www.photocase.com

Entwicklung aus dem Baukasten

Modellierung und Verifikation technischer Systeme

Technische Systeme sind heute allgegenwärtig: Ob im Haushalt, im Auto oder im Flugzeug. Systemtechnik dominiert auch Heizungs- und Solartechnikanlagen, ganze Kraftwerke, Raffinerien oder Stahlwerke. Vernetzte und verteilte Systeme wie Mobilkommunikationssysteme, das Internet oder satellitengestützte Positions- und Mautsysteme umspannen die ganze Welt. Informatische Techniken sind dabei oft nicht nur Bestandteil dieser Systeme, sondern werden auch bei deren Konzeption, Produktion und Qualitätsüberwachung eingesetzt. Das Informatikjahr soll das Bewusstsein schärfen, dass unsere Gesellschaft ohne Technik und diese wiederum ohne Informatik überhaupt nicht mehr funktionieren würden.

Wer hat sich nicht schon mit diversen „Bugs“ in der PC-Software herumgeschlagen? Was am häuslichen Computer nur ärgerlich ist, kann bei *eingebetteter Software* in technischen Systemen große Schäden verursachen und im Extremfall sogar Menschenleben kosten. Um Fehler von Anfang an zu vermeiden, wird heutzutage zunächst ein Modell des technischen Systems und der dazugehörigen Software erstellt. Es dient zur Klärung der Anforderungen, als Referenz für die Entwickler und zur Festlegung, was „korrektes“ Funktionieren eigentlich bedeutet. Nach der Implementierung wird dann mit Hilfe verschiedener Verfahren überprüft, ob der erarbeitete Code mit diesem Modell konsistent ist.

Am Beispiel von Kraftfahrzeugen soll im Folgenden ein kleiner Eindruck von informatischen Techniken zur Modellierung und Verifikation vermittelt und die vielfältigen Schnittstellen und Vernetzungen zu anderen Gebieten aufgezeigt werden.

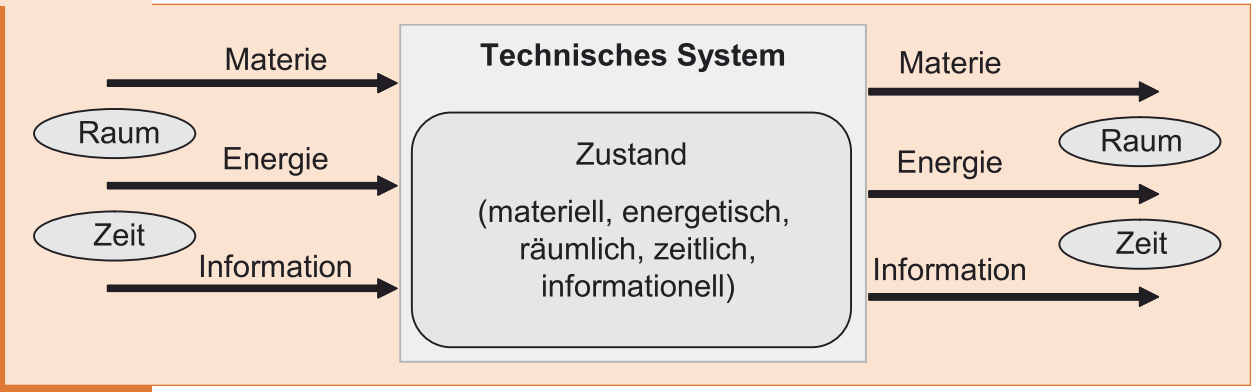

Abbildung 1:
Technisches System

Technische Systeme

Menschen können eigentlich alles besser als der Computer – nur bei einfachen Rechen- und Speicheroperationen ist die Maschine schneller. Informatische Techniken zielen deshalb darauf ab, komplexe Aufgaben- und Problemlösungen in einfache, computertaugliche Schritte zu überführen. Die gewünschten Anforderungen und Funktionalitäten müssen möglichst genau, eindeutig und detailliert beschrieben sowie gegeneinander abgegrenzt werden.

Ein technisches System verfolgt immer einen Zweck. Es besteht aus Elementen, die Eigenschaften besitzen und zur Verfolgung gesetzter Ziele durch geeignete Relationen miteinander verknüpft sind (Abb. 1). Ein Kraftfahrzeug wird zum Beispiel für den umfassenden Zweck gebaut, Transporte effizient und sicher von Punkt A zu Punkt B zu ermöglichen. Aus diesem umfassenden Zweck lassen sich viele (Teil-) Ziele und Aufgaben ableiten, die das technische System „Kraftfahrzeug“ erfüllen muss.

In fast allen technischen Systemen befinden sich heutzutage spezielle Rechnersysteme, die nach außen weitgehend unsichtbar sind und deshalb als *eingebettet* bezeichnet werden. Im Unterschied zu PCs interagieren eingebettete Systeme über Sensoren und Aktoren direkt mit der realen Welt, beeinflussen also auch direkt Energie und Materie in Raum und Zeit.

Bei jeder Systembetrachtung ist deshalb immer auch die Umgebung mitzubetrachten, soweit sie mit dem System in Beziehung steht.

Für die Modellierung der Kraftfahrzeugbewegung hat sich zum Beispiel in der wissenschaftlichen Literatur ein Drei-Ebenen-Modell etabliert (Abb. 2). Darin zählen Straßennetz, Straßenverlauf, Verkehrsregeln, Straßenoberfläche, Fahrzeugzustand sowie Sicht und Witterung zur Umwelt. Fahrer und Assistenzsysteme werden in eine Stabilisierungs-, eine Führungs- und eine Navigationsebene eingeordnet. Ein elektronischer Bremsassistent (ABS) oder eine Gierstabilisierung (ESP) gehören zum Beispiel in die Stabilisierungs-, ein abstandsgeregelter Tempomat (ACC) in die Führungsebene.

Arbeitsteilung im Auto

Das System Kraftfahrzeug wird mit zahlreichen unterschiedlichen Modellen in Subsysteme zerlegt und beschrieben. Jedes dieser Subsysteme umfasst einen bestimmten, klar definierten Aufgabenbereich. Beispiele für mechatronische Subsysteme sind neben den schon erwähnten Fahrer- und Assistenzsystemen beispielsweise Motor-, Getriebe- oder Antriebsstrangregelungen sowie Steuerungen für Scheibenwischer, Licht, Fensterheber, Türöffnungssysteme, Sitze, Spiegel oder Klimaanlagen. Diese Subsysteme bestehen aus kombinierten Elementen wie Software (SW), Hardware (HW) und Bussystemen (CarWideWeb) sowie Elektronik, Elektrik, Mechanik und Hydraulik (Abb. 3).

Die Funktion eines technischen Systems ist das Vermögen, bestimmte Eingangsgrößen in Ausgangsgrößen zu überführen. Für das Kraftfahrzeug drückt sich die Funktion der effizienten und sicheren Fortbewegung in einem breiten Spektrum mit vielen unterschiedlichen Funktionalitäten aus, die wiederum in verschiedensten Sub-Systemen betrachtet werden können.

