Softwaretechnik und Methodik [1]

Maritta Heisel

Zusammenfassung Ich lege dar, warum sich meiner Meinung nach die Softwaretechnik noch in einem vorindustriellen Stadium befindet und skizziere meinen Beitrag zur Überwindung dieser Situtation.

Software and cathedrals are much the same:
first we build them, then we pray.
Sam Redwine

1 Ein undenkbares Szenario?

Stellen Sie sich vor, Sie hätten gerade ein neues Auto gekauft. Freudestrahlend fahren Sie vom Hof des Händlers und gehen auf Spritztour. Nach kurzer Zeit jedoch versagt die Lenkung des Autos. Gerade noch rechtzeitig vor einer Kurve kommen Sie zum Stehen. Erbost und geschockt gehen Sie zur nächsten Telefonzelle und rufen Ihren Autohändler an. Dort sagt man Ihnen, daß es sich wohl um einen Konstruktionsfehler handelt, den man beim Nachfolgemodell sicher beheben werde. Für Sie bestehe jetzt aber nur die Möglichkeit, den Fehler auf Ihre Kosten provisorisch zu beheben zu lassen.

Undenkbar? Für Autos sicherlich, nicht aber für Software. Softwarehersteller geben keinerlei Garantie für das korrekte Funktionieren ihrer Produkte. Wenn Sie Probleme haben, dürfen Sie gerne die Hotline anrufen. Ihre Fehlermeldungen werden für die nächste Version dankend entgegengenommen. Ersatz für verlorene Daten und Geschäftsausfallzeiten leistet Ihnen jedoch niemand. Würde man die auf dem Softwaremarkt herrschende Situation auf Autos übertragen, so würde ein Auto unter 1000 Euro kosten. Allerdings würden von Zeit zu Zeit Autos einfach ohne äußeren Grund explodieren.

Sie könnten nun einwenden, daß der eingangs geschilderte Vergleich unzulässig sei, weil der Ausfall der Lenkung eines Autos viel gefährlicher sei als der Absturz eines Computers. Dies ist leider falsch. Durch fehlerhafte Software sind schon Menschen ums Leben gekommen [17]: Das Bestrahlungsgerät Therac-25 verabreichte zwischen Juni 1985 und Januar 1987 in mindestens sechs Fällen Patienten eine massive Überdosis radioaktiver Strahlung.

Außer der Gefährdung von Menschenleben entstehen durch Softwarefehler auch finanzielle Schäden in Millionenhöhe: Der durch Softwarefehler verursachte Absturz der Ariane-5-Rakete[2]

[1] Niederschrift einer *gedachten* Antrittsvorlesung. Der am 11.12.1997 gehaltene Vortrag hatte den Titel „Systematische Softwareentwicklung mit Agenden und Strategien".

[2] siehe http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html

kostete die europäischen Steuerzahler 500 Millionen Dollar. Die Inbetriebnahme der neuen Flughäfen in Denver[3] und Hongkong[4] wurde ebenfalls von millionenteuren Computerproblemen überschattet, und der Telefonriese AT&T geriet nur deshalb nicht in existenzbedrohende finanzielle Schwierigkeiten, weil der softwarebedingte Ausfall eines Telefonvermittlungsknotens auf einen Feiertag fiel. Softwarefehler[5] sind also nicht so harmlos, wie sie dem ab und zu frustrierten, sonst aber begeisterten PC-Benutzer erscheinen mögen.

2 Softwaretechnik im vorindustriellen Stadium

In früheren Zeiten wurde die Standfestigkeit von Bauwerken durch Tests überprüft. Beispielsweise soll die Dresdner Elbbrücke „Das blaue Wunder" getestet worden sein. Heute entspricht ein solches Verfahren nicht mehr dem Stand der Technik und würde sicherlich massive Proteste der Bevölkerung hervorrufen. Anstatt die Bauwerke erst zu bauen und dann durch Tests auf Standfestigkeit zu überprüfen, bedient man sich heute *mathematischer* Techniken, die es gestatten, im vorhinein *auszurechnen*, ob das Bauwerk standfest ist oder nicht.

Die Softwaretechnik hat dieses Stadium der Reife noch nicht erreicht. Testen ist heute immer noch das wichtigste Mittel der Softwarequalitätssicherung, und das obwohl, wie E. Dijkstra richtig bemerkte, Testen nur die Anwesenheit, niemals aber die Abwesenheit von Defekten zu zeigen vermag. Obwohl auch für Software inzwischen mathematische Techniken entwickelt worden sind, mit denen erwünschte Eigenschaften *positiv* nachgewiesen werden können, werden diese Techniken in der Praxis bisher kaum eingesetzt.

Die gegenwärtige Situation in der Softwareentwicklung ist von großer Unsicherheit und von Fehlschlägen gekennzeichnet. Gemäß einer Untersuchung der Standish Group aus dem Jahre 1995[6] werden 31% aller Softwareprojekte erfolglos abgebrochen, und 53% kosten 189% der ursprünglich geplanten Kosten. Nur 16% der Softwareprojekte werden rechtzeitig und ohne Kostenüberschreitung fertiggestellt. Bei großen Unternehmungen sind es sogar nur 9%.

Watts Humphrey schreibt in seinem Buch „A Discipline for Software Engineering" [16] (S.3):

> The intuitive software development methods generally used today are acceptable only because there are no alternatives. The current practice is nearer to a craft than an engineering discipline. ... Most finished software products usually can be made to work only after extensive testing and repair. From a scientific viewpoint, the process is distressingly unpredictable.

Worauf ist dieser beklagenswerte Zustand zurückzuführen? Die Antwort ist, daß Software und materielle Dinge wie Maschinen oder Gebäude fundamental verschieden sind. Nicht nur, daß einer dreitausendjährigen Erfahrung im Brückenbau eine nur etwa fünfzigjährige Erfahrung in der Softwareentwicklung gegenübersteht. In seinem vielzitierten Artikel „No Silver

[3] Die nichtfunktionierende Software zur Steuerung des Gepäcktransportsystems kostete die Stadt Denver $1,1 Mio pro Tag.

[4] Presseberichten zufolge wird die Hongkonger Wirtschaft durch den Zusammenbruch des Frachtverkehrs einen Schaden von mehr als einer Milliarde DM erleiden.

[5] Weitere Beispiele befinden sich in Peter Neumanns Buch [18] und der von ihm moderierten News-Gruppe *comp.risks*.

