Wissenschaftliche Berichte
FZKA 6177

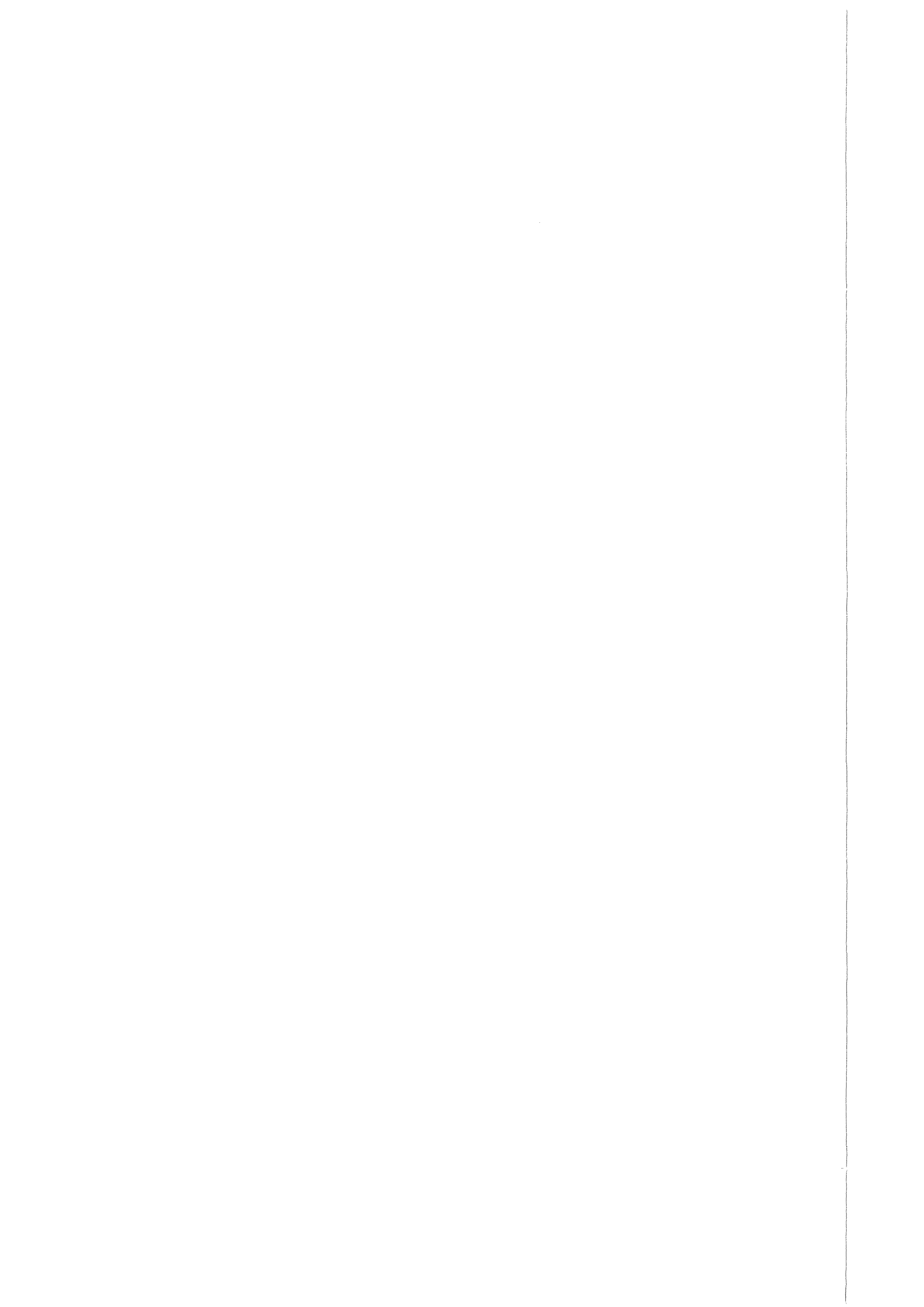
Entwicklung von Software-Systemen mit Ada

Ada-Deutschland Workshop Bremen 1998

H. B. Keller (Hrsg.)

Institut für Angewandte Informatik
Projekt Schadstoff- und Abfallarme Verfahren

November 1998



Forschungszentrum Karlsruhe

Technik und Umwelt

Wissenschaftliche Berichte

FZKA 6177

Entwicklung von Software-Systemen mit Ada

- Ada-Deutschland Workshop Bremen 1998 -

Hubert B. Keller (Hrsg.)

Institut für Angewandte Informatik
Projekt Schadstoff- und Abfallarme Verfahren

Forschungszentrum Karlsruhe GmbH, Karlsruhe

1998

Als Manuskript gedruckt
Für diesen Bericht behalten wir uns alle Rechte vor

Forschungszentrum Karlsruhe GmbH
Postfach 3640, 76021 Karlsruhe

Mitglied der Hermann von Helmholtz-Gemeinschaft
Deutscher Forschungszentren (HGF)

ISSN 0947-8620

Zusammenfassung

Am 22. und 23.4.1998 veranstaltete Ada Deutschland als GI-Fachgruppe 2.1.5 Ada in Bremen den Workshop „Entwicklung von Software-Systemen mit Ada“. Tagungsort war DASA-RI. Bei 12 Vorträgen konnten nahezu 80 Teilnehmer begrüßt werden. Dieser Bericht stellt die Beiträge zur Verfügung.

Development of Software Systems with the Programming Language Ada
Workshop of Ada-Deutschland, the Special Group 2.1.5 Ada
of the German Society for Computer Science

Abstract

On April, the 22nd and 23rd of 1997, Ada-Deutschland, the GI-working group 2.1.5 Ada, held the workshop "Entwicklung von Software-Systemen mit Ada" at the DASA-RI, Bremen. 12 talks were held and near eighty participants could be welcomed. This report is publishing the papers.

Vorwort

Am 22. und 23.4.1998 veranstaltete Ada Deutschland als GI-Fachgruppe 2.1.5 Ada in Bremen den Workshop „Entwicklung von Software-Systemen mit Ada“. Als Tagungsort konnte DASA-RI gewonnen werden. Bei 12 hochinteressanten Vorträgen konnten nahezu 80 Teilnehmer sowohl aus Deutschland als auch aus ganz Europa begrüßt werden.

Unter den Teilnehmer waren Vertreter von Hochschulen, Forschungseinrichtungen, der Industrie und auch Studenten. Die Ausstellungsmöglichkeiten wurden von mehreren Firmen intensiv genutzt.

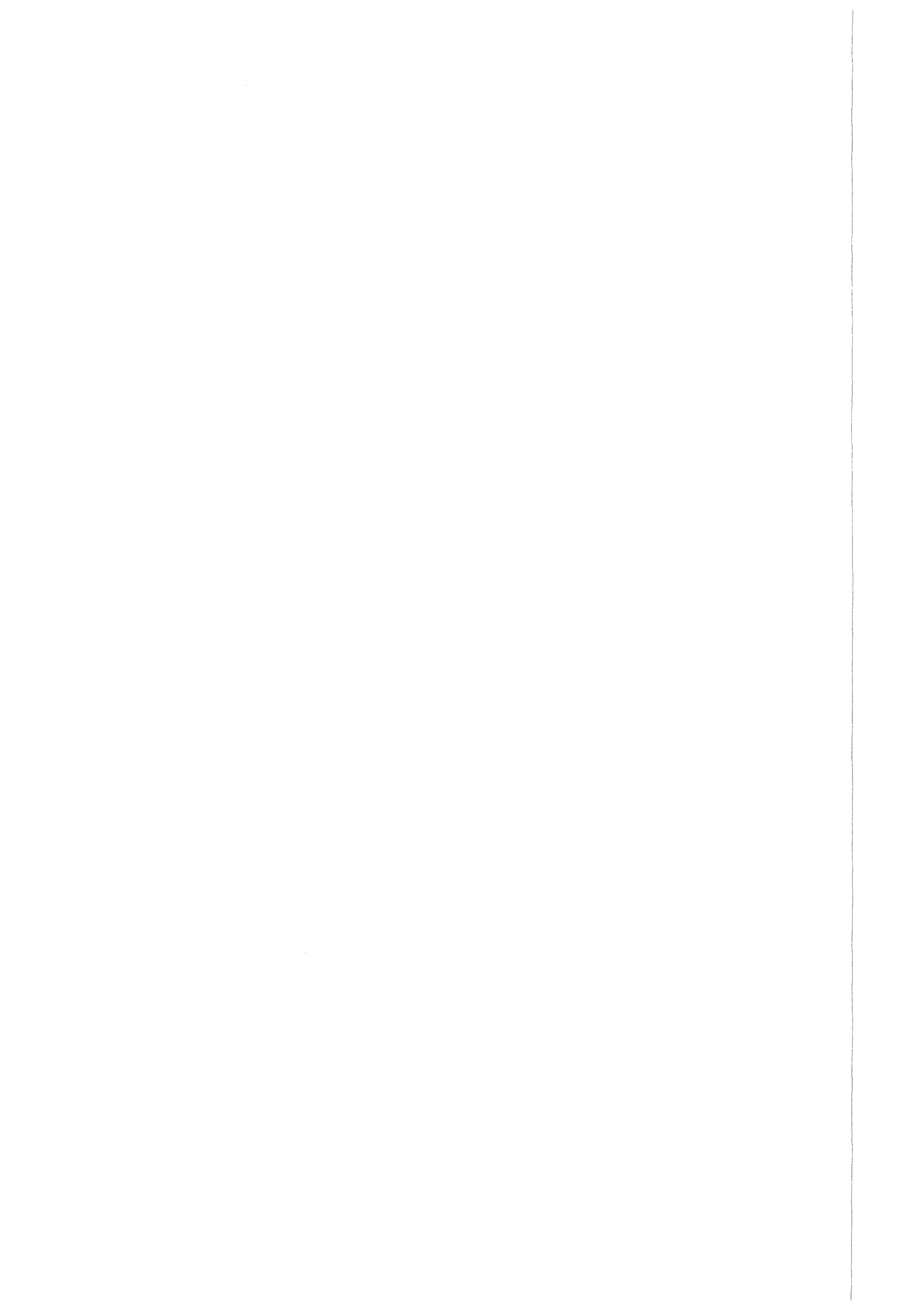
Dieser Bericht stellt die Beiträge als "Proceedings des Ada-Deutschland Workshops 1998, Bremen" zur Verfügung. Aufgrund der großen Resonanz wird dieser Workshop zukünftig regelmäßig als jährliche Tagung von Ada-Deutschland stattfinden.

Für die Bereitstellung der Tagungsräume und die hervorragende lokale Unterstützung in der Organisation sei an dieser Stelle Herrn Dr. Peter Lux und Frau Marianne Gärtner-Frank, DASA-RI Bremen, sehr herzlich gedankt.

Ein besonderer Leckerbissen war das gemeinsame Abendessen zum gegenseitigen Informationsaustausch. Durch die freundliche Unterstützung der Firma CCI, Meppen, konnte dies in beeindruckender Weise im Bremer Ratskeller mit einer Führung über ganz besondere "Speichertechnologien" durchgeführt werden. Besten Dank an Herrn Rudolf Landwehr, CCI Meppen, für den merklichen Support.

Die Gesamtorganisation und die technische Abwicklung konnten mit Hilfe des Forschungszentrums Karlsruhe äußerst effizient erfolgen. Den entsprechenden Stellen sei an dieser Stelle ebenfalls herzlich gedankt.

Hubert B. Keller
Ada-Deutschland – GI FG 2.5.1 Ada



Inhaltsverzeichnis

1 Software-Diversität und ihr Beitrag zur Sicherheit 13

1.1 Einleitung 13
1.2 Definitionen 14
1.3 Software-Diversität 15
1.4 Unterschiedliche Realisierungen der Software-Diversität 19
1.5 Fehlermodelle 20
1.6 Kostenmodell 22
1.7 Nutzen der Diversität 24
1.8 Schlußbemerkung 25
1.9 Literatur 26

2 Systematische Prüfung funktionaler und temporaler Korrektheit von Ada-Software 31

2.1 Einleitung 31
2.2 Systematischer Test funktionaler Aspekte 32
2.3 Vorgehen zum Test des Zeitverhaltens mit evolutionären Algorithmen 37
2.4 Teststrategie - Kombination von funktionaler und temporaler Prüfung 39
2.5 Zusammenfassung und Ausblick 39
2.6 Literatur 40

3 Einsatz von Ada 95 in einem Forschungsprojekt: Erfahrungen aus Software-Engineering-Sicht 43

3.1 Einführung 43
3.2 Ada im Entwicklungsprozeß 44
3.3 Die Rolle von Ada 95 im Entwurf 47
3.4 Ada 95 in der Implementierung 49
3.5 Test und Debugging 51

3.6 Fazit ... 52
3.7 Literatur ... 52

4 Sicherheit mit Standard-Software - Was kann Ada95 dazu beitragen ... 57
4.1 Einleitung ... 57
4.2 Die Stellung des Annex H und die Rolle der HRG ... 58
4.3 Der Inhalt des Annex H ... 59
4.4 Sprach-Einschränkungen für sicherheitskritische Systeme ... 62
4.5 Alternative Ansätze ... 63
4.6 Schlußfolgerungen ... 64
4.7 Literatur: ... 64

5 Ada as a First Language ... 69
5.1 The First Language Problem ... 69
5.2 Simplicity above all ... 71
5.3 The Compiler Environment ... 76
5.4 Summary ... 77
5.5 Bibliography ... 77

6 Ada-Metriken und ihre Anwendung ... 81
6.1 Der Zweck von Maßen ... 81
6.2 Sichtweisen für Ada95-Programme ... 82
6.3 Gerichtete Graphen als allgemeine Strukturmodelle ... 84
6.4 Charakterisierung von Ada-Projekten ... 86
6.5 Charakterisierung der Einbettung (Realzeitumgebung, Verteilung) eines Programms ... 87
6.6 Charakterisierung der globalen Programmstruktur ... 88
6.7 Charakterisierung der Prozeßstruktur ... 96
6.8 Zusamenfassung: Statische Programmanalyse und dynamische Modelle ... 97
6.9 Literatur ... 98

7 HotAda - Möglichkeiten zur Qualitätssicherung bei der verteilten Ada95-Programmierung ... 103
7.1 Einleitung ... 104
7.2 Qualitätserfordernisse ... 104
7.3 Allgemeine Betrachtung der Vorgehensweise ... 106

7.4 Konkrete Umsetzung durch die Werkzeuge 107
7.5 Zusammenfassung und Ausblick 109
7.6 Literatur 109

8 Ada, Concurrency and a Safety Critical Subset 113
8.1 Introduction to the Brewing process. 113
8.2 The Monitoring System 114
8.3 Comparison between Cyclic and event driven implementation 118
8.4 Ravenscar Profile 119

9 Software-Entwicklung mit Ada bei Dasa/RI 125
9.1 Das Projekt 125
9.2 Das Produkt 125
9.3 Einige Projektzahlen 126
9.4 Heutiger Stand der Entwicklung 127
9.5 Entwicklungsumgebung, Konfigurationsmanagement 127
9.6 Probleme bei der Softwareentwicklung 128
9.7 Erfahrungen 130

10 Einsatz von Ada im Experimentellen Führungsinformationssystem EIGER 133
10.1 Einleitung 133
10.2 Struktur des Gesamtsystems 134
10.3 Das Darstellungssystem 136
10.4 Der Aufbau des Kernsystems 137
10.5 Der Aufbau der Software des Kernsystems 138
10.6 Bewertung der Implementierung 142
10.7 Ansatz für die Implementierung von EIGER mit Ada 95 143
10.8 Prinzipien des ATCCIS-Replikationsmechanismus 145
10.9 Literaturhinweise 148

11 Ada in den Streitkräften - Bilanz und Tendenz- 151
11.1 Was hat die Bundeswehr veranlaßt, was hat sie getan? 152
11.2 Wie war und ist die Resonanz in der Bundeswehr und in ihrem Wehrtechnik- und Beschaffungsamt? 154
11.3 Ergebnis der Aktion 1995 157
11.4 Ergebnis der Aktion 1998 160

11.5 Verteilung der Programmiersprachen im BWB 161
11.6 Zeitliche Entwicklung der Anzahl Ada-Vorhaben 163
11.7 Erfahrungen aus dem Einsatz von Ada 163
11.8 Fazit und Tendenz 166

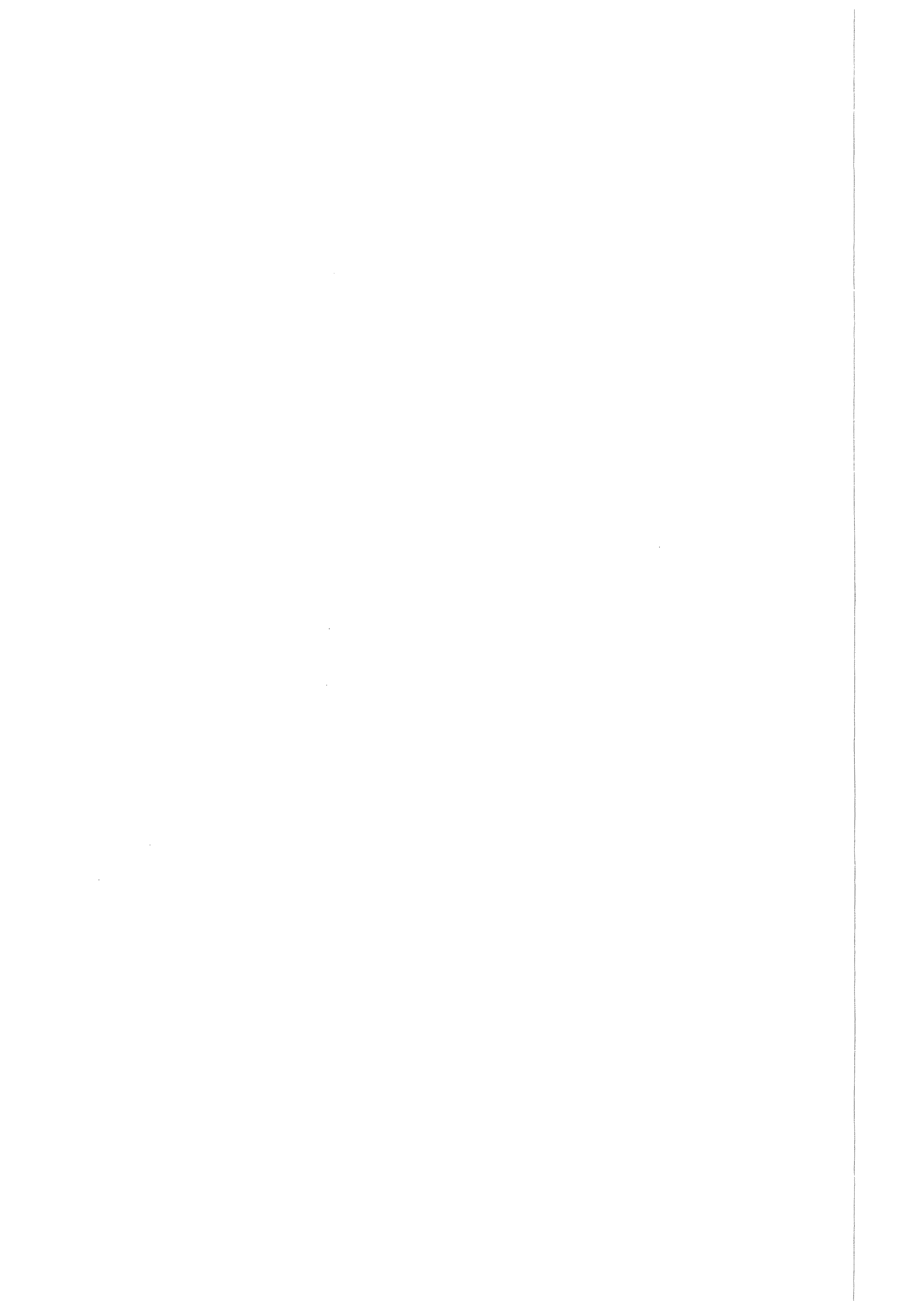
12 Ada im praktischen Einsatz 171
12.1 Kurzfassung des Referats 171
12.2 Folien 174

13 Am Ende des Übergangs von Ada 83 zu Ada 95: Brauchen wir noch Ada Compiler Validierungen? 187
1.1 Einleitung 187
1.2 Prinzipien der Ada Compiler Validierung 188
1.3 Beschreibung der Werkzeugumgebung CANDY 190
1.4 Zusammenfassung und Ausblick 195
13.1 Literatur 198

14 Ada, adé ? - 10 Jahre Erfahrungen mit Ada - ein persönliches Resumée 203
14.1 Übersicht 203
14.2 Zunächst einmal: 203
14.3 Kenngrößen unserer Ada-Projekte 204
14.4 Erfolgreich abgeschlossene Ada-Projekte 204
14.5 Erfolgsfaktoren dieser Projekte 205
14.6 Erfolgsfaktoren (Forts.) 205
14.7 Wertschöpfung aus diesen Projekten 206
14.8 Wertschöpfung (Forts.) 206
14.9 Derzeit laufende Ada-Projekte 207
14.10 Herausforderungen 207
14.11 Skepsis ist angebracht, denn 208
14.12 Prinzipiell ist zu hinterfragen 208
14.13 Hat Ada eine Zukunft? 208

15 Teilnehmer 211

Software-Diversität und ihr Beitrag zur Sicherheit



1 Software-Diversität und ihr Beitrag zur Sicherheit

Udo Voges
Forschungszentrum Karlsruhe GmbH
Institut für Angewandte Informatik
Postfach 3640, D-76021 Karlsruhe
voges@iai.fzk.de

Zusammenfassung

In sicherheitsrelevanten Anwendungsbereichen werden an die lebens- oder kapitalkritischen Systemen und damit auch an die zugehörigen Rechnersysteme hohe Zuverlässigkeits-Anforderungen gestellt. Zur Erreichung einer hohen Rechnerzuverlässigkeit, die eine entsprechende Software-Zuverlässigkeit beinhaltet, sind unterschiedliche Wege möglich. Neben dem Einsatz von Fehlervermeidungstechniken ist der Einsatz von Software-Diversität ein möglicher Ansatz. In diesem Beitrag wird die Software-Diversität als eine Form der Software-Fehlertoleranz in unterschiedlichen Ausprägungen beschrieben. Der damit verbundene Aufwand wird analysiert und der potentielle Nutzen dargelegt.

Keywords: dependability, diversity, redundancy, safety, software.

1.1 Einleitung

In immer stärkerem Maße werden Rechner in Systemen eingesetzt, die eine hohe Sicherheitsverantwortung tragen. Dies betrifft u. a. Anwendungen in der Medizin (z. B. Steuerung von Bestrahlungseinrichtungen, Chirurgieroboter), im Verkehr (z. B. ABS im Auto, Signaltechnik bei der Eisenbahn, Autopilot im Flugzeug) wie auch in der Industrie (z. B. Automatisierungseinrichtungen). Dabei handelt es sich um Systeme, deren Fehlverhalten zu einer Gefährdung von Menschenleben und/oder Sachen führen kann. Entsprechend der jeweils gültigen Normen muß z. B. aufgrund einer Risikoanalyse die für diese Anwendung geltende Risikoklasse (Safety Integrity Level) bestimmt werden, und die dementsprechenden Vorkehrungen und Sicherheitsmaßnahmen müssen bei der Systementwicklung getroffen werden.

Der Einsatz von Rechner-Hardware und -Software in Systemen mit Sicherheitsverantwortung macht es erforderlich, daß nicht nur auf der Systemseite für Sicherheit gesorgt werden muß, sondern neben der Rechner-Hardware auch die entsprechende Software hohen Zuverlässigkeitsanforderungen genügt. Dabei ist allerdings zu beachten, daß die Software alleine noch keine Sicherheitsgefährdung darstellt, sondern erst im Zusammenwirken innerhalb des Systems ein Softwarefehler zu einem Risiko bzw. zu einer Gefahr werden kann. Die sicherheitstechnische Betrachtung wird also immer den Kontext der Software zum Gesamtsystem

berücksichtigen. Dennoch ist von seiten der Software zumindest für eine fehlerfreie, korrekte Funktion oder allgemeiner für eine hohe Software-Qualität zu sorgen, da dies die Grundvoraussetzung für ein zuverlässiges System ist.

Idealerweise wird dieses Ziel durch entsprechende konstruktive Fehlervermeidungsverfahren gewährleistet - d.h. während der Erstellung der Software werden Methoden und Werkzeuge eingesetzt, die möglichst zur „Zero-defect“-Software führen -, und durch analytische Verfahren wird anschließend innerhalb der Validation und Verifikation (V&V) nachgewiesen, daß dieses Ziel erreicht wurde. Die Fehlervermeidung ist ein optimistischer Ansatz: man hofft auf ein fehlerfreies System. Die derzeitig verfügbaren bzw. benutzten Fehlervermeidungsverfahren, angefangen von der formalen Spezifikation bis hin zu Programmierrichtlinien, erfüllen diese Anforderungen allerdings oft nicht vollständig bzw. erfordern wiederum einen hohen Aufwand, der in keinem Verhältnis zum erwarteten (und nicht immer garantierten) Nutzen steht. In der Regel muß auch trotz der eingesetzten Fehlervermeidungsverfahren mit Restfehlern in der Software gerechnet werden.

Daher muß neben der Fehlervermeidung als eine weitere Maßnahme die Fehlertoleranz als pessimistischer Ansatz eingesetzt werden, um die gewünschte Zuverlässigkeit der Software bzw. des Systems zu erreichen. Im folgenden wird die Software-Diversität als eine derartige Fehlertoleranzmaßnahme erläutert. Einige unterschiedliche Realisierungsformen der Software-Diversität werden vorgestellt und der damit verbundene Aufwand beschrieben. Auf den damit erzielbaren Nutzen wird eingegangen. Eine Kosten-/Nutzen-Analyse muß für das jeweilige Projekt mit für die Entscheidung herangezogen werden, welche Fehlervermeidungs- und Fehlertoleranz-Maßnahmen einzusetzen sind. Dies wird ggf. auch in Absprache mit der betreffenden Genehmigungsbehörde bzw. den Gutachtern unter Heranziehen der entsprechenden Normen (z. B. /DIN94/) zu klären sein.

1.2 Definitionen

Damit ein Fehler, der während des Betriebs eines sicherheitsrelevanten Systems auftreten kann, nicht zu einem kritischen Ausfall des Systems führt, muß ein solcher Fehler erkannt und, da eine automatische Fehlerbeseitigung in den wenigsten Fällen möglich ist, ggf. toleriert werden. Eine mögliche Fehlertoleranz-Maßnahme ist die Redundanz. **Redundanz** ist die Existenz von Teilen, die nicht unbedingt für die eigentliche Funktion eines System notwendig sind. Dabei ist zu unterscheiden zwischen **homogener Redundanz** (Redundanz mit gleichartigen Mitteln) und **diversitärer Redundanz** (Redundanz mit ungleichartigen Mitteln). Homogene Redundanz wird z. B. verwendet, wenn mit einem alterungsbedingten Ausfall zu rechnen ist (zwei Glühbirnen im roten Teil einer Ampel). Die Wahrscheinlichkeit, daß die redundanten Teile gleichzeitig ausfallen, ist im allgemeinen wesentlich geringer als der Ausfall eines Teils, wenn wir von einer statistischen Unabhängigkeit dieser Ereignisse ausgehen können. Diese Form der Redundanz wird auch in der Regel in der Rechnerhardware eingesetzt. Dabei wird davon ausgegangen, daß z. B. durch fertigungsbedingte Unterschiede die redundanten, d.h. hier duplizierten Teile unterschiedliches Ausfallverhalten an den Tag legen und dadurch eine Fehler- (bzw. Ausfall-) Erkennung möglich ist und nicht zu einem totalen Versagen des Systems führt. Das gleichzeitige Ausfallen beider redundanten Teile wird als unwahrscheinlich (bzw. mit ausreichend geringer

Wahrscheinlichkeit des Eintretens behaftet) angesehen, und eine Reparatur des fehlerhaften Teils vor dem Ausfall des zweiten wird angestrebt.

Da die Herstellung von Software-Duplikaten ein einfaches Kopieren ist, die unterschiedlichen Kopien aufgrund ihrer Identität die gleichen Fehler beinhalten und damit ein einheitliches Verhalten haben, also immer gleichzeitig entweder fehlerhaft oder fehlerfrei reagieren, ist durch den Einsatz von homogener Redundanz bei Software kein Zugewinn an Zuverlässigkeit zu erreichen. Unterschiedliches Fehler- bzw. Ausfallverhalten kann nur durch unterschiedliche Software erlangt werden, d.h. diversitäre Redundanz bzw. Diversität. Während wir bei homogener Redundanz von Kopien sprechen, nennen wir die diversitären Software-Komponenten Varianten (In der Literatur wird zwar meist der Begriff „Versionen“ verwendet, aber m.E. führt dies insbesondere im Softwarebereich zu Verwechslungen mit den im Laufe der Zeit fortentwickelten Versionen eines Softwaresystems. Im folgenden wird hier daher der Begriff **Varianten** verwendet.).

1.3 Software-Diversität

Als Ausgangspunkt für die Systemerstellung haben wir in der Regel eine identische Anforderungs-Spezifikation für die Software. Die diversitär auszulegenden Teile sollen also die identische Funktion besitzen, zur Erreichung einer Diversität aber mit unterschiedlichen Mitteln realisiert werden. Die Software-Diversität kann auf unterschiedlichste Weise erreicht werden, z. B.:

- unterschiedliche Teams können die verschiedenen Software-Varianten erstellen,
- unterschiedliche Verfahren, Methoden und Werkzeuge können bei der Software-Erstellung eingesetzt werden,
- unterschiedliche Randbedingungen werden genutzt (z.B. Daten-Diversität, Zeit-Diversität),
- unterschiedliche Lösungsverfahren bzw. Algorithmen können programmiert werden.

Jede der hier genannten Realisierungsarten kann i. a. mit jeder anderen verkoppelt werden. Weiterhin kann sich die Anwendung der Diversität entweder nur auf einige Softwareentwicklungsphasen beschränken oder den gesamten Lebenszyklus betreffen. Die Diversität kann geplant bzw. forciert werden durch eine gezielte Vorgabe unterschiedlicher Methoden, oder sie kann mehr zufällig sein, indem nur unterschiedliche Teams ohne weitere Vorgaben an die Problemlösung gesetzt werden. Auf den Hardware-Einsatz hat die Software-Diversität meist keinen unmittelbaren Einfluß, d.h. die Software kann auf homogen-redundanter oder auf diversitärer Hardware, ggf. sogar auf der identischen Hardware als parallele oder sequentielle Tasks laufen.

Im Bereich der Software-Diversität haben sich im wesentlichen zwei unterschiedliche Ausprägungen durchgesetzt: die **Rücksetz-Blöcke** (RB, Recovery Blocks) /Randell75/ und die **N-Varianten-Programmierung** (NVP, N Version Programming) /Avizienis77/. Ein wesentlicher Unterschied dieser beiden Ansätze ist die Art der Ausführung: ein System mit Rücksetz-Blöcken (vgl. Abb. 1) beinhaltet einen Überprüfungsmodul bzw.

Test, der das Programmergebnis auf Korrektheit bzw. Plausibilität kontrolliert. Wird keine Unregelmäßigkeit entdeckt, dann wird nur die erste Variante ausgeführt, wird hingegen das Ergebnis nicht akzeptiert, so wird eine weitere Variante - bei Rücksetz-Blöcken auch Alternative genannt -, ggf. mit Performance-Verlusten, aktiviert und deren Ergebnis ebenfalls überprüft. Wird das Ergebnis von keiner der Varianten akzeptiert, so wird eine Fehlerbedingung aktiviert. Für die Ausführung der Alternativen bei Rücksetz-Blöcken müssen die Startbedingungen für die Alternativen identisch sein, d. h. die entsprechenden Datenbereiche (Eingabewerte) müssen wiederhergestellt werden, die Rücksetz-Punkte müssen entsprechend definiert werden.

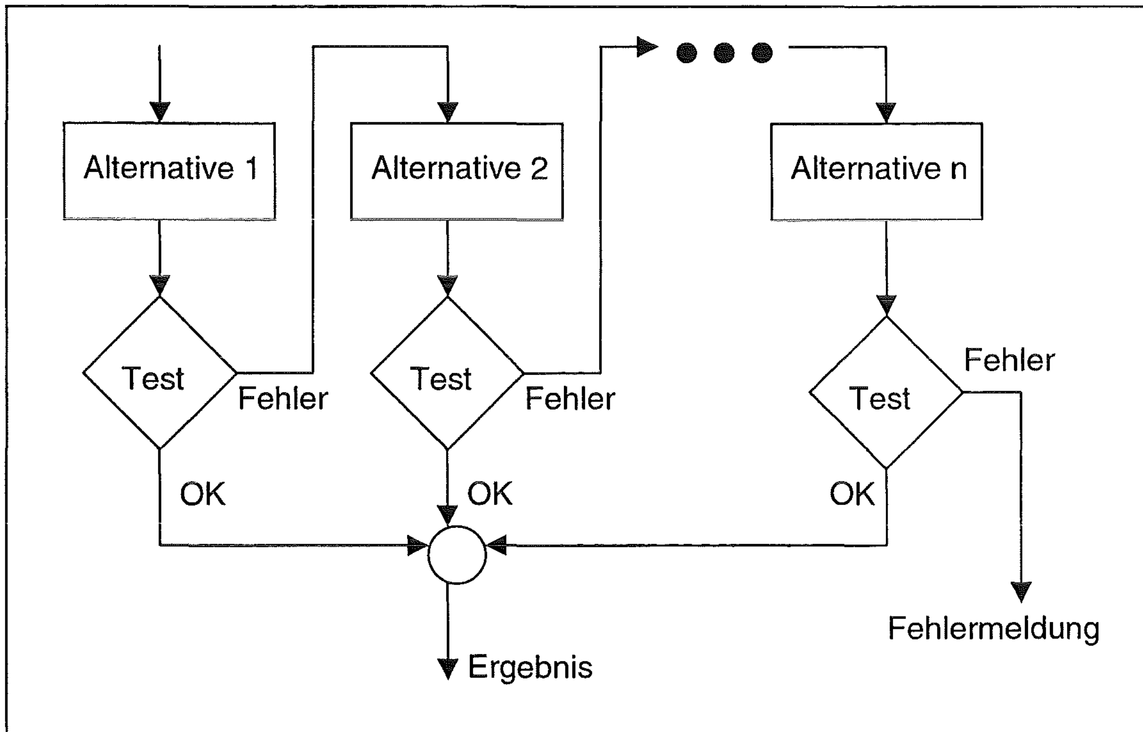


Abb. 1: Struktur der Recovery-Blocks

In einem NVP-System (vgl. Abb. 2) werden im allgemeinen alle Varianten gleichzeitig aktiviert, die Ergebnisse von allen Varianten werden miteinander verglichen und in der Regel wird das gültige Resultat von einem Abstimmungsmechanismus (Voter) weitergeleitet. Je nach der Art des Voters wird z. B. das Ergebnis nur weitergeleitet, wenn alle Varianten übereinstimmen oder wenn sich zumindest eine Mehrheit auf ein Ergebnis einigen kann.

Eine Übersicht über die wesentlichen Unterschiede und Charakteristika von N-Varianten-Programmierung und Rücksetz-Blöcken ist in Tabelle 1 zusammengestellt. Neben den hier beschriebenen Grundformen der N-Varianten-Programmierung und Recovery Blocks gibt es eine Reihe von Mischformen zwischen NVP und RB (vgl. Parhami96, Sullivan95, Fuhrman95/).

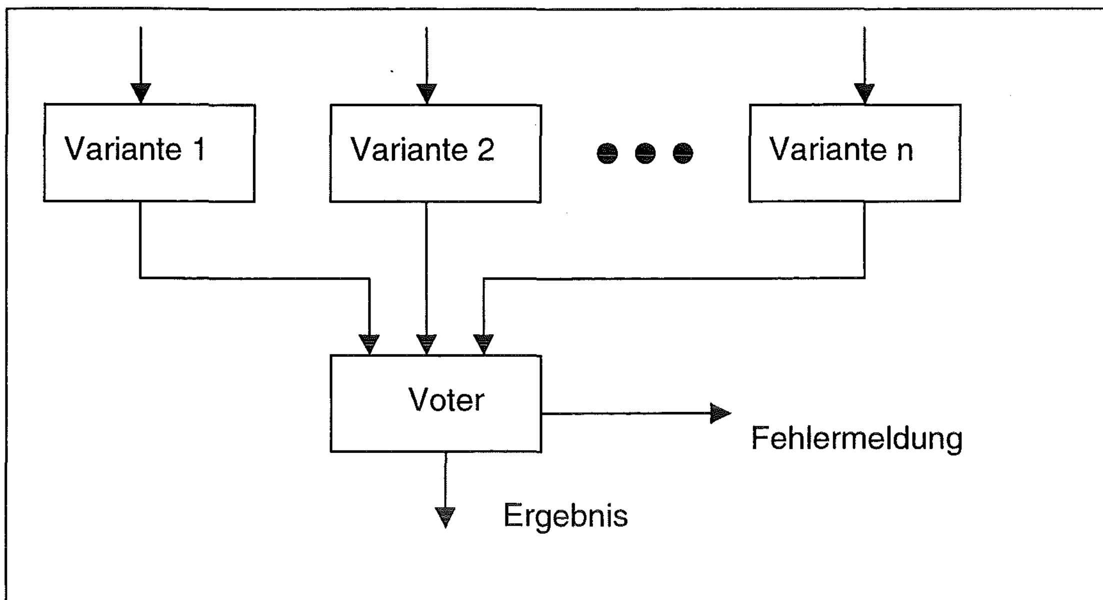


Abb. 2: Struktur der N-Varianten-Programmierung

Tab. 1: Vergleich zwischen N-Varianten-Programmierung und Rücksetz-Blöcken

N-Varianten-Programmierung	**Rücksetz-Blöcke**
unterschiedliche Varianten	unterschiedliche Alternativen
Fehlererkennung durch Vergleich	Fehlererkennung durch Akzeptanz-Test
Fehlerkorrektur (-maskierung) durch Mehrheitsbewertung	Fehlerkorrektur durch Rücksetz-Punkt und Aktivierung einer anderen Alternative
Mehrheitsentscheidung (Relativ-Test)	Akzeptanz-Test (Absolut-Test)
Mehrheit erforderlich	Übereinstimmung von einer Alternativen mit Akzeptanz-Test erforderlich
Ausführung aller Varianten immer erforderlich	Ausführung mehrerer Alternativen nur im Fehlerfall erforderlich
statische Redundanz	dynamische Redundanz
im allgemeinen parallele Ausführung der Varianten	im allgemeinen serielle Ausführung der Alternativen

Der Abstimmung über die Ergebnisse kommt bei der Software-Diversität eine entscheidende Bedeutung zu. Hier gibt es für die Realisierung verschiedene Möglichkeiten, die sich unterschiedlich auswirken und deren Einsatz z. T. auch applikationsabhängig ist. Die Abstimmungspunkte stellen auch Synchronisationspunkte dar, der Voter kann erst zu einer positiven Entscheidung kommen, wenn ausreichend viele Varianten (z. B. die Mehrheit) ihre miteinander übereinstimmenden Ergebnisse abgeliefert haben. Je nach Anforderung kann es auch erforderlich sein, daß alle Varianten miteinander übereinstimmen. Auf den Voter innerhalb eines diversitären

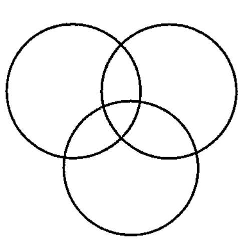
Systems kommt es entscheidend an, inwieweit sich die Diversität positiv oder negativ auswirkt. Ist z. B. das Ergebnis einer Berechnung nicht nur ein binärer Wert, sondern eine Real-Zahl, so ist mit leichten Abweichungen zwischen den Werten zu rechnen. Die Art der Übereinstimmung kann mit einer zugelassenen Bandbreite erfolgen, d.h. die Werte müssen nicht notwendigerweise vollständig übereinstimmen, sondern z. B. bei Dezimalzahlen nur bis zu einer bestimmten Stelle nach dem Komma. Ist der Toleranzbereich zu klein, so wird selten eine Übereinstimmung zwischen den Varianten erreicht und das System führt zum Abbruch. Ist andererseits der Toleranzbereich zu groß, so werden Fehler nicht erkannt und die ggf. durchgeführte Mittelwertberechnung über die von den Varianten gelieferten Werte liefert einen ungünstigen Wert. Das gleiche gilt für den Akzeptanztest innerhalb der Rücksetz-Blöcke.

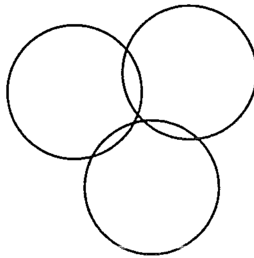
Je enger die einzelnen Abstimmungspunkte beieinander liegen, desto geringere Möglichkeiten für eine diversitäre Entwicklung bestehen, aber desto eher ist andererseits ein Fehler erkennbar.

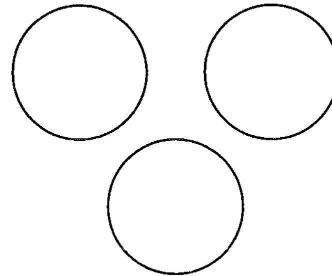
Nach dem Voter können die einzelnen Varianten entweder mit ihren eigenen Werten weiterrechnen, sofern diese als korrekt zugelassen wurden, oder es werden einheitliche Werte an alle Varianten zur weiteren Verarbeitung verteilt.

Bei der Wartung von diversitärer Software ist darauf zu achten, daß die Diversität nicht zerstört wird. D.h. auch für die Wartung sind einige Regeln einzuhalten, damit der ursprüngliche Zuverlässigkeitsgewinn nicht verloren geht.

Der wesentliche Aspekt bei diversitären Systemen ist, daß man davon ausgeht, daß nicht alle Varianten des diversitären Systems gleichzeitig versagen oder fehlerhaft reagieren. Ideal wäre es, wenn die Fehlerbereiche der Varianten alle disjunkt wären, d.h. daß nie zwei Varianten gleichzeitig versagen, bei den gleichen Eingabewerten einen identisch fehlerhaften Wert abliefern (vgl. Abb. 3c). Leider trifft dieser Idealfall nicht zu, wir müssen auch bei diversitären Systemen mit sogenannten identischen Fehlern rechnen: die Fehlerbereiche der einzelnen Varianten überlappen sich, mehr als eine Variante liefert für einen Eingabefall ein fehlerhaftes Resultat. Der Grad an Diversität bestimmt entscheidend auch den Grad an Fehlerüberdeckung, mit dem gerechnet werden muß. Ein sich insgesamt fehlerfrei verhaltenes System ist also im allgemeinen nicht erreichbar, sondern nur ein nach außen im Vergleich zu einem einzelnen System fehlerärmeres System. Dabei muß versucht werden, die Fehlerbereiche soweit zu separieren, daß nicht alle Varianten sich in einem Teil überlappen (vgl. Abb. 3a), sondern höchstens eine Untermenge der Varianten (vgl. Abb. 3b). In diesem Fall ist zumindest eine Fehlererkennung möglich. Erst wenn davon ausgegangen werden kann, daß nur eine Minderheit von einem Fehler betroffen ist, kann über die Fehlererkennung hinaus auch eine Fehlerkorrektur vorgenommen werden. Ansonsten wird man in sicherheitsrelevanten Bereichen die Redundanz lieber nur zur Fehlererkennung hernehmen und auf den möglichen Zugewinn an Verfügbarkeit zugunsten der Sicherheit verzichten.







a) Normale Fehlerüberlappung

b) Fehlererkennung möglich

c) Ideale Fehlerseparation

Abb. 3: Unterschiedliche Formen der Überlappung von Fehlerbereichen

Bei der Betrachtung der Software-Diversität muß man ferner berücksichtigen, wo Fehler entstehen und wo sie sich auswirken. So können Fehler in den verschiedenen Phasen der Software-Erstellung entstehen, ohne sich notwendigerweise in dieser Phase auch auszuwirken. Die Auswirkung kann in einer späteren Phase erfolgen, hoffentlich spätestens in der Testphase, aber leider auch oft erst während des Betriebs.

1.4 Unterschiedliche Realisierungen der Software-Diversität

In der Regel wird man für die Erstellung von den unterschiedlichen Varianten eines diversitären Software-Systems in erster Linie auch unterschiedliche Teams einsetzen, wobei auch hier wieder graduelle Unterschiede möglich sind, z. B. unterschiedliche Erfahrung, unterschiedlicher Hintergrund, unterschiedliche Ausbildung in den Teams. Neben dieser personenbezogenen Diversität gibt es hauptsächlich noch die werkzeugbezogene Diversität: der Einsatz unterschiedlicher Sprachen, unterschiedlicher Werkzeuge, unterschiedlicher Vorlieferanten. Als eine weitere Ausprägung gibt es eine Mischung aus Software-Diversität und Hardware-Diversität. Diese Form findet z. B. im Airbus seinen Einsatz /Traverse88/ wie auch in modernen speicherprogrammierbaren Steuerungen für hohe Sicherheitsanforderungen /Pilz97/.

Wenn wir einen Phasenplan für die Software-Erstellung mit den Phasen Spezifikation, Entwurf, Codierung, Test und Betrieb voraussetzen, so ist in jeder dieser Phasen Diversität erreichbar. So kann in der Spezifikation mit unterschiedlichen Spezifikationssprachen bzw. Werkzeugen die gewünschte Anzahl von Varianten erstellt werden. Um hierbei einen ausreichenden Diversitätsgrad zu erreichen, werden die verschiedenen Varianten in der Regel auch von unterschiedlichen Personen erstellt.

Nicht nur bei der Erstellung der Software ist auf eine korrekte Implementierung der Diversität zu achten, sondern auch während der Betriebsphase. Werden die unterschiedlichen Varianten, die von getrennten Teams erstellt wurden, später von einem einzigen Team und ohne Rücksicht auf die Diversität gewartet, so kann der erwünschte Nutzen schnell verloren gehen. Auch hier sind gewisse Regeln einzuhalten, um den Zuverlässigkeitsgewinn durch die Diversität zu erhalten.

Es hat eine ganze Reihe von internationalen Experimenten mit den verschiedensten Formen der Diversität gegeben (vgl. z. B. Aufstellung in /Voges89/). Als wesentliche

Erkenntnisse daraus kann man ableiten, daß die ursprüngliche euphorische Meinung, durch die Diversität würde man in einem fehlerfreien Raum landen, da es keine identischen Fehler gibt, sich nicht bestätigt hat. Eine vollständige Unabhängigkeit der einzelnen Varianten existiert nicht, u. a. dadurch hervorgerufen, daß es in der Regel doch immer irgendeine Gemeinsamkeit gibt. Auch die Neigung der Menschen, gewisse Fehler zu wiederholen, hat hieran seinen Anteil (vgl. /Grams90/). Dennoch hat sich die Überzeugung durchgesetzt, daß ein durchdachter und gezielter, vorsichtiger Einsatz von Software-Diversität seinen Nutzen hat. Daher gibt es auch nicht nur Experimente, sondern auch eine ganze Reihe von Realisierungen der Software-Diversität in Systemen, die auch heute noch erfolgreich im Einsatz sind. Hierzu zählen Bereiche in der Luftfahrt /Traverse88/ und im Eisenbahnverkehr /Hagelin88, Kantz95/, sowohl aus der Frühzeit der Diversitätsuntersuchungen wie auch aus heutiger Zeit. Neben diesen Bereichen ist die Software-Diversität in Überlegungen in der Normung von Betriebssystemen /Watanabe95/ und auch in der Anwendung von Ada zu finden /Shokri96/.

1.5 Fehlermodelle

Um den Nutzen von Diversität besser verstehen zu können, müssen wir uns mit den unterschiedlichen Formen und Auswirkungen von Fehlern beschäftigen. Fehler in der Software sind immer vorhanden, werden aber nicht immer aktiviert. Nur bestimmte Eingangsvoraussetzungen führen zu einer Fehleraktivierung, und die Eintrittswahrscheinlichkeit dieser Voraussetzungen ist im allgemeinen nicht bekannt. So kann sich z. B. ein System A sehr lange Zeit korrekt verhalten und erst bei einem seltenen Ereignis fehlerhaftes Verhalten an den Tag legen, während ein anderes System B ständig fehlerhaft ist. In beiden Systemen kann aber die Ursache für dieses Fehlverhalten ein einfacher Tippfehler im Programm sein, bei A in einem Segment, das selten durchlaufen wird, in B in einem oft benutzten Teil.

Da sich die Software im Laufe der Zeit nicht verändert, also nicht altert, entstehen die Fehler in der Software auch nicht von alleine, sondern werden vom Ersteller mit kreiert. Dies kann in jeder Phase der Erstellung geschehen. Die Ursache für einen Fehler kann einerseits in der Phase selbst begründet sein (z. B. Tippfehler) wie auch an einer früheren Phase liegen (z. B. schwer verständliche und daher falsch umgesetzte Spezifikation). Bei einer einfachen Software-Entwicklung spielt dies keine entscheidende Rolle, für die Software-Diversität allerdings schon. So kann eine mißverständliche Spezifikation von verschiedenen Teams unterschiedlich interpretiert und umgesetzt werden, was zu unterschiedlichen Lösungen führt, die sich - hoffentlich - nicht alle gleichzeitig fehlerhaft auswirken und somit später als fehlerhaft erkannt werden. Das Fehlermodell für diversitäre Software ist also entsprechend aufwendiger.

Bei einem mehrfach diversitär redundantem System ist außerdem zu berücksichtigen, ob wir es mit identischen oder nicht-identischen Fehlern zu tun haben. Identische Fehler sind solche Fehler, die in ihrer Auswirkung identische Folgen haben, also bei gleichen Eingabebedingungen denselben fehlerhaften Wert liefern. Dies sind die Fehler, die trotz Diversität nicht erkannt werden können und die damit möglichst vermieden werden sollen (bzw. minimiert werden sollen, da ein vollständiger Fehlerausschluß kaum möglich ist).

Wenn wir der Einfachheit halber und ohne Einschränkung der Allgemeingültigkeit davon ausgehen, daß wir die Software-Entwicklungsphasen Spezifikation, Entwurf, Codierung, Test und Betrieb haben, so setzt sich die Menge der Fehler F in einem einfachen System aus den Fehlern zusammen, die in den einzelnen Phasen gemacht und nicht wieder eliminiert wurden:

(1) $F(\text{Strang}) = F(\text{Spez}) \cup F(\text{Entw}) \cup F(\text{Cod}) \cup F(\text{Test}) \cup F(\text{Betr})$

Die Anzahl der sich auswirkenden, nicht erkannten Fehler FA ergibt sich zu

(2) $\text{FA}(\text{Strang}) = | F(\text{Strang}) |$

Andererseits ist FE als die Anzahl der in einem einsträngigen System (oder auch in einem homogen redundanten System) erkennbaren Fehler

(3) $\text{FE}(\text{Strang}) = 0$

In einem dreifach modularen System (TMR-System) haben wir analog für die Menge der Fehler die Vereinigung der Fehler in den einzelnen Strängen:

(4) $F(\text{TMR}) = F(\text{Strang A}) \cup F(\text{Strang B}) \cup F(\text{Strang C})$

Die sich auswirkenden Fehler FA reduzieren sich aber im Fall eines 2-von-3-Voters auf die Fehler, die in mindestens zwei Strängen gleichzeitig auftreten:

(5) $\text{FA}(\text{TMR}) = | F(\text{Strang A}) \cap F(\text{Strang B}) | + | F(\text{Strang A}) \cap F(\text{Strang C}) |$
$+ | F(\text{Strang B}) \cap F(\text{Strang C}) |$
$- 2 | F(\text{Strang A}) \cap F(\text{Strang B}) \cap F(\text{Strang C}) |$

Für die erkennbaren Fehler FE ergibt sich im Fall eines 2-von-3-Voters in einem TMR-System:

(6) $\text{FE}(\text{TMR}) = | F(\text{Strang A}) \setminus F(\text{Strang B}) \setminus F(\text{Strang C}) |$
$+ | F(\text{Strang B}) \setminus F(\text{Strang A}) \setminus F(\text{Strang C}) |\setminus$
$+ | F(\text{Strang C}) \setminus F(\text{Strang B}) \setminus F(\text{Strang A}) |$

d.h. alle Fehler, die nur in einem Strang auftreten, können erkannt werden und sind damit im wesentlichen gefahrlos.

Die Summe der erkennbaren Fehler und der sich auswirkenden Fehler ergibt wiederum die Gesamtzahl der Fehler im TMR-System:

(7) $\text{FA}(\text{TMR}) + \text{FE}(\text{TMR}) = | F(\text{TMR}) |$

Als Beispiel für die Anwendung des Fehlermodells ist in Tabelle 2 aufgezeigt, wie sich die Fehlerzahlen in einem TMR-System mit homogener Redundanz von denen mit unterschiedlicher diversitärer Redundanz unterscheiden. Unter der Annahme, daß 30% der Fehler jeweils in der Spezifikation, dem Entwurf und der Codierung sowie die restlichen in Test und Betrieb gemacht werden und im Rahmen einer diversitären Entwicklung 10% der Fehler identische Fehler sind, ergibt sich für die sich auswirkenden Fehler eine Reduktion von 100% im homogen redundanten System auf 79% bzw. 37% bei zwei unterschiedlichen Arten der diversitären Realisierung. Dabei ist

von der konservativen Annahme ausgegangen, daß keine Nebeneffekte auftreten. Diese Nebeneffekte der Diversität sind i.a. positiv, d.h. der Einsatz von Diversität in einer Phase hat positive, fehlerreduzierende Auswirkungen auf andere Phasen der Programmentwicklung.

	Spez.	**Entwurf**	**Codierung**	**Test**	**Betrieb**	**Summe**
Fehlerverteilung	30	30	30	5	5	100
ident. Fehler = 10%	3	3	3	0,5	0,5	10
2v3 homogen	30	30	30	5	5	100
2v3 div. Codierung	30	30	9	5	5	79
2v3 div. S+E+C	9	9	9	5	5	37

Tab. 2: Beispiel für die Anwendung des Fehlermodells
Prozentualer Anteil der Fehler

1.6 Kostenmodell

Je nach Art der Software-Diversität, die innerhalb eines Projektes realisiert wird, entstehen auch unterschiedliche Kosten: n-fache Diversität bedeutet nicht pauschal n-fache Gesamtkosten. Um am Anfang eines Projektes entscheiden zu können, ob Software-Diversität als Fehlertoleranz-Methode eingesetzt werden soll, oder ob lieber Fehlervermeidungs-Methoden benutzt werden sollen, ist eine Kosten-Nutzen-Analyse durchzuführen. Diese benötigt Abschätzungen sowohl für die Kosten als auch den Nutzen der jeweiligen Verfahren.