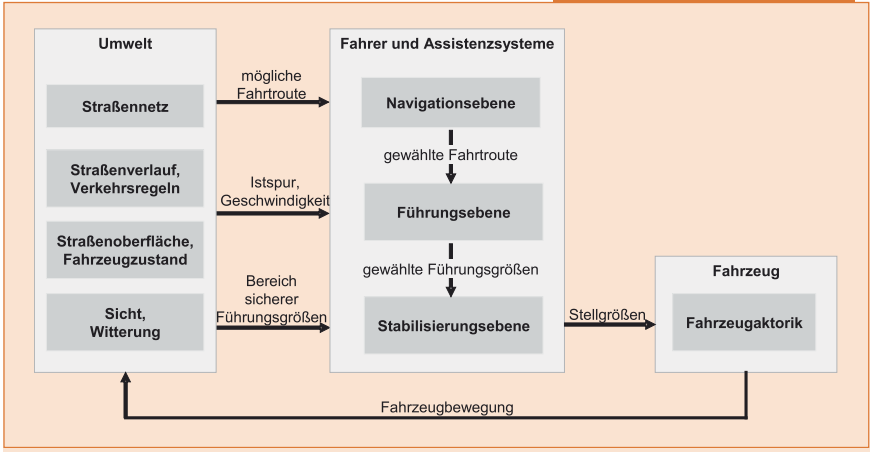

Abbildung 2: Drei-Ebenen-Modell der Fahrzeugbewegung (nach D. Ehmanns et al: Zukünftige Entwicklungen von Fahrerassistenzsystemen und Methoden zu deren Bewertung. 9. Aachener Kolloquium Fahrzeug- und Motorentechnik. 2000.).

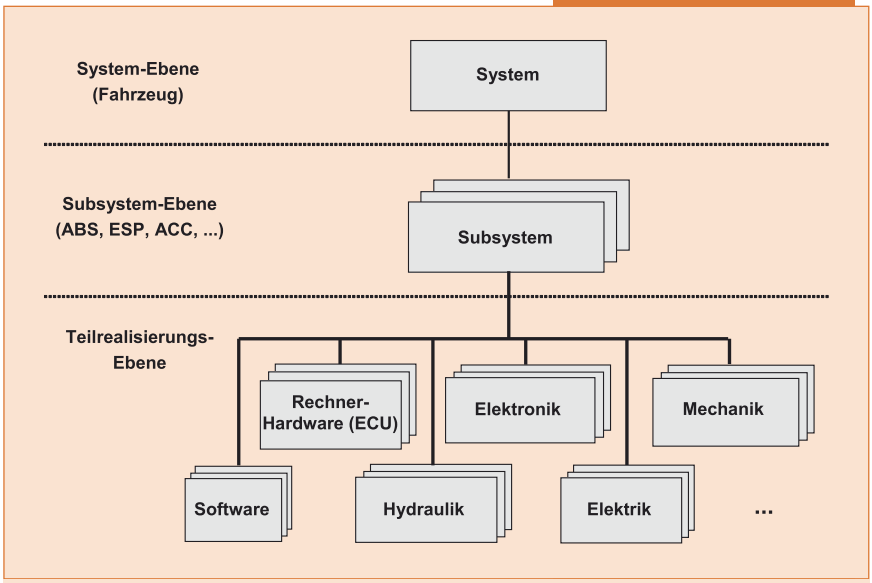

Abbildung 3: Fahrzeug mit Beispiel-Subsystemen der Funktionen Bremsassistent (ABS), Gierstabilisierung (ESP) und abstandsgeregelter Tempomat (ACC).

Systemverhalten

Bei *ereignisgetriebenen* Systemen wird das Systemverhalten durch externe Aktionen infolge (zufällig) eintretender Ereignisse – hierzu zählt aus Systemsicht in der Regel auch jede Benutzereinwirkung – oder interne Aktionen (als Funktion der Momentanwerte der Zustandsgrößen) aktiviert. Ereignisse treten, abhängig von den externen Prozessen, mit vor der Laufzeit nicht vorhersehbarer unterschiedlicher Häufung auf. Die Reihenfolge ihres Auftretens mag (jedenfalls im Normalbetrieb) für einzelne Systemteile definiert sein, im ganzen System ergibt sich aber eine Unmenge verschiedener möglicher Sequenzen. Diese Sequenzen sind nicht vorhersehbar und lassen sich deshalb auch in der Regel nicht vollständig statisch analysieren.
Bei *zeitgetriebenen* Systemen ist das Systemverhalten allein durch seine Abhängigkeit von der Zeit geprägt. Das System wird ständig oder periodisch an bestimmten Zeitpunkten aktiviert, wobei es selbst (in einer gewissen Weise entkoppelt von Ereignissen im Umfeld) bestimmt, welche Eingangsgrößen auf Änderungen untersucht und verarbeitet werden sollen. Weil die zeitliche Abfolge der einzelnen Systemfunktionen nicht direkt von den tatsächlichen Ereignissen im Umfeld abhängt, kann die gesamte Systemfunktion statisch analysiert werden. Damit ist es auch möglich, vor der Laufzeit eine fixe Sequenz von Aufgaben festzulegen.

Die Elemente eines solchen Systems stehen untereinander in Beziehungen, woraus sich die Struktur des Systems beschreiben lässt. Die Systemstruktur und Systemfunktion stehen zueinander in Beziehung. Sind die Funktion der Elemente eines Systems und seine Struktur bekannt, so lässt sich daraus die Funktion des Systems ableiten.

Das Verhalten eines Systems umfasst alle Veränderungen seiner Eigenschaften, also der Eingangs-, Ausgangs- und Zustandsgrößen. Die Eingangsgrößen werden durch die Systemfunktion in Ausgangsgrößen überführt, die Zustandsgrößen beschreiben den Systemzustand. Dabei wird zwischen *ereignisgetriebenen* und *zeitgetriebenen Systemen* unterschieden (siehe Kasten „Systemverhalten“).

Die meisten technischen Systeme enthalten sowohl zeitgetriebene als auch ereignisgetriebene Anteile. Typische Ereignisse in einem fahrenden Kraftfahrzeug sind Ein- und Ausschaltvorgänge für das Fahrlicht, das Betätigen von Blinker und Scheibenwischer oder die Vorgabe einer Geschwindigkeit für den Tempomaten. Typisch zeitgetriebenes Verhalten stellen komplexe Regelvorgänge dar wie etwa die Drehzahlregelung des Motors oder die katalytische Abgasreinigung; informatische Techniken sind hierbei ein wichtiger Teil im mechatronischen Gesamtsystem Kraftfahrzeug.

Simulation, Verifikation, Validation

Das technische System Kraftfahrzeug in Struktur und Verhalten in seiner Gesamtheit zu beschreiben, ist aufgrund seiner Komplexität bis heute nicht vollständig gelungen. Deshalb werden (informatische) Modelle erstellt, die jeweils zur Lösung von Teil-Aufgaben benutzt werden.

