Qualitätssteigerung der Automotive-Software durch formale Spezifikation funktionaler Eigenschaften auf der Abstraktionsebene des Modellentwurfs

Richard Bogenberger
BMW Group Forschung und Technik
Richard.Bogenberger@bmw.de

David Trachtenherz
BMW Group Forschung und Technik
David.Trachtenherz@bmw.de

Abstract: Softwarebasierte Funktionalitäten gewinnen zunehmend an Bedeutung für moderne Automobile – ein Großteil der Innovationen wird von Elektronik und Software getrieben. Zahlreiche Sicherheits- und Komfortfunktionen werden durch Software ermöglicht. Automobile eingebettete Systeme bilden ein hochkomplexes heterogenes Netzwerk, bestehend aus bis zu 70 Steuergeräten. Die Beherrschung der Komplexität dieser Netzwerke stellt zurzeit eine der größten Herausforderungen für Softwareingenieure im Automobilbereich dar. Während testbasierte Methoden zur Qualitätssicherung für solche vernetzten Systeme an ihre Grenzen stoßen, eröffnen formale Methoden durch Behandlung von Systemspezifikationen auf höherer Abstraktionsebene eine Perspektive der Qualitätssicherung in frühen Entwicklungsphasen, in denen die Netzwerkarchitektur entworfen wird.

1 Einführung: Problemstellung und Motivation

Wenn von komplexen Softwaresystemen die Rede ist, denkt man zunächst an Datenbanken, Computer-Betriebssysteme, Telekommunikationsnetze und wohl auch an das LKW-Mauterfassungssystem Toll Collect. In letzter Zeit stellt allerdings die Beherrschung komplexer eingebetteter Softwaresysteme auch die Automobilindustrie vor neue Herausforderungen. Der deutsche ADAC und der österreichische ARBÖ kommen in ihren aktuellen Pannenstatistiken übereinstimmend zu dem Ergebnis, dass ca. 11% aller Pannen durch Elektronik verursacht werden. Würden nur neuere Fahrzeuge berücksichtigt, fiele dieser Anteil deutlich höher aus.

Vor fünfzehn Jahren spielte die Software-Entwicklung für die Automobilindustrie keine besondere Rolle – im Jahr 1990 belegte die Software in einem 7er BMW weniger als 100 KB. Das von Intel-Mitbegründer Gordon Moore aufgestellte Gesetz, nach dem sich die Transistoranzahl in Halbleiter-Chips alle 18-24 Monate verdoppelt, scheint jedoch auch auf die eingebettete Software in Automobil-Steuergeräten übertragbar zu sein – ihr Umfang in einem aktuellen 7er BMW beträgt mehr als das Hundertfache gegenüber dem 7er im Jahr 1990. Allein die sicherheitskritische Software belegt über 15 MB. Nehmen wir die Infotainment-Software hinzu, so erhalten wir über 100 MB Software pro Fahrzeug [Fri04].

Die Funktionalität wichtiger Sicherheitssysteme, wie Airbags, ABS (Antiblockiersystem) oder DSC (Dynamische Stabilitätskontrolle), die aus einem modernen Automobil nicht

mehr wegzudenken sind, wird durch Software ermöglicht. Gleiches gilt für zahlreiche weitere Komfort-Funktionen, von der mittlerweile fast selbstverständlichen GPS-Navigation bis hin zu aktuellen Innovationen wie die ”Aktivlenkung”. Ein Verzicht auf die Elektronik/Software scheidet damit als Lösung für aktuelle Elektronikprobleme aus. Im Gegenteil: die gleich lautende Einschätzung mehrerer namhafter Automobilhersteller und des VDA besagt, dass über 90% aller Innovationen im Automobil von Elektronik und Software getrieben werden. Sie machen einen beträchtlichen Anteil der Herstellungskosten aus. Betrachten wir den Entwicklungsaufwand, so entfällt ungefähr die Hälfte der Entwicklungskosten eines elektronischen Steuergeräts (ECU) auf die Software [Fri04], womit sich der Bogen von der Fahrzeug-Entwicklung zum Software-Engineering schließt.

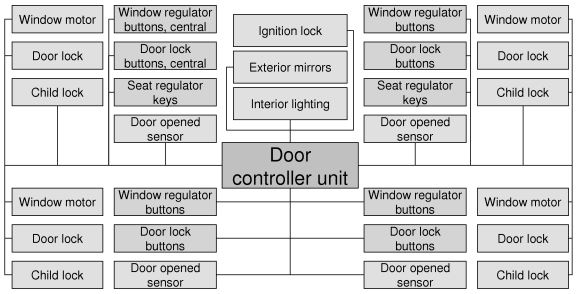

Abbildung 1: Türsteuergeräte-Vernetzung – vereinfachte Sicht

Die Komplexität der Steuergerätesoftware ist nicht ausschließlich durch den Funktionsumfang bedingt, sondern auch durch den hohen Grad der Vernetzung und die wechselseitigen Abhängigkeiten zwischen Funktionalitäten im Steuergeräte-Netzwerk. Ein modernes Premium-Fahrzeug verfügt über etwa 70 verschiedene Steuergeräte – zur Veranschaulichung dieser Komplexität wird auf Abb. 1 ein stark vereinfachtes Netzwerk rund um ein Türsteuergerät gezeigt. Steuergeräte kommunizieren über bis zu fünf verschiedene Busse, die sich in Eigenschaften wie Bandbreite, Protokolle, Echtzeitverhalten u.a. unterscheiden. Um eine Vernetzung im Fahrzeug zu ermöglichen, werden mehrere Gateways in das Bordnetz integriert. Damit kann jede einzelne Funktion eines Steuergeräts mit jedem anderen Steuergerät kommunizieren und es beeinflussen. Die Beherrschung dieser komplexen Netzwerke aus verschiedenen, häufig bereits für sich komplexen Komponenten ist heute eine der größten Herausforderungen an die Software-Entwicklung im Automobilbereich.

2 Methoden und Techniken

2.1 Abstraktion und Qualitätssicherung

Ein oft beschrittener Weg zur Beherrschung der Komplexität softwaretechnischer Systeme ist die Erhöhung des Abstraktionsgrades – der Schritt von Assembler zu Fortran und C, sowie später von prozeduralen zu objekt-orientierten Sprachen sind nur ein Beispiel dafür.

Jede höhere Abstraktionsebene ermöglichte die Beherrschung größerer und komplexerer Systeme. Während die Entwicklung und Instandhaltung eines 100 KB großen Programms in Assembler eine Herausforderung darstellt, ist dies mit den Mitteln objektorientierter Programmiersprachen Bestandteil vieler Aufgaben in technischen Studiengängen.