Bei den Kosten für Software-Diversität kann in einer ersten groben Näherung davon ausgegangen werden, daß bei Einsatz von n Teams die entsprechenden Kosten ebenfalls n-fach sind. Bei genauerer Analyse wird man aber feststellen, daß dies nicht in allen Punkten gilt: es gibt diversitätsunabhängige Kosten, linear diversitätsabhängige Kosten und nichtlineare diversitätsabhängige Kosten, aber auch Zusatzkosten und ggf. Einsparungen durch die Diversität.

Die Kosten einer diversitären Softwareentwicklung ergeben sich aus den Kosten in den einzelnen Entwicklungsphasen:

$$K_{dSW} = K_{dS} + K_{dE} + K_{dC} + K_{dT} + K_{dB}$$

mit

$$K_{dX} = (1-a_X-d_X)*K_X + f_X(n_X)*d_X*K_X + n_X*a_X*K_X + g_X(n_X)*D_X$$

Dabei bedeutet

$X \varepsilon \{ S, E, C, T, B\}$ die einzelnen Phasen Spezifikation, Entwurf, Codierung, Test und Betrieb

$(1-a_X-d_X)*K_X$	die diversitätsunabhängigen Kosten
$f_X(n_X)*d_X*K_X$	die partiell diversitätsabhängigen Kosten
$n_X*a_X*K_X$	die linear diversitätsabhängigen Kosten
$g_X(n_X)*D_X$	die diversitätsbezogenen Zusatzkosten

Diese Kosten sind der bei einer singulären Entwicklung entstehenden Kosten gegenüberzustellen (vgl. Abb. 4).

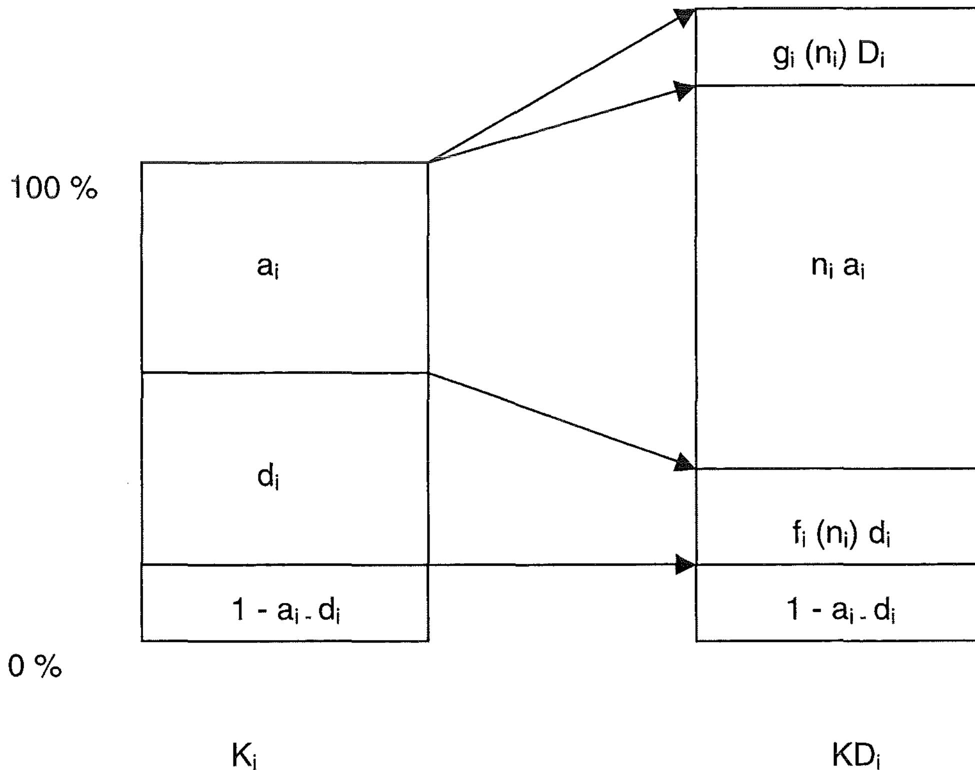


Abb. 4: Kosten einer Softare-Entwicklungshase i bei einfacher Erstellung und bei n_i-facher Diversität

In Abb. 5 sind die Kosten der Software-Enwicklungsphase Testen einer einfachen Entwicklung verglichen mit denen einer n_c-fachen Diversität. Dabei bezieht sich a_t z. B. auf den Modultest, der für jeden Modul gemacht werden muß und wo die Kosten damit n_c-fach sind. Die Kosten für den Vergleichstest (d_T) hingegen können sich reduzieren, da kein Vergleichsmodul zu entwickeln ist, die diversitär entwickelten Varianten können hierfür genutzt werden. Die Testdatenerstellung ($1 - a_T - d_T$) ist nur einmal zu machen, die entsprechenden Kosten verändern sich nicht durch die mehrfach Codierung. Zusätzlich muß in der Testphase auch der Toleranzvoter getestet werden, was einen Mehraufwand ($g_T (n_C) D_T$) gegenüber der einfachen Erstellung bedeutet.

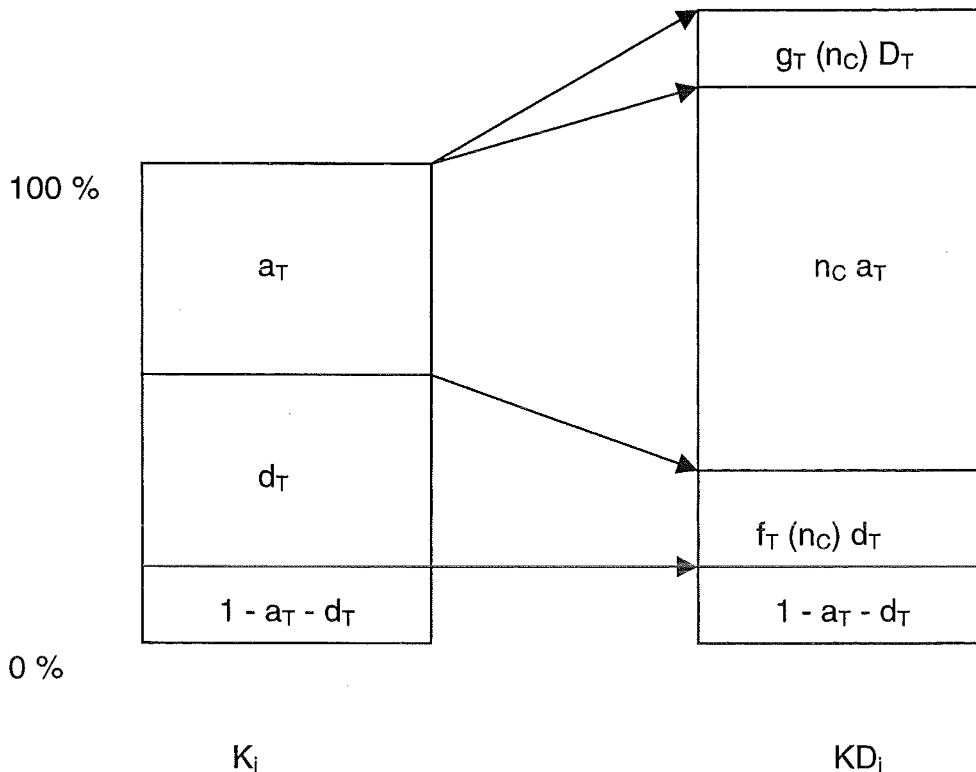


Abb. 5: Kosten einer Softare-Entwicklungshase Testen bei einfacher Erstellung und bei n_c-facher Diversität

Neben dem Vergleich der Kosten für die singuläre und diversitäre Entwicklung sind die Fehlermodelle und die jeweiligen Zuverlässigkeitsmodelle für die Abschätzung des Nutzens zu benutzen. Dabei sind nicht nur allgemeine Erfahrungswerte aus der Literatur heranzuziehen, sondern auch die eigenen Projekterfahrungen. Darauf kann dann eine Entscheidung für oder gegen den Einsatz von Software-Diversität aufbauen.

1.7 Nutzen der Diversität

Zusammenfassend soll noch einmal der Nutzen der Diversität geschildert werden. Im Vergleich zu einer homogen redundanten Software-Lösung haben wir bei der diversitär redundanten Software-Lösung die berechtigte Erwartung, daß sich die Zahl der identischen Fehler reduziert. In verschiedenen Experimenten und Realisierungen der Diversität hat es sich gezeigt, daß die Zahl der identischen Fehler (bzw. die Zahl der in einer Mehrheit der Varianten vorliegenden Fehler) im Mittel bei etwa 10% lag (vgl. /Voges89/). Neben dieser Reduktion der sich auswirkenden Fehler haben wir eine aktive Fehlererkennung sowohl während der Entwicklung der Software (wenn wir die verschiedenen Varianten zwischendurch vergleichen bzw. auch z. B. durch die direkte oder indirekte Kommunikation der Entwicklungsteams über die Interpretation der Spezifikation) als auch während der Benutzung und dem Betrieb. Diese Fehlererkennung kann insbesondere in sicherheitsrelevanten Bereichen eine hohe Bedeutung erhalten, da es sich oft zeigt, daß es die vergessenen seltenen Fälle und die sich ändernden Randbedingungen sind, die Probleme bereiten. Zumindest die Wahrscheinlichkeit, daß sich die diversitär entwickelten Programme nicht alle gleichartig fehlerhaft verhalten, ist größer als in einem homogenen System.

Während des Betriebs haben wir, je nach Auslegung des Voters bzw. Vergleichers, auch eine aktive Fehlermaskierung. So werden sich Fehler, die in einer Minderheit der Programme auftritt, nicht auf das Systemverhalten negativ auswirken. Zumindest eine partielle Fehlertoleranz, nämlich der Fehler, die nur in einer Minderheit auftritt, ist dadurch gegeben. Bei der Realisierung eines sicherheitsrelevanten Systems sollte man aber darauf achten, daß diese maskierten Fehler nicht einfach unter den Tisch fallen, sondern daß eine sorgfältige Dokumentation und Analyse dieser Fehler erfolgt. Dies kann aber off-line erfolgen.

Als eine einfache Form der Bestimmung des Grades der Diversität kann man den Anteil der identischen Fehler im System heranziehen. Daraus kann sich ergeben, daß sich im Laufe der Benutzung des Systems eine Neuberechnung der Diversitätsgrades ergibt.

Der Einsatz der Diversität kann sich auch auf einzelne Entwicklungsschritte beschränken. So kann es sinnvoll sein, eine parallele Programmentwicklung zu vollziehen, die Programme auch einem Test zu unterziehen, aber dann z. B. aufgrund der Fehlerzahlen das beste Programm zu selektieren und nur ein Programm im Betrieb zu übernehmen.

Ein wichtiger Punkt ist, daß die Software-Diversität nicht nur Auswirkungen auf die Software hat, sondern auch auf die Hardware. Dies betrifft sowohl Design- wie auch Alterungsfehler in der Software. Die Wahrscheinlichkeit, daß sich z. B. ein Designfehler in der Hardware bei Einsatz von diversitärer Software identisch auswirkt, ist gering. Ebenso ist es mit Alterungsfehlern, die bei diversitärer Software, die auf derselben Hardware abläuft, in der Regel auch zu unterschiedlichen Aktivierungen bzw. Ergebnissen führt und daher erkennbar wird.

Der Modellierung der Software-Diversität ist ausreichend Augenmerk zu schenken. Neben den hier präsentierten Ansätzen gibt es eine Reihe weiterer Modelle mit unterschiedlichen Schwerpunkten (vgl. Eckhardt85, Littlewood89, Popov98, Bastani96, Kim96/).

1.8 Schlußbemerkung

Der Einsatz von Software-Diversität in einem System mit hohen Zuverlässigkeitsanforderungen sollte gut abgewogen werden. Ein blindes Vertrauen auf den vermeintlichen Nutzen der Diversität kann leicht ins Auge gehen. Vor dem Einsatz von Diversität ist abzuwägen, welche Fehlervermeidungsverfahren alternativ und mit welchem Nutzen eingesetzt werden können. Die Form der Diversität, d.h. in welcher Phase der Systementwicklung wird welche Art der Diversität realisiert, ist sorgfältig zu wählen und zu kontrollieren. Untersuchungen der Varianten müssen aufzeigen können, daß der erforderliche bzw. erwartete Grad an Diversität auch erreicht wurde, da sonst irrtümlicherweise von einer hohen Zuverlässigkeit des Systems ausgegangen wird. Es muß klar sein, welchen Beitrag die Software-Diversität zur System-Zuverlässigkeit leisten soll und im Endeffekt auch leistet. Die Planung muß anhand von Meßergebnissen überprüft werden. Nur so ist der Einsatz von Diversität zur Steigerung der Sicherheit des Systems sinnvoll und gerechtfertigt.

1.9 Literatur

/Avizienis77/
A. Avizienis, L. Chen: "On the Implementation of N-Version-Programming for Software Fault-Tolerance during Program Execution." Proc. COMPSAC'77, Chicago, IL, USA, Nov. 1997, pp. 149-155.

/Bastani96/
F. Bastani, B. Cukic, V. Hilford, A. Jamoussi: "Towards Dependable Safety-Critical Software." Proc. 2nd Workshop on Object-Oriented Real-Time Dependable Systems - WORDS'96, 1996.

/DIN94/
DIN V VDE 0801/A1: Grundsätze für Rechner in Systemen mit Sicherheitsaufgaben; Änderung A1: 1994-10. Beuth Verlag Berlin 1994.

/Eckhardt85/
D.E. Eckhardt, L.D. Lee: "A Theoretical Basis for the Analysis of Multiversion Software Subject to Coincident Errors." IEEE TSE, SE-11 (1985), pp. 1511-1517.

/Fuhrman95/
C.P. Fuhrman, S. Chutani, H.J. Nussbaumer: "Hardware/Software Fault Tolerance with Multiple Task Modular Redundancy." Proc. Intern. Symp. on Computers and Communications, ISCC'95, 1995.

/Grams90/
T. Grams: "Denkfallen und Programmierfehler." Springer Verlag Heidelberg, 1990.

/Hagelin88/
G. Hagelin: "ERICSSON Safety Systems for Railway Control." In /Voges88/, pp. 11-21.

/Kanekawa98/
N. Kanekawa, T. Meguro, K. Isono, Y. Shima, N. Miyazaki, S. Yamaguchi: "Fault Detection and Recovery Coverage Improvement by Clock Synchronized Duplicated Systems with Optimal Time Diversity." Proc. 28th Ann. Intern. Symp. on Fault-Tolerant Computing - FTCS-28, 23-25 June 1998, Munich, Germany, pp. 196-200.

/Kantz95/
H. Kantz, C. Koza: "The ELEKTRA Railway Signalling-System: Field Experience with an Actively Replicated System with Diversity." Proc. 25th Ann. Intern. Symp. on Fault-Tolerant Computing - FTCS-25, 1995.

/Kim96/
K. Kim, M.A. Vouk, D.P. McAllister: "An Empirical Evaluation of Maximum Likelihood Voting in Failure Correlation Conditions." Proc. 7th Intern. Symp. on Software Reliability Engineering ISSRE'96, 1996.

/Littlewood89/
B. Littlewood, D.R. Miller: "Conceptual Modelling of Coincident Failures in Multi-Version Software." IEEE TSE, SE-15 (1989), pp. 1596-1614.

/Parhami96/
B. Parhami: "Design of Reliable Software via General Combination of N-Version Programming and Acceptance Testing." Proc. 7th Intern. Symp. on Software Reliability Engineering ISSRE'96, 1996.

/Pilz97/
Pilz: "Programmierbare Sicherheitssteuerungen sicher + wirtschaftlich." SPS Magazin, 10. Jahrg., Ausg. 7 (Oktober 1997), pp. 12-14.

/Popov98/
P. Popov, L. Strigini: "Conceptual Models for the Reliability of Diverse Systems - New Results." Proc. 28th Ann. Intern. Symp. on Fault-Tolerant Computing - FTCS-28, 23-25 June 1998, Munich, Germany, pp. 80-89.

/Randell75/
B. Randell: „System Structure for Software Fault Tolerance.“ IEEE Trans. Soft. Eng., SE-1 (1975), pp. 220-232.

/Shokri96/
E.H. Shokri, K.S. Tso: "Ada 95 Object-Oriented and Real-Time Support for Development of Software Fault Tolerance Reusable Components." Proc. 2nd Workshop on Object-Oriented Real-Time Dependable Systems - WORDS'96, 1996.

/Sullivan95/
G.F. Sullivan, D.S. Wilson, G.M. Masson: "Certification of Computational Results." IEEE Trans. on Computers, Vol. 44, No. 7, July 1995.

/Traverse88/
P. Traverse: „Airbus and ATR System Architecture and Specification.“ In /Voges88/, pp. 95-104.

/Tsai98/
T. Tsai: "Fault Tolerance Via N-Modualr Software Redundancey." Proc. 28th Ann. Intern. Symp. on Fault-Tolerant Computing - FTCS-28, 23-25 June 1998, Munich, Germany, pp. 210-206.

/Voges88/
U. Voges (ed.): "Software Diversity in Computerised Control Systems." Springer-Verlag Wien 1988.

/Voges89/
U. Voges: "Software-Diversität und ihre Modellierung - Software-Fehlertoleranz und ihre Bewertung durch Fehler- und Kostenmodelle." Informatik-Fachberichte 224, Springer-Verlag Berlin 1989.

/Watanabe95/
A. Watanabe, K. Sakamura: "Design Fault Tolerance in Operating Systems Based on a Standardization Project." Proc. 25th Ann. Intern. Symp. on Fault-Tolerant Computing - FTCS-25, 1995.

Systematische Prüfung funktionaler und temporaler Korrektheit von Ada-Software

2 Systematische Prüfung funktionaler und temporaler Korrektheit von Ada-Software

Ines Fey

Daimler-Benz AG

Forschung und Technologie

Alt-Moabit 96a

10559 Berlin

email: Ines.Fey@dbag.bln.daimlerbenz.com

Abstract

In dem vorliegenden Papier wird eine bei der Daimler-Benz-Forschung entwickelte Teststrategie vorgestellt, die den Test funktionaler Aspekte und die Prüfung des Zeitverhaltens von Ada-Software verbindet und in ein durchgängiges rechnerunterstütztes Vorgehen einbindet. Für den funktionsorientierten Test wurde der Klassifikationsbaum-Editor CTE mit dem Ada-Testwerkzeug AdaTEST der Firma IPL verbunden, um eine weitgehende Durchgängigkeit der Werkzeugunterstützung zu realisieren. Das Zeitverhalten von Ada-Programmen wird auf Basis von evolutionären Algorithmen durch die Suche nach maximalen bzw. minimalen Laufzeiten geprüft, die die in der Spezifikation vorgegebenen Grenzwerte über- bzw. unterschreiten. Den Ausgangspunkt bei dieser Suche bilden die für den funktionalen Test ermittelten Testdaten. Der Gesamtansatz wurde bereits erfolgreich an Applikationen der Daimler-Benz-Forschung erprobt.

2.1 Einleitung

In verschiedenen Geschäftsbereichen des Daimler-Benz-Konzerns, insbesondere in der Luft- und Raumfahrt, werden Software bzw. software-basierte Systeme mit der Programmiersprache Ada codiert. Speziell beim Einsatz in sicherheitskritischen Bereichen ergeben sich hohe Anforderungen an die Qualität und Zuverlässigkeit dieser Systeme. Ein zentraler Aspekt ist hierbei die sichere Funktion der Software. Dementsprechend muß sich der Entwicklungsprozeß durch den Einsatz von Verfahren auszeichnen, die das Vertrauen speziell in die Verläßlichkeit der Software-Anteile bestärken. Die in der Praxis am weitesten verbreitete analytische Qualitätssicherungsmaßnahme ist der dynamische Test. Eine gründliche Prüfung unter Berücksichtigung der realen Umgebung stellt die verläßliche Funktion sicher und fördert die Akzeptanz des Systems, ist jedoch in der Regel sehr zeitintensiv. Der Einsatz leistungsfähiger Methoden und Werkzeuge im Bereich des Testens kann den Aufwand bei der Entwicklung von Software deutlich reduzieren. Für den Test von Ada-Programmen sind eine Reihe von kommerziellen Werkzeugen erhältlich. In der Regel bieten diese eine

spezialisierte Funktionalität für ausgewählte Bereiche des Tests an, wie Testausführung, Monitoring und Testauswertung. Ein Testsystem, daß eine umfassende Werkzeugunterstützung für sämtliche Testaktivitäten bereitstellt, ist nicht verfügbar. Ebenso fehlen noch praxisgeeignete Verfahren und Werkzeuge für die häufig notwendige Prüfung des Laufzeitverhaltens.

Das Software-Technologie-Labor der Daimler-Benz-Forschung beschäftigt sich in der Abteilung Methoden und Tools mit der Erarbeitung effizienter Verfahren und Vorgehensweisen für die Entwicklung und den Test von Software- bzw. Hardware/Software-Systemen sowie mit der Konzipierung entsprechender Werkzeugunterstützung. In diesem Zusammenhang wurde ein Vorgehen zum systematischen Test von Ada-Programmen entwickelt. Im vorliegenden Beitrag wird eine Kombination aus einem Ansatz zur systematischen Durchführung funktionsorientierter Tests und einem Vorgehen zur Abschätzung der Ausführungszeiten vorgestellt.

Kapitel 1 enthält eine Einführung in die für einen systematischen Test notwendigen Testaktivitäten. Es folgen die Beschreibung der Klassifikationsbaum-Methode und des zugehörigen Werkzeugs CTE als Unterstützung für die Testfallermittlung, die Darstellung des für die Integration mit dem CTE ausgewählten Testsystems AdaTEST der Firma IPL sowie die Beschreibung der Kombination dieser Werkzeuge. In Kapitel 2 wird ein Ansatz zur Prüfung des temporalen Verhaltens bei Realzeit-System vorgestellt. Die Kombination der Verfahren zum funktionalen und zeitlichen Test in einer Teststrategie ist in Kapitel 3 erläutert. Eine Zusammenfassung mit Ausblick auf weitere Arbeiten schließt den Beitrag ab.

2.2 Systematischer Test funktionaler Aspekte

Der systematische Test funktionsorientierter Gesichtspunkte ist ein wesentlicher Teil des Verifikations- und Validationsprozesses für Software, da nur so geprüft werden kann, ob alle spezifizierten Requirements umgesetzt wurden. Ziel des Tests ist es, vorhandene Fehler im funktionalen Verhalten des Testobjekts aufzudecken. Zusätzlich bestärken fehlerfreie Ausführungen des Testobjekts mit ausgewählten Eingabekonstellationen das Vertrauen in die Funktion der Software.

Der Testprozeß kann in einzelne voneinander abgegrenzte Aktivitäten unterteilt werden, wie die Testfallermittlung, die Testdaten- und Sollwertbestimmung, die Testdurchführung, das Monitoring, die Testauswertung und die Testdokumentation [1]. Während der Testfallermittlung werden die zu testenden Eingabesituationen festgelegt. Ein Testfall abstrahiert von konkreten Testdaten und beschreibt die Eingabekonstellation nur soweit, wie für den gewünschten Test notwendig. Im Verlauf der Testdatenbestimmung werden für jeden Testfall konkrete Werte, die Testdaten, ausgewählt, die den Bedingungen der Testfallbeschreibung genügen. Die Vorgabe von Ergebnissen (Sollwerten), die zu den einzelnen Testfällen erwartet werden, ist Inhalt der Sollwertbestimmung. Das Testobjekt wird mit den ausgewählten Testdaten ausgeführt und dabei werden die aktuellen Ausgabewerte (Istwerte) berechnet. Die Testergebnisse werden durch einen Vergleich von Soll- und Istwerten ermittelt. Zusätzlich kann Monitoring genutzt werden, um das Testobjekt während des Testlaufs zu überwachen und weitere Informationen zum Verhalten zu erlangen. Eine gebräuchliche Form ist die Instrumentierung des Source Codes gemäß einem Strukturtest-Kriterium.

Aufgrund des Umfangs der Software können im allgemeinen keine vollständigen Prüfungen aller möglichen Eingabesituationen durchgeführt werden. Der Test hat somit immer Stichprobencharakter. Deshalb ist die entscheidende Voraussetzung für einen gründlichen Test die Ermittlung relevanter Testfälle, da damit die Qualität des Tests maßgeblich beeinflußt wird. Ein Testfall beschreibt eine zu testende Eingabesituation und umfaßt eine Menge von Eingabewerten aus dem Eingabedatenraum des Testobjekts. Für jedes Element dieser Menge soll sich das Testobjekt gleich verhalten, d.h. beispielsweise die gleiche Funktionalität, der gleiche Programmpfad oder der gleiche Systemzustand soll ausgeführt/erreicht werden. Eine geeignete Methode für die Unterstützung der Testfallermittlung ist die Klassifikationsbaum-Methode. Mit dem Klassifikationsbaum-Editor (CTE) steht außerdem ein entsprechendes Werkzeug zur Verfügung, das speziell für den Einsatz dieser Methode entwickelt wurde.

2.2.1 Die Klassifikationsbaum-Methode

Die Klassifikationsbaum-Methode, eine Erweiterung der von Ostrand und Balcer entwickelten Category-Partition-Method, ist ein Verfahren für die Ermittlung von Testfällen auf der Basis einer Spezifikation und der Schnittstellenbeschreibung des Testobjekts. Wesentlicher Aspekt der Methode ist die graphische Visualisierung der beim Test berücksichtigten Gesichtspunkte und der für den Test getroffenen Fallunterscheidungen. Eine ausführliche Beschreibung der Methode mit praktischen Beispielen und Erfahrungen aus dem Projekteinsatz enthält [3].

Im folgenden wird die Methode an dem Beispiel der Ada-Funktion *is_line_covered_by_rectangle* erläutert. Diese aus der graphischen Datenverarbeitung stammende Funktion prüft, ob eine gegebene Linie von einem Rechteck überdeckt wird. Die Funktion hat zwei Strukturen als Parameter, von denen die eine die Endpunkte der Linie enthält und die andere das Rechteck durch den linken oberen Eckpunkt, die Breite und die Höhe beschreibt. Wenn die Linie durch das Rechteck überdeckt wird, soll das Testobjekt den Wert *true* als Rückgabewert liefern, andernfalls *false*.

Abbildung 1 zeigt mögliche Positionen der Linienendpunkte und definiert acht Bereiche in der Umgebung des Rechtecks, um die Punktpositionen zu beschreiben.

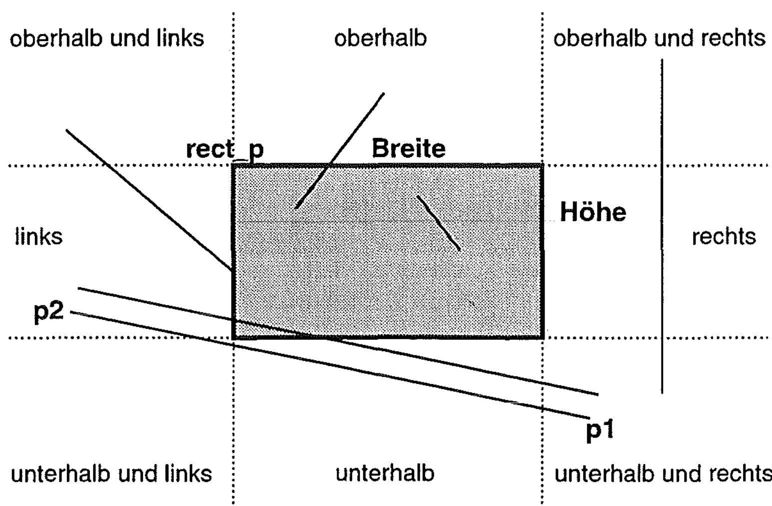


Abbildung 1 Rechteck mit Beispiel-Linien

Die Testfallermittlung mit der Klassifikationsbaum-Methode erfolgt in zwei Schritten:

1. Zunächst muß der Tester die für den Test relevante Gesichtspunkte des Eingabedatenraums festlegen. Unter jedem Gesichtspunkt wird die Menge der möglichen Eingaben in Teilmengen, sogenannte Klassen, zerlegt. Entstandene Klassen können wiederum durch weitere Klassifikationen unterteilt werden. Diese stufenweise Zerlegung des Eingabedatenraums in Klassen wird in Form eines Klassifikationsbaums graphisch dargestellt und dann als Kopf einer Kombinationstabelle verwendet.

2. Der zweite Schritt umfaßt die eigentliche Definition von Testfällen. Die Testfälle werden durch die Kombination von Klassen unterschiedlicher Klassifikationen spezifiziert. Für die Festlegung der Testfälle werden in der im ersten Schritt erzeugten Kombinationstabelle Markierungen derart gesetzt, daß verschiedene Gesichtspunkte für den Test miteinander kombiniert werden. Die Zeilen der Tabelle symbolisieren die Testfälle, die Spalten werden durch die nicht weiter zerlegten Klassen des Klassifikationsbaums gebildet. Durch die Markierung von Tabellenkreuzungen werden dann wie in Abbildung 2 gezeigt die Testfälle definiert.

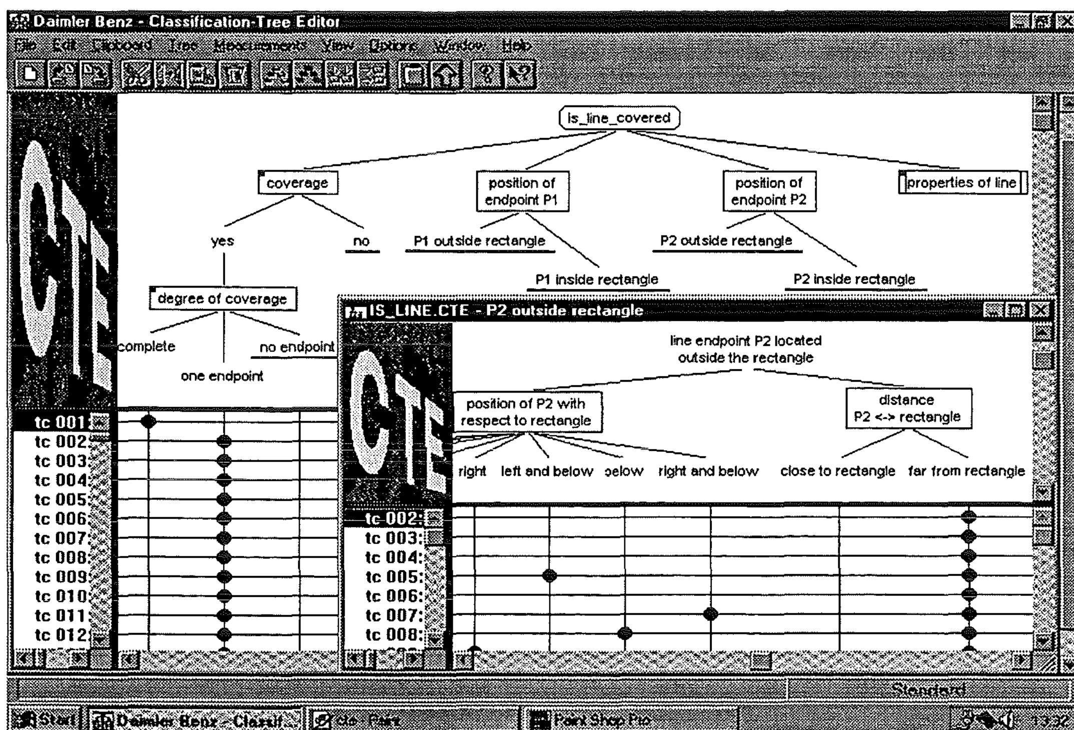


Abbildung 1 Testfallermittlung für das Beispiel

Die graphische Darstellung der zu testenden Aspekte und die Definition von Testfällen in der Kombinationstabelle bieten eine kompakte Testfallermittlung und anschauliche Testfallspezifikation.

2.2.2 Der Klassifikationsbaum-Editor CTE

Der CTE ist ein graphisches Werkzeug zur Ermittlung von Testfällen mit der Klassifikationsbaum-Methode. Der Klassifikationsbaum-Editor CTE ermöglicht dem Tester eine objektorientierte Erstellung und Bearbeitung von Klassifikationsbäumen und Kombinationstabellen. Dem zweistufigen Vorgehen bei der Testfallermittlung entsprechend ist die Arbeitsfläche des CTE in einen Zeichenbereich (oberer Teil des Bildschirms) für den Klassifikationsbaum und eine Tabelle (unterer Teil des Bildschirms) für die Festlegung der Testfälle unterteilt. Abbildung 2 zeigt die typische Bildschirmmaske des CTE.

Zur Dokumentation der systematisch ermittelten Testfälle können verschiedene Reports generiert werden. Beispielsweise ist das Ausdrucken des Klassifikationsbaums mit der zugehörigen Kombinationstabelle möglich. Außerdem kann auf Basis der Tabelle eine textuelle Darstellung der Testfälle erzeugt werden. Einerseits bietet diese Textdarstellung der Testfälle eine gute Dokumentationsmöglichkeit, andererseits ist dies auch eine geeignete Darstellung, die in nachfolgenden Testaktivitäten wie der Testdatenbestimmung verwendet werden kann. Da der CTE sich auf die Testfallermittlung konzentriert, sind zusätzliche Werkzeuge erforderlich, um einen effizienten, weitgehend automatisierten Testprozeß zu erreichen.

2.2.3 Integration von CTE und AdaTEST

Klassifikationsbaum-Methode und CTE können zusätzlich zu anderen Testwerkzeugen eingesetzt werden und so die erwähnte Lücke bei der Unterstützung des Testprozesses für Ada-Software ausfüllen. Um hier eine möglichst weitreichende Automatisierung und damit effizientere Arbeitsweise beim Test zu ermöglichen, wurde eine Schnittstelle zwischen dem CTE und einem leistungsfähigen Ada-Testwerkzeug (AdaTEST) realisiert.

Untersuchungen verschiedener Ada-Testsysteme zeigten, daß sich die Werkzeuge vor allem auf die Automatisierung der Testdurchführung und das Monitoring konzentrieren [6]. Ausgewählte Testwerkzeuge bieten Komponenten für die Testdaten- und Sollwertbestimmung an. Eine dem CTE adäquate Funktionalität für die Testfallermittlung wird von keinem der Testwerkzeuge angeboten. Um eine möglichst umfangreiche Unterstützung für alle Testaktivitäten mit weitgehender Automatisierung zu erreichen, wurde ein Konzept zur Integration des CTE mit einem kommerziell verfügbaren leistungsfähigen Ada-Testsystem entwickelt. Aus den am Markt erhältlichen Werkzeugen wurde AdaTEST von der Firma IPL [1] für die Anbindung des CTE ausgewählt. Das Konzept kann aber auch auf andere Testwerkzeuge mit entsprechenden Importmöglichkeiten übertragen werden.

Kern der Integration ist die sogenannte Test Case Definition Datei (TCD-Datei), die dem Werkzeug AdaTEST als Input für die Testtreiber-Generierung dient. Um die Anbindung des CTE an AdaTEST zu realisieren, wurde der CTE um eine Export-Schnittstelle erweitert. Diese Schnittstelle erleichtert die automatische Erzeugung von TCD-Dateien, die die graphisch spezifizierten Testfälle zu einem Klassifikationsbaum widerspiegeln. Zusammen mit den Testfallnummern wird eine textuelle Repräsentation der Testfälle generiert und in die TCD-Datei eingetragen. Namen und Kommentare zu einem Testfall können optional ebenfalls in die TCD-Datei übernommen werden.

Die erzeugte TCD-Datei kann von AdaTEST direkt für die Generierung des Testtreibers weiterverwendet werden. Die mit dem CTE erstellten Testfälle stellen in der Regel eine abstrakte Beschreibung der zu prüfenden Eingabesituationen dar und enthalten somit noch keine konkreten Testdaten oder Sollwerte. Deshalb muß der Benutzer in den meisten Fällen die TCD-Datei noch um Testdaten und Sollwerte zu den einzelnen Testfällen ergänzen.

Neben der Generierung des Testtreibers führt AdaTEST auch die Instrumentierung des Testobjekts automatisch durch. Das instrumentierte Testobjekt und der Testtreiber werden anschließend übersetzt und zu einem ausführbaren Testprogramm zusammengebunden, das die Testdurchführung, die Testauswertung und das Monitoring automatisiert. Als Ergebnis wird die Testdokumentation mit den Testresultaten generiert.

Im folgenden wird die Verwendung der beiden Werkzeuge am Beispiel der Funktion *is_line_covered_by_rectangle* illustriert.

2.2.4 Beispiel

Das Beispiel-Testobjekt *is_line_covered_by_rectangle* ist als Funktion in dem Package *graphics* realisiert.

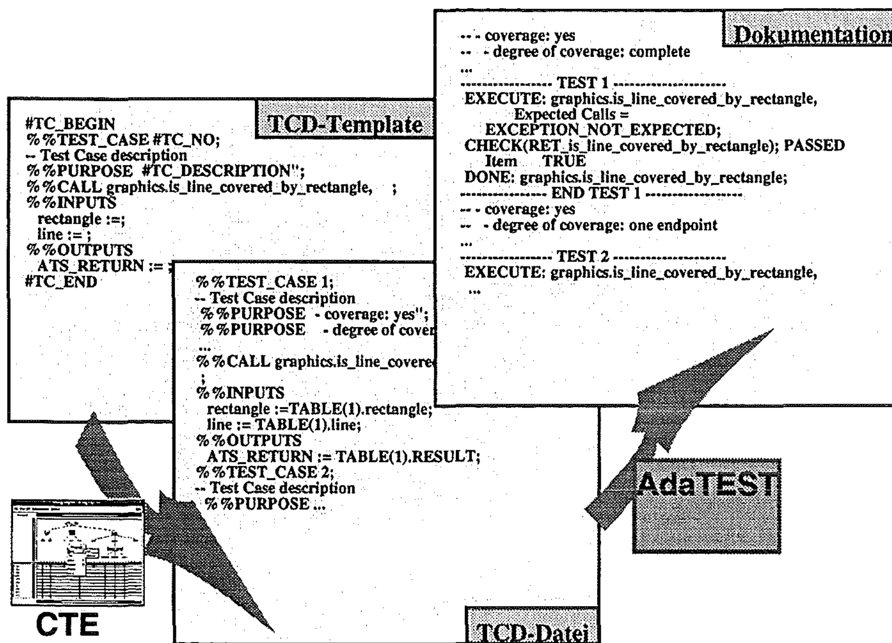


Abbildung 3: Schnittstelle zwischen CTE und AdaTEST

Den ersten Schritt beim Testen mit CTE/AdaTEST bildet die bereits in Kapitel 2 beschriebene systematische Testfallermittlung, die mit dem CTE durchgeführt wird. Zunächst wird mit Hilfe des CTE ein Klassifikationsbaum erstellt, der die testrelevanten Aspekte abdeckt. Auf der Basis dieses Klassifikationsbaums werden im Anschluß die Testfälle in der Kombinationstabelle des CTE festgelegt. Für die Beispielfunktion wurden 49 relevante Testfälle festgelegt. Die CTE-Darstellung in Bild 2 zeigt einen Ausschnitt dieses Klassifikationsbaums und der zugehörigen Kombinationstabelle.

Abbildung 3 zeigt zusammenfassend Ausschnitte der drei wesentlichen Dateien, die bei der Kombination von CTE und AdaTEST für die Beispielfunktion erstellt bzw. generiert wurden (TCD-Template, TCD-Datei, Testdokumentation) sowie den Zusammenhang dieser Dateien und ihren Bezug zu den Werkzeugen.

2.2.5 Praktische Erfahrungen

Der Klassifikationsbaum-Editor CTE wird bereits seit mehreren Jahren erfolgreich in verschiedenen Projekten für die systematische Testfallermittlung eingesetzt [4]. Im Vergleich zur ursprünglichen Herangehensweise an den Test konnte die Qualität der Testfälle, insbesondere in bezug auf die Fehleraufdeckungsrate, den Überdeckungsgrad und die Redundanzfreiheit, deutlich verbessert und der Testaufwand spürbar gesenkt werden. Um den CTE auch mit anderen Produkten integrieren zu können, wurde die oben beschriebene Exportschnittstelle realisiert.

Die Werkzeugumgebung CTE/AdaTEST wurde an mehreren praktischen Beispielen aus der Luft- und Raumfahrt sowie an internen Applikationen der Daimler-Benz-Forschung erfolgversprechend erprobt.

Für den funktionsorientierten Test hat sich dieser systematische Ansatz in Verbindung mit einer möglichst weitreichenden Werkzeugunterstützung sowohl als qualitäts- als auch als effizienzsteigernd erwiesen. Der Aspekt des Zeitverhaltens wird bei dem Vorgehen jedoch nur ungenügend betrachtet. Zur Prüfung von Zeitanforderungen müssen zusätzlich andere Verfahren eingesetzt werden.

2.3 Vorgehen zum Test des Zeitverhaltens mit evolutionären Algorithmen

Zusätzlich zur Prüfung der Funktion muß häufig das Zeitverhalten des Systems getestet werden, um die Einhaltung minimaler bzw. maximaler Laufzeiten sicherzustellen. In der Praxis werden in der Regel heuristische Schätzungen benutzt oder statische Analysen des Programmcodes vorgenommen. Eine exakte Bestimmung der maximal oder minimal erreichbaren Ausführungszeit für ein Testobjekt ist im allgemeinen nicht möglich. Für eine möglichst genaue Abschätzung von längsten bzw. kürzesten Ausführungszeiten wurde bei der Daimler-Benz-Forschung ein Ansatz entwickelt, der auf der Idee der evolutionären Optimierung aufsetzt [8].

Evolutionäre Algorithmen bilden Prozesse der natürlichen Genetik und der biologischen Evolution nach und machen sie für technische Optimierungsanwendungen nutzbar. In einem iterativen Verfahren werden ausgehend von einer Initialpopulation solange neue Populationen gebildet, bis eine optimale Problemlösung gefunden wurde oder ein zuvor definiertes Abbruchkriterium erreicht wurde. Eine Population enthält jeweils eine Menge unterschiedlicher Individuen. Für die Initialpopulation erfolgt die Individuenauswahl im allgemeinen zufällig. Basierend auf den Individuen der aktuellen Generation werden durch Anwendung genetischer Operatoren, wie Rekombination und Selektion, neue Individuen und damit eine nachfolgende Kindgeneration erzeugt (Abbildung 4). Von welchen Individuen Informationen in die Folgegeneration übernommen werden wird, in Abhängigkeit von den gewählten genetischen Operatoren und nach dem Prinzip *survival of the fittest* entschieden.

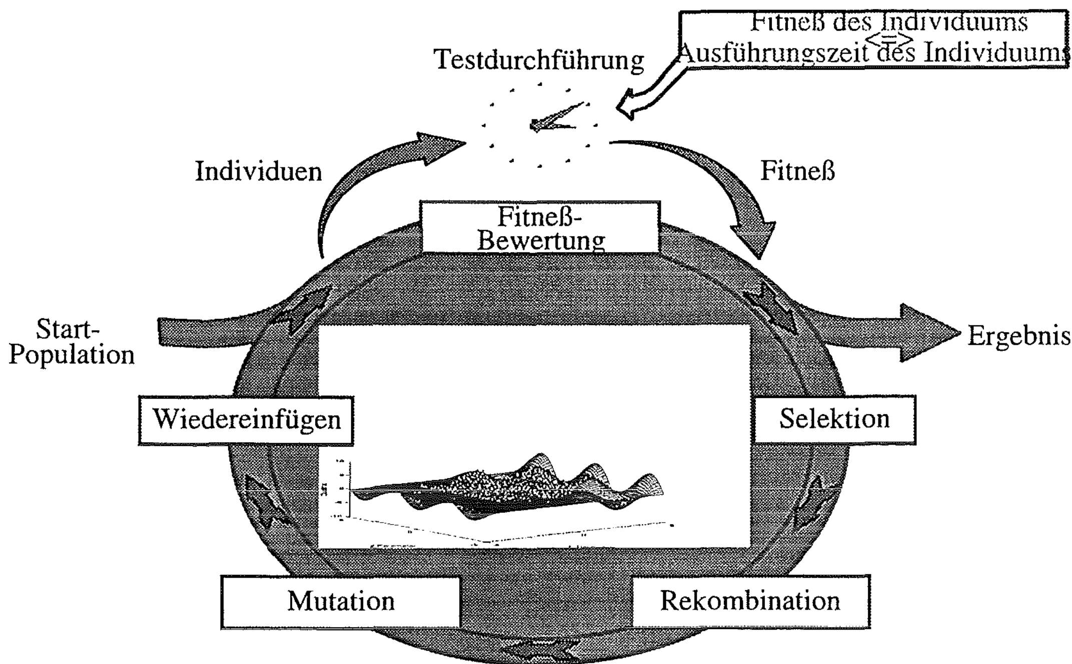


Abbildung 4 Prüfung des Zeitverhaltens mit evolutionären Algorithmen

Die Idee der Verwendung evolutionärer Algorithmen für die Prüfung des Zeitverhaltens besteht darin, daß die Suche nach längsten und kürzesten Ausführungszeiten als Optimierungsproblem betrachtet wird. Abbildung 4 zeigt den Ablauf einer Optimierung für den Test der Zeitaspekte. Bei einem Optimierungslauf werden die Eingangsparameter des Testobjekts unter Anwendung der genetischen Operatoren variiert. Mit jedem Datensatz wird das Testobjekt ausgeführt, und die benötigte Laufzeit wird gemessen. Die Ausführungszeiten spiegeln die Fitness der Testdatensätze und somit ihre Chance zur Informationsweitergabe in die Kindgeneration wider. Ermittelt man im Ergebnis der evolutionären Optimierung Laufzeiten, die nicht in den laut Spezifikation vorgegebenen Grenzen liegen, so verhält sich das Testobjekt bezüglich der Zeitanforderungen fehlerhaft.

Für das in Kapitel 2.2.1 eingeführte Beispiel *is_line_covered_by_rectangle* wurde nach einem Optimierungslauf mit 600 Ausführungen des Testobjektes (12 Generationen à 50 Individuen) eine maximale Laufzeit von 526 Prozessor-Zyklen auf einer Sparc ULTRA ermittelt.

Weitere Experimente mit komplexeren Beispielen und ein Vergleich mit Ergebnissen von statischen Analysen haben gezeigt, daß die mit den evolutionären Algorithmen ermittelten Werte gute Abschätzungen der extremen Laufzeiten ermöglichen [5]. In einigen Fällen, wie beim Sortieren einer Liste mit dem Bubble Sort-Verfahren, werden die extremen Laufzeiten durch sehr spezielle Eingabekonstellationen hervorgerufen. Bei derartigen Anwendungen sind unter Umständen sehr viele Optimierungsgenerationen erforderlich, um zu diesen Werte zu gelangen. Häufig wurden diese Eingabedaten bereits für den Test des funktionalen Verhaltens verwendet. Aus diesem Grund wurde eine Teststrategie entwickelt, die den systematischen Test und die evolutionäre Optimierung zur Prüfung des Zeitverhaltens kombiniert.

2.4 Teststrategie - Kombination von funktionaler und temporaler Prüfung

Die Teststrategie kombiniert die Ansätze des systematischen Tests und der temporalen Prüfung mit dem Ziel, deren Vorteile zu verbinden. Der systematische Test wird in jedem Fall zur Prüfung des funktionalen Verhaltens durchgeführt. Die im Verlauf der Testfallermittlung und Testdatenbestimmung angefallenen Daten enthalten testobjektspezifisches Wissen aus der Spezifikation. Kern der Teststrategie ist es, dieses Wissen für die evolutionäre Optimierung nutzbar zu machen.

Spezielle Eingabesituationen, die beim systematischen Test ermittelt werden, können mit Hilfe evolutionärer Algorithmen unter Umständen erst nach mehreren Iterationen erzeugt werden. Deshalb wird im Rahmen der Teststrategie das vorhandene Wissen aus dem systematischen Test verwendet, um dem evolutionären Algorithmus spezielle Wertekombinationen in der Initialpopulation zur Verfügung zu stellen. Im Gegensatz zu einer zufällig erzeugten Anfangspopulation fließt so Wissen über das System in die Suche nach den extremen Ausführungszeiten ein. In Experimenten hat sich gezeigt, daß dadurch eine Stabilisierung und teilweise sogar eine Beschleunigung der Suche erreicht werden kann. Abbildung 5 veranschaulicht die Teststrategie.

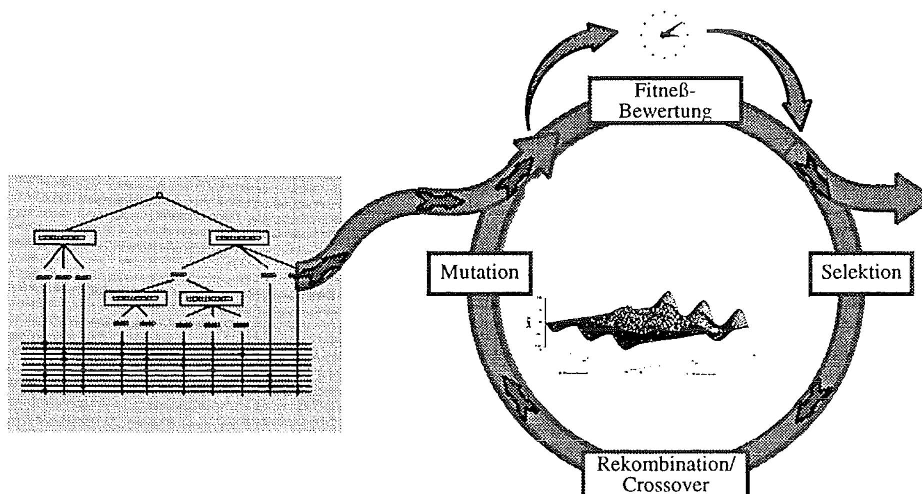


Abbildung 5 Teststrategie

Für die Prüfung des Beispiels *is_line_covered_by_rectangle* wurden die Testdaten, die den 49 definierten Testfällen zugeordnet sind, als Startpopulation verwendet. Die mit 526 Zyklen längste Ausführungszeit wurde mit dieser Teststrategie bereits nach 12 Iterationen ermittelt.

2.5 Zusammenfassung und Ausblick

Durch die beschriebene Kombination von verschiedenen, sich ergänzenden Ansätzen für den funktionalen und den Echtzeittest sowie die Nutzung entsprechender Werkzeuge steht eine wirkungsvolle Unterstützung für den umfassenden dynamischen Unit-Test von Ada-Programmen zur Verfügung.

Ziel der Arbeiten zum funktionalen Test war die Integration des Testsystems AdaTEST mit dem CTE, um eine möglichst weitgehende, durchgängige Rechnerunterstützung für den funktionalen Test von Ada-Programmen auf der Basis etablierter Testwerkzeuge zu erreichen. Erste Erprobungen wie das dargestellte Beispiel haben die Vorteile dieser Werkzeugverbindung für eine systematische Prüfung aufgezeigt. Außerdem kann die zusammenhängende Dokumentation von Tests und den zugehörigen Testfallbeschreibungen im Rahmen von Traceability-Anforderungen an den Entwicklungsprozeß sehr nützlich sein. Ein weiterer Vorteil der beschriebenen Realisierung zur Anbindung von CTE und AdaTEST liegt in der flexiblen Gestaltung der CTE-Exportschnittstelle. Auf Basis dieses Konzeptes kann die Integration mit anderen Werkzeugen einfach durch eine Variation einer Beschreibungsdatei erfolgen.

Der Ansatz zur temporalen Prüfung mittels evolutionärer Algorithmen ist eine geeignete Möglichkeit zum Test von Systemen mit Realzeit-Anforderungen. Dieses Verfahren läßt sich unabhängig von der Komplexität des Systems anwenden und kann vollständig automatisch ablaufen. Die praktischen Anwendungen des Vorgehens haben zusätzliche Fragestellungen aufgedeckt. In zukünftigen Arbeiten wird daher dieser Ansatz weiter ergänzt und verfeinert. Außerdem sind weitere Eprobungen der Teststrategie erforderlich.

2.6 Literatur

[1] *AdaTEST Script Generation Version 1.0 User Guide & Reference Manual*, IPL Information Processing Limited, 1995.

[2] Grimm K.: *Systematisches Testen von Software – Eine neue Methode und eine effektive Teststrategie*, GMD-Bericht Nr. 251, R. Oldenbourg Verlag, München/Wien, 1995.

[3] Grochtmann M., Grimm K.: *Classification Trees for Partition Testing*, Software Testing, Verification and Reliability, Vol 3, No 2, S.63-82.

[4] Grochtmann M., Wegener J.: *Test Case Design Using Classification Trees and the Classification-Tree Editor CTE,* Proceedings of Quality Week '95, San Francisco, Juni 1995.

[5] Müller, F., Wegener, J.: *A Comparison of Static Analysis and Evolutionary Testing for the Verification of Timing Constraints,* To appear in Proceedings of the 4th IEEE Real-Time Technology and Application Symposium (RTAS'98), Denver, Colorado, USA, Juni 1998.

[6] Wegener J., Fey I.: *Systematic Testing of Ada Programs,* Reliable Software Technologies – Ada Europe '97 Proceedings, London, Juni 1997.

[7] Wegener J., Grimm, K., Grochtmann, M., Sthamer, H.–H., and Jones, B.F.: *Systematic Testing of Real–Time Systems*, Proceedings of Fourth European International Conference on Software Testing, Analysis & Review EuroSTAR '96, Amsterdam, Dezember 1996.

[8] Wegener J., Sthamer, H.–H., Jones, B.F. and Eyres, D.E.: *Testing Real–Time Systems Using Genetic Algorithms*, Software Quality Journal. Vol. 6, No. 2, pp. 127 - 135, 1997.

