Erfahrungen mit der UML beim Entwurf von Kfz-Steuerungen

Marco Götze
<marco.goetze@imms.de>

Wolfram Kattanek
<wolfram.kattanek@imms.de>

Institut für Mikroelektronik- und Mechatronik-Systeme (IMMS) gGmbH
Langewiesener Str. 22
98693 Ilmenau

16. Februar 2001

Zusammenfassung

Dieses Papier setzt sich unter praktischen Gesichtspunkten damit auseinander, inwieweit die UML zur Bewältigung der zunehmenden Komplexität heutiger Aufgabenstellungen im Bereich der Kfz-Elektronik geeignet ist. Es wird die UML zunächst vorgestellt und dann die Art und Weise am Beispiel illustriert, in der sie beim Entwurf eingebetteter Kfz-Steuerungen Anwendung finden kann. Letztlich werden aus dem verallgemeinerten Prozess unter Berücksichtigung des aktuellen Angebots an Entwicklungswerkzeugen Schlußfolgerungen gezogen.

Einleitung

Die Fahrzeugelektronik sieht sich, ähnlich anderen Bereichen der Elektronikindustrie, mit einer wachsenden Komplexität der zu entwerfenden Geräte konfrontiert. Im Automobilsektor betrifft dies die Komplexität des Gesamtsystems Fahrzeug ebenso wie die einzelner Elemente: mechanische Komponenten werden verstärkt durch elektronische ersetzt, Trends in Richtung zunehmender Vernetzung (intern wie extern), steigenden Komforts für den Nutzer und elektronischer „Intelligenz“ sind erkennbar. All dies wird überschattet von hohen Anforderungen an Qualität, Entwicklungszeit, Wartbarkeit, Konfigurierbarkeit und Flexibilität gegenüber Änderungen der Spezifikation. Hierdurch stoßen konventionelle Entwurfsmethoden an ihre Grenzen — alternative Ansätze sind gefragt, um die neuen Anforderungen zu meistern.

Die gegenwärtig verbreitete strukturierte Herangehensweise leidet unter Defiziten im Hinblick auf Standardisierung, Abstraktion und Wiederverwertbarkeit. Konkret bedeutet dies, dass der Markt einschlägiger Designwerkzeuge von Nischenlösungen dominiert wird — werkzeugübergreifende Designmethoden sind schwer realisierbar, und Brüche im Design-Ablauf sind unausweichlich. Den eingesetzten Techniken mangelt es zudem an Vielschichtigkeit: Hierarchische Modelle und selektive Sichten auf individuelle Aspekte eines Systems werden von den vorhandenen Tools und Sprachen nur unzureichend unterstützt. Ebenso ist die Wiederverwertung gefundener Lösungen kaum vorgesehen, und selbst innerhalb eines Modells kommt es damit zu Redundanzen.

Die Modellierungssprache UML

Mit ähnlichen Problemen sah sich die traditionelle Softwaretechnik bereits in den 60er Jahren konfrontiert, und diese resultierten schließlich in der Entwicklung des objektorientierten Paradigmas. In den 70er Jahren begannen sich strukturierte Analyse (SA) und Objektorientierung (OO) durchzusetzen, und gegen Ende jenes Jahrzehnts begann die Entwicklung der objektorientierten *Modellierung.* Mitte der 90er Jahre taten sich Methodiker dieser Ansätze in Form der OMG[1] zusammen, um eine OO-Standardnotation zu entwickeln: die *Unified Modeling Language*, kurz *UML* [Dou98b]. Inzwischen hat sich diese auf dem Gebiet der Geschäftssoftwareentwicklung etabliert.

Vorteile der UML liegen neben den allgemeinen Vorzügen der Objektorientierung — Modularisierung, Wiederverwendbarkeit und Abstraktion, herbeigeführt durch die Kombination der Konzepte Datenkapselung, Vererbung und Polymorphismus — in einer weiterführenden Abstraktion des modellierten Systems, verschiedenen Sichten, Unterstützung von Nebenläufigkeiten und einer (weitgehenden) Unabhängigkeit des Modells von der Implementation.

In Anbetracht des Fortschritts, den die klassische Softwaretechnik durch die Einführung objektorientierter Modellierungstechniken erfahren hat, liegt der Gedanke nahe, den Entwurf eingebetteter Kfz-Steuerungen mit denselben Techniken zu reformieren.

UML-Konzepte und -Notationen

Die UML fasst die Beschreibungsmittel der objektorienten Analyse (OOA) und des objektorientierten Designs (OOD) zusammen[2] und erweitert sie um funktionsbezogene Darstellungen. Die Standard-UML umfasst im Wesentlichen die folgenden grafischen Notationen (Diagrammtypen) [Bur97][3]:

- *Anwendungsfalldiagramme* (*use case diagrams*) stellen als Ausgangspunkt der weiteren Modellierung die grobe Funktionalität des Systems *statisch* dar und grenzen es nach außen ab. Ein einzelnes solches Diagramm umfasst mehrere bis viele *Szenarien* (*scenarios*), d.h. mögliche funktionelle Abläufe.
- *Sequenzdiagramme* (*sequence diagrams*) detaillieren den Programmablauf für einzelne Anwendungsszenarien entlang einer Zeitachse und ergänzen damit das in Anwendungsfalldiagrammen statisch dargestellte Gesamtbild um dynamische Aspekte.
- *Kollaborationsdiagramme* (*collaboration diagrams*) sind weitgehend isomorph zu Sequenzdiagrammen, stellen jedoch die Struktur des Modells beim Zusammenspiel einzelner Komponenten gegenüber der Zeitachse in den Vordergrund.
- *Klassen-* und *Objektdiagramme* (*class/object diagrams*) dienen der Zerlegung des Systems in Klassen bzw. Objekte und legen deren Beziehungen untereinander dar. Konzepte wie Muster (*patterns*), Stereotypen (*stereotypes*) und Pakete (*packages*) erlauben Abstraktionen jenseits der bloßen OO-Strukturierung.
- *Zustandsdiagramme* (*state diagrams*) bezeichnen Statecharts in angepasster Harel-Notation [Har87], die das Verhalten von Klassen bzw. Objekten modellieren.
- *Aktivitätsdiagramme* (*activity diagrams*) stellen eine Alternative zu Zustandsdiagrammen dar, die sich aufgrund ihrer Ähnlichkeit zu Programmablaufplänen zur Modellierung algorithmischer Abläufe eignet.