Erste Schritte, um der Komplexität von Softwaresystemen in der Automobilindustrie zu begegnen, wurden mit Werkzeugen zur modellbasierten Entwicklung ASCED-SD, MATLAB/Simulink u.a. vollzogen. Ein weiterer Schritt zur Erhöhung der Softwarequalität wäre neben der Modellierung von Software deren Verifikation. Hierzu ist der Abstraktionsgrad konventioneller Qualitätssicherungstechniken nicht ausreichend. Gegenwärtig stellt Testen der ausführbaren Spezifikation die wichtigste Technik zur Qualitätssicherung dar – Tests werden entweder direkt auf dem Programmcode oder als Simulationsläufe auf dem Modell durchgeführt. Das inhärente Problem des Testens ist aber, dass ein vollständiger Test aufgrund der Anzahl möglicher Eingabekombinationen im Allgemeinen nicht möglich ist – das gilt sowohl für Tests auf Programmcode-Ebene als auch für Modellsimulationen. Wenn wir das Wort *Verifikation* genau nehmen, handelt es sich beim Testen um kein Verifikationsverfahren – das Ergebnis eines Tests ist entweder *Fehler gefunden* oder *keine Fehler gefunden*, aber leider niemals *keine Fehler vorhanden*. Testen kann somit keinen Korrektheitsnachweis liefern. Hier liegt das Potential des Entwurfs auf höherer Abstraktionsebene – der Abstraktionsgrad ist ausreichend, um die Verifikation bestimmter Eigenschaften mit vertretbarem Aufwand durch einen mathematischen Beweis zu ermöglichen (wobei hier das Risiko ungeeigneter Formalisierung informaler Anforderungen besteht).

Eine formale Verifikationstechnik, die in der Hardwareentwicklung breite Verwendung findet und in der Softwareentwicklung zunehmende Bedeutung erlangt, ist Model-Checking [McM93, CGP99]. Wenn eine Systemspezifikation als maschinenorientiertes Systemmodell (beispielsweise als E/A-Automat) gegeben ist, und temporallogische Anforderungen (typischerweise in CTL oder LTL) formuliert sind, versucht ein Model-Checker die Anforderungen verifizieren, indem für alle erreichbaren Systemzustände überprüft wird, ob die Anforderungen in ihnen gelten. In letzter Zeit werden Plug-Ins für verschiedene Model-Checking-Werkzeuge sukzessive in industrielle Software-Modellierungswerkzeuge wie MATLAB/Simulink oder ASCET-SD integriert [BG00, BBB$^+$04]. Dabei werden zu verifizierenden Anforderungen entweder auf der Ebene des Systemmodells oder auf der Ebene des generierten Programmcodes überprüft.

Ein wesentlicher Vorteil von Model-Checking ist, dass es sich hierbei um eine automatische Technik handelt. Das Hauptproblem ist die Zustandsraumexplosion – die Größe des Zustandsraums eines System kann mit der Anzahl von Komponenten und Variablen exponentiell wachsen. Daher sind größere Systeme, die aus mehreren Komponenten bestehen, oft schwer oder letztendlich nicht model-checkbar.

Model-Checking kann sowohl auf der Abstraktionsebene des Modellentwurfs, als auch auf der Ebene des Programmcodes operieren. Tests werden vorwiegend auf Ebene des Programmcodes durchgeführt, einschließlich des aus Modellen generierten Codes. Sowohl Verifikation durch Model-Checking, als auch testbasierte Qualitätssicherung teilen einen entscheidenden Aspekt – eine maschinenorientierte Spezifikation der Systeme wird benötigt, denn ohne ausführbaren Code bzw. E/A-Automaten, die das Verhalten des Systems definieren, ist Testen oder Model-Checking nicht möglich.

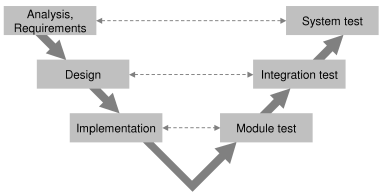

Abbildung 2: Vereinfachtes V-Modell

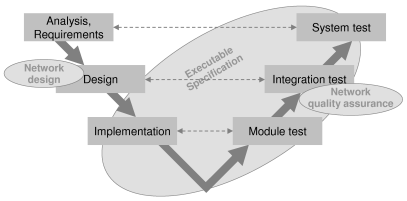

Abbildung 3: Vernetzte Funktionalitäten: Entwurf und Qualitätssicherung im V-Modell

Wir wollen die Auswirkungen dieses Aspekts im Kontext industrieller Softwareentwicklung betrachten. Vorgehensmodelle der Softwareentwicklung für große industrielle Systeme beschreiben eine Top-Down-Vorgehensweise in frühen Phasen und eher eine Bottom-Up-Vorgehensweise für Qualitätssicherung. Die Abb. 2 stellt das V-Modell [BD95] vereinfacht dar, das ein ISO-Standard ist und als Basis für das Software-Lebenszyklusmodell bei BMW [Sch02] sowie weiteren Automobilherstellern dient. Andere verbreitete Vorgehensmodelle wie Rational Unified Process (RUP) [Kru00] weisen eine ähnliche Aufwandsverteilung zwischen Qualitätssicherung und Entwicklung auf – die Qualitätssicherung findet größtenteils während und nach der Implementierungsphase statt, während der Softwareentwurf in frühen Entwicklungsphasen durchgeführt wird.

Die Bottom-Up-Vorgehensweise bei der Qualitätssicherung ist eine logische Folge davon, dass Tests nur auf ausführbare Module anwendbar sind – wir können aus diesem Grund keine testbasierte Qualitätssicherung vor der Implementierungsphase durchführen. Iterationen während des Softwareentwicklungsprozesses wie in Boehms Spiralmodell [Boe88] mildern die Auswirkungen dieser Einschränkung in gewissem Maße ab, können dennoch das inhärente Problem testbasierter Qualitätssicherung nicht beseitigen – bevor Programmcode vorliegt, ist keine Qualitätssicherung für funktionale Eigenschaften möglich außer informalen Mitteln wie Spezifikationsreviews. Formale Techniken wie Model-Checking sowie Verifikationsverfahren, die für andere maschinenorientierte Formalismen (z.B. die B-Methode [Abr96]) angewendet werden können, sind ebenfalls erst einsetzbar, wenn der Systementwurf eine maschinenorientierte Verhaltensspezifikation enthält.