Einsatz von Ada 95 in einem Forschungsprojekt: Erfahrungen aus Software-Engineering-Sicht

3 Einsatz von Ada 95 in einem Forschungsprojekt: Erfahrungen aus Software-Engineering-Sicht

Stefan Krauß, Anke Drappa
Abteilung Software Engineering, Universität Stuttgart
Breitwiesenstraße 20 - 22, 70565 Stuttgart
Stefan.Krauss@informatik.uni-stuttgart.de

Zusammenfassung

In diesem Artikel berichten wir über unsere Erfahrungen beim Einsatz von Ada 95 im Forschungsprojekt SESAM-2. Es wurde ein Simulator für Software-Entwicklungsprojekte realisiert, der der Ausbildung von Projektleitern dient. Die Erwartungen hinsichtlich der Qualität des Produkts konnten durch den Einsatz von Ada 95 erfüllt werden. Es sind jedoch einige Realisierungsaspekte von der Sprache nicht oder nur unzureichend unterstützt worden. Der Entwicklungsprozeß gestaltete sich hingegen durch die Verwendung von Ada 95 äußerst transparent und ermöglichte eine gut koordinierte Projektabwicklung.

3.1 Einführung

SESAM ist ein Ausbildungs- und Forschungswerkzeug. Es simuliert ein komplettes Software-Entwicklungsprojekt. Nur der Projektleiter wird nicht simuliert; diese Rolle wird vom Bediener, dem „Spieler“, eingenommen [Melchisedech et al. 1996]. Der Spieler wird durch Mitteilungen des SESAM-Systems mit Informationen über ein Software-Projekt versehen. Er kann den Verlauf dieses Projekts in ähnlicher Weise beeinflussen wie bei realen Projekten der Projektleiter [Drappa et al. 1995]. Der Spieler lernt wie bei vielen Lernspielen, z.B. Ökolopoly von F. Vester [Vester 1987], somit aus der Erfahrung.

Der SESAM-Simulator führt Modelle aus, die Effekte in der Software-Entwicklung beschreiben [Schneider 1994]. Die Modelle bestehen aus drei geschichteten Bestandteilen: Das attributierte Entity-Relationship-Schema führt die Gegenstände (Entitäten) und ihre möglichen Beziehungen ein. Die Anfangssituation ist eine spezielle Ausprägung dieses Schemas. Die Änderungen der Situation werden durch Regeln beschrieben. Allmähliche Änderungen der Attributwerte sind nach dem Konzept von „System Dynamics“ definiert [Forrester 1971]; sprunghafte Änderungen (Einführung oder Beseitigung von Objekten und Beziehungen) sind durch Graph-Grammatik-Produktionen vorgegeben [Göttler 1987].

Die Idee zum Projekt SESAM (Software-Engineering-Simulation durch animierte Modelle) entstand kurz nach der Gründung der Abteilung Software Engineering am Institut für Informatik der Universität Stuttgart im Oktober 1988 [Ludewig 1989]. Die ersten Arbeiten begannen 1990 und führten zu einem Pilotsystem. Die Evaluation des 1994 fertiggestellten Pilotsystems SESAM-1 führte zu einigen konzeptionellen Erweiterungen, die eine Reimplementierung erforderlich machten: SESAM-2.

Die wichtigsten technischen Anforderungen an das neue System lassen sich wie folgt zusammenfassen:

- Der Simulationsmechanismus auf der Basis von Graph-Grammatiken benötigt viel Rechenzeit. Damit auch größere Modelle ausführbar sind, muß die Neuimplementierung sehr effizient arbeiten.
- Das neue System SESAM-2 wird für die nächsten Jahre die Basis unserer Arbeiten bilden und noch einige Änderungen und Erweiterungen erfahren. Die Anforderungen an Wartbarkeit und Erweiterbarkeit sind sehr hoch. Die Unterstützung eines modularen Programmaufbaus ist daher nicht nur für die Aufteilung der Entwurfs- und Implementierungsarbeiten notwendig.
- Es ist geplant, das Simulationssystem auch außer Haus einzusetzen und an Interessierte weiterzugeben. Auf Dauer möchten wir daher die gängigsten Betriebssysteme und Hardware-Plattformen unterstützen, womit wir auf eine sehr portable Implementierung angewiesen sind.
- Als Abteilung Software Engineering war es uns natürlich ein wichtiges Anliegen, daß unsere Studenten und wir selbst die Entwicklung von SESAM-2 nach Maßgabe des Software Engineerings durchführen können.

3.2 Ada im Entwicklungsprozeß

3.2.1 Entscheidung für Ada 95

Nach den oben genannten Anforderungen wählten wir Ada 95 unter mehreren Alternativen aus. Zunächst sollte nur der Simulator in Ada 95 implementiert werden, zur Steuerung des Simulators und für die Benutzungsoberfläche wird die Skript-Sprache Tcl/Tk verwendet [Ousterhout 1995]. Die Gründe für die Wahl von Ada 95 waren:

- Ada 95 unterstützt alle Programmierkonzepte, die für eine strukturierte Entwicklung wartbarer und sicherer Programme aus Sicht des Software Engineerings notwendig sind (z.B. die strenge, statisch prüfbare Typisierung).
- Das Programm kann in Ada 95 durch hierarchisch angeordnete Pakete mit eigenständiger Schnittstellendefinition gegliedert werden. Auf diese Weise ist es möglich, auf mehreren Ebenen Teilsysteme zu bilden und die Schnittstellen bereits im Entwurf festzulegen.
- Optimierende Compiler können Ada-Code in schnellen Maschinencode übersetzen. So ist es möglich, eine größtmögliche Laufzeiteffizienz zu erreichen.
- Ada 95 ist gut spezifiziert und international standardisiert [ISO 1995]. Die Sprache bildet damit eine stabile Arbeitsgrundlage.
- Bei vielen Studenten und den Mitarbeitern der Abteilung waren Erfahrungen mit Ada 83 oder Ada 95 vorhanden. Der Einarbeitungsaufwand konnte so, auch für studentische Hilfskräfte, gering gehalten werden.
- Mit dem kostenlos, für viele Betriebssysteme erhältlichen Compiler GNAT (Gnu Ada Translator) ist es möglich, SESAM-2 ohne große Zusatzkosten für verschiedene Betriebssysteme anzubieten.

Noch ein paar Worte zur Konkurrenz: Gegen Java sprach die schnelle Fortentwicklung der Sprache, die fehlende Standardisierung und die mangelnde Ausführungs-

geschwindigkeit des Byte-Code-Interpreters, der Java Virtual Machine. C++ bot wegen geringer Typsicherheit und der insgesamt schlechten Fehlerprüfung zur Laufzeit ebenfalls keine Alternative. Eiffel ist nicht sehr verbreitet und in der Abteilung gab es keine konkreten Erfahrungen. Außerdem fehlt es an einem frei erhältlichen Compiler, der viele Betriebssysteme abdeckt. Smalltalk hatte sich bereits im Pilotprojekt SESAM-1 in den Bereichen Laufzeiteffizienz und Wartbarkeit als unzureichend erwiesen.

3.2.2 Projektorganisation

Die Arbeitsgruppe bestand aus fünf wissenschaftlichen Mitarbeitern der Abteilung Software Engineering. Alle Mitarbeiter waren mit SESAM vertraut und hatten bereits Erfahrungen mit Ada 83 oder Ada 95. Studenten wurden für Teilaufgaben hinzugezogen; sie arbeiteten im Rahmen ihrer Diplomarbeiten oder als studentische Hilfskräfte am Projekt SESAM-2 mit.

Der Test wurde zusammen mit einer Programmiererin der Abteilung durchgeführt. Die Fehlersuche und die Beseitigung der Fehler erfolgte aber durch die Personen, die die betreffenden Teilsysteme entworfen und programmiert hatten.

Da SESAM-2 als Forschungsprojekt in einem universitären Umfeld entstand, wurden keine festen Zeitpläne und harten Fertigstellungstermine vorgegeben. Deshalb wurde das Projekt nicht vollständig am Anfang geplant, sondern phasenweise während der gesamten Projektlaufzeit.

3.2.3 Projektverlauf und Ergebnisse

Tabelle 1 faßt den zeitlichen Verlauf, den Aufwand und die Ergebnisse des Projekts zusammen. In einem ersten Schritt wurden im Rahmen von Diplomarbeiten die Spezifikation des SESAM-2-Systems und ein einfacher Prototyp erstellt. Die Spezifikation wurde durch ein ausführliches Begriffslexikon komplettiert.

Ein einziger Mitarbeiter erstellte in der zweiten Jahreshälfte 1996 den Systementwurf. Er identifizierte acht Teilsysteme und definierte die Schnittstellen zwischen diesen Teilsystemen. Während der gesamten Feinentwurfsphase und der Implementierung stand dieser Mitarbeiter als Entwurfskoordinator ständig zur Verfügung. Da er alle Entwickler aus der Perspektive des Gesamtentwurfs beraten konnte und alle Änderungen zwischen den Teilsystemen abstimmte, trug diese Rolle entscheidend zum Projekterfolg bei.

Die Teilsysteme wurden dann auf die Mitarbeiter und studentischen Hilfskräfte aufgeteilt. Die Erstellung der Feinentwürfe, die Implementierung der Teilsysteme und der Modultest lagen so in der Verantwortung der einzelnen Entwickler. Änderungen an den Teilsystemschnittstellen durften nur in Absprache mit dem Entwurfskoordinator und allen betroffenen Teilsystementwicklern vorgenommen werden.

Tätigkeit	Mitarbeiter[1]	Zeitraum[2]	Aufwand[3]	Ergebnis[4]	
Spezifikation	1 WM 1 DA	43/1995 - 10/1996	70 MT	76 Seiten 6.231 LOC	Spezifikationsdokument Prototyp
Definition der Begriffe	1 WM	27/1996 - 39/1996	26 MT	14 Seiten	Begriffslexikon (128 Begriffe)
Systementwurf	1 WM	32/1996 - 51/1996	39 MT	40 Seiten 717 LOC 2.063 Zeilen	Dokument Systementwurf Spezifikation (8 Teilsysteme) ergänzender Kommentar
Vorbereitung der Arbeitsumgebung	2 WM	18/1996 - 03/1997	77 MT	98 Seiten	5 Richtlinien
Feinentwurf	5 WM 2 SH	03/1997 - 16/1997	57 MT	78 Seiten	5 Feinentwürfe (62 Schnittstellendef.)
Entwurfs-koordination	1 WM	03/1997 - 19/1997	17 MT		Abgleich von Änderungen zw. Teilsystemen
Implementierung	5 WM 2 SH	07/1997 - 18/1997	86 MT	18.320 LOC 4.374 LOC 2.060 LOC	74 Ada-Packages generierter Ada-Code aus AYacc/AFlex-Code
Modultest	5 WM 2 SH	13/1997 - 23/1997	61 MT	9.149 LOC	zusätzlicher Testcode
Integration	1 WM	24/1997 - 25/1997	8 MT		lauffähiges Gesamtsystem
Systemtest	1 WM 1 PR	40/1997 - 09/1998	21 MT	31 9	festgestellte Fehler Änderungsvorschläge
Fehlersuche und -korrektur	5 WM	40/1997 - 09/1998	20 MT		fehlerarmes Gesamtsystem
Summe	8 Pers.	43/1995 - 09/1998	482 MT	306 Seiten 36.477 LOC	Dokumentation Programmcode

Tabelle 1: Der Verlauf des SESAM-2-Projekts

Der Systemtest wurde im wesentlichen von der Programmiererin der Abteilung ausgeführt, die Fehlersuche und -korrektur wurde hingegen von den ursprünglichen Entwicklern der betroffenen Teilsysteme durchgeführt. Insgesamt wurden bisher 40 Mängel festgestellt, die relativ schnell behoben waren. Ein Teil der Mängel war allerdings auf kleinere konzeptionelle Versäumnisse zurückzuführen und nicht der eigentlichen Implementierung anzulasten.

Bis zum Jahresende 1996 wurde mit 77 Mitarbeitertagen ein recht hoher Aufwand in die Vorbereitung der Arbeitsumgebung gesteckt. Dazu gehörte die Installation und Konfiguration des Compilers und die Bereitstellung der Konfigurationsverwaltung.

[1] WM: wissenschaftliche/r Mitarbeiter/in; DA: Diplomand; SH: stud. Hilfskraft; PR: Programmiererin
[2] Angaben in Kalenderwochen
[3] Angaben in Mitarbeitertagen (1 MT = 8 Arbeitsstunden)
[4] LOC: Programmzeilen, die nicht leer sind und nicht ausschließlich Kommentartext enthalten

Außerdem wurden eine Anleitung für die Entwicklungsumgebung geschrieben und Richtlinien verfaßt, auch eine Programmierrichtlinie für Ada 95.

3.3 Die Rolle von Ada 95 im Entwurf

Die Entscheidung für Ada 95 beeinflußte bereits den Entwurf. Die wesentlichen Aspekte werden in diesem Abschnitt diskutiert.

3.3.1 Ada 95 als Entwurfsbeschreibungssprache

Ada 95 konnte bereits im Entwurf eingesetzt werden. Mit Hilfe der Spezifikationspakete wurden die Schnittstellen der Teilsysteme exakt spezifiziert. In einer zweiten Entwurfsphase wurden die Teilsysteme mit Hilfe des hierarchischen Paketkonzepts weiter verfeinert. So konnte mit Ada 95 ein Entwurf auf zwei unterschiedlichen Ebenen realisiert werden. Das Schnittstellengerüst war in jeder Phase übersetzbar, syntaktisch eindeutig und konnte ohne Veränderung in die Implementierung übernommen werden. Nur die als privat markierten Teile der Schnittstellenpakete mußten später vom Implementierer ergänzt werden.

```
private
      type ADT_Elem is record
                          null;
                       end record;
      type ADT is access ADT_Elem;
```

Beispiel 1: Unvollständige Definition im Spezifikationspaket

Während des Entwurfs wurden die privaten Definitionen nur unvollständig, aber syntaktisch korrekt in die Spezifikationspakete eingesetzt. Dies wurde in der Regel durch leere Records erreicht (Beispiel 1), die erst vom Implementierer durch geeignete Definitionen ersetzt wurden. Es hat sich als weniger fehleranfällig erwiesen, wenn die Struktur der privaten Typen bereits im Entwurf festgelegt wurde. Implementierer und Verwender eines Pakets sollten wissen, ob es sich bei einem Datentyp um einen Zeiger handelt. Bei Zuweisungen macht es zum Beispiel einen Unterschied, ob es sich bei den beteiligten Objekten um Zeiger oder Werte handelt (Alias-Problematik, siehe Abschnitt 4.3).

Der Schnittstellenentwurf mit Ada 95 hat allerdings auch Grenzen. Spezifikationspakete beschreiben die statische Sicht der Teilsystemschnittstelle. Die Beschreibung dynamischer Zusammenhänge und die Semantik der Teilsysteme müssen in einem separaten Entwurfsdokument erfolgen. Verständliche Bezeichner und der Einsatz passender Typen unterstützen zwar ein intuitives Verständnis, ein erklärendes Entwurfsdokument ersetzt die Verwendung von Ada 95 als Entwurfssprache aber nicht.

3.3.2 Mischung von objektorientierten und prozeduralen Elementen

SESAM-2 wurde nicht als „reines“ objektorientiertes System entworfen und implementiert. Ada 95 enthält aber im Gegensatz zum Vorgänger Ada 83 einige Mechanismen aus der objektorientierten Welt, die auch in hauptsächlich prozedural programmierten Systemen eingesetzt werden können. So sind zum Beispiel mit Hilfe von Tagged Types und der Vererbung voneinander abgeleitete Datenstrukturen elegant modellierbar (Datentypen und Spezialisierungen hiervon); die Auswahl der zugehörigen

Bearbeitungsroutine kann dank dynamischer Bindung automatisch erfolgen [Barnes 1995].

```
package value_manager is
      type Value_Type is abstract tagged private;
      type Value is access all Value_Type'Class;

private package value_manager.date_value is
      type Date_Value_Type is tagged private;
    private
      type Date_Value_Type is new Value_Type with ...
```

Beispiel 2: Definition des Typs Value und eines erweiterten Typen

Als Beispiel sei der Typ `Value` genannt, der in SESAM-2-Modellen Attributwerte beschreibt (siehe Beispiel 2). Der Typ `Value` umfaßt eigentlich mehrere Typen, die für die Grundtypen der SESAM-2-Attributwerte stehen (z.B. `Integer_Value` oder `Date_Value`). Jeder Grundtyp ist als Erweiterung von `Value` definiert und in einem eigenen, privaten Kindpaket untergebracht. Das Schnittstellenpaket sorgt für die Weitergabe der Funktionsaufrufe an die Kindpakete.

Der Vorteil dieser Konstruktion besteht darin, daß an der Schnittstelle lediglich der Typ `Value` zu sehen ist und alle anderen Teilsysteme nur diesen verwenden können. Für die Trennung und korrekte Behandlung der verschiedenen `Value`-Typen ist allein das Teilsystem verantwortlich, in dem Value definiert wurde. Dort ist für jeden `Value`-Typ ein eigenes Kindpaket vorgesehen; dies erhöht den Überblick und erlaubt eine einfache Ergänzung um weitere `Value`-Typen.

3.3.3 Organisation von System- und Feinentwurf

Der Entwurf wurde in zwei Phasen aufgeteilt. In der ersten Phase wurde ein Systementwurf erstellt, der alle grundlegenden Mechanismen und Konzepte des Programms beschreibt. Die Teilsysteme wurden auf dieser Ebene in Form von Spezifikationspaketen ausgeführt. Die Syntax der Schnittstellen und die Typen der ausgetauschten Daten waren damit eindeutig definiert.

Im zweiten Entwurfsschritt wurden die Teilsysteme den einzelnen Entwicklern zugewiesen. Diesen oblag dann die Aufgabe, einen Feinentwurf für die Teilsysteme zu erstellen. Jeder Entwickler konnte relativ unabhängig im Rahmen des Systementwurfs die ihm zugewiesenen Teilsysteme verfeinern, mußte Änderungen an den Schnittstellen oder der grundsätzlichen Funktionsweise aber mit dem Entwurfskoordinator absprechen. Die Systembeschreibung wurde so durch eine große Anzahl weiterer Spezifikationspakete erweitert.

Als besonders nützlich erwies sich, daß bereits im Systementwurf ein konsistentes Typgerüst zusammen mit den Schnittstellen definiert wurde. Da die meisten Typen als abstrakte Datentypen nicht vollständig ausgeführt werden mußten, bedeutete dies auch keine Vorwegnahme von Implementierungsdetails. Die Schnittstellen wurden dadurch aber klarer und alle Entwickler der Teilsysteme konnten auf dem gleichen Typgerüst aufbauen. Änderungen hieran waren immer an den Entwurfskoordinator gebunden, der für die Konsistenz des Gesamtsystems sorgte.

3.4 Ada 95 in der Implementierung

3.4.1 Generische Pakete und Information Hiding

Generische Pakete erlauben es, ähnliche Probleme mit einer einzigen Implementierung zu lösen. So benötigt SESAM-2 Listen für die unterschiedlichsten Basistypen, die als Instanzen einer generischen Liste implementiert wurden.

Sehr häufig tritt der Fall auf, daß sowohl ein abstrakter Datentyp als auch eine Liste über diesen benötigt wird. Die Typdefinition und die Instanzierung des generischen Listenpakets muß im gleichen Spezifikationspaket erfolgen, wenn Prozeduren des abstrakten Datentyps eine Liste des Typs als Parameter erhalten (im Beispiel `Do_Something`). Unseren ersten Ansatz zeigt Beispiel 3.

```
   type ADT is private;
      package ADT_List is new Generic_List (ADT);
      procedure Do_Something (Element: in out ADT;
                              List: in     ADT_List);

private
      type ADT is ...
```

Beispiel 3: Implementierungsversuch für einen abstrakten Datentyp

Das gezeigte Code-Stück läßt sich aber nicht übersetzen. Der Grund hierfür ist, daß generische Pakete nur mit Typen instanziert werden können, deren Definition komplett ist. Die Implementierung des abstrakten Datentyps sollte aber nicht sichtbar sein und wird daher erst im nachfolgenden privaten Teil definiert. Zyklische Importe würden das Problem lösen, sind aber in Ada 95 ebenfalls nicht erlaubt (siehe auch Abschnitt 4.4).

Die einzige Möglichkeit, das geschilderte Problem zu umgehen, besteht darin, die Implementierung des abstrakten Datentyps zu veröffentlichen. Damit wird jedoch das Prinzip des Information Hidings verletzt.

```
   type ADT_Elem is private;
      type ADT      is access ADT_Elem;
      package ADT_List is new Generic_List (ADT);

private
      type ADT_Elem is ...
```

Beispiel 4: Die Freigabe des abstrakten Datentyps löst das Instanzierungsproblem

Die meisten abstrakten Datentypen, auf die dieses Problem zutraf, sollten sowieso als Zeiger auf einen privaten Elementtyp implementiert werden. Wie in Beispiel 4 gezeigt, besteht daher eine Lösungsmöglichkeit mit geringem Geheimhaltungsverlust.

3.4.2 Dynamische Speicherverwaltung

Ein Simulationssystem wie SESAM-2 benötigt sehr viel dynamisch allozierten Speicher. Die Vermeidung von Speicherlecks und von Zugriffen auf freigegebenen Speicher (dangling references) ist schwierig und fehlerträchtig; auch Ada 95 bringt hierbei keine Erleichterung. Ada 95 enthält wie viele andere Programmiersprachen eine vordefinierte Operation zur Allokation von Speicher, eine entsprechende Deallokationsoperation fehlt hingegen. Diese muß für jeden einzelnen Datentyp instanziert werden.

```
type Value is access all Value_Type'Class;
    type Int_Val_Type is new Value_Type with ... end;
-- Int_Val_Ptr wird nur fuer Deallokation benoetigt
    type Int_Val_Ptr is access all Int_Val_Type;
    procedure Free is new
      Ada.Unchecked_Deallocation (Int_Val_Type, Int_Val_Ptr);
```

Beispiel 5: Aufwendige Instanzierung der Deallokationsroutine

Zur Instanzierung der Deallokationsroutine wird außer dem Datentyp selbst ein entsprechender Zeigertyp benötigt. Ohne diesen kann die Routine nicht instanziert werden. In SESAM-2 benutzen wir für manche abstrakte Datentypen klassenweite Zeigertypen. Deshalb muß für viele abstrakte Datentypen ausschließlich zur Instanzierung der Deallokationsroutine ein neuer Zeigertyp eingeführt werden (Beispiel 5).

3.4.3 Parameter-Modi und Alias-Effekte

Wie oben bereits beschrieben, wurden viele abstrakte Datentypen als Zeiger implementiert, was nicht immer verborgen werden konnte. Für den Benutzer macht es unter Umständen auch einen Unterschied, ob ein Datentyp als Zeiger implementiert wird. Bei Zuweisungen wird dann nämlich eine Referenz auf das Objekt gespeichert und keine Kopie des Objekts.

```
function Equal (Left, Right: in ADT) return Boolean is
    begin
      -- nicht zulaessig:     Left := Right
      -- moegliche Zuweisung: Left.all := Right.all
      ...
    end Equal;
```

Beispiel 6: Parametermodus in schützt nur den Zeiger vor Veränderung

Die Verwendung von Zeigern wirft eine Unstimmigkeit auf. Der Parametermodus „in“ schützt nur den Zeiger selbst vor Veränderung, nicht aber das referenzierte Objekt. Eine Zuweisung wie in Beispiel 6 ist daher möglich, wird aber vom Paketbenutzer, der nur die Schnittstellendefinition kennt, nicht erwartet.

3.4.4 Zyklische Importe

In SESAM-2 ist die Situation nicht untypisch, daß Prozeduren in der Implementierung eines abstrakten Datentyps Parameter von anderen abstrakten Datentypen erwarten. Außerdem wurde aus Übersichtsgründen jeder abstrakte Datentyp in einem eigenen Paket untergebracht.

```
with ADT_Function; use ADT_Function;
    package ADT_Statement is
      type Statement is private;
      function Create_Function_Call (Fct: in Function; ...)
        return Statement;
      ...
    end ADT_Statement;
with ADT_Statement; use ADT_Statement;
    package ADT_Function is
      type Function is private;
      function Create_Function (Code: in Statement; ...)
        return Function;
      ...
    end ADT_Function;
```

Beispiel 7: Verbotene zyklische Abhängigkeiten

Diese Vorgehensweise stellt kein Problem dar, solange die Spezifikationspakete nicht zyklisch voneinander abhängen. Wechselseitige Abhängigkeiten sind aber nicht auszuschließen, wie das Beispiel 7 zeigt. In diesem Fall ist es nicht mehr möglich, die beiden abstrakten Datentypen in eigenen Paketen unterzubringen. Das geschilderte Problem führte in SESAM zu einigen sehr großen, unüberschaubaren Paketen.

3.5 Test und Debugging

Bereits während der Fehlersuche bekamen wir einen ersten Eindruck von der Wartbarkeit des entstandenen SESAM-2-Systems. Änderungen waren dank der klaren Struktur und der guten Kapselung leicht einzuarbeiten, während die Fehlersuche oft sehr mühsam war.

3.5.1 Werkzeugunterstützung

Die relativ geringe Verbreitung von Ada 95 kann man sehr deutlich an der Anzahl und am Preis von verfügbaren Entwicklungswerkzeugen spüren. Die Konzentration auf Ada 95 schränkte unsere Werkzeugauswahl bereits ein. Weitere Kriterien wie Betriebssystem, Preis, Verfügbarkeit einer Campus-Lizenz, etc. führten dazu, daß wir praktisch ohne Werkzeuge auskommen mußten.

Zur Fehlersuche verwendeten wir daher nur sehr einfache Mittel. In der Regel wurden in den Code Aufrufe für die Ausgabe von Debugging-Informationen eingesetzt. Die sonst sehr vorteilhafte Typsicherheit von Ada 95 erschwert den Entwicklern dieses Vorgehen aber, denn für jede Testausgabe müssen entsprechende Bibliotheken eingebunden und Typumwandlungen vorgesehen werden.

```
Do_It;
    -- -A- Put_Line ("Debug: Do_It ausgeführt");
    -- normaler Kommentar ohne Debug-Code
```

Beispiel 8: Spezielle Kommentarzeilen enthalten Debug-Code

Nach einiger Zeit wurde der Debug-Code mit speziellen Kommentaren versehen (siehe Beispiel 8) und im Code belassen. Mit einem einfachen Perl-Skript können die so gekennzeichneten Kommentare entfernt werden und der Debug-Code seine Wirkung entfalten. Der Kennbuchstabe zwischen den beiden Bindestrichen erlaubt außerdem das selektive Aktivieren von Debug-Code.

3.5.2 Debugcode in abstrakten Datentypen

Das Verfolgen komplexer Datenstrukturen ist selbst mit guten Debuggern (die uns nicht zur Verfügung standen) ein recht mühsames Unterfangen. Da ein Debugger die Kapselung des abstrakten Datentyps durchbricht, kann er den Inhalt eines entsprechenden Objekts nur ungenügend darstellen. Für den menschlichen Tester sind semantische Ausgaben über das Objekt viel nützlicher als die realisierungstechnischen Einblicke in die verwendeten Datenstrukturen.

Jeder abstrakte Datentyp sollte daher über zusätzliche Funktionen zur Fehlersuche verfügen. Eine solche Funktion könnte beispielsweise eine kurze textuelle Darstellung eines Objekts liefern. Wie oben beschrieben kann zusätzlicher, zunächst aber ein auskommentierter Debug-Code von diesen Funktionen Gebrauch machen. Leider haben wir versäumt, bereits im Entwurf solche Debugging-Funktionen einzuplanen.

3.6 Fazit

Unser wichtigstes Anliegen, die Erstellung einer qualitativ hochwertigen Software, konnte durch die Verwendung der Sprache Ada 95 erfüllt werden. Der resultierende Programmcode hat sich im Betrieb als zuverlässig, stabil und effizient erwiesen, obwohl wir noch keine speziellen Optimierungen vorgenommen haben. Änderungen konnten dank der guten Strukturierungsmöglichkeiten von Ada 95 schnell in den Code einfließen.

Die beschriebenen Probleme mit Ada 95 konnten durch einfache Gegenmaßnahmen umgangen werden. Meist betrafen die Änderungen nur die Sichtbarkeit der entsprechenden Datentypen, echte Einschränkungen gab es keine. Lediglich das Prinzip des Information Hidings konnte nicht immer vollständig durchgehalten werden.

Der relativ hohe Vorbereitungsaufwand machte sich insgesamt bezahlt, da auf diese Weise die Einarbeitungszeiten der Entwickler gering gehalten werden konnten. Trotzdem ist der Einarbeitungsaufwand in Ada 95 vergleichsweise hoch, da die Sprache sehr komplex ist und eine Vielzahl an Möglichkeiten bietet. Durch die Größe des Projekts spielte dieser Aufwand aber nur eine untergeordnete Rolle.

Besonders vorteilhaft konnte das hierarchische Paketsystem mit separaten Schnittstellendefinitionen eingesetzt werden. Der zweistufige Entwurf und die klare, durch die Programmiersprache unterstützte Trennung der Aufgabenbereiche trug wesentlich zum Erfolg des Projekts bei. Der Entwicklungsprozeß wurde sehr transparent und ermöglichte eine gut koordinierte Projektabwicklung.

3.7 Literatur

Barnes, J.: *Programming in Ada 95*. Addison-Wesley, Wokingham, 1995.

Drappa, A.; Deininger, M.; Ludewig, J.; Melchisedech, R.: *Modeling and Simulation of Software Projects*. Proceedings of the Twentieth Annual Software Engineering Workshop. 29. - 30. November 1995, Greenbelt, MD (USA), Seite 269 - 275, 1995.

Forrester, J. W.: *Principles of System.* 2. Ed., Wright-Allen Press, Cambridge, Massachusetts, 1971.

Göttler, H.: *Graphgrammatiken in der Softwaretechnik.* Informatik-Fachberichte, Springer, Berlin, 1987.

ISO: *International Standard ISO/IEC 8652-1995: Ada Reference Manual.* 1995.

Ludewig, J.: *Modelle der Software-Entwicklung – Abbilder oder Vorbilder?* Softwaretechnik-Trends (9), Nr. 3, Seite 1 - 12, 1989.

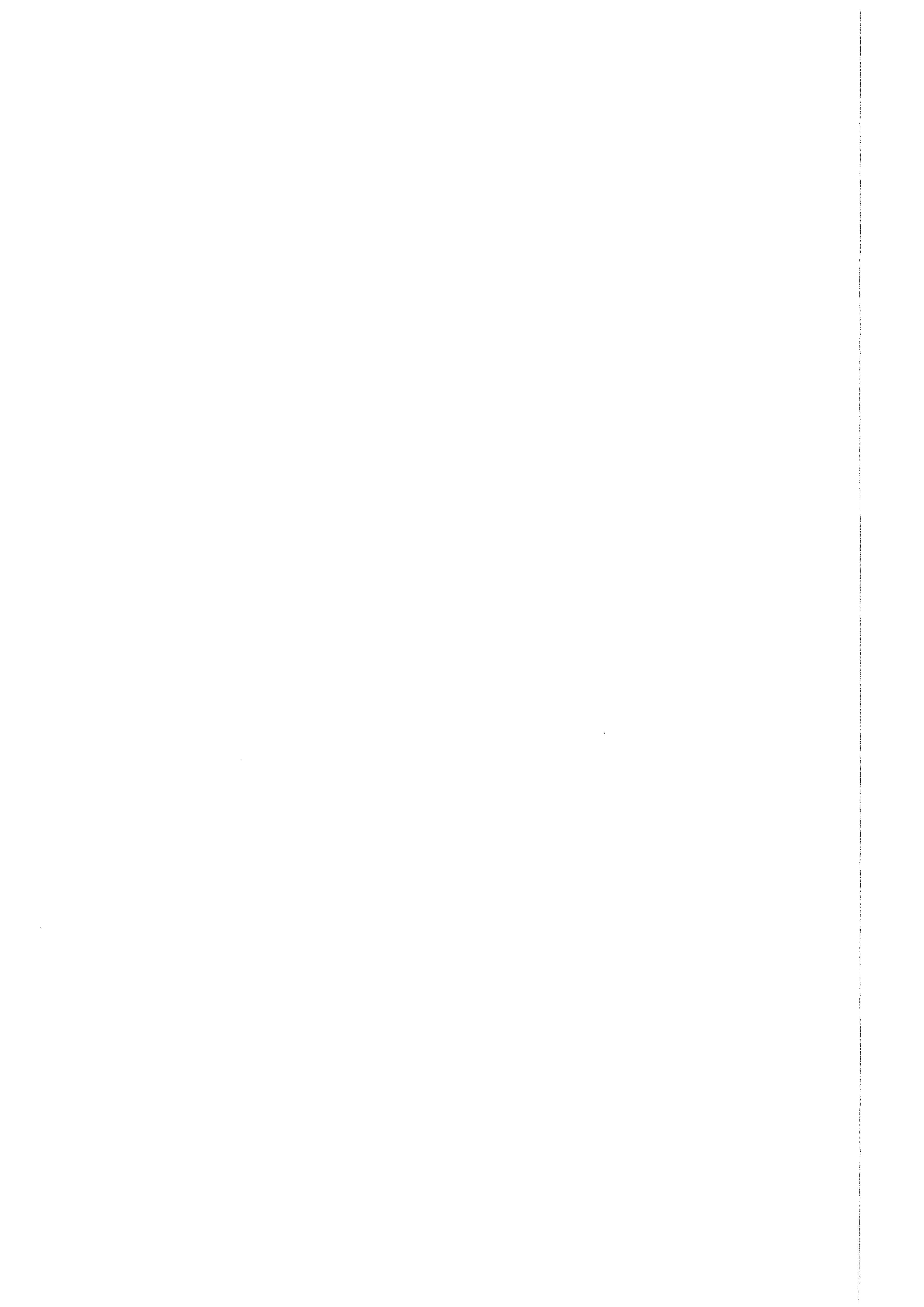
Melchisedech, R.; Deininger, M.; Drappa, A.; Hoff, H.; Krauß, S.; Li, J.; Ludewig, J.; Mandl-Striegnitz, P.: *SESAM – A Software Engineering Education Tool Based on Graph Grammars.* Bulletin of the European Association for Theoretical Computer Science, Nr. 58, Seite 198 - 221, 1996.

Ousterhout, J. K.: *Tcl und Tk.* Reihe Professional Computing, Addison-Wesley, Bonn, 1995.

Schneider, K.: *Ausführbare Modelle der Software-Entwicklung.* vdf Hochschulverlag, Zürich, 1994.

Vester, F.: *Ökolopoly – Ein kybernetisches Umweltspiel.* Otto Maier Verlag, Ravensburg, 1987.

Sicherheit mit Standard-Software – was kann Ada95 dazu beitragen



4 Sicherheit mit Standard-Software - Was kann Ada95 dazu beitragen

Karlotto Mangold

ATM Computer GmbH

78467 Konstanz

Zusammenfassung

Dieser Beitrag soll zeigen, welche Möglichkeiten in Ada95 - speziell im Annex H - vorgesehen sind, um sichere, das heißt zuverlässige Systeme für sicherheitskritische Anwendungen zu implementieren und diese Systeme gemäß den üblichen Prozeduren zu zertifizieren.

Summary:

This paper will show two aspects: The features which are available in Ada95 - especially in the specialized needs annex H - to implement trusted systems for safety-critical applications. How these systems can be certified according to existing procedures.

4.1 Einleitung

Bei einem Titel „Sicherheit mit Standard-Software“ denkt man zunächst an spezielle Produkte, die entweder der Sicherheit in besonderem Maße dienen oder die im Hinblick auf Sicherheitsanforderungen zertifiziert sind.

Um die Jahreswende 1996/97 wurde in den SoftwareTechnik-Trends [1] ein Aufsatz veröffentlicht, in dem die Autoren zeigen wollten, daß die Sprache C zur Implementierung sicherheitskritischer Systeme geeignet sei. Dabei wurde jedoch ignoriert, daß im einschlägigen internationalen Standard IEC-1508 [2] C bei den beiden höchsten Sicherheitsstufen SIL3 und SIL4 als „not recommended“ eingestuft wird, während ein Subset von Ada in allen 4 Stufen die Einstufung „highly recommended“ erhielt. Diese einseitige Lobeshymne auf C blieb denn auch nicht lange unwidersprochen, schon im nächsten Heft folgte eine Gegendarstellung [3]. Eine einfache Erklärung für die meines Erachtens irrigen Annahme über Ada in [1] findet sich im Informatik-Duden [4]. Dort wird unter dem Stichwort ADA (!) seit nunmehr 10 Jahren unrevidiert behauptet: *„Bekannte Informatik Fachleute warnen daher vor der Anwendung von ADA insbesondere in Bereichen, bei denen die Sicherheit im Vordergrund steht“.* Im Gegensatz zu den anonym bleibenden „Fachleuten“ des Dudens soll hier mit konkreten Details aus dem verabschiedeten ISO-Standards [8] gezeigt werden wie Ada und insbesondere Ada95 die Implementierung sicherheitskritischer Systeme unterstützt.

Generell gibt es bei bekannt sicherheitskritischen Systemen, wie Flugzeugen, Bahnen oder Kraftwerken seit langem Prozeduren, die einen Sicherheitsnachweis erbringen sollen. Mit zunehmendem Einfluß von Rechnern und damit auch von Software in diesen Systemen gibt es auch Verfahren und Standards, welche die Zuverlässigkeit von Software sicherstellen sollen. Beispielhaft seien [2], [5] und [6] genannt. Etwas verallgemeinernd kann gesagt werden, daß diese Standards dazu gedacht sind, existierende Prüfverfahren zu unterstützen und zu ergänzen, wobei davon ausgegangen wird, daß die implementierte Software langsam Funktionen übernimmt, die früher in Hardware realisiert wurden.

Bei der Definition von Ada95 wurde bereits in der Anforderungsspezifikation (Requirements) [7] ein ganzes Kapitel den sicherheitskritischen und vertrauenswürdigen Anwendungen gewidmet. Die dort genannten Anforderungen betreffen die Deterministik der Programm-Ausführung, die Zertifizierbarkeit von Programmen und die Erzwingung sicherheitskritischer Programmier-Praktiken.

Es muß hier ganz deutlich darauf hingewiesen werden, daß aus der Tatsache, daß in Ada95 ein spezieller Annex H für Safety und Security existiert, nicht geschlossen werden kann, daß die Sprache ohne diesen (optionalen) Anhang nicht für die Implementierung sicherheitskritischer Systeme geeignet wäre. Vom Sprachkonzept her ist Ada deutlich besser geeignet für die Implementierung solcher Systeme als manche andere Sprache. Insbesondere wird die häufig verwendete Sprache C in der einschlägigen Literatur [2] als kritisch eingestuft. Die Motivation für diesen Annex liegt vielmehr darin, daß bei der Definition von Ada95 bekannt war, daß sicherheitskritische Systeme spezielle, in der Vergangenheit bewährte Zertifizierungs-Prozeduren durchlaufen müssen, um für den jeweiligen Einsatz freigegeben zu werden. Diese eingespielten Prozeduren erfordern jedoch Unterstützung durch die jeweilige Implementierungssprache und deren Übersetzungssystem. Da Ada95 vom Einsatzbereich her embedded systems auch in sicherheitskritischen Anwendungen zum Ziel hat, wurde mit dem Annex H ein Ansatz geschaffen, solche Applikationen einfacher zertifizieren zu können.

4.2 Die Stellung des Annex H und die Rolle der HRG

Bekanntlich enthält das Ada95 Reference Manual [8] die Kapitel 1 bis 13 und die Anhänge A bis P (annexes). Dabei bilden die 13 Kapitel zusammen mit den Anhängen A, B und J die sogenannte *core language*. Die Anhänge C bis H sind die *specialized needs annexes* und unterstützen spezielle Anwendungsgebiete. Diese Anhänge sind für die Implementierungen normativ, die die entsprechenden Anhänge unterstützen. Der Annex H enthält unter der Überschrift "Safety and Security" Anforderungen an die Implementierung von Ada-Übersetzungssysteme, die zur Erzeugung sicherheitskritischer Systeme verwendet werden können. In diesem Beitrag soll dargestellt werden, welche Eigenschaften gefordert werden und welche Konsequenzen dies auf die Nutzung von Ada hat.

Der Annex H ist - wie alle *specialized needs annexes* - zwar eine Option, in dem Sinne, daß nicht jede Implementierung diesen Annex unterstützen muß, daß aber andererseits jede Implementierung, die vorgibt, diesen Annex zu unterstützen, an diesem Annex validiert wird. Damit besteht zumindest die Möglichkeit, eine gewisse, standardisierte Unterstützung zu erhalten.

Bei den Forderungen handelt es sich um die Zusicherung spezieller Eigenschaften, die Möglichkeit "sicherheitskritische" Sprachkonstrukte gezielt auszuschließen und zusätzliche Dokumentationsanforderungen, um eine eventuelle Zertifizierung zu erleichtern.

Zusätzlich zum Language Reference Manual [8], das als Standard für den Nutzer nicht immer leicht verständlich ist, gibt es die Ada95 Rationale [9], wo auch zum Annex H Hintergrund-Information dargestellt ist und mögliche Erläuterungen zum normativen Text des Standards gegeben werden. Leider wurde im Rahmen des Standardisierungsprozesses bei ISO/IEC JTC SC22 WG9 (der für die Ada-Standardisierung zuständigen ISO-Arbeitsgruppe) zugunsten der Einhaltung des Endtermins beim Annex H auf die Ausformulierung in derselben Tiefe verzichtet, wie dies bei der *core language* und den übrigen *specialized needs annexes* der Fall war. Um diese eventuelle Lücke zu schließen, gibt es als Untergruppe der WG9 die Annex H Rapporteur Group (HRG), die sich mit der genauen Spezifizierung des Annex H befaßt. Zur Zeit der Verabschiedung des Ada95 Standards hieß diese Gruppe noch VRG (Verification Rapporteur Group). Diese sehr aktive Gruppe unter der Leitung von Brian Wichman hat im Rahmen der ISO ein work item zu bearbeiten, das unter dem Titel „Guidance for the Use of Ada in High Integrity Systems“ [10] die Umsetzung des Annex H in praktische Regeln für die Zertifizierung von Systemen erzielen soll. In der entsprechenden Resolution [11] hat die WG9 konkrete Vorgaben für die Arbeit der HRG gemacht. Ende April 1998 hat die HRG ihren Bericht an die WG9 fertiggestellt und es kann voraussichtlich davon ausgegangen werden, daß die WG9 im Sommer dieses Jahres diesen Bericht verabschiedet und den zuständigen ISO-Gremien zur Standardisierung weiterleitet. Mit diesem Prozeß werden die Anforderungen an Ada für die Implementierung sicherheitskritischer Systeme ähnlich detailliert und präzise festgelegt, wie dies in den anderen Anhängen für spezielle Bedürfnisse (special needs annexes) des Language Reference Manuals [8] seit mehr als drei Jahren bereits der Fall ist.

4.3 Der Inhalt des Annex H

Aus Sicht der Validierung und Zertifizierung sicherheitskritischer Systeme lassen sich beim Einsatz von Programmiersprachen drei Problemgebiete identifizieren:

- Wie jede höhere Programmiersprache läßt Ada95 in der Sprach-Definition eine Reihe von Implementierungs-Details offen, deren Kenntnis für die Validierung und Zertifizierung eines Systems notwendig ist.

- Obwohl die Programmierung in der höheren Sprache erfolgt, ist es notwendig, die Validierung auf Objekt-Code-Ebene durchzuführen, da eine Validierung auf Sprach-Ebene die zusätzliche Verifizierung des Compilers erfordern würde, was bis heute praktisch unmöglich ist.

- Für viele sicherheitskritische Systeme ist die Mächtigkeit der heutigen Sprachen und Betriebssysteme zu komplex und zu unüberschaubar. Es muß deshalb die Möglichkeit geben, das Laufzeitsystem so zu konfigurieren, daß Funktionen, die in einem konkreten System unnötig sind, weggelassen werden können. Dadurch wird die Verifikation vereinfacht.

Als Antwort darauf stellt der Annex H folgende Anforderungen an ein zertifizierbares System:

- Trotz aller Abstraktions-Mechanismen einer höheren Sprache muß die Programm-Ausführung verständlich und nachvollziehbar sein (Keine Mystik!). In der *core language* von Ada95 wurde im Hinblick auf diese Forderung die undefinierte *erronious execution* aus Ada83 durch *bounded errors* - eine behandelbare Ausnahme - ersetzt.

- Der erzeugte Objekt-Code muß überprüfbar sein und der Bezug zum Quell-Code muß hergestellt werden können.

- Sprachkonstrukte, deren Verwendung den Nachweis der Korrektheit erschweren, sollen ausgeschlossen (verboten) werden können.

Im einzelnen werden im Annex H folgende Eigenschaften gefordert:

4.3.1 Das Pragma Normalize_Scalars

Das Pragma *Normalize_Scalars* dient zur definierten Initialisierung von Variablen und gilt für alle Variablen einer Partition. Als Initialwert soll, wenn möglich, ein Wert gewählt werden, der außerhalb des zulässigen Wertebereichs liegt und damit bei der Verwendung zur Laufzeit eine Ausnahme auslöst und behandelt werden kann. Damit wird zwar nicht verhindert, daß uninitialisierte Variablen fälschlicherweise in einem Programm verwendet werden, aber es wird sichergestellt, daß diese Variablen keine Zufallswerte enthalten, sondern stets reproduzierbar mit demselben Wert belegt sind. Mit dem *x'valid* Attribut kann die Gültigkeit eines solchen Wertes dann abgefragt werden, ohne daß eine Ausnahme ausgelöst wird. Dieses Attribut kann insbesondere auch dazu verwendet werden, um die aktuellen Werte von Variablen auf Gültigkeit zu prüfen, wenn diese durch *unchecked conversion*, eine *read*-Prozedur, mit Hilfe des pragmas *Import* oder durch Operationen mit unterdrückten Laufzeit-Prüfungen (*pragma suppress*) gesetzt wurden.

Ein anderer Ansatz zur Lösung dieses Problems wird im Ada-Leitfaden (AU 255) des BMVg [12] gewählt. Dort wird dem Programmierer vorgeschrieben, daß er bei der Deklaration einer Variablen diese initialisieren muß. Mit Hilfe eines Werkzeugs wird dann die Einhaltung dieser Vorschrift überprüft.

4.3.2 Dokumentation von Implementierungs-Entscheidungen

Hier sollen die Auswirkungen aller Situationen beschrieben werden, die gemäß Sprach-Definition zu bounded errors führen oder eine nicht spezifizierte Wirkung haben. Falls bestimmte Auswirkungen der fehlerhaften Programm-Ausführung eingeschränkt werden können, so sollen diese Einschränkungen dokumentiert werden. Diese Dokumentation kann sowohl unabhängig von einer konkreten Übersetzungseinheit oder Partition erfolgen, als auch als Teil eines konkreten Übersetzungsprotokolls.

Insbesondere ist die Parameter-Übergabe-Konvention, die Strategie der Speicherverwaltung in den Laufzeit-Routinen und die Art der Berechnung numerischer Ausdrücke zu beschreiben.

Falls die Parameter-Übergabe eine statische Entscheidung ist (z.B. immer *by reference* außer *entry*-Parameter, die stets *by copy* übergeben werden, so kann dies in der Compiler-Dokumentation beschrieben werden. Falls die Übergabe-Mechanismen von Aufruf zu Aufruf unterschiedlich sein können, muß dies im Compiler-Protokoll dokumentiert werden.

Die Kenntnis der Strategie der Speicherverwaltung ist aus drei Gründen wichtig. Erstens um zu vermeiden, daß beispielsweise in einer Schleife immer wieder Speicher allokiert und nur teilweise wieder freigegeben wird, was bei langer Laufzeit zu einem Speicherengpaß mit möglicherweise katastrophalen Folgen führen kann. In dieselbe Klasse von Problemen kann auch eine Speicherverwaltung führen, die den Speicher nicht nach Art eines Stacks verwaltet, sondern eine Speicher-Fragmentierung zuläßt.

Zweitens um zu vermeiden, daß durch Lücken in der Speicherverwaltung schützenswerte Information nach außen gelangen kann.

Drittens kann es in zeitkritischen Anwendungen effizienter sein, auf Benutzerebene Speicher zu allokieren und wieder freizugeben und so ein definiertes Zeitverhalten der Anwendung zu sichern.

Die Beschreibung der numerischen Berechnungen dient insbesondere der Beurteilung von Genauigkeiten und Zahlenbereichen.

4.3.3 Der Code-Review

Da davon ausgegangen werden muß, daß Compiler trotz aller Tests und Validierungsprozeduren fehlerhaft sind und bleiben werden, wird für sicherheitskritische Systeme eine Verifizierung auf Objekt-Code Ebene gefordert [6]. Der Code-Review und die Code-Validierung werden durch die Pragmas *Reviewable* und *Inspection_Point* unterstützt. Das Pragma *Reviewable* ist ein Konfigurations-Pragma und wirkt auf alle Übersetzungseinheiten in einer *Partition.* Es bewirkt, daß beim Übersetzen zusätzliche Information bereitgestellt wird. Damit erleichtert es die Analyse und den Review des Oject-Codes und unterstützt auch die Bestimmung von Ausführungszeiten und Speicherplatzanforderungen. Zu diesen Informationen gehört ein Objekt-Code-Listing mit den generierten Maschinenbefehlen, den jeweiligen Datenadressen und Verweisen auf das Quell-Programm auf Anweisungsebene. Außerdem sind die Stellen auszuweisen, an denen zur Laufzeit Tests durchgeführt werden oder implizit Laufzeit-Routinen aufgerufen werden. Bei jedem Zugriff auf skalare Objekte muß vermerkt werden, ob bekannt ist, daß diese Größe bereits initialisiert ist oder ob vermutet werden kann, daß sie noch nicht initialisiert ist. Um den erzeugten Code verständlich zu machen, muß auch die tatsächliche Register-Belegung und die Lebenszeit von Objekten dokumentiert werden. Dazu gehört auch eine Kennzeichnung, wie lange ein Objekt in einem Register gehalten wird. Für jedes Konstrukt ist auszuweisen, an welcher Stelle wie viel Speicher dynamisch im Laufzeit-Stack belegt und wieder freigegeben wird. Da die vom Compiler-Hersteller gelieferten Laufzeit-Routinen oder die vom Compiler implizit generierten Befehlsfolgen, wie Initialisierungs- oder Finalisierungs-Code, das System-Verhalten mit beeinflussen, sind diese ebenso zu dokumentieren, wie der vom Compiler aus der Quelle erzeugte Code. Um einen Code-Review mit geeigneten, compiler-unabhängigen Tools unterstützen zu können, ist die geforderte Information sowohl mensch- wie maschinen-lesbar bereitzustellen.

Das Pragma *Inspection_Point[(object_name {, object_name})]* ermöglicht an den entsprechenden Stellen bei der Programm-Ausführung die aktuellen Werte der spezifizierten Objekte zu kontrollieren, das heißt, die Objekte sind an dieser Stelle inspizierbar. Theoretisch bietet dieser Ansatz die Möglichkeit, eine mathematische Programm-Verifikation durchzuführen. Allerdings dürfte dies in der Regel daran scheitern, daß keine adäquate formale Beschreibung der Applikation vorliegt. Durch die Inspektionspunkte ist es möglich, (Zwischen-)Werte zu überprüfen und so Behauptungen (assertions) über den Programmablauf zu verifizieren. Dieses Pragma ist an jeder Stelle des Quellprogramms zulässig, wo eine Deklaration oder eine Anweisung erlaubt ist. Durch dieses Pragma wird ein sogenannter *inspection point* definiert. Ohne Parameter (Objektnamen) macht das Pragma alle Objekte im Sichtbarkeitsbereich des Pragmas inspizierbar, mit Parametern nur die genannten Objekte. An jedem Inspektionspunkt muß die Zuordnung zwischen dem inspizierbaren Objekt (Quellebene) und der zugehörigen Realisierung dokumentiert werden. Für die Implementierung und Optimierung ist jeder Inspektionspunkt wie ein lesender Zugriff auf alle inspizierbaren Objekte zu behandeln.

4.4 Sprach-Einschränkungen für sicherheitskritische Systeme

Eine bewährte Technik, sicherheitskritische Programme überprüfbar zu machen, besteht darin, auf unnötige Komplexität zu verzichten. In seiner Rede anläßlich der Verleihung des Turing-Preises formulierte C.A.R. Hoare dies 1980 so [13]: „*There are two ways of constructing software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies.*“ Die Betonung muß hier auf dem Attribut unnötig liegen. John Barnes [14] zieht hier den meines Erachtens kurzsichtigen Schluß. wenn er Ausnahmen in sicherheitskritischen Systemen mit folgender Begründung generell verbietet: „*The general philosophy is, that for critical programs it is not acceptable for anything to go wrong and so the world might as well end if it does*!“. Wenn beispielsweise ein Steuerprogramm als unendliche Schleife alle Aufgabe erledigen kann, so ist es sicher zweckmäßig auf Tasking in diesem Fall zu verzichten. Ein solches Programm ist dann natürlich auch viel einfacher zu überprüfen, als wenn alle Taskoperationen verifiziert werden müßten. Im Annex H bietet Ada95 die Möglichkeit, mit dem Pragma *restriction* einzelne Sprachkonstrukte zu verbieten, die in bestimmten Anwendungen als Sicherheitsrisiko betrachtet werden könnten. Durch diese Einschränkungen soll das Laufzeit-System verkleinert und damit der Nachweis einer korrekten Programm-Ausführung erleichtert werden.

Die möglichen Einschränkungen betreffen insbesondere das Tasking, die datenorientierte Synchronisation über protected objects, die Speicher-Verwaltung, die Behandlung von Ausnahmen, die Verfügbarkeit numerischer Datentypen. Daneben können einzelne Sprach-Elemente, wie unchecked conversion oder unchecked access oder kritische Eigenschaften, wie Rekursion oder Reentranz, ausgeschlossen werden. Mit der Einschränkung No-Delay wird das Delay-Statement und die Abhängigkeit vom Paket *Calendar* verboten. Sämtliche in Ada95 geforderten Einschränkungen finden sich im LRM [8] verteilt an zwei Stellen. Im Annex D (Real Time Systems) werden im Kapitel 7 die möglichen Einschränkungen im Task-Modell aufgelistet, während im Annex H.4 die übrigen Einschränkungen erläutert sind. Dabei handelt es sich um die minimal notwendigen Einschränkungen um standardkonform zu sein. Es ist dem Compilerhersteller unbenommen weitere Einschränkungsmöglichkeiten vorzusehen.

