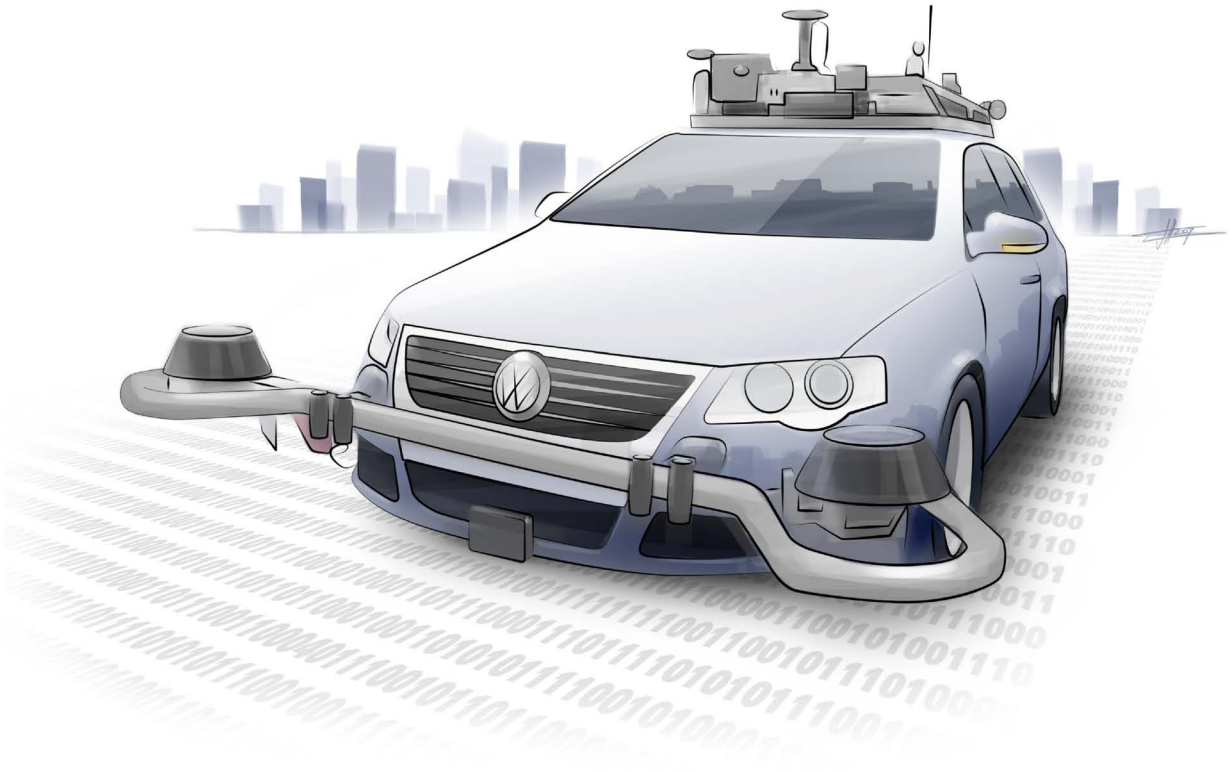


Tagungsband des Modellierungs-Workshops

Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen

Torsten Klein
Bernhard Rumpe

carmeq
software & systems

Software Systems Engineering

Informatik-Bericht 2008-01, Carl-Friedrich-Gauß-Fakultät, Technische Universität Braunschweig

[KR08] T. Klein, B. Rumpe.
Tagungsband Modellierungs-Workshop MBEFF: Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen. Berlin, März 2008,
Informatik-Bericht 2008-01, CFG-Fakultät, TU Braunschweig, 2008.
www.se-rwth.de/publications

Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen

Workshop auf der Modellierung 2008, 12.-14. März 2008

Automobile Anwendungen erreichen eine solche Komplexität, dass traditionelle Techniken der Software-Entwicklung nicht mehr ausreichen. Die hohen Qualitätsanforderungen einerseits, die starken zeitlichen Randbedingungen an die Entwicklung andererseits können nur mit dem Einsatz einer modellbasierten Entwicklung erfüllt werden. Da diese Modelle in der Regel ausführbar sind, können bereits in frühen Entwicklungsphasen Simulationen durchgeführt und die Funktionen durch Versuche im Fahrzeug erprobt und validiert werden. Dabei werden Modelle zu verschiedenen Zwecken in je verschiedenen Phasen der Entwicklung eingesetzt: ein Funktionsmodell dient zur Entwicklung der Funktion an sich; das Streckenmodell beschreibt den Anteil der Systemumgebung, welcher in Interaktion mit der Funktion tritt; das Testmodell stimuliert Funktions- und Streckenmodell, um die Funktion systematisch zu prüfen; aus dem Implementierungsmodell wird schließlich der Applikationscode für das Steuergerät erzeugt. Der Begriff Modell in der modellbasierten Entwicklung bezieht daher auf recht verschiedene Anwendungen. Mittlerweile sind verschiedene Werkzeugketten für die modellbasierte Entwicklung etabliert. Bedeutende Werkzeuge sind etwa MATLAB/Simulink/Stateflow, ASCET SD, SCADE, TargetLink, RealTime Workshop (Embedded Coder). Einige mit dieser Toolkette erstellten Modelle für Softwarefunktionen sind bereits heutzutage in Steuergeräten verbaut und haben so die Tragfähigkeit der Vorgehensweise nachgewiesen.

Mit dem sich stetig verbreitenden Einsatz des modellbasierten Entwicklungsparadigmas entstehen neue Fragestellungen, die in die diesem Workshop aufgegriffen werden. Folgende Themenfelder sollen diskutiert werden:

- Bedeutung von AUTOSAR für die modellbasierte Entwicklung
- Einbettung von Modellen in den Entwicklungsprozess (Abstraktionsebenen, Durchgängigkeit)
- Modellbasierte Entwicklung sicherheitskritischer Systeme
- Modellierung nicht-funktionaler Eigenschaften
- (Umgebungs-)Modelle in komplexen intelligenten Fahrfunktionen
- Konzepte und Lösungsansätze zur Wiederverwendung von Modellen
- Absicherung von Codegeneratoren
- Nutzung von Metamodellen zur Prozess- und Methodikdefinition

Der Workshop soll den Austausch von industriellen Anforderungen und akademischen Lösungsvorschlägen fördern. Insbesondere werden aus dem industriellen Umfeld Beiträge gesucht, die aktuelle Herausforderungen und Rahmenbedingungen aus der automobilen Praxis der modellbasierten Entwicklung vermitteln. Aus dem akademischen Umfeld sind technologische Konzepte gefragt, die zu einer Lösung beitragen können und den Bezug zu automobilen Anwendungen dokumentieren.

Beiträge des Workshops

Erfahrungen bei der Einführung der modellbasierten AUTOSAR-Funktionsentwicklung

Dr. Florian Wohlgemuth*, Christian Dziobek*, Dr. Thomas Ringler**

* Mercedes-Benz Cars Entwicklung
** Group Research & Advanced Engineering
Daimler AG
71059 Sindelfingen
Florian.Wohlgemuth@daimler.com
Christian.Dziobek@daimler.com
Thomas.Ringler@daimler.com

Abstract: Die modellbasierte Entwicklung von Komfort- und Innenraumfunktionen hat sich in den letzten Jahren etabliert. Ausgehend von den bisherigen Erfahrungen mit der modellbasierten Funktionsentwicklung werden die Vorteile der standardisierten AUTOSAR-Architektur und zugehöriger Beschreibungsformate aufgezeigt. Der Beitrag beschreibt das Vorgehen bei der schrittweisen Umstellung der modellbasierten Entwicklung auf die AUTOSAR-Architektur und berichtet von den bisherigen Erfahrungen.

1 Einführung

1.1 Stand der Modellbasierten Entwicklung

Fahrzeugfunktionen und speziell Komfort- und Innenraumfunktionen werden seit einigen Jahren bei der Daimler AG modellbasiert entwickelt [DW07]. Schwerpunkt der modellbasierten Entwicklung im Innenraum sind Hauptfunktionen der Zentralsteuergeräte (Central Body Computers), in denen jeweils mehrere Funktionen integriert sind. Im größeren Umfang wurde die modellbasierte Entwicklung im Innenraum bei der aktuellen C-Klasse (Baureihe 204) eingeführt, die Funktionen sind in Abbildung 1 dargestellt. In den aktuell in der Serienentwicklung befindlichen Baureihen wird der modellbasierte Ansatz konsequent weiter verfolgt, dabei steigt die Anzahl der zu integrierenden Funktionen weiter an.

Die Entwicklung der Fahrzeugfunktionen geschieht innerhalb der Daimler AG arbeitsteilig anhand von mehreren Rollen. Der Komponentenverantwortliche ist für die Hardware-Komponente, also das Steuergerät, verantwortlich. Der Systemverantwortliche ist für eine E/E[1]-Funktion (auch E/E-System genannt) verantwortlich. Daraus folgt, dass sich die Spezifikation eines Steuergeräts aus dem Komponentenlastenheft und einer oder mehrerer Systembeschreibungen zusammensetzt. Diese Rollenverteilung besteht während der Entwicklung und darüber hinaus während des Lebenszyklus der Baureihe fort.

Für die Modellierung eines Systems gibt es die Rolle eines Modellierers. Hier wird die Spezifikation in ein Simulink-Modell umgesetzt, welches mittels Simulation und entsprechenden Visualisierungen mit dem Systemverantwortlichen diskutiert und abgestimmt werden kann.

Weiterhin setzt ein Modellierer die Testspezifikation des Systemverantwortlichen in eine Testimplementierung mit dem Werkzeug TPT [TPT] um.

Je Entwicklungszyklus erhält der Lieferant sowohl das Simulink-Modell als auch das TPT-Modell mit der Testfallimplementierung und ist für die Umsetzung des Modells in Software verantwortlich. Um diese Umsetzung zu erleichtern, werden teilweise auch bereits vorskalierte TargetLink-Modelle an den Lieferanten übergeben. Zur Modellabsicherung ergänzt dieser das TPT-Modell um weitere Testfälle, um eine 100% C1-Abdeckung der Modultests zu erreichen.

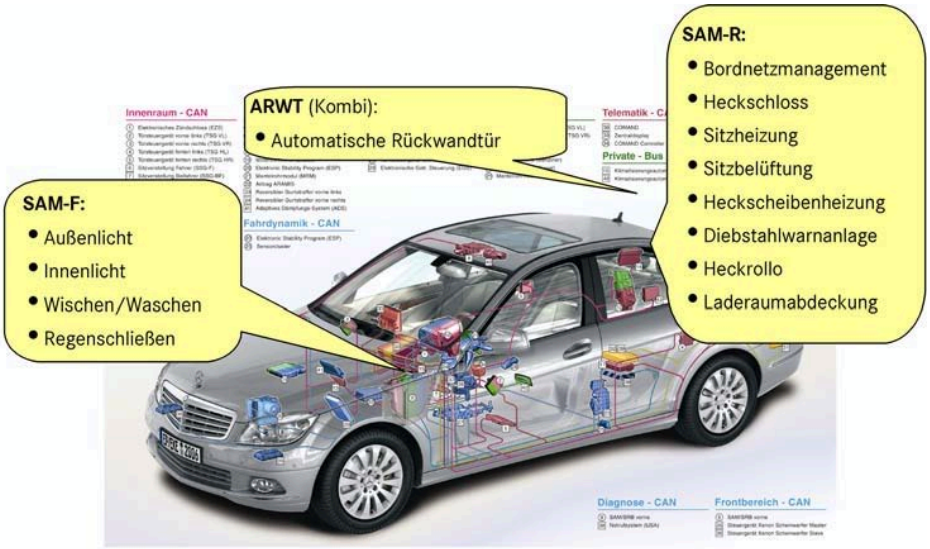


Abbildung 1: Modellbasiert erstellte Funktionen der C-Klasse (Baureihe 204)

[1] Elektrik/Elektronik

Der Fahrzeughersteller hat durch die modellbasierte Entwicklung im Wesentlichen folgende Vorteile:

- Frühzeitige Reifegradabsicherung durch Simulation
- Darstellung und Absicherung von Funktionen im Fahrzeug bereits vor Verfügbarkeit der Steuergeräte durch Rapid-Prototyping
- Lieferantenunabhängige Funktionsentwicklung und Weiterentwicklung
- Wiederverwendung von Funktionen und Schutz geistigen Eigentums

Durch die Simulation kann die Funktionalität bereits früh mit dem Systemverantwortlichen und weiteren Kollegen durchgesprochen werden. Unklarheiten in der Spezifikation werden dadurch früh sichtbar und können innerhalb der Organisation beseitigt werden. Weiterhin kann das Modell zu einem frühen Zeitpunkt getestet werden. Die Reife der Spezifikation und damit auch des Modells kann dadurch bereits in Anfangsphasen der Steuergerätentwicklung erhöht werden.

Mittels Code-Generierung wird diese Reife auf das Steuergerät und letztlich auch auf das Fahrzeug übertragen. Damit ist es auch möglich, die vollständige Funktionalität bereits in frühen Fahrzeugaufbauten darzustellen. Liegt zu diesem Zeitpunkt noch kein Steuergerät vor, kann mit Rapid-Prototyping-Systemen die Funktionalität dennoch in Fahrzeugen erprobt werden.

Die Funktionsentwicklung ist im Vorfeld einer Serienentwicklung unabhängig vom künftigen Lieferanten möglich und eröffnet damit bei der Lieferantenauswahl neue Freiheitsgrade. Modelle können von einer Baureihe zur nächsten Baureihe übertragen werden, selbst wenn unterschiedliche Lieferanten die Steuergeräte entwickeln.

Für den Einsatz einer modellbasierten Entwicklung eignen sich besonders Funktionen bei denen die Wiederverwendung in anderen Baureihen geplant oder absehbar ist sowie bei Funktionen mit hoher Komplexität und Innovationspotential oder bei wettbewerbsdifferenzierenden Funktionen mit Anforderungen an den Schutz des geistigen Eigentums.

1.2 Erfahrungen mit der modellbasierten Entwicklung

Die modellbasierte Funktionsentwicklung hat sich im Hause Daimler im Bereich Innenraum bewährt. Die erstellten Funktionen können mit einer hohen Reife oder zumindest wenigen bekannten funktionalen Fehlern in die ersten Entwicklungsfahrzeuge integriert werden.

Die entwickelten Funktionen kommen in mehreren Baureihen zum Einsatz und werden teilweise auch in bereits laufende Baureihen zu Modellpflegemaßnahmen eingeführt. Die Vorteile der modellbasierten Entwicklung haben sich hier insbesondere bei der Umsetzung zusätzlicher Features in bestehende Funktionen bestätigt. Die so entstandene Bibliothek an Modellen wird kontinuierlich gepflegt und so konsistent zur Systemspezifikation gehalten.

Die Integration der modellbasiert erstellten Funktionen in das Steuergerät durch den Lieferanten erfolgt bisher noch mit einem hohen manuellen Aufwand. Dieser war sehr stark von der jeweiligen Softwarearchitektur des Lieferanten abhängig, auch wenn der Kommunikationsteil der Basissoftware durch den Fahrzeughersteller vorgegeben wurde. Gegebenenfalls musste die Softwarearchitektur angepasst oder speziell erweitert werden. Eine bisher fehlende vollständig standardisierte Softwarearchitektur führte teilweise zu umfangreichen Abstimmungsgesprächen mit den jeweiligen Lieferanten.

Der Abstimmungsbedarf geht jedoch über die Softwarearchitektur hinaus, so muss auch die Beschreibung der Meta-Daten zur Integration der Funktionen, wie z.B. Schnittstellenlisten der Funktionen oder die Abbildung von Applikationssignalen auf Bussignale gemeinsam zwischen Fahrzeughersteller und Steuergerätelieferant definiert werden.

Die Voraussetzung für einen prozesssicheren und breiteren Einsatz der modellbasierten Entwicklung ist somit eine einheitliche lieferantenübergreifende Softwarearchitektur und eine standardisierte Beschreibung der Metadaten.

1.3 Der AUTOSAR-Standard

Der AUTOSAR-Standard [AR] definiert eine Softwarearchitektur für Steuergeräte, ein Integrationsverfahren und die dazu notwendigen Austauschformate. AUTOSAR erfüllt damit weitgehend die oben genannten Anforderungen für eine prozesssichere Integration von modellbasiert entwickelten Funktionen.

AUTOSAR gliedert die Applikationssoftware eines Steuergeräts in mehrere Softwarekomponenten (Software-Components, SWC) die mittels einer Middleware (der RTE – Real Time Environment) miteinander kommunizieren. SWCs kapseln und typisieren die Software und ermöglichen einen Datenaustausch nur über wohldefinierte Schnittstellen. Hierzu werden *Ports* angelegt, die über ein *Interface* typisiert werden. Ein Sender/Receiver-Interface kann aus einem oder mehreren Daten-Elementen bestehen, die wie programmiersprachliche Variablen durch Datentypen typisiert werden. Als weitere Interface-Variante existiert das Client/Server-Interface, welche aus einer oder mehreren Operationen besteht. Mittels Sender/Receiver-Interfaces wird ein Datenfluss zwischen SWCs modelliert, mit Client/Server-Interfaces geht zusätzlich ein Kontrollfluss einher. Dazu müssen jedoch explizit *Assembly-Connections* zwischen den Ports zweier SWC-Instanzen gezogen werden; dies ist jedoch nur dann möglich, wenn die zu den Ports gehörenden Interfaces identisch oder zumindest gemäß standardisierter Regeln zueinander kompatibel sind.

Für die Integration auf ein Steuergerät sind zwei Mapping-Schritte erforderlich: Erstens ein Mapping der SWC-Instanzen auf Steuergeräte und zweitens im Fall von nicht-lokaler Kommunikation ein Mapping der Daten-Elemente auf Netzwerk-Signale.

Weiterhin arbeitet AUTOSAR an der Standardisierung von Schnittstellen zwischen Applikationen und deren Aktorik und Sensorik; hier erhofft man sich langfristig eine leichtere Verbindung von Hersteller-Funktionsmodellen mit der Aktorik/Sensorik-Ansteuerung des Steuergerätelieferanten.

1.4 AUTOSAR-Einführungsstrategie

Für die nächste Steuergeräte-Generation bei der Daimler AG im Innenraum wird ein erster Schritt in Richtung der AUTOSAR-Architektur angegangen. Die Einführung erfolgt hier im Schichtenmodell der AUTOSAR-Softwarearchitektur von oben nach unten, d.h. bei den Applikations-Softwarekomponenten und der RTE beginnend.

Dazu wird die Softwarearchitektur konsequent in Applikations- und Basissoftwareanteile aufgeteilt, die über eine definierte Schnittstelle miteinander kommunizieren. Basis für die Definition dieser Schnittstelle ist der AUTOSAR-Standard, der hier jedoch teilweise eingeschränkt und an einigen Stellen präzisiert wird. Die Standard-Softwarearchitektur basiert in diesem Schritt weiterhin auf dem etablierten Daimler-Standard-Core und wird um ausgewählte AUTOSAR-Softwaredienste (wie beispielsweise Memory-Management von NVRAM[2]-Daten) ergänzt.

In diesem ersten Schritt ist die Netzwerk-Kompatibilität der so entwickelten Steuergeräte mit klassisch entwickelten Steuergeräten weiterhin gegeben. Damit ist eine schrittweise Einführung der AUTOSAR-Technologie möglich.

1.5 Übersicht

Der vorliegende Beitrag ist wie folgt gegliedert. Im Kapitel 2 wird das konkrete Vorgehen bei der Einführung der modellbasierte Entwicklung auf Basis der AUTOSAR-Architektur in dem oben genannten ersten Migrationsschritt vorgestellt. Kapitel 3 berichtet von den bisherigen Erfahrungen bei der AUTOSAR-Einführung, insbesondere im Hinblick auf die eingesetzten Werkzeuge. Der Beitrag schließt mit einer Zusammenfassung und einem Ausblick auf die zukünftigen Herausforderungen bei der weiteren breiten Etablierung von AUTOSAR im Bereich der modellbasierten Funktionsentwicklung.

2 Modellbasierte Entwicklung auf Basis der AUTOSAR-Architektur

2.1 Zwei Modellierungsebenen

Mit AUTOSAR geht eine Modellierung auf zwei Ebenen einher. Neben der Modellierung des Verhaltens der Funktionen z.B. in Simulink/TargetLink ist nun zusätzlich eine formale Beschreibung der Schnittstellen der Funktionen in Form von Softwarekomponenten und deren Vernetzung erforderlich. Die möglichen Vorgehensweisen hierfür sind offensichtlich:

---

[2] NVRAM: non volatile RAM

Für neue Funktionen kann top-down zuerst die Zergliederung der Funktionalität in mehrere Softwarekomponenten und die Definition deren Schnittstellen erfolgen. Anschließend wird das Verhalten der so zergliederten Softwarekomponenten modelliert.

Bei bereits fertig modellierten Funktionen kann bottom-up aus den Modellschnittstellen die SWC-Beschreibung erzeugt werden. Die entstandenen SWCs müssen anschließend noch untereinander verbunden werden.

Bei einer schrittweisen Einführung ist ein vollständiger Top-Down-Entwurf eines gesamten Fahrzeuges nicht leistbar. Umgekehrt liegen für ein Steuergerät nicht alle Funktionen von Anfang an modelliert vor. Hier hat sich eine „Meet-in-the-middle"-Strategie bewährt, die für ein Steuergerät

- aus existierenden Funktionsmodellen die SWC-Beschreibung ableitet,
- bei neuen Funktionen zunächst die Schnittstellen über SWCs definiert,
- Sensor-/Aktor SWCs mittels fester Regeln automatisch erzeugt.

Die so entstandenen SWCs werden in einer *Composition* zusammengefasst und untereinander vernetzt, die verbliebenen unverbundenen Ports werden nach außen geführt und damit zu Ports der Composition. Diese Ports repräsentieren nun die Kommunikationsschnittstelle des Steuergeräts und die durch die Ports referenzierten Daten-Elemente können auf die durch die Kommunikations-Matrix vereinbarten Signale des Steuergeräts abgebildet (*gemappt*) werden.

Damit kann die SWC-Struktur eines Steuergeräts mit vertretbarem Aufwand erstellt werden. In den folgenden Abschnitten werden diese Tätigkeiten genauer erläutert.

2.2 Festlegen der AUTOSAR-Interfaces

Basis für die Entwicklung von Softwarekomponenten ist eine gemeinsame Datenbank von Interface- und Datentyp-Definitionen, die durch den SWC-Entwickler bei der Definition der SWC-Ports genutzt werden können. Diese Datenbank wurde aus den Signaldefinitionen der K-Matrix abgeleitet und wird mit dieser laufend abgeglichen.

Die Ableitungsregeln hierzu legen fest, dass für jedes K-Matrix-Signal ein Interface mit einem Datenelement erzeugt wird. Dadurch wird der Übergang von der bisherigen K-Matrix-Signal-Welt in die AUTOSAR-Welt erleichtert, da anhand der Namensgleichheit die passenden Interfaces gefunden werden können. Die Strukturierungsmöglichkeiten von AUTOSAR, mehrere Daten-Elemente zu einem Interface zusammenzufassen, werden bei neuen Interfaces genutzt, die abgeleiteten Interfaces profitieren hiervon vorerst nicht.

Um die Kompatibilität der SWC untereinander zu gewährleisten, wird durch Modellierungsregeln vereinbart, dass lediglich Interfaces verwendet werden dürfen, die in der Datenbank vorhanden sind. Gegebenenfalls muss diese Datenbank durch einen Koordinator um zusätzliche Interfaces erweitert werden.

2.2.1 Erstellung neuer Funktionsmodelle

Die Modellierung neuer Funktionen startet mit der Beschreibung der SWC-Schnittstellen in der oben genannten Datenbank. Dazu werden Ports angelegt, welche durch Interfaces typisiert werden. Weiterhin werden hier Runnables und deren Teilmenge der SWC-Schnittstelle vereinbart. Über RTE-Events wird die Zykluszeit der Runnables und gegebenenfalls deren ereignisgesteuerte Aktivierung z.B. beim Empfang eines neuen Signals vereinbart.

Die Funktionsmodellierung startet auf Basis dieser AUTOSAR-Beschreibung. Dazu wird die SWC in das AUTOSAR-XML Format exportiert und auf dieser Basis in TargetLink ein Modellierungsrahmen generiert, der einen Ausgangspunkt für die Modellierung der Funktion bietet. Änderungen an der Schnittstelle werden dann stets in der Datenbank vorgenommen; Änderungen werden im Modellierungswerkzeug über das AUTOSAR-Format nachgezogen.

2.2.2 Migration bestehender Modelle

Bestehende Modelle wurden migriert [DW07]. Dabei wurde die in dem Modell vorhandene Schnittstellendefinition verwendet, um Ports und deren Typisierung abzuleiten. Nahezu 90% der Modellumfänge konnten automatisiert migriert werden. da bereits bisher Modellierungskonventionen festgelegt hatten, bei Eingangs- und Ausgangsvariablen den Namen des zugehörigen K-Matrix-Signals zu verwenden. Zunächst wurde hierbei im TargetLink Data Dictionary die SWC angelegt. Danach wurden im Modell die entsprechenden AUTOSAR-Blöcke eingefügt. Anschließend wurde die bei der Code-Generierung entstehende XML-Datei in die SWC-Datenbank eingespielt. Die weitere Pflege der Schnittstelle kann nun in der Datenbank wie oben beschrieben geschehen und automatisiert in die Modelle zurückgespielt werden.

Bei dieser Migration konnte die modellierte Funktionalität vollständig und unverändert übernommen werden, lediglich die Modellschnittstellen mussten an den Standard angepasst werden.

2.2.3 Erweiterung der Modellfunktionalität

Im Zuge der AUTOSAR-Einführung wurde eine Restrukturierung der Funktionsmodelle durchgeführt. Die Modelle wurden vereinheitlicht und standardisiert. Dies wurde notwendig, da in bisherigen Projekten für die Integration der einzelnen Funktionen handcodierte Rahmensoftware durch den Lieferanten implementiert wurde, die den Zugriff auf Inter-Applikations-Daten, den Standard-Core und hardwarenahe Schichten realisiert hat (siehe Abbildung 2 links). Zusätzlich wurde bisher durch die Rahmensoftware eine Signalkonditionierung durchgeführt.

Mit der Einführung von AUTOSAR werden weite Teile dieser Funktionalität durch die RTE abgedeckt. Um die Anzahl der notwendigen SWCs auf einem Steuergerät und damit den Verwaltungsaufwand gering zu halten, wurde darauf verzichtet die verbliebenen Funktionalitäten in eigenen SWCs abzubilden. Stattdessen wurden diese in die Applikationsmodelle aufgenommen; diese wären:

- Überwachung des Empfangsstatus an Port-Eingängen
- Überwachung des Wertebereichs
- Bilden eines geeigneten Ersatzwertes im Fall von Time-Outs oder Verletzungen des Wertebereichs
- Bereitstellung eines gültigen Ersatzwertes nach dem Kaltstart eines Steuergeräts

Hierfür wurde eine Blockbibliothek aufgebaut, die diese Funktionalitäten zur Verfügung stellt. Aktuell wurde eine Migration von ca. einem Dutzend Modellen durchgeführt.

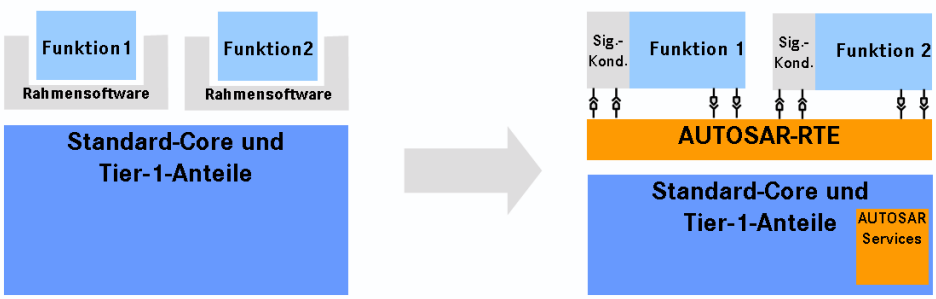


Abbildung 2: Migration bestehender Funktionsmodelle nach AUTOSAR

2.3 AUTOSAR-Werkzeugumgebung

Für die AUTOSAR-Funktionsmodellierung wurde eine entsprechende Werkzeugumgebung aufgebaut und bestehende Werkzeuge erweitert (siehe Abbildung 3 vereinfacht dargestellt).

Ein wesentlicher Baustein der Entwicklungsumgebung ist die Datenbank zur Ablage von AUTOSAR-Interfaces und SWC-Beschreibungen. Sie wurde als Erweiterung einer bestehenden Datenbankanwendung zur K-Matrix-Erstellung realisiert. Neben den AUTOSAR-Modellierungselementen können hierin auch Parameter und NVRAM-Größen beschrieben und mit Werten versehen werden. Durch die Nähe zur K-Matrix Entwicklung können AUTOSAR-Interfaces elegant aus deren Datenbeständen generiert und auch aktualisiert werden. Weiterhin konnte das erprobte Freigabekonzept der K-Matrix-Entwicklung übernommen werden.

Über die Austauschformate von AUTOSAR können die SWC-Beschreibungen im- und exportiert werden. Dadurch ist die Kopplung zum TargetLink Data Dictionary und zu den Autorenwerkzeugen gegeben. Die Autorenwerkzeuge werden zur Vernetzung der SWCs eingesetzt. Diese geschieht dort in der Regel durch grafische Editoren, die eine bessere Visualisierung als rein formularbasierte Datenbankanwendungen bieten. Im Autorenwerkzeug wird auch die Abbildung der nicht-lokalen Daten-Elemente auf Netzwerksignale vorgenommen. Das dadurch entstehende AUTOSAR System-Template wird zur ECU-Integration beim Steuergerätelieferanten eingesetzt.

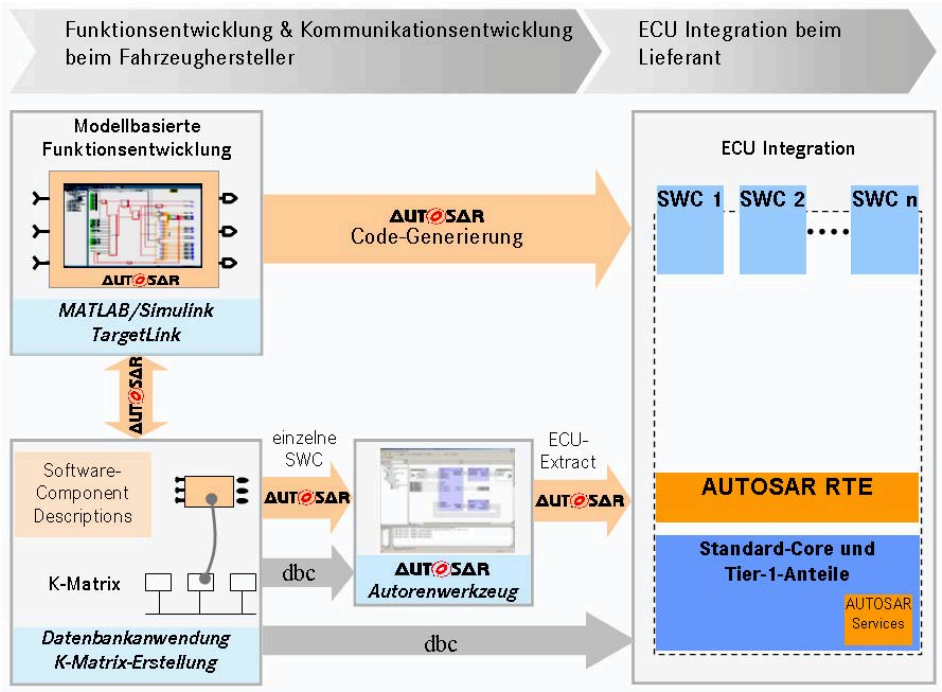


Abbildung 3: AUTOSAR-Werkzeugumgebung

Die Datenbank ist auch als Datenquelle für eine ganzheitliche E/E-Architekturentwicklung in der sehr frühen Konzeptphase einer neuen Baureihe konzipiert. Bestehende SWC können der Datenbank entnommen werden, um damit ein neues Fahrzeug-E/E-Architekturkonzept zu bewerten und abzusichern. Die in dieser Konzeptphase entstehenden Entwürfe von SWC-Strukturen können in die Datenbank übernommen und als Schnittstellenentwurf bei der Verhaltensmodellierung weiterverwendet werden [RSM07].

