Universität Stuttgart
Institut für Automatisierungs- und Softwaretechnik
Prof. Dr.-Ing. P. Göhner

Institut für Automatisierungs- und Softwaretechnik
D-70550 Stuttgart

Prof. Dr.-Ing. P. Göhner

Universität Stuttgart - IAS
Pfaffenwaldring 47
D-70550 Stuttgart

Telefon (0711) 685-7301
Telefax (0711) 685-7302
e-mail: ias@ias.uni-stuttgart.de
www: http://www.uni-stuttgart.de
/UNIuser/iasinfo/

Komponentenbasierte Entwicklung Eingebetteter Systeme (KEES)

Arbeitsbericht von Juli 1997 bis Juni 1998

Nasser Jazdi

Inhaltsverzeichnis

1 Einleitung

1.1 Das Projekt „Komponentenbasierte Entwicklung Eingebetteter Systeme (KEES)“

Der ständige Preisverfall und die Leistungssteigerung der Hardwarekomponenten, vor allem der Mikroprozessoren und Speicher, erschließen ständig neue Einsatzmöglichkeiten für eingebettete Systeme. Dadurch steigt aber auch die Komplexität solcher Systeme, da immer neue Funktionen integriert werden. In einem PKW der Oberklasse sind heute bereits zwischen 50 und 100 Mikrocontroller integriert, die zusammen die Rechenleistung eines Großrechners zu Beginn der 80er Jahre aufweisen.

Viele komplexe Funktionen werden zunehmend in Software realisiert, um im Bereich der Hardware möglichst billige Standardkomponenten einsetzen zu können. Die Wertschöpfung bei der Neuentwicklung von Produkten verlagert sich also auch in diesem Bereich immer mehr in Richtung Software.

Während man in der Hardware immer mehr auf bereits bestehende Standardkomponenten zurückgreift, ist dies in der Software noch nicht in ausreichendem Maße gelungen. Dies hat verschiedene Gründe. Zum einen wird, wie bereits erwähnt, die individuelle Funktionalität der Systeme mehr und mehr in Software realisiert, zum anderen gibt es im Gegensatz zur Hardwareentwicklung keine allgemein verwendbaren Entwurfsmodelle und –methoden bei der Softwareentwicklung.

Daher wird die Software meist noch individuell entwickelt und teilweise von Hand optimiert, um auch noch das letzte Bit Speicherplatz auszunutzen und somit ein paar Pfennige zu sparen, die sich bei Millionen Stückzahlen zu hohen Einsparungen in der Produktion summieren. Mit dieser Vorgehensweise ist man jedoch dem raschen Anstieg der Komplexität der Software bald nicht mehr gewachsen.

1.2 Ziele des Forschungsvorhabens

Ziel des Forschungsvorhabens ist die Entwicklung neuer Methoden und Verfahren für die komponentenbasierte Entwicklung der Software für eingebettete Systeme. Dabei müssen die speziellen Anforderungen an eingebettete Systeme beachtet werden. Die Softwarekomponenten müssen auf einem hohen Abstraktionsniveau zur Verfügung gestellt werden, um eine individuelle Anpassung der Software an die jeweiligen Anforderungen unter möglichst optimaler Ausnutzung der Ressourcen wie Speicher und Prozessoren zu ermöglichen.

1.3 Arbeiten im Zeitraum Juli 1997 – Juni 1998

Im oben genannten Zeitraum wurden die drei Arbeitspakete *„Softwarekomponenten und Strukturierung eingebetteter Systeme“*, *„Entwurfsmethoden“* und *„Werkzeugunterstützung“* bearbeitet.

Die Anforderungen an eingebettete Systeme und Echtzeitsysteme wurden dabei analysiert. Ebenso wurden der Aufbau und die Funktionsweise von Mikrocontrollern für eingebettete Systeme untersucht.

Die verschiedenen Ansätze zu den Spezifikations- und Entwurfsmethoden für eingebettete Systeme wurden miteinander verglichen und auf ihre Anwendbarkeit zur Entwicklung neuer Methoden untersucht. Die weiteren Untersuchungen beziehen sich auf die Entwurfswerkzeuge. Hierbei wurden die Werkzeuge Statemate™ und ASCET-SD™ näher betrachtet.

Das Arbeitspaket *„Softwarekomponenten und Strukturierung eingebetteter Systeme“* wird als erstes bis zum ersten Quartal im Jahre 1999 abgeschlossen sein. Die weiteren Arbeitspakete werden gemäß dem Projektplan bis zum Jahre 2002 abgeschlossen sein.

2 Arbeitspaket „*Softwarekomponenten und Strukturierung eingebetteter Systeme*“

2.1 Definition eingebetteter Systeme

Unter einem *eingebetteten System (embedded system)* versteht man ein Mikrocomputer-basiertes System, das als Teil eines technischen Gerätes oder Produktes in dieses integriert ist. Ein eingebettetes System ist also immer Teil eines wohldefinierten größeren Systems, das oft als *intelligentes Produkt* bezeichnet wird. Ein solches intelligentes Produkt besteht aus einem mechanischen Subsystem, dem steuernden eingebetteten Computersystem und zumeist auch einer Mensch-Maschine-Schnittstelle. Eingebettete Systeme finden sich heute in vielen Produkten unseres täglichen Lebens wie Armbanduhren, Telefone, Mikrowellengeräte, Waschmaschinen, Videorecorder, Heizungsanlagen, Herzschrittmacher, bis hin zu Antiblockiersystemen, Klimaanlagen oder Motorsteuergeräten in Kraftfahrzeugen wieder.

Ein eingebettetes System besteht in der Regel wiederum aus einer Reihe verschiedener Komponenten. Diese lassen sich in drei Gruppen einteilen: analoge Hardware, digitale Hardware und Software. Der Anteil der digitalen Hardware wie Mikroprozessoren, Mikrocontroller, Signalprozessoren, programmierbaren Logik- und Speicherbausteinen sowie anwendungsspezifischen integrierten Schaltungen (ASICs) nimmt dabei gegenüber der analogen Hardware stetig zu. Dadurch wächst auch der Anteil der Software, da zunehmend programmierbare Standardbausteine eingesetzt werden, die in Massenproduktion gefertigt werden und in verschiedensten Varianten und von vielen Herstellern verfügbar sind.

In den letzten Jahren haben Einsatzbereiche und Anzahl eingebetteter Systeme sehr stark zugenommen. Der Markt für eingebettete Systeme wird angetrieben von der kontinuierlichen Verbesserung des Preis-/Leistungsverhältnisses der Halbleiterindustrie. Dadurch werden immer mehr mechanische, hydraulische oder auch elektronische Steuerungen durch wettbewerbsfähige Computer-basierte Systeme ersetzt. Es wird erwartet, daß der Markt für eingebettete Systeme in den nächsten zehn Jahren signifikant wachsen wird. Die Bereiche der Konsum- und Automobilelektronik zählen dabei zu den Schlüsselmärkten und fungieren aufgrund ihrer hohen Anforderungen an korrektes zeitliches Verhalten, Verläßlichkeit und Kosten als „Technologie-Katalysatoren“. Nach einer Studie aus dem Jahr 1994 [Ran94] wird dieser Markt im Vergleich zu anderen Märkten für Informationstechnologie die besten Beschäftigungsmöglichkeiten für Computer-Ingenieure der Zukunft bieten.

2.2 Eigenschaften eingebetteter Systeme

Eingebettete Systeme haben ein Reihe charakteristischer Eigenschaften [Kop97]:

Massenproduktion: Eingebettete Systeme sind in der Regel für einen Massenmarkt bestimmt und werden deshalb für die Massenproduktion auf hochgradig automatisierten Fertigungsanlagen entwickelt. Dies setzt voraus, daß der Produktionsaufwand einer einzelnen Einheit so gering wie möglich sein muß. Die effiziente Ausnutzung der Prozessoren und Speicher ist also von hoher Bedeutung.

Statische Struktur: Das Computersystem ist eingebettet in ein intelligentes Produkt mit vorgegebener Funktion und Struktur. Die vorab bekannte statische Umgebung des Systems kann zur Entwurfszeit analysiert werden, um die Robustheit zu erhöhen, die Effizienz zu steigern und die Software des eingebetteten Systems zu vereinfachen.

Mensch-Maschine-Schnittstelle: Wenn ein eingebettetes System eine Mensch-Maschine-Schnittstelle besitzt, dann muß dieses speziell für den vorgesehenen Einsatzzweck entworfen werden und muß einfach zu bedienen sein. Idealerweise sollte die Benutzung des intelligenten Produkts selbsterklärend sein und weder Schulung noch das Nachschlagen in einer Bedienungsanleitung erfordern.

Minimierung des mechanischen Subsystems: Um die Produktionskosten zu reduzieren und die Zuverlässigkeit des intelligenten Produkts zu erhöhen, wird versucht die Komplexität des mechanischen Subsystems zu minimieren.

Funktionalität in Software: Die Funktionalität eines intelligenten Produkts wird durch die integrierte Software bestimmt, die in einem Festspeicher abgelegt ist. Es gibt kaum eine Möglichkeit, diese Software nach der Freigabe zu modifizieren. Daher muß die Software hohen Qualitätsstandards genügen.

Wartbarkeit: Viele intelligente Produkte sind nicht wartbar, da die Partitionierung des Produkts in austauschbare Einheiten zu teuer ist. Wird ein Produkt jedoch so entworfen, um es während seines Lebenszyklus zu warten, so ist eine Diagnose-Schnittstelle und eine verständliche Wartungsstrategie von hoher Bedeutung.

Kommunikation: Obwohl die meisten intelligenten Produkte zunächst als eigenständige Einheiten auf den Markt kommen, wird an viele Produkte später die Anforderung gestellt, mit anderen größeren Systemen zu kommunizieren. Der Datentransfer sollte dabei über ein einfaches und robustes Protokoll geregelt werden. Die Optimierung der Übertragungsgeschwindigkeit spielt dabei selten eine Rolle.

In der Regel liegt der bei weitem größte Anteil der Kosten eines intelligenten Produkts über seinen gesamten Lebenszyklus gerechnet in der Produktion. Für das eingebettete System sind das die Materialkosten der Hardware. Die Entwicklungskosten und die Kosten der Software machen nur einen kleinen Anteil aus, der je nach Stückzahl des Produkts manchmal unter 5% liegen kann.

Die Wartungskosten können beträchtliche Größen annehmen, insbesondere wenn ein unentdeckter Entwurfsfehler (Softwarefehler) einen Rückruf aller ausgelieferten Einheiten und den Austausch einer kompletten Serie nach sich zieht.