Wichtig ist jedoch, darauf hinzuweisen, daß bis auf ganz wenige, explizit genannte Ausnahmen das Pragma *restriction* auch auf das Laufzeit-System wirkt. Außerdem muß dokumentiert werden, wie sich ein Programm verhält, das mit der Restriktion *No-Exceptions* übersetzt wurde, wenn zur Laufzeit ein automatisch durchzuführender Test fehlschlägt. Generell sind Verstöße gegen spezifizierte Einschränkungen vom Compiler mit Fehlermeldungen, bzw. Warnungen zu kennzeichnen. Können solche Verstöße erst zur Laufzeit erkannt werden, so gilt der Programmlauf als fehlerhaft (*erroneous*) mit allen daraus resultierenden Konsequenzen. Es erscheint mir jedoch fraglich, ob diese Haltung für sicherheitskritische Systeme akzeptabel ist.

4.5 Alternative Ansätze

Einzelne Compiler- und Werkzeughersteller sehen seit längerem die Notwendigkeit, ihren Nutzern bei der Erstellung sicherheitskritischer Systeme Unterstützung zu bieten oder bei der Zertifizierung behilflich zu sein. Hier seien exemplarisch zwei Ansätze genannt: der Ansatz von Praxis Critical Systems Ltd mit SPARK und der von Thomson/AONIX mit dem Safety Critical Handbook. Beide Ansätze wurden schon für Ada83 entwickelt und sind inzwischen oder werden noch auf Ada95 angepaßt.

Der SPARK-Ansatz versucht Programme oder Programm-Teile formal zu verifizieren. Es erscheint mir jedoch fraglich, ob Echtzeit-Systeme überhaupt im strengen Sinne verifizierbar sind, da mir derzeit keine formale Beschreibung von Zeitbedingungen und parallelen Abläufen bekannt ist. Eine kritische Auseinandersetzung mit dem SPARK-Ansatz und seinen Auswirkungen auf ein fliegendes System gibt Roßkopf in [15]. Barnes [14] stellt mit seinem jüngst erschienenen Buch einen Zusammenhang zwischen SPARK und Ada95 her, wobei er jedoch selbst feststellt: „Although the changes to Ada (von Ada83 nach Ada95) were largely outside the subset on which SPARK is based, nevertheless some small changes to the core of Ada are quite fundamental and very relevant to SPARK“. Beim Vergleich der beiden Darstellungen muß jedoch berücksichtigt werden, daß zumindest unterschiedliche Versionen des Werkzeugs, wenn nicht unterschiedliche Werkzeuge mit demselben Namen betrachtet werden.

Ebenfalls die Verifizierung der Software eines fliegenden Systems war der Hintergrund für das Safety Critical Handbook [16] der Firma Thomson Software Products /AONIX. In diesem Handbuch und verwandten Veröffentlichungen [17] wird gezeigt, wie mit Ada83 die kritische Software von Boeings B777 zertifiziert werden konnten. Dabei wird recht praxisnah von den Anforderungen des DO178B [6] ausgegangen. Die hier gewählte Lösung liegt in einem geeigneten *Certifiable Small Run-Time* System (C-SMART), das zusammen mit der Applikations-Software zertifiziert werden kann. Hier wird versucht, möglichst umfangreiche Funktionalität zu bieten und in Erweiterung zu C-SMART ein T-SMART als Obermenge mit eingeschränkten Task-Operationen zu definieren und anzubieten.

Ein herstellerunabhängiger Ansatz die Voraussetzungen für die Implementierung zertifizierbarer Software zu schaffen, ging von dem internationalen Workshop über Echtzeit-Systeme (IRTW) aus. Dieser jährlich in Großbritannien veranstaltete Workshop hat 1997 bei seiner Sitzung in Ravenscar (North Yorkshire) das sogenannte RAVENSCAR- Profil [18] verabschiedet. Dabei wird RAVENSCAR als Akronym benutzt für Reliable Ada Verifiable Executive Needed for Schedulöing Critical Applications in Realtime. Es ist zu erwarten, daß dieser Vorschlag die weitere Entwicklung im Bereich

zerifizierbarer Ada-Software entscheidend bestimmt. Einerseits hat die Firma AONIX bereits im November 1997 die Implementierung eines Ravenscar konformen Systems angekündigt, andererseits wird die HRG dieses Profil in ihren für Ende April 1998 angekündigten Bericht einarbeiten.

4.6 Schlußfolgerungen

Insgesamt kann meines Erachtens festgestellt werden, daß Ada95 mit dem Annex H einen praktikablen Ansatz zur Erstellung von zertifizierbarer Software bietet. Der Ansatz von SPARK erscheint mir zu rigoros und birgt die Gefahr, daß kritische Teile nicht adäquat formuliert werden können und deshalb in einer Grauzone angesiedelt werden. Vor diesem Hintergrund stellt sich mir die Frage, ob nicht das SPARK-Konzept nur auf einen relativ einfachen Teil der sicherheitskritischen Systeme angewendet werden kann. Der AONIX-Ansatz dagegen geht mehr auf die Belange der Implementatoren ein und bietet ein abgestuftes Konzept für unterschiedliche Komplexität der zu lösenden Probleme.

Zusammenfassend kann gesagt werden, daß Ada95 zur Implementierung sicherheitskritischer Software geeignet ist und daß die im Annex H gemachten Spracheinschränkungen ihr Ziel erreichen können, was aber nicht so interpretiert werden darf, daß der komplette Sprachumfang ein Sicherheitsrisiko darstellt.

4.7 Literatur:

[1] Berlejung, H. & Baron, W.: Aspects of the Development oof Saftey-Critical Real-Time Software with the C Programming Language, in Software technik-Trends, Mitteilungen der GI-Fachgruppen 2.1.1 & 2.1.5 - 2.1.9, Band 16, Heft 4 , S. 21 - 25

[2] Draft IEC1508 - Functional safety: safety related systems, June 1995

[3] Romanski, G. & Chelini, J.: A Response to the Use of C in Safety-Critical Systems, in Softwaretechnik-Trends, Mitteilungen der GI-Fachgruppen 2.1.1 & 2.1.5 - 2.1.9, Band 17, Heft 1 , S. 38 - 43

[4] Engesser H. (Hrsg.); Claus V. , Schwill, A. (Bearb.): Duden „Informatik“, Ein Sachlexikon für Studium und Praxis, Mannheim, Wien, Zürich, 1988,

[5] MIL-STD 882C - Military Standard System Safety Program Requirements, January 19,1993

[6] DO-178B Software Considerations in Airborne Systemsand Equipment Certification (revised version of DO178-A

[7] Ada9X Project Report - Ada9X Requirements, Washington D.C., December 1990

[8] Ada95 - The Language Reference Manual & Standard LibrariesISO/IEC 8652:1995, Intermetrics, Cambridge Ma., 1995

[9] Ada95 - Rationale, The Language, The Standard Libraries, Intermetrics, Cambridge Ma. , January 1995

[10] ISO/IEC JTC SC22 WG9 Proposal for a new work-item: Guidance for the use of Ada in High Integrity Systems, 1997

[11] ISO/IEC JTC/SC22/WG9 N331, Resolutions from the meeting # 32 , June 1997

[12] Ada-Leitfaden Allgemeiner Umdruck 255, BMVg, Bonn, 1995

[13] Hoare, C.A.R.: The Emperor's Old Clothes, The 1980 Turing Award Lecture, in Communications of the ACM, Vol. 24, # 2, Febr. 1981, New York, p 75 - 83.

[14] Barnes, J.: Integrity Ada - The SPARK-Approach, Addison-Wesley, Harlow, 1997

[15] Roßkopf, A.: Use of a Static Analysis Tool for Safety-Critical Ada Applications - A Critical Assessment in Strohmeier, A. (Ed): Reliable Software Technologies -Ada-Europe'96, Springer, Berlin, Heidelberg,.. 1996, p. 183 - 197

[16] Safety Critical Software Handbook, Thomson Software/AONIX, 1995

[17] Dobbing, B. & Richard-Foy, M.: T-SMART - Task-Safe, Minimal Ada Realtime Toolset, in Hardy, K. & Briggs, J. (Eds): Reliable Software Technologies - Ada-Europe'97, Springer, Berlin, Heidelberg,.., 1997, p. 244 - 253.

[18] Wellings, A. (ed): Proceedings of the Eight International Real-Time Ada Workshop, acm Ada Letters Vol. XVII Number 5,. New York, Sept/Oct 1997.

Ada as a First Language

5 Ada as a First Language

Prof. Dr. Debora Weber-Wulff
Technische Fachhochschule Berlin
FB Informatik
Luxemburger Str. 10
13353 Berlin
weberwu@tfh-berlin.de

weberwu@tfh-berlin.de

Ada has been the first language of instruction for computer science engineers for more than five years at the University of Applied Science (TFH Berlin). I have taught the first language course four times and have gathered here some observations about and experiences with using Ada as a language for beginning programming language instruction.

Ada has always been an „industrial-strength„ language. Ada 83 was often seen as being a military language, complex and difficult to understand, with extremely expensive compilers. The latter problem has been resolved since the availability of GNAT and student versions of ObjectAda. But with Ada 95 introducing object-oriented constructs without using the „popular„ *class* terminology, even more voices have been raised questioning the suitability of Ada for beginning instruction.

5.1 The First Language Problem

There is a difference between learning a first programming language and learning further programming languages. Understanding is a process in which a congruence between new knowledge and organised knowledge must be constructed [Dut 94]. A beginner has no structures to build on other than everyday experiences as a computer user. A schema must first be found that properly reflects the programming language paradigm used. In learning further languages, one can build on the constructs already understood, even though the complete schema might need to be rebuilt in order to accommodate the new constructs.

This means that the first language is important because it sets the stage for the learning of further languages. In addition, learning the first programming language is often complicated by other skills which are lacking. Many beginners are not comfortable with using an editor and they may not even have a mental model of what happens when they make a copy of a program on a diskette. So in addition to the new schema for programming that they must learn, they are confronted with another schema for constructing a runnable program that may be very complicated.

Ebrahimi [Ebr94] reports typical beginner problems to be with loops (termination and initialisation) and with error handling. Since the error messages from the compiler are seldom understood, beginners will often resort to "syntax guessing", feeling that their

program is correct when it finally compiles. When a very strongly typed language such as Ada is used for beginning instruction, this will cause much frustration, as the probability of discovering a syntactically correct program by chance is not large. As a student of mine one remarked: "Ada is so frustrating because there are so many error messages, usually because I haven't thought out the problem right. But when I finally do get the syntax right, the program tends to be almost right. When I program in C++, the compiler accepts my programs more readily, but they seldom do what I intended them to."

This leads to a perception of Ada being a difficult language when the problem is actually that the Ada compiler is just reflecting the confusion on the part of the programmer! This is unfortunate, as beginners need to experience successes and to see the process of programming in a positive light. Using a language such as Eiffel, C++ or even Java can thus seem easier for beginners, because they see something happening more quickly. They don't realise that they will be spending quite a lot of time tracking down unexplained behaviour in their programs.

So even when it is reported, as in Hornecker [Hor 98], that beginners understand object oriented concepts more easily than general programming concepts, we must be sure to differentiate between what they feel that they have learned and what they really have learned. Besides, even in an object-oriented language, one must eventually deal with questions about storage locations, sequences of actions and the construction of loops. We still have to teach our students about the problems of using a finite machine, we have to help them learn how to find good representations for things they want to use in their programs, and we must teach them how to develop an algorithm.

Teaching Ada as a first language would seem to be a good choice, as students are immediately confronted with constructs designed to keep the harsh realities of programming from causing unexpected problems in industrial use. Programs can be kept readable and thus maintainable, they can be implemented with portability and efficiency in mind[5], and they can have robustness designed into the system. Many good principles of software design such as information hiding or reuse are readily available in the language.

How do children learn their first language? Experiments that were conducted to prove that humans would speak Latin naturally if they were kept without contact to a vernacular failed miserably. Children learn by imitation. We speak with them, we speak around them in correct language. They try out some sounds, and are rewarded by lots of attention and are encouraged to try again. As they get older more attention is paid to correcting their syntax, even though we often understand what they mean. They learn new words as they are needed, and eventually they have a mental model of their mother tongue so strong that no conscious effort is necessary in order to speak or to write.

How do we teach programming? We offer bits of syntax that do not make sense (What do I need a loop for? Why do I need to know what an array is?), give some programs to read, and hand out problems: calculate the area of a circle, input some numbers and calculate the average, convert Roman numerals to decimal numbers and vice versa.

[5] My colleague, Ulrich Grude, is fond of saying: "You can program efficiently but not portably in C, or portably but not efficiently in Java, or just use Ada to have both! "

Today's beginners are often angry when we assign these exercises — they want to learn how to write ray-tracing software and adventure games and network database applications! They demand to be taught these things from the beginning. On the other hand, companies are having trouble with their applications written in COBOL or C++ or they think that they need Java and are also demanding that the students to be taught in these languages.

What is necessary for a language — regardless of the paradigm used — to be successful for beginning programming instruction?

- The language must be simple in the sense that there are no confusing exceptions to syntax rules,
- there should not be numerous ways to do the same thing,
- it must be possible to make the results of computation visible quickly, so that the students can experience successes, and
- the compiler must operate in a very simple and comfortable environment.

No popular language, not even Ada, and no compiler can meet these criteria. Languages that were designed explicitly for instruction such as the original Pascal or Forth or Logo are difficult to scale up. When enough of the language is understood to attempt a large-scale project, the deficiencies of these languages are suddenly seen, and one must learn a new language in order to be able to cope with this particularity or some other problem.

Can Ada be both, a language for beginning instruction and a language for industrial programming? I still feel that it is possible, and wish to discuss in the following pages the problems with Ada that confuse or alienate beginners, and offer some suggestions for avoiding some of the more difficult ones.

5.2 Simplicity above all

It wastes precious teaching time trying to find understandable explanations for the beginner's questions about syntactical oddities that they can probably only understand when they have learned compiler construction. For example in Pascal, since the language does not have an explicit end if marker, a semicolon may not be used before an else so that the compiler can deduce the nesting properly. This type of problem has been nicely taken care of in Ada by having a rule that all structuring constructs must have a corresponding end marker. By making this a rule, it is more easily learned. Interestingly, beginners don't complain about this being "too much to type", on the contrary, since the end markers can be differentiated they can more easily inspect a nested construct for correctness than they can if there is just a collection of terminating closing braces.

If we confine ourselves for the beginning instruction to the "Pascal subset"of Ada, we start off with a simple language that does not suffer from ambiguous or disorienting syntax. We must still include enough of the Ada constructs such as packages so that the programs the students write will run with any Ada compiler. When in the course of

instruction something comes up, for example, the students have just realised that it could be useful to know what caused an error to occur, we are able to introduce just a bit more of full Ada in order to answer the questions.

The first step for beginning instruction is to avoid the LRM — its unambiguous language is just not easily understood by beginners. The American solution of teaching from a textbook is extremely helpful here. Since this is not typical at German institutions of higher learning and since German students do not like to read English, I translated many parts of Feldman and Koffman's book [FK92] for use in class. Handing out scripts, however, has not been as successful as having them copy definitions and small examples from the board and then handing out larger examples for reading and discussing.

What are the problems with Ada?

- **with**
  The necessity for "with"ing Ada.text_io is mystifying. I tell my students that this is just a formality, and ask them to bear with me until we are in a position to understand it. I offer an analogy to something from everyday life so that they can begin to construct their mental models: `with foo;` is like having the keys to foo's apartment. I don't have to go in, but if I need something that is available there, like a spaghetti maker, that I don't have in my apartment, I can get in to use it.

- **use**
  To use or not to use? Here I differ from Feldman and Koffman. They suggest not using use clauses so that the students can see exactly where each procedure and operator comes from. This caused problems with the unnatural use of infix operators in Ada 83, which can be avoided in Ada 95 by using a `use type` clause. But the programs still suffer from bloat, especially as they also use parameter names in all procedures and functions:

  ```
  text_io.get (item => i);
  ```
  vs.
  ```
  get (i);
  ```

  Although the former is indeed much better for programs that will have to be maintained, it is difficult for a beginner to filter out the important lexemes from the ones that are only used as markers. I used to think that it was better to make the students program for maintenance from the start — keep it readable and unambiguous — but I have found that this keeps many students from understanding what they are writing. So I now aim for first learning the structures, and then learning how to use the additional advantages Ada offers.

- **main procedure**

  Not being able to distinguish the main program by inspecting the source text, however useful this may be in a professional context, is a source of confusion. What is the difference between this "outer"procedure and other procedures I declare? The confusion is complete when we begin using packages, as a

package can contain many procedures, none of which are the main, but all of which resemble it! When we first try and link a program, our compiler asks us by way of a window full of little boxes to fill out, which procedure should be the main one. It does suggest the current procedure, but a beginner does not understand why the compiler doesn't just carry on at this point. It would be helpful to have an environment that can toggle all the advanced functions off. Since we don't, I just suggest that they push *okay*, even though I feel bad about training them to just push the button suggested without thinking about what the window means.

- **declarations**
  The declaration part suffers from the general problem of students not understanding the difference between a variable and a type. Ada compounds the problem by having both subtypes and types, which are not readily differentiated. If one only permits the students to use types, the conversions necessary for type compatibility for even simple calculations are overwhelming. If subtypes are used there are now two different kinds of type declarations: numeric ones that use subtype, while type must be used for enumerations, records and arrays. If the student forgets where the dividing line is and writes `type T1 is string (1..6);` the compiler will complain that a `new` is missing. The students type in `new` if that is what the compiler wants, and are angry that now the `puts` and `gets` from `Ada.text_io` will not work.

  Ada also has typed and untyped constants. I try and avoid the latter if at all possible, because the beginners do not understand why there is a need for both kinds. Of course, I encourage them to use constants for all literals.

- **repetition**
  Although there are some analogies in everyday life such as singing a round or working on an assembly line, this is one of the most difficult concepts for beginners to understand and to learn to use. I have tried teaching loop invariants with little success. Having different kinds of loops does not make sense to beginners when they don't even know why they want to repeat something anyway. Ada offers a very elegant way out of the problem: the general loop that uses an `exit`-statement to terminate. This is much easier to teach first, and then to show that for one kind of loop there is an alternate syntax.

- There is still a problem with the `for`-loops, however. Although it makes much sense in professional use to have implicit declaration of the loop variable so that it is not available outside of the loop, this breaks the rule that all variables have to be defined. I have not found anything that helps here, perhaps an editor that colors all loop variables green so that one can explain that only black variables need declarations.

- **input/output**
  The need to instantiate packages for all types for input and output is a major drawback. It is too much to be explained away as "magic", the students want to understand what is behind it. If we take the easy way out and always define an integer I/O-package and just use integer subtypes for the numeric ones we are

fine until we get to arrays and records. If Ada knew how to handle numbers and strings (which are supposed to be arrays), why can't it output a simple array of integers? At this point a digression to at least explain what a generic package is can be useful.

- **parameters**
  Getting beginners to understand the use of parameters is very difficult when they are not quite sure what a variable is. I thought that using named associations to map the formal parameters to the actual parameters would contribute to the construction of a useful mental model, but this does not seem to be the case. It is also confusing for those who have some prior programming experience, as this syntactical model is not used in most other languages.

  Using default values for some of the parameters ist also quite confusing, especially when they are trying to understand overloading. They are not sure which model, overloading or default parameters, is at work. Once I began using all the parameters for the procedures I used, for example `iio.put(i,3);` to specify the width, I had fewer questions to answer. This is perhaps another feature of Ada that is useful professionally, but confusing for beginners.

- **private parts in packages**

  The concept of separating the specification from the body of a package is readily understood by beginners. However, not all understand easily that the body may define additional procedures for local use which are not visible outside of the package. This public/private problem is compounded by the fact that the specification, which is not supposed to mention the private procedures, is forced to specify the private data structures in plain sight. The reasoning behind this is of course to make life easier on the compiler writer, but it is unfortunate that this chance to cleanly separate public and private parts by perhaps requiring an extra file for the private parts of the package was not taken.

- **put/get model outdated**

  The mental model used for the `put` and `get` procedures in `Ada.text_io` is built upon a model of a typewriter. One can only write in a forward direction, when a character has been written the carriage moves on to the next position, at the end of a line the carriage is returned to the first position and a line feed advances to the next line. The problem with this model is that we have had computers for text editing purposes for so long, that the current generation of beginning programmers may never have operated or even seen a typewriter in their lives! The model is completely foreign to them, and thus, they have a hard time understanding how to write to a file or to the standard output.

  The expected model is for a cursor to move to a position in a window and to write at that position! I attempted to alleviate this problem by expanding Feldman and Koffman's simple screen package to permit the students to use this model of output. Unfortunately, the ANSI-sequences used for this purpose are not properly executed on Windows NT machines. We have the situation for

the Windows platforms that there is no longer an easy way to do simple cursor positioning, and beginners are not up to programming with Microsoft Foundation Classes. Even if there exists an Ada package offering a simple window environment for Windows, I am not sure that I want to have to discuss window models with beginners who are not yet firm on how to construct loops.

- **real numbers**
  The beginning students are often interested in using real numbers right away, since they encounter them in currency calculations every day. But since they mostly do not have the background to understand the problems involved in rounding errors, it is better to just stick with integers for the first semester. As a compromise for currency calculations, the teacher could prepare a currency package for use without going into details.
  In order to demonstrate to them the dangers involved with using real numbers without careful thought I use the following program from my colleague Ulrich Grude: We define

```
type REAL is digits 15 range 0.0 .. 5000.0; -- small but exact
type INT  is           range 1   .. 500;    -- for counting
R       : constant REAL := 3.0;
X1      :          REAL := 0.01;
X2      :          REAL := 0.01;
WIE_OFT :          INT;
```

  and then calculate two algebraically equivalent functions, reporting the results. Since the functions are equivalent, the results should be almost the same.

```
for I in 1 .. WIE_OFT loop
    X1 := (1.0 + R) * X1 - R * X1**2;   -- x := (1+r)x-rx**2
    X2 := X2 + R * X2 * (1.0 - X2);     -- x := x+rx(1-x)
    INT_IO.put  (I);
    REAL_IO.put (X1,0);
    REAL_IO.put (X2,0);
    REAL_IO.put (X1/X2,0,5);
    text_io.NEW_LINE;
end loop;
```

  Depending on the compiler and the machine the program is executed, any time between the 15th and the 315th iteration the results begin to diverge. The students first suspect that the compiler or the computer is "broken" until we get deeper into what exactly happens when we multiply real numbers.

- **arrays**
  The need for unconstrained arrays is not readily seen by beginners. They also do not see that the string data type is just a particular kind of array. What is confusing here is the shorthand use of "hello" to represent('h', 'e', 'l', 'l', 'o'). Drawing pictures of the arrays as collections of boxes that have their index written on the front and their values contained in the box is helpful for one-dimensional arrays. Arrays of arrays are difficult for beginners to visualise, it can be helpful to draw pictures for them.

- **exceptions**
  The ease with which exceptions can be introduced in Ada is a major plus. The need for handling exceptions arises when the first `constraint_error` is raised during a program execution. Since one only need to define the block limits for the exception handling block, they can be explained and quickly inserted into a program that needs them. The flow of control is surprisingly easy to understand.

- **pointers**
  Once the beginners make it to the point where they have understood variables and records, it is easy to introduce pointers. Since Ada has typed pointers that can only point to the right kind of object, the programmer is forced to think the desired structure through. Many students have reported that after having learned pointers in Ada, it was very easy to get pointers right in C(++).

5.3 The Compiler Environment

A good compiler environment for beginners is often completely different from a good environment for industrial use. Beginners are quite happy to just have an editor and one button each for "compile" , "link" and "run". They want to see the error messages merged into their texts, as counting lines is a lot of work, and they want their programs to look the same on the screen and on the printer. This means that we have to overcome the tendency of some Windows products for choosing a proportional font as the default one for printing when code is being output — it helps the understanding of nested concepts if the statements are properly indented.

Beginners are quite proud of their work when they have finally managed to get a program to produce output to the screen. They want a copy to take home and a copy to hand in. But since we work in a DOS-Box, there is not much more than `PrintScreen` that can be done to capture the output, which is unfortunate. Redirecting the output to a file is nasty if we also want to request input — the computer appears to be „broken„ as the request is logged to the file and the cursor waits for input. If all input and output were logged by default, we could just print out the logfile of the last run.

A possible solution is to implement a package `test_io`[6] that has the same interface as `text_io`. The implementation mirrors writing to the screen in a logfile, as well as all input. In this manner one can obtain a transcript of a program run just by changing one letter in the `with`-clause and recompiling the program before running it. For example, `put` is implemented like this:

```
procedure PUT      (ITEM: in  character) is
   begin
      ada.text_io.put(item => ITEM);
      ada.text_io.put(item => ITEM, file => PROTOKOLL);
   end PUT;
```

---

[6] Idea and implementation by Ulrich Grude. In his version the input is also collected in a separate file, so that it can be used as test input for future reference.

In this past semester I used ObjectAda from *Aonix*, which was included in the Ada-Tour CDs we obtained from *competence center informatik GmbH*, Meppen. We had a difficult time getting our "Hello World" programs to run — first we had to set up a project, this created a debug and a production directory, we had to choose a target machine and so on, all things that are wonderful for industrial use but quite confusing for beginners. When we finally got everything set up and running, we wanted to make a copy of the executable for taking home. The source files are hidden in a completely different directory from the executables, and the names of the directories are very difficult to type because they are so long and identical in the first 16 letters or so. This complexity is quite useful if you are in a production environment needing to cross-compile to multiple platforms and are using versioning software, but it is overwhelming for beginners. Many were anxious that their programs were "lost". The compiler environment for beginners needs to show clearly where exactly all the files can be found and allow for easy copy of the files to diskette.

5.4 Summary

There are a number of factors that contribute to Ada being considered a complex and difficult language. While the compiler or the operating system is responsible for many of these complexities, there are quite a number of problems that arise from the expressive power of the language. But by not using many of the industrially useful features of Ada, many of the problems can be avoided that are confusing for beginners.

I still believe that Ada is a very good language for beginning programming instruction and that there are a number of good textbooks available. Unfortunately, my school has decided that it must cater to the whims of the students and will be teaching Java as a first language this semester. I wish my colleagues luck, I will concentrate on teaching other courses. The first question I had from the colleague who is preparing the lectures perhaps tells of what is in store for them. He wanted to know if there wasn't an easy way to just read in two numbers from the standard input stream so he could add them and display the results. No, he either has the choice there of creating a panel and putting two input and an output box on it, or of fiddling with streams and casting to get at the integer values. I hope to be able to report at some future *Ada-Deutschland* meeting that we are returning to Ada.

5.5 Bibliography

[Dut94] Dutke, Stephan. *Mentale Modelle: Konstrukte des Wissens und Verstehens — kognitionspsychologische Grundlagen für die Software-Ergonomie*. Verlag für angewandte Psychologie, Göttingen,1994.

[Ebr94] Ebrahimi, Alreza. Novice programmer errors: language constructs and plan composition. In: *International Journal of Human-Computer Studies* 41, Sept. 1994, S. 457-480.

[FK 92] Feldman, Michael B. and Koffman, Elliot B. *Ada: problem solving and program design*. Addison-Wesley, Reading, 1992.

[Hor98] Hornecker, Eva. Programmieren als Handwerkszeug im ersten Semester. In „*Informatik und Ausbildung*„, V. Claus (Hrsg.) Springer Verlag, Heidelberg, 1998.

Ada-Metriken und ihre Anwendung

6 Ada-Metriken und ihre Anwendung

Andreas Schwald

Zusammenfassung

Diese Arbeit gibt einen Überblick zum Thema Metriken für Ada95. Ausgehend von einigen Vorschlägen in der Literatur werden einige auf Ada95 abgestimmte Vorschläge für Metriken und deren Einsatzmöglichkeiten diskutiert.

6.1 Der Zweck von Maßen

Entsprechend dem GQM-Ansatz (Goals - Questions - Metrics, Basisli94) sollen quantitative Kenngrößen („Metriken“ nach dem englischen Sprachgebrauch) Informationen zur Beantwortung von Fragen liefern, die sich beim zielorientierten Handeln auf das Erreichen eines angestrebten Ziels beziehen. Softwaremetriken dienen als Soll-Vorgaben in Projekten und als quantitative Indikatoren für die Qualität von Software. „The measurement of the software development process will not improve it“ (Wearing92) - aber ein Projekt braucht klare Vorgaben für das globale Ziel und für die laufende Koordination. Der Umstand, daß gemessen wird, wirkt sich auch auf die Arbeitsweise der Entwickler aus und ermöglicht ihnen den Nachweis ihrer Leistung.

Insbesondere sind quantitative Kenngrößen wichtig als

- Vorgaben für den Soll-Ist-Vergleich (Aufwand und Termine, Größen- und Qualitätsmaße für (Teil-)Produkte,
- z. B. empfohlene/maximale Größe eines Programmoduls, Richtwerte für Komplexitätsmaße, Dokumentationsmenge u. dgl.)
- Hilfsmittel für die Bewertung von Arbeitsergebnissen (von der projektinternen Abnahme bis zur vergleichenden Evaluation von Produkten)
- Indikatoren für Stark- und Schwachstellen und für mögliche Problembereiche bei Arbeitsergebnissen und bei der Projektdurchführung (Frühwarnung, risk assessment). Bei Komplexitätsmaßen und ähnlichen Kenngrößen kann die Identifizierung von sog. Ausreißern mit ganz untypischen Werten hilfreicher für die Steuerung der Projektarbeit als die Bestimmung von Durchschnittswerten.

(Wearing92) nennt als Anwendungsbereiche „management, training needs, cost estimation, realiability measurement, integrity measurement, productivity measurement; setting goals (measurable achievement).“

Es gibt viele meßbare Programmattribute, die für mehrere Sprachen anwendbar sind, z. B. Maße für Programmgröße und -komplexität, Modulattribute wie fan-in/fan-out, auch

OO-Maße zur Charakterisierung von Vererbungshierarchien und Kopplung von Klassen bzw. Objekten, siehe z. B. (Chidamber94), (Welch96), (Dumke97).

Auch die allgemeinen Modelle zur Aufwandsschätzung sind für Ada - nach entsprechender Validierung bzw. Kalibrierung - in gleicher Weise verwendbar wie für andere Programmiersprachen. Das Studium solcher Modelle zeigt, daß die Programmiersprache nur einer von vielen Einflußfaktoren für den Erfolg eines Projekts und für die Güte eines Produkts ist. Bei der Beurteilung der Auswirkungen der Wahl einer Programmiersprache sind geeignete Indikatoren zu definieren (z. B. Produktivität in Codezeilen, Fehlerzahlen), die sprachspezifischen Unterschiede (z. B. Zählregeln) klarzustellen, die Einflüsse anderer Entscheidungen (z. B. Wahl der Entwurfsmethoden, Softwarewerkzeuge) zu beurteilen, außerdem die Auswirkungen anderer Randbedingungen (allgemeine Vorgaben, personbezogene Attribute) zu berücksichtigen.

Für Ada (bzw. eine andere Programmiersprache) sind folgende Fragen von Interesse:

* Ist die Auswirkung von Ada auf die Projektdurchführung und die Programmqualität meßbar?
* Welche Kenngrößen für Projekte und Produktqualität haben für Adaprogramme bzw. -projekte signifikant andere Werte als für vergleichbare (mit Ada konkurrierende) Sprachen?
* Gibt es für Ada-Programme sprachspezifische meßbare Attribute, die sich als Indikatoren für Qualitätseigenschaften (z. B. Zuverlässigkeit, Wartbarkeit) eignen?
* Wie läßt sich der Einfluß der Programmiersprache von anderen Einflüssen (Entwicklungsmethoden, Werkzeuge etc.) mit einiger Sicherheit unterscheiden? Hier sind auch die Wechselwirkungen einer Sprache mit der Programmierumgebung (z. B. Bibliotheksverwaltung und Konfigurationsmanagement) und mit der Entwurfsmethode zu sehen.

6.2 Sichtweisen für Ada95-Programme

Meßbare Attribute sind in Kontext eines Systemmodells bzw. einer bestimmten Sichtweise zu erklären. Für ein Ada-95-Programm kommen z. B. in Betracht:

- Programm als Text, der in Zeilen (Anzahl Text- oder Codezeilen) bzw. in Module (Anzahl Einheiten) und Übersetzungseinheiten (Anzahl Einheiten and Untereinheiten) gegliedert ist.
- Menge von Partitionen bzw. (Hardware-)Knoten mit ihren Verbindungen für ein verteiltes Programm.
  - Verbindungen zwischen den Partitionen
  - Zugriffe und Informationsflüsse zwischen Partitionen
- Wald von Bibliothekseinheiten mit Zugriffsbeziehungen (with-Relation und Sichtbarkeit zwischen Vater- und Kind-Bibliothekseinheiten) sowie die Verfeinerungen dieser Beziehungen für Untereinheiten
  - Sichtbarkeit (with-Klauseln)
  - Zugriffe auf externe Größen und Informationsflüsse zwischen den Einheiten

- Menge von Spezifikationen (Ergebnis des Entwurfs)
  - Charakterisierung der Exportschnittstelle: Anzahl und Komplexität der exportierten Größen,
  - aufgegliedert nach ihrer Art, z. B. Typen (privat, zusammengesetzt, erweiterbar, ...) mit ihren Operationen,
  - andere Operationen, Konstante, Variable, Prozesse, Ausnahmen
  - Charakterisierung der Importschnittstelle: Anzahl, Herkunft und Komplexität der importierten Größen,
  - aufgegliedert nach Herkunft und Art der Größen - globale Typen, Konstanten, Ausnahmen und Variablen
- Wald von Übersetzungseinheiten mit ihren Abhängigkeiten bezüglich Sichtbarkeit und Übersetzungsreihenfolge einschließlich der Rümpfe
  - With-Abhängigkeiten für Spezifikationen, Rümpfe und Untereinheiten,
  - Verfeinerung: nichtlokalen Zugriffe aus einem Modul auf externe Größen - evt. modulweise und nach Art der
  - externen Größen aufgegliedert (Zugriffscluster)
- Menge von Vereinbarungen von Prozessen und geschützten Objekten, ihre statischen (und dynamischen) Abhängigkeiten
  - Eingangsvereinbarungen und Eingangsaufrufe
  - Aufrufe von Prozeduren und Funktionen von geschützten Objekten
  - Ungeschützter Zugriff auf globale Größen
- Reuse-Sicht
  - generische Einheiten und ihre Ausprägungen (Anzahl generische Parameter einer Einheit),
  - erweiterbare Typen mit ihren Ableitungsbäumen
  - (Anzahl Komponenten und primitive Operationen eines erweiterbaren Typs),
  - Zugriffe auf Bibliotheken
  - parametrisierte Typen (Diskriminanten von Verbundtypen)
  - parametrisierte Unterprogramme

Für diese globalen Sichtweisen sind jeweils die Attribute der Elemente und der relevanten Beziehungen zu betrachten und - soweit möglich und für den jeweiligen Zweck von Interesse - quantitativ zu charakterisieren.

Die Beziehungen zwischen Zielvorgaben bzw. Sollvorstellungen (im Kontext einer bestimmten Sichtweise) wird explizit gemacht durch die Formulierung von Standpunkten (viewpoints). Nach (Roberts79, Zuse85, Fenton91, Chidamber94 u. a.) kann eine Menge von Objekten (hier: Entwürfe bzw. Programme) aufgefaßt werden als empirisches Relationensystem (d. h. einer Menge von Elementen, von Beziehungen zwischen und zweistelligen Operationen auf den Elementen). Ein Maß ist eine Abbildung in ein formales Relationensystem (meistens wird einem Element des

empirischen Systems eine reellen Zahl zugeordnet), welche die empirischen Relationen (z. B. „benutzerfreundlicher“, „zuverlässiger“, „besser wartbar“, „komplexer“) auf eine entsprechende Relation (z. B. „>“) des formalen Systems abbildet. Dies läßt sich am Beispiel der Komplexitätsmessung verdeutlichen. Ein Programmierer hat intuitive (evtl. gut begründete und durch Erfahrung erhärtete) Vorstellungen über die Komplexität verschiedener Objekte (Elemente), z. B. kann er die Auffassung vertreten, daß - ceteris paribus - die Komplexität eines Pakets mit der Anzahl der exportierten Vereinbarungen größer wird. Diese Vorstellung wird als Standpunkt bezeichnet. Ursprünglich wurde dieser Begriff für die Evaluation von Information-Retrieval-Systemen verwendet, von Zuse u. a. auf die Komplexitätsmaße für Programme angewandt. Nach (Fenton91) sind Standpunkte, die das „intuitive understanding“ charakterisieren, der „logical starting point“ für die Definition von Metriken.

Für das Programmieren im Kleinen (Rümpfe von Unterprogrammen, Paketen und Prozessen) bieten sich u. a. folgende Indikatoren für die innere Komplexität eines Moduls und für die der Umgebungsbezüge an:

- Komplexitätsmaße für Flußgraphen
- Datenkomplexität, z. B. der an einer Stelle sichtbare Namensraum; Verschachtelungsmaße für zusammengesetzte Typen
- Bezüge auf nichtlokale Größen (Annahme: Diese sind „schwieriger“ als Zugriffe auf lokale Größen.)
- Charakterisierung der Lokalität einer Komponente (Anzahl der lokalen, exportierten und importierten Größen)
- Charakterisierung der Zusammengehörigkeit eines Moduls aufgrund der internen Struktur (z. B. Implementierung einer Datenkapsel, Operationen eines Typs, andere Kriterien für Gemeinsamkeit) oder der Art der Verwendung (z. B. ein Zugreifer verwendet alle exportierten Größen, mehrere Zugreifer verwenden nur eine bestimmte Teilmenge der exportierten Größen).

Auch Fragen des „program understanding“ im Sinne der Mustererkennung und der Orientierung in einem großen Namensraum (z. B. Woods96) sollten hier berücksichtigt werden.

Die Trennung von Spezifikation und Rumpf einer Programmeinheit gibt die Möglichkeit, Spezifikationsattribute frühzeitig zu erheben. Die Frage nach der Eignung solcher Attribute zur Beurteilung der Entwurfsqualität oder als Indikatoren für besonders schwierig zu implementierende oder besonders fehleranfällige Module (echte Frühwarnung)? ist schon mehrfach untersucht worden (Gannon86, Zage92, Agresti92), zum Teil in Abhängigkeit von einer bestimmten Entwurfsmethode (HOOD in Wearing92).

6.3 Gerichtete Graphen als allgemeine Strukturmodelle

Sehr häufig hat ein Systemmodell die Struktur eines gerichteten Graphen. Kenngrößen beschreiben Eigenschaften des Graphen, wobei ein Knoten zunächst als eine Einheit (mit gewissen Attributen) betrachtet wird. Bei der Detaillierung der Betrachtungsweise wird die innere Struktur eines Knotens deutlich. Damit sind in aller Regel auch die

Kanten und ihre Interpretation zu detaillieren bzw. neu zu interpretieren (z. B. with-Klausel für den Zugriff zwischen zwei Modulen => Aufrufe von Unterprogrammen und Zugriffe auf Objekte => gerichtete Datenflüsse über in- und out-Parameter).

Wichtige Kennzahlen eines gerichteten Graphen:

- n ... Anzahl Knoten
- e ... Anzahl Kanten
- p ... Anzahl zusammenhängender Komponenten
- zyklomatische Zahl: e - n + 2*p
- Die zyklomatische Zahl ist die Zahl der linear unabhängigen Kreise eines ungerichteten Graphen, sie charakterisiert auch die Abweichung von einer Baumstruktur.

Für bewertete Graphen: Weglängen und Kapazitäten.

Lokale Betrachtung (ein Knoten k und seine Umgebung):

Ausgangsgrad ... Anzahl Kanten, die von k ausgehen

Eingangsgrad ... Anzahl Kanten, die in k enden

kürzester Weg ... Pfad minimaler Länge zu einem anderen Knoten

Menge der von k aus erreichbaren Knoten

Speziell: Baumstruktur

(e = n-1, zyklenfrei, Wurzel hat Eingangsgrad 0, alle anderen Knoten Eingangsgrad 1)

n ... Anzahl Knoten

Grad ... Maximalzahl der Söhne eines Knotens (max. Ausgangsgrad)

Höhe des Baumes ... längster Weg von der Wurzel zu einem Blatt

Tiefe eines Knotens: Abstand eines Knotens von der Wurzel (bei Blockstruktur: Verschachtelungstiefe)

Anzahl der Söhne (Ausgangsgrad) eines Knotens

Beispiele für die Verwendung dieser Meßgrößen:

Die Komplexität eines sequentiellen Programmoduls nach McCabe ist die zyklomatische Zahl des Flußgraphen (keine Berücksichigung von Nebenläufigkeit, Datenstrukturen und externen Schnittstellen).

Die in (Chidamber94) definierten OO-Metriken NOC (number of children) und DIT (depth of inheritance tree) gehören zu den angeführten Kennzahlen (Ausgangsgrad eines Knotens bzw. Tiefe in einem Vererbungsbaum).

6.4 Charakterisierung von Ada-Projekten

Für den Vergleich von Ada mit anderen Sprachen (z. B. Reifer89, Zeigler97) werden als Größenmaße gewöhnlich Codezeilen (geeignet normiert) oder Function Points (Zeigler: „features“) gewählt. Die Vergleiche beziehen sich meist auf Qualitätsindikatoren (Fehlerdichte in Fehler/KSLOC, Anzahl-Fehlermeldungen/KSLOC) und Angaben zur Wirtschaftlichkeit. Für den Ressourcenverbrauch verwendet (Reifer89) Kosten/SLOC. Die bekannten Definitionsprobleme für die Programmlänge und andere Größenmaße sind besonders beim Sprachvergleich zu beachten. Sie verhindern gewöhnlich einen detaillierteren Vergleich, der auf soliden empirischen Daten aufbaut. (Jones95) nennt zur Umrechnung von Function Points in Codezeilen für Ada83 einen Bereich von 60 bis 80 LOC/FP, für Ada95 einen Bereich von 28 bis 110 LOC/FP.

Für die Aufwandsschätzung werden in (Boehm89) folgende Änderungen von COCOMO für Adaprojekte angeführt: Der Exponent der Schätzformel ist abhängig von 4 Größen (frühe Beseitigung von Risiken, solide Architektur, stabile Anforderungen und Ada „process maturity“), als neue Aufwandsmultiplikatoren für Ada-Projekte werden der Grad der Wiederverwendung und die Sicherheitsauflagen für ein Projekt angeführt. Mehrere andere Multiplikatoren (z. B. Produktkomplexität, persönliche Fähigkeiten, Erfahrung mit der Programmiersprache, Einsatz von Entwurfsmethoden und Softwarewerkzeugen) wurden geändert. Die folgende Tabelle - Einzelheiten siehe (Boehm95) - zeigt die Ergebnisse der Anpassung und Neukalibrierung von COCO-Modells an die Gegebenheiten von Ada-Projekten sowie die allgemeine Entwicklung der Software-Produktionstechnik..

	COCOMO (1981)	Ada COCOMO (1988)	COCOMO 2 - Stufe 3 (1995)
Exponent der Schätzgleichung Aufand = ΠAufw.Fakt. * SLOC**exp	Art des Projekts: organic - 1.05, semidetatched - 1.12, embedded - 1.20	1.04 - 1.024, abhängig von early risk elimination solid architecture stable requirements Ada process maturity	1.01 - 1.026, abhängig von precedentedness conformity early architecture risk resolution team cohesion process maturity (SEI)
Produkt-Aufwandsfaktoren	RELY, DATA, CPLX	RELY*, DATA, CPLX*, RUSE	RELY, DATA, DOCU*+, CPLX+, RUSE*+
Plattform-Aufwandsfaktoren	TIME, STOR, VIRT, TURN	TIME, STOR, VMVH, VMVT, TURN	TIME, STOR, PVOL (=VIRT)
Personal-Aufwandsfaktoren	ACAP, AEXP, PCAP, VEXP, LEXP	ACAP*, AEXP, PCAP*, VEXP, LEXP*	ACAP*, AEXP+, PCAP*, PEXP*+, LTEX*+
Projekt-Aufwandsfaktoren	MODP, TOOL, SCED	MODP*, TOOL*, SCED, SECU	TOOL*+, SCED, SITE*+

* ... neu kalibriert, + ... andere Skala

Auch beim PRICE-Schätzmodell wurden nach (Reifer89) mehrere Kostenfaktoren für Ada-Projekte rekalibriert (Produktkomplexität, persönliche Fähigkeiten) bzw. neu eingeführt (Realzeitanforderungen, Ausmaß der Wiederverwendung, Wiederverwendungskosten).

(Zeigler97) verwendet - basierend auf Projektdaten über einen längeren Zeitraum in einer homogenen Entwicklungsumgebung für Ada und C - viel detailliertere Maße, z. B. Kommentare/KSLOC, Textzeilen/feature, SLOC/feature, cost/feature (C: 299$, Ada: 183$). Von allgemeinem Interesse sind folgende Wartungs-Kenngrößen:

customer interactions („tickets")	16440
possible defects	5072
deficiency reports (bugs, requests, duplicates)	2004
actual bugs	1242

reported bugs	- critical	C: ca. 200	Ada: ca. 30
	- severe	C: ca. 600	Ada: ca. 50
	- minor	C: ca. 240	Ada: ca. 30

6.5 Charakterisierung der Einbettung (Realzeitumgebung, Verteilung) eines Programms

(Wearing92) nennt für die Messung des „problem size" von Realzeitsystemen analog zur Bestimmung der FP's (on the requirement level) folgende Kenngrößen:

(R1) # external input streams - including sensors using interrupt mechanism

(R2) # external output streams - including output devices

(R3) # interrupt rate for each of (R1) and (R2),
- average and maximum rates for #interrupts / time unit

(R4) # concurrent activities - different functions to be carried out at any time.

(R5) # processors - interacting processors on which the software is loaded.

(R6) # operands or operators or algorithms
- size consumed by mathematical equations specified f or the system.

Für verteilte Systeme scheinen Ergänzungen zweckmäßig (Charakterisierung der Netztopologie, z. B. Anzahl Knoten und Anzahl Verbindungen zwischen Partitionen und auf HW-Ebene, Charakterisierung der Zugriffe und Datenflüsse zwischen Partitionen). Für andere Anwendungsbereiche sind (abhängig von der Aufgabe und der

Spezifikationsmethode) auch andere Kenngrößen für die Charakterisierung der Systemgrenze zweckmäßig.

6.6 Charakterisierung der globalen Programmstruktur

In Ada werden die Beziehungen zwischen Bibliothekseinheiten (Programmkomponenten) durch with-Klauseln dokumentiert; neu in Ada95 sind Hierarchien von Bibliothekseinheiten. Die Identifizierung und Abgrenzung größerer Programmkomponenten (z. B. Subsysteme, UML-packages) kann im Kontext der jeweiligen Spezifikationsmethode erfolgen und auf eine Ada-Spezifikation abgebildet werden.

Im Ada83-Rationale werden drei Gesichtspunkte für die Bildung von Paketen (Zusammengehörigkeit) genannt:

(1)Named collections of declarations: Logically related variables, constants and types to be used in other program units.

(2)Packages of subprograms: Logically related functions, procedures, and possible exceptions, which share internal own data, types, and subprograms.

(3)Encapsulated data types: Definitions of new types and of their associated operations in such a way that the user does not know (or care!) how the operations are implemented.

In den vordefinierten Standardbibliotheken von Ada95 gibt es Beispiele für alle drei Arten der Strukturierung (System und Ada.Numerics für (1), Ada.Numerics.Generic_Elementary_Functions für (2), EA-Pakete für (3)). In Ada95 führen die Möglichkeiten zur Erweiterung von Typen und Bibliothekseinheiten zu neuen Abhängigkeiten zwischen Bibliothekseinheiten. Metriken für Ada95 sollten daher OO-Aspekte umfassen, sich aber keineswegs darauf beschränken.

Grundlegend sind die für den Stuctured Design (Constantine74) formulierten Konzepte der Modulkopplung (coupling) und der Zusammengehörigkeit (cohesion) der Elemente eines Moduls. Dabei ist generell schmale Kopplung und hohe Zusammengehörigkeit wünschenswert. Es gibt verschiedene Charakterisierungen, z. B. die in SA/SD verwendete Ordinalskala für die Kopplung nach der Art der Kommunikation (auf Sprachen wie Fortran und PL/I abgestimmt, teilweise auch für Ada anwendbar).

Für objektbasierte Systeme gibt (Welch96) folgende Erklärung der wünschenswerten Eigenschaften:

- information hiding (Verbergen von Entwurfsentscheidungen und Implementierungsdetails in Modulen)
- cohesion (ein Modul bietet eine Abstraktion)
- encapsulation (zusammengehörige Typen und Methoden werden in einem Modul eingekapselt)
- loose coupling (geringe oder keine Abhängigkeit zwischen Modulimplementierungen). Eine Änderung einer Modulimplementierung wird dann meist keine Änderungen in anderen Modulen erfordern.

Die gute Unterstützung für das Geheimnisprinzip (Trennung Spezifikation - Rumpf, private Typen, Eingänge und Pakete) erleichtert die Identifikation von Kenngrößen zur Charakterisierung von Import- und Exportschnittstellen. Meßbare Attribute des Programmtextes, die gute Indikatoren für „logically related“ darstellen, sind schwieriger zu identifizieren.

Bei der Betrachtung von Export-Import-Schnittstellen ist offensichtlich eine Differenzierung nach der Art der exportierten Größen geboten: Gemeinsame Typen sind Voraussetzung für die Übergabe von Information, und die globale Sichtbarkeit von Ausnahmen ist die Voraussetzung für ihre gezielte Behandlung. Auch die globale Sichtbarkeit von Konstanten ist bezüglich Wartbarkeit und Fehleranfälligkeit anders zu bewerten als die Verwendung von globalen Variablen.

Die Abstraktionsweise des Systemmodells bestimmt die meßbaren Attribute. Es scheint zweckmäßig, vier Darstellungsebenen zu unterscheiden:

(a) Systemmodell in einer Modellierungssprache (z. B. UML, HOOD, siehe z. B. (Wearing92))

(b) Grobentwurf (architectural design, Paketspezifikationen)

(c) Feinentwurf (detailed design, Rümpfe auf Pseudocode-Ebene)

(d) Programmcode

Darauf abgestimmte Betrachtungsweisen für Ada-Spezifikationen:

- Zugriffsbeziehungen aufgrund von with-Relationen und Paketerweiterungen
- Sichtbarkeit von externen Größen
- Zugriffe auf externe Größen (basierend auf dem Feinentwurf).

Einige Gesichtspunkte:

- With-Klauseln sollen so lokal wie möglich plaziert werden.
- Gemeinsame Typen sind notwendig, gemeinsame Variable zu vermeiden.
- Die Vererbungsstruktur soll mit der Zerlegungsstruktur übereinstimmen. Ein erweiterbarer Typ mit seinen primitiven Operationen wird demnach in einem Bibliothekspaket vereinbart, eine eventuelle Erweiterung in einem Erweiterungspaket (child package).
- Ada83-Regel: Eine Paketschnittstelle soll keine Prozeßspezifikation enthalten. solche Spezifikationen sind im Paketrumpf zu verbergen. Die Neuerung der geschützten Objekte und die Möglichkeit, die (globale) Kommunikationsstruktur als Teil der globalen Programmstruktur zu dokumentieren, sprechen m. E. gegen diese Regel.

6.6.1 Kopplungsmetriken

Als quantifizierbare Attribute zur Charakterisierung von Import-Export-Schnittstellen sind zu sehen:

Importschnittstelle:

Globale Sichtbarkeit:

Anzahl Bibliothekseinheiten in with-Klauseln

(evtl. Verfeinerung: Pakete - Unterprogramme, generische - nicht generische Einheiten),

Anzahl Vorfahren für Erweiterungspakete (child packages).

Anzahl der in einem Modul sichtbaren externen Größen (ohne/mit use-Klausel)

Zugriffe auf externe Größen:

Anzahl Zugriffe insgesamt in einer Übersetzungseinheit (inklusive/exklusive Untereinheiten), aufzugliedern nach

Bibliothekseinheiten (Profil für den Grad der Kopplung mit den anderen externen Einheiten)

nach Art des Objekts und des Zugriffs, z. B.

- Typ verwenden oder erweitern,
- generische Einheit ausprägen
- Konstante lesen,
- Variable lesen/schreiben,
- Operationen aufrufen/weitergeben (evt. mit Unterscheidung nach Parameterzahl bzw. nach der Zahl der angegebenen akt. Parameter, nach Eingang-Prozedur-Funktion),
- Ausnahmen auslösen,
- Größen umbenennen

Exportschnittstelle

Exportiere Größen:

Anzahl im sichtbaren Teil vereinbarten Bezeichner (auch Komponenten) sowie

Anzahl im privaten Teil vereinbarten Bezeichner (für Kinder sichtbar) mit Aufgliederung nach

- Typen (privat - elementar - zusamengesetzt - diskriminiert - erweiterbar)
- Konstante
- Variable
- Prozeßtypen mit Anzahl Eingänge
- Pakete (generisch I nicht generisch)

- Operationen (Funktionen/Prozeduren/Eingänge, Parameterzahl und -art, Generizität)
- Ererbte Operationen (für Typerweiterungen)
- Ausnahmen

Zugreifer:

Anzahl Zugreifer per with (Bibliothekseinheiten bzw. Übersetzungseinheiten)

Anzahl der Child-pakete und Anzahl Nachkommen (transitive Hülle)

Charakterisierung der Anpaßbarkeit eines Pakets:

Anzahl generische Parameter

Anzahl exportierte parametrisierter Typen

Anzahl exportierte erweiterbare Typen

6.6.2 Kohäsionsmetriken

Die Charakterisierung der Zusammengehörigkeit erweist sich als ziemlich sprödes Problem, da es vor allem um Indikatoren für die „logische Zusammengehörigkeit" geht, Diese sind vorwiegend vom Problemverständnis bestimmt und nicht immer in sprachliche Kriterien umsetzbar.

Oft gehen Kohäsionsmaße von der inneren Struktur eines Moduls aus, sind also nur vom Modul selber abhängig, nicht von seiner Verwendung. Komplementär dazu ist die Beurteilung der Kohäsion aufgrund der Zugriffe. Ein Zugreifer soll möglichst alle exportierten Größen verwenden. Auch die quantitative Charakterisierung der Einkapselung ist von Interesse (Welch96).