3 Bisherige Erfahrungen bei der AUTOSAR-Einführung

3.1 Werkzeuge zur Modellierung von SWC-Beschreibung und Verhalten

Bei der Einführung der AUTOSAR-konformen Modellierung stellen die unterschiedlichen und zurzeit noch nicht optimal auf einander abgestimmten Werkzeuge eine besondere Herausforderung dar. Dabei wird die Interoperabilität der Werkzeuge untereinander in erster Linie durch den Austausch der relevanten Schnittstellenobjekte auf Basis der AUTOSAR SWC-Beschreibung schon recht gut realisiert.

Die Mächtigkeit der Beschreibung und die teilweise nicht vollständig vorhandene Entsprechung der Objekte in den jeweiligen Werkzeugen führt zu der Notwendigkeit durch geeignete Nutzer- oder Applikationsprofile und Modellierungsmuster, ein prozesssicheres ‚Roundtrip'-Engineering zu gewährleisten. Einige dieser Aspekte wurde auch innerhalb von AUTOSAR bearbeitet (siehe [AR06a]) und führten zu einem Simulink-Styleguide [AR06b].

Der Prozess für die Erstellung und Verwaltung von SWC-Beschreibungen muss sorgfältig geplant und definiert werden und muss zusätzlich das parallele Arbeiten durch eine Vielzahl von Entwicklern ermöglichen. Wie bereits erläutert wurde hierzu eine zentrale Datenbank zur Ablage der AUTOSAR-relevanten Informationen aufgebaut (siehe Abbildung 3 links unten). Die Datenbank erlaubt einerseits ein paralleles Arbeiten mehrerer Nutzer auf dem gleichen Datenpool und andererseits ist eine kontrollierte Freigabe von AUTOSAR-Artefakten möglich. AUTOSAR-Autorenwerkzeuge hingegen konzentrieren sich momentan stärker auf die Beschreibung von SWC-Netzen, und derzeit weniger auf Versionierung und Multi-User-Fähigkeit.

Die Implementierung der Datenbankanwendung als Individualsoftware im eigenen Hause hat sich zum jetzigen Zeitpunkt der Etablierung des AUTOSAR-Standards als sehr hilfreich erwiesen. Unzulänglichkeiten anderer Werkzeuge beim Austausch von SWCs konnten so abgefedert werden. Hierbei steht die Sicherstellung der systemübergreifenden Konsistenz der SWC-Beschreibungen im Vordergrund. Darüber hinaus muss die AUTOSAR-konforme Entwicklungsumgebung neben der Definition von neuen SWC–Beschreibungen auch den Import von vorhandenen noch nicht AUTOSAR-konformen Funktionen ermöglichen.

3.2 Haupttätigkeiten bei der Anwendung des AUTOSAR-Standards

Im Rahmen des Entwicklungsprozesses von AUTOSAR-konformen Softwarekomponenten gibt es im wesentlichen drei Haupttätigkeiten:

1. Erstellung und Definition von SWC-Beschreibungen
2. Erstellung von Verhaltensmodellen mit MATLAB/Simulink und TargetLink
3. Integration der SWC-Beschreibungen mit einem AUTOSAR-Autorenwerkzeug

1) Mit Rücksicht auf die Modellierungsmöglichkeiten des Verhaltenswerkzeugs Simulink werden zur Modellierung der Kommunikation der Applikationskomponenten untereinander ausschließlich Sender/Receiver-Interfaces eingesetzt. Nur für die Kommunikation zum Diagnose-Fehlerspeicher (*Diagnostics Event Manager*) werden Client/Server-Interfaces wie in [AR06c] spezifiziert eingesetzt. Durch die Beschränkung der Softwarekomponenten auf jeweils nur ein Runnable ist es möglich leicht verständliche und unmittelbar simulierbare Modelle erstellen zu können.

2) Die Verhaltensmodelle werden mit MATLAB/Simulink und TargetLink erstellt. Wichtig ist hierbei eine für den Entwickler nachvollziehbare Abbildung der Softwarekomponentenobjekte auf die Verhaltensmodellobjekte sicherzustellen. Hierbei wurde auf eine weitgehende Kompatibilität zu bisher bestehenden Modellierungsmustern geachtet. Es entsteht so eine komplementäre Struktur der Ports, Interfaces und deren Datenelementen zu Simulink-Signalen bzw. TargetLink Data Dictionary Variablenobjekten. Das bedeutet, dass beim Import von Softwarekomponenten-Beschreibungen die Definition entsprechender Datenobjekte und zugehöriger Modellierungsmuster sichergestellt werden muss. Hier bei sind zwei ‚Use-Cases' zu unterscheiden:

   a) Erstellung neuer Softwarekomponenten:
      Beim Import einer neuen Softwarekomponente werden für alle in der Softwarekomponente definierten Objekte ergänzende Objekte zusätzlich neu angelegt. Das Beinhaltet zum Einen TargetLink Data Dictionary Variablenobjekten und zum Anderen ein zur Softwarekomponenten-Beschreibung konformes Rahmenmodell, in das die Funktion integriert werden kann. Zusätzlich dazu werden über zusätzliche Ports (und ab AUTOSAR R2.1 Calibration-Ports), Parameter und Messobjekte definiert und als TargetLink Data Dictionary Objekte angelegt und dem Entwickler zur Modellierung in der Umgebung bereitgestellt. Optional kann der Entwickler Objekte erweitern und über die Softwarekomponenten-Beschreibung exportieren. Über den Export können so vorgenommene Erweiterungen in die zentrale Datenbank zurück gespielt werden. Durch statische Modellchecks wird die Konformität der implementierten Verhaltensmodelle sichergestellt.

   b) Update existierender Softwarekomponenten:
      Für bereits erstellte Softwarekomponenten muss beim Öffnen die Kompatibilität der Softwarekomponenten-Beschreibungen und des Verhaltensmodells überprüft werden. Bei Abweichungen werden die Unterschiede angezeigt und können dann vom Entwickler interaktiv abgeglichen werden. Neben der Umbenennung von Objekten muss hier die Änderung bzw. Erweiterung von Ports und Interfaces besonders berücksichtigt werden. Funktionen zur Refaktorierung sind hier notwendig um Änderungen in die Verhaltensmodelle schnell und sicher übertragen zu können. Für neue Objekte wie z.B. Ports mit Interfaces und Datenelementen können die im Modell notwendigen Modellierungsmuster generiert werden und stehen so dem Entwickler als initiale Bausteine zur Verfügung. Abschließend wird durch statische Modellchecks die Konformität der implementierten Verhaltensmodelle sichergestellt.

3) Eine manuelle Erstellung einer SWC-Vernetzung in einem AUTOSAR-Autorenwerkzeug hat sich als sehr zeitintensiv erweisen. Daher wurden Möglichkeiten geschaffen, diese Vernetzung weitgehend automatisiert abzuwickeln. Zu diesem Zweck werden beim Export aus der Datenbank AUTOSAR-Beschreibungen generiert, die zusätzliche generierte Aktor-/Sensorkomponenten enthalten, um die Erstellung eines vollständigen Funktionsnetzwerkes zu vereinfachen. Die generierten Aktor-

/Sensorkomponenten dienen als Platzhalter und werden durch den Zulieferer komplettiert.

Die Notwendigkeit des Einsatzes von zwei Entwicklungswerkzeugen (zur Verhaltensmodellierung und zur Schnittstellenbeschreibung) beim Entwickeln von AUTOSAR-konformen Funktionsnetzwerken schafft neue Herausforderungen bezüglich der Unterteilung eines Systems in handhabbare und logisch sinnvolle Softwarekomponenten. Änderungen an der SWC-Komponentenschnittstelle einschließlich des Funktionsverhaltens sind beim Übergang zwischen Modellierungswerkzeugen unterschiedlicher Hersteller immer eine besondere Herausforderung, da die standardisierten Schnittstellen den Roundtrip bei Werkzeugwechsel nur unzureichend unterstützen. Weiterhin werden durch die aktuelle Aufteilung der AUTOSAR-Entwicklungsumgebung in Werkzeuge zur Architekturmodellierung und Systemintegration sowie zur Verhaltensmodellierung mit jeweils eigenständigen Codegeneratoren, die realen Software-Systemschnittstellen zu einem gewichtigen Teil durch die Werkzeugdomäne bestimmt. Eine flexible werkzeugübergreifende Systemmodellierung und Ressourcen-Optimierung stößt hier an ihre Grenzen und Architektur- und Designentscheidungen müssen a priori gut überlegt sein!

3.3 AUTOSAR-Modellierung in Simulink

Die AUTOSAR-Beschreibung ist streng typisiert. Die Modellierungswerkzeuge wie z.B. Simulink aber auch TargetLink sind jedoch schwach typisiert. Dies ist historisch begründet; die Werkzeuge kommen aus der Regelungstechnik, und sind darauf optimiert auf schnellem Weg ein simulationsfähiges Modell zu erhalten. So sind gewisse AUTOSAR-Datentypen in Modellierungswerkzeugen nicht vorhanden, Enumerations beispielsweise werden heute noch nicht korrekt typisiert und geprüft.

Der Softwarearchitektur-Ansatz von AUTOSAR, in vielen Möglichkeiten der Modellierung von Datenfluss durch Sender/Receiver und Kontrollfluss durch Client/Server lässt, sich nur teilweise auf die datenflussorientierte Semantik von Simulink abbilden. Die volle Ausprägung der Modellierung von SWC, Runnables, Access-Points und Ports und vernetzter SWC inklusive der Berücksichtigung von Statussignalen ist heute nur mittels Workarounds und nicht sehr prozesssicher möglich. Simulink lässt sich zusammen mit dem TargetLink AUTOSAR-Blockset als Funktionsmodellierungswerkzeug in eine AUTOSAR-konforme Modellierungsumgebung für den Top-Down-Entwurf der Softwarearchitektur integrieren. Dabei werden die notwendigen Teile des Standards, die für eine Etablierung notwendig sind, unterstützt. Für die Zukunft ist eine semantisch korrekte Integration notwendig, um so den vollen Funktionsumfang von AUTOSAR werkzeugübergreifend nutzen zu können.

4 Zusammenfassung und Ausblick

Die modellbasierte Funktionsentwicklung erfährt durch AUTOSAR eine seit langem geforderte Standardisierung von Beschreibungsformaten und Schnittstellen [DW07].

Die typsicheren AUTOSAR-Beschreibungen ermöglichen schon sehr viel früher im Entwicklungsprozess - bereits wenn der Fahrzeughersteller die Funktionsmodelle dem Steuergerätelieferant übergibt - die Konsistenz zwischen unabhängig entwickelten Funktionsmodellen sicherzustellen. Es ist deshalb zu erwarten, dass die Funktionsintegration beim Lieferant sehr viel effizienter ablaufen wird. Abstimmungsrunden zwischen Fahrzeughersteller und Lieferant zur Softwarearchitektur sind durch das durch AUTOSAR vereinheitlichte Vokabular wesentlich produktiver. Darüber hinaus ermöglicht die standardisierte AUTOSAR-Architektur und das RTE-Middleware-Konzept losgelöst von konkreten Serienentwicklungsprojekten eine wiederverwendbare Bibliothek von Funktionsmodellen aufzubauen. Auf dieser Basis ergeben sich auch für die Zukunft eine Reihe von Herausforderungen.

Die Funktionsmodelle sind bisher noch sehr groß; in erster Näherung entspricht ein Funktionsmodell einer Softwarekomponente. Die Komplexität der Funktionen und die Forderung nach der Wiederverwendung der Funktionen in verschiedenen Baureihen erfordern einen Produktlinien-Ansatz von AUTOSAR-Funktionsmodellen, was eine Aufspaltung der Funktionen in zahlreiche Softwarekomponenten zur Folge haben wird. Hierbei ist allerdings noch nicht geklärt, wie die Aufteilung der Variabilität zwischen Softwarekomponente und dem Verhaltensmodell realisiert wird. Die Aufspaltung wird erst dann möglich sein, wenn Code-Generatoren Funktionscode und RTE-Code gemeinsam optimieren, sodass die Kommunikation über die RTE so effizient wie möglich erfolgen kann.

Im Zuge der Modularisierung müssen dann auch geeignete Simulationswerkzeuge verfügbar sein, die mittels einer sog. VFB[3]-Simulation, eine in mehrere Softwarekomponenten zerlegte Funktion effizient simulierbar macht. Darüber hinaus ergeben sich weitere Anwendungsszenarien der VFB-Simulation [Ri06]:

- Validierung einer Funktion mit ausgewählten Basissoftwarediensten [MHRK08]
- Validierung verschiedener interagierender Funktionen eines Steuergeräts
- Validierung über ein Bussystem vernetzter Funktionen, mit einer möglichst abstrakten Modellierung von Hardwareeigenschaften und Kommunikation

[3] VFB: Virtual Functional Bus

Für ein effizientes Arbeiten mit verschiedenen Werkzeugen in einer AUTOSAR-Werkzeugkette ist über die eigentliche Beschreibung der Softwarekomponenten auch ein grafisches Austauschformat notwendig, das Layoutinformationen z.B. Platzierung von Softwarekomponenten in einem Composition-Diagramm ermöglicht. Auf Grund der teilweise noch nicht vollständigen Definition des AUTOSAR-Standards ist eine enge Kooperation mit den Standardisierungsgremien und den Toolherstellern weiterhin notwendig, um so die Umstellung auf den AUTOSAR-Standard zukunftssicher gestalten zu können.

Die wahrscheinlich größte Herausforderung für die Zukunft bleibt das effiziente Handling von Funktionsvarianten. Hier spielen verschiedene Dimensionen von Varianten eine Rolle: Funktionsvarianten mit unterschiedlichem Funktionsumfang, Parametervarianten und Funktionsvarianten mit unterschiedlich benötigtem, bzw. bereitgestellten Umfang an Ein- und Ausgangs-Informationen (Ports). Ein effizientes und aufeinander abgestimmtes Variantenkonzept ist in den Modellierungswerkzeugen wie MATLAB/Simulink [We06], in der AUTOSAR-Beschreibung von Softwarekomponenten und in den AUTOSAR-Autorenwerkzeugen notwendig, um die Komplexität der immer weiter verbreiteten modellbasierten Entwicklung auch zukünftig beherrschbar zu gestalten.

Literaturverzeichnis

[AR] AUTOSAR GbR: Homepage http://www.autosar.org

[AR06a] AUTOSAR GbR: Specification of Interaction with Behavioral Models, Version 1.0.1, 2006

[AR06b] AUTOSAR GbR: AUTOSAR Simulink Styleguide, Version 1.0.1, 2006

[AR06c] AUTOSAR GbR: AUTOSAR Specification of Diagnostics Event Manager, Version 2.0.1, 2006

[DW07] C. Dziobek, F. Wohlgemuth: Einsatz von AUTOSAR bei der Modellierung von Komfort- und Innenraumfunktionen, dSpace User Konferenz, München, Juni 2007

[MHRK08] A. Michailidis, B. Hedenetz, Th. Ringler, S. Kowalewski: Virtuelle Integration von vernetzten Funktionen in frühen Entwicklungsphasen, 28. Tagung „Elektronik im Kraftfahrzeug“, Dresden, Tagungsband „Moderne Elektronik im Kraftfahrzeug“, ISBN-13: 978-3-8169-2819-5, April 2008

[We06] J. Weiland: Variantenkonfiguration von Modellbasierter Embedded Automotive Software, Model-Driven Development & Product Lines, Leipzig, 19. Oktober 2006

[Ri06] Th. Ringler: Modeling and Simulation of Distributed Automotive Systems with Simulink®, MathWorks IAC 2006, Stuttgart, Germany, May 2006

[RSM07] Th. Ringler, M. Simons, R. Beck: Reifegradsteigerung durch methodischen Architekturentwurf mit dem E/E-Konzeptwerkzeug, 13. Internationaler Kongress Elektronik im Kraftfahrzeug, Baden-Baden, VDI-Berichte Nr. 2000, VDI-Verlag GmbH Düsseldorf, 2007

[TPT] PikeTec GmbH: TPT – Time Partition Testing, www.piketec.com

Early simulation of usage behavior of multi-functional systems

Florian Hölzl, Sabine Rittmann

Fakultät für Informatik
Technische Universität München
Boltzmannstraße 3, 85748 Garching
{hoelzlf, rittmann}@in.tum.de

Abstract: In this paper we present a methodology for modeling the usage behavior of multi-functional systems at the early stage of the requirements engineering phase. To that end we hierarchically structure services (i.e. pieces of functionality which are visible to the user) and identify dependencies between these functionalities. We first model the service behavior modularly and then extend this service model with a coordination behavior model. This model is responsible for handling the dependencies between services of multi-functional systems. The methodology starts on the level of textually (informally) given functional requirements, formalizes the requirements step by step, and results in a formal (simulatable) model of the usage behavior. We illustrate our approach by means of a running example from the automotive domain, namely the power seat control system.

Keywords: Model-based requirements engineering, modeling of functionality, service-oriented approach, behavior simulation, coordination.

1 Introduction

Innovative functions - as can be found in the automotive domain - are one of the key potentials for competitive advantage. Medium to large scale systems offer a great variety of functionality to the user. In modern luxury cars we face up to 600 ”features”, ”services”, or ”functions” accessible by the user. Their development poses different challenges. In the requirements engineering phase, these services have to be specified *completely, consistently,* and *unambiguously (precisely)* as they serve as a basis for later phases of the development process and specify what the system-to-be shall look like.

This alone is not an easy task. When it comes to multi-functional systems which are characterized by a high degree of dependencies between functionalities it becomes even worse. Often the relationships between user-visible functions are not specified precisely. However, usually services behave differently when other services are around. This effect is known in literature under the term *feature interaction*.

Furthermore, there is a gap between the *informal* requirements engineering phase (dealing with textually formulated requirements) and the subsequent *formal* design phase (dealing with models). What is needed is a formal model of the usage behavior as early as

possible. This model has to be obtained step by step from the informally given requirements. A formal model precisely specifies the usage behavior and can be used to check for completeness and consistency of the system requirements. Optimally, this model can be simulated and thus serves as a basis for the validation of requirements.

In this paper we present a methodology for modeling the usage behavior of multi-functional systems. The basic building blocks of the approach are *services*. A service in our context is a piece of partial behavior which is visible at the system boundary. It basically relates system inputs to system outputs.

The basic idea of the iterative approach is the following: The starting point of the approach are textually given functional requirements. In the **informal phase** of the approach, the usage behavior is structured hierarchically by means of a service hierarchy. Dependencies between services are captured leading to a service graph. Furthermore, system inputs and outputs are identified. In the **formal phase** the single services are formally specified by means of an I/O automaton, respectively (*service models*). Based on the relationships, the services are combined step by step until the overall usage behavior is obtained. During the combination process the modular service specifications have to be adapted systematically to handle the effects of the other services (feature interaction). The so-called *coordination model* is responsible for handling the dependencies between services and takes care of disabling/interrupting services and conflict solving. The result is a formal model of the usage behavior. It models the pure functionality of the system-to-be and abstracts from issues concerning distribution and technical realization. Nevertheless, it is sufficient to describe and simulate the feature interaction precisely.

1.1 Contributions

The presented approach contributes a methodology for the seamless transition from informal requirements to a simulation model. We focus on the specification of basic system functionality in the form of services. We capture dependencies between these services and refine these relations into a concrete coordination behavior. The result is a model of the usage behavior that can be simulated. Thus the functional requirements can be validated early in the development to make sure that the right behavior is developed. We give a formal foundation of our concepts and provide adequate notational techniques to capture all relevant aspects.

1.2 Related work

The idea of structuring the system functionality hierarchically is not new. For example, FODA [KCH$^+$90] introduced so-called feature trees (including a hierarchical structure of the functionality). FODA also introduced a set of relationships between features. However, these features are used to represent commonalities and differences between systems and not to capture feature interaction. In [Bro05], the (restricted) sub service relationship is formalized. Although it is mentioned that horizontal relationships are very important they are not subject of that approach . Furthermore, no methodological support is given.

Usually, work on feature interaction is concerned with how to identify or avoid feature in-

teraction and not with the systematic adaption of services to handle the relevant influences. [CMKS03] provides a comparison of work about feature interaction.

As far as model-based requirements engineering is concerned, most work deals with extracting data related models (E/R or class diagrams) out of requirements specifications [Abb83, Kof05] or with the tracing of requirements to elements of the design model [GS07].

The coordination behavior introduced by our approach was inspired by concepts of synchronous dataflow languages [CP] and the explicit specification of coordination schemas as presented in [Lee03].

1.3 Outline

The paper is structured as follows. In Section (Section 2) we provide the given requirements of our running example - the power seat control system - in form of natural language descriptions. In Section 3 we explain our methodology step by step and apply it to the case study, respectively. Here, we exemplarily provide the necessary notations for the different models. Section 4 provides a formalization of the approach and the notations, and thus serves as the semantics of the formal models. Section 5 explains how to derive a simulation model given the semantical foundation. A summary, topics for future work and a contribution are given in Section 6.

2 Running example - power seat control system

In order to illustrate our approach we make use of a running example from the automotive domain, namely the power seat control system. In this section we give the functional requirements in textual form. In the following sections we will explain each step of our methodology by applying it to the power seat control system. Due to limitation of space we only give an excerpt of the overall functionality.

The driver can adjust his/her seat according to his/her requirements. The driver has the possibility to do the following adjustments: Adjustment of the angle of the back (forwards and backwards), adjustment of the distance between the seat and the steering wheel (increasing and decreasing), adjustment of the height of the rear area of the seat (up and down) adjustment of the height of the front area of the seat (up and down). In order to adjust the seat the respective motors are controlled.

The manual adjustment of the seat is only possible if the front door is open. The manual adjustment can be triggered by pressing the toggle switches (one for each adjustment: back, distance, rear area, front area) which are attached to the seat. The manual adjustment of the seat is carried out as long as the user request is active, respectively. For each adjustment (back, distance, rear area, and front area) only one direction of movement (e. g. up or down) can be carried out at once in case of the manual adjustment. At most two of the adjustments can be carried out simultaneously for efficient energy utilization.

In case the battery power is too low to perform the manual functionality the seat is not adjusted. If need be the currently active adjustment is aborted.

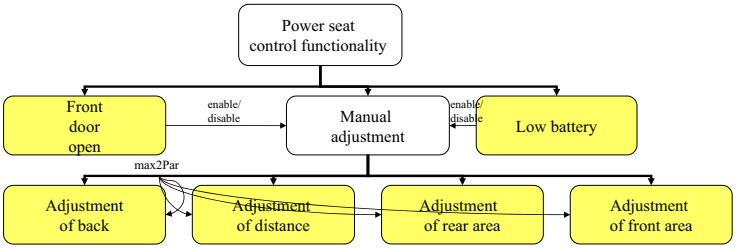


Figure 1: Service graph of the running example

3 Methodology

In this section we present our approach for modeling the usage behavior of multi-functional seat-adjustment system step by step. In this section we discuss the system using appropriate notations. These notations are backed by a formalization in Section 4.

3.1 Hierarchical structuring of the usage behavior and identification of service relationships

The first step of our methodology structures the - possibly large - usage behavior into manageable functionalities and identifies dependencies between them. To that end we first decompose the overall system functionality hierarchically into smaller pieces, namely *services*. A service in our context is a partial piece of behavior that can be observed at the system boundary. Partial means that the behavior is not described completely, i. e. the characteristic behavior is specified but it is possible to leave open how the system shall react in each situation. Note that our notion of service is scalable: Both small pieces and large pieces of behavior can be called a service.

The structuring of the usage behavior results in a tree, the so-called *service hierarchy*. The root of the tree is the overall system functionality. The leaves of the tree are the services that can be called, accessed, and observed by the user, called *atomic services*. The inner nodes, e. g. *composed services*, are a means to structure the system functionality.

There is no strict rule about when to introduce an inner node. This merely follows suitable rules for structuring systems and is even influenced by soft factors like quality criteria. For example, the structure of the requirements specification might provide some hints on the system structure. The structuring is also relevant when we later introduce dependencies between services. Intelligent structuring allows compact representation and enhances the readability and understandability of the model.

The adequate level of granularity concerning the service decomposition is an equal problem. Usually, the engineer's experience greatly influences the quality and suitability of a service decomposition. As a rule of thumb, a service should be decomposed if dependencies reference a sub-behavior and/or if the functionality is still too complex to be captured comnpletely. Subservices are connected to their parent services by *vertical service relationships*, forming of composition relation.

The identified services are not independent from each other, but influence each other. At this point we identify service relationships, i. e. dependencies that may exist between services. Therefore, we introduce *horizontal service relationships* into the service hierarchy. This results in a *service graph*, as is depicted in Figure 1 concerning the seat adjustment example.

The service MANUAL ADJUSTMENT is used to structure the functionality of the basic seat adjustment possibilities.

The services FRONT DOOR OPEN and LOW BATTERY can ENABLE or DISABLE the MANUAL ADJUSTMENT functionality, respectively. Enabling or disabling of a composed service results in the enabling or disabling of all of its subservices simultanously. Formally, we define relations $\mathbb{ENABLE}$, $\mathbb{DISABLE} \in \mathbb{S} \times \mathbb{S}$, where $\mathbb{S}$ is the set of atomic services.

According to the energy usage requirement, only two of the four adjustment services can be carried out in parallel. This is encoded in the horizontal service relationship MAX2PAR. The exact semantics of the horizontal relationships will be defined by its corresponding coordinator function. From the perspective of the requirements a textual description of this behavior should be provided, which might include the use of pre-defined abbreviations like MAX2PAR.

3.2 Identification of the syntactic interface

We explicitly model the syntactic interface of the system under development. To that end, we list the inputs entering the system (i. e. its stimuli) and the outputs leaving the system (i. e. its responses). For each input event and output event, we give a name, an abbreviation, and a description. Events are transmitted over one or more syntactic entities, usually called channels. Note, for reasons of simplicity we do not introduce data types here.

Tables 1 and 2 show an example of the syntactic interface specification.

3.3 Formal specification of atomic services

In this step, we formally model the behavior of the atomic services. Thereby, we specify the services as if they were independent from each other, i. e. the dependencies are not taken into consideration. We call this the *domain behavior* of a service. In the next section we will combine the formal specifications step by step to obtain a formal model of the overall system functionality including the feature interaction coordination.

We make use of System Structure Diagrams (SSDs, [HSE97]) and State Transition Diagrams (STDs, [HSE97]) to model the syntactic and semantic interface of each atomic service, respectively. SSDs describe the static system structure by means of communicating, possibly hierarchically decomposed entities (in our context: services) which are connected to each other via directed channels. STDs are basically input/output automata. The labels of transitions (leading from one state to another) are of the following exemplary form: INCH1?INMSG1; INCH2?INMSG2 / OUTCH1!OUTMSG1; OUTCH2!OUTMSG2 indicates that if INMSG1 is read on input channel INCH1 and INMSG2 is read on input channel INCH2, then the messages OUTMSG1 and OUTMSG2 are sent on the output channels OUTCH1 and OUTCH2, respectively. Note, that [HSE97] used a richer form of

Table 1: Specification of input events (examples only, data type ommited)

Name	Abbr.	Description
Adjust back backwards	back_backw	The driver wants to adjust the back backwards.
Adjust back forwards	back_forw	The driver wants to adjust the back forwards.

Table 2: Specification of output events (examples only, data type ommited)

Name	Abbr.	Description
Move back backwards	m_back_backw	The motor is controlled to move the back backwards.

transitions, which could be used instead of the reduced form presented here.

According to [Bro05], all services adhere to a global discrete notion of time divided into logical steps. During each step, one transition per service is fired. As far as underspecification is concerned, transitions are fired no matter what input is received on under specified input channels. Furthermore, the system leave the reaction open for underspecified output channels.

Services are *strongly causal* in the sense that the output reaction of a service on some input stimuli at time t is observed at time $t + 1$. Coordinators and conflict solvers, introduced later, could be viewed as *weakly causal*. They react instantaneously on some given input at some time interval t. We will use this distinction to the advantage that a service behavior is changed upon a coordination decision, e. g. the service might be disabled at time interval t and thus produce a different output at time $t + 1$ than in case it were allowed to proceed normally.

Figure 2 contains the formal specifications of the atomic services ADJUSTMENT OF BACK, FRONT DOOR OPEN, and LOW BATTERY. The other adjustment services can be specified analogously. As services are partial behaviors we model them with partial automata.

3.4 Combination of services on basis of their service dependencies

In this step, we combine the formal service specifications of the atomic services step by step until we obtain a model of the overall system functionality. Concerning our service hierarchy, we follow a bottom up approach and combine subservices to coarser services. During the combination process we have to handle the dependencies between the services. As services in general behave differently when other services are around, we have to take care of this fact. In particular, we will extend the partial domain behavior with reactions to the coordinator decisions. Coordinators are our means to give servive dependencies an operational meaning.

3.4.1 Coordinator functions

In our methodology, service relationships are realized by coordinators, or more precisely *coordinator functions*. Coordinator functions control the service execution in a way that

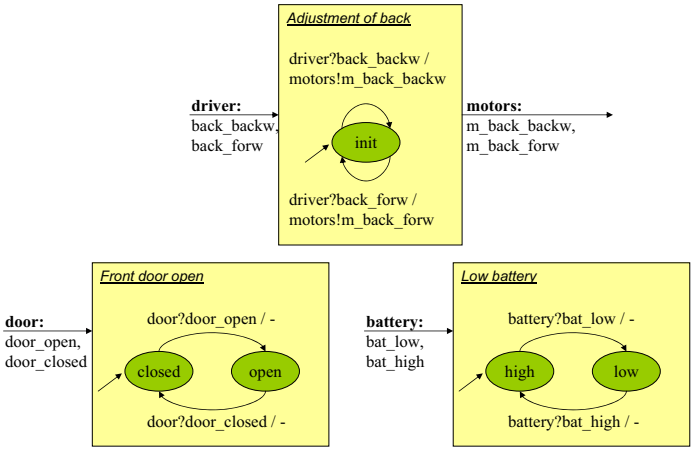


Figure 2: Formal specification of the atomic services ADJUSTMENT OF BACK, FRONT DOOR OPEN, and LOW BATTERY

the service relationships are obeyed. To that end, coordinator functions can ABORT services or make them PROCEED their execution. We restrict ourselves to these coordinator decisions, i. e. services can only be disabled and enabled. However, more complex coordination tasks are possible, i. e. interrupt and continue. We discuss such an extension on a formal basis in Section 4.4.

As mentioned before, coordinator functions are weakly causal, as their controlling of the services must be instantaneous. This is the main idea behind our model of computation and coordination. For example, a strongly causal coordinator is not able to stop adjustment manual services in case that more than two of them are active, because it observes this behavior at the same time the domain behavior effects are already in effect. Thus, the given energy consumption requirement is violated before the coordinator is able to act.

3.4.2 Coordinated behavior of services

In Section 3.3 we specified the behavior of services as if they were independent from each other. We called this behavior the domain behavior of an atomic service. During the combination of the services we have to adapt this behavior as the services have to obey the decision of the coordinator functions. The result is the *coordinated behavior* of a service. The introduction of coordinator functions requires to enlarge the service interface. First, the service has to provide the coordinator functions with its current status and possible enable/disable requests for other services. We call these events the *coordinator information events*. Second, the services have to handle the commands by the coordinator functions (see Section 3.4.2). These events are called the *coordination events*.