[1] Object Management Group, Inc.

[2] Weiterführende Literaturverweise finden sich z.B. in [Bur97, Kapitel 8.2].

[3] deutsche Bezeichnungen gemäß [JO$^+$]

- *Verteilungsdiagramme* (*deployment diagrams*) dienen der Gruppierung von Komponenten des Systems auf höherer Ebene und der Zuordnung von Software- zu Hardwarekomponenten.

Darüber hinaus sind diagrammübergreifend unterstützte Konzepte wie Einschränkungen (*constraints*) und Kommentare, die als Anmerkungen an Diagramme und Diagrammobjekte gebunden werden können, der Präzisierung und Dokumentation des Modells zuträglich.

Es muss betont werden, dass die UML lediglich eine *Notation* mit zugehöriger Semantik darstellt, keine Entwicklungsmethode: Wie die zur Verfügung gestellten Modellierungsmittel eingesetzt werden, obliegt dem Entwickler.

Spezifika eingebetteter Systeme

Da die UML ursprünglich mit Orientierung auf die reine Softwareentwicklung konzipiert worden ist, ist einerseits nicht ihr voller Umfang für die Modellierung eingebetteter Systeme relevant, und andererseits wird sie einigen spezifischen Anforderungen nur unzureichend gerecht.

Parallelität & Kommunikation

Beim Modellieren von Nebenläufigkeiten, wie sie in komplexeren Steuerungen zwangsläufig auftreten, sind konkrete Multi-Threading-Eigenschaften paralleler Kontrollflüsse nicht oder nicht eindeutig modellierbar. Z.B. sind Beziehungen zwischen bestimmten aktiven Instanzen derselben Klasse schwer spezifizierbar; ebenso können Spezifika des verwendeten Schedulers, wenngleich möglicherweise relevant, nicht ausgedrückt werden. Ein weiteres Defizit liegt in der Unmöglichkeit, Systeme mit dynamischen Elementen präzise zu beschreiben (bspw. die Art und Weise, in der sich ein Pool dynamischer Objekte im Laufe der Zeit verändert). Weiterhin sind, bedingt durch die begrenzten Beschreibungsmittel für Beziehungen zwischen parallelen Kontrollflüssen, Netzwerk- und Verteilungsaspekte schwer zu modellieren.

Soweit erforderlich, muss man sich entweder mit einer unzureichenden Modellierung dieser Aspekte begnügen oder auf andere Beschreibungsmittel zurückgreifen, die dann jedoch manuell mit dem UML-Modell konsistent zu halten sind.

Echtzeitanforderungen

Wenngleich die UML zeitliche Constraints prinzipiell unterstützt, stehen diese nur in Sequenzdiagrammen zur Verfügung und haben lediglich Anmerkungscharakter. Da Rechtzeitigkeit unabdingbar für „harte“ Echtzeitanwendungen — wie sie bspw. bei Motor- oder sicherheitskritischen Steuerungen/Regelungen auftreten — ist, sollten *formale* zeitliche Anforderungen in *allen* Diagrammen spezifizier- und durch ein Werkzeug modellbasiert auswert- und verifizierbar sein. Dies erfordert zwar neben Code-Generierung eine intime Kenntnis des Zielsystems, doch würde die Validierung dadurch sowohl einfacher als auch zuverlässiger werden. Sind zumindest Worst-Case-Ausführungszeiten ableitbar, kann immerhin eine Scheduling-Analyse durchgeführt werden. Dieses Verfahren wird jedoch von UML-Werkzeugen i.a. nicht unterstützt und ist damit ggf. manuell oder mit externen Tools durchzuführen. Standardmittel zur Validierung ist in Anbetracht des heutigen Stands der Werkzeuge die Simulation.

Kontinuierliche Steuerungen

Die UML bietet keine explizite Unterstützung für mathematische Modelle etc., wie sie beim Entwurf regelungstechnischer Steuerungen eingesetzt werden, sondern ist primär für zustandsbasierte Steuerungen geeignet. Um dennoch kontinuierliche Aspekte modellieren zu können, muss mit spezialisierten Werkzeugen kooperiert werden, was die Durchgängkeit des Entwurfsprozesses beeinträchtigt.

Einschlägige Notationen

Die UML unterstützt gewisse, im gegenwärtigen System-Design verbreitete Notationen nicht, wie etwa Timing-Diagramme. Diese können zwar in Ergänzung zum UML-Modell verwendet werden, die Konsistenz ist jedoch manuell zu gewährleisten.

Methodik für den UML-Entwurf von Kfz-Steuerungen

Bei Analyse und Entwurf von Steuerungen auf UML-Basis kommen verschiedene Abstraktionsebenen zum Einsatz, die unterschiedliche Sichten auf das System geben. Kern der Entwurfsmethodik sind OOA und OOD, d.h. Daten und zugehörige Funktionalität werden als Einheit betrachtet und in Klassenhierarchien strukturiert, wohingegen die SA Daten und Funktionen trennt. Bei einer OO-Methodik werden Daten und Funktionen in Objekten gekapselt, die in Bezug zu realen oder konzeptuellen Objekten stehen.