Beispiele: (1)

```
package Ada.Numerics is

   pragma Pure (Numerics);

   Argument_Error: exception;

   PI : constant := 3.14159_26536_8979...;

   e  : constant := 2.71828_18284...;

end Ada.Numerics;
```

(2) $\cos(x) = \sin(x+\pi/2)$,

$\tan(x) = 1/\cot(x) = \sin(x) / \cos(x)$.

(1) faßt ein Pragma mit den Vereinbarungen zweier Konstanten zusammen. Welche sprachlichen Kriterien gibt es, um diese Paketbildung als „gut" oder als „zweckmäßig" (wozu?) zu beurteilen?

(2) gibt die mathematischen Beziehungen zwischen den Winkelfunktionen an, die man als „zusammengehörig" betrachtet. Welche Kriterien wenden wir dabei an und wie kann

man diese anhand eines Programmtexts ermitteln? Welche Funktionen soll man mit den Winkelfunktionen in ein Paket zusammenfassen?

In der Ada95-Bibliothek ist π im Paket Ada.Numerics vereinbart, die Winkelfunktionen (Kreis- und Hyperbelfunktionen mit ihren Umkehrfunktionen) im Paket Ada.Numerics.Generic_Elementary_Functions. Wie häufig kommt es vor, daß ein Zugreifer mit den Winkelfunktionen auch ihre Umkehrfunktionen oder die Hyperbelfunktionen braucht?

Kriterien für enge Zusammengehörigkeit:

- Ein Zugreifer verwendet alle importierten Größen (d. h. er muß nicht unnötige Dinge importieren).
- Alle exportierten Operationen arbeiten auf einem gemeinsamen (internen) Datenbestand.
- Alle exportierten Operationen sind Operationen auf einem in der Paketspezifikation erklärten Typ.
- (abstrakter Datentyp, mehrere Beispiele in der Ada-Bibliothek)
- - gegenseitige Aufrufe der exportierten Operationen bzw. Aufrufe von gemeinsamen internen Unterprogrammen in den Rümpfen
- hoher Kapselungsgrad (Anzahl lokale Größen und globalen Größen)

Kriterien für geringe Zusammengehörigkeit:

- Vereinbarung einer Operation für einen anderswo erklärten Typ (parent- oder with-Paket)
- Konstante/Variable, die nur in einer (anderen) Übersetzungseinheit verwendet wird
- Bildung von mehreren weitgehend unabhängigen Vereinbarungsclustern:
- (a) Sicht von außen: verschiedene Zugreifercluster verwenden nur bestimmte Teilmengen der exportierten Größen (Hammons85)
- (b) Sicht von innen: die Operationen/Methoden einer Klasse verwenden keine gemeinsamen Vereinbarungen (Chidamber94), rufen sich nicht gegenseitig auf.

Geheimnisprinzip: Die Lokalisierung einer Vereinbarung verringert die Kopplung und erhöht die Kohäsion.

Kenngrößen für die Anwendung des Geheimnisprinzips:

Anzahl sichtbare - private - lokale Vereinbarungen

Anzahl private Typen

Anzahl private Eingänge

Anzahl private Bibliothekseinheiten

Anzahl private Vorfahren für Typerweiterungen

6.6.3 Metriken für die Entwurfsqualität

Die Trennung von Spezifikation und Rumpf legt die Definition von Kenngrößen nahe, die nur von den Spezifikationen eines Programms abhängen und daher schon früh ermittelt werden können. Designmaße (die nur das Vorliegen der Modulspezifikationen voraussetzen) sind auch von Interesse als Prozeßmaße für den Fertigstellungsgrad eines Programms. Es gibt mehrere Vorschläge (z. B. (Agresti92), (Wearing92), (Hammons85), (Gannon86)); hier wird der Vorschlag von (Zage92) dargestellt, der sich nach mehreren Experimenten gut zur Identifizierung von Schwachstellen („predictor of error prone modules“) eignet. Diese auf Diana basierenden Metriken für Grob- und Feinentwurf („Architecture and Detailed Design“) unterscheiden zwischen interner und externer Entwurfsqualität.

weighted-inflow:	# data items (simple variables, records,...) passsed to the module from superordinate
or	subordinate modules.
weighted-outflow:	# data items (simple variables, records,...) passsed from the module to superordinate
or	subordinate modules.
fan-in:	# superordinate modules directly connected to the given module
fan-out:	# subordinate modules directly connected to the given module

Extern Designqualität DE: DE = weighted- inflow * weighted-outflow + fan-in * fan-out

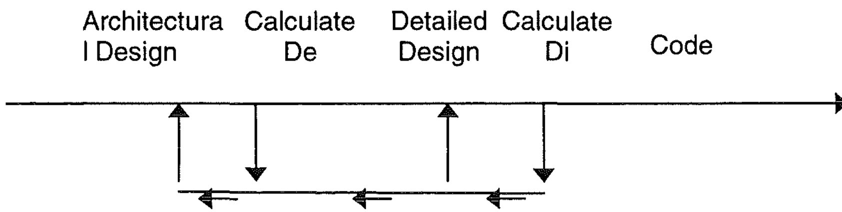
CC (central calls)	# subprogram invocations (to non-library units)
DS (data structure manipulations)	# references to complex data types (which use indirect addressing, e. g. arrays, pointers, records ...)
I/O (input/output)	# of external device addresses

Interne Designqualität DI: DI = w1 * CC + w2 * DSM + w3 * I/O
w1, w2, w3 ... Gewichte

Für einen Ada-Pprogrammentwurf G:

Designqualität D(G): D(G) = k1 * DE + k2 * DI k1, k2 ... Gewichte

DE ist zu ermitteln für den Grobentwurf („structural design“), DI für den Feinentwurf („detailed design“). Die Anwendung zur Identifizierung von Modulen, die besonderer Aufmerksamkeit bedürfen, geschieht in 3 Feedback-Schleifen:



Eignung der externen Entwurfsqualität DE (inflow * outflow + fan-in * fan-out) zur Lokalisierung von Fehlern:

(a) Universitätsprojekte, 2 - 30 KSLOC: „D(G) ... gave excellent results as a predictor of erro-prone modules.
12 % of the modules highlighted as stress points by D(G) contained 97 % of the known errors. “

(b) Projekte von Industriepartnern: „... also excellent results in relatively small projects by simply calculating the DE values. 12 % of the modules identified as stress points by DE contained 53 % of the known errors.“

(c) Anwendung auf drei Versionen (releases) eines großen Ada-Systems (533 KLOC, 680 KLOC, 843 KLOC) mit der Ermittlung von Trends für Entwurfs- und Codemaße.

	Fehlerfreie Module	Fehlerhafte Module
Anzahl Module	2070	314
DE-Mittelwert (guter Indikator)	54	355
DE-Streuung (kein guter Indikator)	375	366

6.6.4 OO-Metriken für Ada95

(Pritchett96) beschreibt die Definition folgender OO-Metriken von (Chidamber94) und (Li95) für Ada95, wobei eine Klasse als Paket mit einem erweiterbaren privaten Typ repräsentiert wird.

- WSC
  Weighted Subunits per Class - gewichtete Anzahl der Untereinheiten einer Klasse,
  Die Gewichtung erfolgt nach einem geeigneten lokalen Komplexitätsmaß.
  Annahmen:
  Klassen mit vielen Untereinheiten sind eher anwendungsspezifisch. Sie sind

schwieriger zu warten und haben größere Auswirkungen auf Unterklassen als solchen mit kleineren WSC-Werten.

- DIT
  Depth of Inheritance Tree - Tiefe einer Klasse im Vererbungsbaum
  Annahmen:
  Tiefe Klassenhierarchien sind schwerer zu warten, da Änderungen in einer Klasse sich in den untergeordneten auswirken können. Die Möglichkeit, einen erweiterbaren Typ im privaten Teil einer Spezifkation zu vereinbaren, ist hier nicht berücksichtigt.
- NOC
  Number of Children - Anzahl der Kinder (Pakete mit direkten Typerweiterungen) einer Klasse
  Annahmen:
  Hoher Wert von NOC bedeutet Allgemeinheit und höheren Wartungsaufwand, da für eine Änderung die Konsequenzen in den Unterklassen zu berücksichtigen sind.
- RFC
  Response for a Class - Gesamtzahl der Operationen (Prozeduren oder Funktionen), die potentiell in einer Klasse ausgeführt werden können, d. h. die primitiven Operationen eines erweiterbaren Typs und die von diesen direkt aufgerufenen Operationen.
  Annahmen:
  Klassen mit einem hohen RFC-Wert sind fehleranfällig und schwer verständlich.
- MPC
  Message Passing Coupling - Gesamtzahl der Aufrufe von externen Unterprogrammen
  Annahmen:
  Klassen mit einem hohen MPC-Wert sind schwer verständlich und wenig wartungsfreundlich.
- DAC
  Data Abstraction Coupling - Anzahl der Objekte mit importierten Typen („instances of other classes“) in einer Klasse
  Annahmen:
  Klassen mit einem hohen DAC-Wert sind schwer verständlich und wenig wartungsfreundlich.
- NUS
  Number of Subunits - Anzahl der Prozeduren und Funktionen, die für eine Klasse definiert sind. WSC unterscheidet sich von NUS durch die Gewichtung der Operationen.
  Annahmen:
  Klassen mit einem hohen NUS-Wert sind fehleranfällig und schwer wartbar.

Die für andere Sprachen entwickelten OO-Metriken sind meist für Ada95 anpaßbar, jedoch in den sprachlichen Kontext einzubetten und neu zu validieren (!) - Ada-Pakete haben nicht nur die Rolle eines Behälters für Klassenvereinbarungen.

Eine quantitative Charakterisierung der Objektorientiertheit eines Adaprogramms ist sicher von Interesse. So ein Profil soll die allgemeinen Charakteristika der Objektorientierung (Booch91: Abstraction, Encapsulation, Modularity, Hierarchy, Typing,

Concurrency, Persistence) einzubeziehen. Insbesondere kann die globale Programmstruktur mit der „reinen OO-Stuktur“ (Wald von Bibliothekseinheiten, jede enthält die Vereinbarung eines Typs mit seinen primitiven Operationen) verglichen werden.

Auch Soll-Vorstellungen wie das sog. Law of Demeter (Lieberherr88) eignen sich zur Identifikation und Auswertung von Metriken. Dieses schränkt die Typen der in einer Methode zu verwendenden Operationen ein und besagt für Ada95, daß in einer primitiven Operation O eines Typs T nur Operationen des Typs T selbst, der Typen der Argumente von O (einschließlich der Objekte, die von O oder aufgerufenen Operationen generiert werden, und von globalen Variablen) und der Typen der lokalen Variablen von P zu verwenden sind.

Mögliche Kenngrößen (mit geeigneter Normierung):

Anzahl von erweiterbaren, private und anderenTypen,

Pakethierarchien („Waldmaße“) und ihre Zugriffsbeziehungen, Anzahl der von einem Paket exportierten Typen, Anzahl der exportierten Namen der „Nicht-OO“-Schnittstelle.

Kapselungs-, Kopplungs- und Kohäsionsmaße (s. 6.2)

Auch die Prozeßstruktur und die Kommunikation zwischen Prozessen ist relevant. Nach E. Schonberg kann die Prozeß- und Kommunikationshierarchie eines Ada(83)-Programms als eine ideale Realisierung des Objektmodells aufgefaßt werden.

6.7 Charakterisierung der Prozeßstruktur

Das hier relevante Systemmodell ist die Menge der Prozesse (Ada95: und der geschützte Objekte) eines Adaprogramms mit ihren statischen und dynamischen Beziehungen. Die statische Struktur (Vereinbarungen und ihre Sichtbarkeit, Aufruf- bzw. Zugriffsbeziehungen, Informationsflüsse) ist eingebettet in die Sichtbarkeitsstruktur eines Programms, die dynamische Struktur (wesentlich für Performancebetrachtungen) ist nicht Gegenstand der statischen Programmanalyse. Da eine Prozeßvereinbarung nicht in der Schnittstelle der sie enthaltenden Bibliothekseinheit stehen muß, sind andere Darstellungen für die Ermittlung von Prozeßstrukturmetriken zu wählen.

(1) Ein Systemmodell, das die (aktiven und passiven) Objekte einer Anwendung mit ihren Kommunikationsbeziehungen modelliert (z. B. die RTS-Datenflußdiagramme in (Nielsen86)). Nach (Nielsen86) und anderen Empfehlungen für Realzeitanwendungen soll dieses Systemmodell dieses Modell zur Prozeßidentifikation dienen, erst die daraus abgeleitete Prozeßstruktur zur Abgrenzung von Paketen.

(2) Ein detaillierteres Entwurfsdokument auf der Abstraktionsebene von Buhr-Diagrammen bzw. Pseudocode.

Neben den - analog zur Ermittlung des fan-in und fan-out zu ermittelnden - Kenngrößen für die Kommunikationsstruktur und deren Interpretation bezüglich Fehleranfälligkeit

oder Wartbarkeit gibt es auch andere Eigenschaften, für deren Ermittlung die Analyse der Kommunikationsstruktur nützlich ist, z. B.

- Identifizierung von potentiellen Verklemmungen
- Identifizierung von Kommunikationsengpässen, z. B. Warteketten bei Relaisprozessen (Nielsen-Shumate)
- Verwendungsart von geschützten Objekten (Taxonomie, Bewertung)
- Identifizierung von passiven Prozessen, die sich bei der Migration von Ada83 nach Ada95 als Kandidaten für geschützte Objekte eignen.

6.8 Zusamenfassung: Statische Programmanalyse und dynamische Modelle

Die statische Analyse führt zu Kennzahlen für die Prognose des Programmverhaltens, die auf unvollständigen Informationen aufbauen (zunächst eine Spezifikation, dann ein kompletter Programmtext ohne die Charakterisierung der Umgebung und ihres dynamischen Verhaltens), wegen ihrer frühen Verfügbarkeit jedoch nützliche Informationen für die Qualitätsprognose bzw. -bewertung und die Projektführung bilden.

Während die Untersuchung von generischen Ausprägungen noch zur statischen Analyse gehört, kann diese über die Anzahl der Objekte eines Typs, die Anzahl der Prozesse und ihr Kommunikationsverhalten, Datenraten bei Ein-Ausgabe, Interaktionsmuster zwischen Partitionen oder zwischen Programm und Prozeßumgebung wenig aussagen, obwohl die rechnergestützte statische Analyse die Konstruktion von recht detaillierten Programmmodellen erfordert, die auch als Basis für Performance-Vorhersagen und Simulation (d. h. für die Prognose des dynamischen Verhaltens) erforderlich sind.

Solche Wünsche gehen weit über den plausiblen Anwendungsbereich der statischen Analyse hinaus, die immer wieder nur Indikatoren liefern kann, Hinweise auf mögliche Probleme. Diese erfordern kompetente und zielorientierte Interpretation und können so zur Vermeidung von Pannen und Fehlentwicklungen beitragen. Nach vielen Fallstudien für Ada und andere Sprachen scheinen früh verfügbare Entwurfsmaße geeignet zur Identifizierung von sog. „hot spots", die besondere Sorgfalt bei der Implementierung und Qualitätssicherung erfordern.

Messen ist kein Selbstzweck. Die Vorgabe und Messung bzw. Überprüfung von quantitativen Größen-, Struktur- und Qualitätsattributen sind unentbehrliche Hilfsmittel für die Projektführung und die Produktbewertung. Sie

- beeinflußt das Verhalten der Entwickler,
- ist ein Leistungsnachweis,
- ist unentbehrlich für die Projektsteuerung und Qualitätssicherung,
- verbessert nicht vorliegende Arbeitsergebnisse, kann jedoch den Anstoß für Verbesserungen geben,
- kann die Auslieferung mangelhafter Produkte verhindern.

6.9 Literatur

{Agresti92} Agresti,W., Evanco, W.: Projecting Software Defects From Analyzing Ada Designs. IEEE Transactions on Software Engineering, Vol. 18.11, November 1992, S. 988 - 997

{Anderson92}Anderson, J.: Managing Ada OO development. Ada-Europe'92, Springer-LNCS-603

{Basi94} Basili, V., Caldiera, G., Rombach, H.: The Goal/Question/Metric Paradigm. In: Marciniak, J. (ed.): Encyclopedia of Software Engineering. Wiley, 1994, S. 528 - 532

{Boehm89} Boehm, B., Royce, W.: Ada COCOMO and the Ada Process Modell. Proceedings, Fifth COCMO Users' Group Meeting, Software Engineering Institute, Pittsburgh, 1989

{Boehm95} Boehm, B., Clark, B., Horowith, E., Westland, C., Madachy, R., Selby, R.: Cost models for future software lifecycle processes: COCOMO 2.0. Annals of Software Engineering, Vol 1. (1995) S. 57 - 94

{Booch91} Booch, G.: Object Oriented Design with Applications. Benjamin/Cummings, Redwood City, 1991

{ Chidamber94} Chidamber, S., Kemerer, C.: A Metrics Suite for Object-Oriented Design. IEEE Transactions on Software Engineering, Vol. 20.6, 1994, S. 476 - 493

{ Constantine 74} Constantine, L., Maers, G., Stevens, W.: Structured Design. IBM Systems Journal, Vol. 13, 1974, S. 115 - 139

{Dumke97} Dumke, R., Foltin, E.: Quality Assessment of Object-Oriented Software Development Methods. Metrics News, Vol. 2.2, December 97, S. 33 - 76

{Erwin96} Erwin, J.: Choosing a Language for Maintainable Software. Ada Letters, Vol. 16.1 (January 1996)

{Fenton91} Fenton, N.: Software Metrics - A Rigorous Approach. Chapman & Hall, London, 1991

{Firesmith88} Firesmith, D.: Mixing Apples and Oranges or what is an Ada Line of Code anyway? Ada Letters, Vol. 8.5, September 1988, S. 110 - 112

{Gannon86} Gannon, J., Katz, E., Basili, V.: Metrics for Ada Packages: An Initial Study. CACM Vol. 29.7, July 1986, p. 616 - 623

{Hammons85} Hammons, C., Dobbs, P.: Coupling, Cohesion, and Package Unity in Ada. Ada Letters, Vol. 4.6, June 1985, p. 49 - 59

{Jones95} Jones, C.: Backfiring. Converting Lines of Code to Function Points. Computer, Vol. 18.11, Nov. 1995

{Li95} Li, W., Henry, S., Kafura, D., Schulman, R.: Measuring Object-Oriented Design. Journal of Object-Oriented Programming, Vol. 8.4, July 1995

{Lieberherr88} Lieberherr, K., Holland, I., Riel, A.: Object-Oriented Programming: An Objective Sense of Style. OOPSLA'88, S. 323 - 334

{Maxwell96} Maxwell, K., Wassenhove, L., Dutta, S.: Software Development Productivity of European Space, Military, and Industrial Applications. IEEE Trans.on Software Engineering, Vol. 22.10, Oct. 96, S. 706-718

{Mayrand96} Mayrand, J., Coallier, F.: System Acquisition Based on Software Product Assessment. ICSE'18, Berlin, S. 210-219

{McCabe76} McCabe, T.: A Complexity Measure. IEEE Transact. on Software Eng., Dec. 76, S. 308 - 320

{Nielsen86} Nielsen, K., Shumate, K.: Designing Large Real-Time Systems with Ada. CACM, Vol 30.8, Aug. 1986, S. 695 - 715

{Pritchett96} Pritchett, W.: Applying Object-Oriented Metrics to Ada95. Ada-Letters, Vol 16.5, Sept. 1996

{Reifer89} Reifer, D.: Economics of Ada: Estimation and Control. 4. Deutscher Anwendercongress, München, April 1989

{Schwald97} Schwald, A.: Metrics, Poeple, and Their Roles in a Software Project. Metrics News, Vol. 2.2, December 97, S. 15 - 22

{Waligora97} Waligora, S. et al.: The Impact of Ada and Object-Oriented Design in NASA Flight Dynamics Division. Ada Letters May 1997

{Wearing92} Wearing, A.: Software Engineering, Ada and Metrics (Ada-Europe'92 Springer-LNCS-603)

{Welch96} Welch, L., Lankala, M., Farr, W., Hammer, D.: Metrics for quality and concurrency in object-based systems. Annals of Software Engineering, Vol. 2, 1996, S. 93 - 113

{Woods96} Woods, S., Qiang, Y.: The Program Understanding Problem: Analysis and a Heuristic Approach. 18th ICSE, Berlin, 1996, S. 6 - 15

{Zage92} W. Zage, D. Zage: Architecture and Design Metrics (Ada-Europe'92, Springer-LNCS-603)

{Zeigler97} Zeigler S.: Comparing Development Costs of C and Ada. Ada Europe News, Jan. 1997

HotAda - Möglichkeiten zur Qualitätssicherung bei der verteilten Ada95-Programmierung

7 HotAda - Möglichkeiten zur Qualitätssicherung bei der verteilten[7] Ada95-Programmierung

Patrick Closhen, Hans-Jürgen Hoffmann
Technische Universität Darmstadt,
Fachbereich Informatik
Fachgebiet Programmiersprachen und Übersetzer

Kurzfassung

Ein häufig auftretendes Problem bei der Systemprogrammierung ist das Erfüllen von Qualitätsansprüchen an die zu erstellenden Produkte. Ein Weg, qualitativ hochwertige Software zu schaffen und zuzusichern, ist es, das Einhalten von Stilrichtlinien, die je nach Programmiersprache und Anwendungsgebiet unterschiedlich gegeben sein können, bereits innerhalb des Entwicklungszyklus zu überprüfen, um so für ihre Durchsetzung zu sorgen. Die Berichte dieser Überprüfung können als Beleg für qualitätssicherndes Arbeiten herangezogen werden.

Der von uns vorzustellende Weg, dies zu erreichen, ist ein Satz von Werkzeugen zum:

- Erfassen von Stilrichtlinien in Abhängigkeit von Programmiersprache und Anwendungsfall zu einem Satz von einzuhaltenden Regeln (oder auch Constraints),
- Aufbereiten von Quellcode für die durchzuführende Überprüfung durch Erstellung einer speziell angepaßten Objektstruktur und
- Durchführen der Überprüfung auf der geschaffenen Objektstruktur.
- Bei inhaltlichen Fragen während der Überprüfung soll der Programmierer interaktiv eingreifen können.

Der Schwerpunkt des vorzustellenden Projekts liegt auf der Programmiersprache Ada95, weshalb die Grundlage für die Überprüfung die Stilrichtlinien des DoD in ‘Ada95 Quality and Style: Guidelines for Professional Programmers’ [Ada95QS] sind.

[7] Wir übernehmen hier den Titel, unter dem wir den Beitrag eingereicht haben, lassen aber den Teil des Projekts, bei dem es um die Verteilung geht, aus. Bei Interesse bitte Rückfragen an die Autoren.

7.1 Einleitung

Die ständige Leistungssteigerung bei modernen Computersystemen und die damit einhergehende zunehmende Komplexität der auf ihnen laufenden Software-Systeme schürt schon längere Zeit die Warnungen vor einer Software-Krise. Diese, so heißt es, würde bei der fortschreitenden Anwendung von Computertechnik in immer kritischeren Umgebungen (Beispiele: Flugverkehr-Überwachungssysteme, Medizintechnik, Bankwesen) die Anfälligkeit für Programmierfehler und die damit verbundenen Risiken derart erhöhen, daß ihr Einsatz nicht mehr tragbar ist.

Deshalb ist die Unterstützung zur Qualitätssicherung bei der modernen Systemprogrammierung ein immer wichtigerer Bestandteil im Software-Entwicklungszyklus, besonders in Bereichen, aus denen schon immer der Ruf nach Validierung und/oder Verifikation zu hören ist.

Vor allem bei Programmiersprachen, deren Leistungsumfang sehr groß ist, sei es durch eine Vielzahl von angebotenen Sprachkonstrukten, oder durch immer häufiger zur Verfügung stehende Programmierbibliotheken. Hier ist eine Unterstützung des Entwicklers, der meist im Team arbeitet, sowohl bei der Code-Erstellung als auch bei der Integration seiner Arbeit in das Gesamtprojekt, unverzichtbar, nicht zuletzt dann, wenn es um die Abnahme eines Produktes durch den Auftraggeber geht.

Ein Ansatz, der fast schon Standard bei solchen Programmiersprachen ist, ist das Ausarbeiten von Programmierstil-Richtlinien, resultierend aus gesammelten Programmiererfahrungen oder firmeninternen Vorgaben. Diese bilden dann die Grundlage für jegliche Software-Entwicklung. Allerdings sind die Entwickler bei der Umsetzung dann meist nur ihrem Gewissen verpflichtet, eine strikte Überprüfung ist aufgrund fehlender automatischer Unterstützung oder zu geringem Budget für Qualitätssicherung im Entwicklungszyklus eher selten, das Resultat: Die Richtlinien werden ignoriert.

Um diesem Mangel abzuhelfen, soll das Werkzeug HotAda Programmquelltexte auf Stilrichtlinien überprüfen, Mängel aufdecken, diese ggfs. automatisch korrigieren und einen Prüfbericht erstellen, der für den Nachweis der qualitätssichernden Arbeit benutzt werden kann.

7.2 Qualitätserfordernisse

Ausgangspunkt für das Konzeptionieren von qualitätssichernden Werkzeugen ist eine Untersuchung und Aufstellung der Anforderungen, auf die man Quelltexte überprüfen will. Hierbei zeigt sich, daß es einige allgemeingültige Anforderungen an die Programmierung gibt, die sich unabhängig von der letztendlich benutzten Programmiersprache formulieren lassen und nur im Detail angepaßt werden müssen. Weiterhin läßt sich eine Kategorisierung der Richtlinien bzgl. ihrer Umsetzbarkeit, ihrer Vorgehensweise und ihrer Automatisierbarkeit aufstellen. Wir wollen dies im weiteren für die Punkte Quelltext-Formatierung, Lesbarkeit und Kommentare, sowie Programmstruktur exemplarisch einzeln erläutern.

Quelltext-Formatierung

Die Quelltext-Formatierung ist eine größtenteils leicht automatisierbare Anforderung, da sie hauptsächlich von der Syntax abhängig ist und nur in wenigen Fällen auf die Semantik von Quellcode zurückgreifen muß. Durch sie erreicht man eine konsistente Darstellung, somit bessere Lesbarkeit und höhere Wartbarkeit.

Beispiel für Quelltext-Formatierung:

Schlechte Variante:

```
procedure  Display_Menu(Title  :  in  String;  Options  :  in
Menus; Choice :
                        out Alpha_Numerics);
```

Bessere Variante:

```
procedure Display_Menu (Title   : in String;
                        Options : in Menus;
                        Choice  : out Alpha_Numerics);
```

Abbildung 2: Quelltext-Formatierung

Lesbarkeit und Kommentare

Unter diesem Punkt sollen weitere Aspekte der Quelltextdarstellung angesprochen werden, die über eine einfache syntaxbasierte Quelltextformatierung hinausgehen, da hier bereits die Semantik des Quelltextes eine Rolle spielt. Ein gutes Beispiel ist hier die Bezeichnerwahl: Obschon ein Werkzeug über die Ableitung eines Bezeichners aus der Anwendungsdomäne nicht urteilen kann, so kann doch immerhin auf eine konsistente Schreibweise für jeden Bezeichner geprüft werden, wenn der jeweilige Compiler, wie bspw. im Falle von Ada nicht zwischen Groß- und Kleinschreibung unterscheidet. Hierbei liegt natürlich die Entscheidung beim Programmierer, für welche Variante er sich entscheidet, wobei das Werkzeug einen „guten“ Vorschlag machen kann.

Kommentare sind zweifellos ein mächtiges Hilfsmittel, wenn es um Dokumentation und spätere Wartbarkeit geht. Das beste Beispiel sind hier in Kommentare eingebettete Schnittstellenbeschreibungen von Prozeduren oder Funktionen. Sie können aber auch für das Aufteilen von Quelltext in geeignete Abschnitte, z.B. das Abteilen von Deklarationsteilen, Unterprogrammen, Funktionen, etc. für eine verbesserte Lesbarkeit

eingesetzt werden. In diesem Falle ist sogar ein automatisiertes Einfügen von Blanko-Kommentaren denkbar.

Programmstruktur

Unter diesem Punkt sind sowohl Aspekte wie Aufteilung von Übersetzungseinheiten auf Dateien, bspw. beim Erstellen einer Programmbibliothek, als auch Beziehungen zwischen Einheiten und/oder Untereinheiten in der Konzeption eines Programmes zu nennen, oder auch bereits innerhalb von Blöcken der Umfang von Vereinbarungen. Dies sind alles Punkte, die nur teilweise von einem Werkzeug selbständig überprüft werden können. Nichtsdestotrotz soll auch hier auf feststellbare Verstöße gegen Stilrichtlinien, wie bspw. das Deklarieren von Variablen an der falschen Stelle im Programmtext, fehlende Kapselung oder das Überschreiten einer vorher festgelegten zulässigen Schachtelungstiefe, eingegangen werden. Durch den starken Einfluß der Programmsemantik auf dieser Anforderungen ist ggfs. das interaktive Eingreifen des Programmierers erforderlich.

Die hier aufgeführten Qualitätserfordernisse stellen nur einen Ausschnitt aus einer noch nicht vollständig untersuchten Menge dar. Schon jetzt aber läßt sich ausmachen, daß man in einem Werkzeug, welches eine Aussage über die Qualität von Programmen machen soll, nach Kategorien, wie „ist automatisierbar“, „ist als Regel formulierbar“, „benötigt Interaktion des Programmierers“ und weiteren Bedingungen unterscheiden und einordnen können muß.

Für eine Übersicht der Stilrichtlinien, die im Rahmen unseres Projektes untersucht werden, seien [Skub96], [Lieberh88] und [Ada95QS] als Quellen genannt.

7.3 Allgemeine Betrachtung der Vorgehensweise

Aus der von uns eingeführten Kategorisierung lassen sich nun mehrere Vorgehensweisen allgemein ableiten, für die dann im nächsten Abschnitt eine konkrete Umsetzung skizziert werden soll.

1. Initialisierung des Werkzeuges

- Parameter für Quelltext-Formatierung festlegen
- Formulieren von Überprüfungsregeln in der „Wenn-Dann“-Form und Anknüpfen an Sprachkonstrukte
- Weitere Abstimmungen in Abhängigkeit der jeweiligen Sprache

2. Syntaxbasiertes Überprüfen des Quelltextes

- Konsistente Formatierung

3. Anlegen einer Bezeichnertafel

- Konsistente Bezeichnerschreibweise

4. Erzeugen eines Konstruktobjekt-Baumes für semantische Überprüfungen

- Sprachkonstrukte werden auf Objekte abgebildet
- Einfache Überprüfungen können direkt auf den erzeugten Konstruktobjekten durchgeführt werden

5. Der Konstruktobjekt-Baum wird zum Abarbeiten der Überprüfungsregeln traversiert, um gemäß den Regeln Zusammenhänge aufzuspüren und zu analysieren, evtl. Rückgriff auf die Bezeichnertafel. Dabei können für eine Überprüfung mehrere Konstruktobjekte involviert sein
6. Der Programmierer wird, wenn erforderlich, zu interaktivem Eingreifen aufgefordert
7. Erzeugen eines Prüfprotokolls
8. Abspeichern des überprüften und ggfs. veränderten Quelltextes

Offensichtlich lassen sich einige Arbeitschritte unabhängig von anderen durchführen (bspw. die Quelltext-Formatierung, die Bezeichnerüberprüfung, etc.). Deshalb kann der Überprüfungsvorgang nach Wunsch zusammengestellt werden und bspw. nach Deaktivieren der Arbeitsschritte, die ein Eingreifen des Programmierers erfordern, auch von der Kommandozeile aus gestartet und das Prüfprotokoll zu Dokumentationszwecken aufbewahrt werden. Inkrementelles Vorgehen ist, zumindest derzeit, noch nicht vorgesehen, weshalb eine Versionierung, z.B. der angelegten Objekte, entfallen kann.

7.4 Konkrete Umsetzung durch die Werkzeuge

Um die oben genannten Arbeitsschritte umzusetzen, bedienen wir uns des an unserem Fachgebiet entwickelten Anwendungsrahmens HotDoc für das Erstellen zusammengesetzter Dokumente [HotDoc1], wobei der Schwerpunkt auf der zur Verfügung stehenden Oberfläche liegt.

Das HotDoc-System, implementiert in VisualWorks Smalltalk, ermöglicht das Zusammenstellen von Dokumenten aus Bausteinen, die vom Benutzer mit Hilfe einer Klassenbibliothek frei programmierbar sind, wobei einige Bausteine bereits im Rahmen der Entwicklung entstanden sind (Textverarbeitung, Vektorgrafik, Geschäftsgrafik, Tabellenkalkulation)[HotDoc2]. Diese Freiheit kann sehr flexibel genutzt werden, bspw. um interaktive Benutzungsschnittstellen schnell herzustellen. Die Bausteine lassen sich auf einer Arbeitsfläche plazieren und können ineinander geschachtelt werden, wodurch sich eine Hierarchie von Bausteinen mit der Arbeitsfläche als Wurzel ergibt.

Bei der Programmierung von Bausteinen stehen dem Benutzer ein Reihe von Leistungsmerkmalen bereits zur Verfügung: Jeder Baustein besitzt ein für ihn typisches Menü, eine Werkzeugleiste, die Möglichkeit, Lineale im Rahmen der Arbeitsfläche darzustellen und die Statuszeile als Ausgabebereich zu nutzen. Bausteine können aus dem Systemmenü der Arbeitsfläche ausgewählt und initialisiert werden. Diese Initialisierung besteht hauptsächlich aus der Wahl eines Gestalters für Kindbausteine, die in den Baustein eingefügt werden sollen. Diese Gestalter erlauben die Positionskontrolle über die vom Baustein beanspruchte Darstelllungsfläche und ordnen eingefügte Bausteine automatisch an.

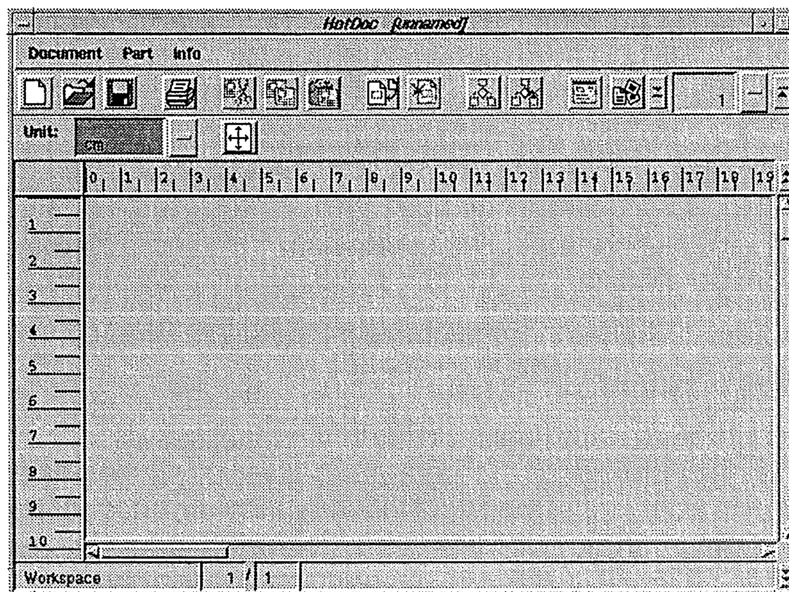


Abbildung 3: Die HotDoc-Arbeitsfläche

Zusätzlich besteht die Möglichkeit, daß mehrere Bausteine untereinander kommunizieren können, d.h. sie können Teile ihrer Daten austauschen. So wird bspw. die dynamische Anbindung des Bausteins Geschäftsgrafik an die Daten des Bausteins Tabellenkalkulation erreicht.

Diese Eigenschaften von HotDoc-Bausteinen, ihre leichte Programmierbarkeit, ihre Fähigkeit, per Gestalter die Position automatisch festzulegen, und die Kommunikation zwischen Bausteinen, ermöglichen es dem Entwickler, seinen Anwendungsfall schnell und effizient in HotDoc zu implementieren, indem er neue, angepaßte Bausteine entwirft. Und auf diesem Umstand fußt unser Ansatz: Konzipiert sind ein Baustein für das Parsen von Quelltext, ein Baustein für die Bezeichnertafel, Bausteine für die Objekte, die die Sprachkonstrukte kapseln sollen, und nicht zuletzt die Überprüfungsregeln selbst innerhalb eines Bausteins oder in komplexen Fällen selbst als Baustein. Diese Komponenten werden dann unter die Regie eines Systembausteins gestellt, über welchen dann die Voreinstellungen vorgenommen werden können, und der die Interaktion des Programmierers mit den betreffenden Bausteinen koordiniert.

Der soeben beschriebene Ansatz schließt aufgrund der Tatsache, daß er auf ein integriertes System aufbaut und hoch interaktiv ausgelegt ist, natürlich die Anwendung von der Kommandozeile aus. Da aber dies nur ein erster Prototyp eines qualitätssichernden Werkzeugs sein soll, wird dies in einer zweiten Fassung, die danach in Ada95 reimplementiert werden soll und auf einige Eigenschaften der HotDoc-Oberfläche verzichten wird, machbar sein. Damit erreicht man dann auch eine bessere Einbettung unseres Werkzeugs in den Entwicklungszyklus, da der Aufruf an den Übersetzungsvorgang selbst angehängt werden kann. Bisher ist diese Einbettung ein wenig umständlich, da das System, auf dem HotAda laufen wird, stark von der Ada Entwicklungsumgebung getrennt ist (VisualWorks [ParcPlace] und ObjectAda von Aonix [Aonix]) und so hohe Anforderungen an die Entwicklungsumgebung stellen. Auch hier wird die spätere Reimplementierung Abhilfe schaffen.

7.5 Zusammenfassung und Ausblick

Ein Werkzeugansatz für die Überprüfung von Programmquelltexten auf die Einhaltung von Stil- und Programmierrichtlinien wurde vorgestellt. Dieser Ansatz berücksichtigt dabei sowohl allgemeine Programmierempfehlungen, unabhängig von Programmiersprachen, als auch spezielle Richtlinien, die sich aus den jeweiligen Entwicklungsumgebungen ergeben können. Es wurde eine Aufteilung dieser Richtlinien nach verschiedenen Kriterien auf Arbeitsschritte vorgeschlagen und eine geplante Umsetzung dieser Arbeitsschritte mit Hilfe des HotDoc Anwendungsrahmens dargestellt.

Kommende Arbeiten werden sich mit der genauen Konzipierung der einzelnen Bausteine befassen. Dazu ist folgendes zu leisten:

- Eine Zusammenstellung von Stilrichtlinien für den Prototypen muß ausgearbeitet werden, um alle geplanten Überprüfungen, also syntaktische, semantisch basierte und per Interaktion unterstützte zumindest exemplarisch umgesetzt zu haben.
- Die drei genannten Überprüfungsmethoden müssen auf hohe Flexibilität hin ausgelegt, um den Anspruch, später auch für andere Sprachen einsetzbar zu sein, zu erfüllen.
- Das Generieren von Protokollen sowie die Ausgabe von formatiertem Quelltext muß ausgearbeitet werden.
- Die Einbettung des Werkzeugs in den Entwicklungszyklus muß auf seine Akzeptanz hin untersucht werden.

7.6 Literatur

[Sku96] Skubliczs et al.: Smalltalk with Style, Prentice Hall 1996

[Ada95QS] Ada 95 Quality and Style: Guidelines for Professional Programmers, SPC-94093-CMC Version 01.00.10 October 1995

[Lieberh88] K. Lieberherr, I. Holland, A. Riel: Object-Oriented Programming: An Objective Sense of Style, OOPSLA 1988 Proceedings, S. 323-334

[HotDoc1] Jürgen Buchner: HotDoc - Ein flexibles System für den kooperativen Aufbau zusammengesetzter Dokumentstrukturen, Dissertation Tech. Universität Darmstadt, Fachbereich Informatik, 1998

[HotDoc2] Jürgen Buchner und Martin Hauck: HotDoc Programmierhandbuch, Diplomarbeit Tech. Hochschule Darmstadt, Fachbereich Informatik, 1997

[Kratzer96] K.P. Kratzer: Ada - Eine Einführung für Programmierer, Hanser Verlag 1996

[ParcPlace] Für weitere Informationen zum Thema VisualWorks siehe folgende URL:

http://www.objectshare.com/

[Aonix] Für weitere Informationen zum Thema ObjectAda siehe folgende URL:

http://www.aonix.com/

Safety, Concurrency and Testing

8 Ada, Concurrency and a Safety Critical Subset

George Romanski
Aonix
200 Wheeler Road, Burlington MA 01803. USA

Overview

A simple monitoring system is described to provide some examples of concurrent code. The monitoring system is written in a subset of Ada and uses Tasking . Elements of this tasking based system are compared with algorithms written using cyclic schedulers. The elements are then compared to show the advantages of the task based approach for safety critical applications. The Ravenscar Subset is then described together with some additional restrictions which are used to make the monitoring program predictable.

8.1 Introduction to the Brewing process.

The “Recirculating Infusion Mash” (RIM) is a recent technique adopted by some home-brewers as a method of controlling the parameters during the first stage (mashing) of the brewing process.

There are usually three temperature steps during the mashing process which influence enzyme extraction, sugar extraction and sugar conversion. The times and temperatures affect the sweetness, the strength and the “body” of the beer. There are many other variables at the disposal of the brewer, such a selection and quantities of malted grain, hops and other additives but these static variables are much easier to control.

The RIM unit consists of a vessel, a re-circulating pump, and a heater. During the mashing process, fluid is recirculated through a grain bed, at increasing temperature steps for various duration's. A 15 gallon RIM unit has been constructed, which is linked to a computer to measure, display and record the temperatures.

The monitoring software uses a subset of Ada specifically selected for Safety Critical applications. It conforms to a subset of the Tasking System defined by the Ravenscar Profile. A runtime system which supports this profile has been developed and is called **Raven**. The monitoring program, reads information from sensors and accepts commands from the brewer. Alarms are sounded if the temperatures stray outside of the specified bands. The brewer can change settings while brewing, but these changes must be incorporated through well defined transitions and should not interfere with the display and recording process.

The brewer acts on information displayed and alarms sounded. The information may be mis-interpreted, reaction delays may cause instabilities and so on. Although the system is not safety critical from the operators point of view, it may be hazardous for the beer as enzymes may become de-natured, malt may be caramelized etc., resulting in poor

yield and low quality beer. Although the RIM system is manually controlled (direct control is planned as a future project) Ada code is used to measure, display, record and sound alarms.

8.2 The Monitoring System

In this implementation, the connection of computer to external devices is through a very simple interface. Simple character I/O is used to communicate between system and user, and simple serial I/O through an RS232 port connects to a temperature sensor. The Ada.Text_IO packages are not used as there is often inadequate evidence for their certification.

All user commands are input one character at a time using the Get_Immediate function. The user inputs information in response to menu prompts and is restricted to the digits of a telephone keypad (i.e. 0 .. 9, *, #). Commands are simple numeric directives, and all numeric character sequences are checked and converted to their corresponding values at source.

The command processing module allows the user to input various system parameters, and to input, display and edit the brewing profile required.

Once the brew profile is complete the brew monitor is started. The brew monitor simply reads a temperature sensor periodically and checks this against the required temperature. If the temperature is outside the expected profile an alarm is sounded. A log of expected against actual temperatures is displayed against the brewing timeline.

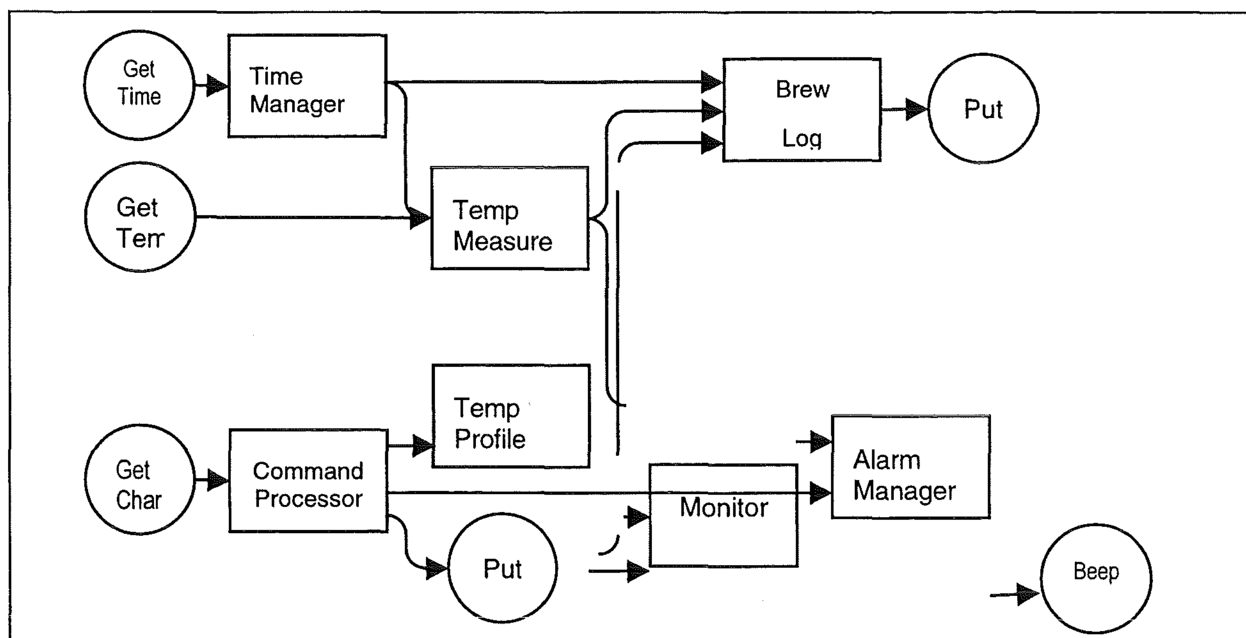


Figure 1: Brew Monitoring System

8.2.1 Command Processing

The command processing system needs to be responsive. When characters sequences are typed, the command processor analyses the characters as they are typed in. It is important to respond quickly as typing command inputs with a delay greater that half a second becomes disconcerting. Verification of characters, values and commands is performed during the character input, i.e. do not wait until a command is completely input but respond directly. The command menu is as follows:

```
Menu

                    1 Input Temperature Schedule
                   2 Display Temperature Schedule
                    3 Clear Temperature Schedule
                           4 Clear Step
                       5 Set Alarm on or off
                           6 Time Now
                       7 Display Actual, Run
```

The first four selections manage the brewing profile. Selection five manages the alarm. Selection six is required so that the brewer can initialize the wall clock timer. As this is an embedded system, there is no assumption about the availability of a battery driven clock. (as typically supplied on a Personal Computer via the BIOS). Selection seven puts the system into monitoring and display mode. While the system is running, it is still possible to display the menu and to modify the brewing parameters.

8.2.2 Brew Profile Manager

The brew profile consists of a series of steps, each with a duration and a temperature. A typical profile is shown in the table 1 below:

Step	Duration in Minutes	Expected Temperature	Time	Description
1	8	150 °F	10:00	Enzyme extraction
2	6	157 °F	10:08	Starch extraction
3	50	162 °F	10:14	Sugar conversion
4	10	170 °F	11:04	“Mash Out” (kill enzymes)
5			11:14	

Table 1: Typical Brew Profile

In a traditional cyclic executive style implementation, this table would be simply drive a loop which was repeated as a certain rate. The number of iterations would be a function of the tick rate and the duration for a particular step. This type of implementation places an overhead on the processor which effectively interrogates the

profile manager a large number of times depending on the tick frequency and length of duration. Two implementations are shown in figures 2 and 3. One shows the implementation uses a cyclic coding style and the other uses a task with an event timer.

Note that the code using events reflects the problem more naturally, is easier to understand and lies dormant during a temperature step. It is invoked only during the temperature transitions and at the precise moment that a transition is required. The polling example has code which is repeated at a frequency determined by the size of tick. The higher the requirement for timing precision, the higher the repetition rate.

```
Counter := Tick * Duration;

procedure Duration_Manager is –

begin -- check step first

if Counter < =0 then

  Counter := Profile(Step+1).Duration*Tick;

  Expected := Profile(Step).Temperature;

  Step := Step + 1;

else

  Counter := Counter –1;

end if;
```

```
task Duration_Manager is

---

for Step in 1 .. Last_Step loop

  Expected := Profile(Step).Temperature;

  delay until Profile(Step+1).Time;

end loop;

-- finished
```

8.2.3 Temperature/Time Manager

The temperature measuring device is an inexpensive sensor which measures and converts to a digital value. The device is controlled and the temperature values are transmitted through a standard serial port. Measurement and conversion takes two seconds (slow in the hard real time control community but fast enough for brewing).

The temperature is measured at the optimum frequency of the input device. As each value is read it is time stamped and the Temp/Time record are manipulated as a single object. A filtering algorithm performs some smoothing to reduce the effects of spurious values.

```
protected body T_Stamper is
  entry Stamp_T (Temp : Temp_Range) when Available is
  begin  -- time stamping the temperature is an indivisible operation
    TT.Temp := Integer(FT(FT(Temp)*0.7) + FT(FT(TT.Temp) * 0.3));
                -- work in fixed point to reduce drift
    TT.Time := Time_Manager.Brew_Time_Now;
  end Stamp_T;
```

```
function T_Stamped return Temp_Time is
begin
  return TT;  -- note that TT is a private type
end T_Stamped;
```

Figure 4: Protected body

The measurement is controlled by a periodic task which calls T_Stamper at the appropriate rate. The body of T_Stamper is an indivisible operation on the Temp/Time record, so the values are always consistent with each other. Other algorithms in the process use the pair of values in their computations which allows the computations to be performed at different rates based on time of data arrival rather than the time that the computation is performed.

Mutual exclusion is provided automatically through the declaration of the protected object. Other languages provide low level mutual exclusion primitives which could be used to provide similar functionality, but the risk of missing or mismatching these calls to such primitives should not be overlooked. Code of the protected body is executed at a priority which is higher than any of the tasks that can call it. Priority inversion, where a number of tasks which use a shared resource cause unnecessary blocking, will not occur.

8.2.4 Brew Monitor

The Brew monitor obtains expected and actual temperatures throughout the brewing process. Both values change over time. This periodic task checks that the actual temperature is within an acceptable margin of the expected temperature. An alarm is signaled as necessary to draw attention to the corrective actions necessary. Two Alarm functions are used Sound_Alarm and Stop_Alarm. As the temperature changes less than two degrees per minute a period of 20 seconds provides adequate responsiveness.

8.2.5 Alarm Manager

The alarm is sounded by printing a Beep character to the keyboard periodically. The period of the beep is determined by the value of Beep_Delay, and continues until Continue_Beep is set false. When the alarm is off, the task consumes no processing resources polling a flag to determine change of status. While the alarm is sounding, a value is Continue_Beep is polled, but the task suspends on a timer event to establish intervals between beeps.

```
task body Intermittent_Beep is
begin
  loop
    Suspend_Until_True (Beep);
    while Continue_Beep loop
      Put (Ada.Characters.Latin_1.BEL);  -- Beep
      Beep_Delay := Beep_Delay + 3.0; -- seconds
      delay until Beep_Delay;
    end loop;
  end loop;
end Intermittent_Beep;
```

Figure 5: Beeper Task

The control of the Alarm system is through the two procedures shown below.

```
procedure Sound_Alarm is
begin
  Continue_Beep := True;
  Set_True(Beep);
end Sound_Alarm;


procedure Stop_Alarm is
begin
  Continue_Beep := False;
end Sound_Alarm;
```

Figure 6: Alarm Controls

The two procedures are called from the Brew_Monitor task and also from the command line interface, to test the alarm mechanism and to switch it off manually once attention has been drawn to the brewer.

8.2.6 PRINT Log Control

The printing system writes a graphic chart on a screen using positions of simple characters to show expected and actual temperature over time.

```
task body Log_Printer is
begin
  Suspend_Until_True(Print_Log_Start);
  loop
    Suspend_Until_True(Print_Log_Now);
    if Print_Continuously then
      Set_True(Print_Log_Now);
    end if;
    Put_Temp_Line(Expected, Actual, Brew_Time);
    Delay_Log (15.0);
  end loop;
end Log_Printer;
```

Figure 7: Printer Task

The task is controlled by two events, one to start the printing process, and one to control when it is to occur, so that it does not interfere with the command processor menu interactions. While the task is waiting for these events it does not poll, it is simply linked to a suspension object. The printing cycle can be broken by setting Print_Continuously to false. The task will then suspend and wait until Print_Log_Now is set to true.

8.3 Comparison between Cyclic and event driven implementation

The actions to implement the brew monitoring algorithm are summarized in the chart below. The repetition times are shown in seconds, and an '@' sign indicates that the time to perform the action depends on the time of a preceding action.

In a cyclic scheduler implementation a number of counters must be maintained, each counter is used to determine when an action is to take place. The granularity of the real time system is determined by the repetition rate of the main loop. The system is governed not by the natural frequencies of the system level events, but by the time taken to execute the object code for each of the individual actions. As the complexity of the system increases, the use of a simple cyclic scheduler becomes more difficult. The computational algorithms have a direct impact on the time at which the data is processed.

Cyclic schedulers often result in data which is stored and manipulated in global memory, increasing the difficulty of showing the coupling between code and data. In an event driven system, data is made available to the code that uses in immediately. “Freshness” of the data is maintained and the underlying system is more responsive.

In a task periodic implementation, timing behavior is predictable, and the sensitivity to variations in time is constrained.