Wie im Abschnitt 1 beschrieben, stellt die Beherrschung vernetzter Steuergerätesysteme eine besondere Herausforderung für Softwareentwicklung im Automobilbereich dar. Die Architektur und Funktionalität dieser Netzwerke wird spätestens in der Entwurfsphase definiert. Die korrespondierende Qualitätssicherung wird mithilfe von Integrationstests durchgeführt (Abb. 3). Damit gibt es eine beachtliche Lücke zwischen der Spezifikation und der Qualitätssicherung für vernetzte Funktionalitäten. Während des Netzwerkentwurfs entstandene Fehler können deshalb die Entwicklung bis zur Systemintegration überdauern und erst bei Integrationstests entdeckt werden. Die Korrektur dieser Fehler kann eine Überarbeitung betroffener Systemteile beginnend mit der Entwurfsphase erfordern – verbunden mit dem Risiko, neue Fehler einzubauen, oder von überarbeiteten Komponenten erbrachte Funktionalitäten zu beeinträchtigen. Es überrascht nicht, dass fast 50% schwerwiegender Fehler ihren Ursprung in der Entwurfsphase haben [Jon91]. Abhilfe können formale Spezifikations- und Verifikationstechniken bringen, die präzise Spezifikationen und

mathematische Beweise für aus mehreren vernetzten Komponenten bestehende Systeme ermöglichen, ohne dass eine vollständige Verhaltensspezifikation der einzelnen Komponenten vorhanden sein muss. Damit würde letztendlich eine Top-Down-Verifikation vernetzter Funktionalitäten möglich, die nebenläufig zum Systementwurf verläuft. Diese Art von Verifikation ist nicht machbar mit Qualitätssicherungsmethoden, die eine ausführbare Spezifikation benötigen, wie es für testbasierte Qualitätssicherung der Fall ist.

2.2 Nachrichtenstromorientierte Spezifikation und Verifikation

Formale Spezifikation und Beweise für vernetzte Systeme vor der ausführbaren Festlegung des Verhaltens erfordern einen Formalismus, der auf die Beschreibung der Kommunikation zwischen einzelnen Komponenten gerichtet ist und damit eine Beobachtung der Systemausführungen mit Hinblick auf Nachrichtenströme erlaubt (in gewisser Analogie zur Übertragung von Nachrichten zwischen Steuergeräten, wie sie beispielsweise über den CAN-Bus und andere Kommunikationsbusse abläuft).

2.2.1 Grundlegende Begriffe

Als semantische Basis für unsere Arbeit wollen wir die Kommunikationssemantik des Modellentwurfswerkzeugs AUTOFOCUS nutzen [SH99], das eine zeit-synchrone Teilmenge des FOCUS-Formalismus implementiert [BS01]. Komponenten kommunizieren mithilfe von Nachrichten, die über gerichtete Kanäle versendet werden.Eine Komponente darf beliebig viele Eingabe- und Ausgabeports besitzen. Ein Kanal kann einen Ausgabeport mit einem Eingabeport verbinden, wobei jeder Eingabeport mit höchstens einem Kanal verbunden sein darf.

Der Kompositionsbegriff ist intuitiv. Mehrere Komponenten können zu einer (Ober-)Komponente zusammengesetzt werden, indem einige ihrer Kommunikationsports durch Kanäle verbunden werden (Abb. 4). Die Schnittstelle der neuen Komponente besteht aus Eingabe- und Ausgabeports der gekapselten komponenten, die nicht durch Kanäle gebunden wurden (Abb. 5). Das verwendete Komponentenmodell kann auch als eingeschränkte Version des in [BBR$^+$00] beschriebenen Komponentenmodells verstanden werden – eine wesentliche Einschränkung ist dabei, dass syntaktische Schnittstellen der Komponenten nicht dynamisch rekonfigurierbar sind.

Die Kommunikation geschieht durch Nachrichten, die zwischen Komponenten über Kanäle versendet werden. Das System besitzt einen globalen Takt. Jeder Taktschritt t kann in folgende Teilschritte unterteilt werden:

- **Nachrichtenübertragung**: Im vorhergehenden Schritt $t-1$ produzierte Nachrichten liegen an Ausgabeports der Komponenten an und werden über Kanäle an verbundene Komponenteneingabeports versendet. Falls an einem Ausgabeport keine Nachricht ausgegeben wurde, wird eine ausgezeichnete leere Nachricht *NoMsg* übertragen.
- **Berechnung**: Alle Komponenten im System führen Berechnungen durch. Dabei werden die Werte an den Eingabeports und eventuell der interne Komponentenzustand zur

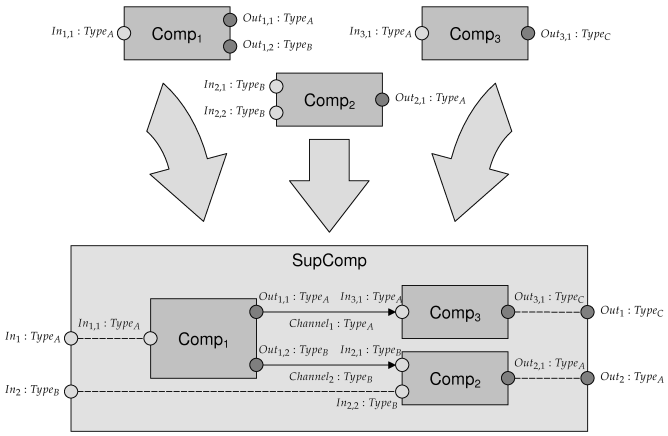

Abbildung 4: Komposition von Komponenten

Berechnung der Ausgabewerte und des neuen Komponentenzustands verwendet.

- **Ausgabe**: Die berechneten Ausgabewerte werden an die Ausgabeports der Komponenten übergeben. Der Schritt t ist damit abgeschlossen.

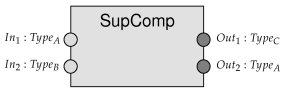

Abbildung 5: Blackbox-Sicht auf zusammengesetzte Komponente

Um die Abläufe eines Systems zu erfassen, für das keine ausführbare Spezifikation vorliegt, müssen wir die Kommunikationsgeschichte des Systems, d.h., die Nachrichtenströme zwischen Komponenten beobachten. Eine intuitive Herangehensweise für Berechnungen und Beweise über solche Systeme sind daher Prädikate auf Nachrichtenströmen. Um die Darstellung und das Verständnis zu vereinfachen, können zusätzlich lokale Zustände und Variablen in Komponenten verwendet werden – damit können Änderungen der Nachrichtenverarbeitung durch die Komponente angezeigt werden, die durch bestimmte Zusammenhänge in der Kommunikationsgeschichte herbeigeführt werden.