Für den Einsatz im Entwurf eingebetteter (Kfz-)Steuerungen hat sich die folgende allgemeine Vorgehensweise bewährt [Dou98a, Dou98b, FMS00]:

1. Formalisierung der Spezifikation — Anforderungsanalyse

Abhängig davon, ob bereits eine Spezifikation vorgegeben ist, wird eine solche erstellt bzw. die existierende umgeformt. Primäres Beschreibungsmittel hierfür sind Anwendungsfalldiagramme, deren Unterscheidung von funktionellen Komponenten des Systems und externen *Akteuren* (*actors*) hilft, die Systemgrenze zu bestimmen. Darüber hinaus wird die gewünschte Gesamtfunktionalität des Systems statisch beschrieben.

Nachdem alle Anwendungsfälle identifiziert worden sind, wird ihr Verhalten beschrieben. Hierzu können komplexe Anwendungsfälle in untergeordneten Diagrammen analysiert werden. „Atomare“ *Use Cases* werden über Kommentare (textuell oder in Pseudo-Code) oder mittels Sequenz- oder Kollaborationsdiagrammen näher spezifiziert.

2. Objektorientierte Dekomposition

Die Grundlage für die weitere Modellierung bildet die Dekomposition des Systems in Klassen, dargestellt in einem oder mehreren Klassendiagrammen; eine durchdachte Zerlegung ist Voraussetzung für eine effiziente Implementation. Leider ist der Schritt von der Spezifikation zum Klassen- und Objektmodell schwierig und kaum algorithmisch zu generalisieren. Erschwerend kommt hinzu, dass Kfz-Steuerungen eher durch Funktionalität denn durch komplexe Datenstrukturen gekennzeichnet sind, während in der Softwaretechnik Klassen i.a. durch Datenstrukturen bestimmt und aus diesen und ihren zugehörigen Methoden gebildet werden [RBP+93].

Um anhand der bereits identifizierten Anwendungsfälle die Dekomposition zu erleichtern, besteht eine Möglichkeit [FMS00] darin, den einzelnen *Use Cases* unter Zuhilfenahme der detaillierteren Beschreibungen jeweils entweder ein Daten-, Steuerungs- oder Schnittstellenobjekt zuzuordnen[4]. Wo immer semantisch sinnvoll, werden diese Objekte zusammengeführt, und zwischen den entstehenden Objekten werden notwendige Beziehungen definiert.

Ein alternativer Ansatz besteht in der formalen Ableitung der Klassen aus einer textuellen und/oder hinreichend detaillierten formalen Spezifikation. Die hierzu in [Dou98b] beschriebene iterative Vorgehensweise besteht im Wesentlichen aus den folgenden Schritten:

1. Erstellung einer Liste von Objekten (realen wie konzeptuellen) aus in der Spezifikation vorkommenden Substantiven, Zuordnung von Verhalten und Beziehungen anhand mit diesen assoziierter Verben
2. Reduktion dieser Liste auf relevante Objekte unter Zuhilfenahme von zuvor erstellten Anwendungsfalldiagrammen oder eines umfassenden inhaltlichen Verständnisses der Spezifikation

[4] Aufgabenstellungen auf dem Sektor eingebetteter (Kfz-)Steuerungen sprechen für eine solche stereotype Unterscheidung von Klassen.

3. Gruppierung von Objekten in Klassen, Identifikation von Beziehungen zwischen diesen
4. Identifikation des Klassenverhaltens
5. Gruppierung von Klassen
6. Validierung der gefundenen Klassen und Objekte gegenüber der Spezifikation

Unabhängig von der Art und Weise, in der das System in Klassen dekomponiert worden ist, werden in einem Objektdiagramm Informationen zu und Beziehungen zwischen Instanzen dieser Klassen dargelegt.

3. Modellierung des Klassenverhaltens

Da Struktur und Funktionalität eines Systems in engem Zusammenhang stehen, ist es i.a. nicht möglich, bereits in der ersten Phase das Verhalten vollständig zu spezifizieren. Aus diesem Grund werden nach Abschluß der Anforderungsanalyse und Dekomposition weitere Sequenz- und/oder Kollaborationsdiagramme erstellt, nach deren Fertigstellung das System vollständig funktionell spezifiert ist.

Daraufhin kann mit der Modellierung des Verhaltens begonnen werden. Hauptinstrumente hierfür sind Zustands- und Aktivitätsdiagramme. Nebenläufigkeit und Algorithmuscharakter sind zwei Hauptkriterien, die über die eingesetzte Notation entscheiden; abhängig vom verwendeten Werkzeug können Hilfsmethoden und solche, die trivial verständlich und nicht in zustandsabhängige Interaktionen mit anderen Objekten involviert sind, auch direkt implementiert werden.

4. Implementation und Test

Wird ein Modellierungs-Tool mit Code-Generator eingesetzt, ist mit der Modellierung auch bereits die Implementation (weitgehend) abgeschlossen. Anderenfalls schließt sich an die Modellierung eine Implementationsphase an, in der das mittels UML-Notationen formalisierte Verhalten in ausführbaren Programmcode zu wandeln ist.

Mit abgeschlossener Implementation kann die Einhaltung der Spezifikation durch die entworfene Steuerung validiert werden. Hierzu in Betracht kommende Methoden sind die Modellausführung und Tests mittels Simulation. Im Rahmen der Ausführung des Modells unterstützen die meisten Werkzeuge Zustands- und Ereignis-Tracings sowie die Animation von Ablaufdiagrammen (Zustands-, Sequenzdiagramme). Eine Simulation kann entweder als (konfigurationsabhängig generierter) Bestandteil des Modells der Steuerung *modelliert* oder direkt implementiert und mit einer die Steuerung enthaltenden Bibliothek gekoppelt werden.

Ob der Entwicklungsprozess als ganzer einem Wasserfallmodell gleicht oder iterativ durchgeführt wird, ist dem Entwickler überlassen und von der konkreten Aufgabenstellung abhängig. Die vorgestellte allgemeine Herangehensweise soll lediglich den grundlegenden Ablauf verdeutlichen.

Beispiel: Sitzsteuerung

Unsere praktischen Erfahrungen bezüglich des Einsatzes der UML beim Design eingebetteter Kfz-Steuerungen beruhen auf der Umsetzung einer Spezifikation für eine Sitzsteuerung, die im Rahmen eines von DaimlerChrysler ausgeschriebenen OO-Modellierungswettbewerbs im September 2000 bekanntgegeben worden ist [DC]. Zum besseren Verständnis wird der UML-basierte Entwicklungsprozess anhand dieses Beispiels illustriert.

Spezifikation

Der in der Spezifikation beschriebene Sitz umfasste sieben Aktuatoren (sechs Motoren und eine Heizung) und 17 Sensoren (sechs Hall-Sensoren und Tri-State-Kippschalter für die Motoren, zwei Taster für die Heizung und drei weitere Taster zum Speichern und Wiederherstellen zweier Sitzpositionen). Die Steuerung erhielt darüber hinaus Informationen über die Position des Zündschlüssel, den Zustand der sitznächsten Tür und die anliegende Versorgungsspannung.

All diese Ein- und Ausgänge waren über ein als Klassendiagramm und Header-Files spezifiziertes Interface mit der zu implementierenden Sitzsteuerung zu koppeln. Ereignisse sollten von außen durch Methodenaufrufe der Schnittstellenklasse übermittelt werden; umgekehrt sollten Aufrufe von Methoden in einer übergebenen Aktuatorenklasseninstanz in Aktionen resultieren. Ergebnis der Implementation sollte eine Bibliothek sein, die mit der Implementation der Schnittstellenklassen gekoppelt werden würde. Ob diese Interface-Vorgabe repräsentativ für Aufgabenstellungen im eingebetteten Automobilbereich ist, ist strittig, soll hier jedoch nicht zum Gegenstand einer Diskussion werden.

Die zu realisierende Funktionalität der Steuerung umfasste u.a. die folgenden Aspekte:

- Erkennung des Erreichens einer Stopp-Position durch einen Motor anhand des Ausbleibens von Hall-Sensor-Impulsen nach einem Timeout
- Kalibrierung der Motoren und der von der Steuerung angenommenen Position unter Ausnutzung eben dieses Sachverhalts
- Ansteuerung der Motoren gemäß Prioritäten innerhalb zweier Gruppen à drei Motoren (je Gruppe durfte maximal ein Motor aktiv sein)
- Bedienbarkeit der sechs Motoren über die zugehörigen Kippschalter
- automatische Mitführung der Kopfstütze bei manuell ausgelöster Bewegung des Sitzes in Längsrichtung
- automatisches vorübergehendes Zurückfahren des Sitzes zur Erleichterung des Ein-/Aussteigens (*Courtesy Seat Adjustment*)
- Speichern und Laden zweier Sitzpositionen
- Einstellbarkeit der Heizstufe über die entsprechenden Taster

Darüber hinaus waren vom aktuellen Zustand abhängige Maßnahmen einzuleiten, wenn z.B. Randbedingungen bezüglich der Versorgungsspannung oder Geschwindigkeitsbegrenzungen hinsichtlich Sitzverstellungen während der Fahrt verletzt wurden.

Modellierung

Der erste Schritt bestand darin, anhand aus der Spezifikation abgeleiteter grober Anforderungen der Aufgabenstellung eine Wahl bezüglich der zu verwendenden Sprache und des zu verwendenden Werkzeugs zu treffen. Als in diesem Zusammenhang wesentliche Eigenschaften der Steuerung wurden Reaktivität, Echtzeitverhalten, Diskretheit (vs. Regelungscharakter), Nebenläufigkeit und Statik des Systems (im Sinne einer konstanten Menge physischer Komponenten) ermittelt. Die genannten Anforderungen liessen auf eine prinzipielle Eignung der UML schließen; als Werkzeug kam I-Logix' *Rhapsody™*[5] *in C++™* zum Einsatz.

Vor Beginn der Modellierung wurde zunächst die textuell gegebene Spezifikation mittels Anwendungsfalldiagrammen veranschaulicht. Konkret enstanden zwei dieser Diagramme, die das Gesamtsystem des Sitzes mit unterschiedlichem Detailliertheitsgrad funktionell charakterisierten (Abb. 1).

[5] *Rhapsody™* ist ein eingetragenes Warenzeichen von I-Logix, Inc.

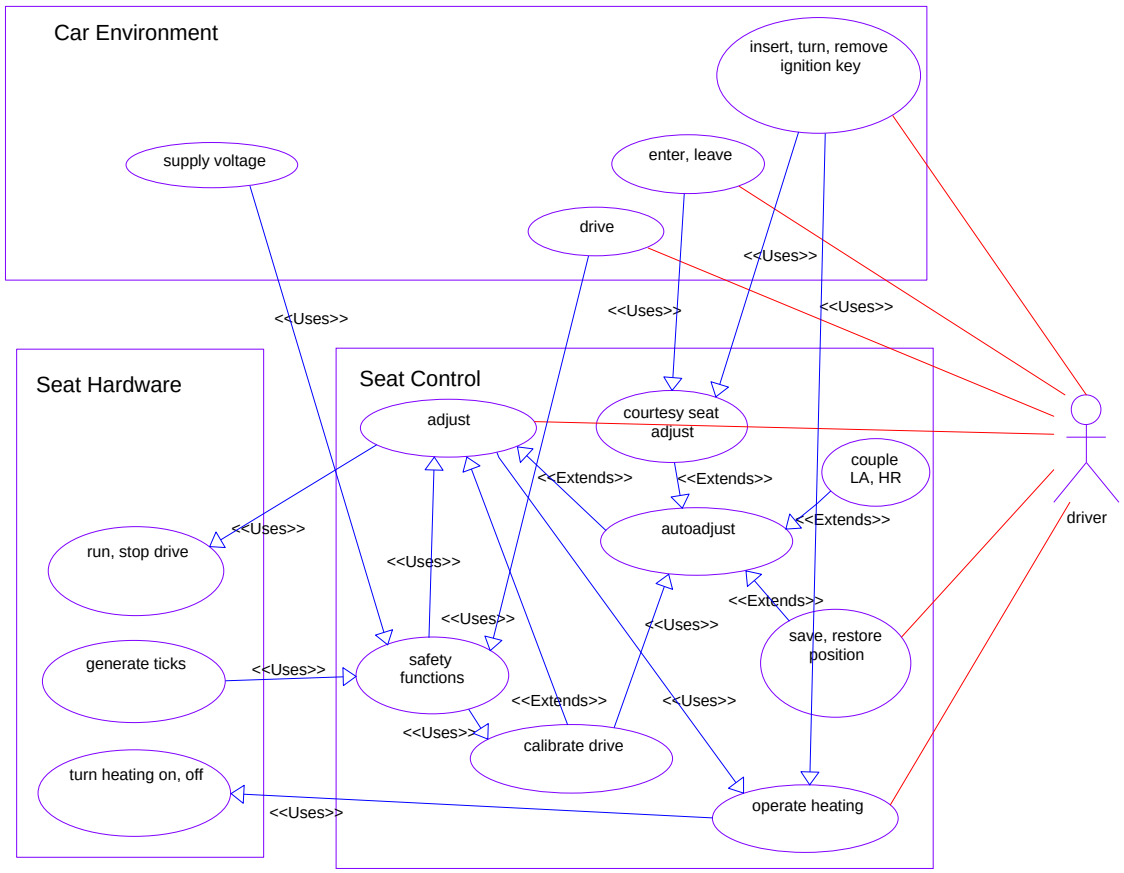

Abbildung 1: Detailliertes Anwendungsfalldiagramm des Sitzes

Anschließend wurde, sowohl unter Verwendung der Anwendungsfalldiagramme als auch der Spezifikation, ein Objektmodelldiagramm, *Rhapsody*™s Konzept eines kombinierten Klassen- und Objektdiagramms, des *Gesamtsystems* „Sitz“ erstellt. Aufgrund der Vorgabe von Schnittstellenklassen reduzierte sich das zu implementierende Klassenmodell jedoch auf die eigentliche Steuerung. Daher entstand ein modifiziertes Diagramm (Abb. 2), welches das Gesamtmodell in zwei Pakete unterteilte: `Interface` und `Controller`. Ersteres umfasste die vorgegebenen Schnittstellenklassen, letzteres die Steuerung, die ihrerseits in mehrere spezialisierte Controller-Klassen unterteilt war. Sinnvoll erschien eine Unterteilung anhand sowohl physischer als auch funktionaler Kriterien.

Im Sinne der Objektorientierung war es naheliegend, die für alle Motoren gleiche prinzipielle Ansteuerungslogik in einer sechsfach zu instanziierenden Klasse `DriveController` zu kapseln. Davon abgeleitet wurde eine spezialisierte Klasse `LADriveController`, die um Besonderheiten des sog. L*ongitudinal* A*djustment Drive*s, nämlich die Mitführung der Kopfstütze und das „*Courtesy Seat Adjustment*“-Feature, erweitert wurde. Dank Polymorphismus musste dieser Controller lediglich als `LADriveController` *instanziiert* und konnte im Folgenden als genereller `DriveController` behandelt werden.

Zur Koordination der Motoren innerhalb der beiden Gruppen wurde eine zweifach zu instanziierende Klasse `DriveGroupController` gebildet, die mit drei `DriveController`-Instanzen assoziiert wurde. Somit bestand der grundlegende Ansatz darin, die sechs Laufwerke von autonomen Controller-Objekten ansteuern zu lassen, die auf Schalterbetätigungen reagierten, bei Notwendigkeit eine Kalibrierung durchführten und die man veranlassen konnte, eine bestimmte Position anzusteuern. Zur Koordination mit den anderen Laufwerken derselben Gruppe setzte die Aktivität eines Motors die Genehmigung

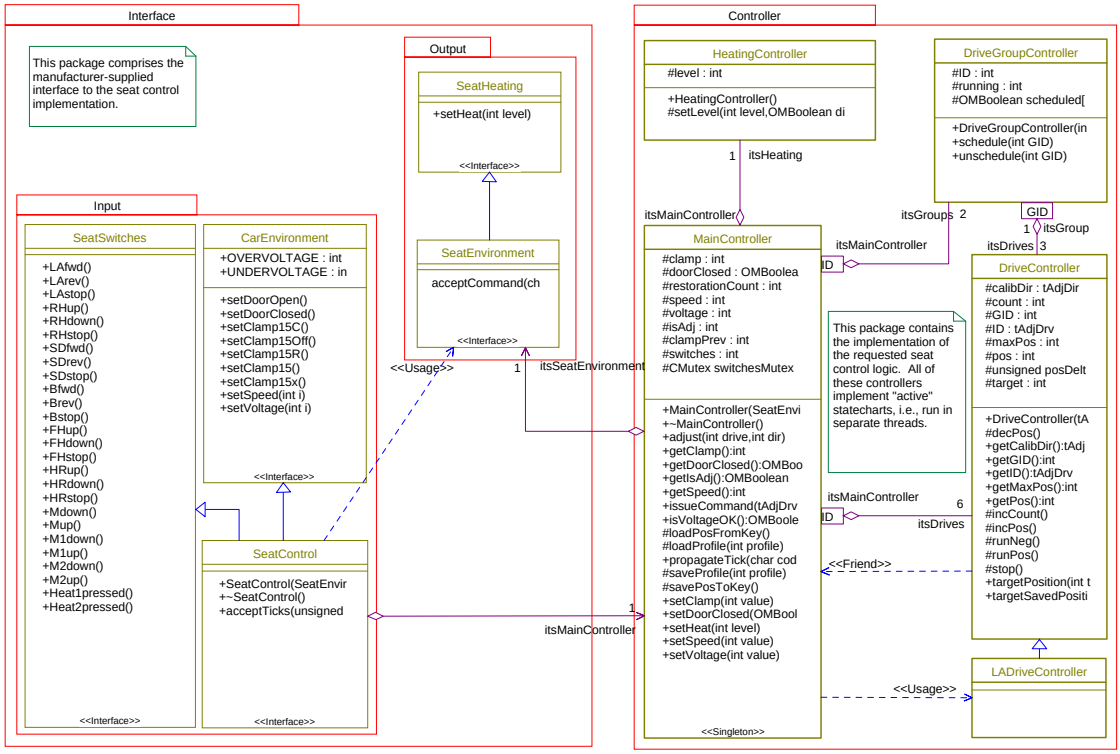

Abbildung 2: Objektmodelldiagramm der Sitzsteuerung

des zugehörigen Gruppen-Controllers voraus, der aufgrund der vorliegenden Genehmigungsgesuche seiner bis zu drei Laufwerks-Controller das Scheduling übernehmen sollte.