2.3 Anforderungen an eingebettete Systeme

Aus den vielfältigen Anwendungs- und Einsatzgebieten eingebetteter Systeme resultieren eine Reihe von Anforderungen, die an ein bestimmtes System gestellt werden können. Diese umfassen neben funktionalen Anforderungen auch zeitliche Anforderungen, Anforderungen an die Verläßlichkeit und an die Produktionskosten.

2.3.1 Funktionale Anforderungen

Funktionale Anforderungen sind Anforderungen, die in Zusammenhang mit den Funktionen stehen, die ein eingebettetes System erfüllen muß. Diese Anforderungen können in die Kategorien Datenerfassung, digitale Steuerung und Regelung und Mensch-Maschine Interaktion eingeteilt werden. Zur Datenerfassung zählt z.B. die Erfassung, Aufbereitung und Filterung von Sensordaten und die Protokollierung von Fehlern und Alarmen. Viele eingebettete Systeme berechnen die Größen der Stellsignale für die Aktoren selbst und können nicht auf ein konventionelles Regelungssystem zurückgreifen. Regelungstechnische Funktionen bestehen aus einer unendlichen Folge von Regelungsperioden. In jeder Periode werden die Sensoren abgetastet, ein Regelungsalgorithmus aufgerufen, der die neuen Stellwerte berechnet und diese an die Aktoren ausgibt. Der Entwurf geeigneter Regelungsalgorithmen ist Thema der Regelungstechnik. Über die Mensch-Maschine-Schnittstelle werden Informationen zum aktuellen Zustand an den Benutzer ausgegeben und Befehle entgegengenommen. Diese muß wie bereits erwähnt einfach und intuitiv bedienbar sein und soll den Benutzer soweit wie möglich unterstützen. Während Datenerfassung und digitale Steuerung und Regelung oft Standardaufgaben sind, ist die Benutzungsschnittstelle für jedes Produkt individuell. Bei bereits am Markt eingeführten Produkten kann diese Benutzungsschnittstelle nicht ohne weiteres geändert werden. Aus dieser Tatsache resultieren zusätzliche Anforderungen für weiterentwickelte Produkte.

2.3.2 Zeitliche Anforderungen

Eingebettete Systeme sind meist auch Echtzeitsysteme, d.h. sie müssen bestimmte Bedingungen in Bezug auf ihr zeitliches Verhalten erfüllen. Die meisten dieser Anforderungen kommen von Regelschleifen, z.B. bei der Regelung der Einspritzung eines Benzinmotors. Die zeitlichen Anforderungen bezüglich der Benutzungsschnittstelle sind demgegenüber weniger strikt, da schon die menschliche Reaktionszeit mit 50-100 Millisekunden um Größenordnungen höher liegt als die Zeitkonstanten schneller Regelkreise.

Man unterscheidet zwischen *harter* und *weicher Echtzeit*. Die Zeitspanne, in der ein Echtzeitsystem auf ein bestimmtes Ereignis reagieren muß, wird durch seine Umgebung bestimmt. Der Zeitpunkt, zu dem eine bestimmte Reaktion erfolgen muß wird als *Deadline* bezeichnet. Ist eine Reaktion auch nach der Deadline noch akzeptabel, dann wird die Deadline als weich kategorisiert. Können durch eine nicht eingehaltene Deadline katastrophale Folgen eintreten, dann ist die Deadline hart. Ein Echtzeitsystem, das zumindest eine harte Deadline einhalten muß, ist ein hartes Echtzeitsystem bzw. ein sicherheitskritisches Echtzeitsystem. Gibt es keine harte Deadline, so handelt es sich um ein weiches Echtzeitsystem [Kop97].

Der Entwurf eines harten Echtzeitsystems unterscheidet sich grundlegend vom Entwurf eines weichen Echtzeitsystems. Während ein hartes Echtzeitsystem ein garantiertes zeitliches Verhalten in allen spezifizierten Last- und Fehlersituationen einhalten muß, ist die Verletzung einer Zeitbedingung (Deadline) bei einem weichen Echtzeitsystem tolerierbar. Dies wirkt sich insbesondere stark auf den Entwurf der Software aus.

2.3.3 Anforderungen an die Verläßlichkeit

Die Anforderungen an die Verläßlichkeit lassen sich untergliedern in Anforderungen bezüglich Zuverlässigkeit, Sicherheit, Wartbarkeit und Verfügbarkeit [Lap92].

Die *Zuverlässigkeit (reliability)* eines Systems ist die Wahrscheinlichkeit, daß ein System während einer bestimmten Zeitdauer gemäß seiner Spezifikation funktioniert. Hat ein System eine konstante Ausfallrate von l Ausfällen/Stunde, so kann die Zuverlässigkeit zum Zeitpunkt t wie folgt angegeben werden:

$$R(t) = exp(- l(t-t_0)),$$

wobei t-t0 in Stunden gegeben ist. Der Kehrwert der Ausfallrate $1/l = MTTF$ ist die mittlere Zeit bis zum Ausfall (mean-time-to-failure).

Sicherheit (safety) ist Verlässigkeit im Hinblick auf *kritische Ausfälle*. Ein kritischer Ausfall ist *schädlich* im Gegensatz zu einem nicht-kritischen Ausfall, der als *gutartig* bezeichnet werden kann. Ein kritischer Ausfall kann eine Gefahr für Material, Umwelt oder Menschen darstellen. Die Kosten eines solchen Ausfalls können um Größenordnungen höher sein, als die normalen Betriebskosten des Systems. Eingebettete Systeme finden immer mehr Einzug auch in sicherheitskritische Anwendungen, wie z.B. ABS oder Airbag-Systeme im Kraftfahrzeug. In vielen Fällen muß ein sicherheitskritisches System durch eine unabhängige Zertifizierungsstelle abgenommen werden.

Die *Wartbarkeit (maintainability)* ist ein Maß der Zeit, die benötigt wird, um ein System nach einem Ausfall wieder in den Zustand „betriebsbereit“ zu setzen. Die Wartbarkeit wird gemessen durch die Wahrscheinlichkeit $M(d)$, daß das System innerhalb der Zeit d nach Auftreten des Ausfalls wieder funktionsfähig ist.

Die *Verfügbarkeit (availability)* ist ein Maß der korrekten Funktionsfähigkeit eines Systems beim ständigen Wechsel zwischen funktionsfähigem und nicht funktionsfähigem Zustand. In Systemen mit konstanten Ausfall- und Reparaturraten besteht zwischen Zuverlässigkeit (*MTTF*), Wartbarkeit (*MTTR*) und Verfügbarkeit (*A*) folgender Zusammenhang:

$$A = MTTF / (MTTF + MTTR).$$

2.3.4 Anforderungen an Abmessungen, Gewicht und Kosten

Zusätzlich zu den oben genannten Anforderungen werden Anforderungen an physikalische Größen, wie Abmessungen und Gewicht und selbstverständlich an die Produktionskosten gestellt, welche die Auswahl verfügbarer Komponenten einschränken bzw. die Neuentwicklung von Komponenten erforderlich machen und somit den Entwurf beeinflussen können.

2.4 Komponentenbasierte Entwicklung eingebetteter Systeme

Um den Kostenaufwand der Software für eingebettete Systeme zu reduzieren und diese effektiver zu gestalten, sind Methoden für die Entwicklung der Software aus vorgefertigten Komponenten zu erarbeiten. Dabei geht es um Methoden, mit deren Hilfe die Unterstützung des komponentenbasierten Entwurfs der eingebetteten Systeme realisiert wird.

Das Gebiet der komponentenbasierten Softwareentwicklung ist derzeit weltweit Gegenstand aktueller Forschung. Trotz dieser globalen Bemühungen ist noch keine standardisierte Methode in Sicht, die den Programmierer bei der Erstellung und Verwendung von Komponenten unterstützen würde. Es herrscht jedoch allgemeine Übereinstimmung darüber, daß die Wiederverwendung möglichst frühzeitig im Entwicklungsprozeß eingeplant werden muß. Die Entwicklung der verschiedenen Methoden ist vergleichbar mit der Evolution im Bereich der objektorientierten Analyse- und Designmethoden, bei denen sich nach ca. zehnjähriger Entwicklung erst jetzt eine Standardisierung mit der Unified Modeling Language (UML) abzeichnet. Eine definierte Enwurfsmethode für eingebettete Systeme würde die komponentenbasierte Softwareentwicklung in diesem Bereich erheblich vorantreiben. Sie würde eine kostengünstige Herstellung der komplexen eingebetteten Systeme mit möglichst wenig Zeitaufwand ermöglichen.

2.4.1 Definition des Begriffs „Software-Komponente“

Existierende Definitionen

- Eine Komponente ist ein *Bestandteil eines Ganzen* (DUDEN Wörterbuch).

- Eine Komponente ist idealerweise ein *sprach*- und *plattformunabhängiges binäres Modul*, das - ähnlich wie ein Objekt - spezielle Funktionen und Daten kapselt, aber selbst nicht notwendigerweise objektorientiert implementiert ist. Anders als ein Framework verbirgt sie die Details ihres Aufbaus. (Eisenecker, 1996 [Eis97])

- Komponenten sind *austauschbare* Bestandteile eines Ganzen (Eisenecker, 1997).

- software components

    1. are ready „off-the-shelf“ components, whether from a commercial source (COTS) or re-used from another system
    2. have significant aggregate functionality and complexity
    3. are self-contained and possibly execute independently
    4. will be used „as is“ rather than modified
    5. must be integrated with other components to achieve required system functionality (A.W. Brown, SEI, 1996 [BrWa96])

- CORBA: Komponenten sind unabhängige binäre Bausteine. Sie sind nur abhängig von einer standardisierten Architekturkomponente (ORB). Diese steht für unterschiedliche Hardware-Plattformen zur Verfügung. Die Schnittstelle (funktional) wird sprachunabhängig in einer Schnittstellenbeschreibungssprache (IDL) beschrieben.

- Komponenten haben folgende Eigenschaften:

Komponenten, die nach einem bestimmten Muster (pattern) entworfen wurden, müssen in ein Framework passen, als wenn sie damit ausgeliefert würden.
Neue Komponenten können aus anderen Komponenten erzeugt werden durch Vererbung und Überschreiben von Teilen des Verhaltens.
Neue Komponenten können durch Zusammenfügen anderer Komponenten erzeugt werden (Aggregation).
Existierende Komponenten, die andere Komponenten beinhalten, können durch Austausch einer oder mehrerer dieser (Teil-)Komponenten modifiziert, das heißt in ihrem Verhalten verändert werden.
Komponenten sind plattformunabhängig und zumindest Source-Code kompatibel.
(Engel Hess, Rational Newsgroup)

- Reusable Software Component (RSC)

A software entity intended for reuse; may be design, code, or other product of the software development process. RSCs are sometimes called „software assets“.
(NATO Standard, 1992 [Nat92])

- Eine Komponente (Baustein) bietet eine hinreichend interessante Funktionalität, ist hinreichend gut dokumentiert, kann einfach benutzt werden und ist qualitativ hochwertig.

(Heuer-Hasenpatt et. all, 1997 [Heu97])

Eine Software-Komponente wird in diesem Bericht wie folgt angesehen:

Eine Software-Komponente ist ein komplexer, speziell für die Mehrfachverwendung konstruierter, qualitativ hochwertiger System-Baustein, der eine abgeschlossene Funktionseinheit bildet, hinreichend gut dokumentiert ist und der unverändert zur Konstruktion von Systemen einer oder mehrerer Anwendungsdomänen verwendet werden kann.

2.4.2 Muß-Kriterien bei Softwarekomponenten

- Die funktionale Einheit muß abgeschlossen sein.
- Die Schnittstellen müssen definiert und spezifiziert sein.
- Die Softwarekomponenten müssen hinreichend dokumentiert und verständlich sein.
- Sie müssen qualitativ hochwertig sein(Produktcharakter).
- Sie müssen einfach und unverändert verwendbar werden können.

2.4.3 Eigenschaften von Software-Komponenten

System-Baustein

Eine Komponente ist ein mehrfach verwendbarer *System-Baustein*. Das heißt, sie ist Bestandteil eines Ganzen, nämlich des jeweiligen Systems, in dem sie eingesetzt wird.

Abgeschlossene Funktionalität

Eine Komponente sollte eine in sich *abgeschlossene Funktionseinheit* darstellen. Bildet eine Komponente keine in sich abgeschlossene funktionale Einheit, so ist die Wahrscheinlichkeit, daß sie in verschiedenen Systemen eingesetzt werden, kann äußerst gering. Außerdem ist die Komponente dann nicht für sich allein verständlich. Eine Komponente deckt also einen bestimmten fachlichen oder technischen Kontext vollständig ab und beschreibt somit eine logisch zusammengehörige Gruppe von Funktionen.

Spezifikation der Schnittstellen

Den Schnittstellen kommt bei der komponentenbasierten Softwareentwicklung eine zentrale Bedeutung zu. Die Schnittstellen von Komponenten sollten daher *vollständig* und möglichst formal *spezifiziert* sein. Bisher wurde als Schnittstelle im allgemeinen nur die funktionale Schnittstelle einer Komponente betrachtet, das sog. API (Application Programming Interface). In unserer Definition umfaßt die Spezifikation der Schnittstelle alle Informationen, die zur Auswahl und zum Einsatz einer Komponente notwendig sind. Dies sind neben funktionalen Schnittstellen und Datenschnittstellen auch alle benötigten charakteristischen Eigenschaften einer Komponente wie z.B. Zeitverhalten, Verbrauchsverhalten, usw.

Dokumentation

Eine Komponente muß *hinreichend gut dokumentiert* sein. Die Dokumentation beinhaltet alles, was ein potentieller Nutzer wissen muß, um zu entscheiden, ob die Komponente für ein bestimmtes System eingesetzt werden kann oder nicht. Dies umfaßt nicht nur die Schnittstellen der Komponente, sondern u.U. auch deren internen Aufbau, die Annahmen bzw. Anforderungen an andere Komponenten, die erforderliche Infrastruktur (Architektur) und die vorgesehene Anwendungsdomäne. Die Dokumentation benennt außerdem den Reifegrad der Komponente. Dieser kann z.B. durch die durchgeführten Tests und Ergebnisse belegt werden. Ist eine Komponente nicht hinreichend dokumentiert, kann sie von anderen Nutzern nicht eingesetzt werden.

Qualität

Eine Komponente ist *qualitativ hochwertig*. Dies betrifft eine Reihe von Aspekten wie Korrektheit, Effizienz, Robustheit, Stabilität, Dokumentation. Bestehen Zweifel an der Korrektheit einer Komponente, dann sinkt die Motivation potentieller Nutzer, diese einzusetzen. Die Korrektheit sollte durch dokumentierte Testfälle validiert worden sein. Robustheit und Stabilität bedeuten, daß beim Einsatz einer Komponente keine

unvorhersehbaren Probleme, wie z.B. Laufzeitfehler, auftreten dürfen. Idealerweise ist eine Komponente ein Produkt mit einer entsprechend hohen Qualität.

Einsetzbarkeit

Eine Komponente ist *unverändert verwendbar*. Das kennzeichnet eine spezielle Art der Wiederverwendung, die Mehrfachverwendung oder Reuse „as is“. Das heißt, die Komponente muß (und soll) für ihren Einsatz nicht modifiziert werden. Das schließt nicht aus, daß eine Komponente nicht angepaßt werden kann. Für den Einsatz einer Komponente muß und soll in der Komponente jedoch kein Programmcode geschrieben oder modifiziert werden. Komponenten sind Fertigprodukte, die nur über definierte und geprüfte Mechanismen (s.o) angepaßt werden können. Ziel ist Verwendung von Komponenten als Black-Box, wobei der interne Aufbau einer Komponente für den Nutzer keine Rolle spielt. Dies ist jedoch problematisch, wie Garlan et. all [3] gezeigt haben, da es in der Regel versteckte Annahmen der Komponenten über ihr Umfeld gibt. Es kann also erforderlich sein, daß auch Details über den internen Aufbau von Komponenten dokumentiert bzw. als Schnittstelle spezifiziert werden müssen. Andererseits sollten alle Details über den internen Aufbau, die voraussichtlich nicht für den Einsatz der Komponente benötigt werden, aber verborgen werden.

Das Ziel der komponentenbasierten Softwareentwicklung ist die Konstruktion von Systemen aus vorgefertigten Komponenten. Dazu müssen die Komponenten nicht notwendigerweise anpassbar sein. Es ist vorstellbar, daß in unterschiedlichen Systemen die gleichen unveränderten Komponenten vorkommen (z.B. HW-Treiber). Eine Komponente kann unterschiedliche Möglichkeiten der Anpassung an verschiedene Anforderungen besitzen. Die Anpassbarkeit umfaßt folgende Bereiche:

- Anpassung an neue Kontexte
- Anpassung an zusätzliche Anforderungen (Erweiterbarkeit)
- Anpassung an neue Plattformen (Portabilität)
- Anpassung an neue Systemarchitekturen

Die Anpassbarkeit einer Komponente kann auf verschiedene Art und Weise erreicht werden:

- Anpassung durch Parametrierung
- Anpassung durch Generierung
- Anpassung durch Transformation

Diese verschiedenen Techniken erlauben jeweils einen unterschiedlichen Grad an Flexibilität.

Die Unabhängigkeit umfaßt ebenso verschiedene Bereiche:

- Unabhängigkeit gegenüber Änderungen
- Unabhängigkeit von anderen Komponenten
- Unabhängigkeit von Architekturen
- Unabhängigkeit von Plattfomen

Die Ausprägungen der beiden Eigenschaften Anpassbarkeit und Unabhängigkeit bestimmen maßgeblich das Ausmaß der Mehrfachverwendbarkeit einer Komponente.

Weitere Punkte

Eine Komponente kann aus mehreren Teilen bestehen. Dies ist insbesondere für verteilte Systeme von Bedeutung. Hier ist es vorstellbar, daß eine abgeschlossene funktionale Einheit verteilt auf mehreren Knoten eines Netzwerkes läuft.

Abstrakte und konkrete Komponenten

Bei der Entwicklung konkreter Software-Komponenten werden in der Regel wichtige Entwurfs- und Implementierungsentscheidungen getroffen, welche die Flexibilität (Anpassungsfähigkeit) der Komponente stark einschränken.

Modellkomponenten

Modellkomponenten sind abstrakte Komponenten, die in Form eines Modells vorliegen. Dieses Modell kann auf einfache Weise parametriert und angepasst werden. Durch die Verwendung von Komponenten auf der Modellierungsebene ergeben sich eine Reihe von Vorteilen.

Die Anpassung einer Modellkomponente wird auf der Modellierungsebene durchgeführt. Danach erfolgt die Generierung des Programmcodes automatisch durch einen Programmgenerator. Modellkomponenten werden heute bereits in vielen mächtigen Entwurfswerkzeugen wie z.B. Matrix X eingesetzt. Die Modellkomponenten stehen dort als Bibliotheken zur Verfügung. Die Programmgeneratoren sind in der Regel in die Werkzeuge integriert bzw. werden mitgeliefert.

3 Arbeitspaket „*Entwurfsmethoden*“

Das Ziel ist die Entwicklung neuer Methoden für den Entwurf eingebetteter Systeme. Dies bezieht sich auf alle Ebenen, beginnend mit der modellbasierten Konstruktion eines eingebetteten Systems aus Hard- und Softwarekomponenten bis hin zur Anpassung der Software an verschiedene Zielplattformen. In diesem Zusammenhang hat die Verbesserung des Entwurfs mit auf Anhieb funktionierender Implementierung eine grundlegende Priorität.

Eingebettete Systeme besitzen in der Regel definierte Funktionalitäten und interagieren mit ihrer Umgebung. Sie müssen zahlreiche Operationen durchführen und sind deshalb gezwungen, sehr schnell auf die Interrupts der Umgebung zu reagieren. Die große Anzahl der unterschiedlichen heterogenen Architekturkomponenten, wie digitale Bausteine, analoge Schaltungen und Software erschweren den Entwurf solcher Systeme. Um den Entwurf zu verbessern, sind neue Methoden notwendig. Diese Methoden helfen, aus verschiedenen Anforderungen an ein eingebettetes System die Struktur eines Gesamtsystems zu entwerfen und dadurch die Entwurfsphase zu verbessern.

Ein erster Schritt beim Entwurf eingebetteter Systeme ist die Erstellung eines Systemmodells. Der Entwickler muß anhand der vorgegebenen und gewünschten Anforderungen (z.B. in einem Lastenheft) das Modell erstellen. Hier werden Funktionalität und Randbedingungen festgelegt. Das Erstellen des Modells ist eine sehr wichtige Phase beim Entwurf solcher Systeme, da während dieser Phase viele funktionale Fehler nicht entdeckt werden. Sie werden erst bei der Implementierung sichtbar. Die Behebung solcher Fehler in einer späteren Phase ist sehr aufwendig. Aus diesem Grund soll versucht werden, Methoden zu entwickeln, die eine Simulation und Verifikation des Systems bereits in frühen Phasen erlauben.

Hier werden Funktionalität und Randbedingungen festgelegt. Dabei spielen bereits bestehende Produkte und neue Anforderungen eine große Rolle. Diese werden dann analysiert und münden in ein abstraktes Modell der Domäne, das Strukturen und Funktionen beschreibt. Auf der Basis dieses Modells und unter Berücksichtigung der Randbedingungen werden Komponenten identifiziert und eine Architektur für die Domäne entwickelt. Diese Architektur beschreibt die Struktur der Domäne durch Komponenten und ihre Beziehungen bzw. Schnittstellen untereinander. Die so entwickelte Architektur ermöglicht die Implementierung von Komponenten, die innerhalb der Domäne produktübergreifend eingesetzt werden können. Mit diesen Ergebnissen (Modell, Architektur, Komponenten) ist es dann möglich, einzelne Systeme oder Produkte des Anwendungsbereichs schneller und mit höherer Qualität zu entwickeln. Hierzu fehlen bis jetzt Erfahrungen. Diese sollen jedoch zukünftig gesammelt werden. Anhand der gewonnenen Erkenntnisse soll eine allgemeine Methodik für die Entwurfsmethodik im Bereich eingebetteter Systeme vorgeschlagen werden.

Die Wahl der Spezifikationssprache ist entscheidend für die Erstellung des Systemmodells, da eine geeignete Sprache der Nachweis der Funktionalität des Systems erleichtert. Für die Durchführung der Spezifikation gibt es verschiedene Sprachen. Die Auswahl der Sprache ist maßgebend für die Reduktion der Fehler. Hierbei ist darauf zu achten, daß möglichst viele Übereinstimmungen zwischen den Charakteristiken des Systems und den Sprachkonstrukten bestehen. Abweichungen erhöhen die Fehlerquote.

3.1 Spezifikationssprachen

Im folgenden wird kurz auf den Bereich der formalen Spezifikation eingegangen. Dann wird ein Überblick über verschiedene Spezifikationssprachen gegeben. Anschließend werden die möglichen Eigenschaften und Charakteristika von Spezifikationssprachen vorgestellt.

3.1.1 Formale Spezifikation

Eine Spezifikation ist eine präzise Darstellung und Beschreibung der Eigenschaften, der Verhaltensweisen, des Zusammenwirkens und des Aufbaus von Modellen, Programmen und Systemen. Im Idealfall stellt die Spezifikation eine vollständig formale, von Implementierungen unabhängige Beschreibung des Systems dar.

Das Ziel einer Spezifikation liegt nicht nur in einem klareren Verständnis des Problems, sondern vor allem auch in der Überprüfung der Korrektheit der Implementierung. Lassen sich die Aussagen der Spezifikation aus dem implementierten Verhalten folgern, so ist die Implementierung verifiziert. Außerdem soll eine Spezifikation zusätzliche Eigenschaften beschreiben wie Zuverlässigkeit, Robustheit und zeitliche Aspekte. Vor allem früher wurden Spezifikationen in natürlicher Sprache formuliert. Hierbei sind Mehrdeutigkeiten und Unklarheiten oft unvermeidbar. Auch bei „halbformalen“ Spezifikationen, die natürliche Sprache zusammen mit Datenflußplänen, Ablauf- und Strukturdiagrammen verwenden, kann es zu Mißverständnissen kommen. Demgegenüber haben formale Spezifikationen die Vorteile, daß sie eine eindeutige Semantik besitzen. Im folgenden werden die verschiedenen Möglichkeiten für eine formale Spezifikation beschrieben.

Allgemeine mathematische Methoden:

Dabei werden alle mathematischen Konstruktionen (Mengen, Abbildungen, Produkte usw.) als gegeben vorausgesetzt und ohne weiteres benutzt.

Axiomatische Spezifikation:

Diese Spezifikationsmöglichkeit ist sehr abstrakt. Die algebraische Spezifikation, die ähnlich wie abstrakte Datentypen aufgebaut ist, aber zusätzliche Typen für Aktionen, Zustände, Komponenten u.ä. bereit hält, ist ein Spezialfall der axiomatischen Spezifikation.

Petri-Netze:

Sie dienen als Modell zur Beschreibung und Analyse von Abläufen mit nebenläufigen Prozessen und nichtdeterministischen Vorgängen.

Spezifikationssprachen:

Diese Sprachen besitzen eine formale Semantik. Die Erstellung und Analyse von Spezifikationen erfolgt rechnergestützt.

Auf die Spezifikationssprachen soll im folgenden näher eingegangen werden.

3.1.2 Beispiele verschiedener Spezifikationssprachen

Spezifikationssprachen sind Mittel zur sprachlichen oder graphischen Darstellung von Programmen, Systemen und Verhaltensweisen. Im Gegensatz zu einer Programmiersprache werden in einer Spezifikationssprache nicht die genauen algorithmischen Abläufe und die konkreten Datenstrukturen, sondern die gewünschten Eigenschaften des Systems beschrieben. Spezifikationssprachen werden in der Entwurfsphase noch vor den Programmiersprachen benutzt. Existierende Spezifikationssprachen lassen sich folgendermaßen gliedern:

Formale Beschreibungssprachen (FDL): Zu dieser Kategorie gehören die Sprachen Estelle, SDL und LOTOS.

Sprachen zur Beschreibung von reaktiven Systemen : Esterel und StateCharts sind typisch für diese Art.

Hardware-Beschreibungssprachen (HDL): VHDL und Verilog werden als Standard zu diesem Zweck eingesetzt.

Prozeßorientierte Sprachen zur Beschreibung paralleler Algorithmen: CSP, OCCAM und CCS gehören zu dieser Kategorie.

Höhere Programmiersprachen: C und C++ können diesem Zweck dienen.

3*.1.3 Charakteristika von Spezifikationssprachen*

Abstraktionskonzepte

In höheren Spezifikationssprachen werden den Entwicklern höhere Abstraktionsmechanismen zur Verfügung gestellt. Diese sollen kurz vorgestellt werden. Bei Overloading (Überladen von Bezeichnern) kann derselbe Bezeichner für verschiedene Operationen verwendet werden, die durch den Kontext unterschieden werden. Als einfaches Beispiel kann die Operation „+“ angeführt werden, die sowohl für ganze wie auch komplexe Zahlen verwendet wird.

Generic Units sind Programmteile, die insgesamt parametrisiert und aktualisiert werden können. Eine weitere mögliche Eigenschaft von Programmiersprachen kann die Vererbung sein. Dabei werden Merkmale und Operationen einer Klasse in ihre Unterklassen übernommen. Dies ist ein charakteristisches Abstraktionskonzept objektorientierter Sprachen. Durch das Prinzip des „*Information hiding*“ werden „*private*“ Informationen vor anderen Modulen oder Prozessen verborgen. Dies beugt einem fahrlässigen Mißbrauch der Daten vor.

Erlaubt eine Programmiersprache das Konzept der Modularisierung, kann das Programm in mehrere Übersetzungseinheiten aufgetrennt werden. Diese können dann separat kompiliert werden.

Kommunikations- und Synchronisationsbedingungen

Mehrere nebenläufig ablaufende Prozesse sollten miteinander kommunizieren können. Dazu ist eine Prozeßsynchronisation nötig. Die Synchronisation und Kommunikation kann beispielsweise über gemeinsame Speicherbereiche (shared memory) ablaufen. Der sendende Prozeß schreibt in eine globale Variable (geteilter Speicher), die von allen Empfängern gelesen werden kann. Ein anderes Kommunikations- und Synchronisationskonzept ist der sogenannte Nachrichtenaustausch (message passing). Hierbei wird ein Prozeß solange blockiert, bis der andere Prozeß auch zur Kommunikation bereit ist. Der Vorteil dabei ist, daß kein zusätzlicher Speicher benötigt wird; der Nachteil liegt in einer eventuellen Leistungsverschlechterung. Bei dieser Synchronisationsart kann die Zieladresse explizit angegeben oder implizit enthalten sein.
Die Kommunikation kann in eine Richtung oder bidirektional und zwischen zwei oder mehreren Prozessen stattfinden. Eine andere Möglichkeit zur Synchronisation ist das Semaphorkonzept von E.W. Dijkstra. Ein Semaphor *S* ist eine ganzzahlige nichtnegative Variable verbunden mit einer Warteschlange W(s). Auf einen Semaphor kann nur mit einer P-Operation (Betreten) und mit einer V-Operation (Verlassen) zugegriffen werden. Die P-Operation wird zu Beginn eines kritischen Abschnitts, die V-Operation zum Ende eines kritischen Abschnitts ausgeführt. Der Anfangswert von s legt fest, wie viele Prozesse sich gleichzeitig in dem kritischen Abschnitt befinden dürfen.

Als letztes Konzept zur Synchronisation nebenläufiger Prozesse sollen Monitore vorgestellt werden. Ein Monitor verwaltet einen eigenen Speicherbereich und stellt ihn auf Anforderung einem Prozeß zur Verfügung. Zu jedem Zeitpunkt darf höchstens ein Prozeß den Monitor belegen (Prinzip des wechselseitigen Ausschlusses).

Test und Verifikation

Bei einer Verifikation wird die vollständige Korrektheit einer Spezifikation mit mathematischen Mitteln nachgewiesen. Beim Test dagegen sollen durch die Ausführung der Implementierung mit vorgegebenen Testdaten und Sollwerten eventuelle Fehler aufgedeckt werden. Ein Test ist also eine Suche nach Hinweisen auf Fehler.

Den Vorgang der Fehlerverfolgung und Fehlerbeseitigung nennt man Debugging. Dabei sind Simulationswerkzeuge und aussagekräftige Fehlermeldungen von Vorteil.

3.1.4 Einsatzmöglichkeiten

Die Spracheignung beschreibt für welche Anwendungsgebiete eine Sprache hauptsächlich eingesetzt wird. Je mehr Anwendungsgebiete (z.B. Prozeßkommunikation, Echtzeitsysteme, etc.) eine Sprache bietet, desto größer ist ihre allgemeine Spracheignung. Auf je mehr verschiedenen Plattformen (z.B. SUN, Dec, PC, etc.) eine Sprache lauffähig ist

(Plattformunabhängigkeit), desto größer sind natürlich auch ihre Einsatzmöglichkeiten. Ein weiteres Kriterium sind die zur Sprache gehörigen Tools. Sie können beispielsweise zur Simulation oder graphischen Darstellung von Programmen dienen. Eine weitere mögliche Eigenschaft von Spezifikationssprachen ist die Fähigkeit, Zeitverhalten zu spezifizieren. Zeitdarstellung ist wichtig, um die Realität zu reflektieren. Timeout-Intervalle beispielsweise beschränken die maximale Zeit in einem Zustand.

Die Sprachen ***Verilog*** und ***VHDL*** werden mehr oder weniger als Standard eingesetzt. Sie eignen sich für die Beschreibung der CSP Modelle aber auch zur Beschreibung von FSM Modellen. Eine andere Sprache, die zur Beschreibung von FSM Modellen verwendet wird, ist die Sprache ***Statecharts***. Für Modelle die aus Datenflußdiagrammen bestehen, wird ***SDL*** (Specification Description Language) als Spezifikationssprache eingesetzt. ***Esterel*** ist eine synchrone Sprache und eignet sich zur Beschreibung der Modelle, die relativ viele Ausnahmezustände und synchrone Ereignisse enthalten.

Am Institut für Automatisierungs- und Softwaretechnik der Universität Stuttgart werden die Sprachen Esterel, SDL und VHDL auf ihre Eigenschaften, Stärken und Schwächen untersucht.

3.2 Hardware Software Codesign

3.2.1 Motivation für Hardware Software Codesign

Eingebettete Systeme bestehen aus Hard- und Softwarekomponenten. Die Hardware bestimmt die Leistung des Systems, die Software in der Regel die Flexibilität und Eigenschaften. Der Entwurf eingebetteter Systeme kann verschiedene Aspekte, wie Timing, Größe, Zuverlässigkeit oder Kosten in den Vordergrund stellen.

Die gängigen Methoden benötigt ein separates Design für Software bzw. Hardware. Eine Spezifikation, die meistens nicht vollständig und verbal geschrieben, wird entwickelt und den Hardware- bzw. Softwareentwicklern zur Verfügung gestellt. Diese Methode hat folgende Nachteile:

- Mangel an einheitlicher Software-Hardwaredarstellung, der die Verifikation des Systems sehr schwierig macht.
- Es gibt keinen übersichtlichen Entwurfsprozeß. Dadurch wird die Änderung und Verbesserung der Spezifikation verhindert und damit die Entwicklungskosten bzw. -zeit in die Höhe getrieben.

HW/SW Co-Design bietet eine einheitliche Darstellung der Soft- bzw. Hardwarekomponente. Diese Methode erlaubt den Entwurf des Gesamtsystems vor der Implementierung. Abbildung 3.1 zeigt diesen Vorgang.

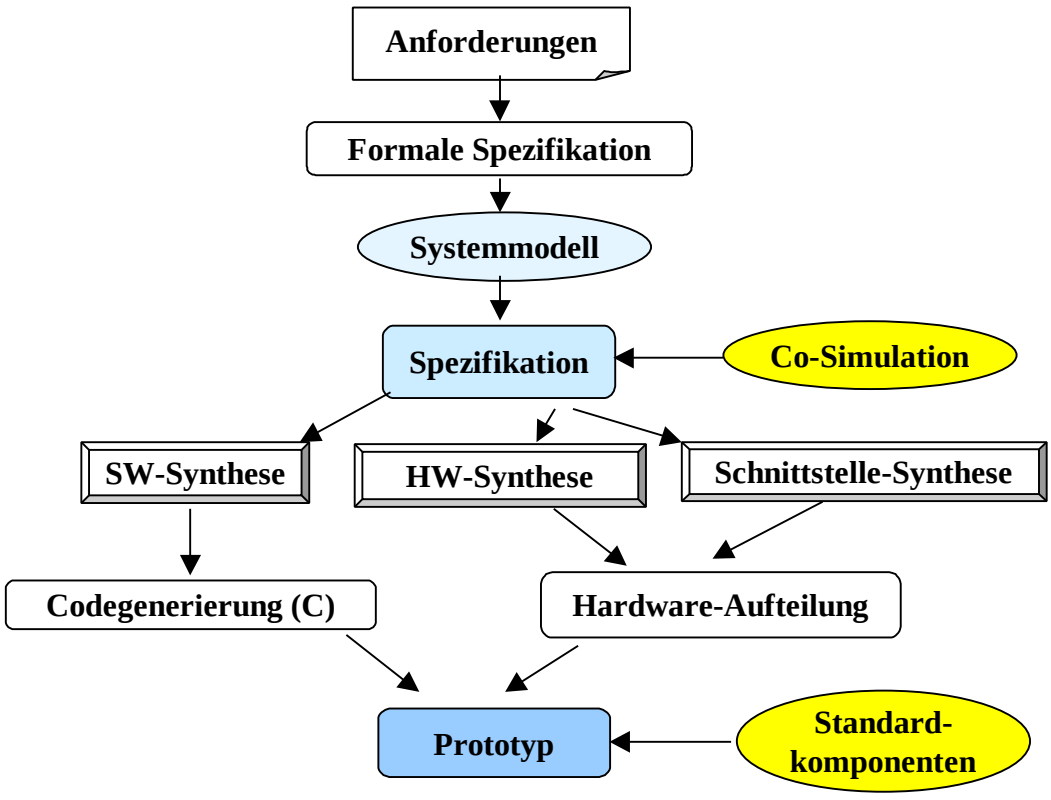

Abbildung 3.1: HW / SW Co-Design

3.2.2 Spezifikation auf abstrakter Ebene

Die Spezifikation wird in einer Spezifikationssprache, wie VHDL, Esterel oder FSM, durchgeführt.

3.2.3 Formale Verifikation

Die formale Spezifikation bietet die Grundlagen für die formale Verifikation. Diese Verifikation basiert auf FSM-Algorithmen. Bei der formalen Verifikation werden Hypothesen aufgestellt (z.B. über Systemzuständen, die niemals eintreten dürfen) und gegenüber der Spezifikation nachgewiesen.

3.2.4 System Co-Simulation

Die Spezifikation muß auf ihre Korrektheit und Vollständigkeit validiert werden können. Eine Spezifikation wird als vollständig bezeichnet, wenn sie alle möglichen Ereignisse, berücksichtigt. Die Korrektheit beruht darauf, daß das System auf solche Ereignisse richtig reagiert. Die Co-Simulation soll dem Entwickler helfen, die Entscheidungen zu verifizieren. Diese Entscheidungen betreffen die HW/SW Aufteilung, die Wahl der CPU bzw. des Schedulers. Die Simulation beinhaltet eine Ausführung der Spezifikation und einen Vergleich der generierten Ausgänge mit den zu erwartenden Werten.

3.2.5 Entwurf auf System-Ebene

Ein Problem bei der Entwicklung eingebetteter Systeme ist die richtige Auswahl der Hardware-Komponenten, auf denen die Systemfunktionalität implementiert werden muß. Auf dem Markt gibt es eine breite Auswahl von verschiedenen Mikrocontrollern, DSPs, Speichern, Bussen etc. Es muß die Entscheidung getroffen werden, ob der Schnelligkeit oder dem günstigen Erwerb der Vorrang eingeräumt wird. Ein ASIC Baustein ist beispielsweise schnell und teuer, ein programmierbarer Mikrocontroller dagegen billig und langsam. Da eingebettete Systeme in großer Stückzahl produziert werden, kann auch eine solche Entscheidung eine maßgebliche Rolle spielen.

Anschließend müssen alle Teile der Spezifikation bestimmten Komponenten zugeordnet werden. Um diesen Schritt optimal durchzuführen, müssen die einzelnen Teile der Spezifikation charakterisiert werden.

Es wird zwischen drei verschiedenen Sorten von Spezifikationsobjekten unterschieden. Der erste Teil ist das ***Speicherelement***. Ein Speicherelement, auch als Variable bezeichnet, muß gespeichert und bei Bedarf aufrufbar sein. Sie wird dem Speicher zugeordnet. Die ***Funktionen*** bilden einen anderen Teil von Spezifikationsobjekten. Sie bewirken eine Änderung der Dateninhalte. Eine If-Schleife ist ein Beispiel hierfür. Funktionen müssen den Standardprozessoren zugeordnet werden. Schließlich bleiben die ***Verbindungen***, die die Daten von einem Knoten zu einem anderen transportieren. Die Verbindungen müssen Bus-Elementen zugeordnet werden.

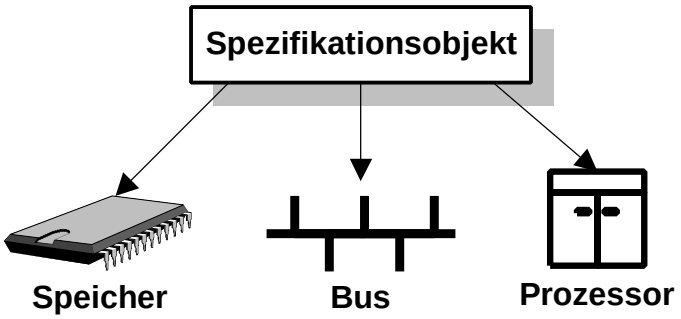

Abbildung 3.2: Zuordnung der Spezifikationsobjekte zu den Komponenten

Bei der Zuordnung der Spezifikationsobjekte zu den Komponenten, müssen stets die Einschränkungen des jeweiligen Systemes berücksichtigt werden. Die wichtigsten Kriterien sind die Größe der Programmbytes für Mikrocontroller bzw. Micoroprozessoren, die Laufzeit der Funktionen, die Anzahl der Gatter bei digitalen Bausteinen und die Bitrate bei den Ein- bzw. Ausgabegeräten.

3.2.6 Verbesserung der Spezifikation

Nachdem die Funktionalität des Systems spezifiziert und in einem weiteren Schritt die alternativen Systemdesigns untersucht wurden, muß die funktionale Spezifikation verfeinert werden. Die verfeinerte Spezifikation beinhaltet die Verbindung zwischen unterschiedlichen Systemkomponenten, wobei jede Komponente funktional spezifiziert wird. Demnach existiert eine Mischung aus funktionalen und strukturellen Teilen.

In einem ersten Schritt soll die Zuordnung der Variablen zum Speicher verfeinert werden. Bisher wurden sie als eine Größe von Daten einem Speicher zugeordnet. In dieser Form sind sie aber für verschiedene Prozesse nicht greifbar. Der Speicher muß beschrieben werden. Die Deklaration der Variablen wird dieser Beschreibung angepaßt. Das Protokoll für den Zugriff auf die Speicherelemente wird jedem Teil des Systems, der eine Variable aus dem Speicher bearbeiten möchte, hinzugefügt.

Ein anderer Bestandteil dieser Phase sind definierte Schnittstellen. Die Komponenten müssen in der Lage sein, optimal miteinander zu kommunizieren. Zur Veranschaulichung sei angenommen, daß die Spezifikation eine Funktion beinhalte, die die Werte einer Variablen lesen müsse, die aber in einer anderen Komponente deklariert ist. Der Wert dieser Variablen muß durch das Busssystem transportiert und der Funktion in der anderen Komponente zur Verfügung gestellt werden. Die Kommunikation zwischen einzelnen Komponenten wird in der Schnittstellenimplementierung der Spezifikation beschrieben.

Bei der Verfeinerung der Schnittstellen müssen verschiedene Aspekte betrachtet und verfeinert werden. Ein Problem ist die Busbreite. Bei der groben Spezifikation wird die Busbreite definiert und versucht diese zu optimieren, damit die Performance des Systems erhöht wird [KoSc94].

Ein weiteres Problem bei parallel ablaufenden Tasks ist die Regelung des Zugriffs auf die Ressource, wie Bus oder Speicher. Hier muß eine klare und definierte Entscheidung getroffen werden, wie im Konfliktfall welche Task als erste Ressource benutzen darf. Während der Verfeinerung der Spezifikation müssen Methoden und Algorithmen in die Spezifikation eingefügt werden, die diese Probleme beseitigen. Dies muß durch ein geeignetes Scheduling–Verfahren geregelt werden. Scheduling–Verfahren können als statisch oder dynamisch klassifiziert werden.

Bei statischen Verfahren werden alle Scheduling–Entscheidungen vorab getroffen, d.h. die Planung des zeitlichen Ablaufs der Tasks erfolgt vor ihrer Ausführung. Bei dynamischen Verfahren dagegen werden die Entscheidungen zur Laufzeit getroffen, indem jeweils eine Task aus der Menge der ablaufbereiten Tasks ausgewählt wird . Es gibt unterschiedliche Scheduling–Verfahren, die in der Praxis eingesetzt werden.

3.2.7 HW-SW Synthese

Nachdem das System durch die Spezifikation beschrieben und in einem weiteren Schritt optimiert wurde, muß nun der Code generiert werden (z.B. der C–Code). Hierfür ist ein spezieller Compiler notwendig, da die Spezifikation spezifische Beschreibungen enthält, die von gängigen Compilern nicht übersetzt werden können. In einem weiteren Schritt müssen diese Beschreibungen konvertiert werden.

Die gleichzeitig ablaufenden Tasks, die Wartezeit einer Task auf externe Ereignisse und ein einwandfreier Datenaustausch zwischen verschiedenen Tasks müssen bei der Compilierung besonderes berücksichtigt werden. Bei der Wahl der Kompilierungsoptionen ist stets auf ein optimales Verhältnis zwischen Schnelligkeit und Größe des Programms zu achten.

4 Arbeitspaket *„Werkzeugunterstützung“*

Um die Qualität der Software zu verbessern, werden CASE – Werkzeuge eingesetzt. Diese Werkzeuge unterstützen verschiedene Methoden, die diesem Zweck dienen. Um diese Werkzeuge richtig einzusetzen, ist eine exakte Untersuchung der Anforderungen bzw. Problemstellungen am Beginn der Entwicklung eingebetteter Systeme notwendig. Hierzu kommt der Entwurf des richtigen Designs. Diese Vorgehensweise vermindert die Entwicklungszeiten bzw. –kosten.

Die Produkte, die nach diesen Methoden entwickelt werden, entsprechen eher den an sie gestellten Anforderungen. Diese Erkenntnisse führen dazu, daß immer mehr Technologien für die frühen Phasen der Entwicklung der Software erarbeitet werden. Für die Entwicklung im Bereich der Hardware sind diese Methoden jedoch nicht einfach einzusetzen. Man denke da z.B. an die besonderen Merkmale der Echtzeit–Systeme, die harten Echtzeitanforderungen unterliegen. Dies bedeutet, daß der Code in der Laufzeit optimiert werden muß. Die geringe physikalische Größe solcher Systeme, stellt einen geringen Speicherplatz zur Verfügung, weshalb der Code möglichst kompakt sein muß. Der Einsatz spezieller Prozessoren verlangt besondere Compiler, die die Einschränkungen solcher Systeme berücksichtigen. Eingebetteten Systeme interagieren mit ihrer Umgebung durch Ereignisse, d.h., der Kontrollfluß steht im Vordergrund und nicht der Datenfluß.

Diese besonderen Merkmale eingebetteter Systeme führen dazu, daß sich die neuen Entwicklungsmethoden noch nicht durchgesetzt haben. Dazu kommt noch die Tatsache, daß bis jetzt eine neue Spezifikation fehlt, die sich auf Vollständigkeit und Konsistenz verifizieren läßt. Eingebettete Systeme benötigen eine Spezifikation, die validierbar ist.

Die Entwicklungsumgebung für eingebettete Systeme muß genaue Anforderungen erfüllen. Sie sollen nachfolgend erläutert werden:

Es ist wichtig, daß während der Entwicklung ein Modell unterstützt wird, das möglichst genau auf das Produkt zugeschnitten ist. Das unterstützende Modell soll unter anderem simulierbar sein. Diese Simulation soll sowohl während der Entwicklung als auch unter realen Zeitbedingungen ablaufen können. Das Modell muß die verwendeten Komponenten unterstützen und eine Zusammenarbeit mit ihnen zulassen. Dies bezieht sich sowohl auf die Software- als auch auf die Hardwarekomponenten. Die Entwicklungsumgebung muß die Möglichkeit zur Verfügung stellen, aus einem speziellen Modell einen Produktionscode zu generieren. Diese Charakteristika sind für die Vorgehensweise einer *dynamischen System–Modellierung* unabdingbar.

Es gibt auf dem *„Embedded Markt“* bereits Werkzeuge, die solche Vorgehensweisen unterstützen: erwähnt sei das Werkzeug STATEMATE zur Modellierung reaktiver Systeme, das Werkzeug Matrix X zur Modellierung der Regelstrecken und ASCET–SD zur Echtzeitsimulation eingesetzt wird.

Am IAS werden verschiedene Werkzeuge für die Entwicklung der Echtzeitsysteme eingesetzt.

Im folgenden Kapitel werden zwei solcher Werkzeuge kurz vorgestellt.

4.1 Modellierung und Simulation mit ASCET-SD

4.1.1 Übersicht über das Werkzeug ASCET-SD

ASCET-SD ist ein Werkzeug zur grafischen Spezifikation von Software. Durch den modularen Aufbau einer Softwarespezifikation in ASCET-SD können objektorientierte Strukturen abgebildet werden. Die Softwaremodule, die als Klassen aufgefaßt werden können, werden in ASCET-SD als *Complex Elements* bezeichnet und in einer Datenbank abgelegt. Sie können bei zukünftigen Softwareentwicklungen wiederverwendet werden.

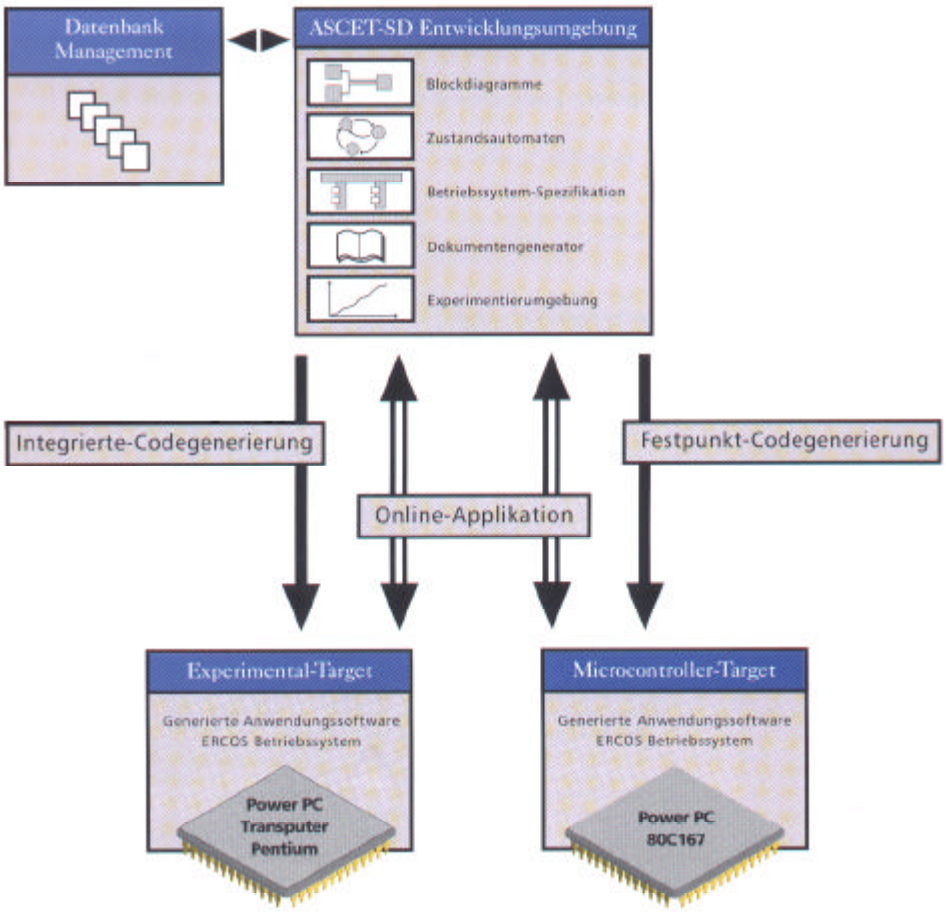

Abbildung 4.1: Die ASCET-SD-Entwicklungsumgebung (Grafik: ETAS)

Wie die Abbildung zeigt, leistet ASCET-SD die automatische Codegenerierung für verschiedene Zielsysteme.

Ein einzelnes ASCET-SD-Modell beschreibt die Software für einen Rechner. Die Spezifikation von Software für verteilte Systeme wird durch ASCET-SD nur insofern unterstützt, als mehrere Modelle erstellt werden können und die Anbindung an den CAN-Bus durch mitgelieferte Bibliotheksfunktionen realisiert wird

4.1.2 Simulation des Modells

Mit ASCET-SD kann das erstellte Modell der Software in Echtzeit simuliert werden. Die

integrierte Experimentierumgebung erlaubt das interaktive Testen am PC (sog. Offline-Betrieb) und die Simulation mit einer speziellen, an den PC angeschlossenen Transputer-Hardware (Online-Betrieb). Die Transputer-Hardware bietet eine Schnittstelle für den CAN-Bus. Die mit ASCET-SD für ein einzelnes Steuergerät spezifizierte Software kann so im echten verteilten System getestet werden, bevor der Code für das Steuergerät erstellt wird. Ebenso kann das ASCET-SD-Modell auch in einem System getestet werden, das mit CANoe simuliert wird.

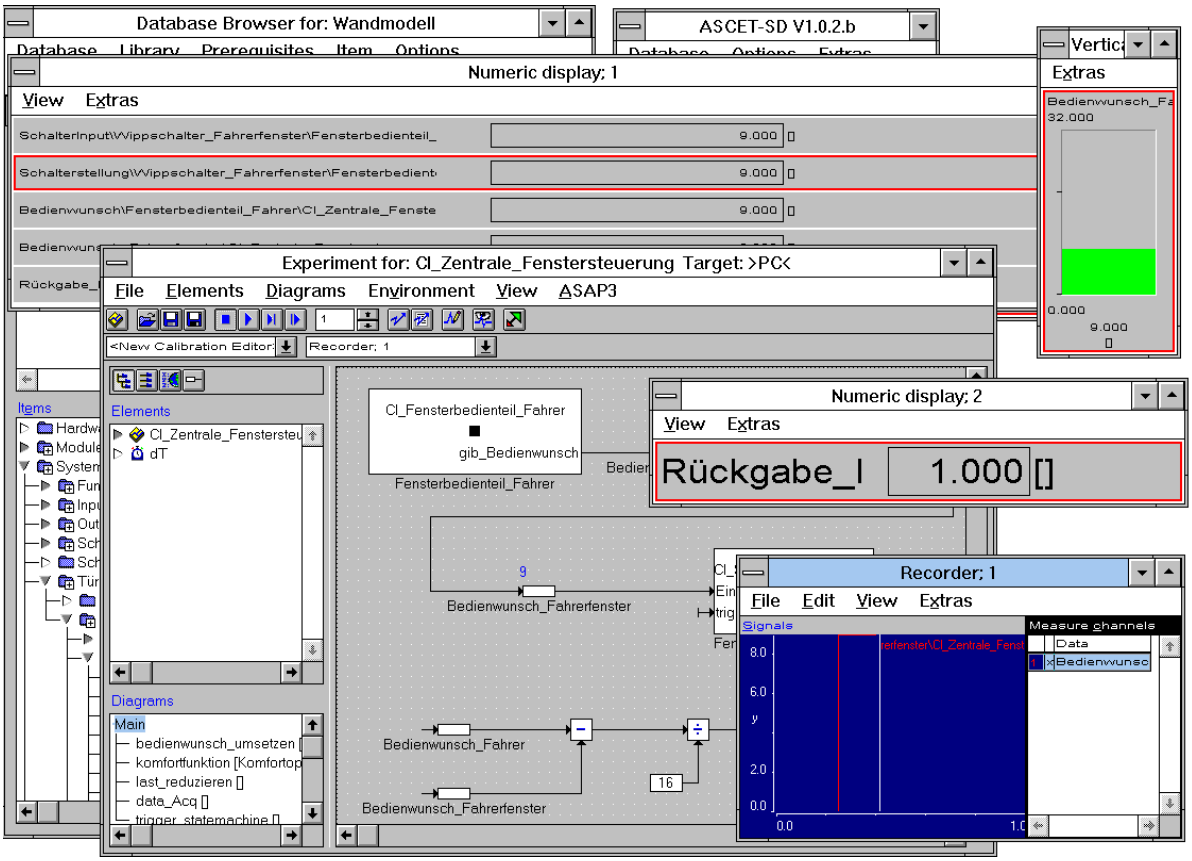

Abbildung 4.2: Experimentierumgebung in ASCET-SD

4.1.4 ASCET-SD

ASCET-SD ist eine integrierte Entwicklungsumgebung und unterstützt die Spezifikation von Software in Blockdiagrammen, die Simulation der Spezifikation in Echtzeit in einer Experimentierumgebung und zukünftig auch die Generierung eines lauffähigen Codes für Mikrocontroller. Dabei setzt der generierte Steuergerätecode auf dem Echtzeitbetriebssystem ERCOS auf.

Das Werkzeug ASCET bietet zwei Möglichkeiten zur Modellierung eingebetteter Systeme. Das Module- und Klassensystem, wobei bei der Verwendung von Modulen Schwierigkeiten vor allem beim „Handling“ auftreten. Aus diesem Aspekt gesehen, bietet ASCET die Ansätze für die komponentenbasierte Softwareentwicklung.

4.2 Statemate

4.2.1 Einführung

Statemate ist eine Software-Entwicklungsumgebung, die für den Entwurf komplexer, reaktiver Systeme entwickelt wurde. Das grundlegende Konzept in Statemate ist die Modellierung eines reaktiven Systems durch Activities, deren Verhalten durch verallgemeinerte endliche Automaten beschrieben wird. Weitere Sprachmittel neben diesen Kontroll- und Datenfluß-Diagrammen sind Data-Items, Events, Actions und Conditions.

Die Untergliederung des Systems in einzelne physikalische oder logische Komponenten wird durch verschiedene Modularisierungskonzepte unterstützt. Statemate besteht aus einer Sammlung von Konzepten und Beschreibungssprachen sowie den darauf basierenden Softwaretools. Diese Tools sind graphische Editoren, ein Prototyper mit Codegenerator nach C oder Ada, sowie eine graphische Animationsumgebung und ein Simulator, der entweder mit einem synchronen oder einem asynchronen Zeitmodell arbeiten kann. Die gemeinsame Basis der graphischen Beschreibungssprachen sind Higraphen. Higraphen sind rekursiv geschachtelte, gerichtete Graphen; d.h. ein Knoten eines Graphen kann wiederum einen Graphen enthalten. Die in den graphischen Beschreibungssprachen nicht oder nur schlecht darstellbaren Informationen werden in sog. Formularen (Forms) erfaßt.

4.2.2 Statemate: eine High-Level-Programmiersprache

Da Statemate eine Art graphische High-Level-Programmiersprache für „sehr stark erweiterte“ endliche Automaten ist, kann mit Statemate das Verhalten von komplexen (insbesondere eingebetteten und/oder Echtzeit-) Systemen leicht beschrieben werden.

Da Statemate über eine formale Syntax, sowie eine formale Semantik verfügt können die mit Statemate erstellten Entwürfe auf Aspekte wie Vollständigkeit und Korrektheit getestet werden, bevor die erste Zeile Code geschrieben wurde.

Durch die Einbindung graphischer Tools kann eine realistische Ein-/Ausgabe Schnittstelle des zu erstellenden Systems geschaffen werden, an der der spätere Benutzer ebenso wie der Entwickler prüfen kann, ob das bisher Entwickelte ihren Vorstellungen entspricht.

Aus diesen Gründen eignet sich Statemate auch besonders gut für Prototyping.

4 2 3 Vorgehensweisen zur Erstellung von Statemate-Beschreibungen

In Statemate selbst gibt es keine Prozess- oder Vorgehensmodelle zur Erstellung der Charts. Die Erarbeitung und Einhaltung solcher Modelle überläßt das Werkzeug dem Benutzer. Als Begründung hierfür wird eine nicht erwünschte Einschränkung der Flexibilität des Werkzeugs angeführt. Ein weiterer Nachteil von Statemate ist, daß die erzeugten Dokumente nicht den einzelnen Phasen der Software-Erstellung z.B. Anforderungsanalyse, Entwurf, usw.

zugeordnet werden können. Für eine ingenieurgemäße Entwicklung von Software sind jedoch Prozess- und Vorgehensmodelle unerläßlich. Ebenso sollte eine Versionsverwaltung der erzeugten Dokumente durchgeführt werden. Beides wird von Statemate nicht unterstützt.

Statecharts

Mit Statecharts wird das Verhalten des Systems beschrieben. Den Control Activities der Activitycharts wird jeweils eine Statechart zugeordnet, die das *wie* der Control Activity beschreibt.

Statecharts sind eine sehr weitgehende Verallgemeinerung der Darstellung endlicher Automaten als Zustandsübergangsdiagramme. Neben Zuständen und Transitionen stehen Events, Conditions und strukturierbare Variablen als weitere Sprachelemente zur Verfügung. Die Details des Entwurfes, die nicht direkt in den Charts beschrieben werden können, werden in Formulare (Forms) eingetragen und im Data-Dictionary abgelegt. Statecharts und Zustände können in Statemate rekursiv geschachtelt werden, d.h. ein Zustand bzw. der Automat den dieser Zustand enthält kann in einer neuen Statechart beschrieben werden.

Activitycharts

Activitycharts beschreiben die funktionale Sicht auf das System. Als Sprachmittel dienen Data Stores, Activities und Control Activities. Das Verhalten der Control Activities wird durch Statecharts beschrieben.

In Activitycharts kann keine Datenmanipulation beschrieben werden, dies ist nur in Statecharts möglich.

Modulecharts

Die Modulecharts beschreiben die physikalische Sicht des Systems, also aus welchen Software- bzw. Hardware-Komponenten das System besteht sowie deren Zusammenhang. Da Modulecharts nur deskriptive, aber keine operationale Semantik tragen, ist ihre Erstellung weniger fehleranfällig, dafür sind aber nicht validierbar.

Der Zusammenhang zwischen Statecharts und Activitycharts

Jede Activity einer Activitychart kann höchstens eine Control Activity enthalten. Das Verhalten der Control Activity wird durch die ihr zugeordnete Statechart beschrieben.

Der Zusammenhang zwischen Activitycharts und Modulecharts

Mit Activitycharts wird die Untergliederung des Systems aus funktionaler Sicht beschrieben, und Modulecharts beschreiben die Untergliederung des Systems aus struktureller Sicht.

4 2.4 Schwierigkeiten bei Statemate

Da mit Statemate sehr schnell ausführbare Komponenten erzeugt werden können, muß sehr sorgfältig kommentiert werden. Dies kann zum einen direkt in den Charts gemacht werden, aber auch im Data-Dictionary und besonders hier wird es gern vergessen.

Wenn in Statecharts an Transitionen Actions benutzt werden, deren Verhalten im Data-Dictionary definiert ist, so können diese nicht von der Generierung eines Events unterschieden werden. Die gleiche Syntax von Events und Actions kann also beim Lesen der Charts zu Mißverständnissen führen.

Die Semantik des Systems wird bei Statemate durch Charts, also graphisch beschrieben. Daraus ergibt sich bei größeren Systemen eine gewisse Unübersichtlichkeit. Um den Zusammenhang der einzelnen Charts untereinander herauszufinden (falls dieser nicht explizit dokumentiert ist), kann man sich eine Baumstruktur aus den Charts ableiten. Ausgehend von der Top-Level Activitychart als Wurzel, werden die einzelnen Activities als Knoten angehängt. Wird eine Activity durch eine Chart modelliert, so werden deren Activities wieder als Knoten an diese Activity angehängt. Nicht weiter ausmodellierte Activities oder ControlActivities werden zu Blättern. Für jede ControlActivity wird nun analog ein Baum für die angehängten Statecharts aufgebaut. Falls es schwer sein sollte, die Top-Level Activitychart zu bestimmen, kann es hilfreich sein, die in den Profiles für den Simulator als Scope angegebenen Charts zu betrachten. Falls es ein Profile für den Test des gesamten Systems gibt, so muß die dort als Scope definierte Activitychart die gesuchte Top-Level Activitychart sein.

4.2.5 Stärken und Schwächen von Statemate

Der Hauptvorteil von Statemate ist, daß das Verhalten des zu entwickelnden Systems während der Entwurfsphase getestet werden kann.

Der Hauptnachteil liegt im Fehlen von Sichtbarkeitsregeln. Ein weiterer Nachteil von Statemate ist, daß die erzeugten Dokumente nicht den einzelnen Phasen der Software-Erstellung z.B. Anforderungsanalyse, Entwurf, usw. zugeordnet werden können. Deshalb ist es oft schwer, im Nachhinein die Gründe oder Ziele für einzelne Modellierungsentscheidungen herauszufinden oder nachzuvollziehen. Mit Statemate können sehr schnell ausführbare Komponenten erzeugt werden, dies führt leicht zu mangelhafter Dokumentation und schlecht strukturierten Charts.

Da die Modularisierung bei Statemate nicht unterstützt wird, eignet sich Statemate nicht für den Einsatz komponentenbasierter Entwicklung.

5 Ausblick

Eingebettete System werden aufgrund ihrer hohen Effizienz zunehmend mehr in verschiedenen Bereichen, wie Konsum-, Unterhaltungs- und Kommunikationselektornik eingesetzt. Immer mehr Funktionen dieser Systeme werden durch Software realisiert. Dies führt dazu, daß einerseits die Flexibilität steigt, andererseits aber auch die Entwicklungs- und Produktionskosten in die Höhe schnellen.

Die Software wird bisher mit sog. „konservativen“ Methoden entwickelt. Der Markt verlangt jedoch neue Methoden, die die Entwicklungsprozesse einfacher und kürzer und dadurch kostengünstiger machen.

Komponentenbasierte Entwicklung ist die Lösung für dieses Problem. Mit dieser sog. „progressiven“ Methode ist es möglich, eingebettete Systeme schneller und kostengünstiger zu erstellen. Die gilt sowohl für die Entwicklung von Software als auch von Hardware. Komponentenbasierte Entwicklung ist ein wichtiger Bestandteil der aktuellen Forschung in Ländern wie der USA, England, Japan oder Deutschland. Die Entwicklung in diesem Bereich steckt allerdings noch in den Anfängen und weiterer Forschungsbedarf ist nötig.

Die Forschungsvorhaben der letzten Jahre haben eine Grundlage für die weiterführende Forschung in diesem Bereich geschaffen. Die Ergebnisse der vorliegende Arbeit sollen dazu genutzt werden, neue Methoden für die komponentenbasierte Entwicklung eingebetteter Systeme zu erstellen.

6 Literaturverzeichnis

[Bal96] Balzert, H.: *Lehrbuch der Softwaretechnik*. Heidelberg Berlin Oxford, Spektrum Akademischer Verlag, 1996

[Berry92] Gérard Berry, Georges Gonthier: *The Esterel Synchronous Programming Language*: *Design, Semantics, Implementation*. Science of Comupter Programming, 19(2): 87-152, November 1992

[Bre93] Break, R.: *Engineering real time system : an object-oriented methodology using SDL*. Prentice-Hall, New York 1993

[Bro96] Alan W. Brown: *Foundations for Component-Based Software-Engineering*. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA 1996

[BrWa96] A.W. Brown, K.C. Wallnau: *Engineering of Component-Based Systems*. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA 1996

[Cle95] Paul C. Clements: *From Subroutines to Subsystems: Component-Based Software Development*. American Programmer, Vol. 8, No. 11, Nov. 1995, Cutter Information. Corp

[Dem79] De Marco, T.: *Structured Analysis and Systemspecification*. New York, Yourdon Press 1979

[Dou97] Douglass, B.: *Real Time UML: Developing Efficient Objecets for Embedded Systems*. Addison Wesley 1997

[Eise97] U.W. Eisenecker: *Das Generative Paradigma oder „ Was kommt nach der Objektorientierung?*. in: OBJEKTspektrum 6/96, S30-34, 1997

[ElHS97] Ellsberger, J.; Hogrefe D.; Sarma A.: *SDL Formal Object-oriented Language for Communicating Systems*. Prentice Hall, Europe 1997

[Est96] *Esterel Manual*;
Ecole des Mines de Paris and INRIA, Valbonne, France 1988-1996

[GaAlOc95] D. Garlan, R. Allen, J. Ockerblom: *Architectural Mismatch (Why Reuse is so hard)*. IEEE Software V12, #6, pp17-26, November 1995

[Gab94] Gabel, D.: *Software Engineering*. IEEE Spectrum Jan. 1994, pp 38-41

[GaHe94] Gamma, E.; Helm, R.: *Design Patterns*. Addison Wesley 1994

[Goe98] Göhner, P.: *Komponentenbasierte Entwicklung von Automatisierungssystemen*. GMA-Kongress 98, VDI Verlag GmbH, Düsseldorf 1998, Seiten 513-521

[Halb93] Nicolas Halbwachs: *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers 1993

[HaPn85] Harel, D.; Pnueli: *On the Development of Reactive Systems: Logic and Models of Concurrent Systems*. Proc. NATO Advanced Study Institute on Logics and Models Verification and Specification of Concurrent Systems, NATO ASI Series F, vol.13, Springer-Verlag, pp. 477-498, 1985

[Harel87] Harel, D.: *Statecharts: A visual Approach to Complex Systems*. Science of Computer Programming, Vol. 8-3, pp. 231-275, 1987

[Heu97] H.Heuer-Hasenpatt et all: *Bausteinorientierte Anwendungsentwicklung: Voraussetzungen, Anforderungen und Auswirkungen*. OBJEKT Spektrum 3/97, S40-51, 1997

[Hoa85] Hoare, C.A.R.: *Communicating Sequential Processes*. Prentice Hall 1985

[Hog89] Hogrefe, D.: *Estelle, LOTOS und SDL Standard-Spezifikationssprachen für verteilte Systeme*. Springer-Verlag Berlin Heidelberg 1989

[HoUl90] Hopcroft, J.E.; Ullmann, J.D.: *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*. 2. Auflage, Addison-Wesley (Deutschland) GmbH 1990

[IEE88] IEEE Inc, N.Y.: *IEEE Standard VHDL Language Reference Manual*. 1988

[ITU88] ITU-T, ITU-T Recommendation Z.100: *Specification and Description Language (SDL)*. ITU-T, Geneva 1988

[KnBu91] Knöll H. D.; Busse J.: *Aufwandschätzung von Software-Projekten in der Praxis.* Mannheim, B.I. – Wissenschaftsverlag 1991

[Kop97] Kopetz, Hermann: *Real-Time Systems: Desgin Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Boston; Dordrecht; London 1997

[KöQu88] König, R.; Quäck L.: *Petri – Netze in der Steuerungs- und Digitaltechnik.* München Wien, R. Oldenbourg Verlag 1988

[KoSc94] Korf F.; Schlör R.: *Interface Controller Synthesis from Requirement Sepcification.* SIGDA Pbulication, ED&TC 1994

[Lap92] Laprie, J.C. (Ed.): *Dependability: Basic Concepts and Terminology – in English, French, German and Japanese*. Springer-Verlag, Wien 1992

[Lau89] Lauber R.: *Prozeßautomatisierung – Band 1*. Berlin Heidelberg New York, Springer–Verlag 1989

[McKo90] McFarland, M.;Kowalski, T.: *Incorporating botton-up design into hardware synthesis*. IEEE Transactions on Computer-Aided Design 1990

[Mur89] Murata, T.: *Petri-Nets: Properties, Analysis and Application*. IEEE Rroceedings, Vol. 77, No. 4, April 1989

[Nat92] NATO: *NATO Standard for the Development of Reusable Software Components*. Volume 1, NATO Communications and Information Systems Agency 1992

[Par72] D. Parnas: *On the criteria to be used for decomposing systems into modules.* CACM, Vol 15, No. 12 pp1053-1058, December 1972

[Pet79] Petri, C. A.: *Ansätze zur Organisationstheorie Rechnergestützter Informationssysteme*. Oldenburg 1979

[Ran94] Randell, B., Ringland, G., Wulf, W. (Ed.): *Software 2000: A View of the Future of Software*. ESPRIT. Brüssel 1994

[Ran96] Randall, Rick. *Space and Warning C2 Product Line Domain Engineering* Guidebook. Version 1.0 1996.

[Sim95] Simonovich, D.: *Merging Processes in Parallel Discrete-event Simulation*. European Simulation Symposium (ESS95) 19965

[Sim96] Simos, M., *et al. Software Technology for Adaptable Reliable Systems (STARS) Organization Domain Modeling.* Guidebook Version 2.0, Lockheed Martin Tactical Defense Systems 1996.

[Sut88] Sutcliffe, A.: *Jackson System Development*. New York, Prentice Hall 1988

[ThMo91] Thomas, D.; Morby P.: *The Verilog Hardware Description Language*. Kluwer Academic Publisher 1991

[WaTh89] Walter, R.; Thomas, D.: *Behavioral transformation for algorithmic Level IC design*. IEEE Transactions on Computer-Aided Design 1989