Vor der Betrachtung eines kleineren Systembeispiels wollen wir einige mathematische Definitionen für die Arbeit mit Nachrichtenströmen machen. Ein gezeiteter Strom s von Nachrichten des Typs M ist eine Abbildung $s : \mathbb{N} \to M$. Mit $s.t$ bezeichnen wir das t-te Element des Stroms s. Dieser ist entweder eine zum Zeitpunkt $t \in \mathbb{N}$ übertragene Nachricht $m \in M$, oder der ausgezeichnete Wert $NoMsg$, falls keine Nachricht zu übertragen war.

Ist ein Eingabeport In mit einem Ausgabeport Out durch einen Kanal verbunden, so gilt:

$$In.0 = NoMsg \ \wedge \ \forall t \in \mathbb{N}_+ .\ In.t = Out.(t-1) \tag{1}$$

Wenn ein Komponentenport $CompPort$ im Interface einer Oberkomponente als $SupCompPort$ sichtbar ist (z.B. $In_{1,1}$ und In_1 auf Abb. 4), dann gilt:

$$\forall t \in \mathbb{N}.\ CompPort.t = SupCompPort.t \tag{2}$$

2.2.2 Formale nachrichtenstromorientierte Spezifikation und Verifikation an einem Beispielsystem

Wir möchten unsere Betrachtungsweise des Systemverhaltens über Kommunikationsgeschichten anhand eines Beispiels erläutern. Wir spezifizieren formal einen Teilaspekt der Systemfunktionalität und verfeinern die Spezifikation, um Anforderungen an Teilkomponenten des Systems zu erhalten. Die Anforderungen an Teilkomponenten werden dabei so formuliert, dass sich die geforderte Systemfunktionalität mit ihnen zeigen lässt, ohne dass weitere Informationen über die Teilkomponenten des Systems notwendig sind. Insbesondere wird dazu keine vollständige Verhaltensspezifikation notwendig sein.

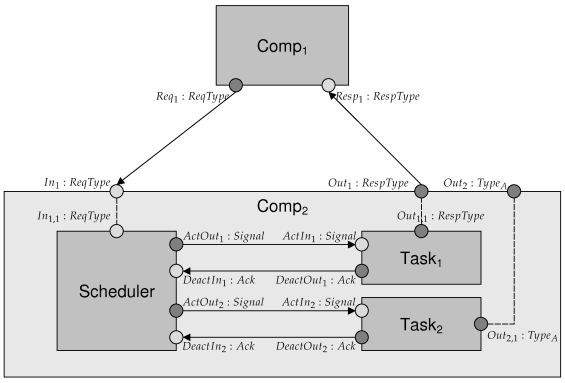

Abbildung 6: Systembeispiel

Die Abbildung 6 zeigt ein System aus zwei Komponenten, die über zwei Kanäle miteinander kommunizieren. Wie spezifizieren die Anforderungen an einen Teilaspekt der Systemfunktionalität zunächst informal:

R1) Das System besteht aus zwei Komponenten $Comp_1$ und $Comp_2$.

R2) $Comp_1$ kann eine *Request*-Nachricht an $Comp_2$ schicken

R3) $Comp_2$ kann eine *Response*-Nachricht an $Comp_1$ schicken

R4) $Comp_2$ enthält zwei Tasks $Task_1$ und $Task_2$.

R5) $Task_1$ und $Task_2$ können von einem Scheduler aktiviert und deaktiviert werden.

R6) Nur eine der Tasks kann gleichzeitig aktiv sein.

R7) Sobald $Comp_2$ eine *Request*-Nachricht von $Comp_1$ empfängt, soll $Task_2$ aktiviert werden und eine *Response*-Nachricht erzeugen. Diese wird an $Comp_1$ verschickt.

Die Zeit zwischen Empfang der Request-Nachricht und Versand der Response-Nachrichten durch $Comp_2$ darf 20 Systemstakte nicht überschreiten, vorausgesetzt der Abstand zwischen zwei Request-Nachrichten beträgt ebenfalls mindestens 20 Takte.

Nun sollen die Anforderungen formalisiert werden. Die Anforderungen R1 bis R4 beschreiben die Struktur des Systems und der Komponenten. Ihre Formalisierung besteht damit im Entwurf einer passenden Systemstruktur, wie beispielsweise auf Abbildung 6.

Die Anforderungen R5 bis R7 beziehen sich nicht nur auf die Struktur, sondern auch auf

Ausführungen des Systems. Wie auf S. 6 angesprochen, wollen wir eine Spezifikation von Systemeigenschaften ohne vollständige Verhaltensspezifikation zu ermöglichen. Dafür werden die Anforderungen als Eigenschaften über Kommunikationsgeschichten des Systems formalisiert.

Als erster Schritt müssen die verwendeten Datentypen deklariert werden:

- $ReqType = \{req\}$: Request-Nachricht.
- $RespType = \{resp\}$: Response-Nachricht.
- $Signal = \{act, deact\}$: Signal zur Aktivierung bzw. Deaktivierung einer Task.
- $Ack = \{ack\}$: Deaktivierungsbestätigung durch eine Task.
- $Type_A$: dieser Datentyp wird nicht näher spezifiziert, da dies für die betrachteten Systemanforderungen nicht benötigt wird.
- $TaskState = \{Active, Idle\}$: Abstrakter Taskzustand. Der Zusammenhang zwischen Taskzustand und der Kommunikationsgeschichte der Task wird später definiert.

Folgende Abkürzungen und Deklarationen sollen zur besseren Lesbarkeit formal spezifizierter Eigenschaften dienen:

- $messages(s, i, j)$: Menge aller Nachrichten in s zwischen den Zeitpunkten i und j:

$$messages(s, i, j) := \{m \mid m = s.t \wedge i \leq t \leq j\} \setminus \{NoMsg\} \tag{3}$$

- $TDeact_i \in \mathbb{N}$: Maximale Zeit zwischen Empfang des Deaktivierungssignals und Versand der Deaktivierungsbestätigung durch $Task_i$ für $i \in \{1,2\}$.
- $TResp \in \mathbb{N}$: Maximale Zeit zwischen Aktivierung der $Task_1$ und dem Versand der Response-Nachricht.
- $TReqResp = 20$: Maximale zulässige Zeit zwischen Empfang der Request-Nachricht und Versand der Response-Nachricht durch $Comp_2$.
- $TaskState_i \in TaskState$: Abstrakter Zustand der $Task_i$ für $i \in \{1,2\}$. $TaskState_i.t$ bezeichnet dabei den Zustand der $Task_i$ zum Zeitpunkt t. $TaskState_i$ kann damit als Strom von Zuständen der $Task_i$ aufgefasst werden.
- $isActive(Task_i, t)$: Aktivierungszustand einer Task

$$isActive(Task_i, t) := (TaskState_i.t = Active) \tag{4}$$

Die Anforderungen an das System sollen nun formalisiert werden. Die strukturellen Anforderungen R1 bis R4 wurden durch den Entwurf der Systemstruktur (Abb. 6) realisiert. Die verbleibenden Anforderungen R5 bis R7 müssen als Eigenschaften über Kommunikationsgeschichten des Systems formalisiert werden.