In order to implement this additional interface, we refine each transition of the domain behavior of a service. We obtain the following 4 segments for each transition which are

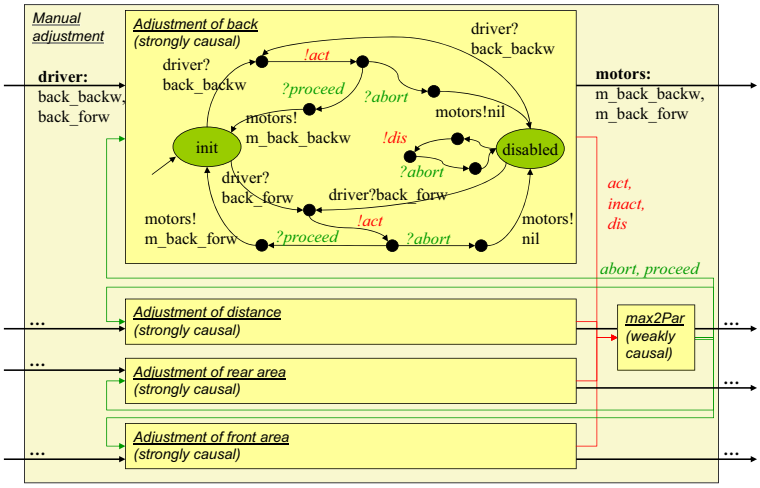


Figure 3: Combination of the adjustment services on basis of their dependencies

executed sequentially:

```
1. Read the system inputs.
2. Send the status that the service is going to be in and the
   intended enable or disable requests (coordinator
   information events).
3. Read the commands sent by the coordination function (coordination
   events).
4. Send the system outputs if not disabled or go to the disabled
   state.
```

Furthermore, we introduce a dedicated DISABLED state and define transitions leading back to the domain behavior.

Figure 3 shows the combination of the adjustment services. The MAX2PAR service relationship is realized by a weakly causal coordinator function which coordinates the service execution of the strongly causal adjustment services. The interface of the adjustment services are extended and the transitions refined according to the steps listed above.

3.4.3 Conflict solver

Usually, a service is influenced by not one but several service relationships. As a consequence, there might occur conflicts, e. g. if a service is concurrently disabled and enabled. To handle that case we also introduce *conflict solvers* which are - like the coordination functions - weakly causal behaviors deciding about the resolution of these conflicts. Although, the coordinated behavior of the overall system gets more complex, this method allows to find leaks in the requirements, e. g. feature interaction, the engineer was unaware of, is exposed and can be treated explicitly by means of coordination and conflict solving.

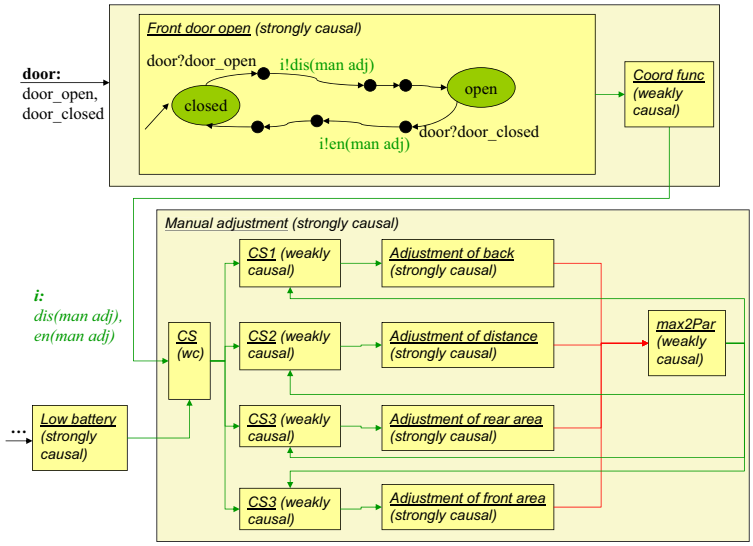


Figure 4: Combination of all services (some input and output channels omitted)

Figure 4 shows the combination of the MANUAL ADJUSTMENT and the services FRONT DOOR OPEN and LOW BATTERY. The conflict solver CS decides what to do in case the service FRONT DOOR OPEN wants to disable the MANUAL ADJUSTMENT while the service LOW BATTERY wants to enable it at the same time. The other conflict solvers (CS1 to CS4) handle conflicts between the MAX2PAR service relationships and the other relationships. We have omitted the concrete definition of each conflict solver, since many different behaviors are possible here. However, for simulation purposes, each conflict solver must be assigned a concrete function implementing the desired behavior.

4 Formalization

In this section we develop the formal foundation to combine services according to the dependencies captured in the service graph. Therefore, we first introduce the notion of coordinator functions. We use these functions to extend the service composition with the coordination functionality. The next step is to extend the basic service functionality, i. e. its domain behavior, with reactive behavior based on the coordinator events. Finally, we introduce conflict solvers formally. We then have obtained a complete specification of the coordinated behavior of a set of services wrt. to the service graph.

4.1 Combination of dependent services using coordinator functions

Definition 4.1 (Coordinator Information Events)**.** *Let $s \in \mathbb{S}$ be a service,* $\mathbb{ENABLE}$ *and* $\mathbb{DISABLE}$ *be the relationships defined by service graph, and $CI_s := (A \times E \times D)$ be the*

set of coordination information events wrt. s. This set is defined as follows:

$A := \{act, inact, dis\}$ *(service state)*

$E := \{en(e) \mid (s, e) \in \mathbb{ENABLE}\}$ *(enable requests)*

$D := \{dis(d) \mid (s, d) \in \mathbb{DISABLE}\}$ *(disable requests)*

□

The set of coordinator information events defines all possible events that can be sent from a service to a coordinator. With each such event, a service indicates its current activity, e.g. *active*, *inactive*, or *disabled*. Furthermore, a service can request the enabling and/or disabling of zero or more other services. The service graph and the dependency relations imply a restriction on the requests given some service s. This service can only enable / disable those services that are related to it by the corresponding dependency relation.

The introduction of the *disabled* notification allows us to design the coordinators stateless, i. e. the coordinator does not need to remember which services were disabled, because each service will annouce this information everytime.

Each service can disable itself by going into the *inactive* state, i. e. this transition is specified by the service itself, while the disabling by going into the *disabled* state is an enforced transition originating from some coordinator decision.

Next to the information sent from services to coordinators, we also need the information sent back as a result of some coordination decision.

Definition 4.2 (Service Coordination Events)**.** *Let $s \in \mathbb{S}$ be a service. The set of coordination events of s, denoted by CE_s is defined as:*

$CE_s := \{proceed_s, abort_s\}$ □

The set of service coordination events defines a set of two unique events for a service. According to this event the service may either proceed or abort its current execution step. Any service subject to coordination receives one of these events from its coordinator during each computation step.

Definition 4.3 (Coordinator Function)**.** *Let $\{s_1, \ldots, s_k\} \in \mathbb{S}$ be a set of coordinated services. A coordinator C for this set is a function*

$C : CI_{s_1} \times \ldots \times CI_{s_k} \rightarrow CE_{s_1} \times \ldots \times CE_{s_k}$ □

A coordinator is used to coordinate services, i. e. it receives the coordinator information events of all connected services, makes a coordination decision, and sends service coordination events to each service according to its decision. The concrete coordination function defines the way a coordination decision is reached. For example, the MAX2PAR coordinator could decide to give priority to two adjustment services and disable the remaining two

services, if all of them announce an active state during some computation step. Note, that the selection of a concrete coordinator function is an important step in the methodology. That is why we propose to make the coordination decision function explicit by introducing the coordinators.

The formalization of coordinator functions presented so far only allows for coordinators that can abort a service. It is not possible to interrupt a service for some time and later continue its execution. We will elaborate on this extension at the end of this section.

4.2 Conflict solvers

Definition 4.4 (Conflict Solver)**.** *Let $s \in \mathbb{S}$ be a service. A conflict solver CS_s is a mapping*

$$CS : CI_s \times CI_s \rightarrow CI_s$$ □

Formally, a conflict solver solves a conflict between two coordination events. For more than two events we either use a set of sequentially aligned conflict solvers or we extend the given definition to arbitrary many input values, which is straightforward.

4.3 Integration of services and coordinators

So far we have introduced the communication between coordinated services and coordinators, and the function specifying coordinator decisions. Now, the coordinator decision must be obeyed by the coordinated services. Therefore, the behavior of detached services, i. e. its *domain behavior*, must be extended, and thereby altered, to take into account the coordinator decision. We call the resulting behavior the *coordinated behavior* of the service. Note, that the observed domain behavior may differ from the coordinated behavior. This is inherent to the problem of feature interaction. We introduce a formal operator that extracts the domain behavior from a given coordinated behavior to clarify the relation of the two behaviors. The domain behavior of some coordinated service could be observed only if it is always allowed to proceed by its coordinator(s).

Definition 4.5 (Domain Behavior)**.** *Let s be a service, $\mathcal{I}_s$ its set of input message tuples, and $\mathcal{O}_s$ its set of output message tuples.*

The domain behavior DB_s of s is a tuple (Σ_s, δ_s, σ_s^0) defined as follows:

- *Σ_s is an arbitrary, non-empty set of service states,*
- *$\delta_s : \Sigma_s \times \mathcal{I}_s \rightarrow \mathcal{P}(\Sigma_s \times \mathcal{O}_s)$ is the service state transition function,*
- *$\sigma_s^0 \in \Sigma_s$ the initial state of the service.*

$\mathbb{DB}_s$ denotes the set of all domain behaviors of s. □

A domain behavior specifies an input/output state machine that consists of its set of service states Σ_s, an initial service state σ_s^0, and the state transition function δ_s, which defines for

each state and input valuation the set of possible follow-up states and the corresponding output values. The power set allows for non-deterministic state machines.

Definition 4.6 (Coordination Behavior)**.** *Let s be a service, $\mathcal{I}_s$ its set of input message tuples, and $\mathcal{O}_s$ its set of output message tuples. The coordinated behavior CB_s of the service s is a tuple (Ψ_s, γ_s, ψ_s^0) defined as follows:*

$\Psi_s := \Sigma_s \uplus \{\mathrm{DIS}\}$ *with Σ_s being an arbitrary, non-empty set of service states, and* DIS *denotes the disabled coordination state,*

$\gamma_s : \Psi_s \times \mathcal{I}_s \times CE_s \to \mathcal{P}(\Psi_s \times \mathcal{O}_s)$,

$\psi_s^0 \in \Sigma_s$ *the initial state of the service.*

The state transition function γ_s has the following properties:

$$\begin{aligned}
\forall\, \sigma \in \Sigma_s, \overrightarrow{i} \in \mathcal{I}_s \quad &: \\
\forall\, (\sigma', \overrightarrow{o}) \in \gamma_s(\sigma, \overrightarrow{i}, abort_s) \quad &: \quad \sigma' = \mathrm{DIS} \qquad \wedge \qquad (1)\\
\forall\, (\sigma', \overrightarrow{o}) \in \gamma_s(\sigma, \overrightarrow{i}, proceed_s) \quad &: \quad \sigma' \in \Sigma_s \qquad \wedge \qquad (2)\\
\gamma_s(\mathrm{DIS}, \overrightarrow{i}, proceed_s) \quad &= \quad \gamma_s(\psi_s^0, \overrightarrow{i}, proceed_s) \qquad (3)
\end{aligned}$$

$\mathbb{CB}_s$ *denotes the set of all coordinated behaviors of s.* □

A coordinated behavior describes the behavior of a service when it is connected to some coordinator and thus receives coordination events from it. In comparison to the domain behavior, the state space of service is extended by a unique disabled state DIS. We call the additional state a coordination states, since it can only be reached through the coordination event $abort_s$.

The properties of the state transition function γ_s enforce the following behavior of the service:

(1) If the service is *aborted*, it goes into the *disabled* state.

(2) If the service may *proceed*, it must reach a non-coordination, i. e. service, state.

(3) If the service is *disabled* and may *proceed*, it behaves as if it started from its initial state.

The modular specification of the coordinated behavior of some service includes a sort of non-determinism wrt. the inputs received from some coordinator. Thus, the domain behavior of the service is a refinement of its coordinated behavior, since every possible observation originating from the domain behavior is also a valid observation of the coordinated behavior. Assuming that the coordinated service is always allowed to proceed, it behaves exactly the same way as its domain behavior does. The domain behavior extraction operator formalizes this relationship between the two behavior specification of the service.

Definition 4.7 (Domain Behavior Extraction Operator)**.** *Let s be a service, $\mathbb{CB}_s$ be all possible coordinated behaviors of s, and $\mathbb{DB}_s$ be all possible domain behaviors of s. The domain behaviorn extraction operator $\circledast$ is a mapping from $\mathbb{CB}_s$ to $\mathbb{DB}_s$.*
Given $c = (\Psi_s, \gamma_s, \psi_s^0) \in \mathbb{CB}_s$, $\circledast(c) = (\Sigma_s, \delta_s, \sigma_s^0) \in \mathbb{DB}_s$ is obtained as follows:

$$\Sigma_s := \Psi_s \setminus \{\mathrm{DIS}\}$$
$$(\sigma', \overrightarrow{o}) \in \delta_s(\sigma, \overrightarrow{i}) \Leftrightarrow (\sigma', \overrightarrow{o}) \in \gamma_s(\sigma, \overrightarrow{i}, proceed_s)$$
$$\sigma_s^0 := \psi_s^0$$

□

Definition 4.8 (Service Coordinator Information Function)**.** *Let s be a service, CB_s its coordinated behavior, and $\mathcal{I}_s$ its set of input message tuples. The coordinator information function of s, denoted as α_s, is a mapping defined as follows:*

$$\alpha_s : \Psi_s \times \mathcal{I}_s \rightarrow CI_s$$

The coordinator information function has the following properties:

$$\begin{aligned} \forall\ \overrightarrow{i} \in \mathcal{I} \quad &: \\ \alpha_s(DIS, \overrightarrow{i}) \quad &= \quad (dis, E, D) \wedge \qquad (4) \\ \forall\ \sigma \in \Sigma_s : alpha_s(\sigma, \overrightarrow{i}) \quad &\neq \quad (dis, E, D) \qquad (5) \\ & \quad\ for\ some\ E,\ D. \end{aligned}$$

□

The coordinator information function determines the information sent to the coordinator by a service s. This piece of information depends on the current state and the current input of the service. The properties of the coordination information function enforce the following constraints:

(4) If the service is disabled, it reports *dis*.

(5) If the service is not disabled, it does not report *dis*.

Note, that the enable and disable requests are not constraint by default. These requests have to be specified during the engineering of the coordination model, i. e. during the definition of the concrete coordinator information function of each service.

4.4 Interrupt / continue coordination extension

For reasons of simplicity, we have constraint coordinators to only support enablement and disablement of services. We now sketch briefly how we could extend coordinators and coordinated services to support temporal interruption and later continuation.

First, we extend the first dimension of CI_s with an additional activity information int issued by a service s if it is currently interrupted. Furthermore, each service can request the interruption and continuation of some other service. Thus, CI_s is extended by two dimension: the *interrupt requests*, $\mathcal{P}(\{int(i) \mid (s,i) \in \mathbb{INTERRUPT}\})$, and the *continue requests* ,$\mathcal{P}(\{cont(c) \mid (s,c) \in \mathbb{CONTINUE}\})$. These relationships $\mathbb{INTERRUPT}$ and $\mathbb{CONTINUE}$ are again defined by the service graph. Accordingly, we introduce the coordination event $suspend_s$ in CE_s. Thereby, coordinator functions and conflict solvers are extended similarly.

The domain behavior remains unchanged, but the coordinated behavior is changed substantially. In order to continue after having been interrupted, each service must remember its previous state (in this context called a history state). Therefore, we introduce, for each service state $\sigma_s \in \Sigma_s$ and input valuation $\overrightarrow{i^\circ} \in \mathcal{I}_s$, an additional coordination state $\mathrm{INT}(\sigma_s, \overrightarrow{i^\circ})$. Furthermore, we have to extend the properties of the transition function γ_s, as follows:

$$\forall\, \sigma_s \in \Sigma_s, \overrightarrow{i} \in \mathcal{I}_s \quad : \tag{6}$$

$$\forall\, (\sigma_s', \overrightarrow{o}) \in \gamma_s(\sigma_s, \overrightarrow{i}, suspend_s) \quad : \quad \sigma_s' = \mathrm{INT}(\sigma_s, \overrightarrow{i}) \quad \wedge \tag{7}$$

$$\forall\, (\sigma_s', \overrightarrow{o}) \in \gamma_s(\mathrm{INT}(\sigma_s, \overrightarrow{i}), abort_s) \quad : \quad \sigma_s' = \mathrm{DIS} \quad \wedge \tag{8}$$

$$\gamma_s(\mathrm{INT}(\sigma_s, \overrightarrow{i^\circ}), \overrightarrow{i}, proceed_s) \quad = \quad \gamma_s(\sigma_s, \overrightarrow{i^\circ}, proceed_s)$$

Property 6 states that a suspended service enters the corresponding interrupted state, property 7 ensures that an interrupted service can be disabled, and property 8 describes the continue transition, executed once the service may proceed. The formalization leaves open the reaction of the service to some input while it is disabled. There are different solutions possible, e. g. the service could ignore any input or it might issue an error message in response to some non-empty input.

Finally, we must extend the coordination information function α_s. In particular we add, for each introduced interrupted state, a property similar to property 4 and property 5. Again, this frees the coordinator function from remembering which service is interrupted, since this information is provided by the particular service itself.

5 Simulation

In this section we briefly outline the necessary steps to derive a simulation model from the specification model and execute it with a suitable computation scheme.

5.1 Building a simulation model

With the formal foundation given in the previous section, we can now construct a simulation model that allows us to validate the coordinated behavior of the services.

For each service s, we must construct a single data entity holding the current state ψ_s, input buffers for input channels (including the coordination event channel), output buffers for

output channels holding the computation results, a computation unit realizing the service coordinator information function α_s, and a computation unit realizing the coordination behavior CB_s.

Additionally, the simulation environment provides input buffers for environment input channels, output buffers for environment output channels, a computation unit for each coordinator function $c \in C$, a computation unit for each conflict solver $cs \in CS$, and a simulation driver executing the simulation algorithm given below during each simulation step.

5.2 Executing the simulation model

The simulation model can be executed by applying the following computation scheme, informally provided below, for each step of the simulation. The environmental input can be given by the user or some unit test, and the output could be shown to the user or automatically checked, respectively.

1. Propagate the output buffers of the services (which contain the computation result of the previous step or the initial value.
2. Propagate the inputs from the simulation environment to the services and their coordinator information function.
3. Compute service coordinator information functions for each service.
4. Afterwards compute the coordinator functions.
5. With the coordinator results, compute each conflict solver and forward the final decision result to the respective service.
6. Compute the coordination behavior of each service updating output buffers and internal state.

6 Summary, conclusion, and future work

In this paper we presented a methodology for modeling the usage behavior of multi-functional systems. The textually given functional requirements serve as starting point for the approach. First, the usage behavior is structured into hierarchically decomposed services (*service hierarchy*). Then the possible inputs and outputs of the system are specified (*syntactic service interfaces*). Furthermore, the dependencies between services are captured (*service graph*). Single services are formally specified and combined according to their dependencies. During the combination process dedicated *coordinators* which are responsible for handling the dependencies are introduced. Possible conflicts between multiple coordinators are solved by introducing explicit *conflict solvers*. Finally, the modular service specifications are adapted systematically to react upon coordinator decisions: the service *domain* behavior is transformed into its *coordinated* behavior.

The work presented here is currently elaborated in a PhD thesis [Rit08]. Due to limitation of space not all aspects are covered in this paper. For example, there can exist data dependencies between services. Furthermore, more complex coordination schemes like interrupt/continue introduce new phenomena like dead-locking or starvation, if not used properly.

Concentrating on service specification and service coordination, our approach opens in-

teresting topics for future research: The result of our methodology is a formal model of the usage behavior of multi-functional systems. It models the pure functionality of a system and abstracts from distribution and the technical realization. It has to be investigated how to proceed to a logical distributed component architecture and further towards an implementation. Usually, part of the coordination tasks is carried out by a suitable runtime environment or operating system that offers system primitives for the given coordination tasks.

Furthermore, product line concepts and non-functional requirements could be incorporated into our approach.

References

[Abb83] Russell J. Abbott. Program design by informal English descriptions. *Commun. ACM*, 26(11):882–894, 1983.

[Bro05] Manfred Broy. Service-Oriented Systems Engineering: Specification and Design of Services and Layered Architectures - The JANUS Approach. *Engineering Theories of Software Intensive Systems*, Seiten 47–81, 2005. Springer Verlag.

[CMKS03] Muffy Calder, E. Magill, M. Kolberg und Reiff-Marganiec S. Feature Interaction: A Critical Review and Considered Forecast. *Computer Networks*, 41/1:115–141, January 2003.

[CP] Paul Caspi und Marc Pouzet. Synchronous Kahn Networks. ACM Sigplan International Conference on Functional Programming, Philadelphia, Seiten 226–238.

[GS07] Eva Geisberger und Bernhard Schätz. Modellbasierte Anforderungsanalyse mit AutoRAID. *GI - Informatik Forschung und Entwicklung*, 2007.

[HSE97] Franz Huber, Bernhard Schätz und Geralf Einert. Consistent graphical specification of distributed systems. Jgg. 1313, Seiten 122–141, 1997.

[KCH+90] K. Kang, S. Cohen, J. Hess, W. Novak und A. Peterson. Feature-Oriented Domain Analysis (FODA) - Feasibility Study. Technical Report CMU/SEI-90-TR-021, Carnegie Mellon Software Engineering Institute, 1990.

[Kof05] Leonid Kof. *Text Analysis for Requirements Engineering*. Dissertation, Technische Universität München, 2005.

[Lee03] Edward A. Lee. Overview of the Ptolemy project. Bericht UCB/ERL M03/25, EECS Department, University of California, Berkeley, July 2003.

[Rit08] Sabine Rittmann. *A methodology for modeling usage behavior of multi-functional systems*. Dissertation, Technische Universität München, 2008. To appear.

Modeling Data Requirements for a Secure Data Management in Automotive Systems

Sandro Schulze, Tobias Hoppe, Stefan Kiltz, and Jana Dittmann

School of Computer Science, University of Magdeburg, Germany,
{sanschul, t.hoppe, kiltz, jana.dittmann}@iti.cs.uni-magdeburg.de

Abstract. IT security and data management are two emerging aspects in automotive systems. In this paper we propose a model which supports the integration of these aspects, already starting in the early stages of development. This systematisation of different kinds of data processed within automobiles, including their specific requirements (with respect to safety, security, comfort etc.), allows for a wide range of applications like implementation guidelines and verification. Such holistic approaches are essential for future automotive data management, especially to cope with the increasing complexity and IT security requirements in data management.

Key words: Automotive, IT security, data management, requirements engineering

1 Introduction

Automotive systems are an evolving area with a rapidly increasing amount of software [1], caused by more and more functionality to be implemented. At the same time the amount of data to be managed by the system is increasing [2]. Furthermore, different kinds of attackers have a variety of motivations in attacking automotive IT [3] in different ways [4]. Thus, more attention will have to be paid to both, effective concepts for IT security and efficient data management during the development of Electronic Control Units (ECUs) [5].

Combining these emerging aspects, IT security and data management, could be useful in several ways, including reliability or confidentiality of data. Moreover, it is indispensable to take both into account if we consider an automotive system as a network providing a number of features to interact and therefore, to interfere with [3].

In this paper, we present an approach, which integrates these new automotive aspects of IT security and data management from the beginning into the development process. Therefore we propose a model, which enables modeling data requirements integrated with aspects of security, safety, data management and comfort. The intention is not to develop yet another model-driven code generator for the automotive domain. This model rather adresses the preliminary considerations of the design or implementation phase, e.g., requirements

analysis. As a result, design recommendations for secure data management may be derived from our model. Subsequently, these recommendations can be used for implementing the respective software, using already existing and upcoming standards and architectures, e.g., AUTOSAR (*www.autosar.org*).

2 Integration of security and data management

Our model, presented in the following, is placed within the requirements analysis phase of the software development process. Thus, specifications for the several data (and ECU's), given by an domain expert (e.g. the car constructor), can be taken directly to model the respective requirements.

2.1 Fundamentals

Before introducing our model for data requirements, we present a formalisation of the automotive system, representing the environment of the considered data.

As a networked IT-system, we formalise a modern car into the classes sensors (s_i), electronic control units (c_i) and actuators (a_i). Information exchange between cars (car-2-car) or between the car and the infrastructure (car-2-infrastructure) is deliberately excluded within the confines of this paper. The underlying network of our formalisation is depicted in Figure 1.

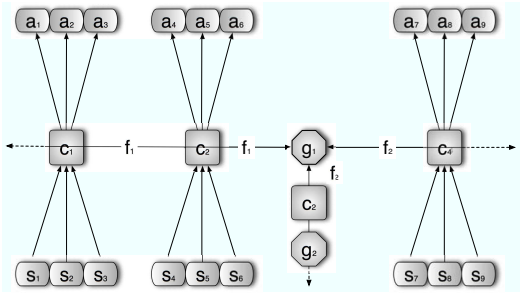


Fig. 1. Formalisation of the automotive IT-network

Each of the corresponding ECUs ($c_i, i \in \mathbb{N}$) gets (analogue or discrete) environmental data from the sensors ($s_i, i \in \mathbb{N}$). With this information the ECU computes values to control the system using actuators ($a_i, i \in \mathbb{N}$). The ECUs exchange information amongst each other by sending data telegrammes via field bus systems ($f_i, i \in \mathbb{N}$). Examples for common automotive field bus architectures are CAN, FlexRay, MoST and LIN.
A special variant of an ECU is the gateway ($g_i, i \in \mathbb{N}, G \subset C$), which connects different subbus systems (e.g. powertrain, body electronics, driver information). This includes the decision whether forwarding messages into a different subbus or not and translating data telegrammes between different bus architectures.

With the help of the control unit set C, we define a topology $T = \{C, Graph\}$, representing the automotive system as a whole. The graph in this context describes the structure of the system (cf. Figure 1), connecting all control units in a specified way.

Based on this topology, we define a formalisation for automotive data, which has to be extended in future. One important aspect is to integrate a formal representation of the desired data flow. Here, existing data flow models, e.g., Clark-Wilson [6], could be employed. For such purposes, the integration of existing formal models (e.g., from IT security) is not yet addressed by this initial draft, which is explained shortly in the following.

$$D = \{d_1, ..., d_m\}, d_j = \{Identifier, C_j, Req_j, P_j\}, d_j \in D,$$
$$C_j = \{c_i \subset C | x_k \in X\}, X = \{R, RW, W\}$$

Within our formalisation, D encompasses all data within the automotive system. Furthermore, C_j contains all ECUs accessing a single data d_j and the way they are allowed to do this, i.e. which rights an ECU has on the considered data. The possible rights, namely *read (R)*, *write (W)* and *read/write (RW)*, are represented by the set X. Finally, the sets Req_j and P_j represent the data requirements and attributes respectively, which have to be described in more detail. Thus, we introduce a model, which helps us to determine requirements for a single data, with respect to security and data management issues.

2.2 Modeling data requirements

Besides the formalisation, the model in Figure 2 is used for the modeling of data requirements. As can be seen, requirements are devided into different aspects, marked as r_1, r_2, r_3 and r_4. Additionally, some *implicit* requirements are caused by data attributes, e.g., changeability, which are made explicit in the model as p_1. Furthermore, certain requirements depend on other requirements, which is denoted for *some* dependencies by *implies/extends*. Here, security and data management are integrated in our model. The specified requirements and attributes for a single data are represented by the sets Req_j and P_j respectively, which become part of the formal data definition (s. Section 2.1).

Considering all data (and its requirements) accumulated at one single ECU, a design guideline of the data management system for this ECU may be derived in two ways. First, requirements can be mapped directly to functional features, e.g. traceability can be mapped to a *logging* feature. Second, several requirements can be mapped to functional features using boolean operators. For instance, the requirements *continuous data flow*, *real-time* and *reliability* concatenated by the AND operator, possibly lead to a *main memory storage* feature, and thus, satisfy the integration aspect yet again. While the resulting features can be seen primarily as recommendations for the design phase so far, we plan to extend our model in that the features support the implementation phase in a straightforward way as well.

3 Applying the model to automotive data

One example for the application of the proposed model is the odometer reading. This data is highly sensitive to malicious alteration (i.e. a violation of IT security

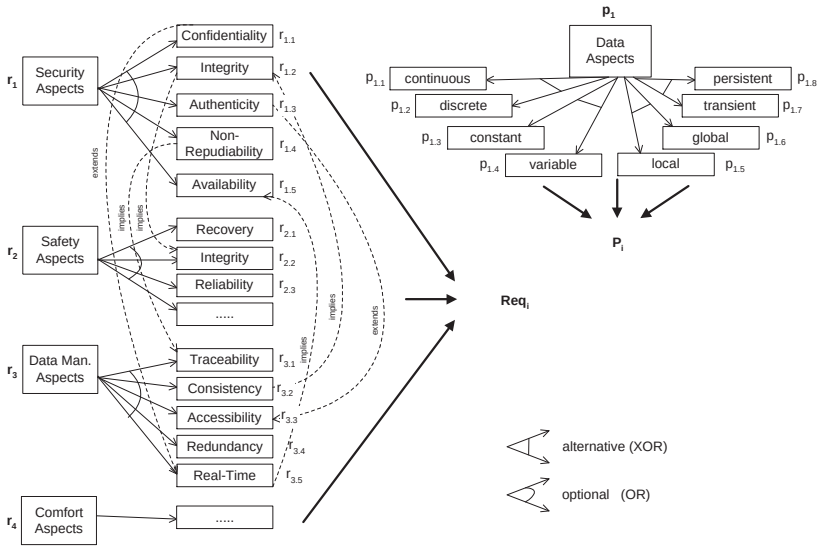


Fig. 2. A Model for data requirements in automotive systems

aspects integrity, authenticity, non-repudiability, see also [4]). A successful attack on the odometer reading could result in a higher resale value of the car and thus, the owner or a dealer willing to sell the car could be interested in manipulating the counter value. As a result, an alteration of the odometer reading can have implications on both the *comfort* and the *safety* of the vehicle. For instance, the onboard computer computes the expected cruising range based on the odometer reading. Since there is a fuel gauge, alarming the driver of a low fuel reading, the consequences of such a miscalculation will not have safety consequences but will be a nuisance. However, a successful malicious alteration (i.e. violation of IT *security*) can also have consequences on the *safety* insofar as important maintenance measures with a specific service interval are not carried out, and thus, may be affect the powertrain-, brakes- and steering systems.

Applying our proposed model to the scenario described above (with respect to the topology mentioned in Section 2.1), leads to the exemplary set of ECU's $C = \{c_1, c_2, c_3, c_4, g_1\}$. C is used with the following exemplary configuration:

- c_1 dashboard
- c_2 onboard computer
- c_3 central lock
- c_4 engine control unit
- s_7 velocity sensor
- g_1 gateway between powertrain-, body electronics- and instrumentation-bus

With the given configuration of the system, the *odometer count* data, chosen as an example, can be modeled as follows:

$d_1 = \{odometer\_count, C_1, Req_1, P_1\}$
$C_1 = \{c_4|\{W\}, c_1|\{R\}, c_2|\{R\}\}$
$Req_1 = \{r_{1.2}, r_{1.3}, r_{1.4}, r_{1.5}, r_{2.1}, r_{3.2}, r_{3.4}, r_{4.1}\}$
$P_1 = \{p_{1.2}, p_{1.4}, p_{1.6}, p_{1.8}\}$

The whole data is described by the formula d_1. As an unique identifier *odometer_count* is chosen. While three ECUs are affected by the data (C_1), the engine control unit c_4 is the only device which has write access to this data. Other devices like the dashboard (c_1) and the on-board computer (c_2) will only have read access, which will be enforced by the DBMS components of the final system. Important requirements for this piece of data are represented in the set Req_1, e.g., the security target integrity ($r_{1.2}$) is a central aspect. While processing the proposed model, the algorithm will demand an appropriate security algorithm to protect the integrity (e.g. digital signatures), ideally captured by the DMS (e.g. within a data validation module). Since the security target integrity ($r_{1.2}$) already implies the related safety aspect integrity ($r_{2.2}$, e.g. implemented by CRC checksums), the latter does not have to be specified in the formula by the engineer anymore. In other words, integrity from the perspective of safety can be seen as a subset of the integrity from the IT security perspective. Finally, P_1 lists important characteristics of this data, e.g. that it is a *global* ($p_{1.6}$) *variable* ($p_{1.4}$), which has to be stored *persistently* ($p_{1.8}$).

Taking the result of the example model above, we can now derive some design recommendations in form of (abstract) functional features for the data management of *every* participating ECU. For instance, considering the dashboard ($c1$), which only has read access to the *odometer_count* data. Possible functional features for this ECU are listed in Table 1 below.