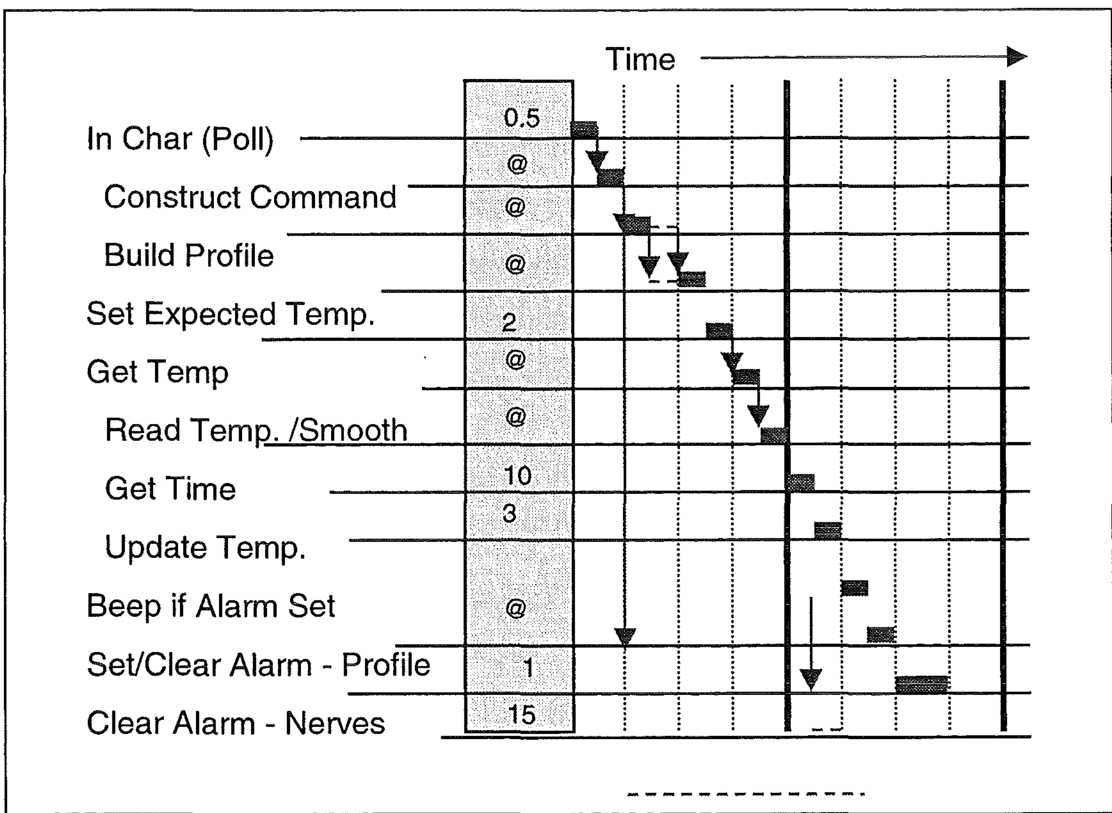


Figure 8: Brew monitoring actions

8.4 Ravenscar Profile

The Ravenscar Profile [ ], may be enforced a number of different ways by the compilation system. One approach which has been used on a runtime system specifically designed to enforce a safety critical subset [Raven] is to use pragma Restrictions.

Ada 95 defines a number of configuration pragmas. Once these pragmas are introduced to the program library they will cause the compiler to enforce their effect on subsequent compilation units. The following pragmas are supported:

8.4.1 Pragmas for Tasking Restrictions which affect Static Configuration

No_Task_Hierarchy	Tasks cannot be nested, they must be declared in library level package. This enable the compiler and linker to allocate space for these tasks in global memory and not rely on a dynamic heap.
Static_Storage_Size	The task code size is dependent of the actual code written. The size of the stack for each task is fixed at link time. Users may specify a size for each task individually. If a size is not provided a system level default size will be used. By the time all of the tasks are ready to start, all of their memory will be allocated and will not be extended at run time.
No_Task_Allocators	Task objects cannot be allocated on the heap.
No_Task_Attributes	Attribute require the use of heap, which is to be avoided.
Max_Tasks => <n>	The maximum number of tasks must be provided by the user. As a program is started, space is allocated to accommodate the information required for all of the tasks which may be declared. If <n> is set to zero, then there are no tasks in the program. The tasking system can be totally eliminated and will not be loaded with the application.

8.4.2 Pragmas for Tasking Restrictions which constrain Dynamic Behaviour

No_Abort_Statements	Tasks cannot not be aborted. They are expected to run forever, but they may be permanently blocked by suitable language constructs.
No_Task_Termination	Tasks cannot not be terminated. Tasks are expected to consist of loops which continue for the duration of the program.
No_Dynamic_Priorities	Priorities cannot be changed dynamically.
Static_Priorities	Priorities must be compile time constants.
Max_Task_Entries => 0	Task Rendezvous are not supported at all. The Rendezvous construct implements complex real time scheduling semantics which are difficult to verify for correctness.

No_Select_Statements	Not required as Rendezvous entries are not supported.
Max_Protected_Entries =>1	A Protected Object may only have one entry. This simplifies the run-time system and makes the application easier to follow. In effect only one call to write is supported.
Max_Entry_Queue_Depth => 1	A protected entry may be blocked while it waits for an event. This restriction request the run-time system to check that there is at most one task waiting for this event and to use a simple pointer rather than a queue of waiting tasks.
Boolean_Entry_Barrier	A check of a Boolean variable may be performed by the run-time system directly. An expression would require user code to be executed during this scheduling operation, hence it is not allowed.
No_Requeue	This simplification makes the run-time implementation more predictable hence easier to certify.

8.4.3 Pragma Restriction to constrain event Management

No_Dynamic_Interrupts	Interrupt handlers must be linked to their corresponding vectors and cannot be changed at run time.
No_Local_Protected_Objects	Protected Objects must be declared in library level packages, which allows the run-time to set up data structures which do not need modification at run time.
No_Protected_Type_Allocators	Protected Objects cannot be allocated on the heap.
No_Exception_Handlers	The standard Ada exception handling mechanisms are typically not used in safety critical systems. If Max_Tasks => 0 then a single default handler is provided at the global level. Any exception raised will transfer control directly to the global handler where user provided routine takes control. If tasks are present, then an exception raised will transfer control directly to a handler associated with the current task. Each task has a single global handler. By default the handler includes a suspension object on which the task is suspended. Users may precede this with code to change execution modes, unblock standby tasks and so on.
No_Relative_Delay	The delay statement is not allowed. Delay until is allowed.

8.4.4 Additional Non-Tasking Restrictions

The following restrictions are not part of the Ravenscar profile, but are included to ensue that the object code is predictable.

No_Standard_Storage_Pools	The usual heap mechanisms are not permitted as they cause storage fragmentation and cause program behavior to be less predictable. Implementation defined storage pools may be used to allow a restricted form of allocation and deallocation which is fast, and predictable.
Discard_Names	Enumeration strings are not generated by the compiler. If enumeration strings are required they must be defined by the user.
No_Enumeration_Maps	Representation clauses may be provided for enumeration types but the compiler will not store the values in the form of a table. This means that certain operations (e.g. 'PRED) are not supported for these types if representation clauses are given.
No_Nested_Finalization	Certain objects require runtime code to be executed when they go out of scope. This finalization code is invoked automatically by the compiler and the effects of this are not directly visible to the user. Such object declarations are only permitted at the global level. This means that they do not go out of scope unless the program terminates. This means that the finalization code will be excluded at link time, and does not disrupt predictability.

8.4.5 Standard Packages

Package Calendar is not supported. Package Ada.Real_Time must be used instead. This provides a higher resolution clock and operations with which to manage it.

The Ada I/O packages are not supported as they provide a lot of functionality which is inappropriate in an embedded real-time system. A simple I/O package which transmits characters, integers and strings through a serial port replaces the fully functional standard Ada package.

Software-Entwicklung mit Ada bei Dasa/RI

9 Software-Entwicklung mit Ada bei Dasa/RI

Erfahrungen aus der Sicht eines Entwicklers

Franz Kruse
Dasa, RIO 64, Bremen
Franz.Kruse@ri.dasa.de

Seit 1988 wurde bei ERNO Raumfahrttechnik (heute Daimler-Benz Aerospace Raumfahrt-Infrastruktur) das *Columbus Ground System* (CGS) entwickelt und damit die ersten Erfahrungen mit Ada als Entwicklungssprache gewonnen. Der Vortrag gibt einen kurzen Überblick über das System, schildert selbstkritisch die Erfahrungen und Probleme mit Ada aus der Sicht eines betroffenen Entwicklers und stellt dar, dass dem hohen Anspruch der Ada-Philosophie in der Praxis oft enge Grenzen gesetzt sind.

9.1 Das Projekt

Europa ist an der geplanten internationalen Raumstation mit dem Projekt *Columbus* beteiligt. Von dem anfänglich wesentlich ambitionierteren Vorhaben, das aus mehreren Komponenten bestand, ist heute die *Columbus Orbital Facility* (COF), ein an der Raumstation fest angedocktes Labormodul, übrig geblieben. Das *Columbus*-Projekt, das 1987 von der *European Space Agency* (ESA) an die europäische Industrie vergeben wurde, wird von Daimler-Benz Aerospace Raumfahrt-Infrastruktur (Dasa/RI) in Bremen als Hauptauftragnehmer geführt. Firmen aus mehreren europäischen Ländern sind mit Unteraufträgen beteiligt.

Ein erheblicher Software-Anteil am *Columbus*-Projekt wurde unter dem Namen *Columbus Ground System* (CGS) seit etwa 1988 ebenfalls unter Leitung von Dasa/RI unter Beteiligung von Unterauftragnehmern aus mehreren europäischen Ländern entwickelt. Für die Software-Entwicklung war Ada vorgeschrieben. Für Dasa/RI, damals noch ERNO Raumfahrttechnik, war dies die erste Erfahrung mit Ada.

9.2 Das Produkt

CGS ist ein integriertes System verschiedener Softwarewerkzeuge, das auf den komplexen Lebenszyklus der *Columbus Orbital Facility* ausgerichtet ist. Es unterstützt den gesamten Bereich von der Anforderungsanalyse über Simulation fehlender Hardware- und Softwarekomponenten, Entwurf, Implementation, Systemintegration, Verifikation bis zum Betrieb.

- CGS läuft auf verteilten heterogenen Hardware-Plattformen: Sun-Workstations für die Benutzerschnittstelle, die Softwareentwicklungsumgebung und den Datenbankserver, HP-Rechnern für Echtzeitkomponenten und Force-Rechnern für Simulation.

Es gibt auch eine (allerdings nicht offizielle) CGS-Version für eine homogene Sun-Solaris-Umgebung.

Die wesentlichen Komponenten sind

- die Konfigurations- und Missionsdatenbank

  Die Datenbank ist das Herz des Systems, sie speichert alle Konfigurationsdaten, Messstellenbeschreibungen, Prozeduren, Displays, Simulationsmodelle sowie Testdefinitionen und Testresultate. Sie hat eine interaktive Benutzerschnittstelle und Softwareschnittstellen zu allen Komponenten des Systems und erlaubt geographisch verteilte Daten-Definitionen.

- zentrale Mensch-Maschine-Schnittstelle
- VICOS (*Verification, Integration and Checkout Software*)

  mit Teilkomponenten für Test-Vorbereitung, Test-Konfiguration, Test-Ausführung und Test-Auswertung

- Software-Entwicklungsumgebung

  Diese beinhaltet die Unterstützung des gesamten Lebenszyklus zur

  - Entwicklung von Ada-Komponenten, die in das System eingebunden werden können
  - Entwicklung von Test- und Betriebsprozeduren sowohl für Boden- als auch Bordbetrieb in der Columbus-eigenen Sprache UCL (*User Control Language*), der interaktiven Kommandiersprache HLCL (*High Level Command Language*) und der Skriptsprache CPL (*Crew Procedure Language*). Sie beinhaltet das *Columbus-Sprachsystem* mit integrierten Editoren, Compilern und Interpretern.
  - Entwicklung von interaktiven Bedienoberflächen (*Synoptic Displays*) für Boden- und Bordanwendungen

- Simulationssystem

  Es beinhaltet eine graphische Modellentwicklungsumgebung sowie eine interaktive Modellausführungsumgebung.

- Prozedurale Schnittstellen (API) zur Einbindung weiterer Software, sowie spezielle prozedurale Schnittstellen (RPI) zur Einbindung verteilt laufender Software. Über diese Schnittstellen wurde z. B. eine Hardwareanbindung an den Systembus und globale Datenspeicher an Bord realisiert.

Darüber hinaus hat CGS Schnittstellen zur Anbindung verschiedener kommerzieller Softwaresystem, z. B. MatrixX.

9.3 Einige Projektzahlen

CGS besteht aus ca. 2,5 Millionen Programmzeilen. Die Entwicklungskosten betrugen etwa 120 MAU (*million accounting units*) = ca. 230 Millionen DM. Zu Spitzenzeiten waren ungefähr 80 Mitarbeiter in Europa am Projekt beteiligt, davon etwa 50 bei ERNO/Dasa-RI, allerdings sind die Mitarbeiterzahlen über den gesamten Entwicklungszeitraum betrachtet sehr schwankend. Insgesamt waren Firmen aus Deutschland, Belgien, Norwegen, Frankreich, Spanien und Dänemark in unterschiedlichem Umfang beteiligt.

9.4 Heutiger Stand der Entwicklung

CGS wurde inzwischen in mehreren Versionen fertiggestellt und ausgeliefert. Es ist in mehr als 20 Systemkonfigurationen bei mehreren Kunden im Einsatz: NASA, Boeing, McDonnel Douglas Aerospace, der russischen Raumfahrtbehörde RKA, Fokker Space, Matra Marconi Space und der niederländischen Raufahrtbehörde NLR. Weitere mögliche Einsätze sind geplant bei der japanischen Raumfahrtbehörde NASDA, Mitsubishi Heavy Industries und Dornier.

Die aktuelle Entwicklung ist darauf ausgerichtet, erkannte Probleme zu beheben und das System um neue Anforderungen aus dem Einsatz bei Kunden zu erweitern.

9.5 Entwicklungsumgebung, Konfigurationsmanagement

Die Entwicklung der Teilkomponenten des Systems wird in mehr oder weniger unabhängigen Teams, zum Teil im Hause, zum Teil bei Unterauftragnehmern im europäischen Ausland, durchgeführt. Nach Fertigstellung der Komponenten werden diese von einem separaten Team ins Gesamtsystem integriert, getestet und zur Auslieferung bereitgestellt.

Um während der Entwicklung jedoch die vielfältigen Schnittstellen der Softwarekomponenten untereinander zu befriedigen, müssen jedem Team alle anderen Komponenten, zu denen Abhängigkeiten bestehen, verfügbar sein. Ursprünglich wurden die Entwicklungsteams als gänzlich eigenständige Einheiten betrachtet, die sich gegenseitig, gemäß ihrer Abhängigkeiten, ihre Software lieferten, und zwar als formalen Vorgang auf Band mit Lieferschein. Dieses Verfahren wurde auch zwischen den Teams im Hause praktiziert. Jedes Team installierte dann für sich jeweils die angelieferte Software, um sie mit ihren eigenen Modulen zu integrieren.

Dieses Verfahren erwies sich aber ziemlich bald als unpraktikabel, und es wurde mit der Einrichtung eines selbstentwickeltenen *Central Repository* begonnen, in dem alle Teilprodukte zentral gehalten werden. Lieferungen zwischen den Teams entfallen. Das Repository besteht aus

- einer Verzeichnisstruktur, die die Teilprodukte und ihre Abhängigkeiten reflektiert,
- den darin eingebetteten Quelltexten aller Softwaremodule und den gemäß ihrer Abhängigkeiten untereinander verknüpften Programmbibliotheken der einzelnen Teilprodukte,
- SCCS zur Versionsverwaltung,
- einer Menge von Shellskripts zur Verwaltung des Repository.

Die Entwicklungsumgebungen der einzelnen Teams sind mit dem Repository so verknüpft, dass jedes Team sein Produkt als mit den übrigen Produkten integriert betrachten kann. Erst mit Lieferung in das Repository wird die neue Version des jeweiligen Produkts für andere Teams sichtbar.

Erst in jüngster Zeit wurde das selbstentwickelte Repository durch *ClearCase* (ein kommerzielles System) ersetzt. Die CGS-spezifischen Bedürfnisse werden durch einen wiederum selbstentwickelten Aufsatz, die PDB (*Project Database*) realisiert.

Eine wesentliche Komplexität, sowohl beim alten *Central Repository* als auch beim neuen *PDB/ClearCase*, besteht darin, dass Teile der Software für unterschiedliche Plattformen (Sun/Solaris, HP/HPUX) konfiguriert werden müssen.

9.6 Probleme bei der Softwareentwicklung

CGS ist ein gutes, vielleicht sogar ein besonders krasses Beispiel dafür, dass dem hohen Anspruch der Ada-Philosophie in der Praxis enge Grenzen gesetzt sind. Die hier mit Ada gemachten Erfahrungen sind, gemessen an den hohen Erwartungen, eher enttäuschend. Dies hat seinen Grund allerdings weniger in der Sprache Ada selbst als vielmehr in verschiedenen Randbedingungen und Begleitumständen der Softwareentwicklung im Raumfahrtbereich. Die durch die Sprache gebotenen Möglichkeiten sind nicht adäquat genutzt worden. Einige der wesentlichen Ursachen sind durch Vorgaben von Auftraggeberseite bedingt, andere sind jedoch durchaus hausgemacht.

❑ **geographisch verteilte Entwicklung**

Von der ESA vergebene Aufträge werden grundsätzlich nach einem strengen Länderschlüssel geographisch verteilt. Im Interesse der Förderung internationaler Zusammenarbeit ist dies sicher zu begrüßen, der Qualität des Produkts ist diese Verteilung jedoch nicht unbedingt zuträglich, insbesondere im Softwarebereich. Abgesehen von erhöhten Entwicklungsaufwänden werden oft unnötige und unnatürliche Schnittstellen geschaffen, die den Gesamtaufwand für die Software in technisch unnötiger Weise erhöhen. Da die verteilt entwickelten Teilprodukte sich weitgehend einer zentralen Kontrolle entziehen, entwickeln sie sich leicht heterogen und tragen zu einem inhomogenen Gesamtentwurf bei. Gemeinsame Konzepte lassen sich nur schwer verwirklichen.

❑ **unvorbereiteter Projekteinstieg**

Die Verwendung von Ada wurde von der ESA gefordert, auf Entwicklerseite wurde es eher als unerwünscht und aufgedrängt empfunden. Weder ERNO noch die beteiligten Firmen waren auf ein größeres Projekt mit Ada vorbereitet. Verschwindend wenige Mitarbeiter hatten überhaupt Ada-Kenntnisse, Projekt-Erfahrungen mit Ada waren in keinem Fall vorhanden. Insbesondere bei einigen Unterauftragnehmern wurde der plötzliche Personalbedarf durch Einstellungen von unerfahrenen Berufsanfängern ohne Ada-Kenntnisse gedeckt. Die im Rahmen des Projekts eilends durchgeführten Ada-Kurse konnten bestenfalls Grundwissen verschaffen, mangelnde Erfahrung jedoch nicht ersetzen.

Dies hatte unmittelbare Auswirkungen auf den Softwareentwurf. Oft wurde nach altem Denkschema (Fortran, C, Pascal) entwickelt. Höhere Ada-Konzepte wurden missverstanden, inadäquat verwendet oder schlicht ignoriert.

❑ **HOOD**

Als Entwurfsmethode war von der ESA HOOD (*Hierarchical Object Oriented Design*) gefordert worden. Bei Projektbeginn steckte die Methode noch in den Kinderschuhen und wurde parallel zum Projekt mitentwickelt. Erfahrungen damit gab es nicht. Anfänglich wurde HOOD als die Allheilmethode schlechthin vertreten, inzwischen herrscht bei allen Beteiligten Einigkeit darüber, dass die Verwendung der Methode der Qualität der Software mehr geschadet als genützt hat.

HOOD schränkt die Verwendung von Ada-Konzepten zum Teil drastisch ein und führt, streng angewendet, zu fragwürdigen Entwurfsmustern:

– Generische Einheiten können nur eingeschränkt verwendet werden.

- Automatische Initialisierungsteile von Paketen sind nicht erlaubt, Pakete müssen durch expliziten Prozeduraufruf initialisiert werden.
- Typen und zugehörige Operationen müssen in separate Pakete getrennt werden.
- Die HOOD-eigene Sprache erzwingt altmodische Einschränkungen, z. B. müssen, wie in altem Standard-Pascal, Konstanten vor Typen deklariert werden.
- Pakete können keine Pakete in ihrer Spezifikation enthalten.
- Variablen sind in Paketspezifikationen nicht erlaubt.

Beispielsweise wäre das Ada-Standardpaket TEXT_IO aus HOOD-Sicht ein miserabler Entwurf, da es alle oben angeführten Einschränkungen missachtet.

Darüber hinaus beinhaltet HOOD ein Konzept zur automatischen Codegenerierung. Die erzeugten Ada-Quelltexte sind allerdings von *Good Ada Style* weit entfernt.

Von verschiedenen Entwicklern wurde HOOD unterschiedlich streng verwendet. Einige bedienten sich nur der graphischen Symbolik, um die Architektur ihrer Software grob darzustellen, und erlaubten sich zum Teil weitgehende Freiheiten, um die HOOD-Einschränkungen zu umgehen. Im Laufe der Jahre wurden solche Verstöße immer weniger geahndet. Andere Entwickler waren weniger freizügig. Insbesondere einige Unterauftragnehmer hatten ihre Teilprodukte streng nach HOOD-Vorgaben entworfen und programmiert und sogar die HOOD-Codeerzeugung penibel von Hand nachgezogen.

❑ Konzept Wiederverwendbarkeit vollständig ausgeklammert

Das Konzept, unabhängige und wiederverwendbare Softwaremodule zu entwickeln, wurde im Columbus-Projekt vollständig ausgeklammert. Einerseits war es von der formalen Projektstruktur her nicht vorgesehen, teamübergreifende gemeinsame Pakete zu entwickeln und zu pflegen. Andererseits wurde dieses Konzept, obwohl von einigen Entwicklern energisch eingefordert, von den leitenden *Engineers* ebenso energisch bekämpft, ausgelöst durch die Angst, einen zusätzlichen formalen Aufwand zu produzieren und zusätzliche Abhängigkeiten zwischen den Entwicklungsteams zu schaffen, insbesondere zu den externen Auftragnehmern. Zudem wurde befürchtet, die bewußte Auslegung von Paketen mit der Zielsetzung Allgemeinheit und Wiederverwendbarkeit würde zusätzliche, unkontrollierte Aufwände schaffen.

Zwar wurden sporadisch in einzelnen Teams wiederverwendbare Pakete entwickelt, die gute Kandidaten für gemeinsam zu nutzende Module gewesen wären - aber erst relativ spät im Projekt, begünstigt durch veränderte personelle Verantwortlichkeiten, wurde ein Vorstoß unternommen, diese Pakete zentral zu sammeln und zumindest allen Teams zugänglich zu machen.

❑ heterogene Sprachschnittstellen

Eine besondere Problematik, die sich unmittelbar aus der Verwendung von Ada ergibt, tut sich bei Verwendung externer Softwaresysteme auf. Die Schnittstellen zu solchen Systemen sind überwiegend C-basiert und erfordern zum Teil umfangreiche Adaptionen. CGS hat Schnittstellen zu

- *UNIX* als dem unterliegenden Betriebssystem
- *Oracle* als zentraler Missionsdatenbank

- *XView* als X-Window-Oberfläche mehrerer CGS-Komponenten
- *Dataviews* zur Visualisierung von Messdaten und zur Erstellung interaktiver Bedienoberflächen (*Synoptic Displays*)

In diesem Zusammenhang muss auch erwähnt werden, dass ein wesentlicher Teil des Simulators in *Smalltalk* entwickelt wurde, das mit dem übrigen, in Ada geschriebenen Teil überhaupt keine direkte Aufrufschnittstelle zulässt. Diese Schnittstellen sind daher auf UNIX-Prozessebene durch Sockets realisiert.

Zusätzlich zur Komplexität der Schnittstellen traten im Zusammenspiel von XView und Dataviews untereinander und mit dem Ada-Laufzeitsystem noch andere Unverträglichkeiten auf, z. B. bei der Verwendung von UNIX-Signalen. Im Fall eines Teilprodukts traten diese Probleme so massiv zutage (allerdings gepaart mit anderen gravierenden Mängeln in Entwurf und Programmierung), dass entschieden wurde, die Komponente in C komplett neu zu entwickeln. Andere Schnittstellenprobleme gab es auch im Zusammenspiel unterschiedlicher Hardware/Software-Werkzeugen, z. B. mit dem VME-Shared-Memory des HP-RT-UNIX, die zum Teil umfängliche Entwurfsanpassungen notwendig machten.

Der lange Entwicklungszeitraum führte auch dazu, dass *state-of-the-art*-Konzepte zwischenzeitlich überholt wurden, und unsere Entwicklungsarbeiten antizyklisch zur kommerziellen Software-Industrie verlaufen.

❑ **Probleme mit Entwicklungsumgebungen**

Für Sun-Plattformen wurde Verdix-Ada, für HP-Plattformen Alsys-Ada verwendet. Beide Compiler-Systeme zeigten gravierende Mängel, die teure Unterstützung vom Hersteller notwendig machten und zum Teil trotz Unterstützung nicht befriedigend gelöst werden konnten.

9.7 Erfahrungen

Die wohl wesentlichste Erfahrung war, dass Ada kein Wundermittel ist, das automatisch zu besserer Software führt. Ungeschickt verwendet kann es sogar gegenteiligen Effekt haben. Softwareingenieure mit guter Ada-Erfahrung sind unabdingbar und müssen bei Projektbeginn zur Verfügung stehen.

Die Entwicklung von CGS bot hinreichend Gelegenheit, massive Erfahrungen mit Ada und der Entwicklung komplexer Systeme in Ada zu sammeln. Heute sind bei Dasa/RI eine Reihe hochqualifizierter Softwareingenieure und Ada-Spezialisten verfügbar, die insbesondere auch vielfältige Erfahrungen zum Zusammenspiel heterogener Softwaresysteme (Ada, UNIX, Oracle, XView usw.) vorweisen können. Diese Erfahrung war allerdings schmerzhaft und teuer.

Einsatz von Ada im Experimentellen Führungsinformationssystem EIGER

10 Einsatz von Ada im Experimentellen Führungsinformationssystem EIGER

Gerhard Bühler
Forschungsgesellschaft für angewandte Naturwissenschaften
Forschungsinstitut für Funk und Mathematik
Abteilung Informationstechnik und Führungssysteme
Neuenahrer Straße 20
53343 Wachtberg-Werthhoven
e-mail: buehler@fgan.de

10.1 Einleitung

Das experimentelle Führungsinformationssystem auf der Grundlage eines Rechnernetzes (EIGER) realisiert wesentliche Funktionen eines Führungsinformationssystems für Stäbe und Hauptquartiere der Bundeswehr. Dies bedeutet, daß das System die Informations- und Verarbeitungsanforderungen einer großen Organisation mit einem breit gefächerten Aufgabenspektrum zu befriedigen hat. Die Organisation besteht wiederum aus vielen Organisationseinheiten, die weit über ein großes Gebiet verteilt sein können. Die einzelnen Organisationsteile haben dabei in der Regel eine begrenzte Aufgaben- und Datenvielfalt, wobei sich jedoch die zu bewältigenden Aufgaben und die benötigten Daten für mehrere Organisationseinheiten überschneiden können.

Neben der kompletten Informationsverarbeitung, also der Verarbeitung, Speicherung, Darstellung und Übertragung von Daten soll das System EIGER auch die folgenden Anforderungen befriedigen:

- Ausfallsicherheit
- Lastverbund
- Funktionsverbund
- Integration von Diensten wie E-Mail, Textverarbeitung usw.
- Zugang zu Fremdsystemen

Bei dem System EIGER handelt es sich nicht um ein operationelles System, sondern es ist ein Experimentalsystem, das derzeit vor allem als Testbett für Funktionen dient, die im Rahmen des ATCCIS-Projektes (Army Tactical Command and Control Information System) der NATO als deutscher Anteil benötigt werden. Im Augenblick steht hier das Thema Replikation von Daten in heterogenen Systemen im Vordergrund.

Das System ist fast vollständig in Ada 83 geschrieben. Es ist aber soweit geändert, daß es von einem Ada 95-Compiler übersetzt werden kann. Einige wenige Teile sind in C

geschrieben. Als Betriebssystembasis wird derzeit UNIX verwendet. Eine Portierung auf Windows NT ist vorgesehen. Grundsätzliche Überlegungen für ein Redesign unter dem Aspekt der Verwendung von Ada 95 wurden durchgeführt.

Zunächst werden in den Abschnitten 10.2 bis 10.4 die Struktur des Gesamtsystems sowie die seiner Komponenten beschrieben. In Abschnitt 10.5 wird dargestellt, wie diese Struktur mit Hilfe der Sprachmittel von Ada 83 in Software umgesetzt wurde. Dabei wird auch von den Schwierigkeiten zu berichten sein, die sich hierbei ergaben. Weiterhin werden im Abschnitt 10.7 noch einige Überlegungen für eine Neustrukturierung des Systems unter Verwendung von Ada 95 vorgestellt.

Im Abschnitt 10.8 wird eine Übersicht über den ATCCIS-Replikationsmechanismus und seine Implementierung in EIGER gegeben.

10.2 Struktur des Gesamtsystems

Die Disloziierung der Organisationsteile der Bundeswehr und die oben genannten Anforderungen an das System erfordern eine verteilte Implementierung des Systems. Mit Sicht auf die Hardware soll das System aus einer Reihe von Rechnern als Verarbeitungsknoten, den ***Teilsystemen***, bestehen, die durch ein Kommunikationssystem miteinander verbunden sind. Dabei werden über die Rechner und deren Betriebssysteme, sowie die Art der Vernetzung nur wenige Annahmen gemacht. Die Rechner von Organisationseinheiten, die auf den selben Daten arbeiten, werden in der Regel jedoch über ein lokales Netzwerk miteinander verbunden sein. Die Teilsysteme einer Organisationseinheit bilden einen ***Bereich***.

Jedes Teilsystem besteht selbst wieder aus einer Reihe von Prozeßsystemen. Im Mittelpunkt steht dabei das ***Kernsystem***, das die zentrale Steuerungsfunktion eines Teilsystems enthält und die eigentliche Verarbeitungskapazität eines Teilsystems zur Verfügung stellt. Diese Komponente ist vollständig in Ada 83 geschrieben. Auf den Aufbau dieser Komponente wird später noch ausführlicher eingegangen.

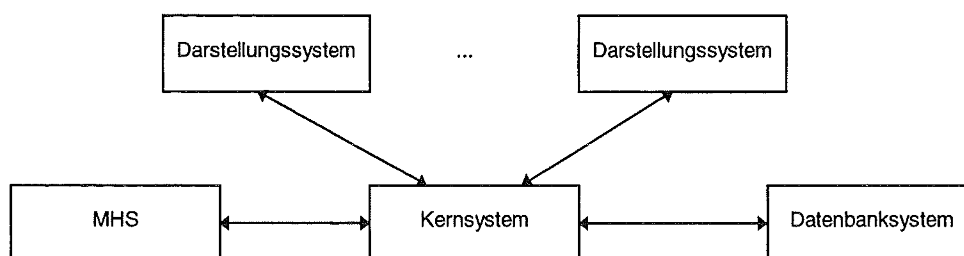


Abbildung 10-1: Struktur eines Teilsystems

Jedes Teilsystem ist ein Teilhabersystem, d.h. viele Benutzer können gleichzeitig mit dem System kommunizieren. Der Zugang zum Teilsystem wird dem Benutzer über das ***Darstellungssystem*** EMMI (EIGER Man Machine Interface) möglich gemacht, das durch ein spezielles Prozeßsystem realisiert wird. Jedem Benutzer ist dabei ein solches

Darstellungsssystem zugeordnet. Das Kernsystem erzeugt bzw. verarbeitet die Daten, die von den Darstellungssystemen dargestellt werden bzw. von ihnen als Ergebnis von Benutzeraktionen angeliefert werden. Diese Darstellungssysteme und das Kernsystem von EIGER brauchen nicht auf dem selben Rechner abzulaufen, aber natürlich müssen die Rechner durch ein irgendwie geartetes Kommunikationsnetz miteinander verbunden sein. Die derzeitige Implementierung sieht hier allerdings nur die Kommunikationsmöglichkeit über TCP/IP vor. Ein Darstellungssystem bietet dem Benutzer eine MOTIF-Oberfläche. Da zu Beginn des Projektes kein stabiles MOTIF-Binding für Ada zur Verfügung stand, ist die Darstellungskomponente zum überwiegenden Teil in der Sprache C programmiert.

Die Kernsysteme aller Teilsysteme eines Bereiches haben Zugriff zu einer relationalen Datenbank, in der die für diese Organisationseinheit relevanten Daten gespeichert sind. Der Zugriff erfolgt aber nicht direkt, sondern über ein spezielles Prozeßsystem. Dieser Ansatz wurde aus mehreren Gründen gewählt. Der „normale" Zugang zu einem relationalen Datenbanksystem über „embedded SQL" ist nicht geeignet, weil das Kernsystem bei jedem Dienstaufruf in einen Wartezustand geraten würde, bis der Dienst, die Ausführung einer SQL-Anweisung, vollständig erbracht ist. Diese Zeit sollte aber für andere Aktivitäten im Kernsystem zur Verfügung stehen. Ein weiterer Grund, der gegen diesen Ansatz mit „embedded SQL" spricht, liegt darin, daß im Kernsystem mehrere Datenbanktransaktionen parallel ablaufen können. Dies ist mit der Funktionalität von „embedded SQL" nicht steuerbar.

Jedem Kernsystem ist ein ***Data Base Service Distributor*** (DBSD) zugeordnet, der die auszuführenden SQL-Anweisungen entgegennimmt. Jeder dieser Anweisungen ist eine Identifikationsnummer zugeordnet, die eindeutig eine Transaktion kennzeichnet.

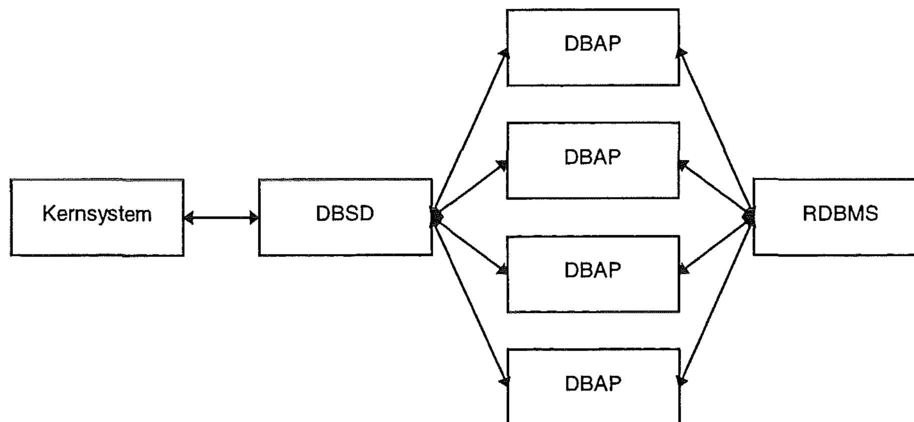


Abbildung 10-2: Anschluß des RDBMS an das Kernsystem

Handelt es sich um eine neue Transaktionsnummer, wird ein ***Data Base Application Program*** (DBAP) als Sohn erzeugt. Andernfalls wird das für diese Transaktionsnummer zuständige DBAP festgestellt. Diesem Sohn wird über eine UNIX-Pipe die SQL-Anweisung übergeben. Das DBAP seinerseits reicht die SQL-Anweisung an das Datenbanksystem weiter und wartet, bis die Ergebnisse vom Datenbanksystem angeliefert worden sind. Die Ergebnisse werden nun in eine spezielle Form gepackt und wiederum über eine UNIX-Pipe dem DBSD zur Verfügung gestellt, der sie schließlich beim Kernsystem abliefert.

Außerdem steht in jedem Bereich ein ***Message Handling System*** nach X.400 zur Verfügung, das von jedem Teilsystem des Bereiches aus erreichbar ist. Auch der Zugriff auf die Funktionen des Message Handling Systems erfolgt nicht direkt aus dem Kernsystem heraus. Dabei wird ein spezielles Prozeßsystem zwischen Kernsystem und Message Handling System geschoben. Die Gründe dafür sind im wesentlichen mit denen identisch, die oben für die Datenbankkomponente beschrieben wurden.

Legende:
MUCO: Mail-UI2MS-Connection
UI2MS: User Interface to Message Store
MS: Messge Store

Abbildung 10-3: Anbindung des MHS an das Kernsystem

Im Gegensatz zur Datenbank nutzen aber nicht alle Teilsysteme eines Bereiches das MHS, da immer genau ein Teilsystem eines Bereiches die „normalen“ Mail-Funktionen für den gesamten Bereich, also Versenden von Mail und Zustellen von empfangener Mail an die Empfänger, und ein zweites Teilsystem alle Mail-Funktionen im Zusammenhang mit dem ATCCIS-Replikationsmechanismus übernimmt. Dabei ist nicht ausgeschlossen, daß beide Funktionen von einem einzigen Teilsystem wahrgenommen werden.

Der ***Message Store*** bietet im wesentlichen die folgenden fünf Funktionen:

- **bind** - exklusives Belegen eines Briefkastens.
- **list** - Inspizieren eines Briefkastens nach verschiedenen Gesichtspunkten.
- **submit** - Abliefern einer Mail zum Versand.
- **get** - Abholen von Mail nach verschiedenen Gesichtspunkten.
- **unbind** - Aufgabe der Belegung des Briefkastens.

Genau diese Funktionen liefert der Prozeß ***UI2MS*** über eine einfache zeichenorientierte Schnittstelle an. Der Prozeß ***MUCO*** benutzt diese Schnittstelle, um im Auftrag eines Kernsystems Mail vom MS abzuholen bzw. beim MS zum Senden abzugeben. Dieser Prozeß MUCO übernimmt damit auch das unerwünschte Warten, bis der *MS* das Ergebnis einer Operation mitteilt und verhindert somit das Warten des Kernsystems auf das Ergebnis der Operation.

In X.400-Terminologie bilden *MUCO*, das Kernsystem und ein Darstellungssystem einen *user agent* (UA).

10.3 Das Darstellungssystem

Das Darstellungssystem bildet die Schnittstelle zwischen einem Benutzer und dem System EIGER. Der Benutzer sieht, nachdem er sich dem System erfolgreich bekannt gemacht hat, eine Reihe von Objekten, die er objektspezifisch bearbeiten kann. Das zentrale Objekt ist die ***Sitzung***, die dem Benutzer seine Arbeitsumgebung zur Verfügung stellt. Hier steht ihm ein ***Notizkorb***, ein ***Postkorb***, mindestens ein

Eingangskorb und mindestens ein ***Arbeitskorb*** zur Verfügung. Im Notizkorb empfängt der Benutzer Mitteilungen des Systems. Im Postkorb stehen ihm die Grundfunktionen eines Mailsystems zur Verfügung. In den Eingangskörben stellt das System dem Benutzer Vorgänge zu, die von ihm zu bearbeiten sind. Die wichtigsten dieser Objekte sind die Aufträge. Mit ihnen hat der Benutzer die Möglichkeit, Verarbeitungen im System auszulösen. Diese Aufträge können mehrere Dialogphasen haben, während denen ein Benutzer mit dieser Verarbeitung kommunizieren kann. Dies geschieht über das nur in diesen Phasen existierende Dialogdokument. Hier werden dem Benutzer z.B. Formulare angeboten, die er ausfüllen muß oder er erhält Ergebnisse in Form von Listen angezeigt.

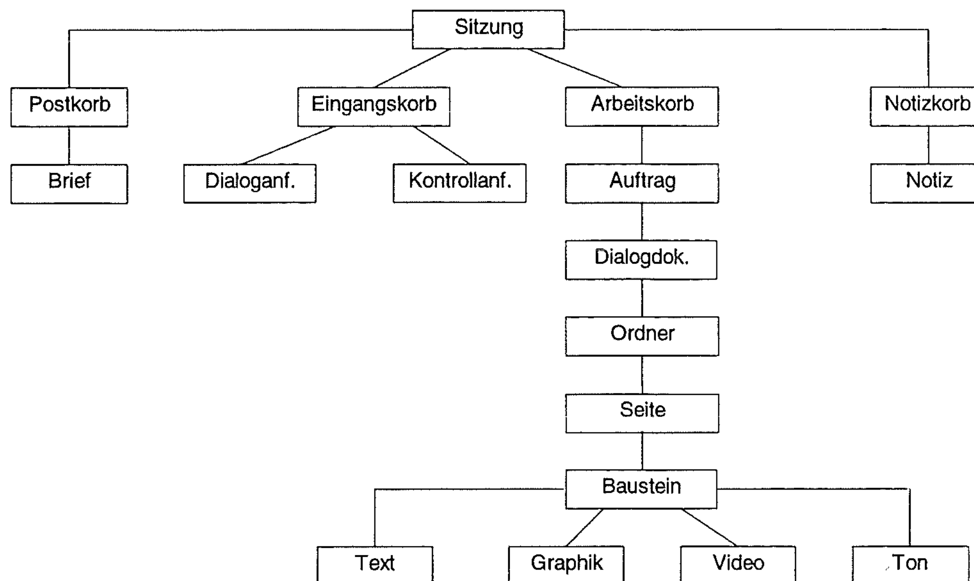


Abbildung 10-1:Die Objekttypen des Darstellungssystems

10.4 Der Aufbau des Kernsystems

Im Zentrum eines Teilsystems steht das Kernsystem, das die eigentliche Verarbeitungskapazität des Systems bereitstellt. Die zentralen Komponenten sind hier die ***Auftragsverarbeitungen***, die aufgrund der Aufträge eines Benutzers über das Darstellungssystem gestartet werden können. Für die Bearbeitung solcher Aufträge werden Dienste benötigt, wie etwa Zugriffe zur Datenbank oder Zugriff auf die Felder eines Formulars im Dialogdokument. Alle diese Dienste werden von der Komponente ***Auftragsteuerung*** erbracht. Daneben hat die Auftragsteuerung auch die Aufgabe, die Zulässigkeit der Dienstanforderung sowie die korrekte Gruppierung von Diensten zu überwachen. Die Auftragsteuerung erbringt allerdings die wenigsten dieser Dienste selbst, sondern nutzt dabei die Dienstleistungsangebote weiterer Komponenten eines Kernsystems. Die Komponente ***Datenbank*** wird benötigt, wenn eine Auftragsverarbeitung Zugriffe zur Datenbank durchführen will, die Komponente ***Briefkasten*** liefert die Zugangsmöglichkeiten zu einem Message Handling System und über die Komponente ***Sitzungssteuerung*** wird die Manipulation des Dialogdokumentes

am Sichtgerät des Benutzers möglich. Auch diese drei Komponenten erbringen nur einen Teil der geforderten Dienste, sondern stützen sich bei der Dienstbearbeitung auf die oben beschriebenen Prozeßsysteme: Die Komponente ***Sitzungssteuerung*** arbeitet mit Darstellungssystem, die Komponente ***Datenbank*** mit dem Datenbanksystem und die Komponente ***Briefkasten*** mit dem MHS zusammen.

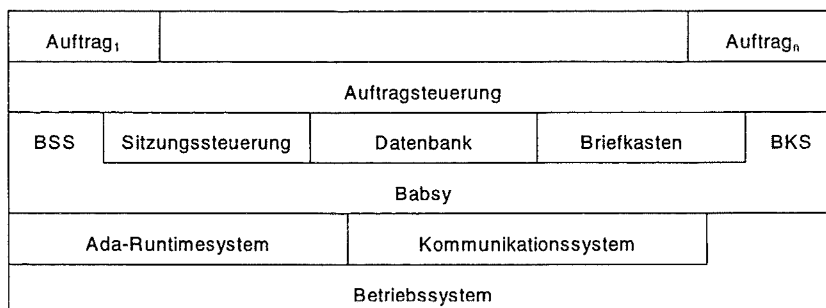


Abbildung 10-1: Struktur eines Kernsystems

Die drei Komponenten nutzen aber auch gegenseitig ihr Dienstangebot. So nutzt z.B. die Komponente ***Datenbank*** die Dienste der Komponente ***Briefkasten***, um den Replikationsmechanismus zu implementieren (siehe Abschnitt 10.8), die Komponente ***Briefkasten*** ihrerseits nutzt die Dienste der Komponente ***Datenbank***, um z.B. empfangene Mail dauerhaft zu speichern.

Die unterste Schicht des Kernsystems, ***Babsy***, stellt den übrigen Komponenten allgemeine Dienste zur Verfügung. Der zentrale Teil dieser Komponente ist das ***Basiskommunikationssystem*** (BKS), das den übrigen Komponenten des Kernsystems, die Dienste zum gegenseitigen Datenaustausch zur Verfügung stellt. Daneben existiert noch das - nicht implementierte - ***Basisschutzsystem*** (BSS), das den Datenfluß zwischen den einzelnen Komponenten überwachen und den Systemprozessen die Basismechanismen für den Aufbau eines Schutzsystems liefern soll.

10.5 Der Aufbau der Software des Kernsystems

Das Kernsystem von EIGER besteht aus einer Reihe von unabhängigen Paketen, die auch von anderen Prozeßsystemen von EIGER benötigt werden, und einer Prozedur als Hauptprogramm.

Zunächst galt es, die in Abbildung 10-1 dargestellte Struktur des Kernsystems von EIGER mit Hilfe der Möglichkeiten von Ada 83 auszudrücken. Hier bieten sich neben Funktionen und Prozeduren vor allem die Pakete und Tasks als Werkzeuge zur Strukturierung an. Dabei werden Tasks immer dann eingesetzt, wenn es Aktivitäten innerhalb eines Systems auszudrücken gilt, die wegen ihrer Unabhängigkeit parallel durchgeführt werden können. Aus diesem Grund sind im Kernsystem die Komponenten ***Auftragbearbeitung***, ***Auftragsteuerung***, ***Sitzungsteuerung***, ***Datenbank*** und ***Briefkasten*** als Tasks ausgebildet. Auch die Komponente ***Babsy*** ist als Task implementiert. Der Grund liegt hier aber nicht in der gewünschten Parallelität zu den übrigen Komponenten, sondern darin, daß in *Babsy* Datenstrukturen verwaltet werden

müssen, auf die alle übrigen Prozesse zugreifen. Diese Zugriffe müssen synchronisiert werden. In Ada 83 ist dies nur in der Form der aktiven Synchronisation durch die Verwendung einer Task möglich. Ada gewährleistet einerseits, daß zwei Tasks innerhalb des Rendezvous synchron sind und sicher Information austauschen können. Andererseits kann zwar eine andere Task diesen Eingang aufrufen, aber es ist sichergestellt, daß es nicht zum Rendezvous kommt, solange die gerufene Task sich noch im Rendezvous befindet. Da auch die anderen Eingänge der gerufenen Task während des Rendezvous nicht aktiviert werden können, kann während des Rendezvous die sichere Manipulation eines Objektes gewährleistet werden.

Für EIGER wurde die grundsätzliche Entscheidung getroffen, die Ada-eigene Rendezvous-Technik nicht zur Kommunikation zwischen den Prozessen zu verwenden, sondern die Kommunikation über einen Botschaftenaustausch zu realisieren. Aus diesem Grunde haben alle oben genannten Tasks außer *Babsy* keine Eingänge. Der Grund für diese Entscheidung lag vor allem in der Tatsache, daß das Ada-Rendezvous unsymmetrisch ist und die gerufene Task nicht sicher erkennen kann, von wem sie gerufen wird. So wird es unmöglich, zu überprüfen, ob die rufende Task überhaupt die über das Rendezvous erreichbare Dienstleistung anfordern darf. Da für das Schutzsystem auch der Datenfluß zwischen den Tasks überwacht werden sollte, muß die Kommunikation über einen zentralen Mechanismus abgewickelt werde. Dieser zentrale Mechanismus wird von *Babsy* mit dem Basiskommunikationssystem und dem Basisschutzsystem angeboten.

Pakete wurden dazu verwendet, um die geschichtete Struktur des Systems auszudrücken. Im Prinzip bietet dabei eine Schicht der oberen Schicht eine Schnittstelle, also die angebotenen Dienste mit den notwendigen Datenstrukturen, in der Paketspezifikation an. Im Paketrumpf erfolgt die Implementierung der Schicht. Benutzt die Schicht eine andere Schicht, liegt deren Paketspezifikation im Rumpf des Paketes der aktuellen Schicht.

Pakete sind außerdem das Sprachmittel in Ada 83, um abstrakte Datentypen zu realisieren. Pakete sind ein wichtiges Hilfsmittel für den objektorientierten Entwurf. Zentrale Objekte in EIGER, die als abstrakte Datentypen ausgebildet sind, sind u.a. das Dialogdokument, die Formulare und die Ausgangspostobjekte.

Aus der oben erläuterten Schichtenstruktur ergibt sich dann das untenstehende Programmgerüst. Dabei wurde teilweise auf die oben erwähnte Trennung von Spezifikationen und Implementierungen sowie große Teile der Spezifikationen verzichtet. Die Originalsoftware besteht ansonsten aus ca. 700 Übersetzungseinheiten.

Bedingt dadurch, daß einige Schnittstellendefinitionen wieder aus einer Menge von geschachtelten Paketen bestehen, oft mit einem privaten Teil, der dann selbst wieder Pakete enthält, ist diese Struktur nur noch schwer in den Quelltexten nachzuvollziehen.

```
with Auftraege_Umgebung;
with Listenverwaltung;
...
procedure Eiger is
  package Anwendungen is
  end Anwendungen;
  package body Anwendungen is
    task type Auftraege;
    package Betriebssystem is
      -- Definition der Schnittstelle zwischen den Tasks vom Typ
      -- Auftraege und dem Betriebssystem durch Datentypen,
      -- Prozeduren und Funktionen
    private
      -- private Typen des Betriebssystems
    end Betriebssystem;
    package body Betriebssystem is
      task type Auftragsteuerungen;
      task type Datenbanken;
      task type Sitzungssteuerungen;
      task type Briefkaesten;
      package Betriebssystemkern is
        -- Definition der Schnittstelle zwischen den
        -- Betriebssystemprozessen der Basis des Betriebssystems
        -- durch Datentypen, Prozeduren und Funktionen
      end Betriebssystemkern;

      package body Betriebssystemkern is
        task type Babsy_Prozesse is
          -- Eingaenge fuer die Dienste
        end Babsy_Prozesse;
        Babsy : Babsy_Prozesse;
        task body Babsy_Prozesse is separate;
      begin
        -- Initialisierung des Teilsystems
        ...
      end Betriebssystemkern;
      task body Auftragsteuerungen is separate;
      task body Datenbanken is separate;
      task body Sitzungssteuerungen is separate;
      task body Briefkaesten is separate;
    end Betriebssystem;
    task body Auftraege is separate;
  end Anwendungen;
begin
  null;
end Eiger;
```

Die vier Systemprozesse *Auftragsteuerung*, *Sitzungssteuerung*, *Datenbank* und *Briefkasten* haben eine einheitliche Grundstruktur. In einer „Endlosschleife“ werden die Botschaften anderer Prozesse des Systems angenommen und zu Dienstkontrollblöcken umgeformt. Die entstandenen Dienstkontrollblöcke werden sofort bearbeitet. Wenn dabei neue Dienstkontrollblöcke entstanden sind oder bereits vorhandene verarbeitungsbereit geworden sind, werden auch diese bearbeitet. Ein Dienstkontrollblock ist dabei als Verbund mit vielen Varianten implementiert. Jedem Dienst entspricht eine Variante.

```
while not Nimmerleinstag loop
  Warte_auf_Eintreffen_einer_Botschaft.
  if Existierender_Dienstkontrollblock_Betroffen then
    Übernehme_Daten_der_Botschaft_in_den_Dienstkontrollblock
  else
    Bilde_aus_Botschaft_einen_Dienstkontrollblock
  end if;
  Vernichte_Botschaft.
  Bearbeite_Dienstkontrollblock.
  while Verarbeitungsbereiter_Dienstkontrollblock_Vorhanden loop
    Bearbeite_Dienstkontrollblock.
  end loop;
end loop;
```

Der Pseudocode *Übernehme_Daten_der_Botschaft_in_den_Dienstkontrollblock* bzw. *Bilde_aus_Botschaft_einen_Dienstkontrollblock* umschreibt den Empfang und die Analyse der Botschaft sowie den Aufbau eines Dienstkontrollblocks. Der Pseudocode *Bearbeite_Dienstkontrollblock* ist im wesentlichen eine umfangreiche case-Anweisung, in der jeder when-Zweig lediglich aus dem Aufruf einer Prozedur besteht, in der der Dienst erbracht wird..

Der objektorientierte Entwurf wird in Ada 83 hinsichtlich der Wiederverwendbarkeit vor allem durch die generischen Einheiten, insbesondere die generischen Pakete, unterstützt. In solchen Einheiten werden Verfahrensmuster definiert, die auf unterschiedliche Datentypen angewendet werden können. Für die Anwendung auf einen bestimmten Datentyp müssen solche generischen Einheiten dann „ausgeprägt“ werden. Eine zentrale Rolle im Kernsystem spielt das generische Paket *Listenverwaltung*, das ca. 200 mal für unterschiedliche Typen ausgeprägt wurde. Dieses generische Paket stellt alle Funktionen und Prozeduren zur Verfügung, die benötigt werden, um doppelt verzeigerte Listen verwalten zu können. Als Parameter läßt dieses Paket jeden beliebigen einfachen oder zusammengesetzten Typ als Listenelement zu.

Mit dem Ziel einer guten Übersichtlichkeit und leichten Wartbarkeit der Programme wurde versucht, die Übersetzungseinheiten möglichst klein zu halten (ca. zwei bis drei DIN A4-Seiten). Dabei wurde von der Möglichkeit Gebrauch gemacht, einerseits Spezifikation und Implementierung einer selbständigen Einheit und andererseits die Implementierung einer in der Implementierung einer anderen Einheit stehenden Spezifikation jeweils in einer Datei unterzubringen. Diese Quelldateien können dann separat übersetzt werden. Diese strikte Trennung von Spezifikation und Implementierung, hat als Konsequenz, daß bei Korrekturen und Änderungen in einer Implementierung nur diese übersetzt werden muß, da eine andere Einheit nie von dieser Implementierung abhängen kann. Wenn allerdings in der Implementierung weitere Pakete verwendet werden, dann müssen natürlich auch deren Implementierung übersetzt werden. Bei der oben beschriebenen Implementierung der Schichtenstruktur des Kernsystems von EIGER kann es so auch bei Änderungen in der Implementierung eines Paketes zu zeitaufwendigen Nachübersetzungen kommen.

10.6 Bewertung der Implementierung

Die Bewertung der Implementierung soll hier ausschließlich unter dem Aspekt der Verwendung von Ada 83 erfolgen.

Zunächst soll bei aller Kritik betont werden, daß sich die Verwendung von Ada in diesem Projekt sehr positiv ausgewirkt hat. Die Gründe dafür sind:

- Die strenge Typbindung. Sie macht es möglich, daß sehr viele Fehler bereits zur Übersetzungszeit erkannt werden. Viele der restlichen Fehler werden dann zur Laufzeit vom Laufzeitsystem entdeckt, wenn die Einschränkungen für die Variablen überprüft werden (Die Ausnahme Constraint_Error ist nicht umsonst der häufigste Abbruchgrund beim Testen!).
- Unterstützung von Geheimnisprinzip und Kapselung, die dabei helfen, die Abhängigkeiten zwischen getrennt entwickelten Modulen zu minimieren.
- Parallelisierbarkeit von Verarbeitungen mittels Tasking.

Anfänglich existierten zwar nur sehr instabile Compiler, die doch wohl eher dazu gebaut waren, die Tests der Validierungssuite zu überstehen, als den Anforderungen von echten Benutzerprogrammen standzuhalten. Glücklicherweise hat sich dies sehr schnell geändert. Der derzeit verwendete Compiler hat sich als sehr stabil erwiesen. Zudem ist der Compiler in eine Entwicklungsumgebung integriert, die die Projektarbeit sehr erleichtert.

In Ada 83 waren die Möglichkeiten, Schnittstellen zu anderen Systemen und Programmen in anderen Programmiersprachen zu bedienen, nicht sehr leistungsfähig. Dafür waren lediglich die Darstellungsklauseln und verschiedene, teilweise sogar herstellerspezifische Pragmas vorgesehen. Die Schnittstellen von EIGER zu anderen Systemen und zu in anderen Sprachen vorliegenden Komponenten waren deshalb immer besonders trickreich konstruiert und sehr fehleranfällig. Im Falle des Darstellungssystems EMMI, das dem Benutzer eine grafische Oberfläche mit Hilfe von MOTIF bietet, führten diese Schwierigkeiten dazu, diese Komponente fast völlig in der Programmiersprache C zu implementieren. Der Hauptgrund für die Schwierigkeiten liegt in diesem Falle darin, daß Ada 83 keine Möglichkeit vorsieht, Prozeduren als Parameter in einem Prozeduraufruf zu übergeben. In vielen Prozeduren, die von MOTIF als Schnittstelle angeboten werden, wird aber gerade von dieser Möglichkeit Gebrauch gemacht, um die Adressen der sog. „call back“-Routinen zu übergeben.

Eine zweite Schwäche von Ada 83 war die schon erwähnte flache Struktur des Bibliothekskonzeptes: Alle Bibliothekseinheiten haben einen gemeinsamen Vater, das Paket SYSTEM, können aber selbst keine Kinder haben. Aus diesem Grund war es z.B. nicht möglich, zwei Bibliothekspakete zu konstruieren, die beide die volle Sicht eines gemeinsamen privaten Typs haben. Eine Konsequenz dieser flachen Struktur ist es, daß z.B. das Paket ***Auftraege*** sehr groß und monolithisch geworden ist (ca. 35 DIN A4-Seiten). Dies ist nur noch schwer zu überblicken und Änderungen darin führen, da es ein zentrales Paket ist, zu sehr vielen und vor allem zeitaufwendigen Übersetzungen.

Wie bereits mehrfach erwähnt, bietet bereits Ada 83 einige Mittel, die für das objektorientierte Programmieren benötigt werden. Allerdings ist die Möglichkeit, das sog. Programmieren durch Erweiterung zu realisieren, in Ada 83 mit den Paketen und den Möglichkeiten der generischen Einheiten nur sehr schwach ausgebildet. Zum einen

zeigte sich dieses Manko darin, daß eine Erweiterung des Systems immer von der Modifikation bestehender ausgetesteter Komponenten begleitet war, wobei sich sehr oft in diesen Komponenten wieder Fehler einschlichen. Viel schwerwiegender aber war die Tatsache, daß keine heterogenen Listen verwaltet werden können und daß es bei Verwendung des generischen Paketes Listenverwaltung nicht möglich ist, indirekt rekursive Verzeigerungen herzustellen. Gerade im letzten Fall blieb nur das fehleranfällige „ungeprüfte Programmieren“.

Eine weitere Schwäche offenbarte Ada 83, wenn es galt, den Zugriff auf Daten zu steuern, die von mehreren Tasks gemeinsam benutzt werden. Dazu bietet Ada 83 als Hilfsmittel lediglich die aktive Synchronisation des Zugriffs mittels einer Task an, es fehlt ein monitorähnliches Konzept mit passiver Synchronisation.

Da in Ada 83 die Eigenschaft des Polymorphismus nicht vorhanden war, ergaben sich bei der Implementierung des Basiskommunikationssystems Schwierigkeiten. Die Datenstrukturen, die zwischen den Komponenten des Kernsystems ausgetauscht werden, sind hinsichtlich Struktur und Umfang sehr unterschiedlich. Da aber in Ada 83 alle Teilbotschaften von genau einem Typ sein müssen, wurde ein varianter Verbundtyp definiert, der jede auszutauschende Botschaft als Variante enthielt. Die Implementierung solcher varianten Verbunde stellt dann immer den Speicherplatz der größten Variante zur Verfügung. Das ist auch sinnvoll, da jedem Objekt von diesem Typ ja auch ein Wert vom Typ der größten Variante zugewiesen werden kann. Da der Platzbedarf der größten Variante bei einem Megabyte lag, zeichnete sich schnell ein Speicherplatzproblem ab, da zu einem Zeitpunkt sehr viele Botschaften existieren konnten. Deshalb wurde die Entscheidung getroffen, die Daten in einer „Name-Typ-Wert“-Struktur zu übertragen. Eine Teilbotschaft enthält dann immer ein einziges Skalar, ein „kleines“ Feld von solchen Skalaren oder auch nur Strukturinformation wie etwa „Beginn eines Feldes mit 123 Elementen“. Deshalb besteht eine Botschaft oft aus sehr vielen Teilbotschaften. Dazu ist sehr viel Codierungs- und Decodierungsarbeit erforderlich, die bei sehr großen Datenmengen pro Botschaft sehr zeitintensiv ist. Deshalb mangelt es der Ada83-Implementierung des Kernsystems gelegentlich auch an Performanz.

10.7 Ansatz für die Implementierung von EIGER mit Ada 95

Ein möglicher Ansatz bei der Weiterentwicklung von EIGER wäre, ab sofort die neuen Sprachmittel von Ada 95 einzusetzen, die Struktur des Gesamtsystems aber unangetastet zu lassen. In diesem Fall ist der Einsatz von Ada 95 allerdings von wenig Nutzen, da dann die neuen Sprachmittel zur Strukturierung und Erweiterung von großen Systemen kaum mehr genutzt werden können und die erkannten Schwachstellen der Implementierung nicht beseitigt werden könnten. Deshalb wurden Überlegungen in die Richtung angestellt, das Kernsystem von EIGER unter Verwendung der neuen Sprachmittel völlig neu zu strukturieren. Hierbei wurden dann das neue Bibliothekskonzept, die Vererbung, Polymorphismus, dynamisches Binden, die geschützten Typen bzw. Objekte („protected types“) und die verbesserten Möglichkeiten der Schnittstellenbildung zu Programmen in anderen Programmiersprachen genutzt. Ein schwerwiegender Grund für eine neue Struktur ergibt sich bei der Verwendung von Typerweiterungen aus der Tatsache, daß bestimmte Erreichbarkeitsregeln („accessibility rules“) für Typerweiterungen eingehalten werden müssen. Die wichtigste Regel besagt, daß eine Typerweiterung nicht an einer Stelle durchgeführt werden darf,

die vom Vatertyp nicht erreichbar ist, also etwa innerhalb eines inneren Blocks. Solche inneren Blöcke bilden z.B. die Rümpfe von Funktionen, Prozeduren und Tasks. Diese Regel stellt sicher, daß einem klassenweiten Objekt kein Wert zugewiesen werden kann, der dann seinen speziellen Typ „überlebt“. Aus diesem Grund wurden bei der Neustrukturierung des Systems alle Typerweiterungen auf die selbe Erreichbarkeitstufe („accessibility level“) gelegt und ausschließlich in Spezifikationen von Bibliothekspaketen gemacht, da solche Einheiten auf einer Erreichbarkeitstufe, dem sog. „library-level“, liegen.

Ausgangspunkt für die Implementierung ist die in Abbildung 10-1 dargestellte Schichtenstruktur des Kernsystems. Die Schichtung und die damit verbundenen Sichtbarkeitsverhältnisse wurden durch eine Hierarchie von öffentlichen und privaten Bibliothekseinheiten nachgebildet. Das Paket *Eiger* bildet dabei die Wurzel des Baumes aller Bibliothekseinheiten des Kernsystems. Die Spezifikation des Paketes enthält lediglich Deklarationen für einige für EIGER spezifische Datentypen, die in allen Komponenten des Kernsystems benötigt werden. Das Paket *Eiger* hat zwei wichtige öffentliche Kinder: das öffentliche Paket *Betsy* und das öffentliche Paket *Anwendungen*. Das Paket *Betsy* mit seinen Kindern realisiert dabei das Betriebssystem, während das Paket *Anwendungen* mit seinen Kindern das Anwendungssystem realisiert. Dabei stellt das Paket *Betsy* lediglich einige Datentypen zur Schnittstellenbildung zwischen den beiden von den beiden Paketen *Betsy* und *Anwendungen* gebildeten Unterbäumen zur Verfügung. Die eigentliche Schnittstelle wird durch das öffentliche Kindpaket *Bearbeitung_As* des Paketes *Betsy* und seiner Kinder zur Verfügung gestellt.

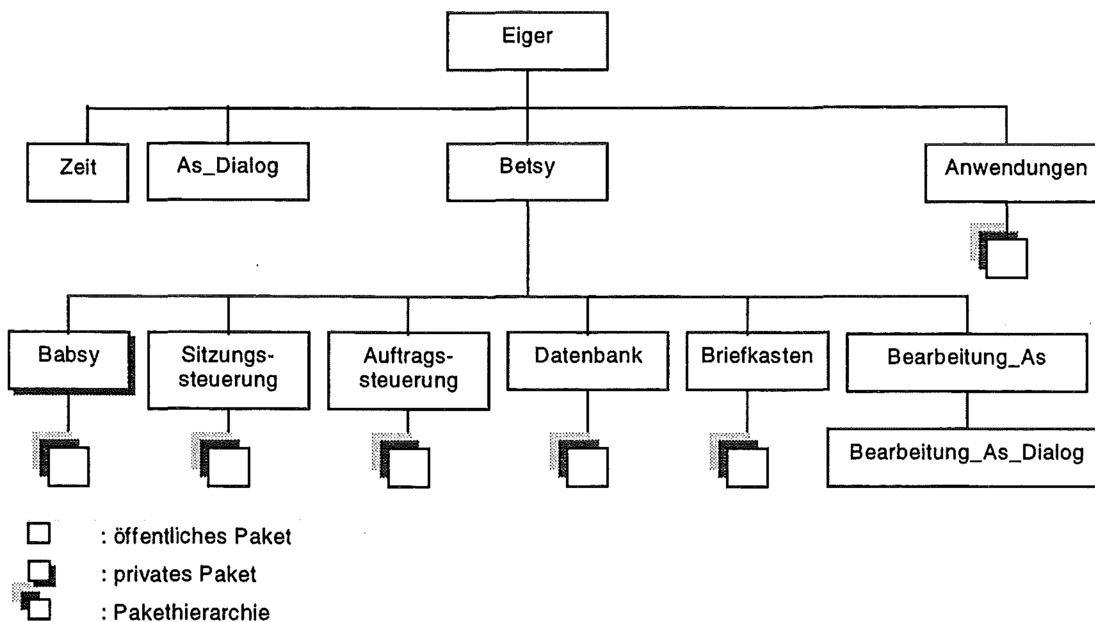


Abbildung 10-1: Hierarchie der grundlegenden Bibliothekseinheiten

Die Systemprozesse *Auftragssteuerung, Sitzungssteuerung, Datenbank* und *Briefkasten* werden in je einem Unterbaum realisiert, dessen Wurzel ein öffentliches Kindpaket des Paketes *Betsy* ist. Bei der Implementierung der entsprechenden Tasks können die Vorteile der Möglichkeit der Typerweiterung genutzt werden. Alle

Botschaften an einen solchen Systemprozeß werden von einem Wurzeltyp abgeleitet. Dasselbe gilt für Dienstkontrollblöcke. Damit reduziert sich die Codierung der beiden Pseudocodezeilen *Übernehme_Daten_der_Botschaft_in_den_Dienstkontrollblock* bzw. *Bilde_aus_Botschaft_einen_Dienstkontrollblock* des Programmgerüsts in Abschnitt 10.5 auf jeweils einen Prozeduraufruf, der entsprechend der Botschaft bzw. des Dienstkontrollblocks dispacht. Dies erleichtert die Erweiterung der Systemprozesse um neue Dienste erheblich.

Die Funktionen des Betriebssystemkerns werden den Systemprozessen durch einen Unterbaum am Paket *Betsy* bereitgestellt. Die Wurzel dieses Unterbaumes bildet das Paket *Babsy*. Im Gegensatz zu der Ada 83-Implementierung des Systems wurden die Funktionen des Betriebssystemkerns nicht mehr auf der Basis einer Task sondern mit Hilfe eines „protected object“ realisiert.

Eine Implementierung auf der Basis dieser Strukturen wurde nur insoweit vorgenommen, daß die Kommunikationsbeziehungen zwischen den Komponenten des Systems sowie die grundlegende Funktionsweise der einzelnen Komponenten überprüft werden konnten. Die dabei gemachten Erfahrungen bestätigen die Aussagen, daß das Konzept der hierarchischen Bibliotheken in Verbindung mit der Möglichkeit der Typerweiterungen die Entwicklung großer Softwaresysteme sehr erleichtert.

10.8 Prinzipien des ATCCIS-Replikationsmechanismus

Im Rahmen der ATCCIS-Studie wird ein Systemkonzept für die zukünftigen taktischen Heeresinformationssysteme im NATO-Bereich Europa Mitte (Central Region) entwickelt. Das Ziel des Konzeptes ist es, sicherzustellen, daß die Interoperabilität zwischen solchen Systemen auf der Basis des NATO System Interconnection Level 5 des NATO Interoperability Planning Document (NIPD) unterstützt wird [1]. Diese Stufe der Interoperabilität erfordert eine Verbindung zwischen zwei Systemen mit einem einheitlichen Kommunikationsverfahren, das bei einem Datenaustausch keine Datenumformungen oder manuellen Eingriffe erforderlich macht. Hierbei kann der lesende und schreibende Datenzugriff jedoch vom Nutzer gesteuert werden, wodurch die Kontrolle über die eigenen Daten jederzeit sichergestellt ist. Zu diesem Zweck wurde in der ATCCIS-Studie ein Replikationsmechanismus spezifiziert, der diesen kontrollierten Datenaustausch in heterogenen Systemen ermöglicht.

Die Basis für diesen Replikationsmechanismus ist, daß für alle ATCCIS-konformen Systeme ein einheitliches Datenmodell [2] festgelegt ist. Dieses Datenmodell basiert auf dem relationalen Datenmodell und beschreibt die Datenbasis eines ATCCIS-konformen Systems in Form von Tabellen. Die Beschreibung selbst ist wiederum in Tabellen enthalten, die das Metadatenmodell bilden. Hier wird u.a. festgehalten, welche Namen die Tabellen des Datenmodells haben, welche Attribute eine Tabelle besitzt und welches die Wertebereiche dieserAttribute sind.

Der Umfang der zwischen zwei Partnersystemen zu replizierenden Daten wird in bilateralen *Verträgen* festgelegt. Sie können weiter ergänzt werden durch die sog. *Filter,* die die Replikation von Zeilen einer Tabelle vom Wertebereich einzelner Attribute abhängig machen.

10.8.1 Das ATCCIS Replication Mechanism (ARM) Reference Model

Das ARM Reference Model [3] sieht fünf Schichten vor, in denen der Replikationsmechanismus abgewickelt wird.

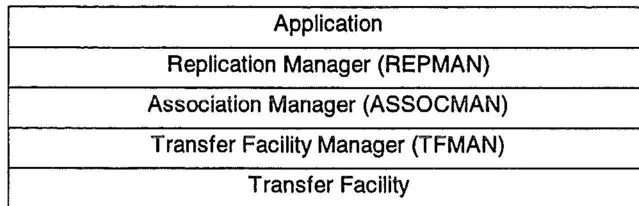


Abbildung 10-1: Das Referenzmodell des ATCCIS-Replikationsmechanismus

10.8.1.1 Die Transfer Facility

Die unterste Schicht wird von der Transfer Facility gebildet und stellt eine Datenübertragungsfunktion zwischen zwei am Replikationsmechanismus beteiligten Knoten zur Verfügung. Dabei können sowohl verbindungslose wie auch verbindungsorientierte Dienste bereitgestellt werden. Die Vereinbarungen der ATCCIS-Arbeitsgruppen sehen hier einen Datenaustausch über ein Message Handling System (MHS) nach X.400, über TCP/IP- und X.25-Verbindungen vor. Derzeit werden in EIGER als Transfer Facilities nur MHS nach X.400 und TCP/IP unterstützt. Geplant ist unsererseits die Implementierung einer CORBA-basierten Transfer Facility.

10.8.1.2 Der Transfer Facility Manager

Diese Schicht ist logisch eigentlich eine Unterschicht der Schicht Association Manager. Eine Instanz dieser Schicht hat die Aufgabe, die von einer ASSOCMAN-Instanz angelieferten Nachrichten in der Form, wie sie von der Transfer Facility erwartet wird, an diese abzuliefern, und umgekehrt, die von der Transfer Facility bereitgestellten Daten in der Form, wie sie von der ASSOCMAN-Instanz erwartet wird, dieser bereitzustellen. Außerdem realisiert diese Schicht das (im ARM Reference Model nicht erwähnte, aber notwendige) Multiplexen bzw. Demultiplexen von zwei ASSOCMAN-associations über eine Transfer-Facility-association. Diese Schicht bietet zwei Dienste an:

- Senden von Daten
- Empfangen von Daten

10.8.1.3 Der Associacion Manager

Die Aufgaben einer Instanz der ASSOCMAN-Schicht entsprechen im wesentlichen denen der Instanzen der Transportschicht des ISO-OSI-Referenzmodells: Sie bringt eingegangene Nachrichten in die richtige Reihenfolge und überwacht, daß keine Nachrichten verloren gehen. Das Protokoll einer Instanz dieser Schicht ist verbindungsorientiert. Die Schicht bietet die folgenden Dienste:

- Eröffnen einer association
- Schließen einer association
- Abbrechen einer association

- Senden von Daten
- Empfangen von Daten
- Austausch von Empfangsbestätigungen

Optional kann auch der Dienst „Anfordern von Übertragungswiederholungen“ bereitgestellt werden.

10.8.1.4 Der Replication Manager

Diese Schicht besteht aus den zwei Unterschichten

- replication protocol manager
- replication data manager

Die „replication protocol manager“-Teilschicht ist für das Senden und Empfangen von Replikationsdaten zuständig und nutzt dabei die Dienste der ASSOCMAN-Schicht. Er bietet der „replication data manager“-Teilschicht die folgenden Dienste:

- Eröffnen einer „session“
- Senden von Daten
- Empfangen von Daten
- Austausch von Verarbeitungsbestätigungen
- Abbrechen einer „session“
- Schließen einer „session“

Der „replication data manager“-Sublayer sammelt die zu replizierenden Daten und sendet sie unter Verwendung der Dienste des „replication protocol manager“ an die Replikationspartner. Umgekehrt hat diese Unterschicht auch die Aufgabe, von Partner empfangene Daten in die eigenen Datenbasis einzuarbeiten. Weiterhin überprüft diese Unterschicht beim Aufbau einer „session“, ob die beiden Datenbanken synchron sind oder dieser Zustand zunächst durch einen „bulk update“ hergestellt werden muß.

10.8.1.5 Die Anwendungen

Der Application Layer enthält die Anwendungsprogramme, die auf der Datenbasis des Systems arbeiten und u.a. dort gespeicherte Daten ändern.

10.8.2 Die Implementierung des ARM in EIGER

Das in EIGER verwendete Datenmodell ist identisch mit dem ATCCIS-Datenmodell, so daß keine Abbildung der beiden Modelle erforderlich ist. Allerdings sind alle Tabellen noch um einige Spalten vergrößert worden, die Verwaltungsdaten für die Implementierung des Replikationsmechanismus enthalten. Für alle Tabellen sind Trigger definiert, die bei jeder Änderung von Daten in der Tabelle durch die Anwendungsprogramme feuern und eine PL/SQL-Prozedur anstoßen, die unter Berücksichtigung der abgeschlossenen Verträge die zu replizierenden Daten („incremental updates“) samt deren Empfänger in einer speziellen Tabelle sammeln. Zusammen mit zwei weiteren PL/SQL-Prozeduren zum Erzeugen eines vollständigen Auszugs aus der Datenbank auf Grund der mit einem Partner bestehenden Verträge („bulk update“) und zum Einarbeiten empfangener Replikationsdaten bildet sie die Implementierung des „replication data manager“.

Die Unterschicht „replication protocol manager“, sowie die Schichten „association manger“ und „transfer facility manger“ wurden vollständig in der Task ***Datenbank*** des Kernsystems von EIGER implementiert. Die Funktionen der „transfer facility“ waren in dem bestehenden System bereits vorhanden und konnten mit geringfügigen Erweiterungen weiter verwendet werden: Die TCP/IP-Funktionen waren u.a. für die Kommunikation von Kernsystemen untereinander vorhanden und die X.400-Dienste wurden von der Task ***Briefkasten*** für die Bearbeitung der Postkörbe und deren Objekte angeboten. Die einzelnen Dienstprimitive der Protokollschichten sind als Dienste in der Task Datenbank implementiert. Diese Dienste werden aber nicht durch externe Ereignisse, also Eintreffen einer Botschaft, ausgelöst, sondern werden aufgrund von taskinternen Ereignissen angestoßen, etwa wenn eine Protokollschicht einen Dienst der darunter liegenden Protokollschicht benötigt. Auf diese Weise fügte sich die Erweiterung der Task durch die Implementierung der Protokolle des ATCCIS-Replikationsmechanismus in die bestehende Software ein

Unsere Implementierung des ATCCIS-Replikationsmechanismus wurde zusammen mit den Implementierungen von Entwicklungsteams aus fünf weiterer NATO-Staaten ausgiebigen Tests unterzogen. Auch in 48-Stundentests hat sich das System als sehr stabil erwiesen, wozu die Verwendung von Ada als Implementierungssprache sicher einen guten Teil beigetragen hat.

10.9 Literaturhinweise

[1] *NATO Interoperability Planning Document (NIPD).* Volume II, Allied Data Systems Interoperability Agency (ADSIA), März 1992.

[2] *ATCCIS Battlefield Generic Hub 3 Data Model Specification.* ATCCIS Working Paper 5-5, Draft 1.0, ATCCIS Permanent Working Group, SHAPE, Belgien, November 1996.

[3] *ATCCIS Replication Mechanism: Concepts and Models.* ATCCIS Working Paper 14-2, Edition 1.0, ATCCIS Permanent Working Group, SHAPE, Belgien, März 1996.

Ada in den Streitkräften - Bilanz und Tendenz-

11 Ada in den Streitkräften - Bilanz und Tendenz-

Ingo Siebert

Oberstleutnant a.D.

Vorwort

Die Informationstechnologie bestimmt zunehmend unser aller Leben. Sie ist an die Stelle gewachsener, traditioneller Techniken getreten, die über Zeitalter hinweg durch das Wissen, die Erfahrung und auch die Kunst der Ingenieure geprägt worden sind.

Die Informationstechnologie unserer Tage ist noch jung, sie kann dies alles noch nicht im gleichen Maße vorweisen. So erleben wir auf dem Gebiet der Hardwareentwicklung, wo klassisches Ingenieurswissen und -können immer noch die entscheidende Rolle spielen, Leistungsexplosionen in immer kürzeren Zyklen. Der Wissensaufbau auf den Gebieten des Software- und Systems- Engineerings und der Sprachtechnologie, den Schlüsseln zur Leistungsentfaltung und -optimierung von Informationsverarbeitungssystemen aller Art, hält nicht mit. Der Schritt vom Denken in analogen Bildern zum Denken in digitalen Netzwerken ist nur schwer zu vollziehen, mentale Hemmnisse und das Generationsproblem ergänzen die schwierige Situation an der Schwelle zu dem neuen Technologiezeitalter. Das Umsetzen der Grundstrukturen und Tugenden bewährter Ingenieursvorgehensweisen erfolgt nur allzu langsam.

Das Verteidigungsministerium hat in diesen Bereichen in Deutschland maßgebliche Beiträge geleistet.

Auf dem Gebiet des Software- und System- Engineerings hat es, ab 1986, in enger Zusammenarbeit mit der deutschen Industrie, bis 1992, den *„Softwareentwicklungsstandard der Bundeswehr“* ,kurz das *„V-Modell“* entwickelt. Dieser Standard hat, im Rahmen seiner Weiterentwicklung zu dem *„Entwicklungsstandard für IT-Systeme des Bundes“, 1997* einen vorläufigen Abschluß gefunden. Wie es der abgeänderte Titel vermuten läßt, ist er für alle Bundesbehörden verbindlich, Länder und Kommunen beginnen zu folgen (über den IMKA und Koop A).

Der Standard ist „public domain“, jedermann kann ihn kostenfrei nutzen. (Anmerkung: Der Standard liegt in CD-ROM-Form, einschließlich eines Ausbildungsprogrammes, vor. Er kann kostenfrei über den Nutzerverein ANSSTAND e.V., http:// www.ansstand.de, bezogen werden.)

Auf dem Gebiet der Sprachtechnologie, insbesondere bei *Ada*, hat das Verteidigungsministerium, beginnend 1979, ebenfalls einen signifikanten Beitrag geleistet, der zwischenzeitlich 1989 zu dem Meilenstein *„Ada-Spracherlaß“* führte.

11.1 Was hat die Bundeswehr veranlaßt, was hat sie getan?

Ihre Aktivitäten reichen fast 20 Jahre zurück. Lassen Sie uns kurz Revue passieren.

Seit 1979 war Deutschland an der Ada-Entwicklung als „distinguished reviewer“ beteiligt.

Seit 1983 wurden die Forschungen und Arbeiten auf dem Gebiet der Ada- Compiler-Technologie unterstützt, Stichworte sind Universität Karlsruhe und „Sperber“.

Seit 1984 wurde der Aufbau und der Betrieb einer Validierungsstelle für Ada- Compiler bei der IABG massiv unterstützt.

Seit 1986 beteiligte man sich am CAIS- Programm.

1989 wurde der „Ada- Spracherlaß“ in Kraft gesetzt, der den Einsatz von Ada verbindlich und grundsätzlich vorgibt (in seinen Regelungen aber Ausnahmen für Fachinformations- und Prüfsysteme zuläßt). Ada wurde „mandatory“ bis auf den heutigen Tag. Mit einem Schlußsatz forderte der Erlaß nach drei Jahren die Vorlage eines Erfahrungsberichtes. Ich werde auf ihn zurückkommen.

Seit 1989 ebenfalls hat das Ministerium den Aufbau und ständigen, aktuellen Betrieb eines „Ada- know- how- Zentrums“ bei der Firma CCI veranlaßt und finanziert.(http://www.cci.de oder E-Mail: Ada-Forum @cci.de)

1993 wurde eine Ada- Vorschrift, der „Programmierleitfaden für Ada“, erarbeitet und in Kraft gesetzt. (Allgemeiner Umdruck Nr. 255). Sie beinhaltet heutzutage den Ada 95-Standard.

Seit 1994 besteht eine Weisung, die besagt, daß Commercial-off-the-shelf-Produkte einzusetzen sind, wenn die Requirements erfüllt werden, ansonsten jedoch Ada zu verwenden ist.

1995 wurde, in enger Zusammenarbeit mit der Industrie, parallel zum Erscheinen des neuen Standards „Ada 95“, eine umfangreiche Initiative zur Steigerung des Bekanntheitsgrades und der Akzeptanz von Ada eingeleitet.

Als Zielgruppen außerhalb der Bundeswehr wurden, allen voran, die Universitäten als Ausbildungsstellen und akademische Meinungsmacher, die Industrie als Hersteller von Informationssystemen aller Art und die Verwaltungen als Entscheidungsträger in Auswahlabläufen, festgelegt. Man war sich einig, daß der Erscheinungszeitpunkt des neuen Standards die letzte Möglichkeit war, Ada außerhalb und innerhalb der Bundeswehr, und hier vor allem der Bundeswehrverwaltung, insbesondere des Bundesamtes für Wehrtechnik und Beschaffung, mehr bekannt zu machen und durchzusetzen.

Zu dem Maßnahmenbündel gehörten die auflagenfreie Verteilung von „Active Ada“-Compilern - und später von „Object-Ada“- Compilern - an Programmier- und Rechen-zentren der Bundeswehr für Schnupperzwecke, die kostenfreie Verteilung von GNAT-Compilern an alle Interessenten in Deutschland, Veröffentlichungen in Fachzeitschriften und militärischen Blättern, Ada- Fachveranstaltungen in Industrie-, in BMVg-, in BWB-Bereichen und in den Programmierzentren der Teilstreitkräfte als quasi Road-shows. Es

wurde ein Internetdienst über Ada und ihre technischen Möglichkeiten eingerichtet, der übrigens weltweit, vorzugsweise durch US-Universitäten, genutzt wurde und wird.(http://www.cci.de)

1995 beauftragte das Ministerium die Entwicklung eines Informations- und Lernwerkzeuges zu Ada 95, die **Ada 95 Tour.**

Als Medium wurde CD-ROM-Technik ausgewählt; leistungsstark, billig und als Massen- oder Schüttgut bestens geeignet.

Eine US-Studie (Marketing Communications Study Precampaign Survey, Conducted for Telos Corporation, May 1994) hatte 1994 folgendes an den Tag gebracht:

Entscheidungsträger benötigten eine Sprache mit den Eigenschaften von Ada, aber Ada war ihnen nahezu unbekannt!

Unsere Konsequenz daraus war, die zeitliche Lücke zwischen dem Erscheinen des Sprachstandards Ada 95 und der Marktverfügbarkeit von Ada 95-Compilern zu nutzen und ein modernes, anspruchsvolles Ausbildungs- und Schulungsmittel für einen unverzüglichen Start der Ada- Sprachausbildung für alle interessierten Gruppierungen in der Bundeswehr, an den Hochschulen und in der Industrie zu schaffen und ohne Lizenzbeschränkung („public domain“) bereitzustellen.

Die konsequenterweise entstandene CD-ROM enthielt neben einem Schulungsprogramm und den Sprachstandards, die GNAT- Compiler und den Object Ada 7.0 Compiler der heutigen Firma AONIX. Sie erschien Anfang Mai 1996, gut ein Jahr nach der Veröffentlichung des neuen Standards und fünf Monate bevor Ada 95-Produktions-Compiler marktverfügbar waren. Dies verschaffte allen Ziel- und Interessengruppen die Zeit, sich mit der überarbeiteten Sprache vertraut zu machen und Entscheidungen vorzubereiten. Die Auflagenhöhe der CD-ROM betrug 30 000 Stück, einschließlich diverser Nachprägungen. Die Inhalte wurden ebenfalls in das Internet eingestellt.

Die Ada 95 Tour hat die ausgesuchten Zielgruppen erreicht, insbesondere die Industrie und die Hochschulen. Berliner Hochschulen haben der Firma CCI, die die CD-ROM konzipiert hat und als Ansprechpartner auftritt, bei Weiterentwicklungen der CD-ROM-Inhalte, ihre Mitarbeit angeboten.

1997 hat das Ministerium eine ergänzte Ada 95 CD-ROM entwickeln lassen. Neben neuesten Compilerversionen enthält sie alle Anhänge -“Annexes“- zu Ada 95 und einen „Ada/Java“-Compiler, der Ada-Programmen den Zutritt zum Internet ermöglicht. Tucker Taft hat diesen Compiler für Intermetrics geschrieben. Die CD-ROM ist für multinationale Projekte zurechtgeschnitten, sie ist zweisprachig, deutsch und englisch, dem internationalen Einsatz sind somit keine Grenzen mehr gesetzt.

Der englischsprachliche Feinschliff erfolgte in USA und England durch freiwillige Hilfe von AONIX- Mitarbeitern. Die CD-ROM ist ein einzigartiges Beispiel für ein organisationen-, sprachen- und länderübergreifendes „Joint venture“.

Der Titel dieser CD-ROM lautet:

Ada-Tour 2.0 i.

Die CD-ROM ist Teil einer Vorschrift der Bundeswehr. Sie wurde international vorgestellt, unter anderem auf der **Tri Ada 97** in New Orleans. Ihre Auflagenhöhe liegt auch bei rund 30 000. Die nationale und internationale Nachfrage ist bis auf den heutigen Tag groß. Es müssen weitere Auflagen geprägt werden.

1997 und für das Jahr **1998** hat das Verteidigungsministerium eine Studie und eine prototypische Nachweisführung beauftragt, daß es möglich ist, mit Hilfe von Ada 95 und ihren automatischen Compiler- Schnittstellen zu anderen Sprachen (Cobol, Fortran und C), in Verbindung mit dem Ada 95/JAVA- Compiler und CORBA, über Internet/Intranet alte, monolithische Fachinformations- und Führungsinformationssysteme in moderne Client-Server-Architekturen kostengünstig zu überführen („Reengineering“).

Ada 95 als Architektursprache für Altsysteme und aber auch für aktuelle Baukastensysteme aus COTS- Bausteinen („Clue-Software“).

Warum diese Möglichkeiten der Sprache nicht nutzen?

So weit die Antworten auf die Ausgangsfrage, was hat das Verteidigungsministerium getan?

Nun zu einer weiteren Fragestellung.

11.2 Wie war und ist die Resonanz in der Bundeswehr und in ihrem Wehrtechnik- und Beschaffungsamt?

Das Verteidigungsministerium hat dem **B**undesamt für **W**ehrtechnik und **B**eschaffung drei Erfahrungsberichte in den Jahren 1992, 1995 und 1997/8 abverlangt und darüber hinaus die Rechen- und Programmierzentren der Teilstreitkräfte um Erfahrungsberichte über die zur Verfügung gestellten Ada- Compiler gebeten. Die Berichte sind in ihren Umfängen und Befragungstiefen unterschiedlich.

Ergebnis der Aktion 1992

Die Schwerpunkte der ersten Umfrage waren:

- Bestandsaufnahme der Ada- Projekte,
- Einsatzerfahrung,
- Werkzeugeinsatz und
- Ausbildungsaspekte.

Der Erhebungszeitraum umspannte die Zeit von **1983 bis 1992.**

Es wurden damals **64** relevante Projekte ausgewertet. In **44** Fällen, das entspricht **69%**, wurde Ada eingesetzt, dabei **13** mal oder in **20%** der Fälle ausschließlich und bei weiteren **31** Projekten oder **48%** in einer Sprachkombination. Dabei war in 75% der Fälle „C“ die Komplementärsprache. In **20** Projekten oder **31%** fand Ada keine Anwendung. Der Sprachanteil von Ada in den in Frage kommenden Projekten reichte von 20% bis hin zu 100%, in 12 Fällen lag er über 98%.

Nach dem Erlaßzeitpunkt 1989 stieg die Anwendung von Ada erwartungsgemäß an - nach Aussage der Befragung - , aber es verbirgt sich hier ein Pferdefuß, der sich erst in den späteren Befragungen zeigen wird, denn in Realisierung und in Planung befindliche Projekte wurden gleichgewichtet in die Auswertung einbezogen. Die mit Ada geplanten Projekte wurden aber in manchen Fällen nicht in Ada realisiert.

Die Gründe, Ada einzusetzen oder in Sprachkombinationen zu verwenden oder sich dagegen zu entscheiden, wurden kombiniert wie die Sprachanwendung selbst. Bei Ada-Projekten standen Auswahlgründe wie beispielsweise

- Vorschriftenlage mit 79%,
- Internationale Vereinbarungen mit 16%, aber auch in hohem Maße
- Wartbarkeits- und Zuverläßigkeitsaspekte mit 49% bzw. 28%

im Vordergrund.

Der ersten Grund - Vorschriftenlage - war nach dem Inkrafttreten des Ada- Erlaßes natürlich dominierend. Der zweite Grund spiegelte die internationale Akzeptanz der Sprache wieder. Der dritte und vierte Grund waren ein Indiz des technischen Vertrauens in die Eigenschaften der Sprache, sie zeigten den hohen Erwartungswert bezüglich der von Ada propagierten Eigenschaften. Gründe wie Kosten oder Zeit oder der Gebrauch von Industriestandards hatten wenig Bedeutung, kumuliert waren dies um die 15%.

Kosten-/ Zeitfaktoren spielten dagegen aber eine erhebliche Rolle bei der Auswahl anderer Sprachen mit Prozentwerten von 37% bzw. 19%.

Diese Werte signalisierten 1992, daß einer erfolgreicheren Anwendung von Ada noch erhebliche Defizite, besonders in technischer Sicht, entgegenstanden.

Betrachtet wurden von den Befragten nur die Entwicklungskosten, nicht aber die Lebenszykluskosten, die letztlich jedoch zu der Entwicklung von Ada beigetragen hatten.

Auf Grund der Vorschriftenlage haben in den Vorhaben bzw. Projekten generell keine Kostenvergleiche stattgefunden mit Ausnahme von drei Fällen. Die Werte sind nicht repräsentativ, dennoch interessant:

- Im ersten Fall war die Vergleichssprache „PEARL“. Das Ergebnis war Kosten- und Zeitneutralität untereinander.
- Das zweite Vorhaben signalisierte Mehraufwände von 300% für Kosten und für Zeit bei der Vergleichssprache „C“.

- Im dritten Projekt wurde es exotisch, die Vergleichssprache war META-S & META-M. Es wurden Mehrkosten von einer Million DM angegeben, ohne jedoch die Gesamtkosten des Projektes anzugeben.

Alle Aussagen sind Ausdruck größter Unsicherheit.

Bis hier läßt sich feststellen, daß

- Erlaßlage und internationale Vereinbarungen Kosten- und Zeitbetrachtungen in den Hintergrund treten ließen.
- Mehraufwände an Kosten und an Zeit auf der anderen Seite dann der Grund sind, andere Sprachen zusätzlich einzusetzen.
- bei einigen Ada- Projekten beobachtet wurde, daß bei einer Hälfte die Kosten und bei einem Drittel der Zeitverbrauch anstiegen.
- jedoch auch Projekte beobachtet wurden, bei denen Kosten- und Zeitreduktionen aufgetreten waren.

Diese Aussagen stellten ein Gemenge dar, sie waren jedoch Trendhinweise, da sich viele der Projekte noch in der Entwicklungsphase befanden und gesicherte Aussagen folglich nicht möglich waren. Lifecycle- Kostenbetrachtungen konnten noch nicht angestellt werden.

Die Untersuchung von 1992 hatte auch Hinweise ergeben, daß die Faktoren „Ada-Entwicklungsumgebung“ , „Ada- Schulung , -Schulungsaufwand neben -Schulungsressourcen“ und „Ada- Know- how- Defizite“ signifikante Auswirkungen auf Kosten und Zeit hatten und sicherlich auch noch haben. Sie haben die Qualität von Knock-out-Tropfen. Diese Erkenntnis überraschte generell nicht, die Folge nur war die Notwendigkeit, beim Einsatz von Ada für die Entwicklungsphase einen höheren Aufwand einzuplanen. Die propagierten, technischen Eigenschaften der Sprache standen außer Frage.

Die Befragung erbrachte Erkenntnisse, die möglicherweise zu Problemen werden könnten. Angesprochen wurden:

- Forderung nach großzügig dimensionierten Hardwareausstattungen für den Einsatz mächtiger Software-Tools.
- Amtsinterne Unterstützung wie fachkundige, permanente Vorhabenbegleitung, Qualitätssicherung und Güteprüfung.
- Differenzierte Compiler für spezielle Zielrechner.
- Ada- Schnittstellen zu handelsüblicher Software. Das Fehl führte und führt teilweise heute noch zur Nutzungsbeschränkung und Zusatzkosten.
- Extrem hohe Compilerpreise auf Grund der Komplexität der Sprache und geringer Verkaufsstückzahlen. Die Preise waren ein Ausschlußkriterium für Hochschulen und Industrie.
- Die Portabilität des Ada- Codes ist trotz validierter Compiler selten gewährleistet.
- Mangelndes Ada- Know- how der Industrie durch fehlende Ausbildungskapizitäten, beispielsweise an Universitäten.

- Problematik hardwarenaher Programmiernotwendigkeiten.
- Mangelhafte Multiprozessorunterstützung.

Heute bleibt festzustellen, daß es 1992 ein Bündel an technischen und aber auch monitärer Problemen gab, die den flächendeckenden Einsatz der Sprache Ada massiv behinderten.

Insbesondere machten sich die fehlenden Ada- Compiler- Systeme für Spezialrechner und die fehlenden Schnittstellen zu handelsüblicher Software negativ bemerkbar, sie verhinderte weitere Anwendungen.

Dem Bericht von 1992 war weiterhin zu entnehmen, daß „Aufgrund der angespannten Finanzlage der Haushalte in der Zukunft verstärkt damit zu rechnen sei, daß Kostenfaktoren der initialen Entwicklungsphase größere Dominanz erhalten werden und somit die langfristig wirkenden Vorteile von Ada übertrumpfen könnten.“

Das Untersuchungsergebnis läßt sich durch wenige Sätze ergänzen:

- Ada wird in den Streitkräften erfolgreich eingesetzt, beispielsweise bei der Realisierung von Entwicklungsumgebungen und bei Realzeitsystemen.
- Erfahrungen aus der Nutzung lagen noch nicht vor.
- Ada zeichnet sich durch richtungsweisende Konzepte aus, die zwangsläufig zu hochwertiger Software führen.
- Das Ziel, mit Ada die Sprachvielfalt durch eine Standardsprache zu ersetzen, wurde nicht erreicht.

Soweit zu dem Blick in die Vergangenheit. Über die Konsequenzen, die das Verteidigungsministerium gezogen hat, habe ich bereits in dem ersten Teil berichtet.

1995 wurde das BWB mit einer weiteren, aber kürzeren Untersuchung beauftragt. Die Gründe hierzu waren mehr psychologischer Art, wir wollten das BWB und seine Vorhaben- bzw. Projektmanager zum einen sanft an Ada und die Erlaßlage erinnern und zum anderen auf Ada 95 „streßfrei“ vorbereiten. Diese Befragung gehörte zu unserer Strategie der Akzeptanzverbesserung der Sprache innerhalb der Bundeswehrvervaltung.

11.3 Ergebnis der Aktion 1995

Im Mai 1995 wurde der Ada- Beauftragte im BWB angewiesen, die Erlaßkonformität der DV-berührten Vorhaben über den Berichtzeitraum September 1992 bis März 1995 zu überprüfen. Die Fragestellung war nicht technisch, sondern managementseitig orientiert. Das Ministerium wollte lediglich wissen wie es 1995 in den Projekten um Ada 83, zum Zeitpunkt des Erscheinens des Standards Ada 95, gestellt war. Es wurde eine Auflistung derjenigen Vorhaben gefordert, die Ada erlaßkonform einsetzten, der Vorhaben, für die eine Ausnahmegenehmigung vorlag und der Vorhaben, die beides nicht erfüllten.

In die Befragung einbezogen wurden alle zur Stichzeit laufende und den Bereichscontrollern bekannte Vorhaben. **205 Vorhaben** wurden insgesamt zur Verwendung von Ada befragt.

146 Vorhaben davon waren DV-Vorhaben oder Vorhaben mit DV-Anteilen.

In **49 oder 33,5%** der Fälle wurde Ada eingesetzt.

In **81 oder 55,5%** der Fälle wurde Ada nicht eingesetzt.

In **5 oder 3,5%** der Fälle lag eine Ausnahmegenehmigung vor.

In **11 oder 7,5%** waren die Vorhaben noch in der Planungsphase, eine Sprachentscheidung lag noch nicht endgültig vor, wobei sich eine 50%- Verteilung abzeichnete.

Als allgemeine Aussage ergab die Umfrage:

- Die Verwendung von Ada ist in den Anwendungsbereichen **und** Abteilungen des BWBs sehr unterschiedlich.
- Für Waffen- und Waffeneinsatzsysteme und Simulationssysteme wird Ada verwendet. Die Situation kann als erlaßkonform bezeichnet werden.
- Der Bereich der Führungsinformationssysteme ist durch den vermehrten Einsatz von markt- oder sonstig verfügbarer Software („**C**ommercial-/**G**overnment-/**M**ilitary-off-the-shelf“) gekennzeichnet. Hier benutzen die Vorhaben- bzw. Projektverantwortlichen aus risikomindernden Gründen für die Erstellung der Restfunktionalität und Integration der einzelnen Produkte Entwicklungsumgebungen, die von den Herstellern der handelsüblichen Produkte bereitgestellt werden. Ada spielt hier eine untergeordnete Rolle.

Analysiert man die Gesamtheit der Vorhaben (Projekte), die Ada **nicht** einsetzen, so ergibt sich, daß die Führungsinformations- und Informationssysteme hier überproportional vertreten sind. Ihnen folgen die ELOKA- Systeme neben den „Altsystemen“, die leistungserhaltenden und -steigernden Maßnahmen, ohne Sprachmigrationen durchzuführen („Reengineering!“), unterzogen werden. In einigen Fällen handelt es sich auch um internationale Projekte, in denen Ada eingesetzt wurde.

Ein bewußtes Ignorieren des Spracherlasses bei neuen Systemen kann in unter 10% der Fälle angenommen werden.

Im Vergleich der einzelnen Unterabteilungen/Referate im BWB, die in ihren Projekten mit DV zu tun haben, reicht die Spanne des Einsatzes von Ada von 0% bis zu 100% in einem Falle. Für künftige, weitere Verbreitung der Sprache im Bundeswehrbereich sind jene Organisationsbereiche von großer Bedeutung, in denen der Spracheinsatz aus einer historischen Entwicklung heraus ausgewogen ist und die neuen Projekte Ada als Programmiersprache ausweisen.

Analysiert man die Projekte oder Vorhaben, die Ada einsetzen, so stellt man verblüfft fest, daß die Bundeswehrführung, das Heer, die Luftwaffe und die Marine sich ihrer bedienen und dies für alle Art von Systemen, in denen DV vorkommen kann. Folgende Beispiele belegen dies:

- Die Bundeswehrführung setzt (bzw. plant) sie in dem Führungsinformationssystem „RUBIN“ oder dem Satellitenkommunikationssystem „SATCOM“ ein.
- Das Heer betreibt den Analyserechner des Spürpanzers „Fuchs“ mit Ada oder den Schießsimulator des Schützen- PZ „MARDER 1“ oder die Rechner des Flak- PZ 1A2 ebenso wie die Anlagen der VHF und HF FmAufkl Mobil HEER, der ELOKA- Zentrale VHF und diverse Drohnen.
- Die Luftwaffe setzt Ada u.a. ein in dem Digitalen Informationsverarbeitungssystem der Lw, den Raketensystemen HAWK und PATRIOT und in Entwicklung befindlicher Nachfolgesysteme. Ada wird eingesetzt im EF 2000 (98%), dem MIDLIFE UPDATE des TORNADOs, den Hubschraubern NH-90 und PAH-2/TIGER und UHU und den Nachtflugsimulatoren für die Hubschrauber UH-1D und CH-53 (Materialverantwortlichkeit für fliegendes Gerät liegt bei der Lw).
- Die Marine verwendet Ada in ihrem U-Jagd- und Langstreckenaufklärungs- LFZ MPA-90 ebenso in der Fregatte Kl. 124 oder der Kampfwertsteigerung der Schnellboote, dem einen oder anderen Unterwasser- Ortungssystem oder in der „Nächstbereich-Fla-Waffenanlage RAM“.

Als Fazit dieser Befragung zeigt sich einerseits, daß prinzipiell jedes Projekt mit Ada angegangen und durchgezogen werden kann, man muß es aber eben nur wollen. Die Beispiele zeigen dies überdeutlich.

Anderseits wird aber auch klar, daß Ada 1995 die führende Sprache für die „Embedded Systems“ der aktuellen Waffensystementwicklungen aller Teilstreitkräfte ist.

Eine Auswertung der Ada- Lehrgangsauslastung der Jahre 1991 bis 1994 an der Bundesakademie für Wehrverwaltung und Technik zeigt eine Lehrgangsbelegung von 95% in den Jahren 1991, 1992 und 60% in den Jahren 1993, 1994. Der Eifer nach dem Erlaßjahr 1989 sinkt bei dem BWB- Personal stetig ab.

Die Luftwaffe als einzige Teilstreitkraft bildet ihr Personal an der Technischen Schule 1 mit voller Lehrgangskapazität bis auf den heutigen Tag aus. Sie wird folglich in kommenden Jahre befähigt sein, die Softwarepflege- und Änderung für ihre neuen Waffensysteme durchführen zu können, sie hat Expertise gesammelt und hinreichend Personal ausgebildet.

Das Ministerium hatte festgelegt, mit der Marktverfügbarkeit der neuen, validierten Ada 95- Compiler und der Einstellung der Zertifizierung von Ada 83 Compilern, März 1998, die Mittel zur unmittelbaren Unterstützung der Durchsetzung von Ada/Ada 95 auf einen Grundsockel abzusenken. Es stand die Grundüberlegung dahinter, daß die Sprache sich bis dahin selbst durchgesetzt haben muß.

Persönlich bin ich der Überzeugung, daß dieses Ziel heute erreicht ist.

Das Ministerium wollte zu diesem Zeitpunkt Bilanz ziehen, ob das Ziel auch im eigenen Bereich erreicht wurde und beauftragte folglich eine letzte, diesmal wieder umfänglichere, Befragung.

11.4 Ergebnis der Aktion 1998

Die Ende 1997 durchgeführte Umfrage untersucht die Verbreitung der Programmiersprachen, insbesondere Ada, im BWB, weiterhin sollten Problembereiche bei der Auswahl der Programmiersprachen, sowie die Ansichten der Projektmanager zur weiteren Entwicklung der Programmiersprachen in der Bundeswehr aufgedeckt werden.

Die Befragten wurden aufgefordert, Aussagen zu laufenden und in den letzten zwei Jahren, 1996 bis 1997, abgeschlossenen Vorhaben mit wesentlichen, eigenentwickelten oder zu entwickelnden Software- Anteilen, zu machen.

Nicht berücksichtigt werden sollten die sogenannten CGM- Produkte (COTS/MOTS/GOTS), die über wenig oder keine Eigenentwicklungsanteile verfügen. Mehrere, zu einem Projekt gehörende, Vorhaben konnten zusammen gefaßt werden, um den Aufwand der Projektverantwortlichen zu begrenzen. Diese Vorgehensweise sollte sicherstellen, daß alle wesentlichen Vorhaben mit DV-Anteilen erfaßt werden. Die Auswertung erfaßt neben der Anzahl der Projekte auch ihre Größe durch die Umrechnung aller Angaben in Lines of code (LOC).

Auffällig war, daß viele Vorhaben, die die Programmiersprachen C/C++ verwenden, keine Angaben zum Umfang der Software machen konnten.

Insgesamt wurden **95** Fragebögen zurückgegeben, 18 Vorhaben davon setzen fast ausschließlich CGM- Produkte ein und werden somit nicht mehr weiter betrachtet. Es verbleiben 77 Vorhaben für die weiteren Erhebungen zum Spracheinsatz. Die Vorhaben verteilen sich auf folgende Organisationsbereiche:

- Zentraler Bereich der Bundeswehr: **38**
- Heer: **24**
- Luftwaffe: **14**
- Marine: **19**

Bei 16 Vorhaben wurden internationale Partner angegeben. Dominant ist hier der Einsatz von Ada und C/C++.

Auf Grund eines Vergleiches mit den Erhebungen der Jahre 1992 und 1995 muß davon ausgegangen werden, daß die zurückgereichten Fragebögen selektiv waren und nicht alle relevanten Vorhaben beantwortet wurden

Eine weitere statistische Unsicherheit resultiert aus der Tatsache, daß einzelne Fragen häufig unbeantwortet blieben. Der Vergleich der drei Befragungen ergibt, daß die Anzahl der Vorhaben, die in zwei der drei Umfragen auftauchen, klein ist.

11.5 Verteilung der Programmiersprachen im BWB

Es ergibt sich folgende Verteilung der eingesetzten Programmiersprachen bei den ausgewerteten DV-Vorhaben bzw. -Projekte:

C/C++	Ada	Pascal	Assembler	Fortran	PL/1	sonst
22	14,5	4,68	3,07	0,76	15,4	16,6
Vorhaben der Programmiersprachen, absolut						
29%	19%	6%	4%	1%	20%	22%
Vorhaben der Programmiersprachen, relativ						

Tabelle 1: Verteilung des Programmierspracheneinsatzes bezogen auf die Anzahl der Vorhaben

Da die Größe der Vorhaben sehr unterschiedlich ist, wurde eine weitere Auswertung mit Berücksichtigung des Umfangs/Größe des Softwareentwicklungsanteils der Vorhaben vorgenommen:

C/C++	Ada	Pascal	Assembler	Fortran	PL/1	sonst
15%	40%	4%	6%	1%	18%	16%
Prozent an gesamten LOC				ohne ? LOC		
4390	11296	1121	1817	271	5209	4484
LOC-bezogen absolut (kLOC)				ohne ? LOC		

Tabelle 2: Verteilung des Programmierspracheneinsatzes bezogen auf LOC

Werden außerdem durch Einsetzen des mittleren Umfangs der Software- Anteile diejenigen Vorhaben berücksichtigt, die keine Angaben zu Größe/Umfang der zu entwickelnden Software gemacht haben, ergibt sich folgende Verteilung:

C/C++	Ada	Pascal	Assembler	Fortran	PL/1	sonst
22%	31%	5%	7%	1%	18%	18%
Prozent an gesamten LOC				mit ? LOC		
7891	11296	1871	2403	318	6372	6406
LOC-bezogen absolut (kLOC)				mit ? LOC		

Tabelle 3: Verteilung des Programmierspracheneinsatzes bezogen auf LOC mit zus. Annahmen

Die Tabellen 1- 3 zeigen, daß Ada eher in großen DV-Vorhaben eingesetzt wird, da der vorhabenbezogene Anteil gegenüber dem LOC-bezogenem Anteil sehr viel kleiner ist.

Für C/C++ gilt, daß diese Programmiersprachen vorwiegend in Vorhaben mit kleineren DV-Anteilen eingesetzt wird. Große Ada-Vorhaben sind z.B. MLRS, EF2000, U- Boot 212 und COBRA.

Die unterschiedlichen Philosophien der Programmiersprachen Ada und C/C++ kommen hier zum Tragen: Ada mit den Vorteilen der relativen Fehlerfreiheit, Lesbarkeit und Eignung für komplexe Software- Vorhaben (Schlagwort: „Software- Engineering“) gegenüber C/C++ mit den Vorteilen beim Rapid Prototyping und Marktverfügbarkeit von Bibliotheken (Schlagwort: „quick and dirty“).

Betrachtet man die Verbreitung von Ada, PL/1 und C/C++ in den einzelnen Abteilungen, bzw. Unterabteilungen der IT II und IT III des BWBs, ergibt sich folgendes Diagramm:

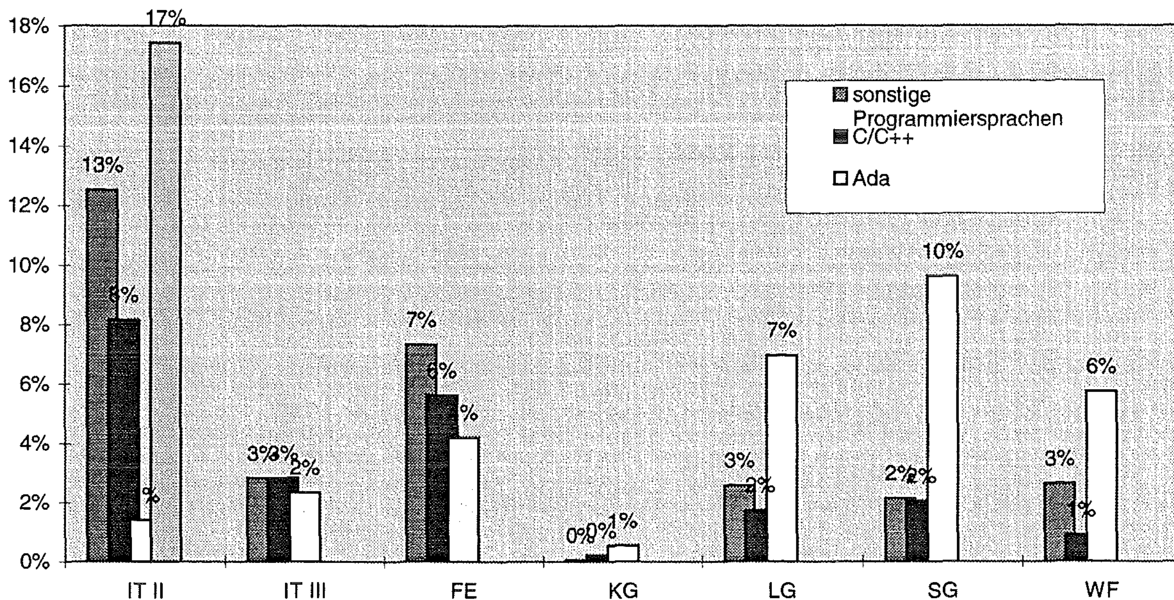


Tabelle4: Verbreitung von Ada, PL/1 und C/C++ in den Abteilungen des BWB

In der Auswertung wurden die Größe der Software- Projekte und die um die Mittelwerte ergänzten Vorhaben berücksichtigt.

In der Unterabteilung IT II (ehemalig BAWV) überwiegt die Verwendung von PL/1 aufgrund der anderen Programmiersprachenpolitik und Anforderungen an die vorhandene DV-Ausstattung des BAWV und der Rechenzentren der Bw (IBM-Host mit Betriebssystem MVS).

Die Verwendung von Ada überwiegt in den Abteilungen LG, KG, SG und WF, d.h. in den Abteilungen die überwiegend Waffensysteme (WS) und Waffeneinsatzsysteme (WES) betreuen. Bei diesen Vorhaben ist der Anteil von COTS-Produkten derzeit noch relativ gering.

In der Abteilung FE wird Ada und C/C++ zu ungefähr gleichen Anteilen eingesetzt. In diesen Vorhaben wird relativ viel COTS-Software eingesetzt. Graphische Darstellungen und die Integration von bestehenden, kommerziellen Software-Moduln besitzen dort

einen hohen Stellenwert. Der Einsatz von Ada ist in diesen Vorhaben häufig unwirtschaftlich und aufgrund der technischen Randbedingungen unmöglich.

11.6 Zeitliche Entwicklung der Anzahl Ada-Vorhaben

Bei der Darstellung der zeitlichen Entwicklung der Anzahl der Vorhaben, die die Programmiersprache Ada einsetzen, ist die unterschiedliche Gesamtanzahl der ausgewerteten DV-Vorhaben in den 3 Umfragen von 1992, 1995 und 1997 zu berücksichtigen. Aufgrund der unterschiedlichen Fragestellung ergibt sich bei der Umfrage 1997 im Vergleich zu 1995 oftmals eine Zusammenfassung von Vorhaben zu einem Gesamtprojekt.

Außerdem ist durch die organisatorische Umsetzung der IT- Referate vom BAWV zum BWB die Anzahl der Nicht- Ada- Vorhaben angewachsen.

	1992	1995	1997	
Anzahl Vorhaben in der Auswertung	64	146	95	(59 ohne IT II)
Anzahl Vorhaben mit Ada	44	49	21	(19 ohne IT II)
relativer Anteil	68 %	33,5 %	22 %	(31 % ohne IT II)

Tabelle 5: Zeitliche Entwicklung der Zahl der Vorhaben, die Ada verwenden

Bei der Umfrage von 1992 sind viele Vorhaben bewertet worden, die noch in der Definitions- bzw. Entwicklungsphase waren. Dabei war die Programmiersprache häufig noch nicht endgültig festgelegt. Es wurde aus Erlaßgründen Ada als „beabsichtigte“ Programmiersprache angegeben. In der jetzigen Umfrage und der Umfrage von 1995 überwiegen im Gegensatz dazu die Anteile von Vorhaben, die sich in der Beschaffungs- und Nutzungsphase befinden. Der Anteil der Ada- Vorhaben insgesamt ist eher rückläufig denn ansteigend .

11.7 Erfahrungen aus dem Einsatz von Ada

Die Erfahrungen der Vorhaben-Manager decken das gesamte Spektrum an negativen und positiven Möglichkeiten ab. (Auszug aus den Anmerkungen.)

- Zu wenig Erfahrung mit Ada bei der Industrie, daher geringe Verfügbarkeit geeigneter Programmierer; Systemstabilität und Fehlerfreiheit erhöht.
- Firma hat Ada- Erfahrung aus anderen Projekten, Schulungen; bei IT III 5 nur Erfahrungen über Ada- Lehrgänge; Integration des Auftraggeber- Personals in die Phase Integration als Einarbeitung für die Softwarepflege und -Änderung.

- Ada sollte unter gleichen Umständen zukünftig gemieden werden, da die Unterstützung und geschultes Personal fehlte, Software- Tools nicht ausgereift waren, Ada nicht genug Verbreitung fand, um Vorteile nutzen zu können.

- Die Forderung nach Ada wird von der Industrie unterstützt.

- Ada verspricht als HOL durch ihren modularen Aufbau Kosten- und Zeitersparnis. Endgültige Erfahrungen stehen noch aus, da noch in der Entwicklungsphase. Erfahrungen bisher positiv.

- Sehr große Anfangsschwierigkeiten wegen Mangel an ausgebildeten Ada-Programmierern und Problemen mit dem Alsys- Compiler und dem Runtime- System (mangelhafte Dokumentation + Support); HA sehr positive Erfahrung, Kosten, Effizienz, Qualität.

- Programmierung läuft ohne erkennbare Probleme.

- Positive Erfahrung bei der verteilten Entwicklung von Software- Einheiten in Ada. Effektive Fehlersuche wird durch implementierte Fehlerbehandlung und Lesbarkeit unterstützt. Multi-Prozessor- Anwendungen und zeitverteilte Anwendung werden wirksam unterstützt.

- Der Einsatz von Ada fördert ein stabiles Laufzeitsystem und stabile Applikationen.

- Die Erfahrungen mit Ada sind insgesamt positiv. Nach einer anfänglichen Lernphase kamen die positiven Aspekte zur Geltung; teilweise über 100 Programmierer; bessere Wiederverwendbarkeit durch gute Lesbarkeit.

- Teure Entwicklungswerkzeuge für Ada; Beispiele für C/C++ Compiler mit besserer Performance als Ada Compiler; COTS-Produkte meist in C/C++.

- Hoher Schulungs- und Einarbeitungsaufwand; hohe Lizenzkosten; Software- Tools nicht, bzw. verspätet verfügbar; Verkürzung der Integrationszeit; höheres Qualitätsniveau.

- Bei diesem Vorhaben handelte es sich um die Portierung eines großen Software-Systems von Dec/VMS auf HP-UX. Dabei hat sich erwiesen, daß auf Grund der verwendeten Programmiersprache sich eine Wiederverwendbarkeit ergab. Auch die Integration der neuen Anwendung ohne Probleme.

Zur „Ada-Politik“ wurden von den Vorhaben-bzw. Projektmanagern folgende Anmerkungen gemacht: (Auszug)

- Ada hat bei großen Projekten unbestreitbare Vorteile. Da allerdings COTS- Produkte selten in Ada entwickelt werden, sollte für die Einbindung bereits vorhandener Software nicht auf Ada bestanden werden. Da es nur sehr wenige Ada-Compiler-Hersteller gibt, kann dies zur Abhängigkeit führen.

- Falls der Erlaß ernsthaft umgesetzt werden soll, müßten die Vertragsreferate angewiesen werden, bei jeder Angebotsaufforderung zusätzlich ein entsprechendes

Angebot bei Entwicklung in Ada einzufordern. Voraussetzung für Ada ist die Schulung der Mitarbeiter.

- Die Wahl der Programmiersprachen ist von der Aufgabenstellung abhängig; für spezielle Anwendungsbereiche gibt es effektivere Programmiersprachen als Ada.

- In Hinblick auf den geforderten Einsatz von COTS-Produkten bildet Ada keine ideale Integrationsplattform.

- Ada- Vorteile insbesondere bzgl. Stabilität, Zuverlässigkeit und Portierbarkeit sind zu wenig bekannt (insbesondere im Nutzerbereich); direkte Ada- Politik ist nicht ersichtlich (zu weich bzgl. Einsatz anderer Sprachen, insbesondere bei Führungs-Informations- Systemen).

- Ada hat sich bei HFlaAFüSys sehr positiv ausgewirkt; Zuverlässigkeit, Wiederverwendbarkeit, Effizienz, gute Wartbarkeit, und Portierbarkeit; Anbindung an COTS-SW war oft nur über C- Interface möglich.

- Bereitstellung preiswerter Ada- Compiler und Entwicklungsplattformen (im Vergleich zu C/C++); bessere Durchsetzung des Sprachenerlasses; Ada als Pflicht an den Universitäten der Bundeswehr.

- 1)Schwer nachvollziehbar ist die Auflage Ada zu verwenden, andererseits möglichst COTS- Produkte zu verwenden; 2)Ebenfalls unverständlich ist, wenn Projekte Ada verwenden sollen und IT- Vorhaben sich an diesen Erlaß nicht halten.

- Ada sollte nicht zu restriktiv gefordert werden, da es oft viele Schnittstellen zu anderen Sprachen (C/C++) gibt, die berücksichtigt werden müssen.

- Der Beobachtungs- und Bewertungszeitraum der Effektivität des Einsatzes von Ada sollte nicht zu kurz angesetzt werden, um die Kosten von möglichst mehr als einem Projekt über dessen Lebensdauer oder die Wiederverwendung (Reuse) von Software- Einheiten in anderen Projekten zu erfassen.

- Solange sich der Auftraggeber nicht eindeutig zum Programmiersprachenerlaß bekennt, wird Ada nach wie vor eine geringe Rolle spielen.

- Bei Einsatz von COTS ist Ada in größerem Umfang nicht verwendbar, da auf zivilem Markt kein Industriestandard.

- Ada sollte weiterhin als Standard gefordert werden. Zukünftig werden jedoch vermehrt PC-Anwendungen mit COTS-Produkten, auch im Bundeswehr-Bereich, eingesetzt. Für die anwenderspezifische Programmierung bieten sich hier andere Sprachen an. Ada tritt in den Hintergrund.

Soweit zu der exemplarischen Auswahl der Kommentare und Erfahrungen der Vorhabens- bzw. Projektmanagern des Bundesamtes für Wehrtechnik und Beschaffung.

11.8 Fazit und Tendenz

Die Erfahrungsberichte überspannen den gesamten, 15-jährigen Lebenszeitraum von „Ada“, von 1983 bis 1998.

Die Pro- und Contra- Argumentationen der Projektmanager zu Ada berühren alle nur denkbaren Bereiche der Stärken und Schwächen, die technischen und organisatorischen, ebenso wie die menschlichen oder managementseitigen. Die rationalen, technischen und organisatorischen Stärken der Sprache müssen sich gegen irrationale menschliche Schwächen, die auch im Management auftreten, behaupten, aber auch gegen technische Mängel der Hilfsmittel, Stichwort ISO 9000.

Die Klagen über die Kosten der Compiler, über ihre Unzulänglichkeiten, die Beschwerdeführungen über mangelhafte Entwicklungsumgebungen, Debuggern und Softwaretools sind auch in der jüngsten Erhebung nicht verstummt, so wurde z.B. die Leistungsfähigkeit der letzten Version des ActiveAda- Compilers für Ada 83 wesentlich höher eingeschätzt als die des ObjectAda- Compilers für Ada 95, Version 7.0. Eher Gnade fand dann schon die Version 7.1.

Dies ist der Grundtenor der Erfahrungsberichte unserer Programmierzentren, denen ab 1995 Compilerpakete zur freien Nutzung und zum vorgabefreien Testen zur Verfügung gestellt worden waren.

Wir müssen alle, wie wir hier versammelt sind, an die Hersteller appellieren, ihre Produkte noch mehr ausreifen zulassen, um die Qualität und somit die Reputation ihrer Produkte zu steigern. Nur Qualität führt zum endgültigen Durchbruch der Sprache.

Für den Bereich der Bundeswehr hat sich Ada bei den Waffen- und Waffeneinsatzsystemen („Embedded Systems“), die in allen Teilstreitkräften eingesetzt werden, durchgesetzt; in anderen Bereichen eher fallweise, je nach Gusto des Projektmanagers.

Es bleibt zu hoffen, daß die notwendigen Renovierungen (Reengineering) der Führungsinformationssysteme der Streitkräfte sich der Möglichkeit des Umsetzens von den alten, wackeligen „Stand alones“ auf zeitgemäße, flexible „Client- Server“-Architekturen mittels des Einsatzes von Ada 95, in Verbindung mit JAVA- Anteilen und Intranet- Technologie, als äußerst preiswerter Architektursprache, bedienen werden.

Der Projektmanager des Führungsinformationssystems des Heeres, HEROS, im BWB hat einen Teil des Systems für den entsprechenden Prototypingversuch zur Verfügung gestellt.

Der Erfolg könnte der Durchbruch Ada's bei Informationssystemen sein, militärischen, aber auch privatwirtschaftlichen.

Siebert, Ingo, OTL a.D. war vom 01. März 1993 bis 31.März 1998 im BMVg, Rü VIII 2, Stv.Referatsleiter und Referent für Software-/System-Engineering , Sprachentechnologie und Grundlagen der IT- Ausbildung für IT- Fachpersonal.

Adresse:

Ingo Siebert
Liegnitzstr. 19
53721 Siegburg
Tel: 02241/64541

Ada im praktischen Einsatz

12 Ada im praktischen Einsatz

Prof. Dr. Erhard Plödereder
Universität Stuttgart
ploedere@informatik.uni-stuttgart.de

12.1 Kurzfassung des Referats

Die Übernahme neuer Technologien in die industrielle Praxis ist bekanntlich ein sehr langwieriger Prozeß. Gegenüber anderen Technologien, etwa im Hardwarebereich, zeigt sich, daß das Ersetzen alter durch modernere Programmiersprachen in der Industrie, wenn überhaupt, erst mit extrem langer Verzögerung erfolgt.

Betrachtet man die Historie der als erfolgreich zu bezeichnenden, heute dominierenden Sprachen, so sind sie nahezu ohne Ausnahme dadurch gekennzeichnet, daß mit ihrer Einführung ein neues Applikationsgebiet erschlossen werden konnte, das bislang nicht mit einer anderen Sprache vorbelegt war. Für Fortran war dies die Formulierung numerischer Algorithmen, für Cobol der Umgang mit großen Datenmengen der kommerziellen Datenverarbeitung, für C die enge Integration mit einem offenen Betriebssystem, UNIX, und für Java das „Computing" im WWW. Betrachtet man den heutigen Einsatz von Programmiersprachen in diesen Gebieten, so sind jene Originalsprachen auch nach 30-40 Jahren immer noch prädominant, typischerweise in einer mehr oder weniger kosmetisch überarbeiteten Form, die aber auch bereits 13-25 Jahre alt ist (Cobol-74 und Cobol-85, Fortran-77, ANSI-C). Progressive Versuche deutlicher Sprachänderungen (Fortran-90) haben kaum Akzeptanz gefunden; lediglich C++ als Teilablösung von C stellt eine Ausnahme dar. Charakteristisch für diese Sprachen ist auch (mit Einschränkungen bei Cobol), daß ihr Erfolg durch die „Vermarktung" seitens großer EDV-Firmen entstand. Einige dieser Sprachen haben sich in der Folge auch in benachbarten Anwendungsbereichen plaziert, in denen eine von der Großindustrie angebotene Sprache fehlte, so zum Beispiel C im Bereich der „embedded systems". Eine Ablösung etablierter Sprachen hat nur dort auf breiterer Basis stattgefunden, wo mit der Einführung einer neuen Sprache ein Quantensprung entstand: die deutlichen Vorteile einer höheren Programmiersprache gegenüber der Assemblerprogrammierung, die prinzipiellen Vorteile der anwendungsspezifischen Generatorsprachen (4GL) gegenüber der COBOL-Programmierung.

Festzuhalten ist in jedem Fall, daß die Industrie Programmiersprachen als eine Technologie ansieht, deren Aktualisierung in bestehenden Anwendungsgebieten sehr niedrigen Stellenwert besitzt, es sei denn, ein Quantensprung scheint erreichbar. Nur so ist die extrem lange Verzögerung zwischen Konzeptentwicklung und industriellem Einsatz im Bereich der Programmiersprachen zu erklären. Allenfalls „aufwärts kompatible" Verbesserungen werden mit erheblicher Verzögerung akzeptiert und dort vermutlich auch oft nur, weil die Infrastrukturkosten des Verbleibs in einer Vorgängerversion zu hoch werden. Die Vermarktung von C++ hat hier geschickt die Argumente der Aufwärtskompatibilität zu C und der Objektorientierung als sprungartige

Verbesserung verbunden. Diese Innovationsträgheit ist nicht nur in den Altlasten zu wartender Systeme begründet; in zunehmendem Maße wird auch das Kapital wiederverwendbarer oder kommerziell erhältlicher Komponenten zu einem mitentscheidenden Faktor. Lediglich ein „Druck von außen“ durch den Kunden kann die Projekt-Industrie dazu bringen, eine neue Programmiersprache einzusetzen, wie das im Falle von Ada durch das Mandat des DoD geschehen ist.

Vor diesem Hintergrund stellt sich die Frage nach der Einordnung von Ada und ihren Zukunftschancen. Mit Ada ist im engeren Sinn kein neues Anwendungsgebiet verbunden und keine der Großfirmen vermarktet Ada als bevorzugte Programmiersprache. Damit hat und hatte Ada es sehr schwer, neue Marktanteile im Rahmen einer allgemeinen Technologieerneuerung zu erobern. Eine plötzliche „Adaphorie“ ist daher auch in Zukunft nicht zu erwarten. Durch das DoD Mandat wurde aber ein nicht-trivialer Marktanteil für Ada erschlossen. In diesem Marktanteil arbeitet die Neuerungsträgheit der Industrie nun zu Gunsten von Ada. Nur wenige Firmen werden, wenn nicht wiederum von außen gedrängt, eine einmal etablierte Ada-Technologie durch eine andere Sprache in absehbarer Zeit ersetzen. Echte Verluste sind allenfalls dort zu erwarten, wo Ada als Zweitsprache eingeführt wurde und die Kosten einer doppelten Infrastruktur nicht vertretbar sind. Ganz abgesehen davon zeigt sich ziemlich einheitlich, daß, nach der oft schmerzhaften und kostspieligen Umstellung, Ada gegenüber Konkurrenzsprachen deutliche Vorteile bringt und eine Ablösung von Ada aus technischen und wirtschaftlichen Gründen nachteilig oder zumindest unnötig ist. Durch den Wegfall des DoD Mandats wird sich daher der bestehende Ada Markt kaum ändern; Zugewinne werden aber erheblich schwerer fallen.

Neben diesen Betrachtungen zur industrieweiten Technologieerneuerung gilt sicherlich auch, daß im Einzelunternehmen die Ablösung einer verwendeten Programmiersprache erfolgen wird, wenn diese sich für die wachsenden Anforderungen als nicht mehr geeignet erweist. Dieser Zeitpunkt ist aber sehr schwer auszumachen. Zu Recht wird man in dieser Situation das Problem erst im „Prozeß“ suchen, nicht bei der Programmiersprache. Dabei wird leider oft übersehen, daß eine Programmiersprache keineswegs prozeßneutral ist, da sie als sprachliches Ausdrucksmittel wesentlichen Einfluß auf die Einhaltung eines einmal etablierten, besseren Prozesses besitzt und, wie Erfahrungen mit Ada zeigen, auch dorthin lenken kann, aber auch bei unpassender Wahl zu einem zusätzlichen Störfaktor des Prozesses werden kann.

In diesem Kontext muß Ada mit anderen Programmiersprachen konkurrieren, insbesondere auch mit jenen Sprachen, die gerade „im Trend liegen“ ungeachtet ihrer jeweiligen Eignung für das jeweilige Anwendungsgebiet. Was spricht nun bei dieser Entscheidungsfindung für Ada ? In vielen Fällen wird in einer derartigen Situation ein abstrakter Sprachenvergleich anhand der angebotenen Sprachelemente und vielleicht der Infrastrukturaspekte gemacht und nach einem „Scoring-Modell“ eine Rangliste der Kandidaten erstellt. Dieses Vorgehen ist meines Erachtens fast immer von vornherein zum Scheitern verurteilt. Zum einen wird die Beurteilung der Sprachen immer durch vorgefaßte Meinungen der Beurteiler und durch deren Vertrautheit mit den jeweiligen Sprachen beeinflußt, so daß die Bewertung sehr schnell zum Feigenblatt einer bereits gefaßten subjektiven Meinung wird. Zum anderen sind auch objektiv durchgeführte Vergleiche letztlich nicht besonders aussagekräftig, da sie nur Eigenschaften beurteilen (z.B., Abstraktionsgrad, Ausdrucksstärke, Sicherheitselemente, usw.), die für sich genommen sekundär sind. Primäre Aspekte, nämlich die Einflüsse auf das zu

erstellende Produkt und seine Entwicklungskosten, sind nur höchst indirekt aus diesen Eigenschaften ableitbar und bleiben daher wenig überzeugendes Postulat.

Ein besserer Weg ist, aus den Erfahrungen anderer zu lernen. Hier setzt nun ein Studie an, die Herr Windholz an der Universität Stuttgart vor kurzem durchgeführt hat. In dieser Studie ging es primär darum, die Literatur auf Daten hin zu sichten, die Auskunft über die Auswirkungen von Ada im praktischen Einsatz in realen Projekten geben. Diesem Vorgehen kann man gleich entgegenhalten, daß die publizierten Ergebnisse aus Einzelprojekten aufgrund der mannigfaltigen Einflußfaktoren nicht zu einer generellen Aussage mißbraucht werden sollten. Dieser Einwand ist sicherlich richtig. Wenn sich allerdings Ergebnisse bei den Einzelprojekten konsistent wiederholen, dann kann man sich der Plausibilität einer Tendenz nicht entziehen, die mit hoher Wahrscheinlichkeit in anderen Projekte replizierbar ist. Insbesondere gilt dies, wenn die Aussagen aus den Ergebnissen vieler Projekte abgeleitet sind, die nach einigermaßen einheitlichen Kriterien in einigen, zum Großteil kommerziell verfügbaren Datenbanken erfaßt werden.

Die wesentlichen für Ada positiven Tendenzen in den veröffentlichten Projektdaten lassen sich wie folgt zusammenfassen:

- Die in Ada geschriebene Software weist zum Zeitpunkt des Integrationstests und der Erstauslieferung signifikant niedrigere Fehlerraten auf als die in anderen Sprachen geschriebene Software.
- Bei großen Systemen nehmen die Entwicklungskosten als Funktion der Produktgröße in Ada weniger schnell zu als in anderen Sprachen.
- Dauer und Kosten der Implementierungs- und Testphase nehmen für Ada einen proportional deutlich geringeren Anteil an den Entwicklungskosten ein als für andere Sprachen.

Einige negativen Tendenzen sind aber auch zu erkennen:

- Der verkürzten Implementierungs- und Testphase steht eine verlängerte Entwurfsphase gegenüber. Kosteneinsparungen bei den gesamten Entwicklungskosten sind, ausgenommen bei großen Systemen, nicht zu erwarten.
- Tendenziell erscheint die Entwicklung in C++ für kleinere und mittlere Systeme aus Sicht der Entwicklungskosten etwas billiger, ein Nachteil, der durch die Wartungskosten wieder wettgemacht wird.

Leider sind die veröffentlichten Daten zu mittel- und langfristigen Wartungskosten von Ada-Software zu spärlich, um diesbezügliche generelle Tendenzen belegen zu können. Die wenigen verfügbaren Daten weisen deutlich reduzierte Wartungskosten für Ada Software aus.

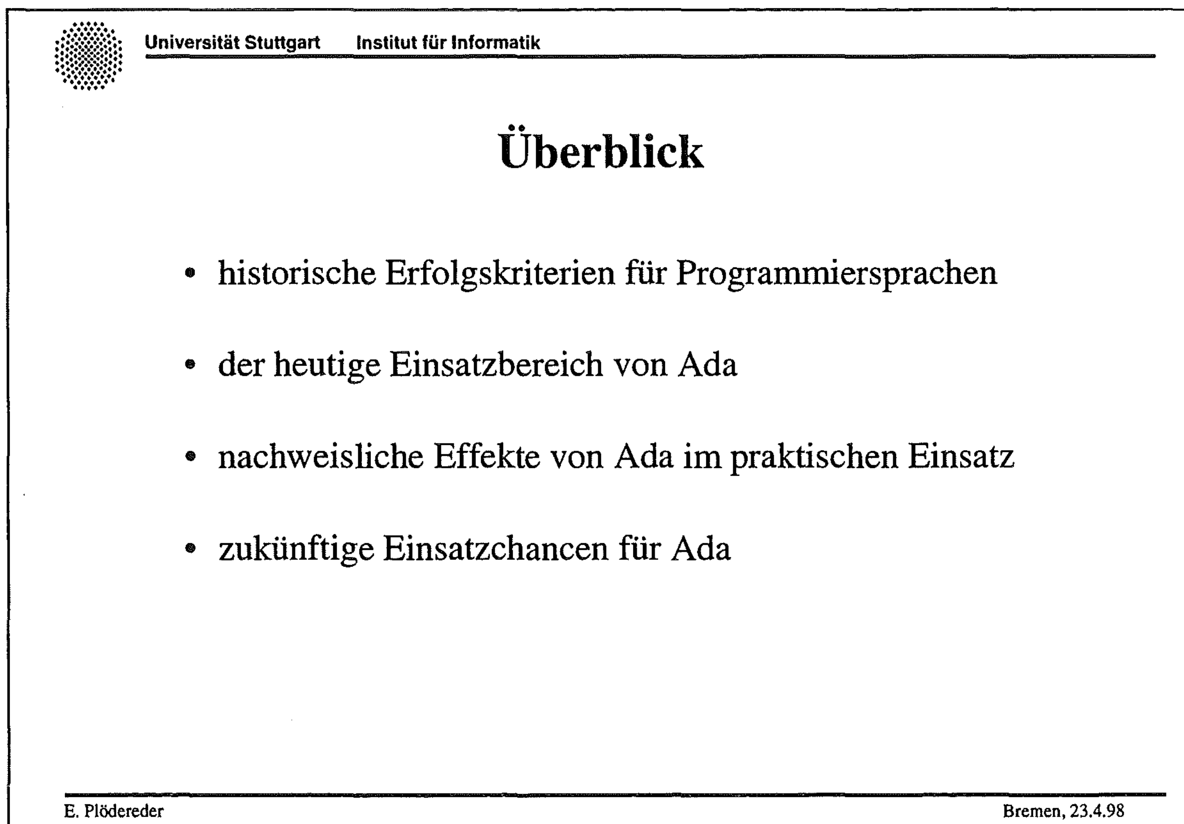
Einige Details der Studie werden auf den folgenden Folien präsentiert. Die Studie und eine annotierte Liste der Literaturquellen sind im WWW verfügbar unter

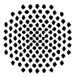
http://www.informatik.uni-stuttgart.de/ifi/ps/Publn/

Es ist meines Erachtens müßig, über die Ursachen der präsentierten Daten zu diskutieren. Sie bestehen als statistische Fakten, wie etwa die Leistungszahlen, anhand derer Rechnerhardware verglichen wird.

Welche Schlüsse lassen sich nun aus diesen Daten für die zukünftigen Marktchancen von Ada ziehen, wenn sich der Markt an diesen Erfahrungen orientiert ? Die Antwort ist offensichtlich: „große Systeme“ und „kritische Systeme“ (als Oberbegriff all jener Systeme, in denen verstärkt auf möglichst geringe Fehlerraten der ausgelieferten Software abgehoben wird). Gerade letzterer Bereich ist in einem enormen Wachstum begriffen, sowohl im Volumen als auch der Komplexität der Anwendungen, so daß hier ein deutliches Wachstumspotential für Ada gegeben wäre. Es ist aber auch anzumerken, daß dieser Bereich in der kommerziellen Anwendung fest in Händen von C zu sein scheint und damit C++ als „kompatible“ Ablösung deutliche Marktvorteile besitzt. Es bleibt abzuwarten, ob die nahezu universelle Ablehnung von C++ in der universitären Ausbildung hier langfristig Wirkung zeigen wird und Ada als eine bessere Alternative Marktanteile gewinnen kann.

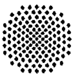
12.2 Folien





Erfolgreiche Sprachen 1998

- **FORTRAN**: unbesetzter Bereich numerischer Anwendungen
- **COBOL**: unbesetzter Bereich kommerzieller DV
- C: systemnahe Programmierung mit BS-Integration, UNIX
- C++: "kompatibler" Ersatz für C mit OOP Zugabe
- **Java**: unbesetzter Bereich des WWW Computing
- (**Basic:** PC-Programmierung)

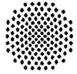


historische Erfolgskriterien

- neue Sprache für einen unbesetzten Anwendungsbereich
- Vermarktung durch EDV-Großfirmen
- "Quantensprung" in Produktivität und Kosten
- Kompatibilität mit Vorgänger

nicht ausreichend:

- (inkompatible) technologische Verbesserungen des Bestehenden

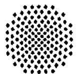


Universität Stuttgart Institut für Informatik

Heutiger Einsatz von Ada

- durch DoD und MoD Mandate geschaffener Markt
- Anwendungen im öffentlichen Bereich in Anlehnung an die Mandate
- visionäre Firmen auf der Suche nach einer besseren Sprache

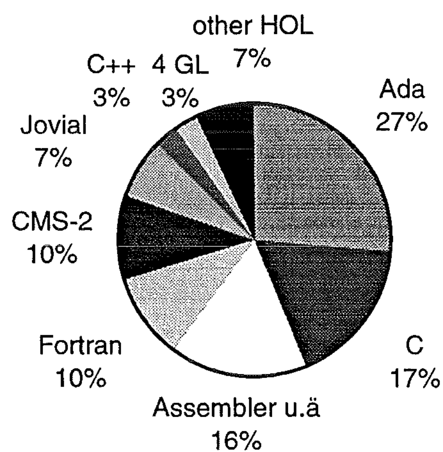
E. Plödereder Bremen, 23.4.98



Universität Stuttgart Institut für Informatik

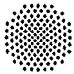
US DoD Weapon Systems

Basis: 187 MSLOC in Weapon Systems



Quelle: 1995 IDA Survey aller 1994 RDT&E Projekte des DoD mit Budget > 15M$

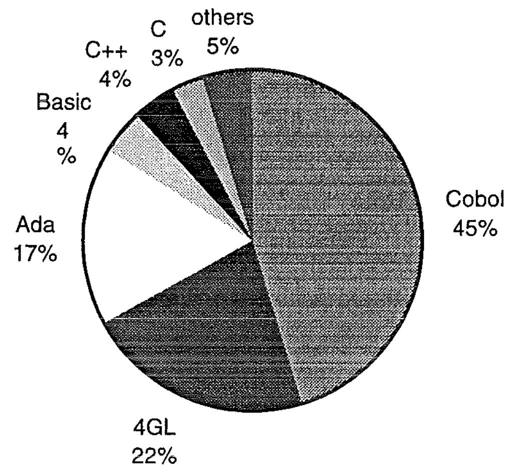
E. Plödereder Bremen, 23.4.98



Universität Stuttgart Institut für Informatik

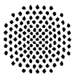
US DoD Information Systems

Basis: 50 MSLOC in Information Systems



Quelle: 1995 IDA Survey der 53 großen DoD IS-Projekte im RDT&E Budget 1994

E. Plödereder Bremen, 23.4.98



Universität Stuttgart Institut für Informatik

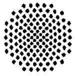
Nachweisliche Effekte von Ada im praktischen Einsatz

durch Analyse statistischer Daten aus abgeschlossenen Projekten

Einige umfangreiche Datenbanken mit derartigen Daten sind kommerziell verfügbar, meistens im Zusammenhang mit dem Kauf eines Werkzeugs zur Planung und Kostenschätzung von Projekten.

Im folgenden sind publizierte Teilauswertungen aus derartigen Datenbanken wiedergegeben.

E. Plödereder Bremen, 23.4.98



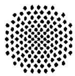
Universität Stuttgart Institut für Informatik

Fehlerraten (pro kSLOC)

	Ada	C	C++	Norm
Information Systems (kom.)	4,0	7,0	5,1	7,0
Information Systems (mil.)	3,0	6,0	4,0	6,0
Commercial Products	2,8	5,0	3,0	4,0
Telecommunication (kom.)	1,6	2,0	1,7	3,1
Telecommunication (mil.)	1,0	1,5	1,2	2,5
Weapon Systems (ground)	0,5	0,8	0,7	1,0
Weapon Systems (air/space)	0,3	0,8	0,6	1,0

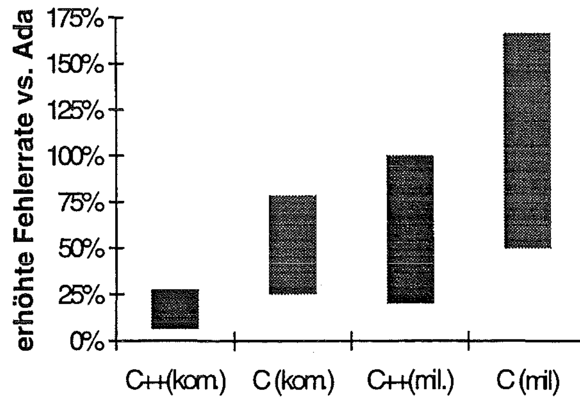
Quelle: Reifer 1996, 190 Projekte 1993-96, Norm aus 1500 Projekten 1989-96

E. Plödereder Bremen, 23.4.98

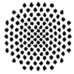


Universität Stuttgart Institut für Informatik

relativer Fehlerzuwachs gegenüber Ada



E. Plödereder Bremen, 23.4.98



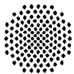
SLOC(Ada) = SLOC(C++) = SLOC(C) ??

Tendenziell:

- deutlich mehr "deklarative" Zeilen in Ada als in C
- Einzelvergleiche scheinen zu zeigen, daß
  - Ada für kleine Programme "zeilenreicher" ist
  - Ada für größere Programme "kondensierter" ist (Vermutung: mehr Wiederverwendung als Folge besserer Spezifikationen)

Im Schnitt: siehe SLOC für Function Points ...

siehe auch: Zeigler, "Comparing Development Costs for C and Ada"

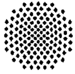


SLOCs pro Function Point

	Min	Av.	Max
C	60	128	170
Ada83	60	71	80
C++	30	53	125
Ada95	28	49	110

(werden von SPR aus einer DB über ca. 5000 Projekte jährlich ermittelt)

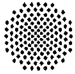
Quelle: Capers Jones, SPR, 1995 (Auszug aus Liste mit 450 Sprachen)



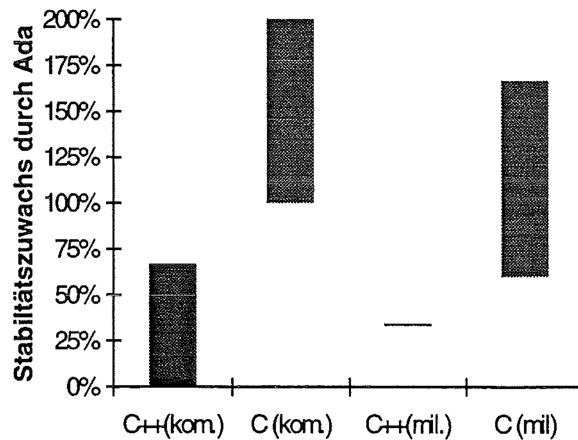
"Mean Time (weeks) to Major Repair Incident"

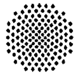
	Ada	C	C++	Norm
Commercial Products	1,0	0,4	1,0	0,5
Information Systems (mil.)	0,8	0,5	k.A.	0,5
Information Systems (kom.)	1,0	0,5	0,6	0,5
Telecommunication (kom.)	3,0	1,0	2,0	1,8
Telecommunication (mil.)	4,0	2,0	3,0	2,0
Weapon Systems (ground)	6,0	3,0	k.A.	2,5
Weapon Systems (air/space)	8,0	3,0	k.A.	2,5

Quelle: Reifer 1996, 190 Projekte 1993-96, Norm aus 1500 Projekten 1989-96



relativer Stabilitätszuwachs durch Ada





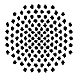
Universität Stuttgart Institut für Informatik

Kosten ($) pro SLOC

	Ada	C	C++	Norm
Information Systems (kom.)	k.A.	25	25	30
Information Systems (mil.)	30	25	25	35
Commercial Products	35	25	30	40
Telecommunication (kom.)	55	40	45	50
Telecommunication (mil.)	60	50	50	75
Weapon Systems (ground)	80	65	50	75
Weapon Systems (air/space)	150	175	k.A.	200

Quelle: Reifer 1996, 190 Projekte 1993-96, Norm aus 1500 Projekten 1989-96

E. Plödereder Bremen, 23.4.98



Universität Stuttgart Institut für Informatik

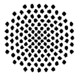
Kosten und Produktivität

	SLOC/PM	$/SLOC	Datenpunkte
Norm	183	70	543
Ada	210	65	153
C++	187	55	23
Ada, 1. Projekt	152		38
C++ 1. Projekt	161		7

Anm: Die CTA Studie macht den Versuch, den Widerspruch in den Daten durch höhere QA-Anteile u.ä. in Ada-Projekten zu erklären.

Quelle: CTA Studie 1991 (größtenteils auf DB von Reifer)

E. Plödereder Bremen, 23.4.98



Universität Stuttgart Institut für Informatik

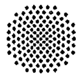
Lernkurve und Produktivität

vom 1. zum 3. Projekt in Ada:
- Reifer 87: +25% SLOC/PM Zuwachs
- NASA SEL 89: +50% Zuwachs, -40% Fehlerrate

weitere Produktivitätsauswertungen:
- NASA SEL 89: ~700 SLOC/PM; 1,0 Fehler/kSLOC
- ESA DB 96: ~400 SLOC/PM (vergl. C: ~260 SLOC/PM)
- Reifer 87: 121-415 SLOC/PM

Reifer 87: Auswertung von 41 Ada Projekten
NASA SEL 89: Report to the Information Resources Mgmt. Council
ESA DB 96: Maxwell et al (IEEE TSE, V22/10, 1996): 99 ESA Projekte, 34 in Ada, 9 in C

E. Plödereder Bremen, 23.4.98



Universität Stuttgart Institut für Informatik

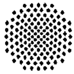
weitere statistische Aussagen

- Reifer 87: die Entwicklungskosten sind in Ada sublinear (Exponent: ~0,95) von der Projektgröße abhängig.

- ESA DB 96: Nach "Firma" ist "Sprache" am stärksten mit Produktivität korreliert.

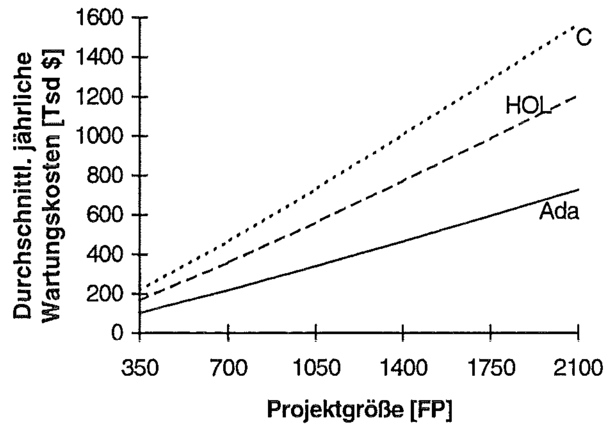
- Reifer 87: Die Kostenverteilung
  "Entwurf : Implementierung : Test"
  ist für Ada 50:15:35, während die Industrienorm eher bei 40:20:40 liegt.

E. Plödereder Bremen, 23.4.98



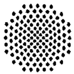
Universität Stuttgart Institut für Informatik

Wartungskosten



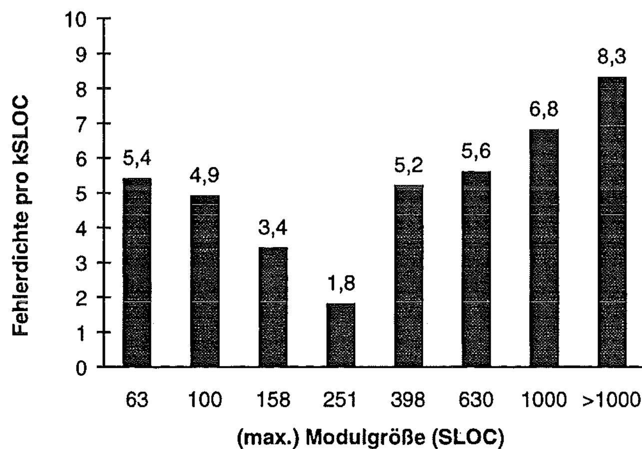
Quelle: MITRE, 95; Auswertung mit dem Martin Marietta PRICE-S Werkzeug zur Kostenschätzung; mittlere jährliche Kosten über einen 10-jährigen Wartungszeitraum

E. Plödereder Bremen, 23.4.98



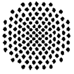
Universität Stuttgart Institut für Informatik

Fehlerdichte vs. Modulgröße



Quelle: Withrow, IEEE Software, Jan. 1990; ähnliche Ergebnisse bei Shen, Basili, Perricone

E. Plödereder Bremen, 23.4.98

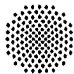


Universität Stuttgart Institut für Informatik

zukünftiger Marktbereich für Ada

1. existierende Anwender
2. große, komplexe Systeme
3. kritische Systeme (Systeme, in denen die Fehlerquote möglichst niedrig zu halten ist)

E. Plödereder Bremen, 23.4.98



Universität Stuttgart Institut für Informatik

weitere Informationen

Literaturverweise u.ä.:

http://www.informatik.uni-stuttgart.de/ifi/ps/Publn

E. Plödereder Bremen, 23.4.98

Am Ende des Übergangs von Ada 83 zu Ada 95: Brauchen wir noch Ada Compiler Validierungen?

13 Am Ende des Übergangs von Ada 83 zu Ada 95: Brauchen wir noch Ada Compiler Validierungen?

Michael Tonndorf
IABG
Ada Validation Facility
85521 Ottobrunn

Tel.: 089 6088 2477
Fax: 089 6088 3418
Tonndorf@iabg.de

Abstract

Nach den Regelungen der *Ada Validation Procedures* endete die Gültigkeit aller Zertifikate, die unter ACVC 1.11, 2.0 und 2.0.1 ausgestellt wurden, am 31. März 1998. In diesem Papier wird eine Bilanz gezogen über die Entwicklung der ACVC Testsuite und die Entwicklung des werkzeuggestützten Validierungsverfahrens bei der Validierungsstelle IABG. Anschließend wird ein Ausblick auf die zu erwartende weitere Entwicklung der Ada Validierung gegeben.

Schlüsselworte: Ada Compiler Validierung, Ada Standardisierung, Ada Sprachpflege.

1.1 Einleitung

Die Programmiersprache Ada wurde zuerst von ANSI im Jahr 1983 standardisiert. Von ISO wurde die Standardisierung 1987 nachvollzogen. Zur Nachweisführung, daß ein Compiler die Sprache gemäß der Sprachdefinition implementiert, wurde die Ada Compiler Validation Capability (ACVC) Testsuite entwickelt. Das US-DoD, Sponsor der Sprache und größter Anwender von Ada-Produkten, verlangte, daß in von ihm beauftragten Projekten nur validierte Compiler zum Einsatz kommen.

Im September 1984 wurde die IABG als Ada Validierungsstelle in Deutschland akkreditiert. Ihre erste Validierung führte die IABG im November 1984 durch. Nach drei Jahren manueller Validierungen wurde begonnen, Unterstützungswerkzeuge für die Validierung zu entwickeln. Die Entwicklung der Werkzeuge steht in Verbindung mit der Verbreitung von portablen Computern und dem Internet. Seitdem hat die IABG 145 Validierungen durchgeführt und ist eine der zwei aktiven Validierungsstellen weltweit.

Dieses Papier ist wie folgt aufgebaut: In Kapitel 2 findet sich eine Zusammenfassung der Prinzipien der Ada Compiler Validierung. Kapitel 3 enthält eine Darstellung der Erfahrungen der IABG und der Entwicklung der Validierungswerkzeuge. In Kapitel 4 wird die Entwicklung zusammenfassend bewertet und ein Ausblick auf die Zeit nach der Schließung des Ada Joint Program Office gegeben.

1.2 Prinzipien der Ada Compiler Validierung

1.2.1 Allgemeines

Mit der Validierung eines Ada Compilers soll demonstriert werden, daß ein Compiler die Sprache genau so implementiert, wie sie im Sprachmanual definiert ist. Es ist offenkundig, daß dieses Ziel auch nicht annähernd erreicht werden kann. Die Komplexität der Sprache macht es zu einem aussichtslosen Unterfangen, einen Compiler auf hundertprozentige Normkonformität prüfen zu wollen. Insbesondere die Kombination von zwei oder mehreren Sprachelementen, wie sie in realen Programmen alltäglich vorkommt, kann nicht mit akzeptablem Aufwand auf korrekte Implementierung getestet werden. Um trotzdem zu einem aussagekräftigen Verfahren zum Prüfen der Normkonformität zu kommen, wurde vom US-DoD das Ada Validierungssystem beauftragt und in Kraft gesetzt. Die Elemente des Systems sind

- die Organisationen, die bei der Validierung mitwirken,
- die Testprogramme einschließlich der Dokumentation (ACVC),
- das Regelwerk, das den Validierungsprozeß definiert und beschreibt. Dazu gehören insbesondere auch die Akzeptanzkriterien. Die Ada Validation Procedures [1] und das AVF Operation Manual sind die wichtigsten Dokumente in diesem Zusammenhang.

1.2.2 Überblick über den Validierungsprozeß

Die Ada Compiler Validation Capability (ACVC) ist die einzige und mit Einschränkungen fachlich akzeptierte Methode zum Prüfen der Normkonformität von Ada Compilern. Von den ACVC Tests werden aber keine Leistungswerte der Implementierung ermittelt. Es werden auch keine Optimierungsverfahren oder besondere Charakteristiken der Implementierung abgeprüft. Die ACVC Testsuite [3] ist eine Sammlung von Testprogrammen, die sich aus sechs Klassen zusammensetzt: A, B, C, D, E und L. Der erste Buchstabe eines Testnamens identifiziert die Klasse, zu der der Test gehört. Die Klassen A, C, D und E enthalten ausführbare Tests, die unter Zuhilfenahme von Report-Funktionen selbstablaufend sind und das Ergebnis in einem standardisierten Format ausgeben. B-Tests enthalten bewußte Sprachverletzungen, die vom Compiler erkannt und zurückgewiesen werden müssen. L-Tests enthalten intendierte Fehler, die beim Binden erkannt und zurückgewiesen werden müssen.

In der folgenden Tabelle 1 werden die Begriffe definiert, die im weiteren Verlauf des Papiers verwendet werden:

Begriff	Bedeutung
Testergebnis	Das Ergebnis eines ACVC Tests (im allgemeinen eine Datei mit einem Compiler/Binder-Listing oder die Ausgabe eines ausführbaren Tests)
Validierungs-ergebnis	Alle Testergebnisse einer Validierung zusammengenommen
Wertungs-kategorie	Eine der Kategorien *PASSED, FAILED, UNSUPPORTED, NOT APPLICABLE* (siehe unten)
Ergebnis-einstufung	Die Zuordnung eines Testergebnisses zu genau einer der Wertungskategorien
Auswertungs-ergebnis (Validierungs-ergebnis)	Alle Ergebniseinstufungen einer Validierung zusammengenommen

Tabelle 1

Die Anzahl der Test in den sechs Klassen A, B, C, D, E, L variierte im Laufe der Zeit. Die größte Anzahl von Tests wurde bei ACVC 1.11 mit 4.071 Tests erreicht. Die nachfolgend angegebenen Zahlen geben für ACVC 1.X größenordnungsmäßig die Verteilung der Tests auf die Testklassen an:

- 500 ausführbare Tests (Klasse A, C, D, E),
- 100 Binder Tests (Klasse L),
- 1500 Compiler Tests (Klasse B). Insgesamt gibt es in allen B-Tests zusammengenommen über 12.000 konstruierte Fehler, die vom Compiler erkannt werden müssen.

Für die aktuelle Version ACVC 2.1, die seit Juli 1997 in Kraft ist, gibt die nachfolgende Tabelle 2 die wichtigsten Kennzahlen wieder:

ACVC 2.1	Total	Core Tests	Specialized Needs Annexes	Foundation	Docu-ments	Sup-port
Anzahl Dateien	4242	3919	245	44	11	23
Anzahl Tests	3662	3474	188	-	-	-

Tabelle 2

Die Technik der Ada-Validierung kann an dieser Stelle nicht im Detail beschrieben werden. Die grundlegenden Ideen wurden 1993 bei der Ada-Europe-Tagung präsentiert [4].

Tabelle 3 gibt einen Überblick über die sechs Schritte einer Validierung. Diese Schritte sind ausführlich in den Ada Validation Procedures [1] beschrieben. Das AVF-Handbuch macht hierzu weitere detaillierte Ausführungen. Einer AVF ist die Methode der Durchführung einer Validierung freigestellt. Sie muß aber auch stets darauf vorbereitet sein, zu einem Audit zur Verfügung zu stehen.

Die Motivation für den Schritt *Prävalidierung* (siehe Tabelle 3) stammt aus den Zeiten, als ein Validierungslauf zwei oder mehr Wochen dauerte. Eine Prävalidierung wird vom Kunden in eigener Zuständigkeit durchgeführt. Damit sollte das Risiko für eine Validierung beim Kunden minimiert werden („Test Readiness Review“). Jedoch ist es immer noch gute Praxis, zumindest eine Prävalidierung für eine zusammengehörige Serie von Validierungen durchzuführen. Mit den Ergebnissen der Prävalidierung kann die AVF eine Datenbasis für werkzeuggestützte Vergleiche mit den Validierungsergebnissen aufbauen und somit den Arbeitsaufwand während der Validierung beim Kunden minimieren.

1.3 Beschreibung der Werkzeugumgebung CANDY

IABG begann 1988, eine Werkzeugunterstützung für Ada Compiler Validierungen zu implementieren. Die Entwicklung ist ausführlicher in [6] dargestellt. An dieser Stelle soll nur die heute im Einsatz befindliche Konfiguration des Werkzeugs CANDY vorgestellt werden.

1.3.1 Anforderungen an Werkzeuge und Prinzipien der Werkzeugentwicklung

Die folgenden Anforderungen lagen der Entwicklung der Validierungswerkzeuge zu Grunde:

- Geringer Lernaufwand und Benutzerfreundlichkeit,
- Wartbarkeit und Offenheit für Weiterentwicklung,
- Anpaßbarkeit an unterschiedliche Layouts von Listings und Compilerdiagnosen,
- Erhöhtes Vertrauen in die Korrektheit des Auswertungsergebnisses im Vergleich zur manuellen Methode, Nachvollziehbarkeit des Ergebnisses,

Schritt	Aktivität	Bemerkung
1	Vertrag über Validierungs-Dienstleistungen	Etabliert eine Kundenbeziehung mit der AVF
2	Prävalidierung	• erster Selbsttest in Eigenverantwortung des Kunden • Klärung offener Punkte • Wertung des Validierungsergebnisses durch die AVF • Aufbereitung der Ergebnisse als Vergleichsbasis für die Validierung
3	Validierung	Ablauf der kundenspezifisch angepaßten Testsuite pro zu testender Implementierung unter Aufsicht der AVF
4	Declaration of Conformance	Unterschriebene Erklärung des Kunden in der er bestätigt, den Ada Standard nicht absichtlich zu verletzen und keine Rechte dritter Parteien zu beeinträchtigen
5	Validation Summary Report	Ein Bericht in einer einheitlichen Struktur, in dem der Validierungsprozeß beschrieben wird und alle Validierungsergebnisse, die nicht PASSED sind, erläutert werden
6	Validation Certificate	Ein vom AJPO ausgegebenes Zertifikat, das die Implementierung, die ACVC Version, die Zertifikatsempfänger und das Verfallsdatum des Zertifikats verzeichnet

Tabelle 3

- Keine Verlagerung von Entscheidungen über Bestehen oder Nichtbestehen eines Tests in das Werkzeug, mit Ausnahme der normalen ausführbaren Tests,
- Unterstützung der Ergebnisauswertung, