genau diese Kommunikation später (in konkreteren Abstraktionsebenen) realisiert wird, wird auf der Ebene der logischen Architektur noch komplett abstrahiert. In der logischen Architektur handelt es sich ausschließlich um gerichtete Kanäle: Die Signale, die über einen Kanal gesendet werden, können also stets nur in eine Richtung fließen. Insbesondere darf *ein* Kanal immer nur einen Ausgabeport mit einem Eingabeport verbinden, wobei die Typen der Ports und des Kanals verträglich sein müssen. Damit ist auch die Richtung des Kanals eindeutig festgelegt. Semantisch verbirgt sich hinter einem Kanal eine potentiell unendliche, geordnete Sequenz von Elementen eines bestimmten Datentyps. Die Ordnung gibt eine grob-granulare Reihenfolge (Halbordnung) vor, in der die Elemente auf den Kanälen eintreffen bzw. gesendet werden. In Abbildung 5 ist z.B. durch *o_avail* ein solcher Kanal gegeben, der die Umwelt mit dem entsprechenden Eingabeport der Komponente *ControlACC* verbindet.

Logische Komponenten Die logische Architektur besteht aus einem Netzwerk von kommunizierenden, logischen Komponenten, die durch gerichtete, getypte Kanäle miteinander verbunden sind. Abbildung 5 zeigt das zum Dienstmodell des vorherigen Kapitels (siehe Abbildung 2) gehörige Modell einer solchen logischen Architektur.

Eine logische Komponente stellt eine Einheit dar, die eine Schnittstellen-, Struktur- und Verhaltensbeschreibung umfasst. Weiterhin kann jede Komponente ihrerseits wiederum aus einem Netzwerk von Komponenten bestehen. Somit kann auch das Gesamtsystem selbst als eine Komponente betrachtet werden. Formal besteht eine logische Komponente aus einer syntaktischen und semantischen Schnittstelle. Die *syntaktische Schnittstelle* beschreibt die Ein- und Ausgabemöglichkeiten der Komponenten mit ihrer Umwelt, indem sie alle Ein- bzw. Ausgabeports der Komponente samt des jeweiligen Datentyps der (logischen) Signale, die über die Ports kommuniziert werden können, erfasst. Durch die syntaktische Schnittstellendefinition – insbesondere die Festlegung von Port-Typen – ist dabei klar vorgegeben, wie die logischen Komponenten miteinander verbunden werden können. Bei den Datentypen handelt es sich dabei um logische Datentypen, wie sie in der Nutzungsebene bereits beschrieben wurden.

Am Beispiel des ACC-Systems heißt das, dass das Gesamtsystem in diesem Fall konkret aus den kommunizierenden Komponenten *ControlACC* und *ObjSpdRel* besteht. Die Benutzereingaben

[5]Für eine detailliertere Diskussion bzgl. Typen siehe [BS01].

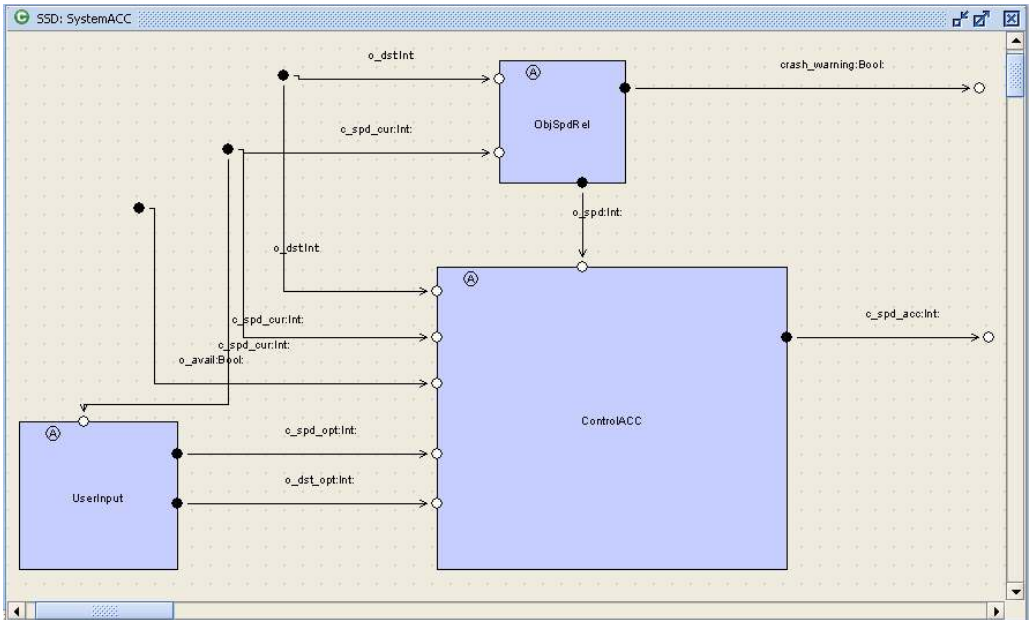

Abbildung 5: Modell einer logischen Architektur des ACC-Systems, modelliert im CASE-Tool AutoFocus2 [HSSS96].

bzw. Eingaben der Umwelt wurden explizit in einer weiteren logischen Komponente *UserInput* modelliert, die aber strenggenommen kein Teil des ACC-Systems ist. Die direkten Eingaben von *UserInput* sind als externe Eingaben zu verstehen, da hier der Benutzer als Teil (bzw. Komponente) des Systems modelliert wurde. Die Komponente *ObjSpdRel* errechnet aus der aktuellen Fahrzeuggeschwindigkeit *c_spd_cur* und der gemessenen Entfernung *o_dst* zu einem vorausfahrenden Objekt die Information, ob eine Unfallwarnmeldung (Kanal *crash_warning*) vorliegt bzw. berechnet die eigentliche absolute Objektgeschwindigkeit *o_spd*, die zur weiteren Berechnung an die logische Komponente *ControlACC* weitergeleitet wird. Die Berechnung der – aus Sicht des ACC-Systems – optimalen Geschwindigkeit (Kanal *c_spd_acc*) ist Aufgabe der Komponente *ControlACC*. Alle vorkommenden Kanäle haben einen der drei Typen *Int, Bool* oder *Dist*. Diese Datentypen stellen jeweils logische Typen dar, die noch von einer konkreten Implementierung des Datentyps abstrahieren.

Die syntaktische Schnittstelle (Black-Box) des *Gesamtsystems* ACC (einschließlich der Benutzereingaben) entspricht der Definition aus Abbildung 3: Den Eingaben *object_captured, object_distance* und *vehicle_speed* des Nutzungsmodells entsprechen die Kanäle *o_avail, o_dst* und *c_spd_cur* der logischen Architektur. Die Eingaben *user_speed* und *user_distance_level*, die sozusagen direkt vom Benutzer an das ACC-System vorgegeben werden können, werden durch die Kanäle *c_spd_opt* bzw. *o_dst_opt* in der logischen Architektur realisiert, die die Komponenten *UserInput* und *ControlACC* miteinander verbinden. Die beiden Ausgaben *crash_warning* und *ACC_speed_opt* werden durch die Kanäle *crash_warning* und *c_spd_ACC* modelliert.

Zwar tauchen in der logischen Architektur Kanäle auf (z.B. der Kanal *o_spd*), die nach außen

für einen Benutzer nicht sichtbar sind, die Black-Box Sicht des gesamten logischen Modells ändert sich jedoch wie bereits erwähnt gegenüber der syntaktischen Schnittstelle des Dienstmodells nicht (modulo Umbenennung der Kanäle).

Die *semantische Schnittstelle* einer logischen Komponente spezifiziert deren Verhalten, wobei im Gegensatz zu Diensten jede logische Komponente ein totales Verhalten besitzt, insbesondere also für jede (syntaktisch) mögliche Eingabe ein wohldefiniertes Ergebnis liefert. Das Verhalten einer logischen Komponente (bzw. die resultierenden Ausgaben) ergibt sich aus der aktuellen Eingabe und dem aktuellen, internen Zustand. Die zugrunde liegende Theorie ist FOCUS [BS01]. Das Verhalten wird zustandsorientiert durch Kontrollzustände, eingabe-abhängige Zustandsübergänge und interne Datenelemente beschrieben. Geeignete formale Beschreibungstechniken dafür sind *erweiterte I/O-Automaten*. Desweiteren werden zur Spezifikation der Interaktion zwischen Komponenten *Message Sequence Charts* (MSC) verwendet.

In unserem Beispiel des ACC-Systems ist das Verhalten der logischen Komponenten *ObjSpdRel* und *UserInput* jeweils durch entsprechende I/O-Automaten festgelegt, auf deren Darstellung wir jedoch an dieser Stelle verzichten. Das Verhalten der Komponente *ControlACC* ist durch

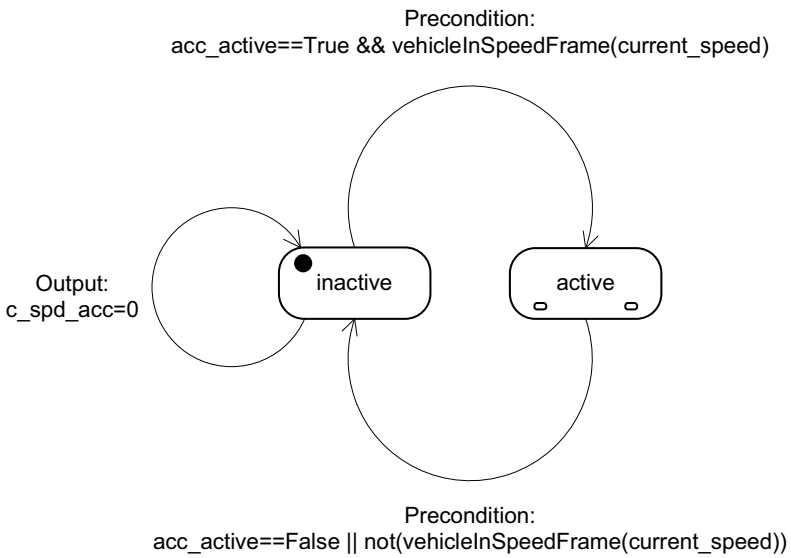

Abbildung 6: Vereinfachtes Verhalten der logischen Komponente *ControlACC*.

einen hierarchischen I/O-Automaten in Abbildung 6 dargestellt. Der I/O-Automat zeigt dabei die Hauptzustände des Verhaltens und wurde zur besseren Übersichtlichkeit gegenüber dem realen, simulierbaren AutoFOCUS2-Modell leicht vereinfacht. Die einzelnen (hierarchischen) Zustände sind wiederum selbst durch eigene I/O-Automaten realisiert.

Neben der "direkten" Repräsentation einer logischen Komponenten durch einen I/O-Automaten kann eine Komponente jedoch ihrerseits wiederrum durch ein Netzwerk von (Sub-) Komponenten realisiert werden. Dies erlaubt eine hierarchische Untergliederung einer Komponenten in eine eigene Sub-Komponenten-Struktur, was implizit auch eine Aufteilung des Verhaltens einer

Komponente in kleinere, modulare Teile erlaubt. Für die Entwicklung großer, komplexer Systeme, z.B. eines modernen Oberklasseautomobils, ist eine derartige hierarchische Untergliederung essentiell und Grundlage für eine effiziente Entwicklung.

Für das Beispiel des betrachteten ACC-Systems zeigt Abbildung 7 die hierarchische Verfeinerung des Zustands *active* aus Abbildung 6, wobei die beiden Abbildungen so zu verstehen sind, dass beim Erreichen des Zustands *active* aus Abbildung 6 das durch den Automaten aus Abbildung 7 modellierte Verhalten ausgeführt wird. Hinter dem Zustand *object_availabe* aus

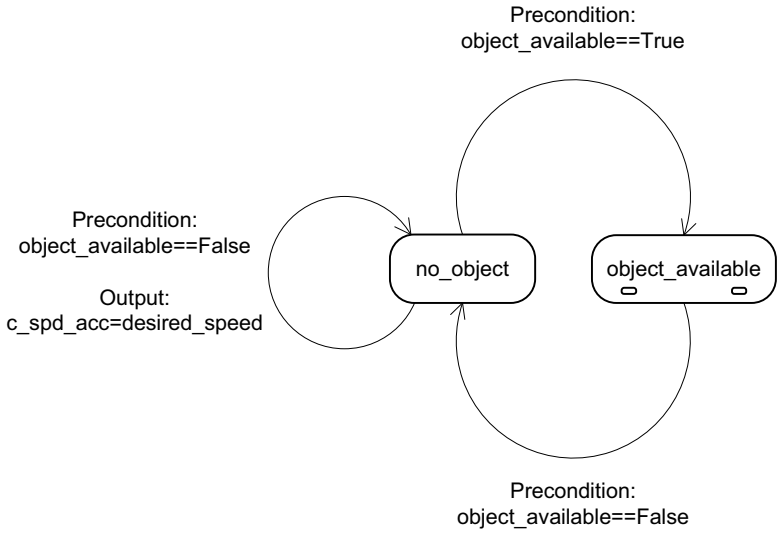

Abbildung 7: Verhalten, das hinter dem Zustand *active* aus Abb. 6 hinterlegt ist.

Abbildung 7 ist selbst wieder ein Automat hinterlegt, der die eigentliche Regelungssteuerung modelliert. Zusammen ergibt sich dadurch das Verhalten des gesamten ACC-Systems.

Zeit Der logischen Architektur liegt ein globales, jedoch flexibles Zeitmodell zugrunde. Flexibel heißt in diesem Zusammenhang, dass bei einer hierarchisch untergliederten Komponente die Zeittaktung jeder direkten Sub-Komponente jeweils eine Verfeinerung des Zeittaktes ihrer Super-Komponente ist. Dadurch wird die hierarchische Strukturierung von Komponenten explizit im Zeitmodell mitbetrachtet. Obwohl also die Dauer eines Zeittaktes für verschiedene Komponenten unterschiedliche lang sein kann, kann trotzdem für das Gesamtsystem ein "kleinster gemeinsamer" Systemtakt errechnet werden, auf den sich die individuellen Takte aller Komponenten des Gesamtsystems herunterbrechen lassen. Daher auch der Begriff "globales" Zeitmodell.

4.4 Sichten

Im Folgenden werden kurz die verschiedenen Sichten auf die logische Architektur vorgestellt.

Struktursicht Die Struktursicht gibt eine Datenflußstruktur wieder, indem sie die Komponenten samt ihren syntaktischen Schnittstellen und den konkreten Verbindungen zwischen den Komponenten durch Kanäle darstellt. Für hierarchisch untergliederte Komponenten wird sowohl die interne Struktur der direkten Unterkomponenten samt den Kanälen der Unterstruktur, als auch die Weiterverteilung von Signalen der Oberkomponenten an die Schnittstellen der Unterkomponenten mittels entsprechenden (internen) Kanälen aufgezeigt. Abbildung 5 gibt konkret die Struktursicht des ACC-Beispiels wieder, wobei der Kanal *o_spd* ein Beispiel für einen internen Kanal ist, der an der Systemschnittstelle nicht beobachtbar ist.

Schnittstellensicht/Datensicht Diese Sicht liefert die syntaktische Schnittstelle einer Komponente mit den Namen aller Ein- und Ausgänge (Ports) samt den jeweiligen (logischen) Datentypen, wie z.B. Natürliche Zahlen, Mengen, Aufzählungen, usw. . Desweiteren sind die Namen und Typen aller Kanäle sichtbar.

Verhaltenssicht In der Verhaltenssicht ist das Verhalten einzelner Komponenten dargestellt. Für Komponenten, die ihrerseits nicht mehr weiter in Sub-Komponenten unterteilt sind, ist in der Verhaltenssicht direkt der zugehörige I/O-Automat hinterlegt. So ist z.B. das Verhalten der Komponente *ControlACC* in Abbildung 6 als I/O-Automat gegeben. Für hierarchisch untergliederte Komponenten ergibt sich das entsprechende Verhalten aus den Einzel-Verhalten der Sub-Komponenten und der Art und Weise, wie diese miteinander interagieren. Das Interaktionsverhalten der Subkomponenten kann z.B. durch MSCs (Message Sequence Charts) spezifiziert werden. Generell muss die Beschreibungstechnik für das Verhalten eine operationelle Semantik besitzen, so dass das Verhalten ausführbar ist. Insbesondere sind deskriptive Verhaltensbeschreibungen (z.B. mittels Assumption/Guarantee-Stil) auf Ebene der logischen Architektur durch äquivalente operationelle Beschreibungstechniken auszudrücken.

Ablaufsicht Die Ablaufsicht liefert die Menge *aller* möglichen Systemläufe des Systems. Ein Systemlauf ist dabei die Abbildung einer Folge von Eingabesignalen auf eine Folge von entsprechenden Ausgabesignalen. Durch die Vollständigkeit unterscheidet sich die Ablaufsicht auf Ebene der logischen Architektur von der Ablaufsicht auf der Nutzungsebene, in der nur eine Teilmenge aller Systemläufe beschrieben wird.