Requirements/Attributes	**Design recommendation**
Authenticity, Accessbility	Access management feature with sophisitcated authenticity validation, e.g. cryptographical algorithm
Integrity, (Redundancy)	Update feature using digital signatures
Consistency, Redundancy	Concurrency control feature
Non-Repudiability (implies Traceability)	Logging feature

Table 1. Exemplary relation of requirements and functional modules

It has to be mentioned, that the recommended features in Table 1 do not claim to be complete, regarding the underlying example. Moreover, we have chosen a quite simple example with only one piece of data considered here. Usually, a control unit has to handle a plenty of data, leading to an increasing number of requirements (and their combinations). Hence, the amount of possible functional features may be increase as well.

4 Conclusions and Future Work

In this paper, we have proposed the integration of IT security and data management, an issue, which was mostly neglected in its definition and detailed analysis in the automotive domain so far. Therefore, we presented a model where data requirements can be specified regarding different aspects and thus, integrate

IT-security and data management with 'classic' disciplines like safety and comfort. In addition, with the suggested model it is possible to derive functional features for a secure data management, even though this happens in a quite abstract way.

As future work, we want to combine our model with a feature diagram, representing the superset of all functionality relevant to automotive data management. This should help to make the relation between requirements and functional features more obvious. Additionally, we intend an implementation of this model, to proof its applicability.

Another future task is to evaluate the impact of single IT-components on the entire system with respect to both, function and structure in terms of IT-security and safety, in more detail.

Furthermore we intend to make appropriate IT security mechanisms for several requirements more explicit and take the data flow and its incorporated ECU's into account by using already existing models.

Finally, we want to examine, where to place this model within the AUTOSAR Methodology, as far as possible.

5 Acknowledgements

The work described in this paper has been supported in part by the European Commission through the EFRE Programme "COMO" under Contract No. C(2007)5254. The information in this document is provided as is, and no guarantee or warranty is given or implied that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

References

1. Pretschner, A., Broy, M., Krüger, I., Stauner, T.: Software Engineering for Automotive Systems: A Roadmap. In: Proceedings of Future of Software Engineering (FOSE) at the ICSE. (2007) 55–71
2. Casparsson, L., Rajnak, A., Tindell, K., Malmberg, P.: Volcano - a Revolution in On-Board Communications. Technical report, Volvo Technologies (1998)
3. Hoppe, T., Kiltz, S., Lang, A., Dittmann, J.: Exemplary Automotive Attack Scenarios: Trojan horses for Electronic Throttle Control System (ETC) and replay attacks on the power window system. In: Proceedings of the 23. VDI/VW Gemeinschaftstagung Automotive Security. (2007) 165–183
4. Lang, A., Dittmann, J., Kiltz, S., Hoppe, T.: Future Perspectives: The Car and its IP-Address - A Potential Safety and Security Risk Assessment. In: Proceedings of the 26th International Conference on Computer Safety, Reliability and Security (SAFECOMP). (2007) 40–53
5. Nyström, D., Tesanovic, A., Norström, C., Hansson, J.: COMET: A Component-Based Real-Time Database for Automotive Systems. In: Proceedings of the Workshop on Software Engineering for Automotive Systems at the ICSE. (2004)
6. Bishop, M.: Introduction to Computer Security. Number ISBN: 0-321-24744-2. Addison-Wesley (2005)

Model-Based Development of an Adaptive Vehicle Stability Control System*

Rasmus Adler[1], Ina Schaefer[2] and Tobias Schuele[3]

[1] Fraunhofer Institute for Experimental Software Engineering (IESE)
Fraunhofer-Platz 1, 67663 Kaiserslautern, Germany
rasmus.adler@iese.fraunhofer.de

[2] Software Technology Group, Dept. of Computer Science, University of Kaiserslautern
P.O. Box 3049, 67653 Kaiserslautern, Germany
ina.schaefer@informatik.uni-kl.de

[3] Embedded Systems Group, Dept. of Computer Science, University of Kaiserslautern
P.O. Box 3049, 67653 Kaiserslautern, Germany
tobias.schuele@informatik.uni-kl.de

Abstract: Safety and availability are major requirements for embedded systems in modern vehicles. However, in the automotive domain, neither conventional shutdown mechanisms nor full-fledged redundancy are appropriate means for handling faults such as failures of sensors and actuators. A suitable means for achieving the high requirements on safety and availability at low costs is graceful degradation by adapting the functionality of a system to the current driving situation and the available resources. However, adaptation significantly complicates the development of embedded systems. In this paper, we present an approach to the model-based design of adaptive embedded systems that allows coping with the increased complexity posed by adaptation. Furthermore, we show how the obtained models can be formally verified and demonstrate the feasibility of our approach by applying it to a vehicle stability control system.

Keywords: Adaptive systems, model-based development, formal verification

1 Introduction

Many advanced vehicle functions, such as antilock braking and vehicle stability control, influence elementary functionalities like braking and steering. Consequently, safety and availability are major requirements for such software-based functions. To fulfill these requirements, compensation for typical faults and deficiencies, e.g., inaccurate sensor values due to special driving situations, is indispensable. Conventional shutdown mechanisms are not appropriate for that purpose, as one software component typically contributes to many fail-operational vehicle functions. Full-fledged redundancy based on redundant hardware

*This work has been supported by the Rheinland-Pfalz Cluster of Excellence 'Dependable Adaptive Systems and Mathematical Modelling' (DASMOD).

is usually too expensive for automotive systems as mass products. Moreover, full-fledged redundancy might not be sufficient for compensating for conceptual faults. For instance, a homogeneous redundant sensor would deliver the same inaccurate sensor values and could not remedy any fundamental deficiencies.

To solve these problems, dynamic adaptation of system components and graceful degradation of the system's functionality have become the state of the art in automotive systems for reacting to dynamically changing driving situations as well as to failures. As many functions in an automobile are based on approximated physical models assuming certain driving situations, it is necessary to adapt to the most reasonable calculation variant for a sensor value depending on the current situation. Additionally, a system may downgrade its functionality autonomously if a failure cannot be compensated by dynamic adaptation, which is referred to as graceful degradation [She03]. For example, if the sensor measuring the yaw rate of a car fails, the vehicle stability control system may adapt to a configuration where the yaw rate is approximated by steering angle and vehicle speed. Of course, the systems may upgrade again in case of transient failures to provide the best functionality possible with the currently available resources.

However, adaptation significantly complicates the development of embedded systems due to the fact that adaptation of a component affects the quality of the services it provides, which may in turn cause adaptations in other components. As a result, sequences of adaptations may take place that are hard to analyze. For this reason, it is not sufficient to consider each configuration separately to ensure system correctness. Instead, the adaptation process as a whole has to be checked, which is a complex and error-prone task, at least without tools for computer-aided verification, since the number of configurations a system may adapt to is, in the worst case, exponential in the number of components.

A promising approach to deal with the increased complexity posed by adaptation is model-based design. In this paper, we present a constructive modeling technique for the design of adaptive embedded systems. As a major advantage, our approach hides the complexity at the system level by fostering independent specification of functionality and adaptation behavior. To this end, the adaptation behavior of the components is specified independently from the functionality. The adaptation behavior of the whole system is determined by the adaptation behavior of the system's components. Furthermore, our approach allows the designer to analyze, validate, and to verify the adaptation behavior. The integration of formal verification into the development process is particularly important to prove that the adaptation behavior meets critical requirements such as deadlock-freeness. We demonstrate the feasibility of our approach by applying it to a vehicle stability control system, which supports dynamic adaptation to react to changing driving situations and graceful degradation in case of failures.

The remainder of this paper is structured as follows: In Section 2, we present the vehicle stability control system. Moreover, we show how adaptation and graceful degradation can be used to increase safety and availability. Section 3 discusses our modeling concepts for the development of adaptive embedded systems. In Section 4, we propose a development process integrating modeling and analysis. Experimental results on the verification of the vehicle stability control system are given in Section 5. Finally, we discuss related work in Section 6 and conclude with a summary and an outlook to future work in Section 7.

2 Vehicle Stability Control System

Our vehicle stability control system consists of three typical software-intensive vehicle functions, which have been implemented on a remote controlled car scaled 1:5. The first vehicle function is a steering angle delimiter, which restricts the steering angle depending on the current velocity of the car. This function helps to prevent the car from spinning if the steering angle intended by the driver might cause instabilities. The second function is a traction control system, which controls the slip of the rear wheels. If the slip exceeds a given threshold, the acceleration of the car is restricted by reducing the gas to prevent loss of traction at drive-away. The third function implements a yaw rate corrector influencing the brakes. For example, if the driver wants to go straight, the system tries to keep the yaw rate close to zero. Normally, a yaw rate controller can be used in any driving situation. In our system, however, the function only considers straightforward driving. This simplification is well-suited for our purposes, since it facilitates the implementation of the functional behavior, while providing the same challenges for modeling and analyzing adaptation and graceful degradation.

In order to implement these vehicle functions, the car is equipped with typical sensors: At each wheel, a sensor is mounted for measuring the revolution. Additionally, there are sensors measuring longitudinal acceleration, lateral acceleration, and yaw rate. Besides the values measured by the sensors, the system receives as input the desired speed and the desired steering angle via remote control from the user. The system controls four servos: The first servo controls the gas and actuates the rear brakes. Hence, the rear wheels are decelerated with the same force. The front brakes have their own servos and are actuated independently from each other. The fourth servo controls the steering.

The system architecture is split into logical sensors, controllers, and logical actuators as depicted in Figure 1. The logical sensors/logical actuators implement the mapping between physical sensor/actuator values and their logical interpretation (e.g., slip of wheels and nominal yaw rate). The controller component contains three subcomponents implementing the vehicle functions described above based on logical sensor and actuator values.

The components in *LogicalSensors* are self-adaptive, since the most reasonable calculation variant for a sensor value typically depends on the current driving situation. This even holds for logical sensor values, where the relation to measured sensor values seems to be straightforward. For example, component *AyCalc*, which determines the lateral acceleration *ay*, implements two different modes of operation, each providing a different quality level. In the following, we will call these externally 'visible' modes of operation *configurations*. For instance, configuration *Measured* derives *ay* from the values provided by an acceleration sensor. However, when the car is going down- or uphill, the acceleration sensor values are influenced by gravity. For this reason, an alternative configuration *VYawVCarRefBased* is used, which calculates *ay* from the speed of the car along the x-axis (*v_carref*) and the angular speed of the car around its z-axis (*v_yaw*).

Dynamic adaptation in the logical sensor components is a powerful and cost-efficient way to minimize the effects of sensor defects. In many cases, it is even possible to fully compensate the occurrence of faults. For instance, the component *VYawCalc* has four different

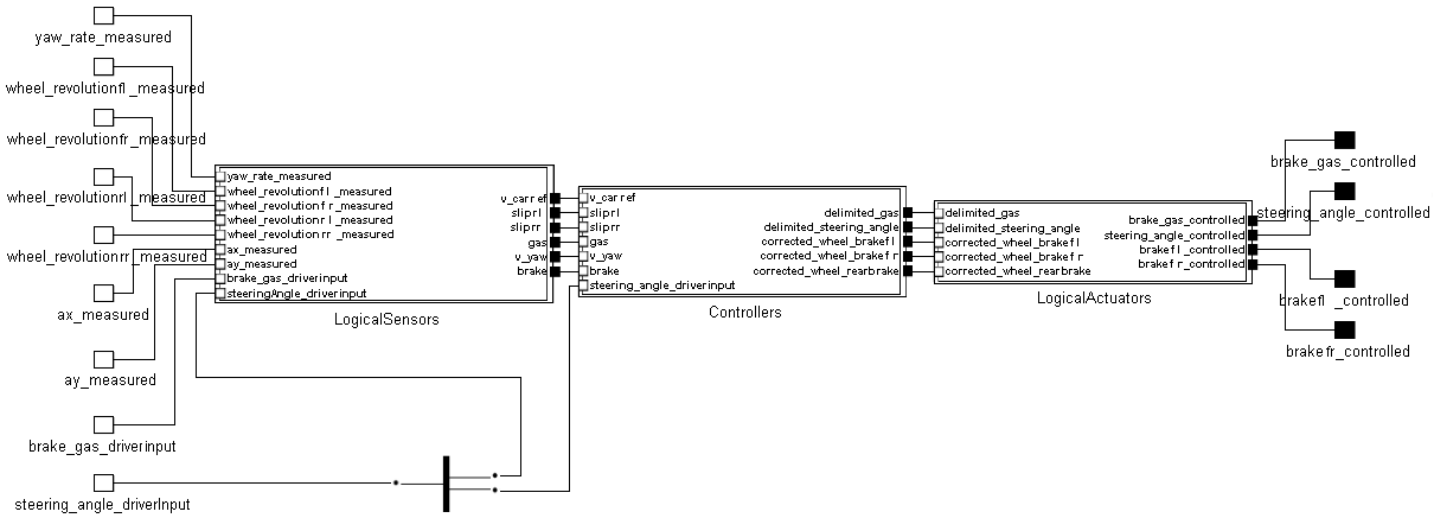


Figure 1: Architecture of the vehicle stability control system

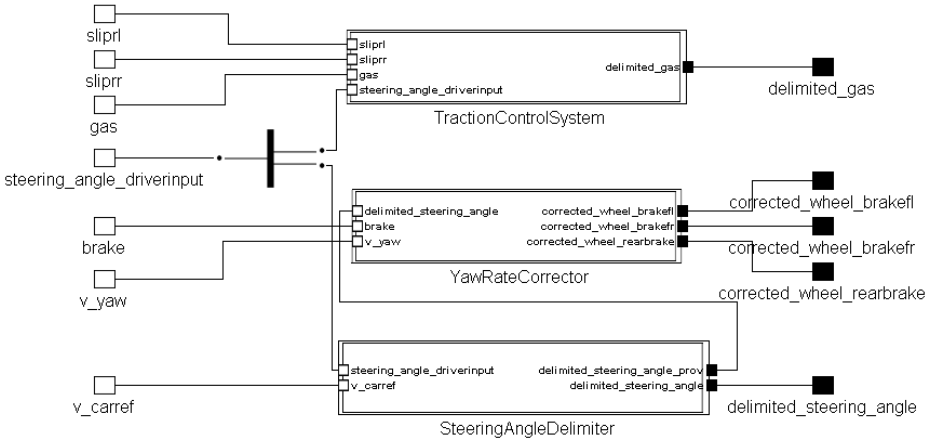


Figure 2: Internal structure of the controller component

configurations for determining the angular speed of the car around its z-axis (*v_yaw*). Configuration *Measured* uses a yaw rate sensor, configuration *SteeringBased* derives *v_yaw* from the steering angle and the speed of the car along the x-axis (*v_carref*), configuration *RWheel* uses the rear wheel speeds to determine *v_yaw*, and configuration *AyBased* uses the lateral acceleration *ay* and *v_carref*. In all configurations, the quality of the provided value is sufficient to perform a yaw rate correction. This means that component *VYawCalc* is able to adapt to the most appropriate configuration according to the availability of the sensors without downgrading the functionality of the system.

In contrast to the logical sensors and actuators, the configurations of the controller components directly correspond to degradation levels of the vehicle functions. Since the considered vehicle functions are reasonably simple, each of the three controller components shown in Figure 2 implements only one safe fall-back layer. Besides the configuration that provides the full functionality, there is another configuration that implements a safe subset of the functionality based on minimal input. The idea behind this fall-back configuration is to ensure that the system is still operational even if the full functionality cannot be provided. Additionally, each component has a configuration *Off* in which the component is shut down. This configuration is primarily used for validation and verification, as shutting down whole components is undesirable in practice.

As an example of a controller component, consider *TractionControlSystem*, which has the configurations *RearWheelSlipControl*, *SlowStart*, and *Off*. The configuration *RearWheelSlipControl* only increases the gas if the slip is below a given threshold. This configuration requires the slip of the right and the left rear wheels as well as the intended gas value for calculating the delimited gas value. In order to avoid a total loss of the delimited gas value when the slip of the wheels cannot be determined or is erroneous for some reason, the component has the fall-back configuration *SlowStart*, which only requires the desired gas value as input. This configuration limits the increase of the gas in a very conservative way by taking into account the last desired gas value.

3 Modeling Adaptation Behavior

As mentioned previously, one major problem in specifying how a system has to adapt in case of failures stems from complex interdependencies between different components. These interdependencies are due to the fact that adaptation of a component may affect the quality of the service it provides, which has to be taken into account by the influenced components. Hence, adaptation of a component may trigger a chain reaction of adaptations in other components until a valid system configuration is reached. As another problem, most vehicle functions are not disjoint, i.e., a component may be used by more than one function at a time.

For these reasons, a constructive modeling technique hiding the complexity at the system level is essential for specifying complex adaptation behavior. In our research project MARS (Methodologies and Architectures for Runtime Adaptive Embedded Systems), we have developed a modeling approach that separates functionality from adaptation behavior by establishing a quality flow in the system [TAFJ07]. This allows encapsulating the specification of the adaptation behavior and modeling it as a separated view [MBE+00] within the components. In this way, developers can focus on the adaptation behavior of single components until the whole system is specified. In a second step, validation and verification techniques may be employed to check correctness of the adaptation behavior at the system level.

3.1 Quality Flow

In architecture description languages [MT00], component interfaces are defined by input and output ports. Each port has an associated data type and can be linked with other ports. The relationship between the data types is often formalized in a global type system, which makes it possible to perform automatic type checks. Components communicate with each other using signals, which are, from a functional point of view, instances of data types.

Corresponding to interfaces in architecture description languages and the data flow between components, we introduce adaptation interfaces and establish a quality flow based on instances of quality types (QT). QTs are used to augment data values with a description of their quality. All QTs are part of a quality type system (QTS), which is an inheritance tree with root node *BASICQT* (see Figure 3).

The root QT (*BASICQT*) has two subtypes *AVAILABLE* and *UNAVAILABLE*. The latter indicates that a data value is unusable due to defects, whereas the former indicates that a data value has no defect making it unusable. All other QTs in the QTS are inherited from *AVAILABLE* and have semantics describing the intuition behind the related data value. For instance, the QT *MEASURED* referring to data values calculated by configuration *Measured* of component *AyCalc* has the meaning ‘gravity-influenced lateral acceleration based on acceleration measurements’. The configuration *VYawBased* of the component *AyCalc* delivers values with the semantics ‘slip-influenced lateral acceleration based on calculations using *v_carref* and *v_yaw*’ (QT *VYAW_BASED*). The QT *AY* with semantics ‘lateral

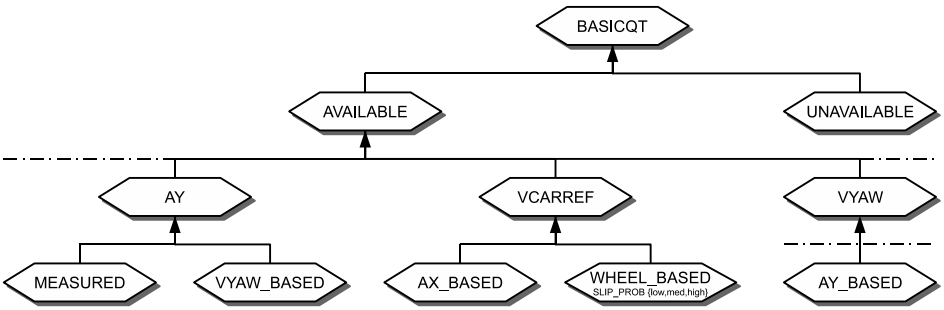


Figure 3: Example for a quality type system (QTS)

acceleration' subsumes these two different QTs for lateral acceleration. Moreover, a QT may define a set of attributes enabling a quantitative and more precise description of the quality. For example, the QT *WHEEL_BASED*, which means 'wheel-based approximation of car speed in x-direction' has an attribute *SLIP_PROB* with the range $\{low, med, high\}$ providing information about the likelihood that a calculated value is affected by the slip of the wheels.

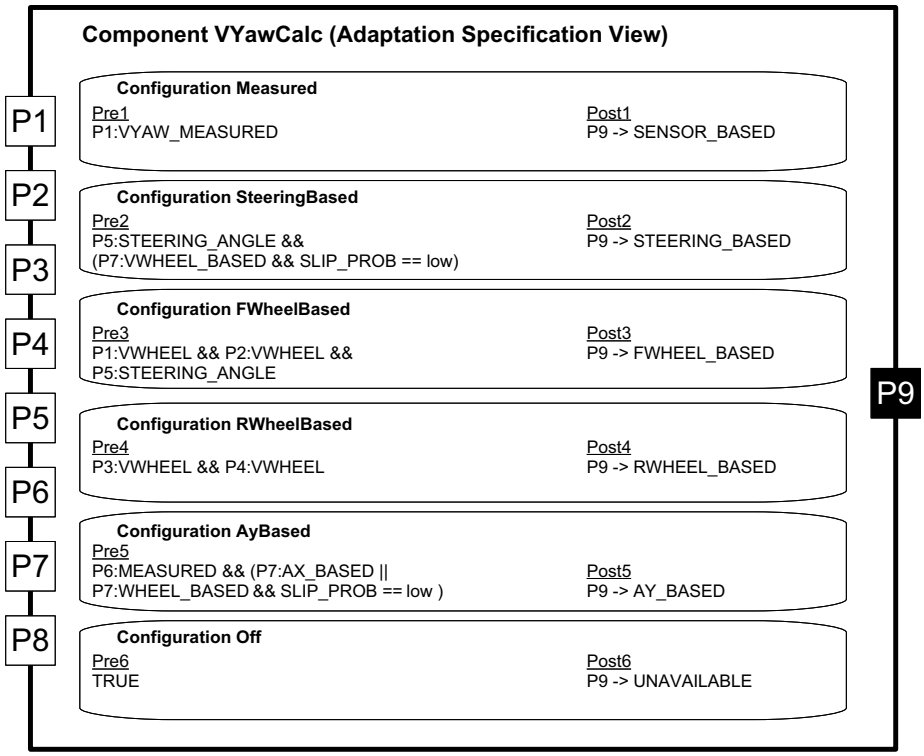
3.2 Specification of Adaptation Behavior

In analogy to the functional interface, the adaptation interface is defined by quality ports. For every data port, a quality port is added to the adaptation interface. Quality ports have a reference to a QT and are declared either as *required* or *provided*. In the same way as the distinction between input and output ports defines the direction of the data flow, the distinction between *required* and *provided* ports defines the direction of the quality flow. Connectivity checks between ports can be extended to the adaptation interface and inconsistent adaptation behavior through directly connected components can be identified automatically. Quality type checks are performed by verifying that the QT of a *provided* port is equal to or inherited from the QT of the connected *required* port.

To specify the adaptation behavior, we attach to each configuration a pair consisting of a precondition and a postcondition. The precondition is used to select the current configuration according to the QTs at the *required* ports of the adaptation interface. The postcondition assigns a QT to every *provided* port of the adaptation interface. The preconditions are evaluated at runtime, where the one with the highest priority, which is implicitly given by their order, determines the configuration to be selected. Once the selected configuration has become active, the QTs at the *provided* ports are set according to the given postcondition.

For example, the component *VYawCalc* contains nine quality ports and six configurations (see Figure 4). The eight quality ports referring to input ports are declared as *required* and the quality port referring to the output port *v_yaw* is declared as *provided*. The configuration *AyBased* implements a yaw rate calculation from the measured velocity *v_carref* and

the lateral acceleration *ay*. Its precondition states that the quality at port *ay* is an instance of QT *MEASURED*. A quality of QT *VYAW_BASED* is forbidden, since it is not reasonable to derive *v_yaw* from *ay* and *ay* from *v_yaw* at the same time. The quality at the port *v_carref* has to be an instance of QT *AX_BASED* or *WHEEL_BASED*. If the quality is of QT *WHEEL_BASED*, then the attribute *SLIP_PROB* must have the value *low*. The last pair of conditions always refers to the configuration *Off*. Clearly, the corresponding precondition has always to be valid and the postcondition assigns the QT *UNAVAILABLE* to all provided qualities.



port number	port name	referenced QT	quality flow
P1:	yawrate	VYAW_MEASURED	required
P2:	v_wheelfl	VWHEEL	required
P3:	v_wheelfr	VWHEEL	required
P4:	v_wheelrl	VWHEEL	required
P5:	v_wheelrr	VWHEEL	required
P6:	steering_angle	STEERING_ANGLE	required
P7:	ay	AY	required
P8:	v_carref	VCARREF	required
P9:	v_yaw	VYAW	provided

Figure 4: Adaptation specification for component *VYawCalc*

4 Development Process for Adaptive Systems

In the following, we will briefly describe the design flow of our approach, which is depicted in Figure 5. This design flow has been used successfully for the development of the vehicle stability control system.

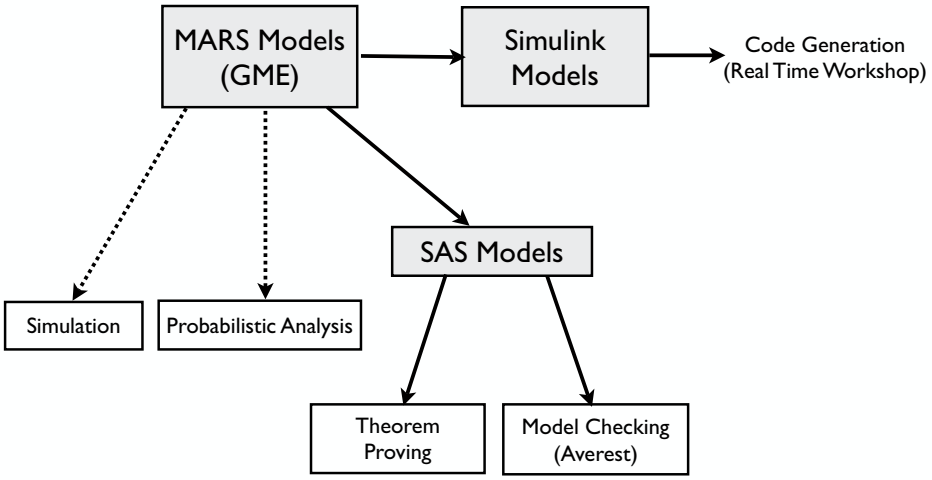


Figure 5: Design flow for the development of adaptive systems

The MARS modeling concepts have been integrated into the Generic Modeling Environment GME[1] [LMB+01] by providing a GME meta-model. GME automatically produces a model representation in XML format, which can be used as input for validation and verification as well as for code generation. Regarding validation, we support the simulation of adaptation behavior and the visualization of reconfiguration sequences using adaptation sequence charts [TAFJ07]. Moreover, it is possible to perform a probabilistic analysis of the adaptation behavior [AFT07]. To this end, the adaptation behavior model is translated into an equivalent hybrid component fault tree. The probability that a configuration of a component is activated can then be derived from the failure rates of the sensors and actuators.

Moreover, MARS models can be formally verified by automatically translating them into synchronous adaptive systems (SAS) [ASSV07]. SAS are a formal framework that allows capturing MARS models at a high level of abstraction and reasoning about their semantics by theorem proving techniques. Moreover, SAS can be used to perform model reductions, e.g., slicing [SPH08], in order to make the models amenable to automatic verification tools, e.g., symbolic model checkers. The correctness of the reductions is established with a translation validation technique [BSPH07] by generating proof scripts for an interactive theorem prover [NPW02].

[1] `http://www.isis.vanderbilt.edu/projects/gme/`

As a model checking back-end, we use Averest,[2] a framework for the specification, implementation, and verification of reactive systems [SS05]. In Averest, a system is given in a synchronous programming language, which is well-suited for describing adaptive systems obtained from SAS models, as both are based on a synchronous semantics. Furthermore, a causality analysis of synchronous programs can be used to detect cyclic dependencies that may occur if the quality flow generated by an output is used as input in the same component. Specifications can be given in temporal logics such as CTL and LTL (see Section 5).

Once the adaptation behavior of a MARS model has been successfully validated and verified, it can be implemented by means of a given adaptation framework. Such a framework might either be a central component in the system coordinating all adaptation processes or be distributed over several components. We have developed a distributed framework in MATLAB-Simulink,[3] which provides an interface for the integration of the functionality and supports code generation using Simulink's Real-Time Workshop. Additionally, the framework offers a generic configuration transition management, which can be used to specify the interaction between functionality and adaptation behavior in more detail. The resulting design can then be evaluated by co-simulations of functional and adaptation behavior. In order to avoid undesirable interference between functional and adaptation behavior, additional techniques may be employed that guarantee smooth blending between functional outputs during reconfigurations [Bei07].

5 Experimental Results

In this section, we present experimental results on the verification of the vehicle stability control system using model checking. Table 1 shows some characteristics of the system.

Number of components	28
Number of configurations	70
Lines of code	≈ 2500
Number of reachable states	$\approx 5 \cdot 10^{18}$
Number of properties	151

Table 1: Characteristics of the vehicle stability control system

As specification languages for reasoning about adaptation behavior, we employ the temporal logics CTL (computation tree logic) and LTL (linear time temporal logic) [CGP99, Sch03]. In both CTL and LTL, temporal operators are used to specify properties along a computation path that corresponds to an execution of the system. For example, the formula $\mathsf{F}\varphi$ states that φ eventually holds and $\mathsf{G}\psi$ states that ψ invariantly holds on a given path. In CTL, every temporal operator must be immediately preceded by one of the path quantifiers A (for all paths) and E (at least one path). Thus, $\mathsf{AG}\varphi$ and $\mathsf{EF}\psi$ are CTL formu-

[2] `http://www.averest.org`
[3] `http://www.mathworks.com`

lae stating that φ invariantly holds on all paths and ψ eventually holds on at least one path, respectively. LTL formulae always have the form $\mathsf{A}\varphi$, where φ does not contain any path quantifiers. None of these two logics is superior to the other, i.e., there are specifications that can be expressed in LTL, but not in CTL, and vice versa [CGP99, Sch03].

Most of the properties concerning adaptation behavior can be divided into four generic classes of CTL formulae. First, we want to verify that no component gets stuck in the shutdown configuration *Off*. The following specification states that whenever a component is in configuration *Off*, there exists a path such that the component is eventually in a configuration different from *Off* (the variable c stores the current configuration).

Property 1 (liveness): $\mathsf{AG}(c = \mathit{Off} \rightarrow \mathsf{EF}\, c \neq \mathit{Off})$

The next property states that every component can reach all configurations at all times. If this specification holds, the system is deadlock-free and no configuration is redundant.

Property 2 (reachability): $\mathsf{AG}(\bigwedge_{i=1}^{n} \mathsf{EF}\, c = \mathit{config}_i)$

Moreover, a component must always be in one of the predefined configurations such that no inconsistent states can be reached.

Property 3 (safety): $\mathsf{AG}(\bigvee_{i=1}^{n} c = \mathit{config}_i)$

Additionally, we want to verify that no configuration is always only transient, i.e., that it can be active for an arbitrary amount of time. The following formula holds iff a component can stay in a configuration.

Property 4 (persistence): $\bigwedge_{i=1}^{n} \mathsf{EFEG}\, c = \mathit{config}_i$

Finally, an important property of adaptive systems is stability [SPH06, ASSV07]. Since adaptation in our approach is not controlled by a central authority, adaptation in one component may trigger further adaptations in other components. While *finite* sequences of adaptations are usually intended, cyclic dependencies between components may lead to an *infinite* number of adaptations, which results in an unstable system. For this reason, we want to verify that the configurations of a component eventually stabilize if the inputs do not change. In contrast to Properties 1–4, stability cannot be expressed in CTL, but in LTL. Suppose that φ_{input} holds iff the inputs are stable for one unit of time. Moreover, let $\varphi_{\mathsf{config}}$ hold iff the configurations are stable for one time unit. Then, the system is stable iff the following formula holds:

Property 5 (stability): $\mathsf{AG}(\mathsf{G}\varphi_{\mathsf{input}} \rightarrow \mathsf{FG}\varphi_{\mathsf{config}})$

Intuitively, the formula states that the configurations will stabilize after a finite amount of time whenever the inputs do not change. In principle, we could use standard model checking procedures for LTL to verify stability. However, as shown in [ASSV07], this is often rather inefficient. For this reason, we employed the approach presented in [ASSV07], which turned out to be more efficient in practice (see below).