FR5) Die Anforderung R5 kann wie folgt aufgeschrieben werden:

$$\forall t \in \mathbb{N}. \forall i \in \{1,2\}. \big(ActOut_i.t = act \;\Rightarrow\; \exists t_0 > t. (isActive(Task_i, t_0)) \big) \tag{5}$$

Wie schnell die Task aktiv wird, wird später in einer Annahme definiert.

FR6) Nach R6 darf nur eine der beiden Tasks gleichzeitig aktiv sein:

$$\forall t \in \mathbb{N}. \neg(isActive(Task_1, t) \wedge isActive(Task_2, t)) \tag{6}$$

FR7) Laut R7 soll nach Empfang einer Request-Nachricht eine Response-Nachricht spätestens nach $TReqResp$ Systemtakten versendet werden. Dabei wird vorausgesetzt, dass zwischen zwei Request-Nachrichten mindestens $TReqResp$ Takte liegen.

$$\begin{aligned}&\big(\,\forall t_1,t_2 \in \mathbb{N}.\,\big((\,In_1.t_1 = req \,\wedge\, In_1.t_2 = req \,\wedge\, t_1 < t_2\,) \,\Rightarrow\\ &\quad(\,t_2 - t_1 \geq TReqResp\,)\big)\big) \,\Rightarrow\\ &\big(\,\forall t \in \mathbb{N}.\big(\,In_1.t = req \,\Rightarrow\, resp \in messages(Out_1, t+1, t+TReqResp)\,\big)\,\big)\end{aligned} \tag{7}$$

Damit die spezifizierten Anforderungen erfüllt werden können, werden Annahmen über das System bzw. seine Komponenten benötigt, aus denen sich die Anforderungen herleiten lassen. Diese Annahmen werden im Allgemeinen nicht eindeutig durch die Anforderungen bestimmt, denn verschiedene Annahmen können dieselben Anforderungen implizieren.

FA1) Der Zustand einer Task zum Zeitpunkt t kann aus ihrer Kommunikationsgeschichte bis zu diesem Zeitpunkt bestimmt werden. Eine Task ist genau dann aktiv, wenn sie ein Aktivierungssignal erhalten hatte und noch keine Deaktivierungsbestätigung verschickt hatte (auch nach Empfang eines Deaktivierungssignals ist eine Task aktiv, bis sie die Deaktivierung bestätigt):

$$\begin{aligned}&\forall t \in \mathbb{N}.\forall i \in \{1,2\}.\big(\,isActive(Task_i, t) \,\Leftrightarrow\\ &\quad\exists t_0 < t.\,(\,ActIn_i.t_0 = act \,\wedge\, \neg\, ack \in messages(DeactOut_i, t_0, t-1)\,)\,\big)\end{aligned} \tag{8}$$

Die Formel impliziert unter anderem, dass eine Task zu Beginn eines Systemlaufs ($t = 0$) inaktiv ist, weil kein $t_0 \in \mathbb{N}$ mit $t_0 < 0$ existiert.

Zusammen mit (4) ergibt sich, dass der Zustand $TaskState_i.t$ einer Task zum Zeitpunkt t eine Funktion der Kommunikationsgeschichte dieser Task bis zu diesem Zeitpunkt ist. Der Zustandstrom $TaskState_i$ dient also zum besseren Verständnis des Systems, fügt aber keine neuen Informationen hinzu.

FA2) Erhält die $Task_i$ ein Deaktivierungssignal, so verschickt sie nach höchstens $TDeact_i$ Systemtakten eine Deaktivierungsbestätigung. Diese wird auch dann verschickt, wenn die Task bereits vor Empfang des Deaktivierungssignals inaktiv war:

$$\begin{aligned}&\forall t \in \mathbb{N}.\forall i \in \{1,2\}.\,\big(\\ &\quad ActIn_i.t = deact \,\Rightarrow\, ack \in messages(DeactOut_i, t, t+TDeact_i)\,\big)\end{aligned} \tag{9}$$

FA3) Erhält die $Task_1$ ein Aktivierungssignal, so verschickt sie nach spätestens $TResp$ Systemtakten eine Response-Nachricht:

$$\forall t \in \mathbb{N}.\,\big(\,ActIn_1.t = act \,\Rightarrow\, resp \in messages(Out_1, t, t+TResp)\,\big) \tag{10}$$

FA4) Der Scheduler aktiviert nicht mehr als eine Task gleichzeitig:

$$\forall t \in \mathbb{N}.\big(\,\neg\,(\,ActOut_1.t = act \,\wedge\, ActOut_2.t = act\,)\,\big) \tag{11}$$

FA5) Wenn eine Task aktiv ist, verschickt der Scheduler keine Aktivierungssignale.

Bevor wir diese Annahme formulieren, definieren wir eine Abkürzung, die den Aktivierungszustand einer Task aus der Sicht der Scheduler-Komponente betrachtet:

$$\begin{aligned} &Sched\_TaskActive(Task_i,t) := \big(\\ &\quad \exists t_0 < t.(\, ActOut_i.t_0 = act \ \wedge \ \neg \, ack \in messages(DeactIn_i, t_0+1, t-1)) \big) \end{aligned} \tag{12}$$

Nach dieser Definition wird der Scheduler eine Task niemals für inaktiv halten, wenn sie aktiv ist:

$$\forall t \in \mathbb{N}. \forall i \in \{1,2\}. \big(\neg Sched\_TaskActive(Task_i,t) \ \Rightarrow \ \neg isActive(Task_i,t) \big) \tag{13}$$

Der Scheduler hält die Task einen Systemtakt vor ihrer tatsächlichen Aktivierung bereits für aktiv und erfährt erst einen Systemtakt nach ihrer Deaktivierung, dass sie inaktiv ist. Diese Verzögerungen sind dadurch bedingt, dass ein Kanal zur Übertragung einer Nachricht einen Systemtakt benötigt.

