Objektorientierte Entwicklung eingebetteter (Echtzeit-)Systeme mit UML?

Lutz Bichler, Andy Schürr
http://ist.unibw-muenchen.de/

Institut für Softwaretechnologie
Fakultät für Informatik
Universität der Bundeswehr, München
D-87577 Neubiberg

Zusammenfassung: In jüngster Zeit werden mehr und mehr objektorientierte Modellierungssprachen, Werkzeuge und Methoden für die Entwicklung eingebetteter Systeme in Betracht gezogen, die Varianten der Standardmodellierungssprache UML darstellen. In diesem Papier werden mit Realtime UML und UML/Realtime zwei solcher UML-Varianten kurz präsentiert. Im Anschluß daran wird ein neuer Vorschlag zur „natürlicheren“ Integration komponentenorientierter Modellierungskonzepte in UML skizziert. Insbesondere die diesem Vorschlag zugrundeliegende sehr einfache Erweiterung des UML-Metamodells weicht von bisherigen Vorschlägen zur Integration neuer komponentenorientierter Modellierungselemente in UML ab.

1. Einleitung

Software spielt in der Form eingebetteter (Echtzeit-)Systeme eine immer größer werdende Rolle in unserem täglichen Leben. Bis heute werden jedoch solche Systeme in vielen Branchen noch mit „ad hoc“-Methoden oder den strukturierten Methoden der 80-er Jahre (in der Softwaretechnik) entwickelt. So bahnt sich etwa im Automobilbau im Bereich der eingebetteten Systeme eine Revolution an, die sowohl die technische Realisierung solcher Systeme als auch deren Funktionalität und die Gestaltung ihrer „Benutzeroberflächen” betrifft [6]:

- Auf der *technischen Ebene* beobachtet man einen sich rapide beschleunigenden Trend von der Realisierung einzelner Funktion in Hardware hin zu deren Realisierung in Software, die Ablösung umfangreicher Kabelbäume durch mehr oder weniger leistungsstarke Bussysteme, den schrittweisen Übergang von der Assembler- und C-Programmierung zu Sprachen wie C++ oder gar Java sowie den Einsatz mächtigerer z.B. an CORBA angelehnter Middleware-Schichten.
- Auf der *funktionalen Ebene* nimmt im Gegenzug dazu die Anzahl und Komplexität der in Software realisierten Funktionen dramatisch zu. Als einige Beispiele seien hier nur bereits länger etablierte Sicherheitsfunktionen wie ABS (Anti-lock Brake System), ETC (Electronic Track Control) und ESP (Electronic Stability Program) sowie Komfortfunktionen wie Email, Navigationssystem, … genannt.
- Für ihre Bedienung werden völlig neu konzipierte *Benutzeroberflächen* eingesetzt. Zentrale Multifunktionsdisplays und Bediengeräte lösen bereits heute mehr und mehr auf einzelne Funktionen angepasste Bedien- und Anzeigeelemente ab.

Hinzu kommen Anforderungen wie Möglichkeiten (1) zur flexiblen (Um-)Verteilung von Softwarefunktionen auf eine geringere Anzahl leistungsfähigerer Steuergeräte, (2) zum „Nachladen“ neuer Softwarefunktionen oder (3) zur individuellen Zusammenstellung von Softwarekonfigurationen.

Dies hat zur Folge, dass die traditionell eingesetzten Vorgehensweisen und Werkzeuge zur Entwicklung eingebetteter (Echtzeit-)Software im Automobilbau (aber nicht nur dort) überholt sind und — ausgehend vom „State of the Art" der Softwaretechnik — neue Vorgehensweisen entwickelt werden, die die *integrierte Entwicklung* eingebetteter Systemsoftware im klassischen Sinne und die Entwicklung neuer Komfortfunktionen mit komplexen multimedialen Benutzeroberflächen unterstützen. Hierfür eignet sich aus folgenden Gründen das *objektorientierte (OO-)Paradigma* (siehe auch [8] und [9]):

- OO-Programmiersprachen und -methoden haben sich bei der Konstruktion interaktiver Benutzeroberflächen bewährt.
- Bewährte Konzepte zum Entwurf von Softwaresystemen werden durch das Prinzip der Datenabstraktion und Vererbung sowie (mit Einschränkungen) durch das Prinzip der komponentenorientierten Softwareentwicklung unterstützt.
- Die sich auf breiter Front in vielen Anwendungsbereichen für die Analyse und das Design von Softwaresystemen durchsetzende Modellierungssprache UML beruht auf dem OO-Paradigma.

Wie bereits der Titel dieses Vortrags verspricht, werden wir uns im folgenden mit der Eignung der „*Unified Modeling Language = UML*" [20] als dem „De-fakto"-Standard einer objektorientierten Modellierungssprache für die Entwicklung eingebetteter Systeme auseinandersetzen. Der folgende Abschnitt 2 wird zunächst das hier verwendete Beispiel der Steuerung einer (aus Platzgründen sehr einfachen) Klimaanlage einführen und einen Überblick über einen UML-basierten Entwicklungsprozess geben. Im nachfolgenden Abschnitt 3 werden wir uns kurz mit *Realtime UML*, einer durch das Werkzeug Rhapsody® [10] unterstützten Variante von UML auseinandersetzen, die speziell auf die Entwicklung eingebetteter Echtzeitsysteme zugeschnitten ist.

Ausgehend von einer Hauptschwäche von Realtime UML, der mangelhaften Unterstützung für die Entwicklung wiederverwendbarer Softwarekomponenten, werden wir uns dann in Abschnitt 4 mit *UML/Realtime* auseinandersetzen. UML/Realtime wird von dem Werkzeug Rational Rose/Realtime® [14] unterstützt und bietet insbesondere Erweiterungen von UML für die komponentenorientierte Modellierung reaktiver verteilter Systeme an. Ausgehend von den dabei festgestellten Schwächen von Rational Rose/Realtime® und verwandten Ansätzen zur UML-basierten Modellierung eigenständiger Komponenten werden wir schließlich in Abschnitt 5 unseren Vorschlag für eine konservative(re) Erweiterung von UML zur komponentenorientierten Modellierung vorstellen und im folgenden Abschnitt 6 die dafür notwendigen Erweiterungen des UML-Metamodells skizzieren. Abschnitt 7 fasst die in diesem Aufsatz ausgebreiteten Überlegungen zusammen und diskutiert eine Reihe noch offener Punkte.

Aus Gründen der Vollständigkeit sei darauf hingewiesen, dass

- es neben Rhapsody® und Rational Rose/Realtime® noch einige weitere UML-basierte Softwareentwicklungswerkzeuge (CASE-Tools) mit ähnlichem Funktionsumfang gibt, wie etwa Artisan Software's Real-time Modeler [1],
- aus Platzgründen andere einschlägige Modellierungssprachen und ihre Werkzeuge wie insbesondere SDL [4] in diesem Beitrag weitgehend ignoriert werden
- und insbesondere auf im betrachteten Anwendungsfeld mehr oder weniger weitverbreitete nicht objektorientierte Werkzeuge wie MATLAB® [13], Statemate® [11], TITUS® [5], etc. hier nicht eingegangen werden kann.

Aus naheliegenden Gründen verzichten wir auch an dieser Stelle auf die oft sehr emotionsbeladene Diskussion der zu einer bestimmten Modellierungsart besonders geeigneten Implementierungssprache bzw. auf die oft gerne geführte Diskussion, ob höhere Programmiersprachen wie ADA mit ihren wohldefinierten und praxiserprobten Modularisierungskonzepten nicht wesentlich besser als grafische Modellierungssprachen wie UML für das *Design* von Softwaresystemen geeignet sind. Hierzu sei etwa auf die Diskussion in [18] oder die Darstellung der Schwächen des UML-Paketkonzeptes in [16] verwiesen.

Schließlich sei der Leser noch darauf hingewiesen, dass ein ausführlicherer Vergleich von Realtime UML und UML/Realtime in [2] zu finden ist, während in [3] weitere Überlegungen zu datenflußorientierten Erweiterungen des hier präsentierten Komponentenkonzeptes publiziert wurden. Für eine ausführlichere Kritik der Erweiterungen des UML-Metamodells für UML/Realtime und entsprechende Gegenvorschläge, die eine Vorstufe der hier vorgestellten Erweiterung des UML-Metamodells darstellen, wird der Leser auf [15] verwiesen.

2. Objektorientierte Entwicklung mit UML

Im folgenden werden wir die Steuerung einer (Auto-)Klimaanlage als fortlaufendes Beispiel verwenden, um die Vor- und Nachteile verschiedener Ansätze zur Modellierung eingebetteter Systeme zu diskutieren. Unsere hypothetische *stark vereinfachte Klimaanlage* besteht aus einer Kühleinheit, einem Heizungsaggregat, einem Thermostat und einem Temperatursensor. Das Heizungsaggregat zerfällt in zwei Unterkomponenten, dem Standardaggregat, das im laufenden Betrieb die gewünschte Temperatur aufrecht erhält, und einer Zusatzstufe, die nach dem Kaltstart gegebenenfalls zugeschaltet wird, um so schneller die gewünschte Temperatur zu erreichen. Einige der Anforderungen an die Steuerung einer solchen Klimaanlage sehen in etwa wie folgt aus:

(1) Das für die Steuerung der Heizungsaggregate zuständige Subsystem besitzt nebem dem Zustand off zwei weitere Zustände level1 und level2. Im Zustand level1 ist das Standardheizungsaggregat aktiviert, im Zustand level2 auch die Zusatzstufe.

(2) Nach einigen Minuten Aktivität wird die Zusatzstufe automatisch abgeschaltet, die Steuerung geht also von Zustand level2 in Zustand level1 über.

(3) Bestimmte Ereignisse wie das Abziehen des Zündschlüssels (ignitionKeyOff) führen zur sofortigen Deaktivierung aller Aktivitäten der Klimaanlage, andere Ereignisse (wie z.B. ignitionKeyCold) zur Deaktivierung der Klimaanlage nach einigen Minuten.

(4) Schließlich gibt es eine ganze Gruppe von Ereignissen, die nur temporäre Deaktivierungen der Klimaanlage auslösen.

Die objektorientierte Entwicklung der entsprechenden Software mit UML kann im einfachsten Fall wie folgt durchgeführt werden:

(1) Zunächst werden die gewünschten Funktionen in Form eines „normalen“ *Lastenheftes* festgehalten, das sich auch für die Kommunikation mit Personen eignet, die nicht mit Modellierungsprachen vertraut sind.

(2) Anschließend wird die im Lastenheft skizzierte Funktionalität in eine Reihe von *Anwendungsfällen* (Use Cases) zerlegt. Jeder dieser Anwendungsfälle beschreibt eine gewünschte Funktion des betrachteten Systems mit ihren Auslösern, Nebenbedingungen, Ausnahmefällen, etc. Als Beschreibungsmittel werden immer gleich strukturierte Texte und die Anwendungsfalldiagramme der UML eingesetzt.

(3) Im nächsten Schritt bzw. teilweise parallel zur Erfassung der Anwendungsfälle wird das eingebettete System als „Initial System Model” aus Hardware-Sicht modelliert. Hierzu eignen sich die *Klassendiagramme* der UML und ihre *Verteilungsdiagramme* (Deployment-Diagramme) im beschränkten Maße. Aus diesem Grunde bieten Werkzeuge wie der Real-time Modeler [1] für diesen Zweck zusätzliche Diagrammarten an.

(4) In der Regel wird das „Initial System Model” später durch ein feineres „Static Software Model” ersetzt, das eine verfeinerte Form der zuvor entwickelten Klassendiagramme darstellt. Spätestens zu diesem Zeitpunkt werden meist *Paketdiagramme* zur Zerlegung eines Entwurfes in überschaubare Teildiagramme eingesetzt.

(5) Schließlich wird das statische Softwaremodell um dynamische Aspekte ergänzt. Zu jeder Klasse (Komponente) wird ein Automat (*Statechart*) definiert, der die Reaktion ihrer Instanzen auf eintreffende Nachrichten (Ereignisse) festlegt. Da UML [20] zur Zeit noch keine standardisierte Aktionssprache besitzt, werden zusätzlich Programmfragmente der später eingesetzten Programmiersprache zur Beschriftung von Transitionen verwendet, falls die erstellten Statecharts für Simulationszwecke oder Rapid-Prototyping-Zwecke ausführbar sein müssen.

(6) Bislang nicht aufgeführt wurde der Einsatz von *Sequenzdiagrammen* zur präziseren Beschreibung einzelner Anwendungsfälle. Aufgrund vieler Schwächen dieser Diagrammart lohnt sich unserer Meinung nach ihr Einsatz in diesem Anwendungsbereich meist nur dann, falls Werkzeuge verwendet werden, die auf Basis solcher Sequenzdiagramme Teilaktivitäten des Testens automatisieren.

(7) Ebenfalls hier nicht vorgeschlagen wird der Einsatz von *Aktivitätsdiagrammen*, etwa für das grafische „Ausprogrammieren“ einzelner Teilaktivitäten. Anders als beispielsweise bei der Beschreibung betrieblicher Abläufe mit Aktivitätsdiagrammen erscheint uns ihr Nutzen im Vergleich zur Verwendung von Programmiersprachenfragmenten fraglich.

Zusätzlich zu den oben diskutierten Diagrammarten bietet die UML weitere Diagrammarten (Einsatz- und Komponenten-Diagramme) für den systematischen Übergang vom prinzipiell ablauffähigen Systemmodell zur tatsächlichen Implementierung oder einem „Rapid Prototype“ an, auf die wir aus Platzgründen jedoch hier nicht näher eingehen können.

3. Objektorientierte Entwicklung mit Realtime UML

Verwendet man für die Umsetzung des oben skizzierten Entwicklungsprozesses neben den üblichen „Requirements-Engineering“-Werkzeugen das Modellierungswerkzeug Rhapsody® mit seiner Realtime-Variante der UML [10], so konzentriert sich die Modellierungsarbeit auf die Erstellung von Klassendiagrammen und die Erstellung von Automaten (Statecharts) mit möglicherweise nebenläufigen Teilautomaten (and-States). Anders als bei den meisten UML-Werkzeugen unterstützt Rhapsody® die Darstellung von Kompositionsbeziehungen zwischen Klassen durch Schachtelung ihrer Darstellungen. Dies wurde in Abbildung 1 ausgenutzt, um das Gesamtsystem, das durch eine Instanz der Klasse AirConditionControl repräsentiert wird, in jeweils eine Instanz der vier Klassen TemperatureSensor, Thermostat, CoolingSystem und HeatingSystem zu zerlegen. Nicht ohne Grund sind zwischen den wiederverwendbaren Klassen TemperatureSensor und Thermostat und den für dieses System spezifischen Klassen keine Assoziationen eingetragen. Die Verwendung solcher Assoziationen zur Modellierung von Verbindungen, die nur im Kontext eines bestimmten umfassenden Systems (AirConditionControl) existieren sollen, wirft nämlich einige Probleme auf.

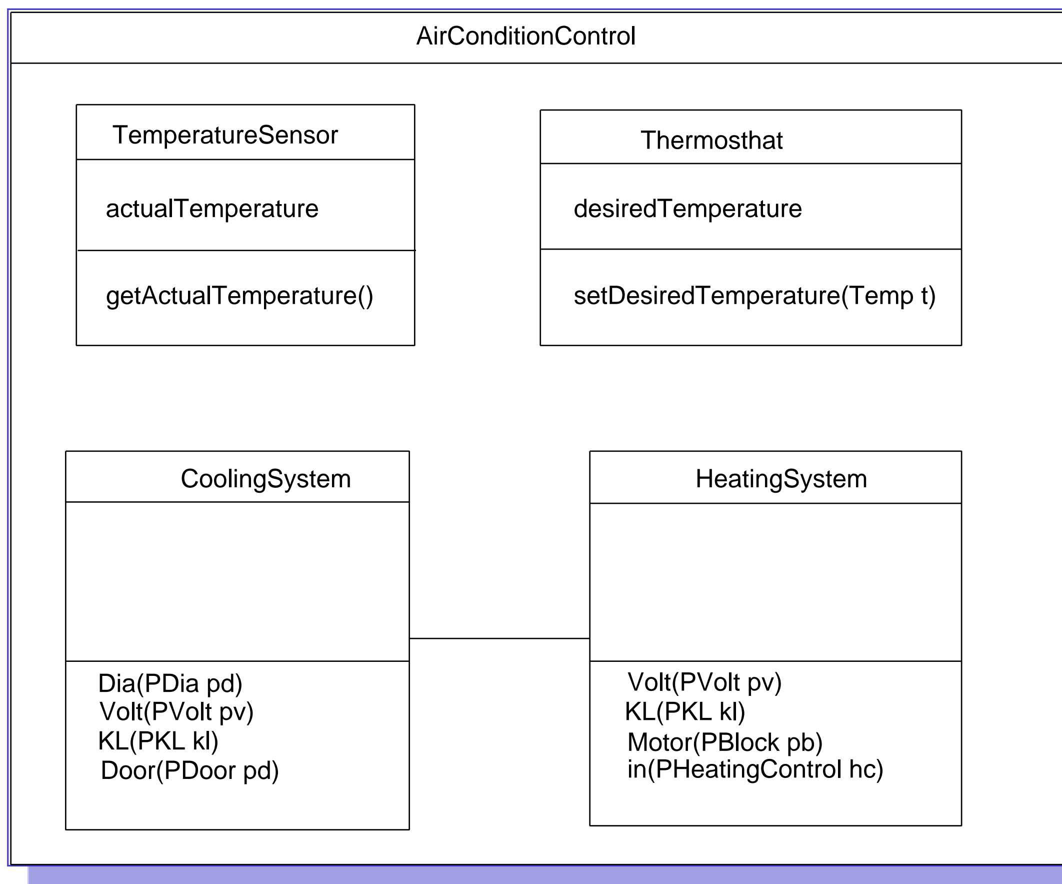


Abbildung 1: Realtime-UML-Klassendiagramm der Klimaanlagensteuerung

Nehmen wir beispielsweise einmal an, dass die eigentlich existierenden bidirektionalen Beziehungen zwischen den für dieses Beispiel spezifischen Klassen CoolingSystem und HeatingSystem und den allgemein verwendbaren Klassen TemperatureSensor und Thermostat als Assoziationen in das Diagramm von Abbildung 1 zusätzlich eintragen würden. Dann wäre völlig unklar, ob die Implementierungen der Klassen TemperatureSensor und Thermostat immer Verweise auf Objekte der Klassen CoolingSystem und HeatingSystem enthalten sollten oder ob es sich um Verweise handelt, die nur im betrachteten Kontext von Interesse sind. Zudem wäre unklar, wie man diese Verweise in der (generierten) Implementierung ggf. so realisiert, dass die Klassen TemperatureSensor und Thermostat völlig unabhängig von den Klassen CoolingSystem und HeatingSystem verwendbar bleiben. Im folgenden Abschnitt 4 werden wir sehen, wie in UML/Realtime® solche Probleme durch die Einführung von „Ports“ als eine neue Form von UML-Klassenschnittstellen gelöst werden; eine andere Lösung des skizzierten Problems, die näher am UML-Standard bleibt, wird in Abschnitt 5 skizziert.

Doch nun zurück zur Modellierung des Verhaltens unserer Klimaanlage. Das Werkzeug Rhapsody® hebt sich hierbei von den meisten anderen UML unterstützenden Modellierungswerkzeugen dadurch ab, dass es die Ausführung nahezu aller im Standard festgelegten Statechart-Konstrukte erlaubt. Somit liegt es nahe, das gewünschte Verhalten der Klasse HeatingSystem mit einem einzigen Statechart zu beschreiben, das aus drei parallel geschalteten Teilautomaten besteht (siehe Abbildung 2). Das Normalverhalten der Heizungskomponenten der Klimaanlage wird durch den obersten Teilautomaten erfasst, die endgültige oder temporäre Abschaltung auf-

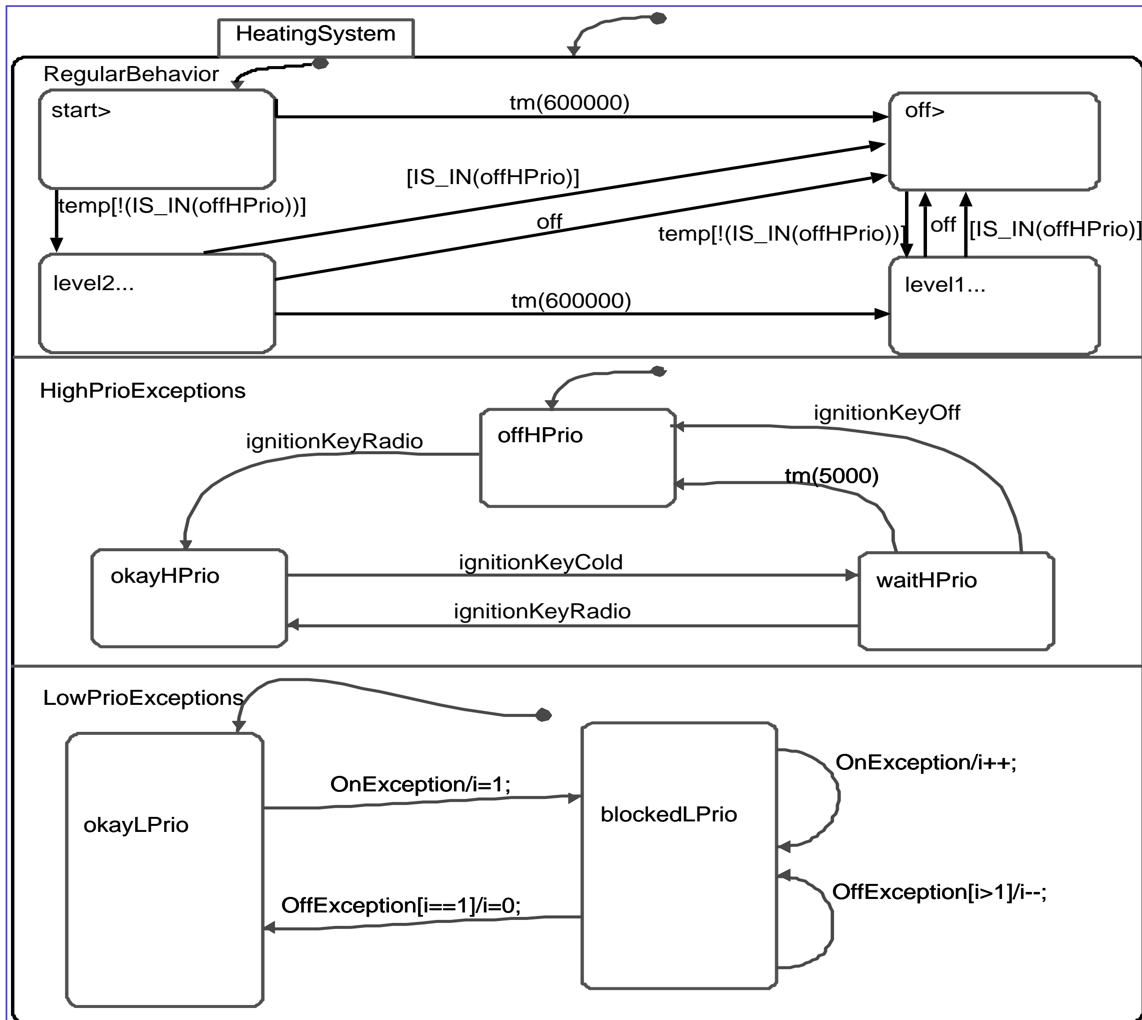


Abbildung 2: Realtime-UML-Statechart eines Teils der Klimaanlagensteuerung

grund bestimmter Ereignisse wird durch die beiden anderen Teilautomaten geregelt. Der Leser sei an dieser Stelle darauf hingewiesen, dass Schaltbedingungen der Form IS_IN(Zustand) im obersten Teilautomaten Transitionen auslösen, wenn der mittelere Teilautomat seinen Zustand wechselt. So wechselt beispielsweise der oberste Teilautomat von dem Zustand level2 in den Zustand off, wenn mittlere Teilautomat (vom Zustand waitHPrio nach 5000 Zeiteinheiten in den Zustand offHPrio wechselt. Zusätzlich werden die beiden Zustände level1 und level2 des obersten Teilautomaten durch eigene Automaten verfeinert, die auf Zustandsänderungen des untersten Teilautomaten reagieren und temporäre Abschaltungen regeln. Auf die Darstellung dieser Teilautomaten wird hier aus Platzgründen verzichtet, für weitere Details sei der Leser auf [2] verwiesen.

Die Hauptnachteile des damit skizzierten Modellierungsstils bestehen darin, dass die Zusammenhänge der eingeführten Klassen nicht auf den ersten Blick klar werden, die Erstellung wirklich einzeln wiederverwendbarer Klassen (Komponenten) eine hohe Disziplin beim Modellierer erfordert und auch die Wechselwirkungen zwischen den Teilautomaten des Statecharts einer Klasse oft nicht auf den ersten Blick ersichtlich sind.

4. Komponentenorientierte Entwicklung mit UML/Realtime

UML/Realtime, der UML-Dialekt, der von dem Werkzeug Rational Rose/Realtime® unterstützt wird [14], erzwingt eine deutlich andere Modellierung unserer Klimaanlage als im vorigem Abschnitt gezeigt. Zunächst einmal erlaubt UML/Realtime mit der neu eingeführten Diagrammart der *Strukturdiagramme* eine bessere Zerlegung eines Systems in wiederverwendbare Komponenten und eine klarere Modellierung von Anknüpfungspunkten(Schnittstellen) an sogenannte Kapselklassen. So sieht man beispielsweise in Abbildung 3, dass unsere gesamte Klimaanlagensteuerung drei Schnittstellen (dargestellt als weiße Quadrate) zu anderen Teilsystemen besitzt, um Informationen über (1) den Zustand der Fahrzeugtüren, (2) des Zündschlüssels und der (3) Spannungsversorgung zu erhalten. Zudem sieht man genau, wie die verschiedenen Teilkomponenten des Gesamtsystems über interne Schnittstellen miteinander kommunizieren.

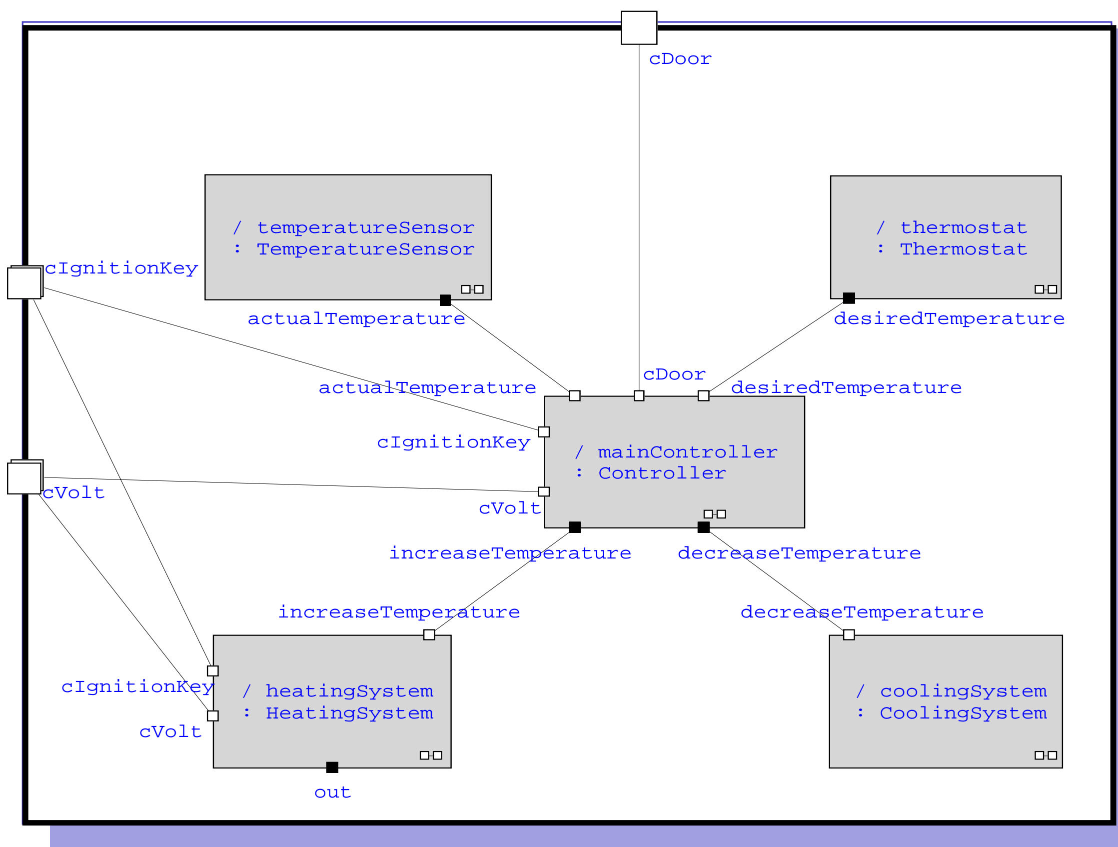


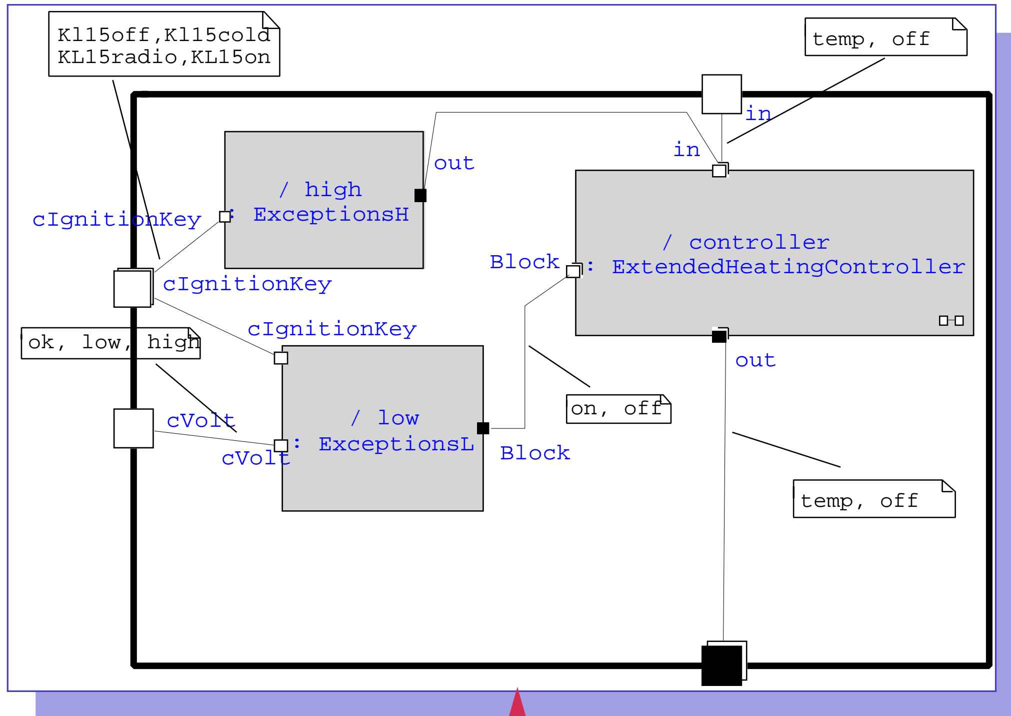
Abbildung 3: Oberstes UML/Realtime-Diagramm der Klimaanlagensteuerung

Die neue Diagrammart der *Strukturdiagramme* wurde von ROOM [17] übernommen und als Variante der Kollaborationsdiagramme in UML integriert. Kapselklassen, wie in Abbildung 3 verwendet, unterscheiden sich in folgenden Punkten von „normalen“ Klassen:

- *Kapselklassen* sind (re-)aktiv und reagieren auf das Eintreffen asynchroner Botschaften, während normale Klassen passiv ihre Dienste in Form synchroner Methodenaufrufe anbieten.
- Kapselklassen besitzen anstelle einer Schnittstelle öffentlich sichtbarer und damit aufrufbarer Methoden eine beliebige Anzahl von *Ports*. Mit diesen Ports verknüpfte Protokolldefinitionen regeln, welche Botschaften über einen Port eintreffen (können) und welche Botschaften über denselben Port verschickt werden können.

- Instanzen von Kapselklassen werden nicht durch Links (Assoziationsinstanzen) miteinander verbunden, sondern durch *Kommunikationskanäle*, die zueinander passende Ports miteinander verbinden.
- Instanzen von Kapselklassen dürfen Instanzen von Kapselklassen und Instanzen normaler Klassen enthalten, aber Instanzen normaler Klassen dürfen nur Instanzen normaler Klassen enthalten oder auf sie verweisen.

a) Kapseldiagramm (Strukturdiagramm):



zwei Sichten

b) Klassendiagramm:

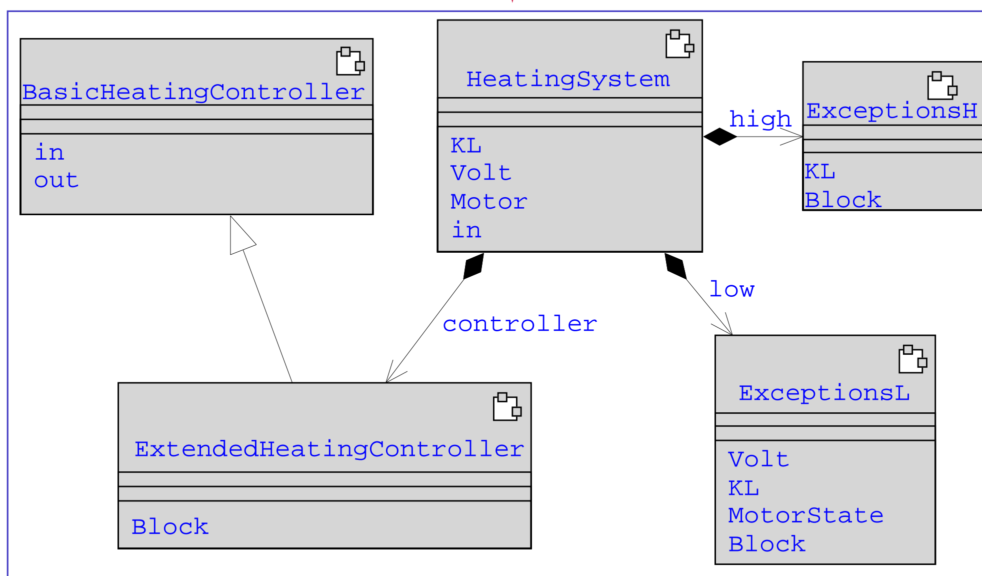


Abbildung 4: Teilansichten des UML/Realtime-Modells

Die fehlende Unterstützung von parallelen and-Zuständen erfordert, dass die in Abbildung 1 für die Modellierung mit Realtime UML eingeführte und in Abbildung 3 wieder eingesetzte Klasse HeatingSystem(Controller) in drei Klassen zerlegt wird, deren Statecharts (Automaten) jeweils ein Teildiagramm des komplexen Statecharts von Abbildung 2 realisieren. Das Zusammenwirken jeweils einer Instanz dieser drei neuen Klassen wird in Abbildung 4a) in Form eines sogenannten Kapsel- oder Strukturdiagramms dargestellt.

Damit unterstützt UML/Realtime die Zerlegung der Steuerungssoftware in *wiederverwendbare Komponenten*, die sowohl die angebotenen Dienste für andere als auch die benötigten Dienste von anderen Softwarekomponenten durch Portprotokolle festlegen. Das Strukturdiagramm von Abbildung 3 zeigt uns beispielsweise, dass es einen Hauptsteuerungsbaustein gibt, der Daten von einem Temperatursensor und Thermostat einliest und in entsprechende Anforderungen für Instanzen der Klasse HeatingSystem und CoolingSystem unwandelt. Eine Instanz der Klasse HeatingSystem (siehe Abbildung 4a) besteht wiederum aus einem Hauptsteuerungsbaustein ExtendedHeatingController, der für die Steuerung der Heizaggregate zuständig ist. Er erhält vorverarbeitete Informationen über Ausnahmezustände, die eine (temporäre) Deaktivierung der Heizungsaggregate erfordern, von jeweils einer Instanz der Kapselklassen ExceptionsH(igh) und ExceptionsL(ow).

Wie in dem zusätzlich existierenden Klassendiagramm von Abbildung 4b) zu sehen, ist die Kapselklasse ExtendedHeatingController eine Spezialisierung der Kapselklasse BasicHeatingController. Auf diese Weise verdeutlicht das erstellte Modell eine Trennung zwischen dem Grundverhalten der modellierte Steuerung (in BasicHeatingController) und der zusätzlichen Berücksichtigung von Spezialfällen (in ExtendedHeatingController). Für eine Diskussion genauerer Details des erstellten Modells sei der Leser auf [2] verwiesen. Dort wird nicht nur erläutert, wie in UML/Realtime durch Spezialisierung abgeleitete Kapselklassen die Schnittstellen und das Verhalten ihrer Oberklasse verfeinern können, sondern auch das Konzept der Ports genauer erklärt. Zudem werden dort verschiedene Modellierungsstile miteinander verglichen, die zusammen mit UML/Realtime verwendet werden können.

5. Komponentenorientierte Entwicklung mit UML

Komponentenorientierte Softwareentwicklung verfolgt das Ziel, komplexe Softwaresysteme aus (relativ) einfachen, wiederverwendbaren Komponenten zusammenzusetzen. Um Wiederverwendbarkeit zu unterstützen, müssen sowohl die Schnittstellen zu den Diensten, die von einer Komponente angeboten werden, als auch die Schnittstellen, die von einer Komponente benötigt werden, exakt spezifiziert werden. Eine komponentenorientierte Modellierungssprache muss daher die Definition von *Import- und Exportschnittstellen* einer Komponente erlauben. Um komplexe Komponenten oder Systeme aus einfachen Komponenten zusammensetzen zu können, muss außerdem das Ineinanderschachteln von Komponenten modellierbar sein.

Trotz dieser großen Bedeutung der *komponentenorientierten Modellierung auf logischer Ebene* wird sie von Standard-UML bzw. Realtime UML kaum unterstützt. Die UML-Komponentendiagramme erlauben nur die Definition binärer Systemkomponenten mit sehr einfachen Exportschnittstellen, nicht aber logischer Komponenten. Diese Komponenten können zudem keine unverbundenen Importschnittstellen besitzen. Außerdem können UML-Komponenten nur aus Klassen und nicht aus anderen Komponenten zusammengesetzt werden. Auch die in Realtime UML zum Einsatz kommende Schachtelung von Klassen zusammen mit der extensiven Verwendung von eigenständigen Interface-Klassen löst das skizzierte Problem nicht. Glei-

ches gilt für das in der Literatur vorgeschlagene Konzept zur Definition von logischen Komponenten mit Hilfe spezieller Klassen- und Paketdiagramme. Diese etwa in [12] skizzierte Lösung erfordert sehr viel Modellierungsdisziplin und erlaubt zudem keine klare Beschreibung von Import-Schnittstellen.

Für die in Abschnitt 4 vorgestellten UML/Realtime-Strukturdiagramme mit ihren Kapselklassen gilt, dass sie zwar die logische Komponentenmodellierung mit Import- und Exportschnittstellen unterstützen, aber sich auf die Definition aktiver Komponenten mit ereignisbasiertem Nachrichtenaustausch konzentrieren. Unglücklich ist dabei aus unserer Sicht die harte Trennung zwischen dem Schnittstellenkonzept von Standard-UML (mit Lollipops) für normale Klassen und dem portbasierten Schnittstellenkonzept von UML/Realtime für Kapselklassen. Damit verhindert UML/Realtime den fließenden Übergang von der Modellierung mit “traditionellen” Klassendiagrammen zur komponentenbasierten Modellierung sowie die schrittweise Verfeinerung von Klassendiagrammen um Entwurfsentscheidungen wie „passive versus aktive Klassen” oder „synchrone Operationsaufrufe versus asynchrone Ereignisversendung” etc.

Anhand des folgenden einfachen Beispiels, das einen kleinen, bislang nicht diskutierten Teilaspekt der Klimanlagensteuerung behandelt, wird der fließende Übergang von der objektorientierten zur komponentenorientierten Modellierung dargestellt, wie wir ihn anstreben:

> *Ein Schalter besitzt eine LED, die seinen Zustand anzeigt. Der Schalter meldet das Signal „pressed” an einen Controller, wenn er betätigt wird. Der Controller schaltet über die Signale „on” bzw. „off” die LED am Schalter ein bzw. aus und aktiviert bzw. deaktiviert einen hier nicht weiter betrachteten Aktor.*

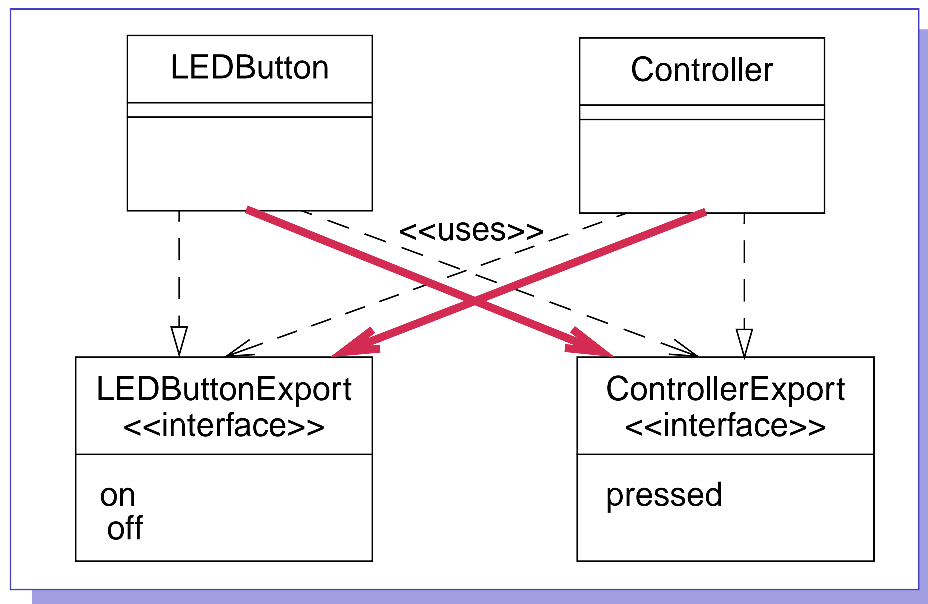


Abbildung 5: UML-Klassendiagramm mit kaum wiederverwendbaren Klassen

Die „rein“ objektorientierte Modellierung des Beispiels in Abbildung 5 kann durch direktes Übersetzen der realen Objekte in entsprechende Modellobjekte erfolgen:

- Es werden zwei Klassen LEDButton und Controller definiert, die den Schalter bzw. den Controller abbilden.

- Die Klasse LEDButton besitzt eine Schnittstelle, an der die Operationen on und off angeboten werden.
- Die Klasse Controller besitzt eine Schnittstelle, die eine Operation pressed bereitstellt.
- Damit jedes LEDButton-Objekt “sein” Controller-Objekt kennt und umgekehrt, muss man zwischen den beiden Klassen LEDButton und Controller eine Assoziation einführen.

Mit der Einführung einer direkten Verbindung (Assoziation) zwischen den beiden betrachteten Klassen wird die Forderung nach ihrer Wiederverwendbarkeit in anderen Konstellationen verletzt. Eine komponentenorientierte Modellierung des Beispiels erfordert nämlich die Auftrennung der direkten Beziehung (Assoziation) zwischen LEDButton- und Controller-Objekten. In einem ersten Versuch werden dafür in Abbildung 6 die Schnittstellen LEDButtonExport und ControllerExport definiert. Die Klasse LEDButton implementiert die Schnittstelle LEDButtonExport, benutzt (importiert) die Schnittstelle ControllerExport und referenziert ein Objekt, das die Schnittstelle ControllerExport realisiert. Umgekehrt implementiert die Klasse Controller die Schnittstelle ControllerExport, benutzt die Schnittstelle LEDButtonExport und referenziert ein Objekt, das die Schnittstelle LEDButtonExport realisiert. Leider wird bei der gewählten Darstellung der Import- und Export-Beziehungen in Abbildung 5 nicht klar, dass wir in einem konkreten Anwendungsfall Instanzen der Klasse LEDButton nur mit Instanzen der Klasse Controller kombinieren wollen (und nicht mit beliebigen Implementierungen der Schnittstellen LEDButtonExport und ControllerExport).

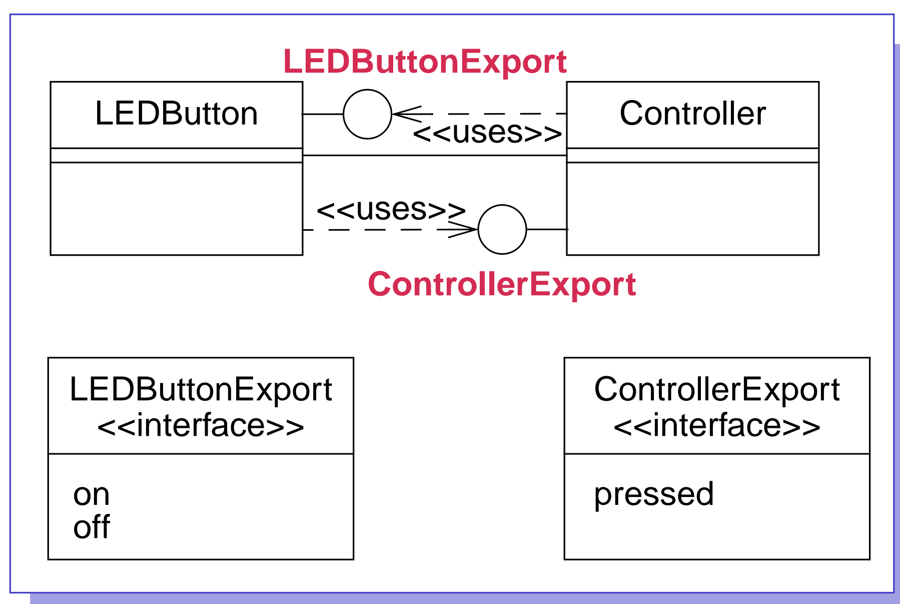


Abbildung 6: “Lollipop”-Notation für Definition wiederverwendbarer(er) Klassen

In Standard-UML kann das Anbieten einer Schnittstelle aber auch durch einen Export-Lollipop modelliert werden. Die Benutzung dieser Schnittstelle durch eine andere Klasse wird durch eine Benutzt-Beziehung klarer und kompakter dargestellt. Eine Modellierung des Beispiels mit diesen Elementen zeigt Abbildung 6. Diese zeigt aber, dass bei einer Modellierung in Standard-UML wiederum eine direkte Beziehung zwischen den beiden Klassen besteht (da zwischen ihren Lollipop-Schnittstellen die benötigte Assoziation nicht eingetragen werden kann) und der Zusammenhang zwischen den beiden benutzten Exportschnittstellen nicht deutlich wird.

Um diese Schwachstellen der Modellierung zu beseitigen, wird im folgenden ein neues Modellelement eingeführt und dessen Abbildung auf Standard-UML angegeben. In der Regel stellen Komponenten nicht nur Schnittstellen zur Benutzung der von ihnen angebotenen Ressourcen bereit (Exportschnittstellen), sondern benötigen auch Schnittstellen für den Zugriff auf Ressourcen anderer Komponenten (Importschnittstellen). Zur Modellierung solcher Importschnittstellen wird das Gegenstück zum Export-Lollipop aus Standard-UML, der schwarz ausgefüllte *Import-Lollipop*, eingeführt.

Der Einsatz dieses Modellierungselementes führ zu der in Abbildung 7 dargestellten Konstruktion. Die Klasse LEDButton bietet die als weißer Kreis dargestellte Schnittstelle LEDButtonExport an und benötigt die als schwarzer Kreis dargestellte Schnittstelle ControllerExport, während die Klasse Controller die Schnittstelle ControllerExport anbietet und die Schnittstelle LEDButtonExport benötigt. Die durchgezogenen Pfeile von den Import- zu den Export-Lollipops sind eine Abkürzung für eine sonst gestrichelte reine Benutzt-Beziehung sowie eine gerichtete (unidirektionale) Assoziation. Damit haben wir eine Modellierungsvariante vorgestellt, bei der die beiden Beispielklassen separat voneinander wiederverwendbar sind und nur über Schnittstellen miteinander verbunden werden.

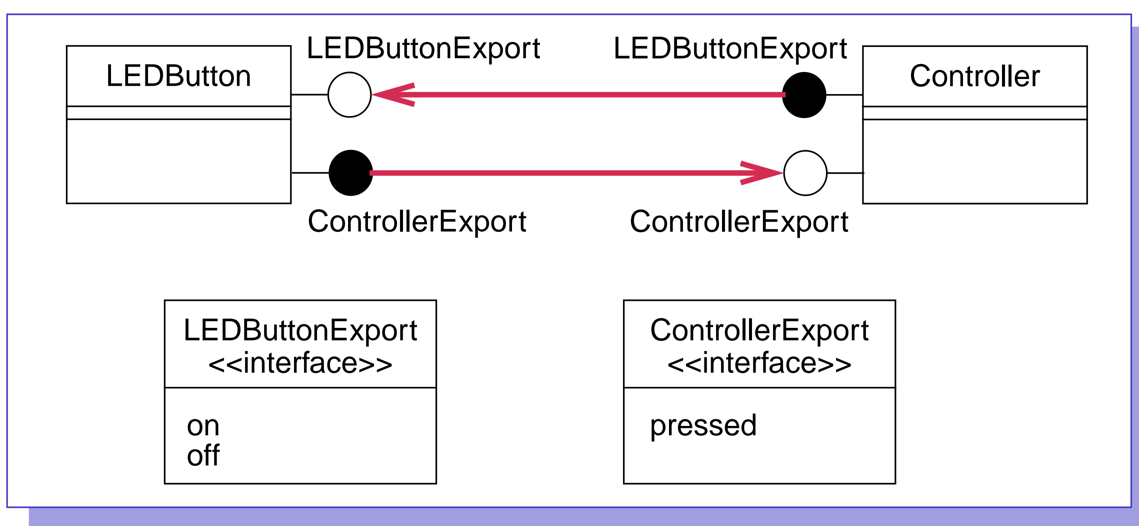


Abbildung 7: Symmetrische Modellierung mit Export- und Import-Lollipops.

Im übrigen sei der Leser darauf hingewiesen, dass genau genommen Abbildung 7 nur die Anwendung und Verschaltung unserer beiden Beispielklassen in einem nicht angegebenen Systemkontext darstellt. Es gibt also neben dem gezeigten Diagrammausschnitt andere Diagramme, in denen diese Klassen mit ihrem Innenleben und ihren Schnittstellen im Detail beschrieben werden.

Um die Zusammenhänge zwischen Schnittstellen besser darstellen zu können, werden zusätzlich *gemischte Import-/Exportschnittstellen* eingeführt, die den Port-Protokollen in UML/Realtime entsprechen [14]. Wie im hier verwendeten Beispiel ist es nämlich häufig erforderlich, dass zwei Komponenten paarweise zueinander passendede (konjugierte) Import- und Exportschnittstellen besitzen, um miteinander verwendet werden zu können. In unserem Beispiel muss eine Controller-Komponente sowohl die Schnittstelle LEDButtonExport importieren als auch die Schnittstelle ControllerExport anbieten, um sinnvoll mit der Komponente LEDButton zusammenzuarbeiten.

Zur Modellierung gemischter Import-/Exportschnittstellen wird eine Interface-Klasse eingeführt, die einen ‹‹export››- und einen ‹‹import››-Abschnitt enthält. Die Interface-Klasse umfasst alle Operationen der enthaltenen Import- und Exportschnittstellen. Bietet eine Klasse die im ‹‹export›-Teil angegebenen Operationen an und benötigt die im ‹‹import››-Teil angeführten Operationen, so wird dies durch einen Lollipop angegeben, dessen obere Hälfte weiß und dessen untere Hälfte schwarz ist. Anbieten der im ‹‹import››-Teil angegeben Operationen und Benötigen der im ‹‹export››-Teil angeführten Operationen wird durch einen sogenannten „konjugierten“ Lollipop dargestellt, dessen obere Hälfte schwarz und dessen untere Hälfte weiß ist. Abbildung 8 zeigt die Modellierung des Beispiels unter Verwendung gemischter Import-/Exportschnittstellen.

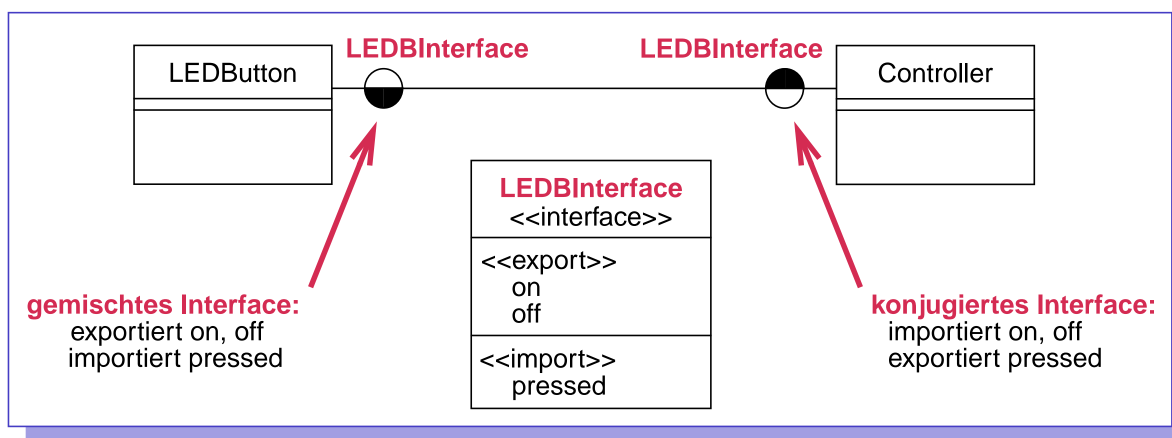


Abbildung 8: Gemischte Klassen-Schnittstellen (wie Ports in UML/Realtime)

Im Unterschied zu einer Klasse kann eine Komponente auch mehrere *benannte Schnittstellen* besitzen und auch dieselbe Schnittstelle — unterschiedlich benannt — mehrfach anbieten oder benötigen. Mehrere Schnittstellen werden durch verschiedene Import- oder Export-Lollipops modelliert. Zur Modellierung des mehrfachen Vorhandenseins einer Schnittstelle an einer Komponente gibt es mehrere Möglichkeiten. Ein Lollipop kann mit einer Kardinalität versehen werden, die die Anzahl der Schnittstellen angibt. Dies ist für Exportschnittstellen geeignet, da diese i.d.R. unabhängig von einer bestimmten Instanz benutzt werden und anderenfalls einer Operation eine Referenz auf den Aufrufer als Parameter übergeben wird.

Bei der Modellierung einer endlichen Anzahl von identischen Importschnittstellen muss dagegen beachtet werden, dass diese möglicherweise im Code explizit angesprochen werden sollen. In diesem Fall ist es besser, jede Schnittstelle durch einen eigenen Lollipop zu modellieren (Abbildung 9). Dies hat zudem den Vorteil, dass für die einzelnen Schnittstellen „sprechende” Namen verwendet werden können, was die Verständlichkeit des Modells verbessert.

Die folgende Abbildung 9 zeigt im übrigen auch, wie bei der Deklaration einer Klasse Subsystem deren Innenleben durch Verschalten angewandter Auftreten anderer Klassen definiert wird. Es ist das Kennzeichen der *rollenorientierten Modellierung*, dass ein Diagramm mehrere Auftreten derselben Klassen in unterschiedlichen Rollen enthalten kann. In unserem Beispiel enthält jede Instanz der Klasse Subsystem genau eine Instanz der Klasse Controller und zwei Instanzen der Klasse LED. Die beiden Instanzen der Klasse LED können von ihrer Subsystem-Instanz über die beiden Rollennamen L1 und L2 getrennt voneinander angesprochen werden. Die Controller-Instanz C kennt hingegen — wie bei den Kapseldiagrammen von UML/RT —

die “benachbarten” LED-Instanzen und ihre Rollennamen nicht direkt. Sie kann sie nur über die unterschiedlich benannten Importschnittstellen I1 und I2 getrennt voneinander ansprechen.

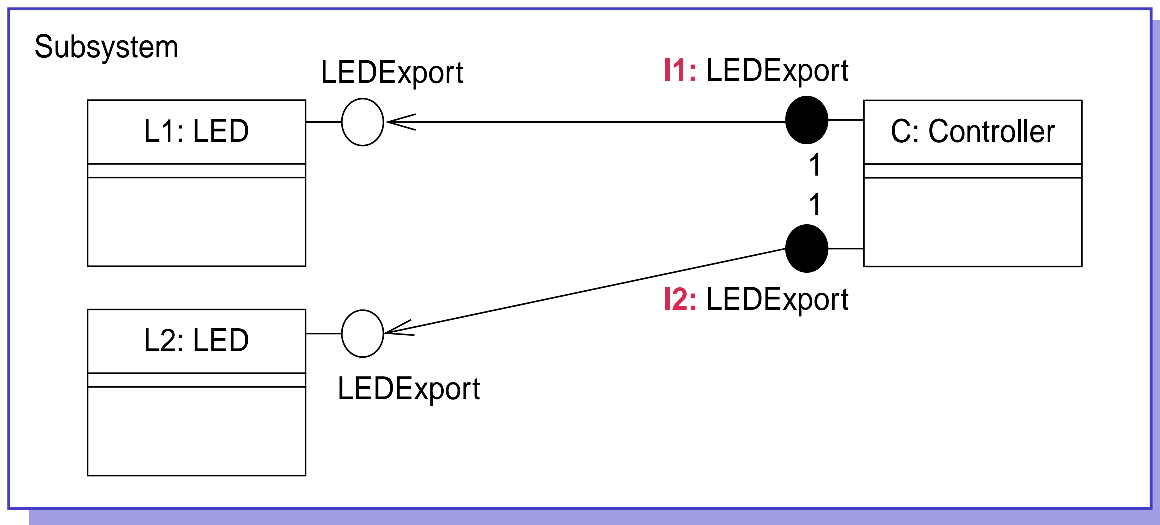


Abbildung 9: Benannte Import- und Exportschnittstellen

6. Erweiterung des UML-Metamodells

Will man die vorgeschlagenen Erweiterungen von Klassendiagrammen um benannte Import- und Exportschnittstellen in UML integrieren und im Quelltext verfügbare UML-Editoren wie ArgoUML [19] entsprechend umbauen, so muss zunächst folgende Frage geklärt werden:

> *Wie lassen sich die vorgeschlagenen Erweiterungen der UML-Lollipop-Notation in das Metamodell von UML integrieren?*

Die Beantwortung dieser Frage hat uns überraschend viel Zeit gekostet, da bereits die Lollipop-Notation für unbenannte Exportschnittstellen im UML-Metamodell (Version 1.x) nicht korrekt wiedergegeben wird. Sie ist laut UML-Standard [20] “nur” eine Notationsvariante der normalen Implementierungs- und Benutzungsbeziehungen zwischen Klassen und Schnittstellen und wird deshalb im UML-Metamodell nicht separat behandelt. Das ist aber falsch, da die Lollipop-Notation den Aufbau einer dreistelligen Beziehung (C, I, C’) zwischen einer Klasse C, die eine Schnittstelle I realisiert, und einer anderen Klasse C’, die die Schnittstelle I der Klasse C benutzt, erlaubt. Die normalen Realisierungs- und Benutzungsbeziehungen von UML erlauben hingegen nur den Aufbau binärer Beziehungen zwischen C und I sowie zwischen I und C’. Hier geht also die Information verloren, dass C’ nicht eine beliebige Implementierung der Schnittstelle I benutzt, sondern vielmehr genau deren Implementierung durch die Klasse C.

Interessanter Weise besitzt andererseits die UML-Metaklasse Association für die Deklaration beliebiger n-stelliger Beziehungen zwischen Klassen genau die fehlenden Mechanismen für den Aufbau solcher ternärer Beziehungen. Wie Abbildung 10 zeigt, besitzen die Assoziationsenden einer Assoziation zwei verschiedene Meta-Assoziationen zu der Meta-Klasse Classifier. Im Normalfall werden type-Links in der internen Darstellung von UML-Klassendiagrammen dafür eingesetzt, die an einer Assoziation beteiligten Klassen zu referenzieren. Die zusätzlich benutzbaren specificiation-Links werden im UML-Standard kaum erläutert und in allen uns bekannten Beispiele nicht verwendet.

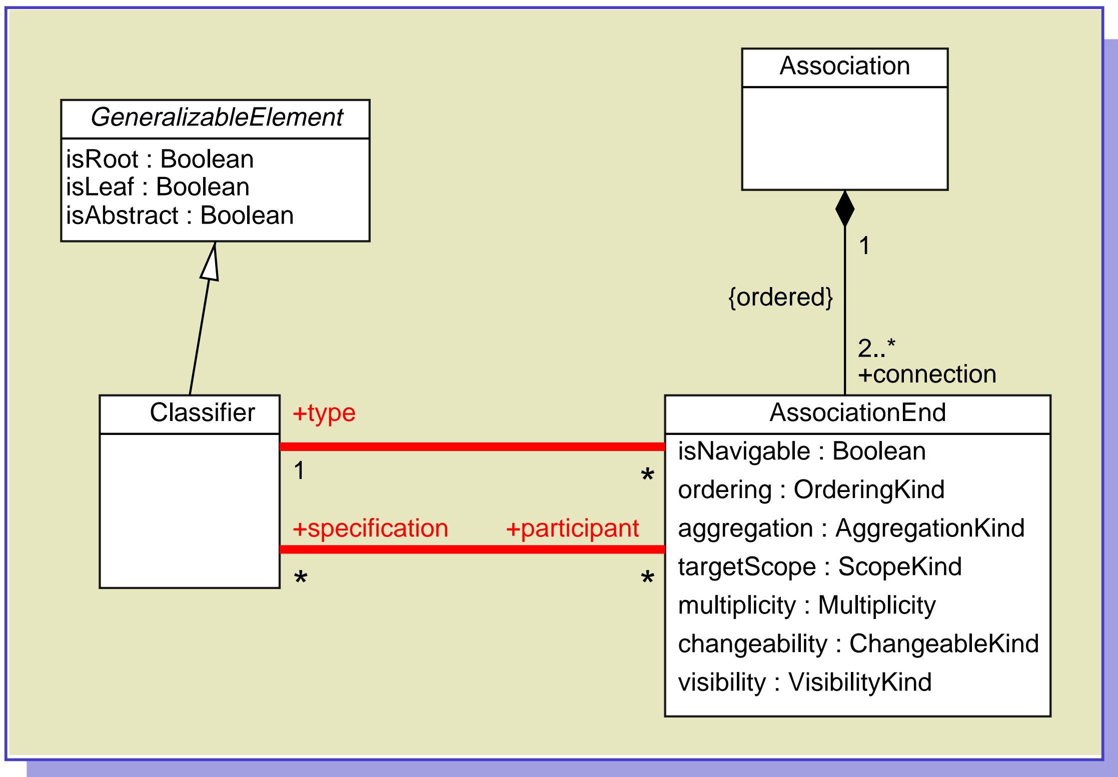


Abbildung 10: Ausschnitt des UML-Metamodells für Assoziationen

Somit könnte man eine der Verbindungen in Abbildung 9 zwischen einer Importschnittstelle der Klasse Controller und der Exportschnittstelle der Klasse LED als Instanz einer binären Assoziation (Association) mit zwei Enden (AssociationEnd) auffassen. Jedes dieser Enden wäre im einfachsten Fall über type mit der gerade betrachteten Klasse Controller oder LED verbunden und über specification mit der dazu gehörigen Schnittstellenbeschreibung (Interface-Klasse) LEDExport. Damit werden Verbindungen zwischen Export- und Import-Schnittstellen als vierstellige Relationen mit dem Aufbau

(Export-Klasse, Export-Interface, Import-Klasse, Import-Interface)

modelliert. Diese Art der Darstellung liefert genügend Informationen, wenn jede Klasse jedes definierte Interface nur einmal verwendet (implementiert), wie das in Standard-UML vorgesehen ist. Erlaubt man aber wie in in Abbildung 9, dass eine Klasse (Controller) mehrere unterschiedlich benannte Schnittstellen (I1 und I2) eines Schnittstellen-Typs (LEDExport) unterstützt, dann muss eine noch etwas aufwändigere interne Darstellung des betrachteten Diagramms eingesetzt werden.

Wie Abbildung 11 zeigt, kann man tatsächlich durch geschickten Einsatz von type- und specification-Links ohne große Erweiterungen des UML-Metamodells unsere erweiterten Klassendiagramme darstellen. Dafür darf man allerdings angewandte Auftreten einer Klasse in einer anderen Klassen nicht als einfache Kompositionsbeziehungen verstehen, sondern muss sie intern als Instanzen der Metaklasse AssociationClass darstellen. So wird das mit a bezeichnete angewandte Auftreten der Klasse A in der Deklaration der Klasse D in der internen Modelldarstellung auf eine Instanz der Meta-Klasse AssociationClass abgebildet, deren Quellassoziationsende src die Markierung für eine Kompositionsbeziehung enthält. Aus ähnlichen Gründen

müssen auch Import- und Exportschnittstellen angewandter Auftreten von Klassen als Instanzen von Meta-Assoziationsklassen aufgefasst werden. Hier wird allerdings eine zusätzliche von AssociationClass abgeleitete Metaklasse Bind benötigt, die ein boolsches Meta-Attribut für die Unterscheidung von Import- und Exportschnittstellen besitzt.

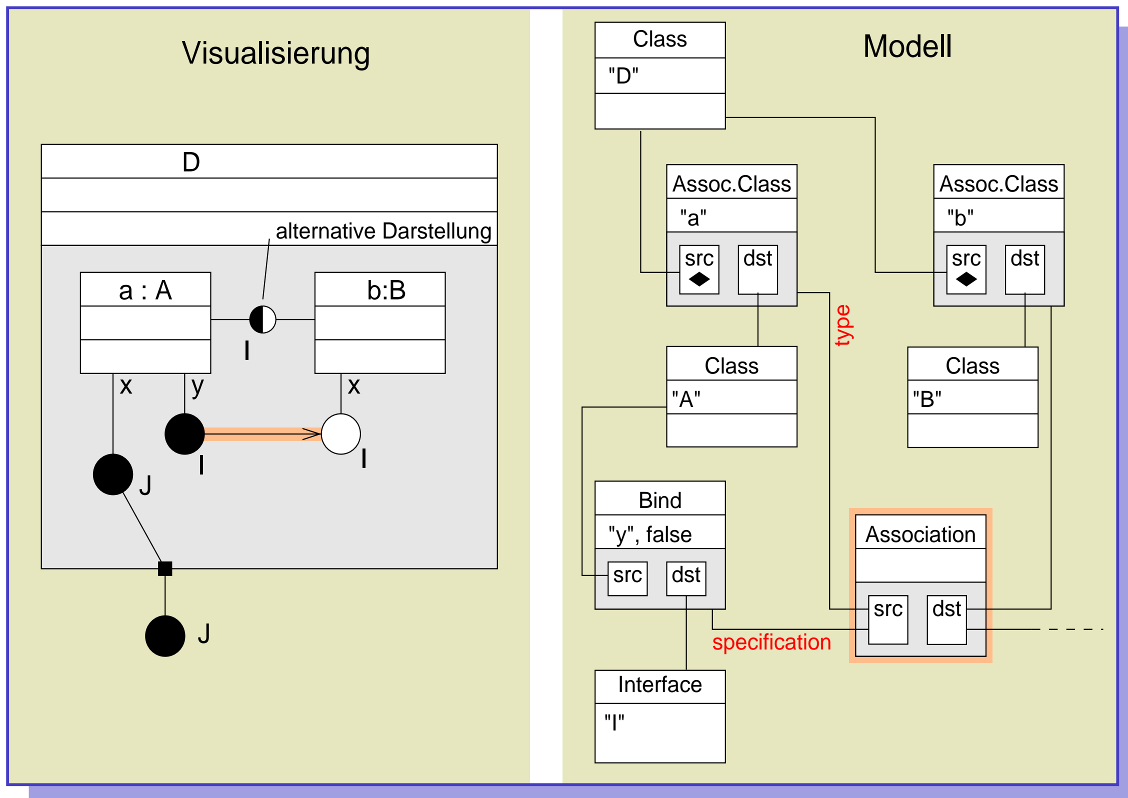


Abbildung 11: Interne Darstellung der erweiterten Klassendiagramme

Auf diese Weise lässt sich die in der linken Hälfte von Abbildung 11 hervorgehobene Bindung der mit y benannten Importschnittstelle von a : A an die mit x benannte Exportschnittstelle von b : B durch die orange markierte Assoziation darstellen. Sie ist über type-Kanten mit den Assoziationsklassen a und b verbunden sowie über specification-Kanten mit der Deklaration der Schnittstelle I. Letztere legt fest, welche Operationen über y von a : A importiert werden bzw. welche Operationen über x von b : B exportiert werden. Abbildung 11 zeigt im übrigen auch, wie die Schnittstellen der internen Komponenten einer Klasse mit Schnittstellen dieser Klasse selbst verbunden werden. Die interne Modellierung solcher Verbindungen der Schnittstellen einer Klasse mit den Schnittstellen ihrer Komponenten sieht genauso aus, wie die gerade diskutierte und in Abbildung 11 dargestellte Modellierung von Schnittstellenverbindungen.

Auf Basis der hier skizzierten Überlegungen zur Erweiterung des UML-Metamodells wurde von uns das *Open-Source Case-Tool ArgoUML* [19] erweitert. ArgoUML wurde ausgewählt, weil es samt Quellcode unter einer freien Lizenz erhältlich ist und zudem eine übersichtliche und vergleichsweise vollständige Implementierung des UML-Metamodells bietet. In ArgoUML werden die einzelnen Elemente des UML-Metamodels auf standardkonforme (aus dem Metamodell generierte) Java-Klassen abgebildet. Eine Erweiterung des Metamodells bedeutet daher im wesentlichen nur das Hinzufügen der neuen Elemente zum UML-Metamodell und ihrer Repräsentationsformen in Gestalt weiterer Java-Klassen zum ArgoUML-Quellcode..

7. Zusammenfassung

Die komponentenorientierte Modellierung spielt bei der Entwicklung eingebetteter Systeme oft eine herausragende Rolle. Wie in diesem Papier begründet, wird sie jedoch von Standard-UML bislang nur sehr unzureichend unterstützt. Die deshalb hier vorgeschlagene Erweiterung der UML-Lollipop-Notation für Klassenschnittstellen hat im Vergleich etwa zu dem notationell zunächst sehr ähnlichem Portkonzept in UML/Realtime [14] folgende Vorteile:

- Anstelle der Einführung einer neuen Diagrammart oder der Uminterpretation der UML-Kollaborationsdiagramme muss die Lollipop-Notation in UML-Klassendiagrammen für Schnittstellen nur geringfügig erweitert werden.
- Anstelle einer strikten Trennung zwischen aktiven Kapselklassen und passiven normalen Klassen erlaubt die hier vorgeschlagene Notation einen fließenden Übergang vom Einsatz normaler Klassen zu echten Komponentenklassen.
- Die notwendigen Erweiterungen des UML-Metamodells sind zudem sehr gering und beseitigen ein bislang ungelöstes Problem bei der internen Darstellung von „Lollipops“.

Laufende Arbeiten befassen sich u.a. mit einem genaueren Studium der Wechselwirkung der neu eingeführten Konzepte mit den UML-Konzepten zur Parametrisierung und Generalisierung von Klassen sowie mit der Vervollständigung ihrer Implementierung auf Basis von ArgoUML.

References

[1] Artisan Software: Real-time Modeler/Studio, http://www.artisansw.com/products/products.asp (zuletzt besucht: 12/2001)

[2] L. Bichler, A. Radermacher, A. Schürr: *Evaluating UML Extensions for Modeling Real-time Systems*, appears in: Proc. WORDS of the 2002 IEEE Workshop on Object-oriented Realtime-dependable Systems (2002)

[3] L. Bichler, A. Radermacher, A. Schürr: *Integrating data flow equations with UML/Real-time*, appears in: Real-Time Systems - The International Journal of Time-Critical Computing Systems , Kluwer Academic Publishers (2002)

[4] J. Ellsberger, D. Hogrefe, A. Sarma: *SDL: Formal Object-oriented Language for Communicating Systems*, Prentice Hall (1997)

[5] U. Freund, A. Burst: *TITUS - A Graphical Design Methodology for Embedded Automotive Software*, in: [9], 67-69 (2002)

[6] P. Geretschläger, P. Hofmann: *Objektorientierte Entwicklung eingebetteter Echtzeitsysteme im Automobil*, in: [8], 61-66 (1999)

[7] D. Harel, M. Politi: *Modeling Reactive Systems with Statecharts*, McGraw-Hill (1998)

[8] P. Hofmann, A. Schürr (Eds.): *Object-Oriented Modeling of Embedded Realtime Systems*, Proc. Int. OMER-Workshop, TR 99-01 Fakultät für Informatik, UniBw München, http://IST.UniBw-Muenchen.DE/GROOM/OMER/ (1999)

[9] P. Hofmann, A. Schürr (Eds.): *Object-Oriented Modeling of Embedded Realtime Systems*, Proc. 2nd Int. OMER-Workshop, TR 2001-03 Fakultät für Informatik, UniBw München, http://ist.unibw-muenchen.de/GROOM/OMER-2/ (2001)

[10] I-Logix: *Rhapsody*, http://www.ilogix.com/products/rhapsody/ (zuletzt besucht: 12/2001)

[11] I-Logix: *Statemate Magnum*, http://www.ilogix.com/products/magnum/ (zuletzt besucht: 12/2001)

[12] P. Kruchten, P.: *Modeling Component Systems with the Unified Modeling Language*; Int. Workshop on Component-Based Software Engineering, http://www.sei.cmu.edu/cbs/icse98/papers/p1.html (1998)

[13] Mathworks: *MATLAB - Simulink/Stateflow*, http://www.mathworks.com/products/ (zuletzt besucht: 12/2001)

[14] Rational Software Corporation: *Rational Rose/Realtime*, http://www.rational.com/products/rosert/index.jsp (zuletzt besucht: 12/2001)

[15] B. Rumpe, M. Schoenmakers, A. Radermacher, A. Schürr: *UML + ROOM as a Standard ADL*, in: Proc. ICECCS'99 5th Int. IEEE Conf. on Engineering Complex Computer Systems, IEEE Computer Society Press, 43-53 (1999)

[16] A. Schürr , A.J. Winter: *UML, the Future Standard Software Architecture Description Language?*, in: H. Kilov, B. Rumpe, and I. Simmonds (eds.): *B*ehavioral Specifications for Businesses and Systems, Kluwer Academic Publishers, 193-206 (1999)

[17] B. Selic, G. Gullekson, P. Ward: *Real-Time Object-Oriented Modeling*, John Wiley (1994)

[18] Th. Tempelmeier: *UML is great for Embedded Systems - Isn't It?*, in: [9], 67-72 (1999)

[19] Tigris: *ArgoUML*, http://argouml.tigris.org/index.html (zuletzt besucht: 12/2001)

[20] UML Revision Task Force: *OMG Unified Modeling Language Specification v 1.4*, http://cgi.omg.org/cgi-bin/doc?formal/01-09-67 (zuletzt besucht: Dec. 2001)