Die Steuerung der Heizung wurde an eine Klasse `HeatingController` delegiert, und die Koordination der spezialisierten Controller sowie einige keinem spezifischen Gerät zuzuordnende Funktionen (wie die Speicher-Option) wurden einer Klasse `MainController` übertragen, die die übrigen Controller aggregierte. Diese stellte zugleich die Schnittstelle des `Controller`-Paketes zum `Interface`-Paket dar.

All diese Controller wurden als *aktive* Objekte entworfen, die nebenläufig ausgeführt werden würden.

Nach abgeschlossener Dekomposition der Steuerung wurden die wichtigsten Szenarien wesentlicher Funktionen in Sequenzdiagrammen präzisiert und detailliert. Anschließend erfolgte die Modellierung des Verhaltens der einzelnen Controller-Klassen mittels Statecharts (Abb. 3).

Da *Rhapsody*™ über einen Code-Generator verfügt, entfiel eine separate Implementierungsphase: sämtlicher Programmcode wurde entweder explizit im Klassen-Browser des Modelers eingegeben oder automatisch aus Statecharts generiert.

Getestet wurde die Steuerung durch eine extern implementierte Simulation, deren grafisches Nutzerinterface es ermöglichte, Sensorsignale zu simulieren und die von der Heizung veranlassten Zustandsänderungen des simulierten Sitzes zu beobachten. Im Interesse einer möglichst zuverlässigen Validierung der Implementation wurden systematische Tests anhand einer auf Spezifikation und Modell-Spezifika basierenden Check-Liste durchgeführt und das Verhalten in Grenzfällen (Hardwareversagen, nicht vorgesehene Bedienungsmuster) durch Simulation und Analyse der relevanten Zustandsdiagramme diskutiert.

Bedingt durch die vergleichsweise geringe Komplexität des Steuerungsbeispiels entsprach der Entwurf insgesamt keiner Reinform eines gängigen Entwicklungsprozesses; zwar wurde iterative modelliert und getestet, jedoch wurde auf eine strenge Einhaltung der Phasen innerhalb dieser Spirale verzichtet. Den-

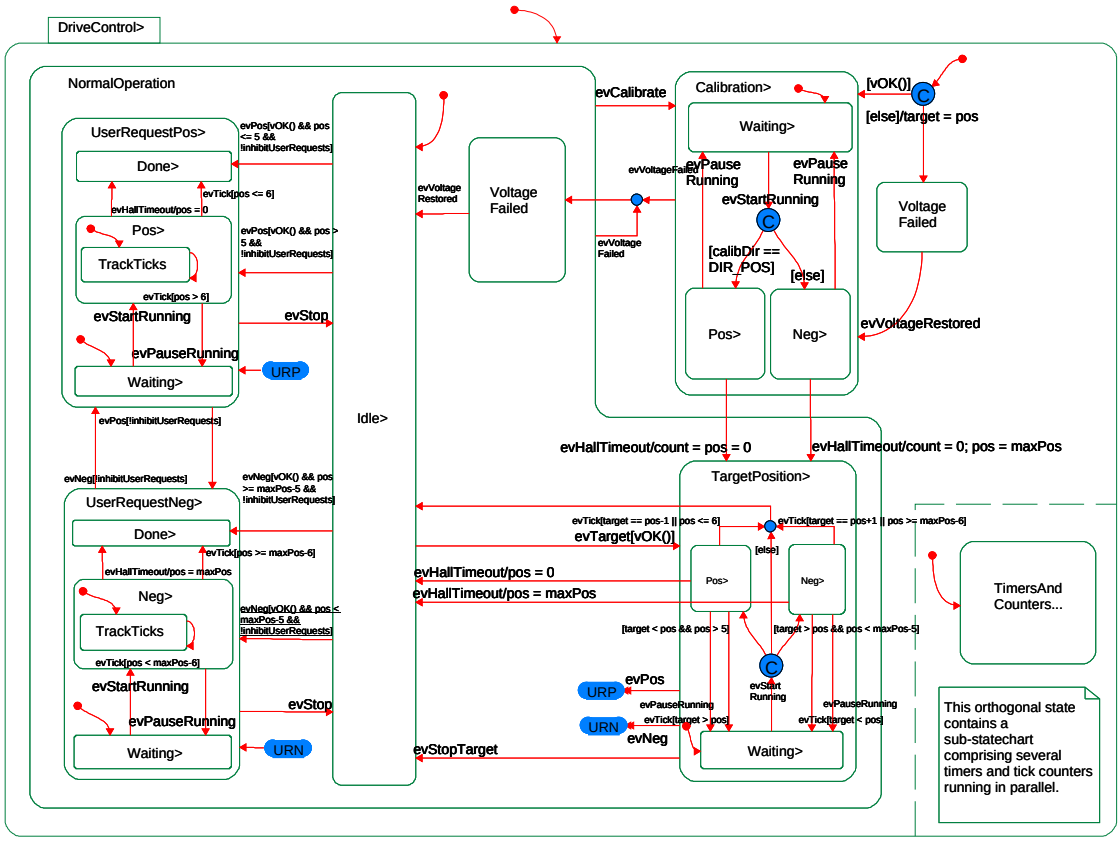

Abbildung 3: Zustandsdiagramm der `DriveController`-Klasse

noch spricht nach den gemachten Erfahrungen nichts dagegen, dass ein UML-basierter Entwurf auch in prozessmodelltreueren Phasen durchführbar und einer größeren Komplexität gewachsen ist — die UML unterstützt diese Vorgehensweise vielmehr sogar durch die Anschaulichkeit ihrer Notationen und die Möglichkeit, Diagramme in Folgezyklen zu verfeinern.