Table 2 shows the number of fixpoint iterations performed during model checking and the time required to check Properties 1–4 on a Pentium 4 processor with 2.8 GHz and 512 MB RAM. As these properties actually represent sets of formulae (one formula for each component/configuration), we have determined minimal, average, and maximal values in addition to the total ones. The total time required to check all properties was less than three minutes. Stability could not be checked in less than one hour by means of standard model checking procedures for LTL. However, using the approach presented in [ASSV07], it could be checked in less than half an hour (1723 seconds). Nevertheless, there is still a large gap compared to the runtimes for Properties 1–4. The main reason for this is that the latter can benefit from abstractions such as cone of influence reduction [CGP99], whereas checking stability requires considering the complete system.

Property	**Fixpoint Iterations**				**Time [seconds]**			
	Min.	**Avg.**	**Max.**	**Total**	**Min.**	**Avg.**	**Max.**	**Total**
P1 (liveness)	5	8.3	17	223	< 0.1	3.1	71.5	84.0
P2 (reachability)	9	18.4	43	496	< 0.1	2.4	52.1	63.8
P3 (safety)	2	2.0	2	54	< 0.1	0.1	0.2	0.7
P4 (persistence)	6	10.8	20	758	< 0.1	0.3	4.5	19.6

Table 2: Experimental results of model checking

It should be mentioned that not all of these properties were satisfied in the initial design of the vehicle stability control system. For example, it turned out that the configuration *FWheelBased* of the component *VYawCalc* was never active for more than a single cycle (violation of Property 4). This behavior was due to interdependencies between the component *VYawCalc* and other components that caused it to switch to the configuration *SteeringBased* in the next cycle. Although this particular case might not be safety-critical, the results of verification are often useful for gaining a better understanding of the whole system.

Besides the properties described above, which are concerned with generic features of the adaptation behavior, we also checked a number properties ensuring that the system always provides its basic functionality given that the minimal required sensors and actuators are operational. In particular, if the sensors and actuators concerning steering angle, brake force, and gas are operational, neither of the corresponding controller components is in the shutdown configuration *Off*. In this way, it is guaranteed that the adaptation behavior implemented in the system does not corrupt any essential functionality. For example, if the intended steering angle is available and the steering servo works, i.e., *delimited_steering_angle_quality* is available, then the component *SteeringAngleDelimiter* is not in the configuration *Off*:

$$\mathsf{AG}((\mathit{steering\_angle\_driverInput\_quality} = \mathit{available} \wedge \mathit{delimited\_steering\_angle\_quality} = \mathit{available}) \rightarrow c(\mathit{SteeringAngleDelimiter}) \neq \mathit{Off})$$

As another example, the following property ensures that the braking system does not fail

if the required actuators are operational:

$$\mathsf{AG}((\mathit{corrected\_wheel\_brakeFL\_quality} = \mathit{available} \wedge \\ \mathit{corrected\_wheel\_brakeFR\_quality} = \mathit{available} \wedge \\ \mathit{corrected\_wheel\_rearBrake\_quality} = \mathit{available}) \rightarrow c(\mathit{YawRateCorrector}) \neq \mathit{Off})$$

6 Related Work

In recent years, a number of frameworks for dynamic adaptation and reconfiguration have been developed, e.g., [RL05, COWL02]. Most of these frameworks are based on a dedicated control that contains all information on the adaptation behavior and triggers adaptation of the components. Thus, even minor changes in one of the components might require a complete redesign of the component controlling the adaptation behavior. In contrast, our approach does not depend on a central control encoding the complete adaptation behavior. This simplifies the design of adaptive systems and improves their maintainability.

A major challenge in the model-based development of adaptive systems is to find models that lead to reasonable and safe degradation behavior. To solve this problem, the verification of adaptation behavior has become an active area of research [KB04, Str05, ZC05, ZC06]. However, these approaches take the specification of the system for granted and provide no constructive modeling technique. Moreover, directly specifying the global adaptation behavior as required by these approaches is hardly feasible for complex systems like the Electronic Stability Program (ESP).

Regarding the modeling of adaptation behavior, approaches like [SKN01] or the MUSIC project[4] have adopted ideas from variability modeling used to manage product lines. The basic idea of these approaches is to shift the binding time of variation points from design time to runtime. Thus, the systems autonomously determine all valid configurations at runtime and select one of them according to an objective function. However, determining the next configuration in this way is computationally intensive and thus too slow for real-time systems. Furthermore, the complexity of computing the next configuration strongly limits model-based analysis, validation, and verification, which are indispensable in safety-critical areas like the automative domain.

For a detailed description of MARS, the reader is referred to [TAFJ07]. Further information on the translation of MARS models to SAS and their verification can be found in [ASSV07]. Initial work on the verification of MARS models was presented in [SST06]. In this paper, we improved on [TAFJ07, ASSV07] in two ways: First, we introduced a new quality type system based on inheritance trees. Second, we applied our approach to a vehicle stability control system to demonstrate its feasibility for large real-world examples.

[4] `http://www.ist-music.eu/`

7 Conclusion and Future Work

Adaptation and graceful degradation have become the state of the art in the automotive domain to meet the high demands on safety and availability. Adaptation is frequently employed to react to changing driving situations that require switching between different modes of operation. Graceful degradation enables a system to continue operating properly in case of failures, e.g., defective sensors. However, adaptation and graceful degradation significantly complicate the design of a system.

We presented a constructive modeling approach for the development of adaptive embedded systems, which facilitates independent specification of adaptation behavior and functionality. This helps the designer to focus on each aspect separately and to resolve complex interdependencies. Furthermore, conventional techniques for the design of fault-tolerant systems can be integrated smoothly into our approach. For example, fault-tolerance with respect to certain input values can be modeled by a scenario where a component adapts to a configuration that does not affect the quality of the provided service.

Our approach supports the integration and belated specification of legacy components, e.g. components whose adaptation behavior is unspecified. This is accomplished by assigning the basic quality type *BASICQT* to every quality port and by establishing appropriate preconditions/postconditions. The component's adaptation behavior can then be successively refined until the desired behavior is obtained. In this way, ASCET-SD models from the industry have been successfully annotated.

In safety-critical areas, careful analysis of the adaptation behavior is a crucial concern, as interdependencies between the components of a system may result in complex adaptation sequences that are hard to analyze. Besides validation techniques such as simulation, our approach also supports formal verification. Our experimental results indicate that most properties concerning the adaptation behavior of complex systems such as the vehicle stability control system can be verified efficiently within a couple of minutes.

In our future work, we want to extend the approach presented in this paper by concepts for the hierarchical specification and compositional analysis of adaptive systems. Furthermore, we plan to model at a high level of abstraction the relationship between a configuration and its underlying functional behavior. In this way, the functionality can be analyzed in combination with the adaptation behavior at the modeling level before the functionality is actually implemented.

References

[AFT07] R. Adler, M. Förster, and M. Trapp. Determining Configuration Probabilities of Safety-Critical Adaptive Systems. In *International Symposium on Ubisafe Computing (UbiSafe'07)*, Niagara Falls, Canada, 2007. IEEE Computer Society.

[ASSV07] R. Adler, I. Schaefer, T. Schuele, and E. Vecchié. From Model-Based Design to Formal Verification of Adaptive Embedded Systems. In *International Conference on Formal*

Engineering Methods (ICFEM'07), volume 4789 of *LNCS*, pages 76–95, Boca Raton, USA, 2007. Springer.

[Bei07] Andreas Beicht. Entwicklung eines Frameworks zur Entwicklung und Analyse adaptiver eingebetteter Systeme, 2007. Diplomarbeit, TU Kaiserslautern, Germany.

[BSPH07] J.O. Blech, I. Schaefer, and A. Poetzsch-Heffter. Translation Validation of System Abstractions. In *Workshop on Runtime Verification (RV'07)*, volume 4839 of *LNCS*, pages 139–150, Vancouver, Canada, 2007. Springer.

[CGP99] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT, London, England, 1999.

[COWL02] J.M. Cobleigh, L.J. Osterweil, A. Wise, and B. Lerner. Containment Units: A Hierarchically Composable Architecture for Adaptive Systems. In *Symposium on Foundations of Software Engineering (SIGSOFT FSE'02)*, Charleston, USA, 2002. ACM.

[KB04] S.S. Kulkarni and K.N. Biyani. Correctness of Component-Based Adaptation. In *Symposium on Component Based Software Engineering (CBSE'04)*, volume 3054 of *LNCS*, pages 48–58, Edinburgh, Scotland, 2004. Springer.

[LMB+01] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi. The Generic Modeling Environment. In *Workshop on Intelligent Signal Processing (WISP'01)*, Budapest, Hungary, 2001. IEEE.

[MBE+00] S. Mann, A. Borusan, H. Ehrig, M. Grosse-Rhode, R. Mackenthun, A. Sünbül, and H. Weber. Towards a Component Concept for Continuous Software Engineering. Technical Report 55/00, Fraunhofer-ISST, 2000.

[MT00] N. Medvidovic and R.N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1), 2000.

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

[RL05] O.A. Rawashdeh and J.E. Lumpp, Jr. A Technique for Specifying Dynamically Reconfigurable Embedded Systems. In *Aerospace Conference*, Big Sky, USA, 2005. IEEE.

[Sch03] K. Schneider. *Verification of Reactive Systems – Formal Methods and Algorithms*. Texts in Theoretical Computer Science (EATCS Series). Springer, 2003.

[She03] C.P. Shelton. *Scalable Graceful Degradation for Distributed Embedded Systems*. PhD thesis, University of Pittsburgh, Pennsylvania, USA, 2003.

[SKN01] C.P. Shelton, P. Koopman, and W. Nace. A Framework for Scalable Analysis and Design of System-Wide Graceful Degradation in Distributed Embedded Systems. In *Workshop on Reliability in Embedded Systems*, New Orleans, USA, 2001. IEEE.

[SPH06] I. Schaefer and A. Poetzsch-Heffter. Towards Modular Verification of Stabilisation in Self-Adaptive Embedded Systems. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'06)*, volume 4280 of *LNCS*, pages 584–585, Dallas, USA, 2006. Springer.

[SPH08] I. Schaefer and A. Poetzsch-Heffter. Slicing for Model Reduction in Adaptive Embedded Systems Development. In *Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2008)*, 2008.

[SS05] K. Schneider and T. Schuele. Averest: Specification, Verification, and Implementation of Reactive Systems. In *Conference on Application of Concurrency to System Design (ACSD'05)*, St. Malo, France, 2005.

[SST06] K. Schneider, T. Schuele, and M. Trapp. Verifying the Adaptation Behavior of Embedded Systems. In *Software Engineering for Adaptive and Self-Managing Systems (SEAMS'06)*, pages 16–22, Shanghai, China, 2006. ACM.

[Str05] E.A. Strunk. *Reconfiguration Assurance in Embedded System Software*. PhD thesis, University of Virginia, Charlottesville, USA, 2005.

[TAFJ07] M. Trapp, R. Adler, M. Förster, and J. Junger. Runtime Adaptation in Safety-Critical Automotive Systems. In *IASTED International Conference on Software Engineering (SE'07)*, Innsbruck, Austria, 2007. ACTA.

[ZC05] J. Zhang and B.H.C. Cheng. Specifying Adaptation Semantics. In *Workshop on Architecting Dependable Systems (WADS'05)*, pages 1–7, St. Louis, USA, 2005. ACM.

[ZC06] J. Zhang and B.H.C. Cheng. Model-Based Development of Dynamically Adaptive Software. In *International Conference on Software Engineering (ICSE'06)*, pages 371–380, Shanghai, China, 2006. ACM.

Reuse of Innovative Functions in Automotive Software: Are Components enough or do we need Services?

Holger Giese
Hasso Plattner Institute at the University of Potsdam,
Prof.-Dr.-Helmert-Str. 2-3, D-14482 Potsdam, Germany
holger.giese@hpi.uni-potsdam.de

Abstract: Software has become an important factor for the development of modern vehicles and is of paramount importance for many innovative functions. While some more traditional functions can be allocated to a single ECU, many innovate functions of an automotive system require the cooperation of different parts of the vehicle or even multiple vehicles. In this position paper, it is advocated that the component-based approach advocated in approaches such as AUTOSAR is at most sufficient for the reuse of commodity functions while the reuse of innovative functions which inherently require coordination cannot be addressed using the component-based paradigm. The service-oriented approach is sketched as alternative and the benefits with respect to the sketched reuse problem for innovative functions are discussed.

1 Introduction

Software has become an important factor in the development of modern high-end vehicles. This fact is highlighted by the observation that the size of the software is growing at an exponential rate and that today about 70% of the innovations in these cars are software driven (cf. [HKK04]). Therefore, the percentage of costs due to the development of software is expected to grow from 20-25% up to 40% in the next few years (cf. [Gri03]).

Single control devices have been employed in the past to fulfill a single function (often replacing a pure mechanical device). Connections to other devices were required only in rare cases. Today, however, more and more functions can only be realized by the cooperation of different control devices and therefore a dramatic increase of the complexity can be observed (cf. [Gri03]). In the future functions realized by multiple vehicles will further emphasize the need for coordination.

As example for an innovative function which requires coordination we consider an intelligent energy management for cars such Dynamic Electrical Energy Management[1] from Siemens VDO or eBalance [RSS05] from Hella KGaA Hueck & Co., which guarantee that the storage devices are always properly charged and that, in case of problems, the energy consumption of comfort functions is restricted.

In this position paper it is advocated that the component-based approach employed in approaches such as AUTOSAR is at most sufficient for the reuse commodity functions

[1] http://www.siemensvdo.com/products_solutions/chassis-carbody/body_chassis_electronics/battery-energy-management/Battery-and-energy-management-BEM.htm

while innovative functions with required coordination due to their characteristics cannot be handled using the component-based paradigm in an appropriate manner. In addition the vision of a service-oriented approach presented in more detail in [Gie08] is sketched as alternative and the benefits are discussed. In this approach components are used as basic units which provide localized basis functionality which can be reused, while services capture complex reusable functionality which requires the interaction of multiple units. In [Gie08] it is sketched how all phases of the development process can benefit from a service-oriented approach and that advanced synthesis techniques can be employed to reuse the components and services and compose them with only minimal manual efforts. The approach exploits that functions in automotive systems only have to be recombined in a restricted manner in order to enable reuse across car series.

It is important to emphasize that the advocated concepts are to some extent already implicitly present in the current practice. We therefore think that the service-oriented view is not only the result of the employed concepts but already inherently present in the function-oriented description which is commonly employed in the automotive field.

This position paper is structured as follows: We first discuss in Section 2 how the example function can be addressed in a pure component-based approach. It follows the description of the service-oriented alternative in Section 3 and the benefits are discussed. Finally, we provide a conclusion and an outlook on planned future work.

2 The Component-Based Approach

A component-based architecture ensures that the components only interact with their environment via well-defined interfaces (cf. [Szy98]) and thus provide a reasonable decoupling of the different components. Today an open and flexible software architecture which facilitates reuse is often missing in automotive systems and thus many functions are nearly built from scratch for each vehicle model (cf. [Gri03]). However, initiatives such as AUTOSAR[2] are the first step towards and open and flexible component-based software architecture.

Example: Following this component-based principle, an architecture for an intelligent energy-management system inspired by the eBalance system [RSS05] can be derived as depicted in Figure 1 only considering the employed UML standard notation for components, ports, and connectors.

A core component serves as the central management unit which controls all consumers and observes all producers. The core may in addition interact with the higher-level vehicle control by providing relevant status information and reading relevant information about the vehicle such as the current operation mode and requested system characteristics (omitted here). In addition, special driver components operate as wrappers to the different energy producers and consumers, and can be either located on the central ECU or on the local ECU of the producer resp. consumer. Between the driver and core component, a standard

[2] www.autosar.org

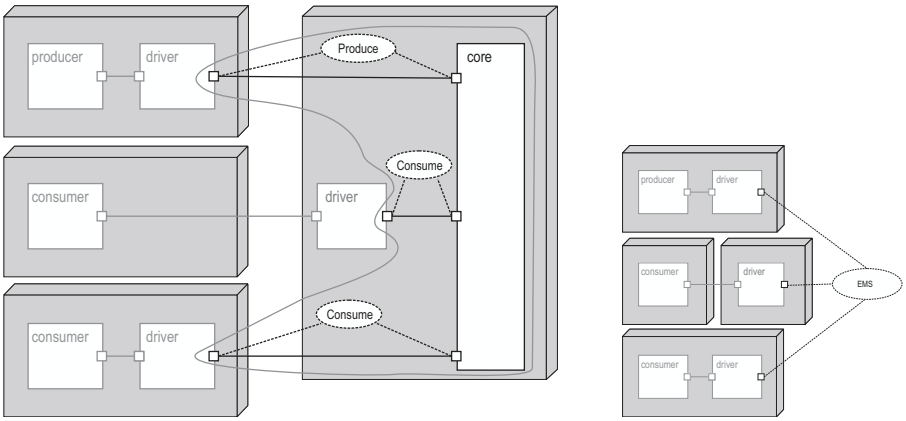


Figure 1: Architecture for the software of the energy management system (cf. [RSS05, Gie08])

protocol can be employed which permits reusing the core in many settings, while the driver components have to be adjusted depending on the application-specific interface they can employ to control and monitor the different producers and consumers.

Discussion: In case of a component-based architecture the definition of standard interfaces for the software components ensures that components from different suppliers and vendors can technically interoperate. However, at the application level the need for an application-specific design of the coordination between these components still hinders reuse. In our example this means that the components have to be developed taking a standardized interface for the eBalance system into account. At least they have to provide any energy management interface which can then be employed using the drivers: an assumption which is not justified for the considered innovative functions.

Also software product families for automotive software (cf. [TH02]) are a promising direction for different variants of software for a type series using components. However, the resulting reuse is restricted to a single product family as the software components can only be reused if the architectural context is quite similar and the variation can be pre-planned. Thus in our problem the eBalance system functionality has to be taken into account upfront when planning the product line. Again this is a not realistic assumption.

The observed problems of the component-based approach to cope with the considered innovative additional function are crucial, as innovative functions in software are a main competitive factor for automotive systems. Therefore, we cannot assume that for these functions integration into employed standard product-line architectures is already at hand when required. In contrast, the task which is required to enhance a vehicle or vehicle family by the innovative function is to enhance the architecture or product line architecture.

This required enhancement is particularly difficult, as the different software components of a car or car series are developed by a multitude of suppliers within a complex vendor supplier network. Thus in contrast to standard software interfaces, the interfaces must not only serve as means to protect and decouple the components of the developed system from

each other in order to handle the development complexity, but to also guarantee that the division of labor can really take place. Thus a solution is required where reuse can also happen at each stage of a complex supplier chain.

3 The Service-Oriented Alternative

Following the definition of services in [DMRK05, BKM07], we define a service by "the interaction among entities involved in establishing a particular functionality."[3] The entities are further called the service roles. This general definition includes simple request/response interaction scheme as well as complex coordination patterns between different independent roles [GTB+03].

Example: The success of patterns has shown that the interaction scheme rather than the elements of that interaction are the right vehicle for reuse at the conceptual level. As depicted in Figure 1, the full potential for reuse in our example can also only be realized when we consider the whole web of interacting roles and embedded components as a reusable asset.

While the decomposition of components into a set of subcomponents is simply the standard way how the architectural principle of hierarchical decomposition is employed, applying the same principle to services such that a subset of a system with only unconnected pattern roles results in a *service* is not present in the component-based approaches (see Figure 1).

The assignment of roles to architectural components is described in Figure 2 (a). The producer and consumer component both take over the corresponding role.

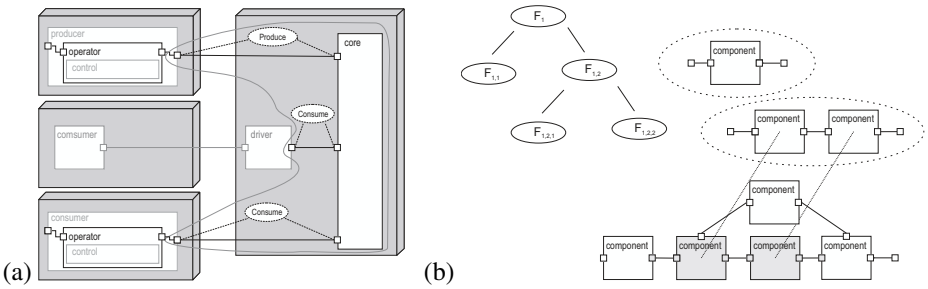


Figure 2: Service-oriented architecture for the software of the energy-management system with operator and controller micro-architecture

The constraints for the components can be derived from the combination of the constraints which must hold for the roles of a service. Usually, the simple logical conjunction of the role constraints is sufficient here. However, when shared resources are described, also

[3]It is to be noted that there exist other definitions of the term service which focus more on their run-time composition. However, in the considered domain of automotive software such a limited view on what constitutes a service is too restrictive.

additive constraints are possible. E.g., if one role requires that the energy usage is below a certain threshold and a second role referring to another function of the same component makes a similar constraint, the sum of both thresholds has to be ensured.

In [Gie08] is outlined how you can systematically and with tool support combine the different roles specified for a service using a particular micro-architecture within the component to derive a correct synchronization between the different roles a component may play for the services it is involved in. The operator part of the component is responsible for the coordination and initiates that the reused parts of the controller are activated as indicated by the role constraints.

Discussion: The benefit of the sketched service-oriented description of the innovative eBalance function is mainly that the service notion permits to capture all required aspects of the function within a single coherent concept, while a pure component-based description would fail to do so. The most obvious benefits of the service-oriented description thus result for the earlier phases of system development where the function can be modeled and discussed using a partial view on the final architecture which focus only on the contained components and involved roles and their constraints.

However, as shown in [Gie08] also later stages can benefit from this concept. In particular, the clear conceptual description leads to the potential of synthesizing the operator part of the component behavior of the component micro-architecture to coordinate the different roles [Gie08, GV06]. This separation into the coordination of the roles and their coordination of the components further allow to employ approaches for the compositional verification of the real-time coordination [GTB+03] and the integrated description and modular verification of discrete behavior and continuous control with components [GBSO04].

As depicted in Figure 2 (b), the services at different levels of abstraction correspond to functions in a functional hierarchy (potentially with variants in the tree) and thus describe potentially overlapping views on parts of the system right like suggested in [GHK+07] in form of views on function nets. As outline in [GV06] the separate development and documentation of such views and their systematic composition is a very helpful application of the separation of concerns principle to reduce the complexity of the overall design task.

4 Conclusion and Future Work

We discussed in the paper the limitations of a component-based approach for automotive software when it comes to integrating new innovative functions which cannot be realized by a single component but require the coordination of multiple sometimes already existing components. A service-oriented approach which focuses on the interaction of multiple roles rather than single components as units for design and reuse was presented and the benefits have been discussed.

We plan to further elaborate the approach in our future work and will in particular look into the problem of integrate legacy component-based elements in the approach.

References

[BKM07] Manfred Broy, Ingolf H. Krüger, and Michael Meisinger. A formal model of services. *ACM Trans. Softw. Eng. Methodol.*, 16(1):5, 2007.

[DMRK05] Martin Deubler, Michael Meisinger, Sabine Rittmann, and Ingolf Krüger. Modeling Crosscutting Services with UML Sequence Diagrams. In Lionel C. Briand and Clay Williams, editors, *Model Driven Engineering Languages and Systems, 8th International Conference, MoDELS 2005, Montego Bay, Jamaica, October 2-7, 2005, Proceedings*, volume 3713 of *Lecture Notes in Computer Science*. Springer, 2005.

[GBSO04] Holger Giese, Sven Burmester, Wilhelm Schäfer, and Oliver Oberschelp. Modular Design and Verification of Component-Based Mechatronic Systems with Online-Reconfiguration. In *Proc. of 12th ACM SIGSOFT Foundations of Software Engineering 2004 (FSE 2004), Newport Beach, USA*, pages 179–188. ACM Press, November 2004.

[GHK+07] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, and Bernhard Rumpe. View-Based Modeling of Function Nets. In Matthias Gehrke, Holger Giese, and Joachim Stroop, editors, *Preliminary Proceedings of the 4th Workshop on Object-oriented Modeling of Embedded Real-Time Systems (OMER4), 30./31. October 2007, Paderborn, Germany*, volume tr-ri-07-286 of *Tech. Rep., Computer Science Department*. University of Paderborn, 2007.

[Gie08] Holger Giese. Reusable Services and Semi-Automatic Service Composition for Automotive Software. In Manfred Broy, Ingolf Krüger, and Michael Meisinger, editors, *Post Workshop Proceedings of the Automotive Software Workshop San Diego 2006*, Lecture Notes in Computer Science (LNCS). Springer Verlag, 2008. (to appear).

[Gri03] Klaus Grimm. Software technology in an automotive company: major challenges. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 498–503, Washington, DC, USA, 2003. IEEE Computer Society.

[GTB+03] Holger Giese, Matthias Tichy, Sven Burmester, Wilhelm Schäfer, and Stephan Flake. Towards the Compositional Verification of Real-Time UML Designs. In *Proc. of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE-11)*, pages 38–47. ACM Press, September 2003.

[GV06] Holger Giese and A. Vilbig. Separation of Non-Orthogonal Concerns in Software Architecture and Design. *Software and System Modeling (SoSyM)*, 5(2):136 – 169, June 2006.

[HKK04] Bernd Hardung, Thorsten Kölzow, and Andreas Krüger. Reuse of software in distributed embedded automotive systems. In *EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software*, pages 203–210, New York, NY, USA, 2004. ACM Press.

[RSS05] Marc Rosenmayr, Matthias Schöllmann, and Rainer Schmidt. Intelligentes Energiemanagement. In *Internationaler CTI-Automobil-Technologie-Kongress AutoTec*, 2005.

[Szy98] Clemens Szyperski. *Component Software, Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[TH02] Steffen Thiel and Andreas Hein. Modeling and Using Product Line Variability in Automotive Systems. *IEEE Software*, 19(4):66–72, July/August 2002.

[WW03] Matthias Weber and Joachim Weisbrod. Requirements Engineering in Automotive Development: Experiences and Challenges. *IEEE Software*, 20(1):16–24, January/February 2003.

Modellgetriebene Softwareentwicklung für eingebettete Systeme auf Basis von OMG Standards

Markus Schmidt
Technische Universität Darmstadt
FG Echtzeitsysteme, D-64283 Darmstadt
markus.schmidt@es.tu-darmstadt.de

Abstract: Die Entwicklung von Software wird zunehmend schwieriger, da die zu realisierenden Systeme immer komplexer werden. Ein aktueller Trend zur Beherrschung umfangreicher Softwaresysteme ist die modellgetriebene Softwareentwicklung. Neben einigen proprietären Modellierungssprachen, haben vor allem die Sprachen der OMG weite Verbreitung gefunden. Zahlreiche Werkzeuge ermöglichen die Erstellung von UML Modellen und somit zumindest eine übersichtliche Dokumentation. Die Erzeugung von ausführbaren Applikationen aus diesen Modellen ist, wenn überhaupt, nur sehr eingeschränkt möglich. Typischerweise werden nur Strukturinformationen aus Klassendiagrammen verwendet und daraus Java oder C++ Code erzeugt. Im Bereich der eingebetteten Systeme können diese Teillösungen nicht verwendet werden, da dort hardwarenahe Programmiersprachen wie C eingesetzt werden müssen. In dieser Arbeit wird vorgestellt, wie auf Basis von OMG Standards eine DSL für eingebettete Systeme erstellt werden kann, deren Modelle in ausführbare Applikationen überfuhrt werden können. Es wird aufgezeigt, in welchen Bereichen die OMG Standards direkt verwendet werden können, und wann eigene Lösungen verwendet werden müssen. Die Praxistauglichkeit des beschriebenen Vorgehens wird anhand des Betriebssystems OSEK-OS und des Werkzeugs Visual Paradigm demonstriert.

Schlüsselwörter: MDD, DSL, UML, ausführbare Modelle, OSEK

1 Einleitung

Die klassische Softwareentwicklung ist geprägt von der Erstellung von Textdateien, wobei die Texte in der Syntax einer Programmiersprache erstellt sind. In den letzten Jahren wurden zwar die Programmiersprachen immer mächtiger und erreichten so einen immer höheren Abstraktionsgrad, aber die grundlegende Darstellungsform blieb gleich.

Weil die zu realisierenden Systeme immer komplexer werden, ist man auf der Suche nach besseren Entwicklungsmethoden. Ein aktueller Trend ist dabei die modellgetriebene Softwareentwicklung. Visuelle Modelle bieten umfangreiche Notationen und einen höheren Abstraktiongrad an und sollen so die Entwicklung erleichtern. Besonders im Bereich der objektorientierten Programmierung wird die modellgetriebene Softwareentwicklung eingesetzt, da es einfache Abbildungen von visuellen Modellelementen auf Elemente der Programmiersprache (z.B. `C++`) gibt.

Neben einigen proprietären Modellierungssprachen, haben vor allem die Sprachen der OMG weite Verbreitung gefunden. Zahlreiche Werkzeuge ermöglichen die Erstellung von UML Modellen und sogar die Erzeugung von Codefragmenten für Java oder C++.

Diese Teillösungen der modellgetriebene Entwicklung können im Bereich der eingebetteten Systeme nicht direkt verwendet werden, da dort hardwarenahe Programmiersprachen wie C eingesetzt werden müssen. Trotzdem können die verfügbaren Werkzeuge und Standards benutzt werden, um mit sinnvoller Kombination und zusätzlicher Eigenentwicklung eine modellgetriebene Softwareentwicklung zu erzielen. Dafür wird in dieser Arbeit ein Vorgehen beschrieben, mit dem auf Basis von OMG Standards eine DSL für eingebettete Systeme erstellt werden kann, wobei aus Modellen der DSL eine ausführbare Applikationen erzeugt werden kann.

Der Einsatz der Modellierungssprache UML zur Entwicklung von eingebetteten Systemen wurden schon häufig untersucht. Die grundlegenden Vorteile von UML 2 wurden von Selic [Sel03] dargestellt. Ein konkreter Ansatz zur Modellierung von Verhalten ist in [JHB06] aufgeführt. Das Verhalten wird mit Interaktionsdiagrammen beschrieben, wodurch eine automatische Erzeugung von Code möglich sein soll. Wie diese Erzeugung funktioniert, ist leider nicht näher beschrieben. Die in diesem Papier als Beispiel verwendete Domäne OSEK, wurde bisher nur wenig betrachtet. In den Arbeiten von Moore [Moo02] und Zonghua Gu et al. [GWS03], wird eine DSL für OSEK durch ein UML Profil definiert. Beide Arbeiten thematisieren nur die Modellierung von OSEK Applikationen und enthalten keine Hinweise zur Erzeugung der ausführbaren Applikation.

Das nachfolgende Kapitel beschreibt das Vorgehen schematisch und führt die wichtigsten Schritte sowie die zu verwendenden Werkzeuge auf. Anschließend wird das Vorgehen bei der Erstellung einer DSL für das Betriebssystem OSEK-OS genauer vorgestellt. Dazu wird in Kapitel 3 das Metamodell schrittweise aufgebaut. Die Definition der konkreten Syntax der DSL und damit die Verwendung von UML erfolgt im Kapitel 4. Die Erzeugung der ausführbaren Applikation und die dafür notwendige Erzeugung von Textdateien wird in Kapitel 5 behandelt. Dort wird auch die Realisierung mit dem Werkzeug Visual Paradigm angesprochen. Kapitel 6 faßt diese Arbeit zusammen und bietet einen Ausblick auf weiterführende Aktivitäten.

2 Entwicklung einer ausführbaren DSL

Dieses Kapitel beschreibt ein Vorgehen zur Entwicklung einer Domänenspezifischen Sprache (DSL) auf Basis von Standards der Object Management Group (OMG). Die Bezeichnung *ausführbare DSL* soll dabei zum Ausdruck bringen, dass Modelle dieser DSL nicht nur zur Dokumentation dienen, sondern aus diesen auch eine ausführbare Applikation erzeugt werden kann.

Es wird aufgezeigt, in welchen Bereichen diese Standards verwendet werden können, und wann eigene Lösungen verwendet werden müssen. Die beiden wichtigsten Standards für das nachfolgend beschriebene Vorgehen sind

- Meta Object Facility (MOF) [OMG03]
- Unified Modeling Language (UML) [OMG07]

MOF und UML sind beides grafische Modellierungssprachen, die aber auf unterschiedlichen Ebenen angesiedelt sind. MOF dient zur Beschreibung von Modellierungssprachen, während UML eine Modellierungssprache ist. Dabei ist UML weit mehr als eine Instanz von MOF, da für UML eine zusätzliche konkrete Syntax definiert ist.

2.1 Erstellung der DSL

Das primäre Ziel bei der Entwicklung einer DSL ist die Definition einer Darstellung, die eine übersichtliche und möglichst vollständige Beschreibung eines Systems ermöglicht. Diese Beschreibung ist dabei ein grafisches Modell, welches von dem System abstrahiert, aber (fast) alle Bereiche des Systems darstellen kann. Ein weiteres Ziel ist oft die Ausführbarkeit der Modelle, d.h. aus einer grafischen Beschreibung kann automatisch eine ausführbare Datei erzeugt werden.

Im Bereich der eingebetteten Systeme kann man dies am Übergang von der Programmierung in Maschinensprache zur Programmierung in C gut erkennen. Die Programmiersprache C abstrahiert von dem unterliegenden Prozessor und dessen Maschinensprache. Beispielsweise ist in C die Deklaration von Unterprogrammen und deren Aufruf sehr einfach zu beschreiben. In einem äquivalenten Programm in Maschinensprache müsste die Übergabe der Parameter in Registern oder den Stack erfolgen, wobei jeweils der Datentyp eines Parameters beachtet werden muss.

Die Erstellung einer DSL ist in Abbildung 1 schematisch dargestellt.

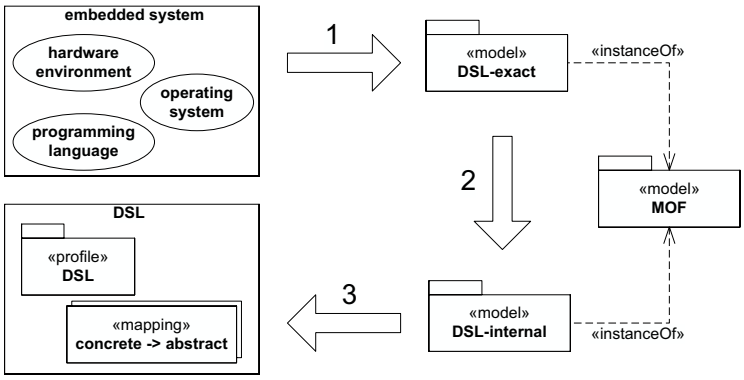


Abbildung 1: Erstellung einer DSL auf Basis von MOF und UML

Ausgangspunkt ist das eingebettete System, welches in die Bereiche Hardware, Programmiersprache und Betriebssystem aufgeteilt ist. Jede Ausprägung dieses Systems wird in der klassischen Softwareentwicklung durch eine Menge von Tetxdateien beschrieben und

Übersetzer erzeugen daraus die ausführbaren Applikation. Aus den Informationen über das eingebettete System wird ein exaktes Metamodel (**DSL-exact**) erstellt. Somit kann zu jeder Ausprägung des eingebetteten Systems eine Instanz des Metamodels erstellt werden, die diese Ausprägung vollständig beschreibt. Die Mächtigkeit des Metamodells ist aber gleichzeitig auch ein Nachteil für die Anwendbarkeit. Da das Metamodell eine direkt Abbildung von dem eingebetteten System ist, sind die Instanzen des Metamodells ähnlich kompex wie die klassische textuelle Beschreibung.

Daher wird in einem zweiten Schritt aus dem exakten Metamodell **DSL-exact** ein vereinfachtes Metamodel **DSL-internal** erstellt. In diesem vereinfachten Metamodel sind bestimmte Kombinationen aus dem ursprünglichen Metamodell eingeschränkt oder nicht mehr möglich. Weiterhin können bestimmte Kombinationen die in den meisten Ausprägungen des eingebetteten Systems vorkommen als separate Metaklasse modelliert sein. Dadurch sind Instanzen von **DSL-internal** kleiner und kompakter als Instanzen von **DSL-exact**. Die Notationsvielfalt ist aber noch immer eingeschränkt, da Instanzen von Metamodellen nur aus Blöcken (Instanzen von Klassen) und Linien (Instanzen von Assoziationen) bestehen.

Deshalb wird im dritten Schritt eine Modellierungssprache verwendet die selbst eine Instanz von MOF ist und zusätzlich noch über eine umfangreiche Menge von Notationselementen verfügt. Diese Sprache ist UML und ermöglicht durch die Verwendung von Profilen eine Anpassung an eine bestimmte Domäne. Zusätzlich wird noch festgelegt, wie Notationselemente von UML im Metamodell **DSL-internal** repräsentiert werden. Damit wird die Abbildung von der konkreten Syntax (UML) auf die abstrakte Syntax (**DSL-internal**) definiert. Nach dem dritten Schritt hat man somit eine DSL die eine interne Darstellung in Form von **DSL-internal** hat und für den Anwender als spezieller UML Dialekt zur Verfügung steht.

2.2 Erzeugung einer ausführbaren Applikation

Mit den Überlegungen aus dem vorherigen Abschnitt sind wir in der Lage ein eingebettetes System mit der erstellten DSL in einem UML Werkzeug zu modellieren. Damit haben wir zunächst nur eine übersichtliche Darstellung des Systems. Wie bei der textuellen Softwareentwicklung, möchte man aus dieser Beschreibung auch die ausführbare Applikation erzeugen. Wie dies realisiert werden kann, ist in Abbildung 2 schematisch dargestellt.

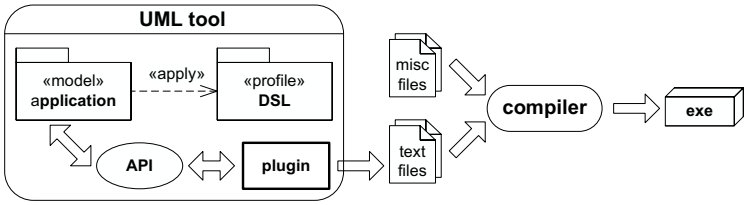


Abbildung 2: Erzeugung einer ausführbaren Applikation

Generell soll die Realisierung mit einem geringen Anteil von Eigenentwicklungen auskommen und möglichst viele vorhandene Werkzeuge verwenden. Vorhandene Werkzeug sind dabei der Übersetzer, der aus einer Menge von Textdateien (z.B. C Quelltexte) die ausführbare Applikation erzeugt und das UML Werkzeug, welches als grafischer Editor dient.

Das Modell wird in einem beliebigen UML Werkzeug erstellt, wobei zur Erstellung das Profil **DSL** verwendet wird. Zwar sind einige UML Werkzeuge in der Lage aus Modellen Textdateien zu erzeugen, aber dies ist zum einem auf bestimmte Modelltypen (meistens Klassendiagramme) beschränkt und zu anderen sind die Textdateien dann Quelltext für Objektorientierte Programmiersprache wie C++ oder Java.

Daher wird ein Programm benötigt, welches aus den Elementen des Modells in geeigneter Weise eine Menge von Textdateien erzeugt, die dem vorhandenen Übersetzer als Eingabe dient. Die gängigen UML Werkzeuge ermöglichen dies durch die Integration eines Plugin in das Werkzeug. Dieses Plugin greift auf die Elemente des Modells über eine spezifische API zu und erzeugt aus diesen Informationen die notwendigen Textdateien. Somit ist das Plugin der einzige Teil, der selbst entwickelt werden muss.

Kapitel 5 enthält weitere Information zur Realisierung dieser Eigenentwicklung. Dort wird auch eine existierende Realisierung für das Werkzeug Visual Paradigm besprochen.

3 Erstellung des Metamodells einer DSL

Um die Ausführungen des vorhergehenden Kapitels zu verdeutlichen, wird in diesem Kapitel eine DSL für ein konkretes Beispiel erstellt. Das Beispiel ist das Betriebssystem OSEK-OS [OSE05], welches im Bereich der eingebetteten System verwendet wird. OSEK-OS bietet ein Menge von Betriebssystemobjekten (z.B. Task, Resource, Event) an und Funktionen die auf diesen operieren. Die verwendete Programmiersprache ist C und zur Erstellung einer ausführbare Applikation wird neben den C Quelltexten auch eine texuelle Strukturbeschreibung (OIL-Datei [OSE04]) benötigt.

3.1 Ein exaktes Metamodell

Da eine ausführliche Behandlung von OSEK-OS den Umfang dieses Papiers sprengen würde, wird hier nur ein kleiner Teil behandelt. Von der Gesamtmenge der Betriebssystemobjekte werden nur Task, Resource und Event und deren Funktionen verwendet. Weiterhin werden wir bei diesen Objekten nicht alle möglichen Attribute verwenden. Das Metamodell dieser Teilmenge von OSEK-OS ist in Abbildung 3 dargestellt.

Die vier Klassen im linken Bereich der Abbildungen repräsentieren dabei die entsprechenden Einträge in der OIL-Datei, wobei einige Attribute (z.B. Task.schedule) weggelassen wurden. Der rechte Bereich beschreibt das Verhalten in Form von C-Code (*C-Behavior*) und Aufrufen von OSEK Funktionen (*OSEK-Service*). Wichtig ist dabei, dass das Asso-

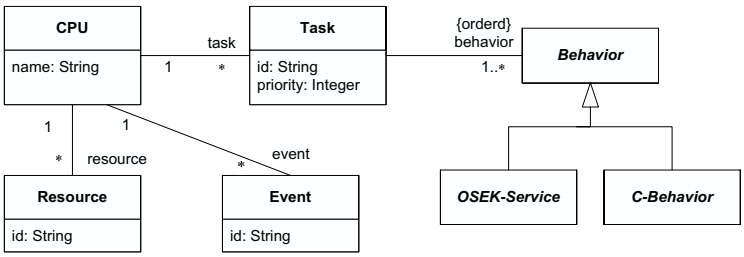


Abbildung 3: Metamodell einer OSEK Applikation

ziationsende behavior geordnet ist. Dadurch ist das gesamte Verhalten eines Task, durch eine geordnete Folge von einzelnen Verhaltensspezifikationen (*Behavior*) definiert.

Die konkreten Klassen, die Bestandteile eines C Programms repräsentieren (z.B. Zuweisungen oder Abfragen), werden hier nicht weiter dargestellt. Stattdessen betrachten wir die Aufruf von OSEK Funktionen genauer. In Abbildung 4 sind Funktionen modelliert, die auf den erwähnten Objekten (Task, Resource, Event) von OSEK arbeiten.

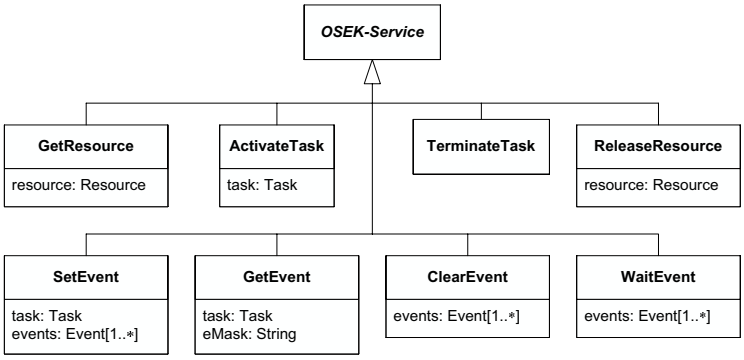


Abbildung 4: Metamodell einiger OSEK Systemfunktionen

Alle konkreten Klassen dieser Abbildung repräsentieren die gleichnamige Funktion, wobei Parameter der Funktion als Attribute der Klasse modelliert sind.

Damit haben wir ein einfaches Metamodell einer OSEK Applikation erstellt und können so Applikationen als Modelle darstellen. Modelle sind dabei Instanzspezifikationen des Metamodells und bestehen daher hauptsächlich aus Blöcken und Linien. Eine Applikation ist in Abbildung 5 dargestellt.

Da es bisher keine sinnvolle Notation von Instanzen geordneter Assoziationsenden gibt, wurde dabei die Position als Zahl auf dem Link notiert. Die Applikation besteht aus den beiden Tasks `T1` und `T2` mit den Prioritäten 10 und 20. Task `T1` wartet auf das Event `E1` und beendet sich danach selbst. Task `T2` schickt zuerst das Event `E1` an `T1`, um anschließend auf `RES1` zuzugreifen. Diese Applikation ist noch einmal in textueller Form in Abbildung 6 angegeben.

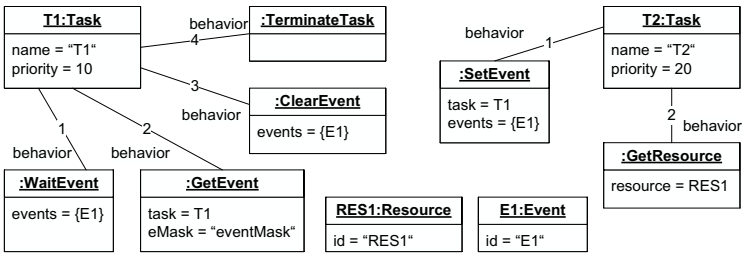


Abbildung 5: OSEK Applikationen als Instanzspezifikation

Vergleicht man die beiden Darstellungen miteinander, erscheint die textuelle Darstellung doch deutlich übersichtlicher und verständlicher. Eine Darstellung als Modell ist also nicht immer besser als eine textuelle Darstellung, vielmehr kommt es auch auf die zur Verfügung stehenden grafischen Elemente an.

```
TASK T1 {
  PRIORITY = 10;
  EVENT = E1;
};

TASK T2 {
  PRIORITY = 20;
  RESOURCE = RES1;
};
```

```
TASK (T1) {
  EventMaskType eventMask;

  WaitEvent(E1);
  GetEvent(T1, &eventMask);
  ClearEvent(E1);

  TerminateTask();
}
```

```
TASK (T2) {
  SetEvent(T1, E1);

  GetResource(RES1);
}
```

Abbildung 6: OSEK Applikation in klassischer Darstellung

Das Problem der unzureichenden grafischen Darstellung wird im nächsten Kapitel behandelt. Ein Problem das beide Darstellungen haben bezieht sich auf Task `T2`. Die Ausführung von `T2` endet in einem Laufzeitfehler. Zum einen wird der Task nicht wie von OSEK vorgeschrieben beendet (**TerminateTask** oder **ChainTask**) und zum anderen wird die angeforderte Ressource nicht wieder freigegeben.

Es wäre gut, wenn solche Fehler schon vor dem Starten der Applikation erkannt werden. Bei der textuellen Entwicklung kann eine Quelltextanalyse solche Überprüfungen vornehmen. Bei der Erstellung der DSL können wir aber noch einen Schritt weitergehen und die DSL so aufbauen, dass einige Fehler nicht modelliert werden können. Weiterhin könnten in der DSL gängige Kombinationen als ein untrennbares Teil modelliert werden. Ein neues Metamodell für OSEK Funktionen ist in Abbildung 7 dargestellt.

Nur die Klasse **SendEvent** wurde unverändert übernommen. Der Zugriff auf eine Ressource wird jetzt nur noch durch die Klasse **ResourceBlock** repräsentiert. Damit wird ein Block repräsentiert der am Anfang die Ressource blockiert und am Ende diese wieder freigibt. Innerhalb dieses Blocks kann beliebiges Verhalten stehen. Aktivierung und Terminierung eines Task werden durch die Klasse **TaskService** dargestellt, wobei jetzt das Attribut task optional ist. Ist das Attribut nicht angegeben handelt es sich um eine Terminierung, ansonsten ist es eine Aktivierung. Der Empfang von Events wird nur noch durch die Klasse **ReceiveEvent** repräsentiert und ist somit eine Kombination aus den Klassen **WaitEvent**,

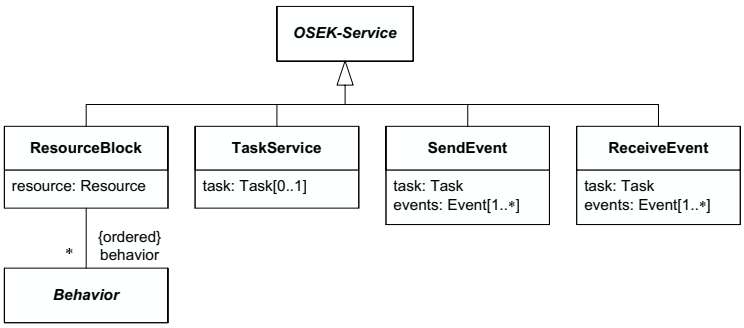


Abbildung 7: vereinfachtes Metamodell einiger OSEK Systemfunktionen

GetEvent und **ClearEvent** des exakten Metamodells. Der Übergang von Abbildung 5 auf Abbildung 7 entspricht somit dem Übergang von **DSL-exact** auf **DSL-internal**. Der Task `T1` ist als Instanz des vereinfachten Metamodells in Abbildung 8 dargestellt.

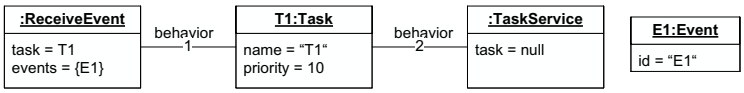


Abbildung 8: Ein Task als Instanz des vereinfachten Metamodells

4 Definition der konkreten Syntax der DSL

Wichtig für den Anwender einer DSL ist eine übersichtliche und reichhaltige Menge von Notationselementen, die einen Bezug zur dargestellten Domäne haben. Mit dem entwickelten Metamodell aus Kapitel 3 lassen sich zwar auch Modelle darstellen, aber die Notation ist doch stark eingeschränkt da nur mit Instanzspezifikationen des Metamodells gearbeitet werden kann. Es fehlt demnach eine Notation, die von der Instanzspezifikation auf dem Metamodell abstrahiert.

Prinzipiell kann dies auf zwei verschiedenen Wegen erreicht werden. Durch die Definition einer vollkommen eigenen Notation erreicht man die optimale grafische Darstellung der betrachteten Domäne. Dadurch ist man aber gezwungen einen eigenen Editor zu entwickeln und dies kann eine relativ zeitaufwendige Arbeit sein. Ein anderer Weg ist, eine vorhandene Menge von Notationselementen als Basis zu nehmen, und daraus bestimmte Element auf die Domäne abzubilden. Dadurch lassen sich bestehende Editoren verwenden, aber die grafische Abbildung auf die Domäne ist nicht immer optimal.

Im Folgenden wird hier der zweite Weg beschritten und UML2 als Basis für die konkrete Syntax gewählt. Dabei wird der vorhandene Erweiterungsmechanismus von UML2 verwendet und mit Hilfe von Profilen die konkrete Syntax der DSL definiert.

4.1 Konkrete Syntax für Strukturelemente

Der bisher besprochen Teil von OSEK-OS enthält als Strukturelemente nur Task, Resource und Event. Da ein Task auch Verhalten besitzt (siehe Abbildung 3), lässt er sich gut als Klasse darstellen. Eine Klasse in UML kann neben den strukturellen Elementen wie Attribute oder Operationen auch Verhaltensspezifikationen besitzen. Dabei kann die gesamte Klasse eine Verhaltensspezifikationen besitzen und auch das Verhalten einer einzeln Operation separat spezifiziert sein.

Ressourcen und Events haben kein Verhalten und außer dem Namen auch keine weiteren Eigenschaften. Weiterhin kann aus dem Verhalten aller Tasks, die Menge der verwendeten Ressourcen und Events ermittelt werden. Daher brauchen Ressourcen und Events nicht explizit als Strukturelemente modelliert werden, weil die notwendigen Informationen in den Verhaltensspezifikationen enthalten sind. Trotzdem werden Ressourcen explizit als Strukturelemente modelliert, weil dadurch die Funktionalität im Vergleich zum OSEK Standard erweitert wird. Nach dem OSEK Standard ist eine Ressource lediglich ein binäres Flag, welches einen exklusiven Zugriff auf einen bestimmten Bereich (z.B. gemeinsamer Speicherbereich) ermöglicht. Eine direkte Beziehung zwischen der Ressource und dem zu schützenden Bereich gibt es nicht. Der Programmierer muss jeden Zugriff auf diesen Bereich separat durch die Benutzung der Ressource absichern. Wird eine Ressource als Klasse modelliert, ist dadurch auch der zu schützende Bereich ein Bestandteil der Ressource. Die Attribute der Klasse repräsentieren dabei die Daten des zu schützenden Bereichs und die Operationen die Zugriffsmethoden auf diese. Ein Beispiel dazu ist in Abbildung 11 angegeben.

Tasks und Ressourcen werden somit explizit als Klassen modelliert und Informationen zu den Events werden aus den Verhaltensspezifiaktionen ermittelt. Damit diese Festlegung in einem UML Werkzeug verwendet werden kann, wird ein Profil erstellt, welches die Festlegung formal darstellt. In Abbildung 9 ist diese Profil angegeben.

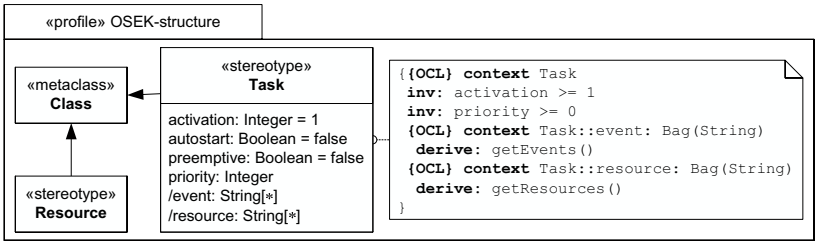


Abbildung 9: Profil für Strukturelemente von OSEK-OS

Die OSEK Objekte Task und Resource werden durch Klassen modelliert und sind daher als Stereotypen angegeben, die die Metaklasse **Class** erweitern. Der Stereotype **Task** enthält weitere Attribute, die den Einträgen in der OIL-Datei entsprechen. Die beiden letzten Attribute sind abgeleitete Attribute, d.h. der Wert wird nicht angegeben sondern berechnet. Dadurch wird die Menge der empfangenen Events, sowie die Menge der verwendeten Ressourcen aus dem Verhalten ermittelt.

4.2 Konkrete Syntax für Verhaltensbeschreibung

Die Verhaltensbeschreibung der hier verwendeten Domän besteht aus C Programmen, die spezielle Funktionen von OSEK aufrufen können. Daher werden Notationselemente benötigt, die die üblichen Konstrukte einer imperativen Programmiersprache (Abfragen, Schleifen, usw.) darstellen, sowie spezielle Funktionen von OSEK in geeigneter Form repräsentieren.

In UML lässt sich Verhalten durch Verwendung der folgenden Diagrammarten beschreiben.

- Zustandsdiagramme
- Aktivitätsdiagramme
- Interaktionsdiagramme (vier verschiedene Diagrammarten)

Zustandsdiagramme sind sehr gut geeignet das Verhalten in Form von Zuständen und den Übergangen zwischen diesen zu beschreiben. Zur Beschreibung von beliebigen C Programmen sind sie aber ungeeignet, da die Darstellung des Kontrollflusses stark eingeschränkt ist. Aktivitätsdiagramme bieten einen viel allgemeineren Weg Verhalten zu beschreiben, wobei sowohl der Datenfluss als auch der Kontrollfluss beschrieben werden kann. Auch für das Senden und Empfangen existieren eigene Notationen, die aber leider zu restriktiv sind. So kann beim Senden nur das Signal angegeben werden, aber nicht der Empfänger.

UML2 bietet vier verschiedene Arten von Interaktionsdiagrammen an. Dies sind Sequenz-, Kommunikations-, Timing- und Interaktionsübersichtsdiagramm. Diese vier Diagramme basieren alle auf dem gleichen Konzept der *Interaktion*, erlauben aber unterschiedliche Sichten auf bestimmte Details. Das mächtigste dieser Diagramme ist das Sequenzdiagramm. Es enthält alle wichtigen Interaktionskonzepte, sowie Elemente zur Darstellung von Schleifen und Abfragen. Daher werden Sequenzdiagramme als konkrete Syntax der Verhaltensspezifikation verwendet.

Aus Platzgründen wird hier nicht die vollständige Abbildung dargestellt, sondern die wesentlichen Punkte an einem Beispiel erläutert. In Abbildung 10 ist das Verhalten von Task **T1** als Sequenzdiagramm und als C Funktion angegeben. Im linken Bereich ist das Verhalten als Sequenzdiagramm dargestellt und die äquivalente C Funktion befindet sich im rechten Bereich.

Die Teilnehmer der Interaktion sind als Lebenslinien dargestellt, wobei der Stereotype den genauen Typ kennzeichnet. Das Senden und Empfangen von Events wird durch eine asynchrone Nachricht mit dem Namen *Events* dargestellt und die einzelnen Events sind die Argument dieser Nachricht. Der exklusive Zugriff auf geschütze Bereiche darf nur in einem kritischen Block (**critical** Fragment) erfolgen, wobei der Aufruf der Operationen durch synchrone Methoden dargestellt wird. Das Aktivieren eines Tasks wird durch einen Erzeugungsnachricht dargestellt und die Terminierung durch die Destruktion der Lebenslinie (großes Kreuz). Die Sequenzdiagramme bieten weitere Fragment, die gängige Kon-

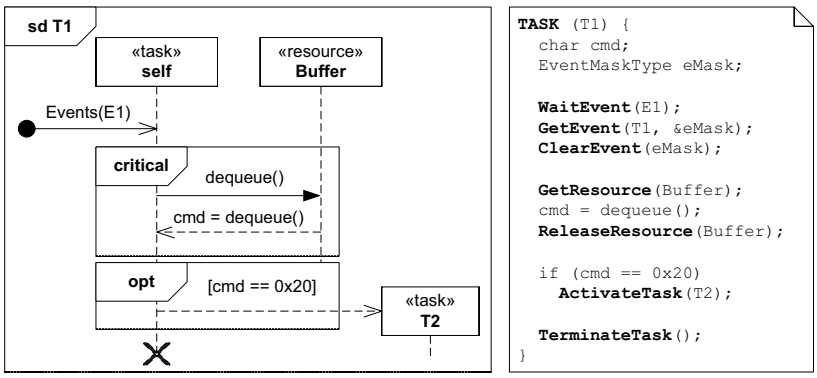


Abbildung 10: Verhalten von Task **T1** (Sequenzdiagramm und C Code)

strukte von Programmiersprachen darstellen. Als Beispiel ist in der Abbildung 10 das **opt** Fragment angegeben, welches einer einfachen Abfrage entspricht.

4.3 Erweiterungen der konkreten Syntax

Im vorherigen Abschnitt wurden den Elementen des Metamodells **DSL-internal** Notationselemente von UML zugeordnet und damit die konkrete Syntax festgelegt. Selbst wenn damit allen Elementen des Metamodells eine konkrete Notation zugeordnet wurde, muss die Verwendung von UML damit nicht abgeschlossen sein. Die noch nicht verwendeten Notationselemente können zur Erweiterung der konkreten Syntax dienen ohne das Veränderungen am Metamodell nötig sind. Sie bilden damit eine weitere Sicht auf das unterliegende Metamodell.

Ein naheliegendes Beispiel ist die Verwendung von abstrakten Klassen oder abstrakten Operationen in Kombination mit Vererbung. Da ein Task als Klasse modelliert wird, kann dadurch ein abstrakter Task modelliert werden. So kann eine Hierarchie von Tasks modelliert werden, deren gemeinsames Verhalten nur einmal modelliert ist. Am Ende dieser Hierarchie stehen konkrete Tasks, für die der entsprechende Code erzeugt werden kann.

Ein weiteres Beispiel betrifft die Parametrisierung, welches im Folgenden auch grafisch dargestellt wird. Durch parametrisierte Klassen wird das Konzept des Parameters, auf die Ebene der Klassen transportiert. Die Definition eines Ringpuffers in Abbildung 11, zeigt eine Anwendung dieses Konzepts.

Eine Ressource **FIFORing** ist mit den Daten und den Zugriffsmethoden angegeben. Das Verhalten der beiden Zugriffsmethoden ist durch die Sequenzdiagramme angegeben. Da solche Ringpuffer häufig verwendet werden und sich nur in der Größe und dem Datentyp unterscheiden, wäre es gut, wenn diese Eigenschaften durch Parameter dargestellt werden können. Dies ist in der Abbildung geschehen, wobei T der Parameter für den Datentyp ist und S der Parameter für die Größe.

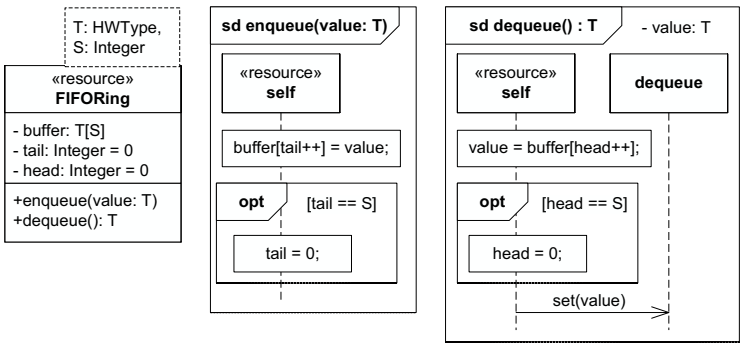


Abbildung 11: Ringpuffer mit Parameter für Datentyp und Größe

Abbildung 12 zeigt, wie die parametrisierbare Ressource verwendet wird. Die Ressourcen **ByteQueue** und **WordQueue** setzen die Parameter auf die angegebenen Werte und definieren so einen Ringpuffer mit maximal 32 Bytes und einen Ringpuffer mit maximal 16 Wörtern.

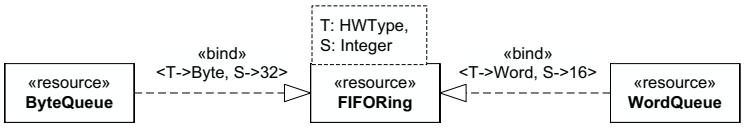


Abbildung 12: Ringpuffer für unterschiedliche Datentypen

Am Anfang dieses Kapitels wurden die Interaktionsdiagramme zur Darstellung des Verhaltens ausgewählt. Die Sequenzdiagramme dienten dann zur Darstellung des Verhaltens von genau einem Task. Dadurch kann mit einer Menge von Sequenzdiagrammen das Verhalten einer gesamten Applikation beschrieben werden. Eine andere Sicht auf das Verhalten einer Applikation, kann durch Verwendung eines anderen Interaktionsdiagramms dargestellt werden. Einen Überblick über die Kommunikation ermöglichen die Kommunikationsdiagramme, die vom exakten Verhalten der einzelnen Bestandteile abstrahieren. Die Kommunikation von drei Tasks ist in Abbildung 13 dargestellt.

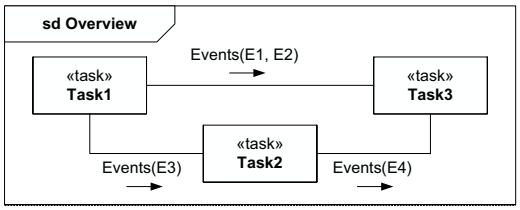


Abbildung 13: Überblick der Kommunikation von drei Tasks

Drei Tasks kommunizieren miteinander, wobei **Task1** nur sendet und **Task3** nur empfängt.

Dieses Diagramm kann zum einen zur Dokumentation dienen und aus den einzelnen Sequenzdiagrammen erzeugt werden (Modelltransformation). Zum anderen kann es vor der Erstellung der einzelnen Sequenzdiagramme während eines Design Prozesses entwickelt worden sein und dient dann zur Überprüfung der einzelnen Sequenzdiagramme.

5 Vom Modell zur ausführbaren Applikation

In diesem Kapitel wird beschrieben, wie aus einem Modell eine ausführbare Applikation erzeugt werden kann. Dabei sollen möglichst viele vorhandene Werkzeuge verwendet werden und der Anteil der Eigenentwicklung gering bleiben.

Die beiden wichtigste Arten von Werkzeuge sind der Editor zum Erstellen des Modells und der Übersetzer zum Erzeugen der ausführbaren Applikation. Beide Werkzeuge werden als vorhanden und funktionsfähig vorausgesetzt und es fehlt somit nur eine Schnittstelle zwischen diesen Werkzeuge. Diese Schnittstelle muss als Plugin eines UML Werkzeugs selbst entwickelt werden.

5.1 Modellierungswerkzeuge

Werkzeuge, mit denen UML Modelle erstellt werden können, gibt es sowohl in kommerziellen Versionen als auch in freien Versionen in ausreichendem Maße. Um das beschriebene Vorgehen realisieren zu können, muss ein Werkzeug eine API für den Zugriff auf das Modell haben und einen Plugin-Mechanismus zur Verfügung stellen. Von den existierenden Werkzeugen, die diese minimalen Anforderungen erfüllen, wurden die folgenden Werkzeuge näher untersucht.