[6] siehe http://www.standishgroup.com/chaos.html

Bullet“ [2] nennt Frederick Brooks Faktoren, die dazu führen, daß existierende ingenieurmäßige Verfahren nicht auf die Entwicklung von Software zu übertragen sind:

Komplexität Software sei komplexer als andere von Menschen geschaffene Dinge, weil es keine genormten Teile gebe. Große Systeme entstünden aus kleineren nicht durch Wiederholung derselben Elemente.

Konformität Da Software häufig als letzte Komponente in bereits existierende Systeme eingeführt werde, müsse sie sich vorgefundenen Gegebenheiten anpassen, was zu weiterer Komplexität führe.

Änderbarkeit Software stehe unter einem permanten Änderungsdruck, im Gegensatz zu materiellen Produkten, die durch Weiterentwicklungen[7] abgelöst würden.

Unsichtbarkeit Software könne nicht visualisiert werden. Es gebe keine Entsprechung zu Bau- oder Konstruktionsplänen.

Weitere wichtige Unterschiede zwischen Software und technischen Bauteilen sind, daß Software sich nicht durch Gebrauch, sondern durch die sogenannte Wartung abnutzt und daß Software unstetig ist. Das heißt, daß eine kleine Änderung im Programmtext große Änderungen im Programmverhalten hervorrufen kann. Daraus folgt, daß man bei Software keine Sicherheitsmargen einbauen kann, wie dies bei Gebäuden und Maschinen üblich ist, wo beispielsweise Träger einfach ein bißchen dicker als unbedingt nötig gebaut werden.

Trotzdem wird häufig so verfahren, als könne Software genauso behandelt werden wie materielle Bauteile. Insbesondere bei sicherheitskritischen Systemen kann dies katastrophale Folgen haben. An Bord der Ariane-5 befanden sich beispielsweise zwei gleichartige Rechner, auf denen jeweils dasselbe (fehlerhafte) Programm ablief. Für materielle Systeme ist diese Art Redundanz angemessen, da man in vielen Fällen davon ausgehen kann, daß gleichartige Teile trotz ihrer Gleichartigkeit unabhängig voneinander ausfallen. Identische Software jedoch mehrfach parallel auszuführen bringt nicht die geringste Erhöhung der Zuverlässigkeit des softwarebasierten Anteils eines Systems.

Ein weiteres Beispiel für die Vernachlässigung des Unterschiedes zwischen Software und technischen Bauteilen sind die absoluten Zahlen, mit denen im allgemeinen die Zuverlässigkeit technischer Systeme angegeben wird („dieses Flugzeug hat eine Ausfallwahrscheinlichkeit von 10^{-9}“). Solche Zahlen ergeben sich aus Dauertests der technischen Teile, die nach einer gewissen Zeit wegen Materialermüdung ausfallen. Bei Software hingegen gibt es keine Ermüdung, sondern alle eventuellen Fehler sind schon von Anfang an vorhanden. Fehler, die mittels Tests entdeckt werden, werden behoben, wobei oftmals andere Fehler eingeführt werden. Die für das zuerst getestete System ermittelten Zuverlässigkeitsdaten lassen sich kaum auf das so veränderte System übertragen. Deshalb sind meiner Meinung nach absolute Zuverlässigkeitsangaben für softwarebasierte Systeme (und die meisten sicherheitskritischen Systeme sind heute softwarebasiert) äußerst fragwürdig.

Stellen wir also fest, daß die Softwaretechnik nicht einfach die Methoden traditioneller Ingenieurdisziplinen übernehmen kann, sondern daß zur ingenieurmäßigen Entwicklung von Software andere Methoden vonnöten sind, die ihre besondere immaterielle Natur berücksichtigen.

[7] In der Tat kämen sicherlich nur wenige Besitzer eines Trabant auf die Idee, nachträglich in ihr Fahrzeug Katalysator, Airbag und Antiblockiersystem einbauen zu wollen.

3 Von der Kunst zum ingenieurmäßigen Vorgehen

Um das vorindustrielle Stadium der Softwaretechnik zu überwinden und zu einer ingenieurmäßigen Vorgehensweise zu gelangen, sind meiner Meinung nach verstärkte Anstrengungen auf zwei Gebieten nötig: Erstens müssen mathematische Techniken zum integralen Bestandteil eines jeden Softwareentwicklungsprozesses werden. Zweitens müssen konkrete und aussagekräftige Vorgehensmodelle entwickelt werden, damit der Softwareentwicklungsprozeß weniger unvorhersagbar (Humphrey) wird, als dies bisher der Fall ist.

3.1 Mathematische Techniken

Softwarequalitätssicherung darf sich nicht im mehr oder weniger systematischen Finden und Beheben von Fehlern (bei gleichzeitigem Einbau neuer Fehler) aus einem beinahe zufällig entstandenen Produkt erschöpfen.

Stattdessen sollte die Software von vorneherein so kontruiert werden, daß – ähnlich wie bei Gebäuden oder Maschinen – gewünschte Eigenschaften postiv garantiert werden können. Die einzige Möglichkeit, dies zu erreichen, besteht in expliziten Einbeziehung der *Semantik* des zu entwickelnden Produktes in den Entwicklungsprozeß und damit der Anwendung mathematischer Techniken. Heutezutage ist jedoch die Situation eher dadurch gekennzeichnet, daß nicht einmal die gewünschten Eigenschaften des Softwareproduktes eindeutig und vollständig spezifiziert werden („You guys start coding while I'll go find out what the customer wants").

Natürlich sind mathematische Techniken kein Allheilmittel. Wenn beispielsweise ein in einer höheren Programmiersprache geschriebenes Anwendungsprogramm bezüglich einer Spezifikation als korrekt nachgewiesen ist, bedeutet dies noch lange nicht, daß es auch zur Zufriedenheit seiner Benutzer funktionieren wird. Die Spezifikation könnte die Benutzerwünsche ungenügend wiedergeben, Übersetzer oder Betriebssystem könnten Fehler enthalten, die Hardware könnte fehlerhaft arbeiten, oder die Benutzungsoberfläche des Programms könnte unergonomisch sein. Wie in anderen Ingenieurdisziplinen, wo z.B. auch eine korrekt geplante Brücke einstürzen kann, wenn minderwertiger Beton verwendet wird, können mathematische Techniken keine absolute Sicherheit bieten. Sie sollten deshalb informelle Vorgehensweisen (z.B. eingehende Benutzerbeteiligung, Tests) nicht ersetzen, sondern ergänzen.

Praktiker wenden oft ein, daß die Anwendung formaler Techniken[8] in der Softwareentwicklung zu aufwendig sei. Dies liegt teilweise daran, daß die zur Zeit zur Verfügung stehenden formalen Techniken noch nicht zur Gänze ausgereift sind. Anstatt sie jedoch als eine Notwendigkeit zu begreifen und an ihrer Verbesserung zu arbeiten, werden diese Techniken oftmals ganz und gar abgelehnt.

Zur Zeit besteht für Softwarehersteller in der Tat auch kein Anlaß, mathematische Techniken einzusetzen, denn der Markt belohnt die Entwicklung von Qualitätssoftware nicht. Oft ist der Termin der Markteinführung wichtiger als die Produktqualität, um Konkurrenten zuvorzukommen. Softwarehersteller haben kein Problem, ihre Kunden als Beta-Tester einzusetzen,

[8] Für die hier angestellten Überlegungen ist es nicht nötig, zwischen mathematischen und formalen Techniken zu unterscheiden. Der Unterschied zwischen beiden ist, daß formale Techniken immer auch einen Kalkül beinhalten sollten, in dem Sätze nach syntaktischen Regeln abgeleitet werden können. Eigentlich sollte in der Literatur mehr von mathematischen Techniken als von formalen Methoden die Rede sein, denn oftmals existiert weder ein Kalkül noch eine Methode, sondern nur eine formale Sprache mit Semantik. Da aber der überwiegende Sprachgebrauch diese Unterschiede nicht berücksichtigt, werde auch ich die beiden Worte „formal" und „mathematisch" wie Synonyme behandeln.

da sie hiervon keine Nachteile zu erwarten haben. Würden Softwarehersteller jedoch in gleicher Weise für ihre Produkte zur Haftung herangezogen wie Autohersteller, würde sich die Situation grundlegend ändern. Dann würde es sich lohnen, Software so zu entwickeln, daß gewünschte Qualitätskriterien garantiert werden können.

Diese „Goldgräbermentalität", die in der Softwareentwicklung weit verbreitet ist, äußert sich außerdem in der Tatsache, daß viele Softwareentwickler angelernte oder umgeschulte Kräfte sind, die über keine fundierte Informatikausbildung verfügen, sondern die in Kursen den Umgang mit einer gängigen Programmiersprache erlernt haben. Um wieder die Analogie zu den klassischen Ingenieurwissenschaften heranzuziehen: Wer würde gerne die Konstruktion von Gebäuden oder Maschinen angelernten Kräften überlassen? Eine fundierte Ausbildung ist hier selbstverständlich; ihre Notwendigkeit wird nicht ernsthaft in Zweifel gezogen[9]. Die Leichtigkeit, mit der auch Laien syntaktisch korrekte Programme produzieren können (die sogar für einige Eingaben zufriedenstellend funktionieren) führt dazu, daß die Schwierigkeiten bei der Softwareproduktion drastisch unterschätzt werden und bedingt eine wenig professionelle Arbeitweise, deren Folgen ich schon in den Abschnitten 1 und 2 geschildert habe.

In einem kürzlich erschienenen Artikel [19] stellt Parnas fest:

> Something has gone wrong in the field that we (improperly) call "software engineering". Software practitioners see their real job as something that they can do without using mathematics; they view mathematics or "formal methods" as a kind of add-on, an option.

Um also die Softwaretechnik aus dem vorindustriellen Zeitalter herauszuführen, müssen mathematische Techniken endlich als notwendiges „Handwerkszeug" anerkannt werden, das jeder kompetente Softwareingenieur beherrschen muß. Der Weg dorthin führt zum einen über eine Verbesserung der Techniken selbst, zum anderen über eine verbesserte Ausbildung.

Doch wie ich schon zu Anfang dieses Abschnitts andeutete, fehlt es bisher nicht nur an Kompetenz in der Anwendung mathematischer Techniken, sondern es fehlen ganz allgemein brauchbare Anleitungen, wie die technischen Teile des Softwareprozesses[10] durchzuführen sind. Dieser Mangel wirkt sich umso schlimmer aus, wenn mathematische Techniken angewendet werden sollen.

3.2 Methodische Unterstützung von Softwareentwicklungsaktivitäten

Wie bereits ausgeführt, haben sich in der Softwaretechnik noch keine befriedigenden Prozesse zur Durchführung von Softwareentwicklungsaktivitäten etablieren können. Humphrey [16] (S.3) analysiert die Situtation wie folgt:

> There are no disciplined frameworks, no sets of acceptable standards, no coaching systems, no conducted rehearsals. Even agreement on what would characterize "good" professional performance is lacking. Software engineers are left to figure out their own working methods and standards without the guidance and support that professionals find essential, for example, in sports, the arts, or medicine.

[9] Dies ist in der Informatik leider nicht der Fall. Die Berechtigung der Informatik als eigene Wissenschaft ist nicht allgemein anerkannt. Für manche Kritiker besteht Informatik darin, Ingenieuren das Programmieren beizubringen.

[10] Dies sind die Teile, die nicht das Projektmanagement betreffen, sondern die direkt mit dem entstehenden Softwareprodukt (und seiner Semantik!) zu tun haben. Mit den organisatorischen Aspekten der Softwareherstellung befaßt sich das Forschungsgebiet der Software-Prozeßmodellierung, siehe z.B. [7].

Humphreys Lösungsansatz ist der *persönliche Softwareprozeß*. Jede mit Softwareentwicklungsaufgaben betraute Person soll sich über die Tätigkeiten, die sie ausführt, Rechenschaft ablegen, Fehler festhalten und analysieren und so die eigene Arbeit optimieren. Der persönliche Softwareprozeß[11] konzentriert sich also auf *Individuen*, die in die Lage versetzt werden sollen, alle Softwareentwicklungsaktivitäten erfolgreich durchzuführen.

Mein eigener Ansatz zur Systematisierung der Softwareentwicklung geht dagegen davon aus, daß es für alle Softwareentwicklungsaktivitäten eine „methodische Essenz" gibt. Wenn diese explizit und kommunizierbar gemacht wird, können verschiedene Individuen eine Softwareentwicklungsaktivität in ähnlicher Weise ausführen.

Beide Ansätze ergänzen einander, denn um die „methodische Essenz" einer Softwareentwicklungsaktivität herauszuarbeiten, bedarf es Experten, die als Individuen die betreffende Aktivität sehr gut beherrschen. Im folgenden möchte ich meinen Ansatz näher beschreiben.

3.2.1 Das Agendenkonzept

Um „methodische Essenzen" für Softwareentwicklungsaktivitäten explizit zu repräsentieren, schlage ich das Konzept einer *Agenda* vor. Das Agendenkonzept baut auf der Beobachtung auf, daß der technische Teil der Softwareentwicklung aus verschiedenen Aktivitäten besteht, deren Ergebnis jeweils ein *Dokument* ist. Beispiele sind Pflichtenhefte, Programmspezifikationen, Programmcode oder Testfälle. Eine weitere Beobachtung ist, daß erfahrene Softwareingenieure die verschiedenen Aktivitäten besser und schneller ausführen können als Neulinge, da sie mit der Zeit ein fundiertes problemspezifisches *Wissen* erworben haben, wie die verschiedenen Aktivitäten am besten durchzuführen sind. Leider wird solches Expertenwissen bisher nicht explizit gemacht[12], was dazu führt, daß Neulinge alte Fehler wiederholen und das Rad neu erfinden müssen.

Agenden sind ein Mittel, Wissen über die Durchführung von (technischen) Softwareentwicklungsaktivitäten zu dokumentieren und damit kommunizierbar und wiederverwendbar zu machen. Eine Agenda gibt Anleitung, wie eine spezifische Softwareentwicklungsaktivität durchzuführen ist. Agenden können für viele verschiedene Aktivitäten und Kontexte aufgestellt werden. Beispiele sind Anforderungsermittlung, Spezifikationsakquisition, Softwareentwurf mittels Architekturstilen, objektorientierte Analyse und Entwurf, sowie die Entwicklung von Code aus Spezifikationen.

Technisch gesehen besteht eine Agenda aus einer Liste von Schritten, die auszuführen sind, um eine Softwareentwicklungsaktivität durchzuführen. Das Ergebnis der Aktivität ist ein Dokument, das in einer bestimmten Sprache (sei sie formal oder informell) ausgedrückt ist. Den Schritten einer Agenda können schematische Ausdrücke der Sprache zugeordnet sein, in der das Dokument ausgedrückt wird. Diese schematischen Ausdrücke repräsentieren eine mögliche Form, in der das Ergebnis des Schrittes ausgedrückt werden kann. Wenn die Agenda abgearbeitet wird, werden sie geeignet instantiiert oder angepaßt.

Agenden dienen aber nicht nur zur Anleitung von Softwareentwicklungsaktivitäten, sondern auch zur Qualitätssicherung der entwickelten Dokumente. *Validierungsbedingungen*, die den einzelnen Schritten zugeordnet sein können, repräsentieren notwendige semantische Be-

[11] Der persönliche Softwareprozeß wurde von Humphrey als eine Ergänzung des Capability Maturity Model [15] entwickelt, das sich mit Softwareprozessen auf Unternehmensebene befaßt.

[12] Seit kurzer Zeit gibt es Bestrebungen, spezielle Arten von Softwareentwicklungswissen wiederverwendbar zu machen. Beispiele sind Entwurfsmuster [8] und Softwarearchitekturen [20]. Diese Konzepte sind jedoch auf bestimmte Paradigmen bzw. Prozeßphasen beschränkt.

dingungen, die das Dokument erfüllen muß, um seinen Zweck zu erfüllen. Diese Bedingungen können informelle Erinnerungs- oder Merkhilfen für die Person sein, die die Agenda ausführt, sie können aber auch formale Beweisverpflichtungen beinhalten, die mit Maschinenunterstützung erledigt werden können.

Agenden werden als Tabellen dargestellt, wie in Tabelle 1 gezeigt. Die Schritte einer Agenda müssen nicht unbedingt in der angegebenen Reihenfolge bearbeitet werden. Zur Definition einer Agenda gehört auch ein Abhängigkeitsgraph, der angibt, welche Schritte voneinander abhängen und welche nicht.

Nr.	Schritt	Schema	Validierungs-bedingungen
i	informelle Beschreibung	schematischer Ausdruck der benutzten Sprache	$\vdash$ formal oder $\circ$ informell

Tabelle 1: Schema zur Darstellung von Agenden

Normalerweise wird ein Durchlauf durch eine Agenda nicht ausreichend sein, um eine befriedigende Produktqualität zu erhalten. Wie beim Spiralmodell der Softwareentwicklung [1] werden in der Praxis die Schritte mehrmals auszuführen sein.

3.2.2 Beispielagenda

Um das Agendenkonzept genauer zu illustrieren, möchte ich nun eine Beispielagenda vorstellen, die der formalen Spezifikation eingebetteter sicherheitskritischer Software dient. Diese Agenda wurde im Rahmen des Projektes ESPRESS[13] entwickelt und ist in [9] genauer beschrieben.

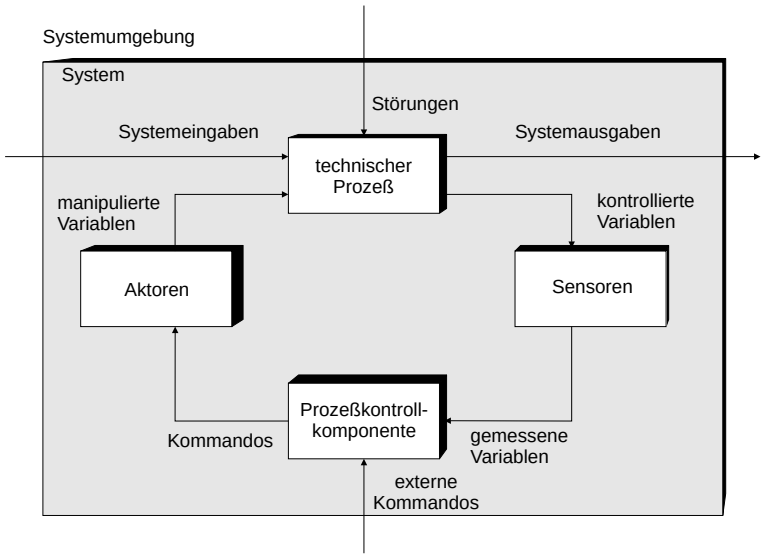


Abbildung 1: Eingebettete Systeme

[13] ESPRESS [6] ist ein vom BMBF gefördertes Verbundprojekt mit Partnern aus Industrie, Forschungseinrichtungen und Universitäten, das sich mit der ingenieurmäßigen Entwicklung eingebetteter sicherheitskritischer Systeme befaßt.