Die Anforderung, dass der Scheduler keine Aktivierungssignale verschickt, während eine Task aktiv ist, kann nun ausgehend von der Kommunikationsgeschichte der Scheduler-Komponente formuliert werden:

$$\forall t \in \mathbb{N}. \forall i \in \{1,2\}. \big(\, Sched\_TaskActive(Task_i,t) \ \Rightarrow \ \neg ActOut_i.t = act \big) \tag{14}$$

FA6) Falls alle Tasks inaktiv sind, und eine Request-Nachricht empfangen wird, aktiviert der Scheduler unmittelbar die $Task_1$:

$$\begin{aligned} &\forall t \in \mathbb{N}. \big(\ (\, In_1.t = req \ \wedge \ \neg (\exists i \in \{1,2\}. \, Sched\_TaskActive(Task_i,t)) \ ) \ \Rightarrow \\ &\quad ActOut_1.t = act \ \big) \end{aligned} \tag{15}$$

FA7) Falls eine Task aktiv ist, und eine Request-Nachricht empfangen wird, schickt der Scheduler ein Deaktivierungssignal an die aktive Task:

$$\begin{aligned} &\forall t \in \mathbb{N}. \forall i \in \{1,2\}. \big(\ (In_1.t = req \ \wedge \ Sched\_TaskActive(Task_i,t)) \ \Rightarrow \\ &\quad ActOut_i.t = deact \ \big) \end{aligned} \tag{16}$$

FA8) Falls eine Task aktiv ist, und eine Request-Nachricht empfangen wird, aktiviert der Scheduler die $Task_1$, nachdem die aktive Task ihre Deaktivierung bestätigt.

Zur bequemeren Formulierung definieren wir eine weitere Abkürzung, über die ermittelt werden kann, ob der Scheduler eine Request-Nachricht empfangen hatte, anschließend ein Deaktivierungssignal an die momentan aktive Task verschickt hatte, und noch keine Deaktivierungbestätigung erhielt:

$$\begin{aligned} &Sched\_Req\_Awaiting\_Deactivation(t) := \big(\\ &\quad \exists t_1, t_2 \in \mathbb{N}. \ \exists i \in \{1,2\}. \ (\, t_1 < t_2 \ \wedge \ t_2 < t \ \wedge \\ &\quad (In_1.t_1 = req \ \wedge req \notin messages(In_1, t_1+1, t)) \setminus * \text{Last } req * \setminus \wedge \\ &\quad (ActOut_i.t_2 = deact \ \wedge \ ack \notin messages(DeactIn_i, t_2+1, t)) \ ) \ \big) \end{aligned} \tag{17}$$

Nun kann die Annahme formuliert werden, dass der Scheduler nach Empfang einer Request-Nachricht und Deaktivierung der aktiven Task die $Task_1$ aktiviert:

$$\forall t \in \mathbb{N}.\exists i \in \{1,2\}.\big(\ (Sched\_Req\_Awaiting\_Deactivation(t-1) \wedge DeactIn_i.t = ack)\ \Rightarrow\ ActOut_1.t = act\ \big) \tag{18}$$

FA9) Für die Verarbeitungszeiten $TDeact_i$ und $TResp$ soll gelten:

$$\forall i \in \{1,2\}.\ TDeact_i + TResp + 5 \leq TReqResp \tag{19}$$

Auf S. 8 wurde $TReqResp = 20$ definiert. Die zusätzlichen 5 Takte sind eine obere Schranke für Zeit zur Kommunikation innerhalb von $Comp_2$. Eine mögliche Verfeinerung dieser Annahme zu Annahmen für einzelne Teilkomponenten $Task_1$ und $Task_2$ ist:

a) $TDeact_1 = 5$ b) $TDeact_2 = 8$ c) $TResp = 7$

Ausgehend von den formalisierten Annahmen über das Verhalten der Systemkomponenten können die gestellten Anforderungen formal verifiziert werden, ohne dass eine vollständige maschinenorientierte Verhaltensspezifikation der Systemkomponenten vorhanden ist. Aus Platzgründen verzichten wir auf ausführliche Beweise und geben Beweisideen an:

- Die Anforderung R5, formalisiert durch FR5, ergibt sich direkt aus FA1 – wenn eine Task das Aktivierungssignal erhält, ist sie im nächsten Zeitschritt aktiv.
- Zur Erfüllung der Anforderung R6, formalisiert durch FR6, werden die Annahmen FA4 und FA5 benötigt. Sie stellen sicher, dass der Scheduler niemals zwei Tasks gleichzeitig aktiviert und niemals eine Task aktiviert, wenn eine andere aktiv ist.
- Die Anforderung R7, formal aufgeschrieben durch FR7, benötigt die meisten Annahmen. Zwei Fälle müssen unterschieden werden:

  1. Zum Zeitpunkt des Empfangs der Request-Nachricht ist keine der Tasks aktiv. Gemäß FA6 wird unmittelbar die $Task_1$ aktiviert. Nach weiteren höchstens $TResp$ Systemtakten wird laut FA3 die Response-Nachricht versendet. Die Antwortzeit beträgt damit $TResp + 1$ Takte und liegt wegen FA9 unter der Zeitgrenze $TReqResp$.

  2. Ist bei Empfang der Request-Nachricht eine Task aktiv, so muss diese deaktiviert, und anschließend $Task_1$ aktiviert werden. Dafür sorgen die Annahmen FA7, FA2 und FA8. Nach spätestens $max\{TDeact_i \mid i \in \{1,2\}\} + 3$ Takten ist $Task_1$ aktiv. Nach weiteren höchstens $TResp$ Takten wird die Response-Nachricht versendet. Damit ergibt sich eine Antwortzeit von höchstens $max\{TDeact_i \mid i \in \{1,2\}\} + TResp + 3$ Takten, was laut FA9 die Zeitgrenze $TReqResp$ ebenfalls nicht überschreitet.

Somit verfeinerten wir die formal spezifizierten Anforderungen zu mehreren Annahmen und skizzierten auf ihrer Basis die Beweise zur formalen Verifikation der Anforderungen.

2.2.3 Ansatz zur Top-Down-Spezifikation und -Verifikation

Wie wir sahen, konnten die formalisierten Anforderungen an das System unter den gemachten Annahmen gezeigt werden. Hierbei beziehen sich die Annahmen FA1 bis FA8 auf jeweils nur eine Komponente. Auch die Annahme FA9 wurde zu Annahmen FA9a,

FA9b und FA9c verfeinert, die sich nur auf einzelne Komponenten beziehen. Auf diese Weise wurden Anforderungen, die sich auf mehrere Systemkomponenten beziehen, zu Anforderungen verfeinert, die sich auf jeweils einzelne Systemkomponenten beziehen. Mit anderen Worten, konnten die formalen Anforderungen an das System zu formalen Anforderungen an Teilkomponenten des Systems modularisiert werden, sodass sie sich auf den hierarchischen Top-Down-Entwurf der Systemarchitektur abbilden lassen (Abb. 7).