Ein Modell ist ein zum Teil abstrahiertes und vereinfachtes Abbild eines realen Systems. Unterschiedliche Sichtweisen und Prioritäten können zu unterschiedlichen Modellen desselben Systems führen. Ein Modell wird in einer formalen Beschreibungssprache exakt dargestellt und ist so zur rechnergestützten Verarbeitung geeignet. Da ein Modell die Realität nie vollständig abbilden kann, ist immer mit einer Abweichung zwischen dem Verhalten von realem System und Modell zu rechnen. Informatisch-technische Modelle können dem Erkenntnisgewinn, dem Verständnis von Zusammenhängen oder der Simulation der Funktion eines Systems dienen. Es gibt ebenso Modelle, die den Menschen bei einer bestimmten Aufgabe modellieren – zum Beispiel Fahrermodelle in Modellen der Fahrzeugbewegung.

Prinzipiell können Experimente direkt mit dem realen System durchgeführt werden. In der Realität ist dies allerdings oft nicht oder nur schwer möglich, weil

- das reale System noch zu entwickeln ist und für Experimente gar nicht zur Verfügung steht;
- Experimente mit dem realen System gefährlich sein können, etwa bei fahrdynamischen Versuchen in Grenzbereichen;
- die Messung der Systemausgangsgrößen manchmal unvermeidbare Rückwirkungen auf das System selbst hat, was zur Verfälschung des Experimentes führt;
- manche Experimente wie zum Beispiel Crashtests zu teuer sind;
- einige Experimente zu lange dauern – etwa Langzeitversuche zur Lebensdauer von Bauteilen.

Die Simulation ist jederzeit wiederholbar – im Gegensatz zu einigen realen Prozessen, die nur einmal laufen können. Sie ist eine mögliche Tech-

nik zur Verifikation eines Modells. Allerdings kann man sich nie ganz sicher sein, ob man wirklich alle möglichen Fälle simuliert oder durchgetestet hat, da es unendlich viele Verwendungsszenarien gibt.

Daher sind Verifikationstechniken wünschenswert, die in gewissem Rahmen zu mathematischen Korrektheitsbeweisen führen. Unter Verifikation versteht man die Überprüfung der Übereinstimmung zwischen einem Produkt und seiner *Spezifikation*. Im Unterschied dazu ist die Validation die Überprüfung der Eignung beziehungsweise des Wertes eines Produktes in Bezug auf seinen *Einsatzzweck*.

Software Engineering

Früher wurden mit Software klassische Datenverarbeitungsprobleme wie zum Beispiel Lohn- und Gehaltsabrechnungen gelöst. Heute wird Software zunehmend in technischen Systemen eingesetzt – oft in sicherheitskritischen Bereichen, beispielsweise bei Flight Management-Systemen, den so genannten Autopiloten moderner Flugzeuge.

Diese Situation stellt eine große Herausforderung für die Softwareentwicklung dar, da nun das Versagen der Software nicht mehr nur unangenehm, sondern potenziell lebensbedrohend sein kann.

Aus diesem Grund muss Software für technische Systeme mit besonderer Sorgfalt entwickelt werden. In diesem Abschnitt wird eine Entwicklungsmethodik für eingebettete Software beschrieben, die in der Arbeitsgruppe „Software Engineering" der Fakultät für Ingenieurwissenschaften der Universität Duisburg-Essen unter Leitung von Maritta Heisel entwickelt wurde.

Kernstück der Methodik ist eine sorgfältige Problemanalyse. Nur ein Problem, das bis ins Detail verstanden ist, kann adäquat gelöst werden. Zur Problemanalyse gehören insbesondere

- die Beschreibung der *Umgebung*, in der die Software eingesetzt werden soll, wobei man die Eigenschaften der Umgebung, die durch die Software nicht beeinflusst werden können, *Domänenwissen* nennt;
- die Beschreibung der *Anforderungen*, die die Software erfüllen soll. Hierbei ist wichtig, dass die Anforderungen nicht die *Software* beschreiben, sondern das Verhalten der Umgebung, nachdem die Software in Betrieb ist. Außerdem wird bei der Problemanalyse nicht beschrieben, wie die Software *arbeiten*, sondern was sie *bewirken* soll.

Um ein Problem zu lösen, muss aus den Anforderungen eine Beschreibung der zu entwickelnden Software abgeleitet werden. Diese *Spezifikation* bildet die Grundlage des Softwareentwicklungsprozesses und wird mit Hilfe des Domänenwissens aus den Anforderungen gewonnen.

Am Beispiel eines intelligenten, abstandsgeregelten Tempomaten soll verdeutlicht werden, wie Software für technische Systeme entwickelt werden kann. Die *Systemmission*, der Zweck des Tempomaten, besteht darin, den Fahrer eines Fahrzeugs vom Konstanthalten der Geschwindigkeit zu entlasten. Moderne Tempomaten messen dabei auch den Abstand zum vorausfahrenden Fahrzeug und unterschreiten bei Bedarf die voreingestellte Geschwindigkeit. Die folgenden Anforderungen (R wie *requirements*) präzisieren die Systemmission:

- **R1** Die Geschwindigkeit des Fahrzeugs darf nur geregelt werden, wenn der Tempomat aktiviert ist.
- **R2** Die Aktivierung des Tempomaten erfolgt über die Bedienhebel „schneller" oder „langsamer". Die eingestellte Geschwindigkeit ist die gegenwärtige Geschwindigkeit des Fahrzeugs.
- **R3** Die Geschwindigkeit des Fahrzeugs wird aufgrund der gegenwärtigen Geschwindigkeit und der Sollgeschwindigkeit geregelt. Die Sollgeschwindigkeit wird aus der Geschwindigkeit des vorausfahrenden Fahrzeugs und der eingestellten Geschwindigkeit ermittelt.
- **R4** Die Sollgeschwindigkeit ist entweder die eingestellte Geschwindigkeit oder eine geringere Geschwindigkeit, die ausreicht, um einen Sicherheitsabstand zu dem vorausfahrenden Fahrzeug zu halten.
- **R5** Wenn der Tempomat aktiviert ist, kann die eingestellte Geschwindigkeit mit den Bedienhebeln „schneller" und „langsamer" herauf- und herabgesetzt werden.
- **R6** Die Deaktivierung des Tempomaten erfolgt durch das Betätigen des Bedienhebels „aus" oder das Betätigen des Gas- oder Bremspedals.

Das Domänenwissen, das nötig ist, um diese Anforderungen mittels Software erfüllen zu können, besteht beispielsweise aus den Regeln, wie die Sollgeschwindigkeit aus gegenwärtiger Geschwindigkeit und Geschwindigkeit des vorausfahrenden Fahrzeugs ermittelt wird.