Abbildung 1 zeigt das Prinzip eingebetteter Systeme. Eine in Software realisierte Kontrollkomponente soll einen technischen Prozeß überwachen. Sensoren übermitteln relevante Daten an die Softwarekomponente, und mittels Aktoren kann die Softwarekomponente Einfluß auf den technischen Prozeß nehmen.

Solche Kontrollkomponenten können auf verschiedene Art und Weise realisiert werden. Eine sehr wichtige und in der Praxis weit verbreitete Klasse von Kontrollkomponenten ist die der *zyklischen* Kontrollkomponenten. Bei dieser Architektur werden in regelmäßigen Zeitabständen die Sensorwerte abgefragt und aufgrund dieser Sensorwerte Kommandos an die Aktoren gegeben, wie in Abbildung 2 gezeigt.

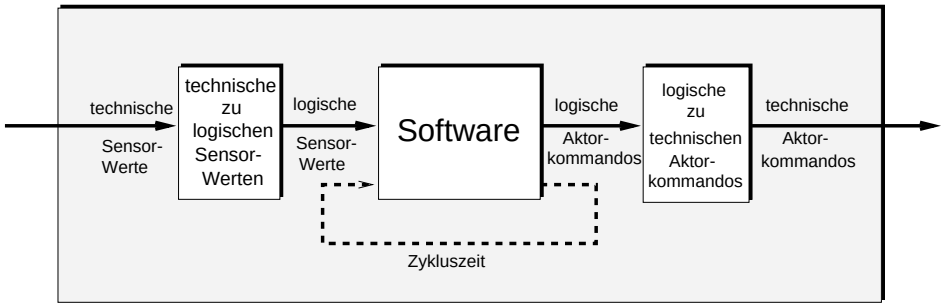


Abbildung 2: Zyklische Kontrollkomponente

Die im folgenden beschriebene Agenda gibt Anleitung zur Entwicklung einer formalen Spezifikation einer zyklischen Kontrollkomponente. Die verwendete Spezifikationssprache $\mu\mathcal{SZ}$ ist eine Kombination der Statemate-Sprachen (Statecharts und Activity charts) [10] mit der formalen Spezifikationssprache Z [21]. Da ich hier das Agendenkonzept im allgemeinen und weniger die konkrete Agenda besprechen möchte, ist ein genaueres Verständnis der Sprache nicht notwendig.

Die Spezifikation der zyklischen Softwarekomponente erfolgt in drei Phasen, wie in Tabelle 2 gezeigt. Phase 1 muß zuerst durchgeführt werden, während die Phasen 2 und 3 unabhängig voneinander sind, also in beliebiger Reihenfolge durchgeführt werden können. Für jede der Phasen ist eine eigene Agenda definiert.

Phase	
1	Kontexteinbettung
2	Sicherheitsanforderungen
3	Modellkonstruktion

Tabelle 2: Phasen der Agenda für zyklische Softwarekomponenten

Kontexteinbettung. Hier muß die Schnittstelle der Softwarekompoment mit dem Rest des Systems spezifiziert werden. Tabelle 3 zeigt die entsprechenden Schritte.

In Schritt 1.1 müssen die technischen Schnittstellen des Systems, die normalerweise vorgegeben sind, formal spezifiziert werden. Die Spezifikationen von Sensoren und Aktoren werden jeweils zusammengefaßt.

Schritt 1.2 ermöglicht es, eine *logische* Schnittstelle der Software zu ihrer Umgebung zu definieren. Dies ist dann nötig, wenn die technischen Sensor- bzw. Aktorwerte auf einem so

<table>
<tr><th>Schritt/Schema</th><th>Validierungsbedingungen</th></tr>
<tr><td>

1.1 Spezifiziere die technische Schnittstellen:

TechnicalDefs
. . . [. . .] . . .

TechnicalSensors
TechnicalDefs
*PORT S*1
. . .
. . .

TechnicalActuators
TechnicalDefs
*PORT A*1
. . .
. . .

</td><td>

○ die Wertebereiche der Sensoren und Aktoren sind mit den technischen Spezifikationen konsistent

○ Sensor- und Aktorfehler werden in Betracht gezogen

⊢ Invarianten sind konsistent

</td></tr>
<tr><td>

1.2 Spezifiziere die logischen Schnittstellen und ihren Zusammenhang zu den technischen Schnittstellen:

LogicalDefs
. . .

LogicalSensors
LogicalDefs
*PORT LS*1
. . .

LogicalActuators
. . .

*MapLS*1
LogicalSensors; . . .
BEHAVIOR
. . .

*MapLA*1
. . .

</td><td>

○ technische Sensorfehler werden in Betracht gezogen

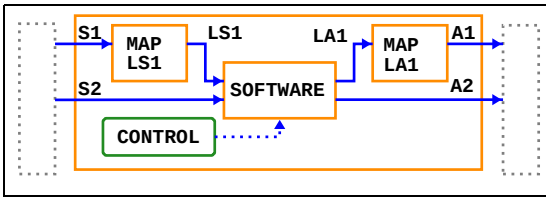
⊢ Invarianten sind konsistent

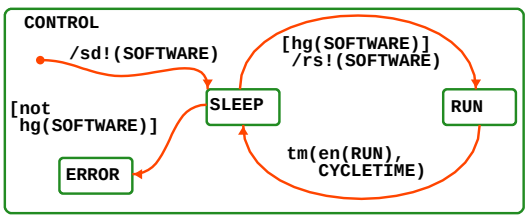
⊢ der Zusammenhang zwischen technischen und logischen Schnittstellen ist eindeutig und total

</td></tr>
<tr><td>

1.3 Spezifiziere den Informationsfluß zwischen Software und Kontext sowie die Zykluskontrolle:

Context
TechnicalSensors; *TechnicalActuators*
LogicalSensors; *LogicalActuators*





</td><td>

keine Validierungsbedingungen

</td></tr>
</table>

Tabelle 3: Agenda für Kontexteinbettung

niedrigen Abstraktionsniveau definiert sind, daß eine angemessene problemorientierte Spezifikation der Softwarekomponente nicht oder nur schwer möglich wäre. Die Abbildungen von technischen auf logische Sensorwerte und von logischen auf technische Aktorkommandos müssen eindeutig und total sein.

Schritt 1.3. erfordert keine Kreativität, sondern besteht nur in der geeigneten Zusammenfassung der bisher entwickelten Teile der Spezifikation. Das Activity chart (oben) stellt den Informationsfluß dar (vgl. auch Abbildung 2). Die Softwarekomponente muß so spezifiziert werden, daß sie die Sensorwerte liest, die Aktorkommandos berechnet und dann terminiert. Das Statechart `CONTROL` (unten) spezifiziert, daß die Softwarekomponente alle `CYCLETIME` Zeiteinheiten erneut aufgerufen wird.