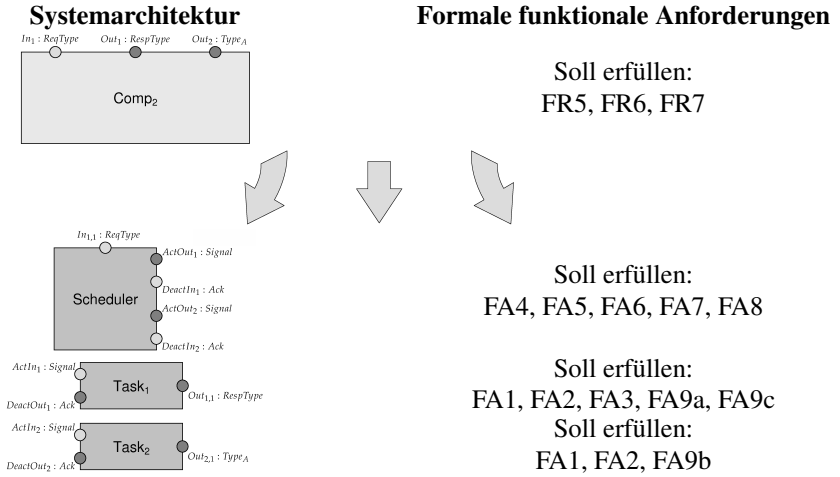

Abbildung 7: Top-Down-Entwicklung der Systemarchitektur und der funktionalen Anforderungen

Die formale Top-Down-Spezifikation funktionaler Anforderungen parallel zum Top-Down-Entwurf der Systemarchitektur bringt wesentliche Vorzüge mit sich:

a) Der Umfang spezifizierter Eigenschaften bleibt handhabbar, indem nur diejenigen Eigenschaften definiert werden, die für die aktuelle Abstraktionsebene und die aktuell vorhandenen Anforderungen benötigt werden. Beispielsweise wurde im Abschnitt 2.2.2 in keiner Weise spezifiziert, was $Task_2$ tut – wir interessierten uns nur für ihre Aktivierungs- und Deaktivierungsspezifikation.

b) Wir behalten bestimmte Freiheitsgrade für spätere Verfeinerung und Implementierung, weil Verhaltenaspekte, die in früheren Entwicklungsschritten unterspezifiziert blieben, im Laufe der weiteren Entwicklung nach Belieben verfeinert werden können, solange die verfeinerte Spezifikation die vorher festgelegten Eigenschaften erfüllt. Beispielsweise konnten wir bei der Verfeinerung von FA9 jedes Wertetupel für $TDeact_1$, $TDeact_2$, $TResp$ wählen, das die Ungleichung (19) erfüllt.

2.3 Verwandte Arbeiten

Die vorliegende Arbeit ist am ehesten mit [SS03] verwandt, welche ebenfalls auf FOCUS basiert – die Beispielspezifikation in Abschnitt 2.2.2 kann als Spezifikation eines *Services* im Sinne von [SS03] verstanden werden. In [JZOA01] wird die Interaktion zwischen Kom-

ponenten durch *Traces* dargestellt, die als Nachrichtenströme aufgefasst werden können. Jedoch ist der Formalismus auf objektorientierte Systeme ausgerichtet – die Nachrichten sind in ihrer Bedeutung eingeschränkt und stellen Methodenaufrufe dar. Die *Rich Components* [DVM+05] sind ein Konzept zur Beschreibung komponentenbasierter Systeme, das auf automatenbasierte Semantik aufbaut und Kommunikationsströme nicht expliziert.

3 Formale Methoden und industrielle Softwareentwicklung

Potentiale der Top-Down-Spezifikation und -Verifikation Softwarespezifikationen in der heutigen industriellen Softwareentwicklung stellen meist textuelle Beschreibungen der geforderten Funktionalität dar, ergänzt durch Graphiken und Diagramme. Zur Feststellung der Konformität einer Implementierung zur Spezifikation werden überwiegend Tests genutzt, die gesondert spezifiziert und durchgeführt werden müssen. Ein wesentliches Problem bei dieser Weise der Spezifikation und Qualitätssicherung ist, wie in Abschnitt 2.1 angesprochen, die Integration einzelner Systemteile und Qualitätssicherung für Funktionalitäten, die gemeinsam durch mehrere Komponenten erbracht werden. Ein Grund dafür ist das Fehlen einer Möglichkeit, die für vernetzte Funktionalitäten benötigten Eigenschaften der Systemkomponenten in der Entwurfsphase präzise zu spezifizieren – textuelle Spezifikationen haben keine präzise Semantik und lassen Interpretationsspielräume für die Spezifikation und Implementierung einzelner Komponenten offen, deren Auswirkungen auf vernetzte Funktionalitäten zum Spezifikationszeitpunkt schwer vorherzusehen sind.

Die Integration formaler Spezifikationstechniken zur modularen Top-Down-Spezifikation vernetzter Systemfunktionalitäten, wie in Abschnitt 2.2.3 skizziert, würde an der Wurzel des Problems ansetzen, indem eine Möglichkeit zur präzisen Spezifikation notwendiger Systemeigenschaften zur Verfügung gestellt würde, mit deren Hilfe aus präzisen Spezifikationen vernetzter Funktionalitäten präzise Anforderungen an einzelne Systemkomponenten abgeleitet werden können. Die Schaffung genau beschriebener funktionaler Schnittstellen zwischen Komponenten würde damit zur Erleichterung der Systemintegration und Steigerung der funktionalen Systemqualität beitragen.

Einbindung in industrielle Softwareentwicklungsprozesse Industrielle Softwareentwicklung ist stark werkzeuggestützt. Integration formaler Spezifikationstechniken in frühe Entwurfsphasen benötigt deshalb mehrere Voraussetzungen:

- Geeignete Notation mit präziser formaler Semantik zur Spezifikation von Komponenteneigenschaften. Die Notation soll eine nachrichtenorientierte Spezifikation funktionaler Eigenschaften von Komponenten ermöglichen.
- Werkzeugunterstützung zur Spezifikation der Systemarchitektur und der funktionalen Komponenteneigenschaften mithilfe einer nachrichtenorientierten formalen Notation.