Ergebnisse

Wie spätestens anhand des Beispiels ersichtlich wird, ist die UML nicht nur prinzipiell zum Entwurf von Kfz-Steuerungen geeignet, sondern bietet tatsächlich wesentliche Vorteile, wenngleich das Gesamtbild neben den bereits diskutierten Problemen beim Einsatz der UML für eingebettete Systeme durch einige Defizite getrübt wird. Zudem gibt es zwischen den verschiedenen Werkzeugen funktionelle Unterschiede, die über ihre Eignung für eine gegebene Aufgabe entscheiden können. Ein diesbezüglicher Vergleich verfügbarer Tools würde jedoch den Rahmen dieses Beitrags sprengen.

Abstraktion & Wiederverwendung

Verschiedene, teils hierarchisch ineinandergreifende Notationen und die Verwendung des objektorientierten Paradigmas erlauben Abstraktion und Formalisierung aller Aspekte[6] des entworfenen Systems, was nicht zuletzt die handhabbare Komplexität erhöht. Darüber hinaus unterstützen Objektorientierung, die allgemeine Modularisierung und spezielle UML-Features die Wiederverwertbarkeit sowohl innerhalb des Modells als auch projektübergreifend.

[6] Soweit nicht Element besagter Problemmenge.

Anschaulichkeit & Sichten

Generell erlaubt die Verfügbarkeit verschiedener Sichten dem Entwickler, sich auf jeweils relevante Teilaspekte des Systems zu konzentrieren und irrelevante Details auszublenden. Weiterhin erleichtern kommentierbare grafische Notationen die Kommunikation von Anforderungen mit extra-disziplinären Personengruppen (wie dem Kunden, der Marketing-Abteilung, etc.) und Konzepten innerhalb des Entwickler-Teams und damit die Wartbarkeit und Modifizierbarkeit der Software.

Durchgängiger Designprozess

Die UML bietet effiziente Notationen für die Analyse eines Systems, die dabei helfen, das geforderte Verhalten zu formalisieren. Klassen- und Objektdiagramme stellen eine Standardnotation für die als Basis für die weitere Modellierung dienende OO-Struktur dar. Während der Modellierung wird mit Statecharts ein etabliertes Entwurfsmittel unterstützt, was den Umstieg auf einen UML-basierten Entwurf erleichtert. Die Implementation erfolgt i.a. unter Verwendung eines Code-Generators. Von Werkzeugen zur Verfügung gestellte Debugging- und Animationsfunktionen unterstützen die Validierung der entworfenen Steuerung.

Somit bildet die UML die Grundlage für einen Entwicklungsprozess, der nahtlos von der Analyse oder Spezifikation zur Implementation und Validierung führt.

Rapid Prototyping

Durch den Zwang zur detaillierten Formalisierung der Spezifikation bereits in frühen Phasen werden spätere Verzögerungen und Design-Fehler weitgehend vermieden und, soweit vom Werkzeug unterstützt, prinzipiell eine Modellausführung bereits in diesem Abschnitt und damit die Validierung der Spezifikation mit dem Auftraggeber anhand eines virtuellen Prototypen möglich.

Gesamtsystemmodell

Mit der UML ist es möglich, das *gesamte* System zu modellieren, also Hardwarekomponenten einzubeziehen und deren Zusammenspiel mit Softwareelementen darzustellen. Damit ist die UML für ein Co-Design von Hardware und Software geeignet, wenngleich der Entwurf peripherer Hardware mangels entsprechender UML-Beschreibungsmittel mit dedizierten Werkzeugen und Sprachen erfolgen muss.

Echtzeitaspekte

In allen charakteristischen Notationen sind Nebenläufigkeiten modellierbar. In Sequenzdiagrammen können zeitliche Einschränkungen als Anmerkungen definiert werden; in sämtlichen Diagrammtypen werden allgemeine Constraints unterstützt. Mängel bestehen jedoch, wie bereits beschrieben, hinsichtlich der Beschreibbarkeit komplexer Zusammenhänge innerhalb parallelisierter Systeme und der Formalisierbarkeit zeitlicher Constraints.

Code-Generierung

Die durch ein geschlossenes Modell ermöglichte *vollständige* Code-Generierung kann, sofern vom Werkzeug unterstützt und korrekt arbeitend, nicht nur Zeit und Kosten sparen, sondern sowohl Konsistenz als auch Unabhängigkeit von Modell und Implementation gewährleisten. Ebenso erhöht sich die Flexiblität gegenüber einer sich ändernden Spezifikation: Anpassungen sind lediglich im Modell vorzunehmen.

Im Zusammenhang mit dem objektbasierten Polymorphismus und der Unterstützung verschiedener Zielkonfigurationen durch das Werkzeug ist eine verbesserte Konfigurierbarkeit der Steuerung gegeben, da je nach Bedarf abgeleitete Objekttypen verwendet werden können.

Darüber hinaus kann ein ggf. zur Verfügung gestelltes, modifizierbares Framework, das Ereignisbehandlung, zeitliche Aspekte (Timeouts) und Multi-Threading unterstützt, dazu beitragen, das Modell auf relevante Aspekte zu begrenzen, eine effiziente Umsetzung zu erzielen und die Portierung auf neue Zielplattformen zu erleichtern.

Traceability, also Assoziationen zwischen Anforderungen und zugehörigem Code, kann durch modellobjektgebundene Kommentare erfolgen, die bei gängigen Code-Generatoren in den Zielcode eingebunden werden.

Derzeit zieht die Verwendung von Code-Generatoren Effizienzeinbußen hinsichtlich der Implementation im Vergleich mit einer Umsetzung auf niedrigerer Ebene nach sich. Mit fortschreitender Entwicklung wird dieses Manko jedoch in absehbarer Zeit behoben werden.

Schwierig gestaltet sich auch die Durchführung von Schedulinganalysen, maximalen Speicherbedarfsermittlungen etc. anhand des generierten Codes, da hierzu eine detaillierte Kenntnis des vom Generator verwendeten Echtzeit-Frameworks erforderlich ist. Ist eine detaillierte statische Code-Analyse erforderlich, ist somit der Einsatz eines eigenen, besser analysierbaren Frameworks ratsam. Je nach den Mapping-Fähigkeiten des konkreten Werkzeuges kann dennoch teilweise Code generiert werden.

Modellanalyse

Durch den Einsatz eines geschlossenen Modells verlagert sich die Fehlersuche von der Quellcode- auf die Modellebene. Die Animation verhaltensbeschreibender Diagramme erleichtert die Fehlersuche, und prinzipiell werden dadurch auch modellbasierte Verifikationsverfahren wie Model-Checking oder RMA[7] ermöglicht, die in derzeit verfügbaren Werkzeugen jedoch noch nicht integriert sind.

Interoperabilität zwischen Werkzeugen

Dank der Standardisierung sind UML-Modelle prinzipiell zum Austausch zwischen verschiedenen Tools geeignet, was bspw. die Grundlage für den Einsatz komponentenbasierter Tool-Suites statt monolithischer Werkzeuge bildet und die Unabhängigkeit von einem bestimmten Werkzeug bewahren hilft.

In der Praxis unterscheiden sich die verschiedenen Werkzeuge jedoch hinsichtlich ihrer Implementation der per UML definierten Notationen und ergänzen sie teils um proprietäre Aspekte. Im Zusammenhang mit der Tatsache, dass ein einheitliches Format für Modell-Repositorien nicht Bestandteil der UML-Spezifikation ist, wird der Austausch zwischen verschiedenen Werkzeugen erschwert bis unmöglich. Da die aktuell verfügbaren Werkzeuge erst auf eine vergleichsweise kurze Entwicklungsgeschichte zurückblicken können, ist mit zunehmender Etablierung und Reife jedoch vermutlich mit einer Besserung zu rechnen.

Ausblick

Um die geschilderten Defizite der UML zu überwinden und damit ihre Eignung für eingebettete (Kfz-) Steuerungen zu verbessern, gibt es verschiedene Ansätze. *Automotive UML* ist eine in der Entwicklung befindliche Technologie für eingebettete Systeme in Fahrzeugen [HFG+00]. Darüber hinaus befinden sich Echtzeiterweiterungen der UML bereits im Standardisierungsprozess durch die OMG [SR98]. Weiterhin existieren Bestrebungen, die UML mit existierenden, zur Modellierung von Netzwerk- und Echtzeitaspekten *konzipierten* Sprachen und Methoden, wie etwa der *SDL* [Leb99, Pan99], zu verbinden. Darüber hinaus werden bereits Möglichkeiten der Integration von Model-Checking-Fähigkeiten und RMA-Tools untersucht. Daneben ist zu erwarten, dass sich das Sortiment am Markt verfügbarer, für eingebettete Echtzeitsysteme optimierter UML-Modeler weiter vergrößern wird.

In jedem Fall verspricht der Einsatz objektorientierter Verfahren beim Entwurf von Steuergeräten im Automobilsektor, der steigenden Komplexität besser als herkömmliche strukturierte Verfahren gerecht zu werden und die Effizienz der Entwicklungsprozesse zu steigern.

Literatur

[Bur97] Rainer Burkhardt. *UML – Unified Modelling Language: Objektorientierte Modellierung für die Praxis*. Addison-Wesley-Longman, Bonn, 1997.

[DC] DaimlerChrysler. Object-oriented real-time modeling contest. `http://www.automotive-uml.de/`.

[7] Rate-Monotonic Analysis

[Dou98a] Bruce Powel Douglass. Designing real-time systems with UML. *Embedded Systems Programming*, 3/98, 1998. CMP Media, Inc. http://www.embedded.com/.

[Dou98b] Bruce Powel Douglass. *Real-Time UML – Developing Efficient Objects for Embedded Systems*. Addison-Wesley, 1998.

[FMS00] João M. Fernandes, Ricardo J. Machado, and Henrique D. Santos. Modeling industrial embedded systems with UML. 8th International Workshop on Hardware/Software Codesign, 2000.

[Har87] David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[HFG+00] Peter Hofmann, Johannes Fasolt, Petra Geretschläger, Robert Sakretz, and Florian Wohlgemuth. Automotive UML – eine neue objektorientierte Entwicklungstechnik. *Elektronik*, Automotive 2000:88–95, 2000.

[JO+] Nicolai Josuttis, Bernd Oestereich, et al. Deutsche UML-Begriffe. http://system-bauhaus.de/uml/.

[Leb99] Philippe Leblanc. Developing reactive objects with SDL in UML projects. 1st OMER Workshop on Object-Oriented Modeling of Embedded Realtime Systems, 1999.

[Pan99] Ulf Pansa. SDL & UML, eine vollständige Lösung für die Entwicklung von Embedded Realtime Systems, von der OO-Analyse bis hin zur Target Verification. 1st OMER Workshop on Object-Oriented Modeling of Embedded Realtime Systems, 1999.

[RBP+93] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Objektorientiertes Modellieren und Entwerfen*. Carl Hanser & Prentice-Hall, München, Wien; London, 1993.

[SR98] Bran Selic and James Rumbaugh. Using UML for modeling complex real-time systems, 1998. http://www.objectime.com/.