Sicherheitsanforderungen. Der ESPRESS-Ansatz sieht vor, Sicherheitsanforderungen gesondert zu spezifizieren. Technisch gesehen handelt es sich um *Eigenschaften*, die das in Phase 3 der Agenda konstruierte Modell der Softwarekomponente erfüllen muß. Diese Redundanz trägt dazu bei, daß alle wichtigen Eigenschaften korrekt erfaßt und realisiert werden. Die Schritte für Phase 2 sind in Tabelle 4 gezeigt.

Schritt/Schema	Validierungsbedingungen
2.1 Stelle die relevanten Sicherheitsanforderungen zusammen.	○ die Anforderungen sind realisierbar
2.2 Spezifiziere Modelleigenschaften: *Software* *Obs*1 … *Obs*2 … … *PROPERTY DYN* … ⌈ *Obs*1 ⌉ ⌢ ⌈ *Obs*2 ⌉ …	○ die Beobachtungen ergeben eine problemorientierte Klassifikation möglicher Situationen ⊢ die Eigenschaften sind konsistent

Tabelle 4: Agenda für Definition der Sicherheitsanforderungen

Die Sprache $\mu\mathcal{SZ}$ ermöglicht es, Modelleigenschaften als zeitliche Beobachtungen in einer Temporallogik auszudrücken. Diese Beobachtungen sollen Abstraktionen gängiger an den Schnittstellen beobachtbarer Situationen sein. Ein Beispiel für eine Sicherheitsanforderung an einen Tempomat ist die Forderung, daß dieser die Geschwindigkeit des Autos nur dann regeln darf, wenn er einmal eingeschaltet und zwischenzeitlich nicht wieder deaktiviert wurde.

Modellkonstruktion. Um ein Modell der zyklischen Softwarekomponente zu definieren, gibt es verschiedene Vorgehensweisen. Die Komponente kann z.B. in verschiedene Subsysteme zerlegt werden (siehe [9]). Oftmals ist es auch sinnvoll, verschiedene *Betriebsmodi* zu unterscheiden, in denen die Komponente auf Eingaben von außen jeweils verschieden reagieren kann (typische Betriebsmodi sind z.B. *aktiv*, *passiv*, *Gefahr*). Tabelle 5 zeigt die Schritte zur Definition einer Architektur, die an Betriebsmodi orientiert ist.

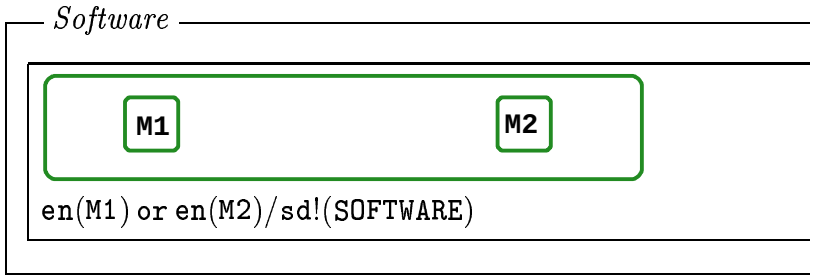
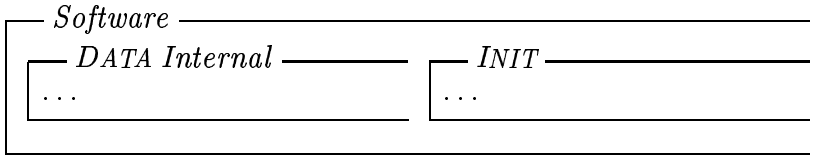
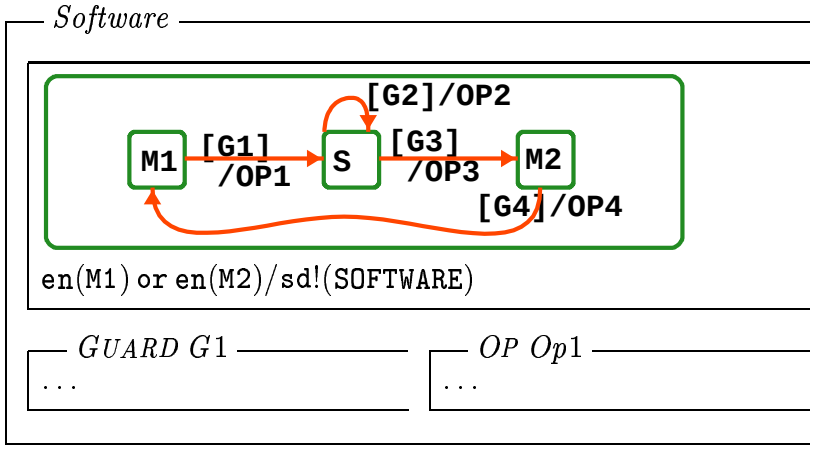
Schritt/Schema	Validierungsbedingungen
3.1 Definiere Betriebsmodi mit initialem Statechart: 	*keine Validierungsbedingungen*
3.2 Definiere internen Datenzustand: 	⊢ Invarianten sind konsistent ⊢ initialer Datenzustand existiert
3.3 Definiere Übergänge zwischen Betriebsmodi: 	○ für jeden Betriebsmodus werden die erwarteten Eingaben systematisch behandelt ⊢ wenn zwei verschiedene Übergänge einen Zustand verlassen, schließen sich ihre Guards aus ⊢ alle Zustände sind erreichbar ⊢ alle Betriebsmodi sind von allen anderen Betriebsmodi mit einer endlichen Anzahl von Übergängen erreichbar

Tabelle 5: Agenda für Modellkonstruktion

In Schritt 3.1 wird entschieden, welche Betriebsmodi notwendig sind. Diese werden in einem Statechart ohne Übergänge festgehalten. Der Ausdruck `en(M1) or en(M2)/sd!(SOFTWARE)` spezifiziert, daß die Softwarekomponente jedes Mal suspendiert wird, wenn ein Betriebsmodus angenommen wurde. Damit wird der Umgebung mitgeteilt, daß ein Zyklus beendet ist.

In Schritt 3.2 wird der interne Datenzustand definiert, den die Softwarekomponente benötigt, um die richtigen Aktorkommandos ermitteln zu können.