Werden diese Voraussetzungen von einem Entwurfswerkzeug erfüllt, so erhält der Software-Designer eine Notation an die Hand, welche ihm ein besseres Verständnis der funktionalen Eigenschaften des Systems ermöglicht und dabei hilft, Unklarheiten und Fehler im Systementwurf in frühen Entwicklungsphasen zu vermeiden.

Der Einsatz formaler Methoden für funktionale Eigenschaften kann einen Mehraufwand für den Systementwurf mit sich bringen, stellt aber gleichzeitig einen Qualitätsgewinn in Aussicht, der eine Verringerung des Testaufwands für Softwaresysteme bewirken kann. Der Anteil der testbasierten Qualitätssicherung beträgt im Allgemeinen 25% bis 40% [Roy00] und kann für sicherheitsrelevante Systeme bis zu 80% der Entwicklungskosten ausmachen [Mon99]. Daher kann sich ein höherer Aufwand für formale Methoden, der zu einem geringeren Aufwand bei der Qualitätssicherung führt, lohnen.

4 Fazit und Ausblick

Traditionelle testbasierte Methoden zur Qualitätssicherung stoßen an ihre Grenzen mit steigender Komplexität vernetzter eingebetteter Systeme in modernen Automobilen. Eine besondere Herausforderung liegt in der Integration einzelner Systemkomponenten zum Gesamtsystem, und der damit verbundenen Integration von Komponentenfunktionalitäten zu vernetzten Systemfunktionalitäten.

Formale Spezifikation in frühen Entwurfsphasen hat das Potential, eine präzise Spezifikation vernetzter Funktionalitäten bei der Festlegung der Systemarchitektur zu ermöglichen. Dabei können die Eigenschaften der Systemkomponenten, die für eine vernetzte Systemfunktionalität erforderlich sind, modular auf die einzelnen beteiligten Komponenten abgebildet werden, und somit eine präzise Top-Down-Spezifikation funktionaler Systemeigenschaften parallel zum Top-Down-Entwurf der Systemarchitektur durchgeführt werden.

Die vorgestellte Arbeit befindet sich in der Konzeptionsphase. Wie in Abschnitt 3 beschrieben, muss eine geeignete nachrichtenorientierte Notation entworfen werden, mit der funktionale Systemeigenschaften über Kommunikationsgeschichten spezifiziert werden können. Vorteilhaft wäre die Verwendung bereits entworfener temporaler Logiksprachen, die jedoch mächtiger als CTL/LTL sein müssen, um zeitliche Intervalle spezifizieren zu können. Weitere Entwicklungsrichtungen wären Anbindungen an formal fundierte modellbasierte Entwurfswerkzeuge, sowie an formale Verifikationswerkzeuge wie Modelchecker oder Beweisassistenten (z.B. [NPW02]), um die Möglichkeiten und Grenzen formaler Korrektheitsbeweise für funktionale Eigenschaften zu untersuchen, die beispielsweise für sicherheitskritische Funktionalitäten eingebetteter Systeme von Interesse sind.

Literatur

[Abr96] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[BBB+04] R. Buschermöhle, M. Brörkens, I. Brückner, W. Damm, W. Hasselbring, B. Josko, C. Schulte und T. Wolf. Model Checking (Grundlagen und Praxiserfahrungen). *Informatik Spektrum*, 27(2):146–158, 2004.

[BBR+00] K. Bergner, M. Broy, A. Rausch, M. Sihling und A. Vilbig. A Formal Model for Componentware. Seiten 189–210, 2000.

[BD95] A.-P. Bröhl und W. Dröschel. *Das V-Modell*. Oldenbourg, 1995, 1995.

[BG00] F. Bitsch und M. Gunzert. Formale Verifikation von Softwarespezifikationen in ASCET-SD und MATLAB. Fachtagung Verteilte Automatisierung, Magdeburg, 2000.

[Boe88] B.W. Boehm. A Spiral Model of Software Development and Enhancement. *Computer*, 21(5):61–72, May 1988.

[Bro03] M. Broy. Automotive Software Engineering. In *25th International Conference on Software Engineering*, Seiten 719 – 720. IEEE 2003, 2003.

[BS99] M. Broy und O. Slotosch. Enriching the Software Development Process by Formal Methods. In *Current Trends in Applied Formal Methods 98*, Jgg. 1641 of *LNCS*, 1999.

[BS01] M. Broy und K. Stoelen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer-Verlag, 2001.

[CGP99] E.M. Clarke, O. Grumberg und D.A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

[DVM+05] W. Damm, A. Votintseva, A. Metzner, B. Josko, T. Peikenkamp und E. Böde. Boosting Re-use of Embedded Automotive Applications Through Rich Components. In *Proceedings, FIT 2005 - Foundations of Interface Technologies*, 2005.

[Fri04] H.-G. Frischkorn. Software-Engineering softwareintensiver Systeme im Automobil. Lecture talk, 2004.

[Jon91] C. Jones. *Applied software measurement: assuring productivity and quality*. McGraw-Hill, Inc., New York, NY, USA, 1991.

[JZOA01] E.B. Johnsen, W. Zhang, O. Owe und D. Aredo. Specification of Distributed Systems with a Combination of Graphical and Formal Languages. In *Proceedings of APSEC 2001*, Seiten 105–108. IEEE Computer Society Press, Dezember 2001.

[Kru00] P. Kruchten. *The Rational Unified Process: An Introduction, Second Edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[McM93] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[Mon99] S. Montenegro. *Sichere und Fehlertolerante Steuerungen*. Hanser Verlag, 1999.

[NPW02] T. Nipkow, L.C. Paulson und M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, Jgg. 2283 of *LNCS*. Springer-Verlag, 2002.

[Roy00] W. Royce. Software Management Renaissance – Top 10 Principles of a Modern Software Management Process. 1st Rational Users Conference resources, Munich, 2000.

[Sch02] J. Schlosser. Requirements for Automotive System Engineering Tools. In *20th International Conference on Computer Design (ICCD 2002) Freiburg, Germany*, Seiten 364–369. IEEE Computer Society, 2002.

[SH99] B. Schätz und F. Huber. Integrating Formal Description Techniques. In *FM'99 – Formal Methods, Proceedings of the World Congress on Formal Methods in the Development of Computing Systems*, Jgg. 1709 of *LNCS*, Seiten 1206 – 1225. Springer-Verlag, 1999.

[SS03] B. Schätz und C. Salzmann. Service-Based Systems Engineering: Consistent Combination of Services. In *Proceedings of 5th International Conference on Formal Engineering Methods, ICFEM '03*, Jgg. 2885 of *LNCS*, Seiten 86–104, 2003.