4.5 Gegenstand aktueller Forschung und offene Fragen

Der logischen Architektur kommt als Bindeglied für den Übergang hin zu implementationsspezifischen Modellen eine zentrale Rolle im Entwicklungsprozess zu. Gegenstand aktueller und zukünftiger Forschung diesbezüglich ist:

- Methodischer sowie modell-technischer Übergang zwischen Dienstemodell und logischer Architektur. Dabei werden Umstrukturierungsmechanismen untersucht, die es erlauben, Dienste – und damit Funktionalität – beliebig zu strukturieren, um die Funktionalität somit besser auf eine Komponentenstruktur abbilden zu können. Die Umstrukturierung kann durch Einsatz von Model Checking Techniken praktisch anwendbar vorgenommen

werden. Die theoretische Grundlage diesbezüglich ist bereits vorhanden und in [GLS08] beschrieben. Arbeiten an einer praktischen Umsetzung in Form eines Tools stellen die nächsten Schritte in diesem Bereich dar.

- Partitionierung der logischen Komponenten und Abbildung auf Softwarefunktionen der Softwarearchitektur;
- Nachweis der Korrektheit der Modellübergänge zwischen den Ebenen.

5 Technische Architektur

5.1 Kurzbeschreibung und Ziele

Die technische Architektur dient als „Zielmodell“ für die modellbasierte Entwicklung von Software für eingebettete Systeme, d.h. sie stellt die Ebene mit dem niedrigsten Abstraktionsgrad im Architekturmodell dar.

Der Abstraktionsgrad dieser Ebene ist dabei so gewählt, dass

- der Nachweis der Erfüllung von Echtzeitanforderungen durch statische Analysen möglich ist,
- zumindest Teile des Verhaltens simulierbar sind (dynamische Analyse),
- explizit erkennbar ist, an welchen Stellen neues Verhalten (aufgrund des Abstraktionsgrades der logischen Architektur sind auf der Ebene der logischen Architektur bewusst bestimmte Verhaltensanteile (z.B. Busprotokolle) nicht oder idealisiert dargestellt worden) gegenüber der logischen Architektur hinzugekommen ist und wie es sich auf das Verhalten der logischen Architektur auswirkt,
- die Modellkonzepte jede Ausgestaltung der Plattform (zeitgesteuert, ereignisgesteuert oder Mischformen daraus) repräsentieren können,
- alle weiteren Schritte, die von dem Modell der technischen Architektur zur tatsächlichen Realisierung vorgenommen werden müssen, nur noch softwaretechnischer Natur sind, d.h. keinen Einfluss mehr auf das zeitliche Verhalten (z.B. Codegenerierung darf die Ausführungsreihenfolge nicht verändern) haben.

Mit den Modellen auf dieser Ebene wird das in der logischen Architektur spezifizierte Verhalten in ein System, das aus Hardware und Software besteht, umgesetzt. Dazu wird das Modell auf der Ebene der logischen Architektur zunächst in zwei Teile aufgeteilt: In einem Teil befinden sich die Anteile des Modells, die in Hardware realisiert werden sollen, und im anderen Teil befinden sich die applikationsspezifischen Anteile, die in Software realisiert werden sollen. Dieser applikationsspezifischen Teil wird wiederum in Partitionen (= Cluster) aufgeteilt, die dann auf den entsprechenden Komponenten der Plattform ausgeführt werden. Mit Plattform wird sowohl Hardware als auch Software bezeichnet, die zur Integration der Applikation verwendet werden soll (z.B. Treibersoftware, Middleware, ...). Zur Plattform gehören auch die bereits in Hardware realisierten Anteile des Modells der logischen Architektur.

Eingebettete Systeme im Automobil bestehen aus mehreren Steuergeräten, die meist durch Busse miteinander verbunden sind. Ebenfalls mit diesen Steuergeräten über Busse oder einfache Verbindungen verbunden sind Ein- und Ausgabeeinheiten (z.B. Sensoren und Aktoren), welche Informationen aus der Umgebung (z.B. physikalische Größen oder Benutzereingaben) einlesen bzw. die Umgebung (z.B. über Mechanik) beeinflussen oder den Benutzer über Zustände des Automobils informieren.

Auf den Steuergeräten befindet sich meist ein (manchmal auch mehrere) Mikrocontroller, eventuell zusätzlicher Speicher und andere Hardwarebauteile, die z.B. die Ansteuerung der Busse oder der Sensoren und Aktoren übernehmen. Auf dem Mikrocontroller selbst befindet sich das eigentliche Programm, d.h. der Teil der Funktionalität des Systems, der in Software realisiert wurde. Dieses Programm bestimmt weitgehend das Verhalten des Mikrocontrollers. Bei den anderen Hardwarebauteilen ist das Verhalten fest verdrahtet und es kann meist nur gering oder gar nicht beeinflusst werden, wie es oft bei Sensoren und Aktoren der Fall ist.

Die technische Ebene bietet für die Verhaltensbeschreibung aller Hardwarekomponenten (inklusive der auf ihr laufenden Software) ein einheitliches Konzept an, mit dem das gesamte Systemverhalten entsprechend abstrakt dargestellt werden kann. Sie besteht aus der Hardwaresicht, welche im Wesentlichen die Hardwaretopologie darstellt, und aus der Laufzeitsicht, die das Verhalten des gesamten Systems als Komposition der Verhaltensbeschreibungen der einzelnen Hardwarekomponenten angibt.

Zunächst sollen die Konzepte kurz an einem Beispiel eingeführt werden, bevor sie in den darauf folgenden Abschnitten genauer erläutert werden.

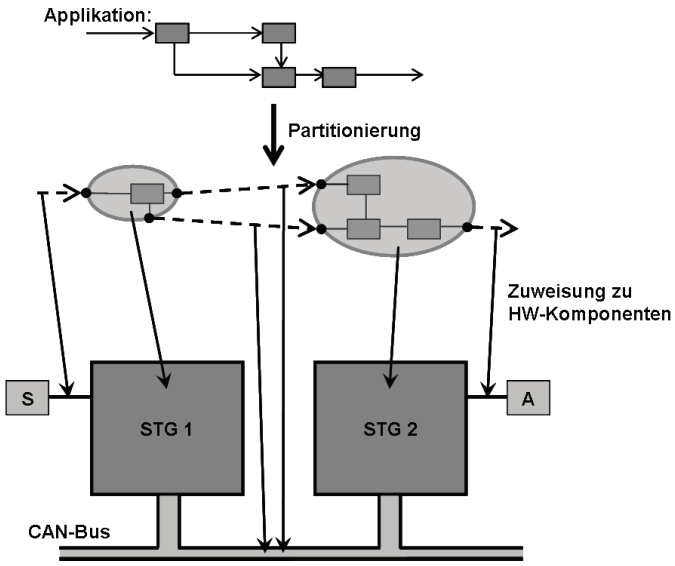

Abbildung 8: Applikation, Partitionierung und Plattform des einführenden Beispiels

Einführendes Beispiel Wie in Abbildung 8 dargestellt, soll eine Applikation, die bereits in zwei kommunizierende Cluster[6] partitioniert worden ist, auf eine Plattform gebracht werden, deren Hardware aus zwei Steuergeräten besteht, die mit einem ereignisgesteuerten Bus (hier CAN-Bus) verbunden sind. Das eine Steuergerät ist über eine einfache Verbindung mit einem Sensor, das andere mit einem Aktor verbunden. Die Cluster sollen auf jeweils einen Mikrocontroller integriert werden, die Kommunikationsbeziehungen zur Umwelt werden an den Sensor bzw. Aktor angeknüpft und die Kommunikationsbeziehungen zwischen den beiden Clustern untereinander werden über den CAN-Bus realisiert.

Die resultierende technische Architektur könnte dabei so aussehen, wie in der Abbildung 9 schematisch dargestellt. Die Hardwaresicht besteht aus sechs Hardwarekomponenten, die miteinander verbunden sind. Die Hardwarekomponenten fassen jeweils mehrere Hardwarebauteile zusammen. In der Hardwarekomponente 1 werden in diesem Beispiel alle Hardwarebauteile zusammengefasst, die den Sensor und seine Verbindung zum Mikrocontroller betreffen. In den Hardwarekomponenten 2 und 4 sind von den jeweiligen Steuergeräten alle Hardwarebauteile zusammengefasst, die für die Ausführung der Software notwendig sind. In Hardwarekomponente 3 sind alle Hardwarebauteile aggregiert, die für die Übermittlung der Daten von STG1 zu STG2 verantwortlich sind: Das sind die entsprechenden Peripheriekomponenten der Mikrocontroller, Treiberbausteine und das Kabel selbst. In Hardwarekomponente 5 befinden sich alle Hardwarebauteile, die für die Verbindung von STG2 mit dem Aktor zuständig sind und in 6 sind alle Hardwarebauteile, die für den Aktor verantwortlich sind, zusammengefasst. Im Gegensatz zur Hardwarekomponte 1 wurde hier die Verbindung separat in einer eigenen Hardwarekomponente modelliert. Diese separate Modellierung kann auch bei einfachen Verbindungen Sinn machen, wenn z.B. die Art und Weise der Verbindung noch nicht genau festgelegt ist, und daher möglichst modular modelliert werden soll.