Die Struktur der Umgebung wird in einem *Kontextdiagramm* dargestellt (Abb. 4). Einfache Kästchen stehen für *gegebene* Domänen. Diese sind bereits vorhanden, und ihre Eigenschaften wer-

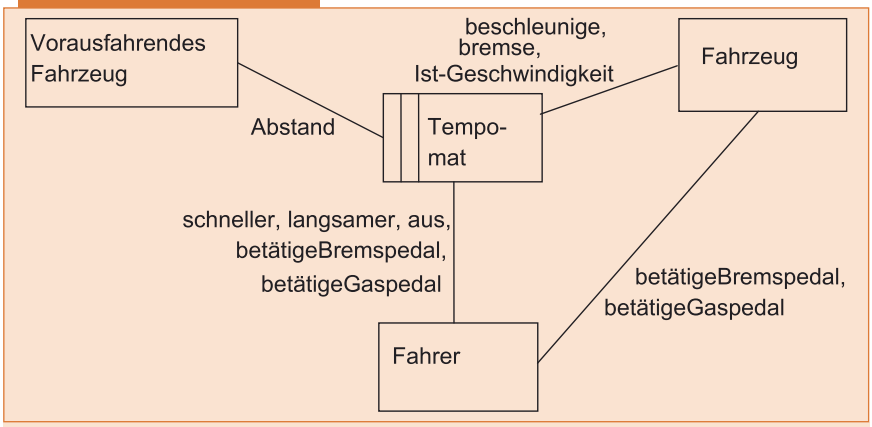

Abbildung 4: Kontextdiagramm für einen abstandsgeregelten Tempomaten.

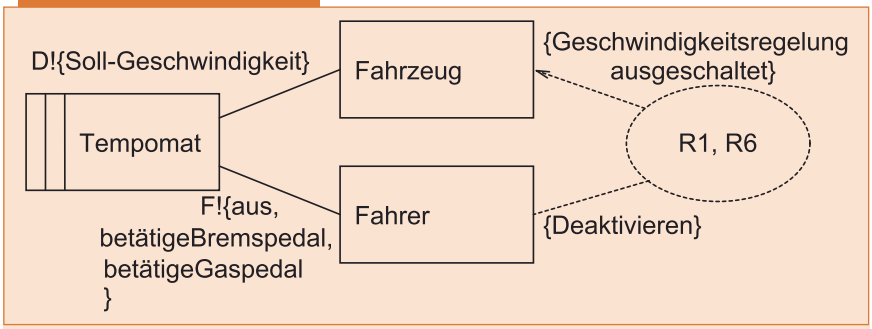

Abbildung 5: Problemdiagramm für einen abstandsgeregelten Tempomaten.

den durch das Domänenwissen beschrieben. Das Kästchen mit den zwei senkrechten Strichen hingegen stellt die zu entwickelnde Software dar. Verbindungslinien zwischen Domänen repräsentieren *Schnittstellen*. Schnittstellen bestehen aus so genannten *gemeinsamen Phänomenen*. Im Beispieldiagramm kann der Fahrer zum Beispiel das Bremspedal betätigen. Dies ist eine Aktion, die sowohl vom Tempomat als auch vom Fahrzeug wahrgenommen wird. Deswegen ist das Phänomen „betätige Bremspedal" in beiden vom Fahrer ausgehenden Schnittstellen enthalten.

Die Umgebung, in der die Software arbeiten soll, ist nun strukturiert, und es ist festgehalten, wie sich ein Fahrzeug mit eingebautem Tempomat verhalten soll. Damit ist die Problemanalyse aber noch nicht beendet, denn das zu lösende Problem ist so komplex, dass es in kleinere Teilprobleme zerlegt werden sollte. Diese Teilprobleme sollten möglichst Problemklassen angehören, für die bereits Lösungsansätze bekannt sind. Beispiele solcher Problemklassen sind Steuerungs- oder Transformationsprobleme.

Jedes Teilproblem wird mittels eines *Problemdiagramms* repräsentiert (Abb. 5). Es entsteht aus dem Kontextdiagramm durch Hinzufügen weiterer Informationen: Ein gestricheltes Oval stellt die Anforderungen dar, die durch das betreffende Unterproblem gelöst werden. Dies sind jeweils inhaltlich zusammenhängende Anforderungen. Gestrichelte Linien zeigen an, auf welche Domänen sich die Anforderungen beziehen. Wenn eine gestrichelte Linie mit einem Pfeil versehen ist, bedeutet dies, dass die betreffende Domäne durch die Anforderung eingeschränkt wird (Domäne Fahrzeug in Abb. 5). Außerdem wird bei den Schnittstellen die Information hinzugefügt, welche Domäne die Kontrolle über ein gemeinsames Phänomen hat. Beispielsweise bedeutet die Schreibweise F!{aus, betätige Bremspedal, betätige Gaspedal}, dass der Fahrer die Kontrolle über die Phänomene „aus", „betätige Bremspedal" und „betätige Gaspedal" ausübt.

Das in Abbildung 5 gezeigte Problemdiagramm beschreibt das Problem des Deaktivierens des Tempomaten. Weitere Teilprobleme sind das Aktivieren des Tempomaten, das Berechnen der Sollgeschwindigkeit sowie die Steuerung der Geschwindigkeit.

Von Anforderungen zur Spezifikation

Bei der Problemzerlegung sollen alle Anforderungen von genau einem Teilproblem abgedeckt werden. Daraus ergibt sich für jedes Teilproblem genau eine Softwarekomponente. Bevor jedoch diese Komponenten genauer betrachtet werden, sind die zu jedem Teilproblem gehörenden Anforderungen in Spezifikationen umzuwandeln. Während Anforderungen die Umgebung beschreiben, beschreiben Spezifikationen die zu konstruierende Softwarekomponente. Allerdings wird diese als „black box" betrachtet, ihre Struktur wird noch nicht festgelegt. Es wird nur beschrieben, wie sie sich an ihren Schnittstellen verhält, welche Signale sie in welcher Reihenfolge aus ihrer Umgebung empfangen kann und welche sie ihrerseits sendet.

Zur Darstellung von Spezifikationen eignen sich *Sequenzdiagramme* gut (Abb. 6). Sie beschreiben den Nachrichtenfluss zwischen verschiedenen Entitäten, das heißt zwischen verschiedenen Objekten. Aufgrund des Problemdiagramms ist bekannt, mit welchen anderen Domänen die Software welche Nachrichten austauschen kann. Gemeinsame Phänomene, die die Softwarekomponente kontrolliert, entsprechen Nachrichten, die die Software *aussenden* kann. Gemeinsame Phänomene, die von einer mit der Software verbundenen Domäne kontrolliert werden, entsprechen Nachrichten, die die Software von der betreffenden Domäne *empfangen* kann. Das Problemdiagramm enthält also schon wichtige Informationen, die bei der Konstruktion der Spezifikation von Nutzen sind und auch Konsistenzprüfungen erlauben.

Das Sequenzdiagramm beschreibt das Deaktivieren des Tempomaten. Die gegebene Spezifikation ist nur gültig, wenn der Tempomat aktiviert ist. Dies wird durch die *Zustandsinvariante* „{Geschwindigkeitsregelung ein}“ ausgedrückt, die sich auf das Fahrzeug bezieht. Wie in Anforderung R6 angegeben, gibt es drei Möglichkeiten, den Tempomaten zu deaktivieren. Diese sind mit „ALT“ (für Alternative) zusammengefasst. Nachdem der Tempomat eine dieser Nachrichten empfangen hat, ist die Geschwindigkeitsregelung deaktiviert. Der mit „NOT“ gekennzeichnete Block gibt an, dass nunmehr das Signal Soll-Geschwindigkeit nicht mehr gesendet werden darf, wie in Anforderung R1 gefordert.