Werkzeug	Version	Plattform	API	Sequenz
Enterprise Architect [EAr07]	7.0	Windows	Java, COM	1.x
MagicDraw [MDr08]	15.0	Java	Java	2.x
Visual Paradigm [VP007]	6.1	Java	Java	2.x

Abbildung 14: Eigenschaften von UML 2 Werkzeugen

Obwohl Enterprise Architect eine Windows Applikation ist, gibt es seit Version 6.5 eine Java API. Somit kann das Plugin in jedem Fall in Java programmiert werden und Werkzeug unabhängige Bereiche wie die Implementierung des Metamodells, könnten mit jedem Werkzeug verwendet werden.

Alle drei Werkzeuge sollen laut Datenblatt die Version 2.0 von UML unterstützen. Die besonders wichtige Unterstützung von neuen Elementen in Sequenzdiagrammen (z.B. Fragmente) ist in Enterprise Architect leider nicht vorhanden. Dort muss man sich mit dem Elementen von UML 1.x begnügen. MagicDraw bietet zwar Fragmente an, aber es fehlen leider Nachrichten ohne Absender. Dadurch kann der Empfang von Events nicht model-

liert werden. Aus diesen Gründen wurde Visual Paradigm zur Realisierung verwendet.

5.2 Vom Modell zum Text

Wie im vorherigen Abschnitt erwähnt, ist die Texterzeugung als Plugin von Visual Paradigm in Java realisiert. Die Realisierung umfasst dabei die Implementierung des Metamodells, das Lesen von Informationen aus dem Modell und das Schreiben von Textdateien. Der Ablauf ist dabei wie folgt aufgeteilt.

1. Erstellung einer Instanz des Metamodells aus Struktur und Verhalten des Modells
2. Berechnung abgeleiteter Eigenschaften (z.B. Events aus Verhaltensspezifikation)
3. Erzeugung der Strukturbeschreibung als OIL-Datei
4. Erzeugung der Verhaltensbeschreibung als C-Dateien

Interessant ist dabei die Implementierung des Metamodells aus Abbildung 7, da sie unabhängig vom verwendeten Werkzeug ist. Ein kleiner Ausschnitt der Implementierung ist als Klassendiagramm in Abbildung 15 angegeben.

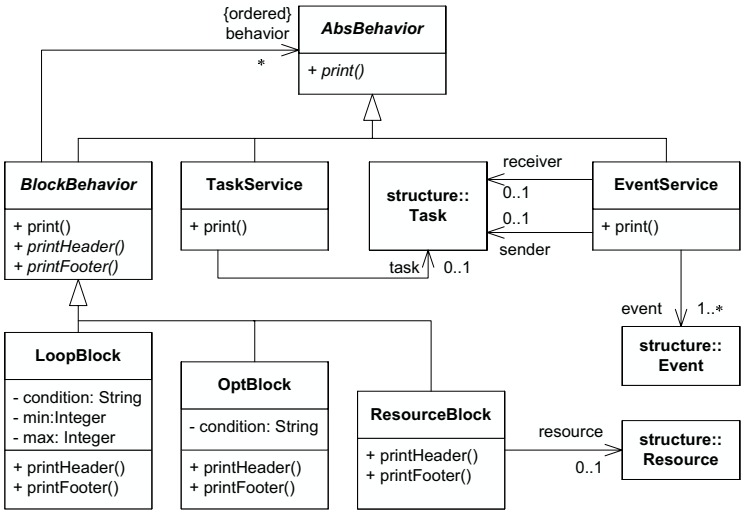


Abbildung 15: Teil der internen Repräsentation des Verhaltens

Die wesentlichen Unterschiede zum Metamodell sind die Methoden zur Ausgabe (print, printHeader, printFooter) und zwei zusätzliche Klassen, die Konstrukte der Programmiersprache C repräsentieren. Hierbei repräsentiert die Klasse **LoopBlock** eine Schleife und **OptBlock** eine Abfrage.

Die Klasse **AbsBehavior** ist die Oberklasse jeder Verhaltensklasse. Die Methode print dient zur Ausgabe des beschriebenen Verhaltens nach C oder OSEK Standard. Zwei konkrete Unterklasse die diese Methode implementieren sind **TaskService** und **EventService**. Neben diesen Einzelverhalten, gibt es auch Klassen die Verhalten als Block gruppieren. Dafür gibt es die abstrakte Oberklasse **BlockBehavior** mit den abstrakten Methoden printHeader und printFooter. In dieser Klasse ist die Methode print so implementiert, das zuerst printHeader aufgerufen wird, danach wird jedes eingebettete Verhalten (Assoziationsende behavior) mit print ausgegeben und am Ende wird die Methode printFooter aufgerufen. Somit wird durch den Aufruf von print das Verhalten des gesamten Blocks ausgegeben. Als Beispiel betrachten wir kurz die Klasse **RessourceBlock**, die die Verwendung von Ressourcen repräsentiert. In dieser Klasse gibt printHeader den Funktionsaufruf zum blockieren einer Ressource aus (`GetResource`) und printFooter den Funktionsaufruf zur Freigabe der Ressource (`ReleaseResource`). Beide Aufrufe enthalten dabei den Namen der Ressource als Argument.

6 Zusammenfassung und Ausblick

In dieser Arbeit wurde beschrieben, wie auf der Basis von MOF und UML eine DSL für eingebettete Systeme entwickelt werden kann und wie aus einem Modell dieser DSL eine ausführbare Applikation erzeugt werden kann.

Der beschriebene Ansatz unterteilt den gesamten Prozess in die Bereiche Erstellung des Metamodells, Definition der konkreten Syntax, und Modell-zu-Text Transformation.

Zur Erstellung des Metamodells sind zwei Schritte notwendig. Dabei erfolgt zuerst eine direkte Abbildung von Elementen des zu beschreibenden Systems auf ein Metamodell. Anschließend wird aus diesem exakten Metamodell ein vereinfachtes Metamodell erstellt, welches bestimmte Kombinationen als unteilbares Ganzes zusammenfasst.

Die Definition der konkreten Syntax erfolgte mit Hilfe der Notationselemente von UML. Neben einer besseren optischen Darstellung von Modellen, erhält man durch die Verwendung von UML auch neue Modellierungsmöglichkeiten. So lassen sich beispielsweise abstrakte Task oder parametrisierbare Ressource modellieren, obwohl es diese in OSEK-OS nicht gibt.

Im letzten Bereich werden aus den Modellen Textdateien erzeugt, die von den vorhandenen Übersetzern als Eingabe verwendet werden. Dabei ist man gänzlich auf eine Eigenentwicklung angewiesen, wobei das erstellt Metamodell als Basis der Implementierung dient.

Neben den theoretischen Überlegungen, ist eine praktische Umsetzung unerlässlich. Deshalb wurden drei gängige UML Werkzeuge untersucht und Visual Paradigm als das am besten geeignete ausgewählt. Die Realisierung erfolgte in zwei Schritten, wobei zuerst die Informationen der modellierten Applikation in eine Instanz des erstellten Metamodells überführt worden. Im zweiten Schritt können daraus alle benötigten Textdateien direkt erzeugt werden.

Der vorgestellte Ansatz ist für eine Teilmenge von OSEK-OS mit dem Werkzeug Visual Paradigm realisiert worden. Aufbauend darauf, wird zur Zeit an einer Erweiterung der Abbildung gearbeitet, um OSEK-OS möglichst vollständig zu beschreiben. Eine Erweiterung um OSEK-COM ist geplant, damit auch verteilte Applikationen modelliert werden können. Dies wird dann ein Einsatzgebiet für die Verteilungsdiagramme von UML sein. Weiterhin ist zu untersuchen, ob dieses Vorgehen auch für andere Betriebssysteme (z.B. QNX) erfolgreich eingesetzt werden kann.

Literatur

[EAr07] Sparx Systems. *Enterprise Architect 7.0*, 2007. `http://www.sparxsystems.com`.

[GWS03] Zonghua Gu, Shige Wang und Kang G. Shin. Issues in Mapping from UML Real-Time Profile to OSEK. In *Workshop on Specification and Validation of UML models for Real Time and Embedded Systems (SVERTS)*, 2003.

[JHB06] Sang-Uk Jeon, Jang-Eui Hong und Doo-Hwan Bae. Interaction-based Behavior Modeling of Embedded Software using UML 2.0. In *9th Symposium on Object and Component-Oriented Real-Time Distributed Computing*, 2006.

[MDr08] No Magic, Inc. *MagicDraw UML 15.0*, 2008. `http://www.magicdraw.com`.

[Moo02] Alan Moore. Extending the RT Profile to Support the OSEK Infrastructure. In *5th Symposium on Object and Component-Oriented Real-Time Distributed Computing*, 2002.

[OMG03] OMG. *Meta Object Facility(MOF) 2.0 Core Specification*. OMG, 2003. `http://www.omg.org/cgi-bin/apps/doc?formal/06-01-01.pdf`.

[OMG07] OMG. *UML 2.1.1 Superstructure Specification*. OMG, 2007. `http://www.omg.org/cgi-bin/apps/doc?formal/07-02-05.pdf`.

[OSE04] OSEK/VDX. *OSEK OIL Version 2.5*, 2004. `http://www.osek-vdx.org`.

[OSE05] OSEK/VDX. *OSEK OS Version 2.2.3*, 2005. `http://www.osek-vdx.org`.

[Sel03] Bran Selic. Model-Driven Development of Real-Time Software Using OMG Standards. In *6th Symposium on Object and Component-Oriented Real-Time Distributed Computing*, 2003.

[VP007] Visual Paradigm Intl. *Visual Paradigm 6.1*, 2007. `http://www.visual-paradigm.com/`.

Modeling Variants of Automotive Systems using Views

Hans Grönniger, Holger Krahn, Claas Pinkernell, and Bernhard Rumpe
Institute for Software Systems Engineering
Technische Universität Braunschweig, Germany

Abstract: This paper presents an approach of modeling variability of automotive system architectures using function nets, views and feature diagrams. A function net models an architecture hierarchically and views are used to omit parts of such a model to focus on certain functionalities. In combination with feature diagrams that describe valid variants, the concepts of feature and variant views are introduced to model architectural variants. The relationship between views, variants and the underlying complete architectural model is discussed. Methodological aspects that come along with this approach are considered.

Keywords: Automotive Systems, Logical Architecture, Feature Modeling, Variability

1 Introduction

A main challenge in developing automotive systems is the high diversity of possible variants of a system architecture. A modern car consists of a lot of features a customer can choose from which leads to thousands of possible configurations. Additionally, the complexity of automotive systems is increasing rapidly coupled with a growing need for reusability of existing functionality. This is not just a development issue, but it is also getting harder to maintain and evolve running systems. Especially an intuitive notation of system architectures that considers the high number of variants is missing in today's approaches.

This paper focuses on the aspect of variability in automotive system architectures. An architecture description language based on function nets is introduced to model hierarchically structured and logically distributed systems. Hierarchical decomposition allows for handling the complexity of a system architecture and the concept of views is introduced to improve system understanding. In this paper, views are used to describe features of a system. In combination with feature diagrams they also give an overview of possible variations. Variant views are introduced that abstract from a complete function net to the relevant parts of a feature in a certain variant. The paper contributes to the understanding of systems with high diversity by introducing an intuitive notation to model features and their variants.

In Section 2 of this paper the modeling of automotive architectures with function nets is explained as introduced in [GHK+07, GHK+08]. The section also focuses on the definition of views. Section 3 describes what a feature is and how feature diagrams model

system variability. The concept of feature views is introduced and variant views are defined. The connection between a feature diagram, feature and variant views is explained by examples. Section 4 considers methodological aspects and in Section 5 related work is presented.

2 Modeling Automotive Architectures using function nets and Views

Function nets are a suitable concept for describing automotive logical architectures [vdB04, vdB06]. In our approach, we propose to use a subset of SysML internal block diagrams for function net modeling, as described in [GHK+07, GHK+08]. The next subsection introduces the notation of internal block diagrams. Syntactically, internal block diagrams are valid SysML internal block diagrams [OMG06], but some additional constraints apply as described in [GHK+08]. Internal block diagrams that describe function nets only use directed connectors and no multiplicities. The main advantages of this notation are the intuitive systems engineering terminology and the distinction between the form of a diagram and its use. It is possible to describe both, views and the underlying complete architecture, with the same notation. As a consequence, we achieve a low learning effort and an easy switching of viewpoints between function nets and views. The notation itself should be acceptable for developers from other domains than computer science, because it uses existing systems engineering terminology.

2.1 Internal Block Diagrams

Internal block diagrams (ibd) used for function net modeling contain logical functions and their communication relationships [OMG06]. *Blocks* in the diagram can be decomposed into further blocks to structure the architecture hierarchically or can be simple functions. The interface of a block is formed by its input and output channels.

Blocks are communicating over so called *connectors* which are unidirectional and logical signals are sent from a sender to a receiver over them. Connectors can directly connect two blocks ignoring hierarchical composition (cross-hierarchy communication). A signal has exactly one sender, but it may have several receivers.

Figure 1 shows an internal block diagram of a car comfort functionality. The block consists of a *central locking system (CLS)*, the *central locking system control unit (CLSControlUnit)*, doors and a trunk. The *CLSControlUnit* consists of two different push-buttons and a status interpreter.

The CLS receives two external signals: *AutoLockState* and *Speed*. The system outputs *CLSState* to an external environment. A door can be decomposed as seen in Figure 2. It consists of a *DoorContact* and a *LockControl*. As you can see in Figure 1, the function Door is instantiated four times. In our approach the instantiation of blocks is supported which increases reusability of functions and avoids redundant function definitions. More information on the instantiation mechanism can be found in [GHK+07].

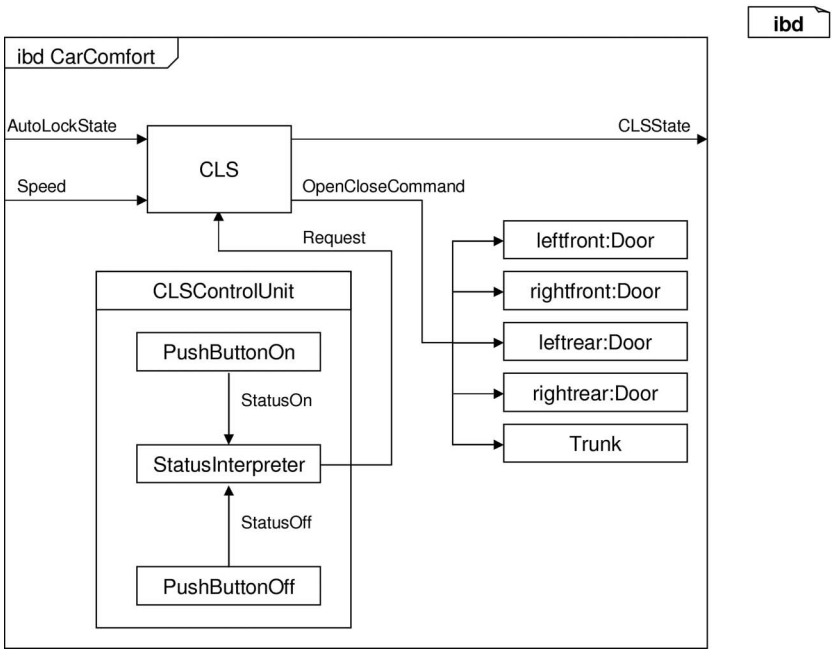


Figure 1: Car comfort function

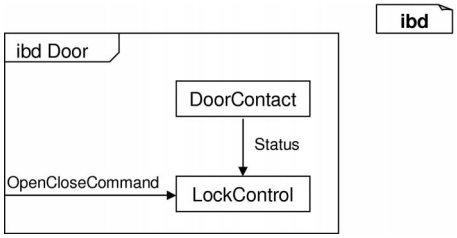


Figure 2: Decomposition of a door

Note that the example is neither complete nor does it illustrate a real design of a comfort function in detail.

2.2 Views

Internal block diagrams are also used to define *views* of a given architecture. A view is denoted by the stereotype ≪view≫ and helps developers to focus on particular functionality by hiding the complexity of a complete logical architecture. A view has to be consistent to an underlying internal block diagram. That means that a view consists of elements of a complete architectural model.

Views do not introduce new elements but abstract from the whole system by omitting blocks and connectors. An exception is the possibility to model the environment of a block and its non-digital communication. Environmental blocks (depicted by stereotype ≪env≫), like actuators or even "the street" can be modeled. There are several possibilities of non-digital communication, e.g., ≪M≫ for mechanical, ≪E≫ for analogous electrical or ≪H≫ for hydraulic stimulation. First elements for an appropriate theory for at least the analogous and digital communication is given in [SRS99]. Modeling the environment is used for better comprehensibility. It is also possible to import so called external blocks. These blocks are marked with the stereotype ≪ext≫ in order to model a view's context in the whole system.

Figure 3 shows a view of the running example. The view depicts a coupe variant of a car comfort function. A coupe has only two doors and a trunk compared to the full model in Figure 1. So the rear doors are omitted.

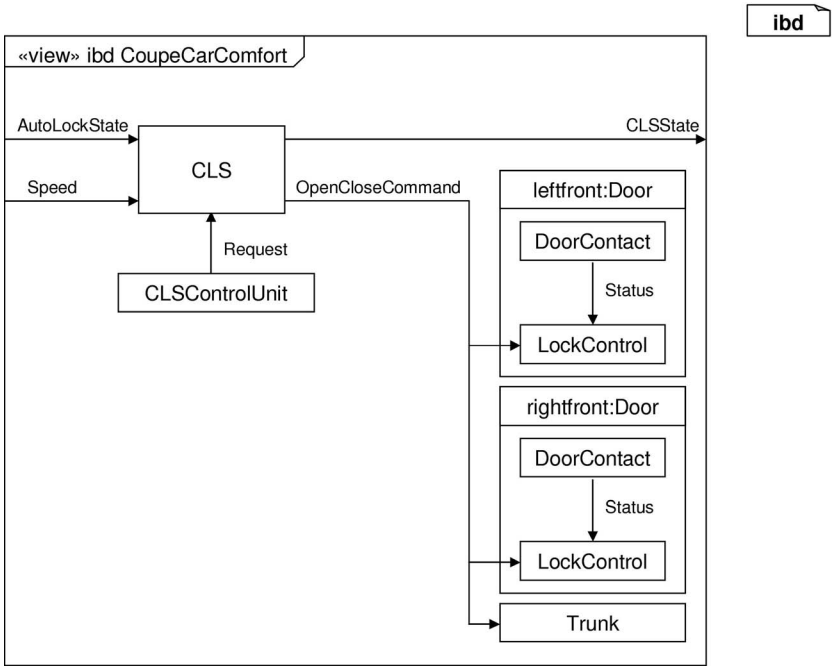


Figure 3: View of car comfort function

Views are abstractions of a complete function net. The following consistency conditions can be checked to validate this relationship [GHK+07]:

1. Each block in the view without a stereotype ≪env≫ must be part of the complete function net.
2. Whole-part-relationships in the view must be present in the complete function net. However, it is permitted to leave out intermediate layers.

3. Elements that are related via a (possibly transitive) whole-part-relationship in the complete function net must also have this relation in a view if both elements are shown.

4. Communication relationships not marked with a stereotype shown in the view must be present in the logical architecture. If the concrete signal is omitted in the view, an arbitrary signal communication must exist in the complete function net.

5. Communication relationships need not be drawn to the exact source or target. Any superblock is sufficient if the exact source or target block is omitted.

Views on views are also supported by the notation. In that case, an additional context condition applies:

6. A view on a function net is a specialization of another view if both are consistent to the complete function net and the blocks and connectors in the specialization function net are a subset of the blocks and connectors shown in the referred function net.

3 Defining Variants with Views and Feature Trees

Our approach of handling variability in automotive systems is based on the idea to use views to describe features and their variants [CE00]. We first recall feature diagrams as a standard way to model the feature variability and possible valid configurations of a system. Next, we define feature views that are used to describe the logical architecture of a particular feature. Subfeatures of a feature are defined as views on the parent feature's view. Variant views can be calculated from the views defined for subfeatures and their relationship expressed in the feature diagram. The connection between the complete logical architecture, a feature diagram, feature views and induced variant views is depicted in Figure 4.

A view of the complete function net is defined for feature F (dashed rectangle). Subfeature views for S1 and S2 are the (overlapping) views indicated by the dotted rectangles. The or-relationship in the feature diagram induces an additional variant "S1 and S2" marked by the solid rectangle.

3.1 Features

A *feature* is an observable unit of functionality of a system. Especially in automotive systems a customer can choose from a variety of features. So there is a multitude of possible feature combinations that make up the complete automotive system. A feature might be a single function or a subset of interacting functions of a complete function net. These functions can also be shared between distinct features.

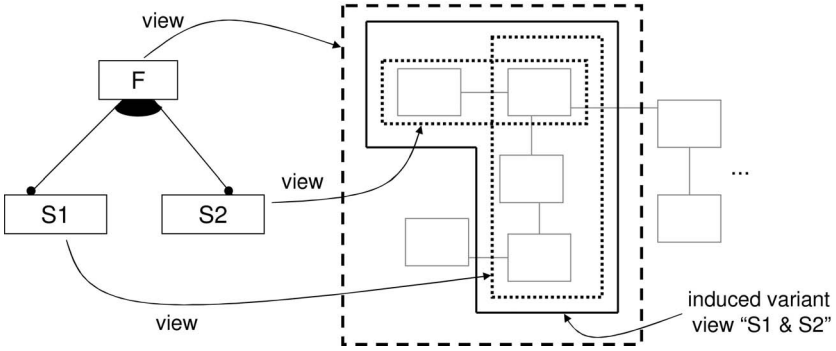


Figure 4: Relationship between the complete logical architecture, its feature views, and their variant views.

Features with subfeatures are usually organized in so called *feature diagrams* (fd) as explained in [CE00]. In our approach we use this notation in addition to function nets and feature views that are described below. A feature diagram, as we use it, is a tree structure with features as its nodes. A feature might be optional (depicted by an empty circle) or mandatory (depicted by a black circle). Features can have a set of alternative subfeatures, depicted by an arc that subsumes all possible alternatives. That means that exactly one subfeature from the set of alternatives is used. A filled arc is used to describe or-features meaning that every combination of subfeatures is allowed. Edges without an arc are used to describe that a feature consists of several subfeatures.

Figure 5 shows an example feature diagram. The car consists of a mandatory feature Engine and an optional feature Comfort Functions. The engine can be chosen from Gasoline, Electric or Hybrid. Any combination of Navigation System, Air Condition and Central Locking System yields a valid Comfort Functions feature.

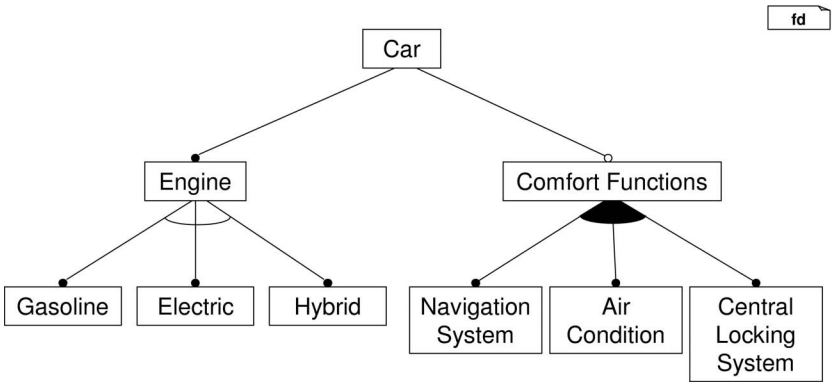


Figure 5: Feature diagram for a simple car

3.2 Feature and Variant Views

As indicated in Figure 4, the (structure of a) feature is defined as a view on a complete function net. This has the advantage that a single feature can on the one hand be understood and analyzed in isolation. On the other hand its embedding in the relatively complex complete function net is explicitly modeled. Views describing features are called *feature views*.

In views, parts of a complete function net that are not relevant for a certain feature are omitted. A feature view only shows relevant blocks and signals. But it may also contain physical devices and even the environment of the car. Figure 6 shows a door with its environment. As explained in Section 2.2, a feature view has to be consistent to a complete function net which can be checked automatically.

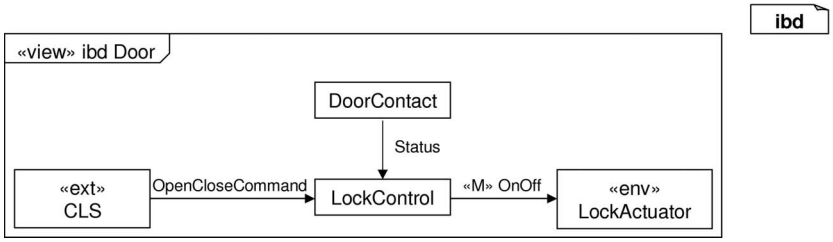


Figure 6: View of a door with its environment consistent to Figure 2

The underlying complete function net that describes the whole logical system architecture consists of all features in all possible variants. The so called complete model or “150 percent model” is an important concept of our approach. The logical architectures of automotive systems are usually 150 percent models. But the realization of the logical architecture as ECUs is normally not a 150 percent model. Still, the realization often contains unused functionality, because it seems more economic to design every car with the same set of basic functionality even if it is not used in the concrete configuration. From a 150 percent model variants can be chosen by parametrization. A disadvantage of this approach is a complex logical architecture. The complexity is handled by views. Each (sub)feature is described by one *view*, c.f. Figure 4. Please note that subfeatures are defined as views on feature views. In that case, the last context condition from Section 2.2 has to hold, i.e., views may not introduce new functions or communication relationships compared to the underlying feature view.

The set of possible variants and according views is calculated from the (sub)feature views and their relationship in the feature diagram which is explained in detail in the next section.

3.3 Variability with Feature Diagrams and Views

We model several feature diagrams and consistent feature and variant views. Note that in the following examples the complete function nets are left out. The given feature views

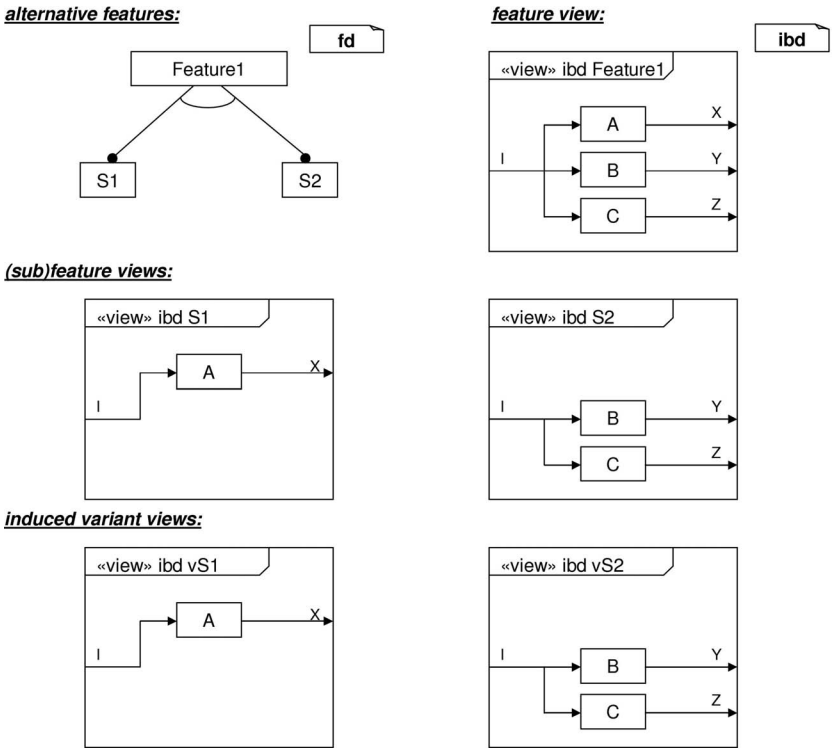


Figure 7: Alternative features

contain the relevant information for the examples.

Alternative Features

Figure 7 shows a feature with its two alternatives subfeatures S1 and S2. The figure also depicts the underlying (150 percent) feature model that is the basis for the two alternatives. Also, all possible variants are shown which, in case of alternatives, coincide with the subfeature views.

We assume that each block has an incoming signal I. Block A has an outgoing signal X, and so on. As motivated later, the complete model still has to be a valid function net in which each signal has exactly one sender, so sharing outgoing connectors is not allowed. Some outgoing signals are only used in one variant. Now, this could mean two things, a) the signals actually have the same signal type and semantics but since it is not possible to have the same signal name with multiple senders, different signal names have to be invented, or b) the signals are not connected in any way. While b) is not a problem, in case a) one could argue that the logical architecture should preserve this information. This could, for instance, be achieved by adding another logical block to the architecture that receives three distinct signals and delegates one common signal. Another possibility

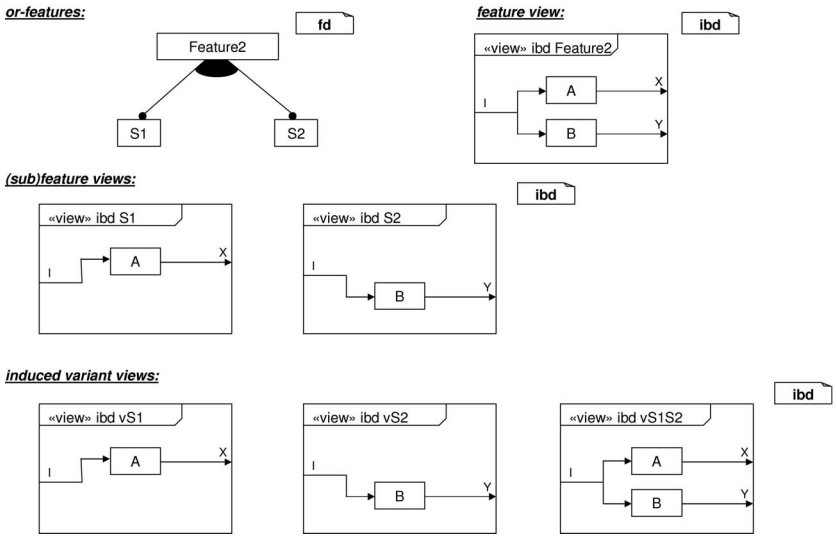


Figure 8: Or-features

would be to add signal types and instances to the function nets.

Or-Features

Figure 8 depicts a feature that may consist of subfeatures S1 or S2 or both of them. As in our previous example we assume that we have an incoming signal I and outgoing signals X, Y. In variant vS1S2 both subfeatures (S1 and S2) are active. So both, X and Y, are sent by Feature2. Again, if X and Y are only different names for (conceptually) the same signal, the receiving blocks or functions have to consider this. In that case, if both signals target at the same function, additional mechanisms (e.g., an arbiter) could be inserted.

Mandatory and optional features

In addition to alternative and or-features we have the composition of subfeatures. In this situation, a subfeature can be mandatory or optional. Figure 9 shows an optional feature S1 and a mandatory feature S2. We assume that we have I as an incoming signal. Block A has an outgoing signal X and B has an outgoing signal Y. The given feature diagram implies two valid variants as seen in our variant views: The variant in which only B is active and another variant in which A and B are active.

Note that there are more combinations of features possible. The basic elements of the notation were introduced here. In [CE00] the normalization of feature combinations is explained in detail.