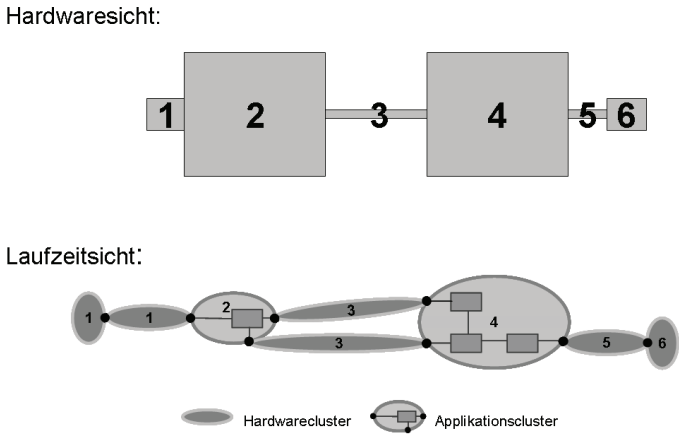

Abbildung 9: Hardwaresicht und Laufzeitsicht des einführenden Beispiels

Das Verhalten der Hardwarekomponenten wird durch die Elemente der Laufzeitsicht (mit den gleichen Nummern wie die entsprechende Hardwarekomponente versehen) repräsentiert. In

[6]Ein Cluster fasst Einheiten der Logischen Architektur zusammen; mehr dazu weiter unten in 5.3

diesem Beispiel ist das Verhalten der Mikrocontroller (Komponenten 2 und 4 in Abbildung 9) besonders stark vereinfacht durch lediglich jeweils einen Applikationscluster dargestellt. Diese Applikationscluster sind identisch mit den Clustern, die aus den Modellen der logischen Architektur gewonnen wurden, d.h. das Verhalten innerhalb der Cluster entspricht genau dem Verhalten der entsprechenden Modelle der logischen Architektur. Das Verhalten der anderen Hardwarekomponenten wird durch ein oder mehrere Hardwarecluster beschrieben. Das Verhalten, das durch die Hardwarecluster repräsentiert wird, ist hier gegenüber dem in der logischen Architektur modellierten Verhalten neu hinzugekommen, so ist es in diesem Beispiel in der Applikation noch nicht modelliert worden und muss nun in der technischen Architektur berücksichtigt und im entsprechenden Abstraktionsgrad abgebildet werden. Im einfachsten Fall kann dieses Verhalten die Identität mit einer möglichen Verzögerungszeit sein, es kann jedoch auch komplexer gestaltet sein und das Verhalten der Hardwarekomponente realitätsnah nachbilden. Die Stellen, an denen eine Kommunikation zwischen den einzelnen Clustern stattfindet, ist in der Abbildung 9 lediglich schematisch mit Hilfe eines schwarz ausgefüllten Kreises dargestellt, der die einzelnen Cluster verbindet.

5.2 Verwandte Arbeiten

Es gibt einige Ansätze, die - wie die technische Architektur - das Verhalten von Software und Hardware integriert betrachten. Hier sind vor allem das Hardware/Software Co-Design und SystemC zu nennen. Giotto/TDL und das Verisoft-Taskmodell sind Ansätze, die Ähnlichkeiten zur Laufzeitsicht der technischen Architektur aufweisen. Die Gemeinsamkeiten und Unterschiede der technischen Ebene des Gesamtarchitekturmodells zu diesen Ansätzen werden in den folgenden Abschnitten dargestellt.

Giotto/TDL Giotto [HHK01] und die daraus hervorgegangene TDL (Timing Description Language) [Tem04] sind Programmiersprachen, die Plattformunabhängigkeit durch Einführung einer Abstraktion unterstützen. Sie bieten eine Entwurfsmethodik an, die erlaubt Zeitaspekte unabhängig von der Funktionalität zu spezifizieren. Die Funktionalität wird in Giotto in Form von Tasks und Modes bereitgestellt. Das Verhalten einer Task wird separat in einer Programmiersprache spezifiziert. Die Modes bestehen aus einer festen Anzahl von Tasks, die innerhalb dieses Modes ausgeführt werden müssen. Zu einem beliebigen Zeitpunkt kann immer nur ein Mode aktiv sein. Durch Mode-Switches kann zwischen den unterschiedlichen Modes umgeschaltet werden. Das Zeitverhalten wird zu jeder Task in Form einer LET (logical execution time), der logischen Ausführungszeit, angegeben. Aus der Funktionalität (Tasks und Modes) und dem spezifizierten Zeitverhalten (LET) wird vom Giotto-Compiler ein Schedule für eine zeitgesteuerte Plattform generiert, der garantiert, dass für jede Task in jedem Mode diese LET genau eingehalten wird, d.h. unabhängig von der tatsächlichen Ausführungszeit einer Task stehen die Ergebnisse der Berechnung immer genau nach Ablauf der LET zur Verfügung. Damit ist das zeitliche Verhalten genau vorhersagbar.

Giotto/TDL setzen die Existenz von Tasks und feste Aufruffrequenzen für diese Tasks voraus. Diese Voraussetzungen machen bei Funktionsanteilen, die Regelungssysteme implementieren, durchaus Sinn, da diese Informationen aus dem Entwurf des zur Anwendung kommenden diskretisierten Regelungsalgorithmus bereits bekannt sind. Bei Funktionsanteilen, die in einer

bestimmten Zeit auf Ereignisse, die in der Umwelt aufgetreten sind, reagieren müssen, ist es jedoch nicht sinnvoll, eine Frequenz anzugeben. In einem ersten naiven Ansatz muss diese Frequenz so klein gewählt werden, dass die entsprechenden Echtzeitanforderungen zu jedem möglichen Zeitpunkt erfüllt werden können. Zusätzlich wird die WCET (worst case execution time) so gewählt, als ob bei jedem Aufruf der Task, das entsprechende Ereignis vorliegt. Damit wird viel Rechenzeit verschwendet. Das führt zu einem komplexeren Ansatz, der mehrere Funktionalitäten in eine Giotto/TDL-Task verpackt. Dieser Schritt wird jedoch nicht von Giotto/TDL unterstützt. Er kann jedoch von der hier vorgestellten technische Architektur unterstützt werden, d.h. Giotto/TDL kann als eine mögliche Plattform für die technische Architektur angesehen werden.

Verisoft Taskmodell In [BGH+06] wird ein Taskkonzept auf Basis eines zeitgesteuerten Bussystems (wie z.B. FlexRay) und eines zeitgesteuerten Betriebssystems (OSEKtime konformes Betriebssystem) vorgestellt. Es stammt aus dem Projekt Verisoft [Ver07] (Teilprojekt 6: Automotive), das sich mit der Verifikation eines Automobilelektroniksystems beschäftigt. Die Tasks in diesem Konzept werden aus einem AutoFocus2-Modell [Aut07a] durch Zusammenfassen von AutoFocus2-Komponenten und Bilden des entsprechenden Produktautomaten erzeugt. Eine Task wird immer dann ausgeführt, wenn entweder alle Werte an den Eingängen vorhanden sind (AND-Eingänge) oder wenn mindestens ein Wert aktualisiert worden ist (OR-Eingänge), je nachdem welche Aktivierung spezifiziert worden ist. Innerhalb einer Taskausführung wird immer der entsprechende Automat solange ausgeführt, bis eine neue Eingabe benötigt wird, d.h. bis ein so genannter Idle-Zustand erreicht wird. Dieses Taskmodell und deren Ausführung kann wiederum im Werkzeug AutoFocus2 dargestellt werden.

Das AutoFocus2-Taskmodell ist speziell auf zeitgesteuerte Plattformen ausgerichtet und kann für diese Art von Plattformen eine Konkretisierung der in diesem Bericht vorgestellten Laufzeitsicht der technischen Architektur darstellen, d.h. es kann mit den Konzepten der Laufzeitsicht repräsentiert werden.