In ähnlicher Weise geht man auch für die anderen Teilprobleme vor und erhält für jedes eine Verhaltensbeschreibung in Form von Sequenzdiagrammen.

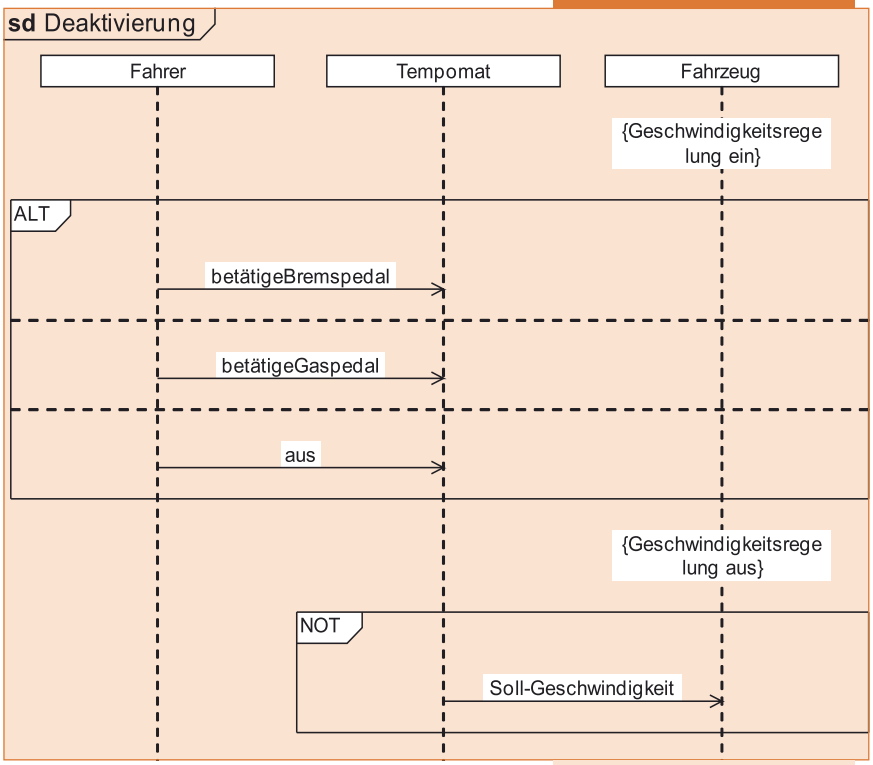

Abbildung 6: Sequenzdiagramm zur Beschreibung der Deaktivierung des Tempomaten.

Softwarearchitektur

Im nächsten Schritt geht es darum, eine Architektur der Software zu entwerfen. Eine *Softwarearchitektur* besteht aus Komponenten, die als Programme in einer Programmiersprache realisiert werden, sowie Konnektoren, die diese Komponenten verbinden und so deren Zusammenarbeit ermöglichen. Da ein kombiniertes Softwaresystem entwickelt werden soll, das alle Teilprobleme löst, muss diese Architektur auch alle Teilprobleme abdecken. Zur Entwicklung der Architektur gibt es komplexe heuristische Methoden, deren Beschreibung allerdings den Rahmen dieses Artikels sprengen würde.

Abbildung 7 zeigt die Gesamtarchitektur des Tempomaten. *Komponenten* werden als Kästchen mit Namen dargestellt. *Konnektoren* entsprechen Linien, deren Endpunkte mit kleinen weißen Quadraten, den *Ports*, versehen sind. Es handelt sich um eine *Schichtenarchitektur*. Die oberste Schicht führt die Berechnungen durch, die für die Funktion des Tempomaten nötig sind. Die unterste Schicht stellt die Verbindung zu der verwendeten Hardware her. Die darüber liegende Schicht beinhaltet Komponenten der so genannten AUTOSAR-Softwarearchitektur. Dies ist eine speziell auf den Automobilbereich abgestimmte Architektur, die von mehreren Automobilherstellern und Zulieferern gemeinsam entwickelt wurde. Sie stellt zum Beispiel System- und Kommunikationsdienste bereit. Der eigentliche Tempomat nimmt die oberen beiden Schichten ein. Die mit SAE (Schnittstellen-Abstraktions-Ebene) gekennzeichneten Softwarekomponenten übersetzen die Signale, die von der darunter liegenden Schicht geliefert werden, in eine höhere Abstraktionsebene, wie sie etwa für die Berechnung der Sollgeschwindigkeit benötigt wird.

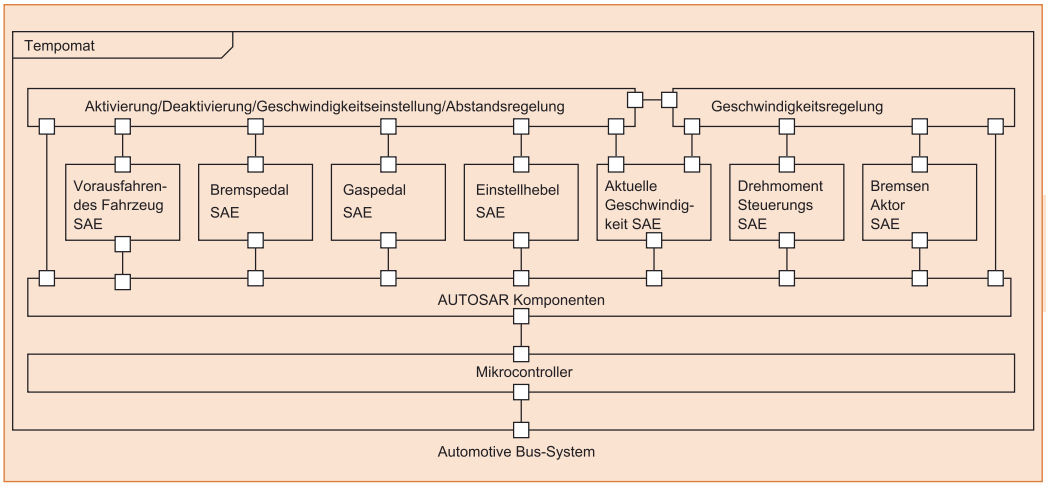

Abbildung 7: Entstehende mögliche Softwarekomponenten in der Fallstudie abstandsgeregelter Tempomat.

EXPERTENWISSEN

Model-Checking

Unter *Model-Checking* versteht man Techniken, die es erlauben, automatisch zu überprüfen, ob ein System seine Spezifikation erfüllt. Diese Techniken sind oft für Systeme, die durch hohen Parallelismus leicht unüberschaubar werden, sehr gut geeignet. Bisher wurde Model-Checking vor allem für Systeme mit endlichem Zustandsraum eingesetzt, seit einiger Zeit konzentriert sich die Forschung in diesem Bereich jedoch immer mehr auf unendliche Zustandsräume, wie sie bei Software vorkommen. Allein schon die Tatsache, dass ein Programm beliebige Zahlen als Eingabe verarbeiten kann, führt zu unendlich vielen Zuständen.