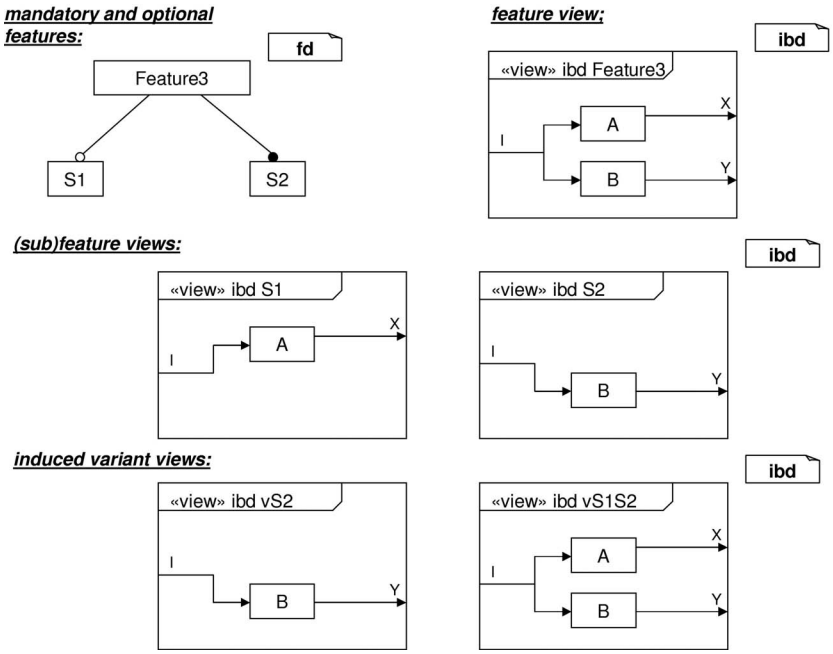


Figure 9: Mandatory and optional features

4 Methodological Aspects

View-based development of an automotive logical architecture can be integrated in a typical automotive development process as described in [GHK+07, GHK+08]. Some additional methodological aspects are discussed below.

Developing Automotive Systems

With this approach we want to introduce an extended methodology and tool chain, that uses an intermediate layer to describe the logical architecture using function nets. This intermediate layer in between the requirements and the implementation phases improves the understandability of automotive systems by focusing on feature modeling and increases reusability.

The features of a system are extracted from requirements that were captured using a requirements management system, e.g., DOORS [Tel08]. The logical architecture is developed by using function nets. The features from requirements are mapped to the logical architecture by feature views. For a better comprehensibility and to abstract from the complete “150 percent model”, feature diagrams and variant views are used. In subsequent development steps the logical architecture can be mapped to a realization, e.g., using the

AUTOSAR [Aut08] methodology.

Automotive systems are not developed from scratch. A new generation of a system is usually based on its predecessor. Being able to reuse large parts of a logical architecture (not just implementations as proposed by AUTOSAR) is hence a key requirement. This is supported by our approach in that the complete logical architecture and feature views can be used as a starting point for the development of the next vehicle generation. Changes are made to the views to fullfil new requirements. Consistency to the complete model is then checked and possible conflicts and inconsistencies can be tracked down automatically and resolved manually since this in general is a complex engineering task.

We hence propose the following development steps to model a logical architecture:

1. Create or adapt feature diagrams.
2. Reuse the complete logical architecture and views from an earlier vehicle generation.
3. Develop self-contained views to describe new features.
4. Adapt feature views to reflect changes in the requirements specifications.
5. Early and repeatedly execute consistency checks between views and the complete function net of a predecessor to identify inconsistencies or unused blocks in order to modify the complete function net.

Advantages and Limitations of a 150 percent model

When developing complete systems, like cars, we can safely assume that at some point the complete system architecture is developed including all possible variations. The so called 150 percent model of an architecture constitutes the basis for modeling system configurations with views. Especially in the automotive domain it is not feasible to generate and test every possible variant configuration. As a consequence, analysis, verification, and testing of an automotive logical architecture can much better be carried out on this 150 percent model that integrates all variants. Therefore, we need the complete logical architecture to be a valid function net (e.g., with one sender per signal only). This however does not mean that we cannot run tests on incomplete models or views. Quite the contrary: It is mandatory to clarify certain functional and non-functional properties early on views and reuse these tests for complete function nets.

Top-down vs. bottom-up development

The development process in general could be a top-down, a bottom-up process or a combination of both. In a top-down process the logical architecture is modeled on a high level of

abstraction and later each element is refined until it has the desired structure. In a bottom-up process modeling stays very close to an implementation. The concept of views can be used to hide the complexity of such an architecture. Today, in the automotive sector most software is designed from bottom-up. One reason is that the placement of a logical function is often already decided because the ECU that is realizing this function is already predetermined. Reuse of system components is not limited to the logical architecture, but is also an aspect on the hardware level. To consider already fixed design decisions, function nets can be equipped with attributes that document these decisions early in the development process [GHK+07].

However, a stringent use of views on function nets can lead to a substantial change of method, as it will be possible to model functional units and assemble them at will. Thus function nets can be developed top-down, bottom-up or even starting in the middle. Furthermore, it will be possible to evolve function nets using calculi like [KR06, PR97, PR99]. Consequently, modeling architectures with function nets and views supports both, top-down and bottom-up modeling. In practice, we assume a mixture of both where new requirements are propagated top-down through the logical architecture that is reused "bottom-up" from an earlier vehicle generation.

5 Related Work

In [CE00] the modeling of features is described using a tree-based notation. This notation is widespread and intuitive. This paper adapts the notation of feature diagrams to model features of automotive systems.

An important concept of handling variability in software systems is introduced in [PBvdL05]. The authors explain how variability of system can be handled as so called product lines. A notation to model the variation points and the variants of a system is introduced. We decided to follow the notation in [CE00] because in our opinion it is more intuitive and often used.

An approach to model the logical architecture of automotive systems with UML-RT is introduced in [vdB04]. This approach is extended by using internal block diagrams from SysML [OMG06] to model logical architectures and views.

In [SE06] the possibility of merging views to a complete system are discussed. Merging views to a complete system could be interesting in a strict top-down development process. Nevertheless, we do not propose to merge views because of constant evolution of requirements. Checking consistency between views and a complete function net eases finding inconsistencies that have to be resolved manually.

In AML [vdBBFR03, vdBBRS02] functions are tailored by introducing a variant dependency in class diagrams. AML uses a strict top-down design of functions. In our approach, logical functions can be shared across multiple features and also variants.

The automated derivation of the product variants of a software product line are explained in [BD07]. The derivation is illustrated by an example product line of meteorological data

systems. As notation a simplified variant of feature diagrams from [CE00] is used.

AUTOSAR [Aut08] is a consortium that standardizes the architecture of automotive systems and improves the reuse of software components. Our approach proposes the AUTOSAR methodology to realize a given function net. But we think it is necessary to have an intermediate layer in between requirements and AUTOSAR to improve the understandability and the reuse of automotive features. Our approach of modeling logical architectures with function nets fills out this gap.

6 Conclusion

In [GHK+07, GHK+08] a notation to model logical architectures as function nets is introduced that fills the gap between requirements and the realization of automotive systems. Based on that notation, the concept of views was explained using the same notation. In this paper the concept was extended to model variants of an architecture. In combination with feature diagrams, feature views and variant views are used to communicate variability of automotive system architectures.

Especially in developing software for automotive systems the explicit modeling and an efficient handling of the high diversity of configurations is important. The variability of a system is mapped from requirements to the logical architecture using views. From there the variability can be mapped on a concrete realization, for instance with help of the AUTOSAR methodology.

Generally, tool support is a fundamental aspect of modeling architectures. So, in future work the tool support has to be extended. With the extension of the tool chain, we think our approach helps to handle variability and complexity of automotive systems in existing development processes. In the future, we also will investigate how the approach can be extended by models that describe the behavior of a feature.

References

[Aut08] Autosar website `http://www.autosar.org`, January 2008.

[BD07] Danilo Beuche and Mark Dalgarno. Software Product Line Engineering with Feature Models. *Overload Journal*, 78:5–8, 2007.

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

[GHK+07] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, and Bernhard Rumpe. View-Based Modeling of Function Nets. In *Proceedings of the Object-oriented Modelling of Embedded Real-Time Systems (OMER4) Workshop*, pages 40–45, October 2007.

[GHK+08] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. Modelling Automotive Function Nets with Views for Features, Variants, and Modes. In *Proceedings of ERTS 2008*, 2008.

[KR06] H. Krahn and B. Rumpe. *Handbuch der Software-Architektur*, chapter Grundlagen der Evolution von Software-Architekturen, pages 133–151. dpunkt-Verlag, 2006.

[OMG06] Object Management Group. SysML Specification Version 1.0 (2006-05-03), August 2006. http://www.omg.org/docs/ptc/06-05-04.pdf.

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, 2005.

[PR97] Jan Philipps and Bernhard Rumpe. Refinement of Information Flow Architectures. In *Proceedings of Formal Engineering Methods*, 1997.

[PR99] Jan Philipps and Bernhard Rumpe. Refinement of Pipe And Filter Architectures. In *FM'99, LNCS 1708*, pages 96–115, 1999.

[SE06] Mehrdad Sabetzadeh and Steve Easterbrook. View merging in the presence of incompleteness and inconsistency. *Requir. Eng.*, 11(3):174–193, 2006.

[SRS99] Thomas Stauner, Bernhard Rumpe, and Peter Scholz. Hybrid System Model. Technical Report TUM-I9903, Technische Univerität München, 1999.

[Tel08] Telelogic DOORS website `http://www.telelogic.com/products/doors/`, January 2008.

[vdB04] Michael von der Beeck. Function Net Modeling with UML-RT: Experiences from an Automotive Project at BMW Group. In *UML Satellite Activities*, pages 94–104, 2004.

[vdB06] Michael von der Beeck. A Formal Semantics of UML-RT. In *Proceedings of MoDELS 2006(LNCS 4199)*, pages 768–782, 2006.

[vdBBFR03] Michael von der Beeck, Peter Braun, Ulrich Freund, and Martin Rappl. Architecture Centric Modeling of Automotive Control Software. In *SAE Technical Paper Series 2003-01-0856*, 2003.

[vdBBRS02] Michael von der Beeck, Peter Braun, Martin Rappl, and Christian Schröder. Automotive Software Development: A Model Based Approach. In *World Congress of Automotive Engineers, SAE Technical Paper Series 2002-01-0875*, 2002.

Modellbasierte Steuerung von autonomen Nutzfahrzeugen

Philipp Wojke, Uwe Berg, Dieter Zöbel

Institut für Softwaretechnik
Universität Koblenz-Landau
Universitätsstraße 1
56070 Koblenz
wojke@uni-koblenz.de
doc@uni-koblenz.de
zoebel@uni-koblenz.de

Abstract: Auf modernen Betriebshöfen werden zur Optimierung komplexer Logistikprozesse vermehrt autonome Nutzfahrzeuge eingesetzt. Zu deren Koordination wird in der Regel eine zentrale Leitsteuerung eingesetzt, die fristgerechte und unfallfreie Abläufe gewährleistet. Für die flexible Entwicklung von Leitsteuerungen wurde ein allgemeines Framework konzipiert. Dieses interpretiert betriebsspezifische Modelle zur Steuerung der Fahrzeuge. Dazu generiert das Framework aus Prozessmodellen individuelle Aufgabenmodelle. Die Anwendbarkeit und der Nutzen des vorgestellten Konzepts wurde mit der Entwicklung zweier Prototypen nachgewiesen.

Schlüsselwörter: modellbasierte Leitsteuerung, Framework, Aufgabemodelle

1 Einleitung

Auf Betriebshöfen lassen sich autonome Nutzfahrzeuge zur Durchführung unterschiedlicher Tätigkeiten einsetzen [Bis00]. Hierbei spricht man von fahrerlosen Transportsystemen (FTS). Zur Koordination der einzelnen Fahrzeuge hat sich in der Industrie der Einsatz von zentralen Leitsteuerungen bewährt, wobei nach [WSHL05] der Trend zu Fahrzeugen mit höherem Autonomiegrad geht.

Vor diesem Hintergrund wurde ein Framework für die Entwicklung von zentralen Leitsteuerungen erstellt, welche im Outdoor-Bereich Nutzfahrzeuge mit hohem Autonomiegrad koordinieren. Die Leitsteuerungen sind funktional angelehnt an die VDI-Richtlinie für Leitsteuerungen für FTS (siehe [VDI05]). Zur kosten- und zeiteffizienten Entwicklung der Leitsteuerungen wird in Anlehnung an [GPR06] ein architekturzentrierter, domänenspezifischer und modellgetriebener Ansatz verfolgt. Handelt es sich bei den meisten FTS um Indoor-Anwendungen oder stark spezialisierte Outdoor-Lösungen für wenige Fahrzeuge, so bietet das vorgestellte Framework eine schnelle und flexible Entwicklung von Leitsteuerungen für eine große Anzahl von Nutzfahrzeugen im Outdoor-Bereich. Zunächst wird ein geeignetes Architekturkonzept vorgestellt und stellvertretend für alle Modelle das Aufgabenmodell erläutert. Die Anwendbarkeit des Konzeptes wird anhand von Prototypen gezeigt.

2 Architekturkonzept

Das Konzept der Leitsteuerungen beruht darauf, dass spezifische Eigenschaften des Betriebshofs und der darauf stattfindenden Abläufe durch Modelle abgebildet werden. Die Modelle werden mit Hilfe zusätzlicher Informationen, wie z.B. Aufträgen, durch die Leitsteuerung interpretiert, umgeformt und in Steuerbefehle für die autonomen Fahrzeuge oder die Infrastruktur des Betriebshofs umgesetzt. Die Interpretation der Modelle geschieht durch auf sie abgestimmte Verfahren und Algorithmen.

Zur Vereinfachung der Entwicklung werden die Leitsteuerungen aus unterschiedlichen Sichtweisen betrachtet und in einfachere Bestandteile zerlegt. Eine funktionale Architektur definiert eine hierarchische Zerlegung der Leitsteuerung in einzelne Funktionsblöcke. Im Sinne von [RHS05] beschreibt eine serviceorientierte Architektur die Kombination der implementierten Komponenten (siehe Abb. 1). Diese trennt die Leitsteuerung in die vier Schichten *Daten*, *Datenverwaltung*, *Verarbeitung/Dienste* und *Vermittlung*. Die Dienste implementieren die Funktionsblöcke der funktionalen Architektur. Der Kontrollfluss zwischen den Diensten wird über die Vermittlung abgewickelt. Zu verarbeitende oder erstellte Daten werden außerhalb der Dienste verwaltet.

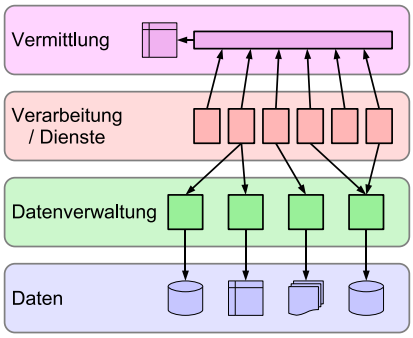


Abbildung 1: Strukturelle Architektur einer Leitsteuerung

Bei den zu verarbeitenden Daten handelt es sich dabei insbesondere um Modelle des Betriebshofs, der Fahrzeuge und der Prozesse. Es werden fixe und variable Modelle unterschieden. Fixe Modelle beschreiben unveränderliche Eigenschaften des Systems, insbesondere das Layout und die Arbeitsabläufe des Betriebshofs sowie die verwendeten Fahrzeuge. Die Speicherung und Spezifikation der fixen Modelle geschieht im XML-Format, was eine einfache Bearbeitung mit unterschiedlichen Werkzeugen erlaubt. Aus den fixen Modellen sowie weiteren Eingaben werden zur Laufzeit variable Modelle generiert, wie z.B. die geplanten Schritte zur Ausführung eines Auftrags. Aus diesen variablen Modellen werden durch Dienste weitere variable Modelle oder Befehle für Fahrzeuge oder Infrastruktur generiert. Eine vollständige Leitsteuerung besteht aus der Implementation der funktionalen und strukturellen Architektur sowie passenden Modellen.

Die Entwicklung einer Leitsteuerung für einen Betriebshof erfolgt dabei in Anlehnung an [MK02] in drei Schritten (siehe Abb. 2), welche ein betriebsunspezifisches Framework in eine betriebsspezifische Leitsteuerung überführen. Die Basis bildet ein allgemeines Framework einer Leitsteuerung für Betriebshöfe. Dieses implementiert die strukturelle Architektur als Rahmen, der durch zusätzliche Module zu einer vollständigen Anwendung ergänzt wird. Das allgemeine Framework implementiert insbesondere die Vermittlung zwischen den Diensten. Weiterhin werden allgemeine Dienste und Teile der Datenverwaltung spezifiziert, als Module implementiert und über Bibliotheken bereitgestellt. Es werden allgemeine Schemata von Modellen, aber keine konkreten Modelle spezifiziert.

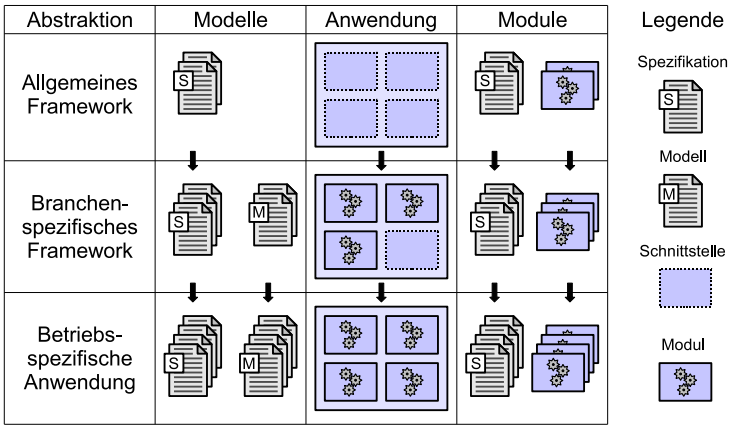


Abbildung 2: Stufenweise Entwicklung einer Leitsteuerung

Im zweiten Schritt wird das allgemeine Framework zu einem branchenspezifischen Framework ergänzt. In diesem Schritt werden insbesondere Verfahren zur Planung und Steuerung der Fahrzeuge festgelegt. Hierfür werden die notwendigen Modelle sowie die Module für Dienste und Datenverwaltung spezifiziert. Bereits feststehende Modelle werden den Schemata entsprechend erstellt. Des Weiteren werden feststehende Module implementiert und in die Anwendung eingefügt. Die Implementierung der Anwendung ist damit weitgehend abgeschlossen. Für eine vollständige Leitsteuerung fehlen insbesondere noch die betriebsspezifischen fixen Modelle.

Der letzte Schritt der Entwicklung besteht darin, das branchenspezifische Framework mit den betriebsspezifischen Modellen zu vervollständigen. Diese beschreiben insbesondere das Layout und die genauen Prozesse eines spezifischen Betriebshofs. Durch die Verwendung von spezifischen Modellen und deren Interpretation und Verarbeitung durch branchenübliche oder allgemeine Module ist eine Anpassung des restlichen branchenspezifischen Frameworks nur in wenigen Fällen notwendig.

3 Aufgabenmodell

Für eine komplette Leitsteuerung werden verschiedenste Modelle benötigt, welche die Eigenschaften und Abläufe auf dem Betriebshof beschreiben. Stellvertretend für alle Modelle wird das Aufgabenmodell am Beispiel eines Speditionshofs beschrieben. Das Aufgabenmodell umfasst alle Arbeitsabläufe in Form von fixen Modellen, aus denen individuelle Arbeitsabläufe in Form von variablen Modellen generiert werden.

Das fixe Modell beschreibt elementare oder zusammengesetzte Aufgaben, aus denen individuelle Abläufe zusammengestellt werden können. Elementare Aufgaben, beispielsweise „Schranke X öffnen“ oder „Fahrzeug X an Laderampe Y andocken“, werden durch zuge-

ordnete Dienste direkt in Befehle für die in Fahrzeugen oder Infrastruktur eingebetteten Systeme übersetzt.

Die Semantik einer Aufgabe wird durch Prädikate über Ein- und Ausgabeparameter beschrieben. So beschreibt eine Vorbedingung die Voraussetzungen für die Ausführung und eine Nachbedingung deren Ergebnisse. Eine Invariante und eine Veränderliche beschreiben Eigenschaften, die nicht aus Vor- und Nachbedingung hergeleitet werden können. Beispielsweise ist die Positionierung des Fahrzeugs mit dem Heck zur Laderampe eine Voraussetzung für das Andocken an diese. Als Ergebnis steht das Fahrzeug bündig zur Laderampe.

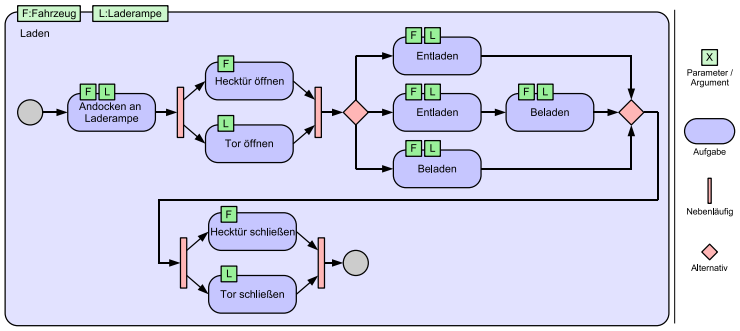


Abbildung 3: Beispiel einer zusammengesetzten Aufgabe

Zur Beschreibung festgelegter Abläufe werden einfachere Aufgaben zu komplexeren zusammengefasst. Hierzu stehen die Verknüpfungen Sequenz, Auswahl, Nebenläufigkeit und Wiederholung zur Verfügung. Komplexe Aufgaben können in Anlehnung an UML-Aktivitätsdiagramme dargestellt werden (siehe Abb. 3). Die Kombinationsmöglichkeiten von Aufgaben werden implizit durch die Prädikate der Aufgaben eingeschränkt. Die Kombinationsmöglichkeiten zweier Aufgaben können auch explizit eingeschränkt werden, beispielsweise auf die Reihenfolge Entladen vor Waschen.

Einige der Aufgaben des fixen Modells können als mögliche Bestandteile von Aufträgen an das System ausgezeichnet werden. Ein solcher Auftrag definiert die auszuführenden Aufgaben, deren Vernetzung und notwendigen Argumente. Gegenüber komplexen Aufgaben werden keine vollständigen Abläufe, sondern nur die wichtigsten Schritte definiert, beispielsweise „entlade Fahrzeug A an Laderampe B, danach betanke A, danach Abmelden von A an Station C“. Dieser Auftrag wird nun unter Verwendung des fixen Modells von einem Planungsmodul in ein erstes variables Modell überführt, das die möglichen Abläufe zur Erfüllung des Auftrags beschreibt. Hierbei werden zusätzliche Aufgaben, z.B. „fahre A von Tankstelle nach C“, anhand der Beschreibung der Aufgaben des fixen Modells ergänzt. Kombinierte Aufgaben werden hierbei noch nicht aufgelöst, um die Komplexität des variablen Modells niedrig zu halten.

Im Folgenden wird das grobe variable Modell schrittweise in ein ausführbares Modell überführt, bei dem alle Aufgaben und Argumente eines Ausführungspfades feststehen. Hierbei werden für noch nicht festgelegte Argumente, beispielsweise Zeiten oder Fahrwe-

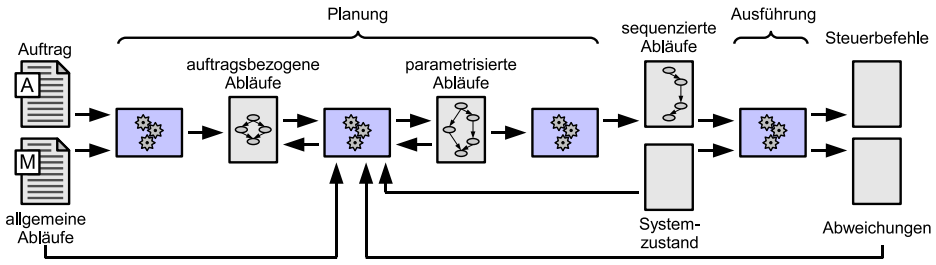


Abbildung 4: Das Aufgabenmodell und seine Verarbeitung

ge, sinnvolle Werte ausgewählt. Dabei können mehrere Alternativen mit unterschiedlichen Argumenten entstehen. Weiterhin werden zusammengesetzte Aufgaben in Teilaufgaben expandiert. Um die Überführung zu beschleunigen werden bei Alternativen nicht zutreffende oder ausführbare Pfade entfernt und kürzere Pfade zuerst bearbeitet. Das Ergebnis der Umwandlung ist ein variables Modell mit mindestens einem vollständig expandierten, parametrisierten und ausführbaren Pfad. Der nächste Schritt ist die Sequenzierung dieses Pfades, bei der elementaren Aufgaben extrahiert und deren genaue Ausführungsreihenfolge festgelegt wird. Diese können dann ausgeführt werden (siehe Abb. 4).

Bei einem Betriebshof handelt es sich nicht um ein vollständig kontrollierbares System. So können manuelle Schritte oder Eingriffe nicht oder nicht exakt geplant werden. Bei der Ausführung der elementaren Aufgaben kann es daher immer wieder zu Abweichungen von geplanten Abläufen kommen. Abweichungen werden mit Hilfe der Prädikate der Aufgaben festgestellt und müssen durch eine Anpassung der geplanten Abläufe berücksichtigt werden. Um eine komplette Neuplanung der Aufträge zu vermeiden, wird dabei auf die in den Zwischenschritten erstellten variablen Modelle zurückgegriffen. Hierbei wird die Ausführbarkeit unterschiedlicher Alternativen unter Berücksichtigung der geänderten Gegebenheiten erneut bewertet.

4 Prototypen

Zur Überprüfung der Anwendbarkeit und des Nutzens der vorgestellten modellbasierten Leitsteuerung wurden zwei spezielle Leitsteuerungen für unterschiedliche Betriebshöfe entwickelt. Bei den Betriebshöfen handelt es sich um einen Speditionshof sowie ein Terminal der Bahn zur Verladung von Sattelzügen (siehe [Woj07]). Beide Leitsteuerungen verwenden für die Planung der Fahrwege das gleiche Verfahren und somit weitgehend die gleichen Modellstrukturen und Module.

Das allgemeine und branchenspezifische Framework sowie die speziellen Leitsteuerungen wurden dabei mit einer Komponenten-Bibliothek erstellt. Diese wurde innerhalb der Arbeitsgruppe speziell für die Entwicklung von Anwendungen des assistierten und autonomen Fahrens entwickelt. Die benötigten fixen Modelle wurden noch ohne Unterstützung spezieller Werkzeuge erstellt, da diese sich noch in der Entwicklung befinden.

Die Erprobung der Leitsteuerungen wurde in einer Simulationsumgebung durchgeführt, da keine realen Betriebshöfe und autonomen Fahrzeuge zur Verfügung stehen. Aus Sicht der Leitsteuerungen ist es nicht relevant, ob es sich um einen realen oder simulierten Betrieb handelt. Eigenständige 3D-Visualisierungen erlauben die Beobachtung der Geschehnisse auf den Betriebshöfen.

5 Resümee

Mit Hilfe moderner Softwareentwicklungsmethoden wurde ein Konzept zur Entwicklung architekturzentrierter, domänenspezifischer und modellgetriebener Leitsteuerungen für Betriebshöfe erarbeitet.

Die Entwicklung der Prototypen hat gezeigt, dass das vorgestellte Konzept praktisch anwendbar ist. Allerdings wurde deutlich, dass eine werkzeugunterstützte Erstellung von Spezifikationen und Modellen nahezu unverzichtbar ist. Eine grafische Darstellung der Modelle erhöht erheblich die Übersicht und eine automatische Verifikation hilft Fehler zu vermeiden. An der Entwicklung geeigneter Werkzeuge wird bereits gearbeitet.

Insgesamt können durch den Einsatz eines allgemeinen Frameworks mit austauschbaren Modulen Leitsteuerungen flexibel und zeiteffizient für betriebsspezifische Anforderungen erstellt werden. Zudem können die erstellten Anwendungen auch nachträglich mit geringem Aufwand flexibel angepasst werden.

Literatur

[Bis00] Richard Bishop. Intelligent Vehicle Applications Worldwide. *IEEE Intelligent Systems*, 15(1):78–81, 2000.

[GPR06] Volker Gruhn, Daniel Pieper und Carsten Röttgers, Hrsg. *MDA. Effektives Software-Engineering mit UML2 und Eclipse*. Springer Verlag, Heidelberg, Juli 2006.

[MK02] Mari Matinlassi und Jarmo Kalaoja. Requirements for Service Architecture Modeling. In J. Bezivin und R. France, Hrsg., *Workshop in Software Model Engineering*, 2002.

[RHS05] Jan-Peter Richter, Harald Haller und Peter Schrey. Serviceorientierte Architektur, 2005. `http://www.gi-ev.de/service/informatiklexikon/informatiklexikon-detailansicht/meldung/118/`, 31.01.2008.

[VDI05] VDI-Gesellschaft Fördertechnik Materialfluss Logistik. *VDI-Richtlinie: VDI 4451 Blatt 7*, Oktober 2005.

[Woj07] Philipp Wojke. Ein Framework für automatisierte Betriebshöfe mit intelligenten Nutzfahrzeugen. In K.Berns und T.Luksch, Hrsg., *Autonome Mobile Systeme 2007*, Heidelberg, 2007. GI-Gesellschaft für Informatik e.V., Springer Verlag.

[WSHL05] Danny Weyns, Kurt Schelfthout, Tom Holvoet und Tom Lefever. Decentralized control of E'GV transportation systems. In *AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, Seiten 67–74, New York, NY, USA, 2005. ACM.

Technische Universität Braunschweig
Informatik-Berichte ab Nr. 2003-06

2003-06	A. Keese	A Review of Recent Developments in the Numerical Solution of Stochastic Partial Differential Equations (Stochastic Finite Elements)
2003-07	M. Meyer, H. G. Matthies	State-Space Representation of Instationary Two-Dimensional Airfoil Aerodynamics
2003-08	H. G. Matthies, A. Keese	Galerkin Methods for Linear and Nonlinear Elliptic Stochastic Partial Differential Equations
2003-09	A. Keese, H. G. Matthies	Parallel Computation of Stochastic Groundwater Flow
2003-10	M. Mutz, M. Huhn	Automated Statechart Analysis for User-defined Design Rules
2004-01	T.-P. Fries, H. G. Matthies	A Review of Petrov-Galerkin Stabilization Approaches and an Extension to Meshfree Methods
2004-02	B. Mathiak, S. Eckstein	Automatische Lernverfahren zur Analyse von biomedizinischer Literatur
2005-01	T. Klein, B. Rumpe, B. Schätz (Herausgeber)	Tagungsband des Dagstuhl-Workshop MBEES 2005: Modellbasierte Entwicklung eingebetteter Systeme
2005-02	T.-P. Fries, H. G. Matthies	A Stabilized and Coupled Meshfree/Meshbased Method for the Incompressible Navier-Stokes Equations — Part I: Stabilization
2005-03	T.-P. Fries, H. G. Matthies	A Stabilized and Coupled Meshfree/Meshbased Method for the Incompressible Navier-Stokes Equations — Part II: Coupling
2005-04	H. Krahn, B. Rumpe	Evolution von Software-Architekturen
2005-05	O. Kayser-Herold, H. G. Matthies	Least-Squares FEM, Literature Review
2005-06	T. Mücke, U. Goltz	Single Run Coverage Criteria subsume EX-Weak Mutation Coverage
2005-07	T. Mücke, M. Huhn	Minimizing Test Execution Time During Test Generation
2005-08	B. Florentz, M. Huhn	A Metamodel for Architecture Evaluation
2006-01	T. Klein, B. Rumpe, B. Schätz (Herausgeber)	Tagungsband des Dagstuhl-Workshop MBEES 2006: Modellbasierte Entwicklung eingebetteter Systeme
2006-02	T. Mücke, B. Florentz, C. Diefer	Generating Interpreters from Elementary Syntax and Semantics Descriptions
2006-03	B. Gajanovic, B. Rumpe	Isabelle/HOL-Umsetzung strombasierter Definitionen zur Verifikation von verteilten, asynchron kommunizierenden Systemen
2006-04	H. Grönniger, H. Krahn, B. Rumpe, M. Schindler, S. Völkel	Handbuch zu MontiCore 1.0 - Ein Framework zur Erstellung und Verarbeitung domänenspezifischer Sprachen
2007-01	M. Conrad, H. Giese, B. Rumpe, B. Schätz (Hrsg.)	Tagungsband Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme III
2007-02	J. Rang	Design of DIRK schemes for solving the Navier-Stokes-equations
2007-03	B. Bügling, M. Krosche	Coupling the CTL and MATLAB
2007-04	C. Knieke, M. Huhn	Executable Requirements Specification: An Extension for UML 2 Activity Diagrams
2008-01	T. Klein, B. Rumpe (Hrsg.)	Workshop Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen, Tagungsband