Hardware/Software Co-Design Unter Hardware/Software Codesign (HSCD) versteht man den gleichzeitigen und verzahnten Entwurf von Hardware- und Softwareteilen eines Systems [KAJW95]. Unter diesem Begriff sind alle Verfahren zusammengefasst, die die Entwicklung eines Systems, das aus Hardware- und Softwareanteilen besteht, ganzheitlich betrachten. Dazu gehören Verfahren zum gemeinsamen, integrierenden und systematischen Entwurf von Hardware und Software wie z.B. Spezifikationstechniken, Allokation, Partitionierung und Software- und Hardware-Synthese. HSCD bringt die drei Systementwurfs-Disziplinen Modellierung auf Systemebene, Hardwareentwurf und Softwareentwurf zusammen.

Im Unterschied zu den HSCD-Verfahren liegt bei der technischen Architektur und beim Übergang zu dieser der modellbasierte Entwurf der Software im Vordergrund. Es werden auf der Ebene der technischen Architektur zwar das Verhalten der Hardware und der Software mit einheitlichen Beschreibungsmitteln beschrieben, jedoch ist nicht der Entwurf der Hardware Ziel dieses Ansatzes, sondern es werden lediglich ihre Eigenschaften, die Auswirkung auf das gesamte Systemverhalten haben, modelliert, um eine zur Hardware passende Software zu erhalten. Auch steht bei den HSCD-Verfahren eine möglichst gute Aufteilung des gesamten Systems in

Hardware und Software im Vordergrund. Diese Aufteilung wird auf der Ebene der technischen Architektur bereits als vorgegeben angenommen und es wird nach einer anforderungskonformen Variante der Softwareverteilung gesucht. Dazu wird auf der Ebene der technischen Architektur der Aspekt der Veränderung des Verhaltens der Softwareanteile besonders in den Vordergrund geschoben. Eine Veränderung des Verhaltens der Softwareanteile ergibt sich vor allem aus dem an die Hardware angepassten Ausführungsmodell, welches sich von dem der logischen Architektur deutlich im Hinblick auf die Synchronität der Ausführung unterscheiden kann. Des Weiteren werden im Gebiet des HSCD Funktionalitäten in Form von ablauffähigen Hochsprachen (z.B. C, Java, usw.) beschrieben. Im vorliegenden Ansatz liegt die Funktionalität in Form von Modellen vor.

SystemC SystemC [OSC08b] ist eine auf C++ basierende Systembeschreibungssprache. Sie erweitert C++ um eine (Klassen-)Bibliothek und bietet sowohl Sprachkonstrukte zur Beschreibung und Synthese von Register-Transfer-Logik (RTL) als auch Sprachkonstrukte, die erlauben Hardware- und Softwarekomponenten und deren Kommunikationsbeziehungen auch auf einer höheren Abstraktionsstufe als RTL in einer gemeinsamen Programmiersprache zu beschreiben. SystemC wird von der Open SystemC Initiative (OSCI) betreut und ist als IEEE-Standard 1666-2005 (The IEEE Standard SystemC Language Reference Manual [OSC08a]) veröffentlicht worden.

Mit SystemC kann zwar das Verhalten von Hardware und Software gemeinsam beschrieben werden, es fehlt jedoch die konzeptionelle Aufteilung der Konstrukte in Funktionalität und Plattform, auf die die Funktionalität integriert werden soll, und damit auch die Möglichkeit, explizit die Auswirkung dieser Integration auf die ursprüngliche Funktionalität zu beschreiben.

5.3 Hauptelemente

Die wichtigsten Elemente der technischen Architektur sind *Hardwarekomponenten* und *Cluster*. In der Laufzeitsicht gibt es noch weitere Elemente, nämlich *Events*, *Buffer* und *Timer*. Um das Dokument aber so knapp wie möglich zu halten, werden diese hier nur soweit erläutert, wie es im Zusammenhang mit dem Element *Cluster* notwendig ist.

Hardwarekomponente Eine Hardwarekomponente fasst die realen Hardwarebausteine der Plattform geeignet zusammen und repräsentiert so einen Teil der gesamten Hardware. Die Hardwarekomponenten können eingeteilt werden in Ein- und Ausgabeeinheiten (Sensoren, Aktoren und Bedieneinheiten), Verbindungen bzw. Busse und ausführende Einheiten wie Mikrocontroller inkl. Speicher. Welche Hardwarebausteine jeweils zusammengefasst werden, ist nicht eindeutig, d.h. es ist frei wählbar, welche Hardwarebausteine zu einer Hardwarekomponente aggregiert werden. Da jede Hardwarekomponente ein eigenes Ausführungsmodell (siehe unten) besitzt, ist hier zu entscheiden, welche echt parallelen und für das gesamte Systemverhalten bedeutende Verhaltensabläufe der einzelnen Hardwarebausteine repräsentiert werden sollen und daher nicht in einer Hardwarekomponente zusammengefasst werden sollen. Mit dieser Sichtweise kann auch ein Multicore-Prozessor repräsentiert werden, indem jeder Core als eine

eigene Hardwarekomponente modelliert wird. Die Hardwarekomponenten sind miteinander verbunden und bilden die Hardwaretopologie der Plattform. Verbundene Hardwarekomponenten können direkt miteinander kommunizieren, d.h sie können gegenseitig auf bestimmte Elemente der Laufzeitsicht zugreifen. Hardwarekomponenten sind keine hierarchischen Gebilde, d.h. sie enthalten selbst keine Hardwarekomponenten. Eine Hardwarekomponente dient als Container für die Elemente der Laufzeitsicht, d.h. das Verhalten wird mit Hilfe von Clustern und den anderen Elementen der Laufzeitsicht im geeigneten Abstraktionsgrad abgebildet. Um zu einer solchen Verhaltensbeschreibung zu kommen, gibt es grundsätzlich zwei Möglichkeiten: Falls das Verhalten schon in der logischen Architektur modelliert worden ist, kann es nun - erweitert um bestimmte Laufzeiteigenschaften (z.B. Durchlaufzeiten) - den entsprechenden Hardwarekomponenten zugeteilt werden. Falls nicht, muss eine abstrakte Verhaltensbeschreibung angegeben werden. Diese kann z.B. aus den existierenden Spezifikationen der entsprechenden Hardwarebausteine gewonnen werden.

Cluster Der Cluster ist das zentrale Element der Laufzeitsicht. Er kapselt ein Stück Funktionalität. Diese wird modelliert durch die entsprechenden Ausschnitte aus der logischen Architektur mit den Konzepten der logischen Architektur. Aus Betriebssystemsicht kann ein Cluster in etwa einer Task gleich gesetzt werden. Cluster sind nicht hierarchisch, d.h. sie enthalten selbst keine Cluster. Ein Cluster ist immer einer Hardwarekomponente zugeordnet und bestimmt zusammen mit anderen Clustern und den anderen Elementen der Laufzeitsicht (*Buffer*, *Events* und *Timer*) das Verhalten dieser Hardwarekomponente. Die direkte Verbindung (so wie sie in der logischen Architektur durch Kanäle gegeben ist) wird zwischen den Clustern beim Übergang auf die technische Architektur aufgetrennt und entweder zu gemeinsamen Buffern übersetzt oder zu Clustern, die das Verhalten von Hardwarekomponenten (z.B. Bussen) nachbilden. Im ersten Fall gibt es keine Verhaltensveränderung, im zweiten kommt jedoch Verhalten hinzu, das die Art und Weise der Kommunikation zwischen zwei Komponenten (die sich in den jeweiligen Clustern befinden) beschreibt, d.h. es wird in die Verhaltensbeschreibung des gesamten Systems noch ein Cluster eingefügt, der nicht aus der logischen Architektur stammt, sondern sich aus Verteilung der Cluster und der Eigenschaften der zugrunde liegenden Hardware ergibt.

Cluster können in zwei Gruppen eingeteilt werden:

Applikationscluster. Applikationscluster sind Cluster, die ein Stück Funktionalität der Applikation erbringen und deren Inhalt hardwareunabhängig ist. Ihr Verhalten stammt aus der logischen Architektur und ist mit den Konzepten der logischen Architektur beschrieben. Um ausgeführt werden zu können, müssen die Applikationscluster auf die einzelnen Mikrocontroller integriert werden. Hierbei können sie prinzipiell auf jeden Mikrocontroller verteilt werden, sofern Speicher- und Echtzeitanforderungen erfüllt werden.