In Schritt 3.3 schließlich wird definiert, unter welchen Bedingungen ein Betriebsmodus `Mi` bzw. ein Hilfszustand `S` verlassen wird (Guards `Gi`) und welche Operationen bei dem Übergang

auszuführen sind (Operationen `OPi`).

Eine genauere Beschreibung der Agenda sowie ihre Anwendung zur Spezifikation eines intelligenten Tempomaten sind in [9] zu finden.

Ich erwarte nicht, daß Sie nach der Lektüre dieses Abschnitts in der Lage sind, zyklische Softwarekomponenten in $\mu\mathcal{SZ}$ zu spezifizieren. Die Beispielagenda sollte jedoch einen Eindruck vermittelt haben, welche Art der der methodischen Anleitung Agenden bieten können. Insbesondere ist festzustellen, daß

- die Schritte der Agenda und die schematischen Ausdrücke detailliert und substantiell sind,
- die Struktur der Spezifikation nicht von der spezifizierenden Person entwickelt werden muß, sondern durch die Agenda festgelegt wird,
- die Validierungsbedingungen auf die Architektur der Softwarekomponente abgestimmt sind und dazu beitragen, häufig auftretende Fehler zu vermeiden.

3.2.3 Agendenbasierte Softwareentwicklung

Wie wird nun eine agendenbasierte Softwareentwicklung durchgeführt? Die Person, die eine Softwareentwicklungsaktivität durchführen will, muß zunächst eine geeignete Agenda für die betreffende Aufgabe auswählen. Im allgemeinen werden verschiedene Agenden zur Verfügung stehen, die unterschiedliche Lösungsansätze repräsentieren. Dieser erste Schritt ist eine weitreichende Entscheidung und erfordert ein gutes Verständnis der zu bearbeitenden Aufgabe.

Ist die Auswahl einer geeigneten Agenda erfolgt, so ist der weitere Prozeß zu einem hohen Grad festgelegt. Jeder Schritt der Agenda muß ausgeführt werden, und zwar in einer Reihenfolge, die die Abhängigkeiten zwischen den Schritten respektiert. Die informelle Beschreibung des Schrittes gibt an, was zu tun ist. Die schematischen Ausdrücke stellen eine mögliche Form dar, wie das Ergebnis des Schrittes aussehen könnte bzw. sollte. Sie müssen „nur“ noch geeignet instantiiert bzw. angepaßt werden.

Falls für einen Schritt Validierungsbedingungen vorhanden sind, sollten diese sofort nachgewiesen werden, um unnötige Sackgassen in der Entwicklung zu vermeiden.

Wenn alle Schritte der Agenda (eventuell mehrmals) ausgeführt wurden, ist ein Produkt entstanden, von dem gewisse anwendungsunabhängige Qualitätskriterien garantiert werden können. Nun sollten sich anwendungsabhängige Validierungsmaßnahmen anschließen.

Mit dem Agendenkonzept verfolge ich *nicht* das Ziel, Softwareentwicklungsaktivitäten zu automatisieren (in dem Sinne, daß die Aktivität vollautomatisch ablaufen kann), sondern mein Bestreben ist es, Softwareentwicklungsprozesse in geordnete Bahnen zu lenken. Problemverständnis und Kreativität können und sollen nicht ersetzt werden.

Agenden strukturieren Softwareentwicklungsaktivitäten. Die Entwicklung eines Dokumentes mittels einer Agenda erfolgt immer durch Ausführung der betreffenden Schritte. Damit werden die durch Agenden beschriebenen Prozesse *standardisiert*[14]. Dies gilt jedoch nicht nur für die Prozesse, sondern auch die Produkte: Alle mit einer Agenda entwickelten Dokumente

[14]Hiermit meine ich nicht Standards wie sie DIN oder ISO herausgegeben werden, sondern ganz allgemein eine Gleichartigkeit.

haben dieselbe Struktur. Diese Tatsache macht die Dokumente für Außenstehende leichter verständlich und erleichtert auch deren Änderung.

Eine ausführlichere Diskussion des Agendenkonzeptes findet sich in [12]. Weitere Agenden für verschiedene Softwareentwicklungsaktivitäten sind in [11, 13, 14, 22] zu finden.

4 Das softwaretechnische Paradies

Sind Agenden nun als Wegbereiter zum softwaretechnischen Paradies anzusehen? Wohl kaum. Wie Brooks [2] bin ich der Meinung, daß es für die Softwaretechnik kein Allheilmittel („silver bullet") geben kann. Doch gibt es Zwischenstufen zwischen Chaos und Paradies. Auch ohne Allheilmittel sollte es möglich sein, Software auf eine Weise zu entwickeln, die die Bezeichnung „ingenieurmäßig" wirklich verdient. Außer den im vorigen Abschnitt genauer ausgeführten halte ich die im folgenden Faktoren für essentiell, um dieses Ziel zu erreichen.

Benutzerpartizipation. Software wird oftmals ohne Beteiligung der Menschen entwickelt, die sie nachher benutzen müssen. Beispielsweise wurden die Piloten bei der Entwicklung der Software für den Airbus A320 nur ungenügend beteiligt. Nachdem sich am 14.9.1993 aufgrund eines Fehlers in der Umsetzung von Anforderungen in eine Systemspezifikation[15] in Warschau ein tödlicher Unfall ereignet hatte, waren die Piloten gezwungen, die Software „auszutricksen"[16], um das von ihnen gewünschte Verhalten des Flugzeuges zu erreichen.

Auch wenn nicht alle Fälle mangelnder Benutzerbeteiligung ein derartiges Gefahrenpotential bergen, so ist doch laut der schon zitierten Untersuchung der Standish Group dieser Mangel eine der Hauptursachen für das Scheitern von Softwareprojekten. Und wem wurde nicht schon einmal ein gewünschter Dienst mit den Worten verweigert: „Das geht leider nicht, unser Computer läßt das nicht zu."?[17] Gegen die ungenügende Erfassung und Beachtung von Benutzer- und Kundenbedürfnissen helfen weder mathematische Techniken noch die systematische Durchführung von Softwareentwicklungsaktivitäten.

Denken als produktive Tätigkeit. Ein weiterer Faktor betrifft die organisatorische Durchführung von Softwareprojekten. Normalerweise wird der Fortschritt eines Softwareprojektes an den produzierten Dokumenten gemessen. Wurden längere Zeit keine Dokumente produziert, scheint das Projekt stillzustehen. Mitarbeiter, die einfach am Schreibtisch sitzen und weder die Rechnertastatur noch das Telefon bedienen, machen sich verdächtig. Aufgrund dessen wird es den Softwareingenieuren oftmals nicht ermöglicht, in ihrer Arbeitszeit schlicht und einfach *nachzudenken*, da dies einen unproduktiven Eindruck macht.