Alle in der Softwarearchitektur enthaltenen Komponenten müssen nun ihrerseits spezifiziert werden, bevor mit ihrer Implementierung begonnen werden kann. Für die Komponenten der obersten Schicht können die zuvor entwickelten Sequenzdiagramme weitgehend wieder verwendet werden.

Natürlich muss nach der Fertigstellung geprüft werden, ob die Software die aufgestellten Spezifikationen auch tatsächlich erfüllt. In der Praxis ist hierzu das Testen der am weitesten verbreitete Ansatz. Doch leider kann Testen nur die Anwesenheit, nicht aber die Abwesenheit von Softwarefehlern zeigen. Seit einiger Zeit gibt es jedoch auch Techniken, mit denen die Abwesenheit von Fehlern nachgewiesen werden kann.

Automatische Verifikation

So müsste es sein: Die erstellte Software wird zusammen mit dem zuvor erstellten Modell oder der Spezifikation in ein Verifikationsprogramm eingegeben, man drückt einen Knopf – und nach kurzer Zeit erhält man eine Antwort, ob die Software auch die Spezifikation erfüllt. Und falls nicht, wird genau angezeigt, an welcher Programmzeile von der Spezifikation abgewichen wird! So in etwa liest sich der Traum jedes Softwareentwicklers.

Und ein Traum wird es bleiben: Es gibt keine vollautomatischen Verfahren, um die Korrektheit eines Programms zu beweisen – nicht einmal für nicht-eingebettete Software. Und bei eingebetteter Software ist die Lage aufgrund ihrer Integration in die physikalische Wirklichkeit noch komplizierter.

Trotzdem ist dies kein Grund zur Verzweiflung: Im Hardware-Bereich werden schon seit längerem erfolgreich Model-Checking-Techniken zur Verifikation angewandt. Die Situation ist bei Hardware allerdings etwas einfacher, da im Gegensatz zu Software von einem endlichen Zustandsraum ausgegangen werden kann (siehe Kasten „Model-Checking“).

Bei herkömmlicher nicht-eingebetteter Software gibt es inzwischen bereits zahlreiche Ansätze (vor allem mit Hilfe von Approximationstechniken), automatische Verifikation zu erreichen. Dabei ist man mit so genannten sicheren Approximationen zufrieden: Wenn das Ergebnis der Verifikation „System ist korrekt“ lautet, dann kann man sich dieser Korrektheit auch sicher sein. Allerdings kann man auch Ergebnisse der Form „Ich weiß nicht“ erhalten.

Bei eingebetteter Software geht es zusätzlich darum, die Echtzeitfähigkeit des Systems zu überprüfen und die nicht-digitale Umgebung zu modellieren. Dabei können Aussagen über die Korrektheit immer nur so gut wie die – vermutlich nicht vollkommen exakte – Umgebungsmodellierung sein.

Trotz dieser Schwierigkeiten gibt es auch für eingebettete Software schon erstaunliche Anfangserfolge, beispielsweise das europäische Projekt DAEDALUS, bei dem die Grundlagen der automatischen Analyse für die sicherheitskritische Software von Airbus-Flugzeugen gelegt wurden.

Wie aber sieht es mit der Verifikation unseres Tempomaten aus? Man könnte alle Komponenten durch so genannte Zustandsübergangsdiagramme (siehe Kasten) modellieren. Die Komponente Aktivierung/Deaktivierung würde dann wie in Abbildung 8 dargestellt werden.

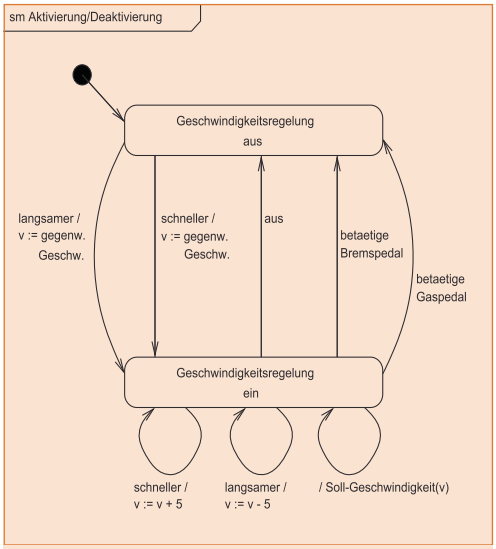

Abbildung 8:
Zustandsübergangsdiagramm für die Aktivierung und Deaktivierung des Tempomaten.

Alle Zustandsübergangsdiagramme der einzelnen Komponenten ergeben ein Diagramm für das Gesamtsystem. Für dieses Diagramm kann man dann automatisch überprüfen, ob bestimmte Eigenschaften erfüllt sind. Eine solche geforderte Eigenschaft ist zum Beispiel: „Der Tempomat regelt die Geschwindigkeit nur, wenn er eingeschaltet und nicht wieder ausgeschaltet wurde“

Car Platoons

In Abbildung 8 ist einfach zu erkennen, dass die Geschwindigkeit tatsächlich nur dann geregelt wird, wenn der Tempomat eingeschaltet ist. Allerdings könnten sich durch die Komposition mit anderen Komponenten Effekte ergeben, die den Tempomaten aktivieren, ohne dass er explizit eingeschaltet wird. Daher muss man entweder das Zustandsübergangsdiagramm des Gesamtsystems betrachten oder sicherstellen, dass die zu überprüfende Eigenschaft beim Zusammensetzen der Komponenten erhalten bleibt.

Es gibt auch Systeme, bei denen die Anzahl von Komponenten nicht von Anfang an festgelegt ist und die sehr stark dynamisches Verhalten zeigen. Ein Beispiel hierfür sind so genannte Car Platoons, bei denen sich Autos zu Verbänden zusammenfinden, die dann gemeinsam über die Autobahn fahren und zentral gesteuert werden, was viele Vorteile wie zum Beispiel kürzere Abstände und Entlastung der Fahrer bietet. Techniken für die Verifikation solcher Systeme stecken noch in den Kinderschuhen, aber es gibt schon viel versprechende Ansätze. Die Arbeitsgruppe „Theoretische Informatik“ der Fakultät für Ingenieurwissenschaften der Universität Duisburg-Essen unter Leitung von Barbara König beschäftigt sich mit Methoden zur Verifikation von Systemen mit stark dynamischem Verhalten, von denen Platoons ein sehr konkretes Beispiel sind.

Eine andere Möglichkeit, die Korrektheit von Software zu garantieren, ist, sie direkt aus dem Modell oder der Spezifikation zu generieren. Dazu werden beispielsweise graphische, schaltkreisähnliche Blockschaltbilder als (Simulink-)Modelle erstellt und dann automatisch in Code übersetzt. Fehler fallen auf dieser Modellebene viel besser und früher auf als im Code und sind dann auch leichter zu beheben. Ein großes Problem gibt es aber auch hier: Wer garantiert, dass die Codegenerierungs-Werkzeuge korrekt arbeiten? Das muss dann wiederum durch automatische Verifikation oder andere Validierungsverfahren sichergestellt werden.

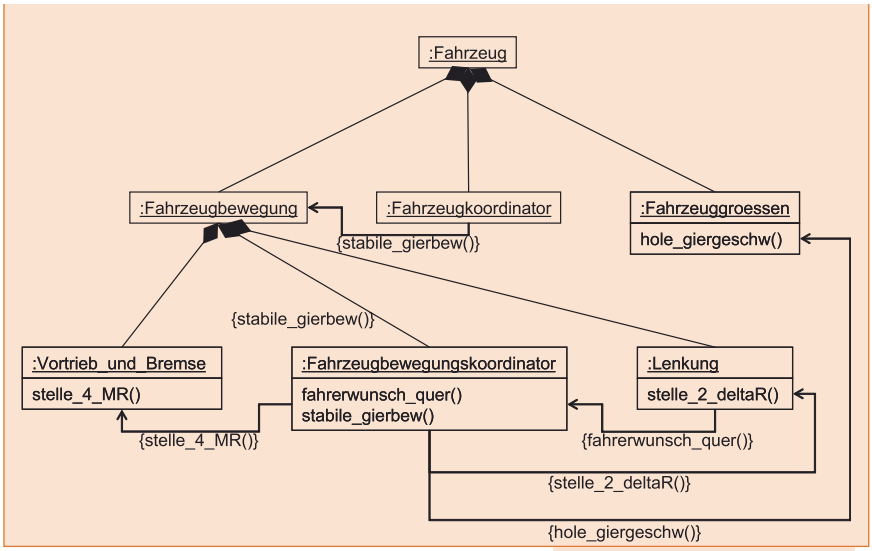

Abbildung 9:
Modell einer vereinfachten Fallstudie zur Gierstabilisierung.

Gekoppelte Modelle

In Zusammenarbeit mit einem Automobilzulieferer entstand unter Leitung von Hans-Dieter Kochs in der Arbeitsgruppe „Informationslogistik“ der Fakultät für Ingenieurwissenschaften der Universität Duisburg-Essen eine Fallstudie zur ESP-Gierstabilisierung. Ergebnis einer Problem- und Anforderungsanalyse waren die in Abbildung 9 dargestellten Komponenten und deren Beziehungen zueinander. Aufgabe der Gierstabilisierung ist es, das Schleudern und Kippen des Fahrzeugs zu verhindern. Die Gierbewegung beziehungsweise die Giergeschwindigkeit ist die Winkelgeschwindigkeit um die Hochachse eines gedachten fahrzeugfesten Koordinatensystems. Ein solches Subsystem interagiert zum Beispiel mit den Subsystemen `Lenkung` sowie `Vortrieb_und_Bremse`.

Ein elektronischer Sensor erfasst kontinuierlich Messwerte für die aktuelle Giergeschwindigkeit (`hole_giergeschw`), die im Fahrzeugbewegungskoordinator softwaremäßig verarbeitet und ausgewertet werden. Dieser aktiviert dann, falls notwendig, hydraulische Aktoren, die zum Beispiel über die automatische Betätigung der Bremse Momente an den vier Rädern stellen (`stelle_4_MR`) oder die beiden Lenkwinkel der Vorderräder ändern (`stelle_2_deltaR`).

EXPERTENWISSEN

Zustandsübergangsdiagramm

Ein solches Diagramm beschreibt alle möglichen Zustände und Zustandsübergänge des Systems. Zustände werden dabei durch Kreise oder Rechtecke mit abgerundeten Ecken symbolisiert, Übergänge durch Pfeile. Solche Diagramme helfen, ein System genau zu spezifizieren, besser zu verstehen und auch zu verifizieren. Zustandsübergangsdiagramme mit wenigen Zuständen sind überschaubar, und man kann leicht sehen, ob korrektes Verhalten gewährleistet ist. Bei realen Systemen können diese Diagramme jedoch schnell sehr groß werden. Da diese Diagramme vom Menschen nicht mehr zu überblicken sind, benötigt man computerunterstützte Techniken wie Model-Checking.

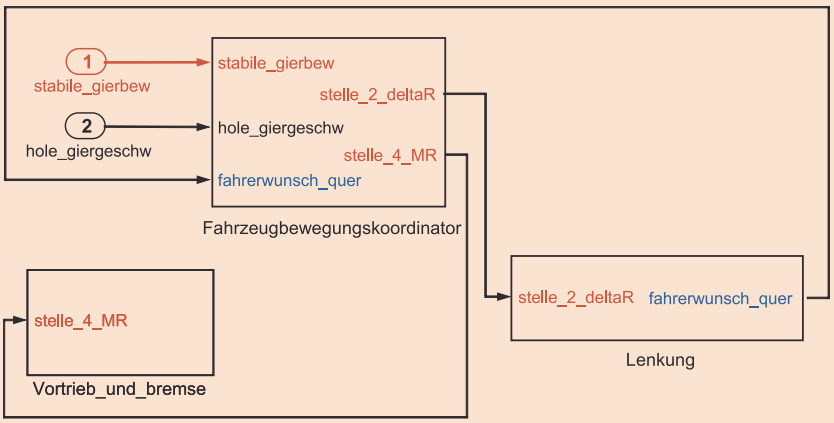

Abbildung 10: Blockschaltbild für die Komponente Fahrzeugbewegung.

Für eine verhaltensorientierte Sicht auf Regelungsaufgaben wie diese eignen sich vor allem datenflussorientierte Modellierungsmethoden. Hierbei interessiert besonders die zeitlich-logische Anordnung von Komponenten, die sich in Blockschaltbildern niederschlägt. Diese bestehen aus Funktionsblöcken mit Signalformungseigenschaften sowie den Signalflüssen zur Übertragung von Datenströmen. Abbildung 10 zeigt ein solches Blockschaltbild für die Komponente `Fahrzeugbewegung`.

Eingebaut in ein umgebendes Gesamtmodell mit einem Umwelt-, Fahrer- und Fahrzeugmodell erhält man eine vollständige Simulationsumgebung, die für eine systematische Entwicklung, Verifizierung und Validierung des Reglers benötigt wird (Abb. 11).

Ein mögliches Ergebnis einer solchen Simulation zeigt Abbildung 12, in der ein Spurwechsel auf rutschiger Fahrbahn ohne und mit Gierstabilisierung gegenübergestellt ist. Oben links ist der vom Fahrermodell vorgegebene Lenkradeinschlag dargestellt. Ohne Gierstabilisierung (blaue Kurven unten) rutscht das Fahrzeug bei einer zu großen Giergeschwindigkeit nach links weg, während mit Regelung (rote Kurven unten) die Giergeschwindigkeit in einem Intervallbereich um den Wert 0 gehalten und der simulierte Spurwechsel ohne Wegrutschen durchgeführt werden kann.

In-the-Loop-Tests

Neben solchen Model-in-the-Loop-Tests auf dem Rechner besteht in speziell ausgestatteten Testfahrzeugen mit Notebookanschluss die Möglichkeit, nach einem so genannten Software-in-the-Loop-Test Prototyp-Software in realer Straßenumgebung auf nicht-öffentlichem Versuchsgelände zu testen. Auf Basis solcher Machbarkeitsstudien werden üblicherweise nach Entwicklung, Bau und Programmierung der Subsysteme Tests in einer echtzeitfähigen Hardware-in-the-Loop-Umgebung durchgeführt. Dazu wird die meist neu entwickelte Hardware zusammen mit der Software als ein sogenanntes Steuergerät auf einem Prüfstand mit anderen konkreten Teilrealisierungen und/oder einem Software-Streckenmodell getestet, bevor diese letztendlich dann im Fahrzeug selbst eingebaut werden.

Eine möglichst durchgängige automatisierte Kopplung solcher verschiedenen Modelle über den gesamten Entwicklungszyklus der Produkte ist dabei eine essentielle Aufgabe, um die Konsistenz einer Vielzahl verschiedener existierender Modelle aufrechtzuerhalten. Aus den unterschiedlichsten Gründen gibt es immer wieder einzelne Änderungen und Erweiterungen, die dann in allen Modellen nachgeführt werden müssen. Andernfalls besteht die große Gefahr, dass solche Änderungen unter Umständen große Auswirkungen auf andere Funktionalitäten bis hin zu deren Versagen haben können. Die Komplexität dieser Aufgabe ist enorm, da viele hundert Menschen in zahlreichen verschiedenen Projekt- und Entwicklerteams weltweit verteilt an den Subsystemen und letztendlich dem Gesamtsystem Fahrzeug mitarbeiten.

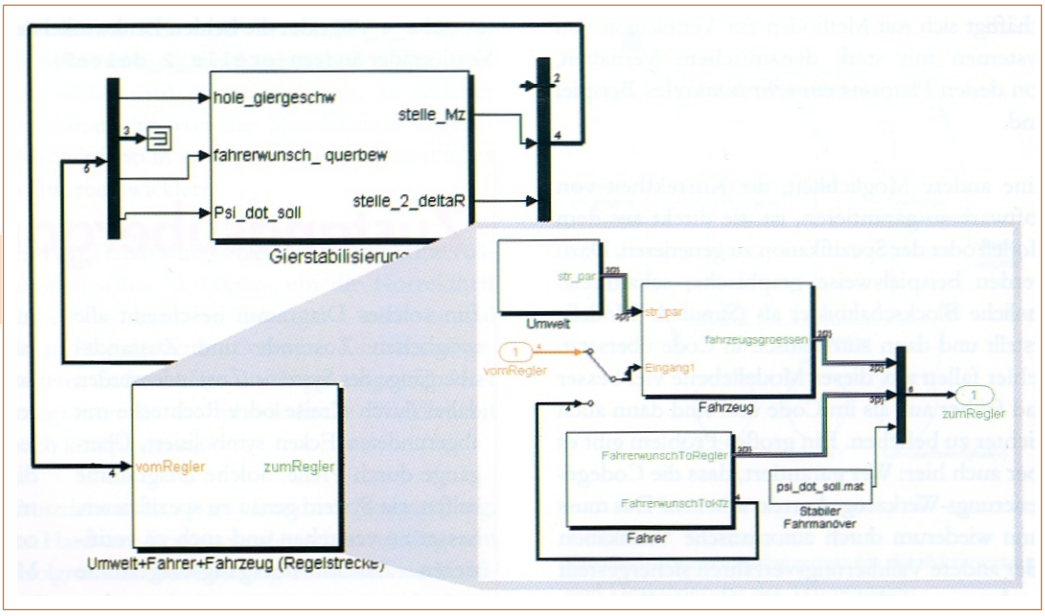

Abbildung 11: Model-in-the-Loop zur Simulation einer vereinfachten Gierstabilisierung.

Informatik ist überall

Projekte zur Entwicklung technischer Systeme werden in interdisziplinären Teams durchgeführt. Die Informatik ist dabei wie die Mathematik heutzutage überall – auf den ersten Blick manchmal auch unsichtbar – vertreten. Software für technische Systeme ist oft sicherheitskritisch und muss deswegen mit besonderer Sorgfalt entwickelt werden. Eine sorgfältige Analyse des Problems ist wichtige Voraussetzung für die Entwicklung einer adäquaten Lösung. Hierzu sind mehrere Schritte erforderlich, die insbesondere die Umgebung, in der die Software eingesetzt werden soll, berücksichtigen müssen. Bei jedem Schritt werden Dokumente entwickelt, die immer genauer werdenden Modellen der Software entsprechen. Auch wenn ihr Einsatz oft sehr aufwändig ist, sind Verifikations- und Validierungstechniken zur Gewährleistung von Qualität, Funktionalität und nicht zuletzt Sicherheit der Nutzer notwendig und unabdingbar.

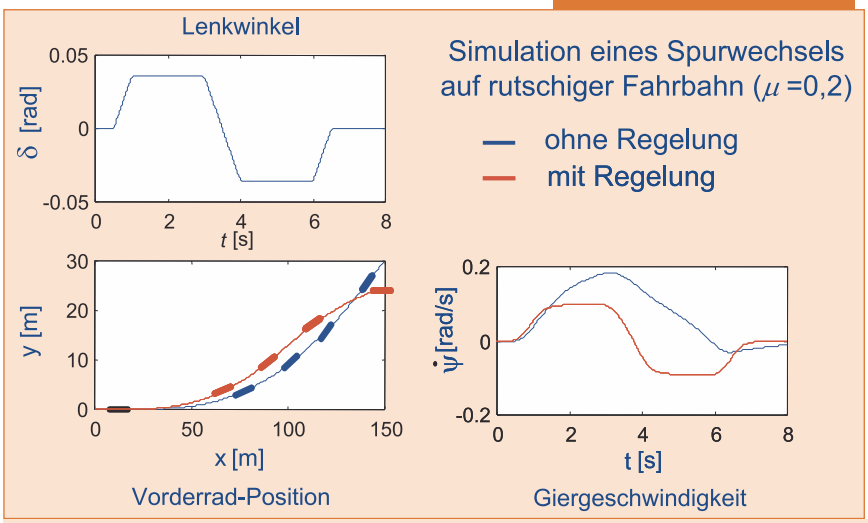

Abbildung 12: Fallstudie zur Simulation einer vereinfachten Gierstabilisierung.

Kontakt

Prof. Dr. rer. nat. Maritta Heisel
Software Engineering

Tel.: 02 03 / 3 79 - 34 65
maritta.heisel@uni-duisburg-essen.de
http://swe.uni-duisburg-essen.de/

Prof. Dr. rer. nat. Barbara König
Theoretische Informatik

Tel.: 02 03 / 3 79 - 33 97
barbara_koenig@uni-duisburg-essen.de
http://www.informatik.uni-duisburg.de/AGThInf/

Prof. Dr.-Ing. H.-D. Kochs
Informationslogistik

Tel.: 02 03 / 3 79 - 22 04
hans-dieter.kochs@uni-duisburg-essen.de

Dr.-Ing. Dipl.-Inform. J. Petersen
Informationslogistik

Tel.: 02 03 / 3 79 - 34 82
joerg.petersen@uni-duisburg-essen.de

http://www.uni-duisburg-essen.de/informationslogistik/