Hardwarecluster. Mit Hardwareclustern wird das Verhalten der nicht frei programmierbaren Anteile der Plattform (z.B. Sensoren oder Aktoren) angegeben.Das Spektrum der Verhaltensangaben kann dabei von der einfachen Identität bis hin zu komplexeren Verhaltensweisen, wie die Simulation von Hardwareausfällen, reichen. In welchem Abstraktionsgrad das Verhalten der Hardwarekomponenten angegeben ist, hängt vom gewünschten Genauigkeitsgrad ab, in dem die Aussagen über bestimmte Eigenschaften des Systems getroffen

werden sollen. Das Verhalten einer Hardwarekomponente kann aus der logischen Architektur abgeleitet werden, falls dort das gesamte System (d.h. Hardware und Software) und nicht lediglich der Softwareanteil modelliert wurde. Hardwarecluster dienen der Verhaltensbeschreibung einer Hardwarekomponente und können nicht frei verteilt werden.

Die Kommunikation zwischen Clustern wird asynchron über *Buffer* realisiert. Zu Beginn seiner Ausführung liest ein Cluster seine Eingaben von den entsprechenden Buffern und am Ende seiner Ausführung schreibt er seine Ergebnisse auf die entsprechenden Buffer.

5.4 Sichten

Die technische Architektur beschreibt abstrakt die Realisierung, welche aus Hardware und Software besteht. Dementsprechend gibt es auf dieser Ebene auch die *Hardwaresicht*, die die Hardwareanteile repräsentiert und die *Laufzeitsicht*, die das Systemverhalten repräsentiert. Die Einbettungsbeziehung, also welche Softwareteile zu welchen Hardwareteilen zugewiesen werden, werden in der *Deploymentsicht* beschrieben.

5.4.1 Hardwaresicht

In der Hardwaresicht wird die Topologie der Hardwareanteile der Plattform abstrakt beschrieben. Es werden alle realen Hardwarebauteile geeignet zu Hardwarekomponenten zusammengefasst und deren wichtigste Eigenschaften spezifiziert. Da in diesem Papier die Entwicklung von Software für eingebettete Systeme im Vordergrund steht, sind nur die Eigenschaften der Hardware von Interesse, die direkten Einfluss auf das Verhalten der Software haben. Solche Eigenschaften sind zum Beispiel bei Bussen die Übertragungszeiten, bei Sensoren die Aktualisierungsrate, bei Mikrocontrollern die Größe des zur Verfügung stehenden Speichers, usw.

Hardwaresicht des begleitenden Beispiels „ACC“ Für das begleitende Beispiel „ACC“ ist die Hardwaresicht in Abbildung 10 dargestellt. Der Hardwareanteil der Plattform kann in 13 Hardwarekomponenten aufgeteilt werden:

- eine ausführende Einheit (Mikrocontroller),
- ein Bus (CAN-Bus),
- eine kombinierte Ein-/Ausgabeeinheit: Panel zur Anzeige der Ergebnisse und zur Eingabe von Testdaten,
- acht Ausgabeeinheiten (8 LEDs zur Anzeige von Systemzuständen inkl. der 8 einfachen Verbindungen zum Mikrocontroller),
- zwei Eingabeeinheiten (2 Potentiometer zur Eingabe von Benutzereinstellungen, inkl. Verbindung zum Mikrocontroller)

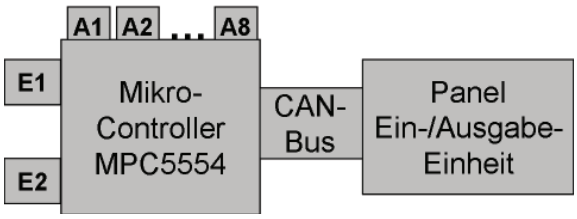

Abbildung 10: Hardwaresicht des begleitenden Beispiels „ACC“

5.4.2 Laufzeitsicht

In der Laufzeitsicht wird das gesamte Systemverhalten als Zusammenspiel von Hard- und Software in Form von *Clustern*, *Events*, *Timern* und *Buffern* abstrakt nachgebildet. Jedes Element der Laufzeitsicht ist dabei eindeutig einer Hardwarekomponente zugeordnet. Alle Laufzeitelemente einer Hardwarekomponente modellieren gemeinsam das Verhalten und den Zustand dieser Hardwarekomponente. Durch die Komposition der Hardwarekomponenten, so wie sie in der Hardwaresicht gegeben ist, und damit ihrer Laufzeitsichtelemente, können dann Aussagen über das gesamte System in Bezug auf Reaktionszeiten und mögliche Systemzustände gemacht werden.

Ein Cluster stellt für das Ausführungsmodell der Laufzeitsicht eine Ausführungseinheit dar. Für jeden Cluster wird ein Event (= Aktivierungsbedingung) definiert, bei dessen Auftreten der Cluster aktiviert und damit rechenbereit wird. Auf jeder Hardwarekomponente kann maximal ein Cluster zu einem Zeitpunkt ausgeführt werden, d.h. es können maximal so viele Cluster parallel ausgeführt werden, wie Hardwarekomponenten vorhanden sind. Es dürfen natürlich nur Cluster auf denjenigen Hardwarekomponenten ausgeführt werden, die diese auch enthalten. Falls mehrere Cluster einer Hardwarekomponente im rechenbereiten Zustand sind, entscheidet das Ausführungsmodell anhand von vorher vergebenen Prioritäten, welcher dieser Cluster ausgeführt wird. Ein Event kann sowohl auf Ereignisse, die in der Umwelt stattfinden, basieren, als auch auf Ereignisse, die innerhalb des Systems stattfinden (z.B. Aktualisierung von Buffern). Events dürfen Cluster von direkt verbundenen Hardwarekomponenten aktivieren. Dadurch können sich die Cluster von direkt verbundenen Hardwarekomponenten synchronisieren. In Echtzeitsystemen muss sehr oft der Ablauf von bestimmten Zeitspannen beachtet werden. Dies wird in der Laufzeitsicht mit dem Element *Timer* modelliert. Der Ablauf eines Timers kann auch ein Event auslösen und damit einen Cluster aktivieren. Mit Hilfe von Timern und Events können zeitgesteuerte Abläufe geeignet modelliert werden. Die Kommunikation zwischen Clustern wird asynchron über Buffer realisiert.

Alle Elemente der Laufzeitsicht besitzen drei Gruppen von Eigenschaften, die ihr Laufzeitverhalten bestimmen, nämlich hardwareunabhängige, hardwareabhängige und kontextabhängige Eigenschaften. Hier werden beispielhaft diese Eigenschaften für das Element Cluster erläutert:

Hardwareunabhängige Eigenschaften: In dieser Gruppe sind alle Eigenschaften zusammengefasst, die für den Cluster immer gelten, egal auf welcher Hardware sie laufen. Hierzu gehören, neben der Funktionalität auch die benötigten Daten und Anforderungen an die Plattform bzw. vorgegebene Constraints, die die Plattform betreffen. Diese Eigenschaf-

ten stehen alle im Moment der Clusterbildung fest und können aus den oberen beiden Ebenen bzw. aus den Anforderungen selbst abgeleitet werden.

Hardwareabhängige Eigenschaften: Hier finden sich alle Eigenschaften, die von der Wahl der tatsächlichen Hardware (z.B. Mikrocontrollertyp, auf dem der Cluster laufen soll) abhängen. Hier sind Eigenschaften wie die WCET (worst case execution time) - ohne Berücksichtigung von möglichen Unterbrechungen - , Speicherbedarf oder ähnliches zu nennen. Diese Eigenschaften stehen erst fest, wenn die zugrunde liegende Hardware ausgewählt worden ist.

Kontextabhängige Eigenschaften: Unter kontextabhängigen Eigenschaften werden alle Eigenschaften verstanden, die sich ergeben, sobald die Umgebung des Clusters bekannt ist. Darunter fällt die Auswirkung der „Vorbelastung“ des Mikrocontrollers durch vorhandene Software genauso wie die Auswirkung der Eigenschaften der an diesen Mikrocontroller angeschlossenen Hardwarekomponenten auf den Cluster. Hauptsächlich spiegeln sich diese Eigenschaften in den zu erwartenden maximalen Verzögerungszeiten bei der Ausführung der Cluster wieder. Diese kann durch unterschiedliche Priorisierung der Cluster beeinflusst werden.

Ausführungsmodell Wie oben schon erwähnt ist zu jedem Zeitpunkt maximal ein Cluster pro Hardwarekomponente aktiv. Falls mehrere Cluster einer Hardwarekomponente rechenbereit sind, wird die Auswahl des nächsten zu rechnenden Clusters nach einem prioritätsbasierten Schedulingverfahren durchgeführt. Es wird immer der rechenbereite Cluster gerechnet, der die höchste Priorität hat. Ein Cluster ist rechenbereit, wenn er durch einen Event aktiviert wurde bzw. durch einen höher priorisierten Cluster unterbrochen wurde. Dies kann sowohl den Schedulingalgorithmus, der durch ein Betriebssystem durchgeführt wird, als auch z.B. so etwas wie die durch Hardware durchgeführte Busarbitrierung bei einem ereignisgesteuerten Bus nachbilden. Das Ausführungsmodell kann in der Theorie zu jedem Zeitpunkt einen Event feststellen, daraufhin den Schedulingalgorithmus aufrufen und prüfen, ob ein anderer Cluster ausgeführt werden soll, d.h. die Auswertung, ob ein Event vorliegt, geschieht parallel und zunächst unabhängig von der Ausführung der Cluster. Ebenso zählen auch die Timer parallel und unabhängig von der Ausführung weiter. Die Eventauswertung und die Cluster haben jederzeit Zugriff auf die Timer.

Dieses Ausführungsmodell erlaubt zusammen mit den entsprechenden Eigenschaften der Elemente der Laufzeitsicht Aussagen über Reaktionszeiten und mögliche Systemzustände. Für eine Simulation muss das Ausführungsmodell bei der Eventauswertung eingeschränkt werden, d.h. diese kann nicht mehr jederzeit stattfinden und es müssen daher geeignete Zeitpunkte für die Eventauswertung gefunden werden (z.B. nach der Beendigung eines Clusters oder feste Zeittakte). Mit dieser Einschränkung sind zwar nicht mehr alle Systemzustände abbildbar, aber das System wird simulierbar.

Statische Analyse In der Laufzeitsicht kann ein Abhängigkeitsgraph extrahiert werden, der für ein bestimmten Eingangsevent alle von diesem Event betroffenen Elemente der Laufzeitsicht beinhaltet. Mit Hilfe dieses Abhängigkeitsgraphen und der Eigenschaften der betroffenen

Elemente können prinzipiell alle möglichen Systemabläufe und damit auch nun die minimalen und maximalen Reaktionszeiten bestimmt werden.

Dynamische Analyse Wie oben schon erwähnt ist das Ausführungsmodell auf der Ebene der technischen Architektur auch für die Simulation des Verhaltens verwendbar. Es müssen lediglich die Zeitpunkte der Eventauswertung angegeben werden.

Laufzeitsicht des begleitenden Beispiels In Abbildung 11 ist ein kleiner Ausschnitt der Laufzeitsicht gezeigt. Abgebildet ist hier ein Teil der Laufzeitsicht des Mikrocontrollers und zwar derjenige, der die direkte Umgebung der Applikation betrifft. Die Applikation findet sich hier komplett in einem Cluster (Cluster „ACC“ in der Abbildung), d.h. die Applikation ist beim Übergang auf die Ebene der technischen Architektur nicht aufgetrennt worden.[7]

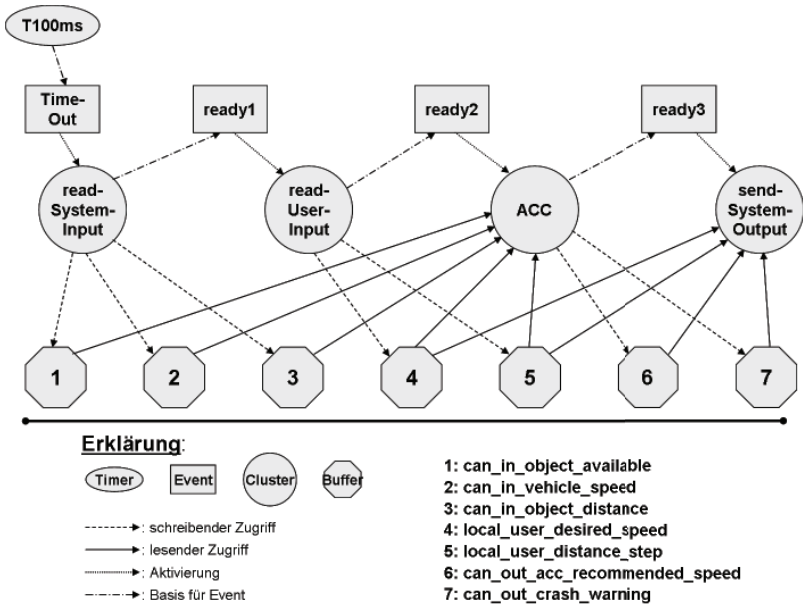

Abbildung 11: Ausschnitt aus Verhaltensicht des begleitenden Beispiels „ACC“

5.4.3 Deploymentsicht

Die Deploymentsicht beinhaltet die Zuordnung der Cluster, Events, Timer und Buffer zu den Hardwarekomponenten.

[7]Eine Aufteilung des Modells wird in AutoFocus2 noch nicht unterstützt

5.5 Gegenstand aktueller Forschung und offene Fragen

- Weitgehende Automatisierung des Übergangs von der Applikation zur technischen Architektur (Partitionierung der logischen Architektur, Generieren von Schedules, Integration auf Plattform, ...).
- Abstrahierung von bestimmten Hardwareeigenschaften: Welche Hardwareeigenschaften sind notwendig und wie stark darf man hier abstrahieren?
- Generierung von gültigen Schedules auf Basis der Echtzeitanforderungen?

6 Zusammenfassung und Ausblick

In diesem Papier wurde ein umfassendes Architekturmodell für die Modellierung eingebetteter Softwaresysteme im Automobilbereich eingeführt und anhand einer typischen Automobilfunktion (ACC: Adaptive Cruise Control) beispielhaft veranschaulicht. Das vorgestellte Gesamtarchitekturmodell gliedert sich in drei aufeinander aufbauende Abstraktionsebenen: die Nutzungsebene, die logische Architektur sowie die technische Architektur. Die *Nutzungsebene* liefert eine Black-Box-Beschreibung des Systemverhaltens, wie es vom Benutzer wahrgenommen wird. Schwerpunkt der *logischen Architektur* ist die Aufteilung des Systems in logische, kommunizierende Einheiten, deren Gesamtverhalten das in der Nutzungssicht festgelegte Verhalten realisiert. In der *technischen Architektur* wird die Hardwareunabhängigkeit aufgegeben und das gesamte System wird als System beschrieben, dessen gesamtes Verhalten sowohl von Hardware als auch von Software erbracht wird. Die technische Ebene ist die konkreteste Ebene unseres Architekturmodells und Ausgangsbasis für die Implementierung. Neben einer kurzen Motivation der jeweiligen Ebene wurden für jede Ebene die Hauptmodellierungselemente und Sichten auf das jeweilige Ebenenmodell vorgestellt.

Potential des Ansatzes Das vorgestellte Architekturmodell stellt eine geeignete Grundlage für einen systematischen Softwareentwicklungsprozess dar. Durch die Gliederung der Systembeschreibung in verschiedene Abstraktionsebenen wird eine systematische Vorgehensweise schon modellseitig unterstützt. Die beschriebenen Konzepte bieten ein hohes praktisches Umsetzungspotential: Ausgehend von der vorliegenden Beschreibung kann ein Metamodell und darauf aufbauend eine Werkzeugunterstützung erarbeitet werden. Durch die formale Fundierung der Modelle der einzelnen Ebenen ergeben sich weitreichende Vorteile: Konsistenzchecks und Verifikation innerhalb der einzelnen Modelle, Verifikation der Ebenenübergänge sowie weitgehende Automatisierungsmöglichkeiten.

Gegenstand aktueller Forschung und offene Fragen Die in diesem Papier vorgestellten grundlegende Ideen eines Gesamtarchitekturmodells werden derzeit in verschiedenen Projekten am Lehrstuhl Software & Systems Engineering der Technischen Universität München weiter ausgearbeitet. Bei der Beschreibung der einzelnen Ebenen wurden bereits ebenenspezifische offene Forschungsfragen identifiziert. Ebenenübergreifende offene Fragestellungen sind:

- Ausarbeitung der Übergänge zwischen den Ebenen und Erfassung des Automatisierungspotentials der jeweiligen Ebenenübergänge;
- Methodische und prozesstechnische Aspekte;
- Requirements Engineering: Klassifizierung der Anforderungen entsprechend der Abstraktionsebenen;
- Test und Verifikation entlang der Abstraktionsebenen;
- Unterstützung der Erfassung und Verwaltung der Modellelemente;
- Umsetzung in ein Metamodell.

Literatur

[Aut04] AutoRAID. AutoRAID Website. `http://www4.in.tum.de/~autoraid/`, 2004.

[Aut07a] AutoFOCUS2. AutoFOCUS2 Website. `http://www4.in.tum.de/~af2/`, 2007.

[AUT07b] AUTOSAR GbR. AUTOSAR Website. `http://www.autosar.org`, Juli 2007.

[BBC+00] Felix Bachmann, Len Bass, Gary Chastek, Patrick Donohoe, and Fabio Peruzzi. The architecture based design method. Technical Report CMU/SEI-2000-TR-001, CMU SEI Pittsburgh, 2000.

[BCK98] Len Bass, Paul Clements, and Rick Kazman. *Software architecture in practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[BGH+06] Jewgenij Botaschanjan, Alexander Gruler, Alexander Harhurin, Leonid Kof, Maria Spichkova, and David Trachtenherz. Towards modularized verification of distributed time-triggered systems. In *FM 2006: Formal Methods*, pages 163–178. Springer Verlag, 2006. available at `http://www.springerlink.com/content/dx40967755h318g6/`.

[BHS99] Manfred Broy, Franz Huber, and Bernhard Schätz. AutoFOCUS - Ein Werkzeugprototyp zur Entwicklung eingebetteter Systeme. *Informatik - Forschung und Entwicklung*, 14:121–134, 1999.

[BO92] Don Batory and Sean O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Trans. Softw. Eng. Methodol.*, 1(4):355–398, 1992.

[Bro05] Manfred Broy. Service-oriented Systems Engineering: Specification and Design of Services and Layered Architectures – The Janus-Approach. In Manfred Broy, Johannes Grünbauer, David Harel, and Tony Hoare, editors, *Engineering Theories of Software Intensive Systems*, number 195 in Proceedings of the NATO Advanced Study Institute on Engineering Theories of Software Intensive Systems, Marktoberdorf, Germany, July 2005.

[Bro06] Manfred Broy. Challenges in automotive software engineering. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 33–42, New York, NY, USA, 2006. ACM.

[BS01] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems:* FOCUS *on Streams, Interfaces and Refinement*. 2001.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[CHE05] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.

[Dam05] Werner Damm. Controlling speculative design processes using rich component models. In *ACSD '05: Proceedings of the Fifth International Conference on Application of Concurrency to System Design*, pages 118–119, Washington, DC, USA, 2005. IEEE Computer Society.

[DGP+04] Martin Deubler, Johannes Grünbauer, Gerhard Popp, Guido Wimmel, and Christian Salzmann. Tool Supported Development of Service Based Systems. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC) 2004, Busan (Korea), November 30 – December 3*, pages 99–108, 2004.

[DVM+05] W. Damm, A. Votintseva, A. Metzner, B. Josko, T. Peikenkamp, and E. Böde. Boosting re-use of embedded automotive applications through rich components. `http://www.tcs-trddc.com/tecs/2006-reading/Boosting%20Re-use%20of%20Embedded%20Automotive%20Applications%20Through%20Rich%20Components.pdf`, 2005.

[ETA08a] ETAS GmbH. ASCET. `http://www.etas.com`, January 2008.

[ETA08b] ETAS Group. ETAS Website. `http://www.etas.com/de/products/ascet_se_software_engineering.php`, Februar 2008.

[ETA08c] ETAS Group. ETAS Website. `http://www.etas.com/de/products/ascet_md_modeling_design.php`, Februar 2008.

[GHH07] Alexander Gruler, Alexander Harhurin, and Judith Hartmann. Modeling the functionality of multi-functional software systems. In *Proceedings of 14th Annual IEEE International Conference on the Engineering of Computer Based Systems (ECBS)*, March 2007.

[GLS08] Alexander Gruler, Martin Leucker, and Kathrin Scheidemann. Modelling and Model Checking Software Product Lines. In *Proceedings of the 10th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS08)*, LNCS, 2008. To appear.

[Gri05] Klaus Grimm. Software-Technologie im Automobil. In Peter Liggesmeyer and Dieter Rombach, editors, *Software-Engineering eingebetteter Systeme: Grundlagen – Methodik – Anwendungen*, chapter 16, pages 407–430. Spektrum, Heidelberg, 2005.

[GS07] Eva Geisberger and Bernhard Schätz. Modellbasierte anforderungsanalyse mit autoraid. *Informatik - Forschung und Entwicklung*, 21(3-4):231–242, June 2007.

[HH08] Alexander Harhurin and Judith Hartmann. Towards consistent specifications of product families. In *FM'08: 15th International Symposium on Formal Methods*, volume 5014 of *LNCS*. Springer Verlag, 2008.

[HHK01] Thomas A. Henzinger, Benjamin Horowitz, and Christoph Kirsch. Giotto: A time-triggered language for embedded programming. In *EMSOFT 2001*. Springer-Verlag, 2001.

[HSSS96] Franz Huber, Bernhard Schätz, Alexander Schmidt, and Katharina Spies. AUTOFOCUS – A Tool for Distributed Systems Specification. In *Proceedings FTRTFT'96 - Formal Techniques in Real-Time and Fault-Tolerant Systems, P. 467–470*. Springer Verlag, LNCS 1135, 1996.

[ITE08] ITEA. EAST-EEA Website. http://www.east-eea.net, January 2008.

[KAJW95] Sanjaya Kumar, James H. Aylor, Barry W. Johnson, and Wm A. Wulf. *The Codesign of Embedded Systems: A Unified Hardware/Software Representation.* Kluwer Academic Publishers, Norwell, MA, USA, 1995.

[KCH+90] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report, SEI, CMU, Pittsburgh, PA, 1990.

[KK99] R. Kazman and M. Klein. Attribute-based architectural styles. Technical Report 22, CMU SEI, 1999.

[KKL+98] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. Form: A feature-oriented reuse method with domain-specific reference architectures. *Ann. Softw. Eng.*, 5:143–168, 1998.

[Kru95] Philippe Kruchten. Architectural blueprints—The "4+1" view model of software architecture. *IEEE Software*, 12(6):42–50, November 1995.

[MGEB03] Nenad Medvidovic, Paul Gruenbacher, Alexander Egyed, and Barry W. Boehm. Bridging models across the software lifecycle. *J. Syst. Softw.*, 68(3):199–215, 2003.

[Mob07] MobilSoft. mobilSoft Website. http://www.itm.tum.de/mobilsoft/, August 2007.

[OMG07] OMG. UML Website. http://www.uml.org, Juli 2007.

[OSC08a] OSCI. IEEE 1666 Standard System C Language Reference Manual. http://standards.ieee.org/getieee/1666/index.html, February 2008.

[OSC08b] OSCI. SystemC Website. http://www.systemc.org, February 2008.

[Sch05] Bernhard Schätz. Building components from functions. In *Proceedings of FACS 2005*, volume 160 of *ENTCS*, 2005.

[Sch06] Bernhard Schätz. Combining product lines and model-based development. In *Proceedings of Formal Aspects of Component Systems (FACS 2006)*, 2006.

[Sch07] Oliver Scheikl. Modellbasierte Entwicklung eines ACC-Systems. Interdisziplinäres Projekt, TU-München, August 2007.

[SZW05] Jing Sun, Hongyu Zhang, and Hai Wang. Formal semantics and verification for feature modeling. In *ICECCS '05: Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)*, pages 303–312, Washington, DC, USA, 2005. IEEE Computer Society.

[Tem04] Josef Templ. TDL Specification and Report. Technical report, Department of Computer Science, University of Salzburg, Austria., http://www.SoftwareResearch.net/site/publications/C059.pdf, March 2004.

[The07] The Mathworks, Inc. Matlab/Simulink. http://www.mathworks.com, Juli 2007.

[The08a] The MathWorks, Inc. The MathWorks Website. http://www.mathworks.com/products/simulink/, Februar 2008.

[The08b] The MathWorks, Inc. The MathWorks Website. http://www.mathworks.com/products/stateflow/, Februar 2008.

[The08c] The MathWorks, Inc. The MathWorks Website. http://www.mathworks.com/products/simcontrol/, Februar 2008.

[Ver07] Verisoft. Verisoft Website. http://www.verisoft.de, december 2007.

[vL03] Axel van Lamsweerde. From system goals to software architecture. In M. Bernardo and P. Inverardi, editors, *School on Formal Methods*, volume LNCS 2804, pages 25–43, Bertinoro, Italy, 2003. Springer-Verlag.

[WBB+06] Rob Wojcik, Felix Bachmann, Len Bass, Paul Clements, Paulo Merson, Robert Nord, and Bill Wood. Attribute-driven design (add). Technical Report CMU/SEI-2006-TR-023, CMU SEI Pittsburgh, 2006.

[WFH+06] Doris Wild, Andreas Fleischmann, Judith Hartmann, Christian Pfaller, Martin Rappl, and Sabine Rittmann. An Architecture-Centric Approach towards the Construction of Dependable Automotive Software. In *Proceedings of the SAE 2006 World Congress*, 2006.