Während eines eingeladenen Vortrages auf der siebzehnten *International Conference on Software Engineering* (ICSE) 1995 in Seattle, in dem über ein bekanntes amerikanisches Softwarehaus berichtet wurde, fiel sinngemäß die Bemerkung, wenn neue Konzepte zu entwickeln

[15] Die Anforderung lautet, daß erst dann Gegenschub gegeben werden darf, wenn das Flugzeug gelandet ist. Als Kriterium hierfür wurde festgelegt, daß die Räder sich drehen müssen. In Warschau aber war die Landepiste wegen Regens naß, so daß sich die Räder wegen Aquaplaning nicht drehten. Der Pilot konnte deswegen keinen Gegenschub geben, und das Flugzeug schoß über die Piste hinaus.

[16] Der „Trick" bestand darin, nur halben Gegenschub zu geben, was von der Software nicht verweigert wurde.

[17] Ein aktuelles Beispiel aus der Presse: Banken verweigern karitativen Organisationen die Benutzung der Kontonummern 202 und 505 mit der Begründung, daß seit der Umstellung auf Elektronik die Überweisung auf die Kurzkontennummern nicht mehr praktikabel sei.

seien, nähme sich der Chef des Hauses einmal zwei Wochen Urlaub. Der entwickelten Software würde es jedoch sicherlich zugute kommen, wenn auch Softwareinginieure zum Nachdenken ermutigt würden. Anscheinend gehen Manager noch in vielen Fällen davon aus, daß Softwareingenieure eigentlich nicht arbeiten wollen, sondern von ihnen dazu angehalten und entsprechend beaufsichtigt werden müssen. DeMarco und Lister [5] sehen das anders:

> The manager's function is not to make people work, it is to make it possible for them to work.

Dies gelingt bisher allerdings nicht sehr gut, denn die Bedingungen, unter denen Softwareentwickler arbeiten müssen, fördern in vielen Fällen nicht gerade die Qualität der entstandenen Produkte. Großraumbüros sind immer noch an der Tagesordnung, und das zum Schaden der betreffenden Unternehmen. DeMarco und Lister [4] haben einen engen Zusammenhang zwischen Arbeitsbedingungen und Arbeitsleistung von Programmierern festgestellt: Programmierer mit hoher Leistung hatten mehr Raum zur Verfügung, ihr Arbeitsplatz war ruhiger, und sie wurden weniger oft gestört als Programmierer mit niedrigerer Leistung. Auch wenn hier Ursache und Wirkung nicht klar zu unterscheiden sind, so gilt doch die Aussage von DeMarco und Lister [5]:

> The top performer's space is quieter, more private, better protected against interruption, and there is more of it .... Does it really matter to you ... whether quiet, space and privacy help your current people do better work or [alternatively] help you to attract and keep better people?

Danksagung Ich danke Thomas Santen für Diskussionen zu diesem Thema im allgemeinen und für seine Kommentare und Anregungen bezüglich dieses Papiers im besonderen.

Literatur

[1] B. W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, 1988.

[2] Frederick P. Brooks. No silver bullet – essence and accidents of software engineering. *IEEE Computer*, S. 10–19, April 1987.

[3] Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering. Anniversary Edition.* Addison-Wesley, 1995.

[4] Tom DeMarco and Timothy Lister. Programmer performance and the effects of the workplace. In *Proceedings 8th International Conference on Software Engineering*, S. 268–272. ACM Press, 1985.

[5] Tom DeMarco and Timothy Lister. *Peopleware: Productive Projects and Teams.* Dorset House, 1987, zitiert nach [3].

[6] ESPRESS. Engineering of safety-critical embedded systems. Projektbeschreibung: http://www.first.gmd.de/~espress.

[7] A. Fuggetta and A. Wolf, Hsg. *Software Process.* Trends in Software 4. Wiley, 1996.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison Wesley, Reading, 1995.

[9] Wolfgang Grieskamp, Maritta Heisel, and Heiko Dörr. Specifying safety-critical embedded systems with Statecharts and Z: An agenda for cyclic software components. In E. Astesiano, Hsg., *Proc. ETAPS-FASE'98*, LNCS 1382, S. 88–106. Springer-Verlag, 1998.

[10] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, and Mark Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16 No. 4, April 1990.

[11] Maritta Heisel. *Methodology and Machine Support for the Application of Formal Techniques in Software Engineering.* Habilitationsschrift, TU Berlin, 1997.

[12] Maritta Heisel. Agendas – a concept to guide software development activites. In R. N. Horspool, Hsg., *Proc. Systems Implementation 2000*, S. 19–32, London, 1998. Chapman & Hall.

[13] Maritta Heisel and Jeanine Souquières. A heuristic approach to detect feature interactions in requirements. In Wiet Bouma, Hsg., *Proc. 5th Feature Interaction Workshop.* IOS Press Amsterdam, 1998. Wird erscheinen.

[14] Maritta Heisel and Jeanine Souquières. Methodological support for requirements elicitation and formal specification. In *Proc. 9th International Workshop on Software Specification and Design*, S. 153–155. IEEE Computer Society Press, 1998.

[15] Watts S. Humphrey. *Managing the Software Process.* Addison-Wesley, 1989.

[16] Watts S. Humphrey. *A Discipline for Software Engineering.* Addison-Wesley, 1995.

[17] Nancy G. Leveson and Clark S. Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, S. 18–41, Juli 1993.

[18] Peter G. Neumann. *Computer-Related Risks.* ACM Press, 1995.

[19] David L. Parnas. "Formal methods" technology transfer will fail. *The Journal Systems and Software*, 40:195–198, 1998.

[20] Mary Shaw and David Garlan. *Software Architecture.* IEEE Computer Society Press, Los Alamitos, 1996.

[21] J. M. Spivey. *The Z Notation – A Reference Manual.* Prentice Hall, 2nd edition, 1992.

[22] Kirsten Winter, Thomas Santen, and Maritta Heisel. An agenda for event-driven software components with complex data models. In W. Ehrenberger, Hsg., *Proceedings of the 16th International Conference on Computer Safety, Reliablity and Security (SAFECOMP).* Springer-Verlag, 1998. Wird erscheinen.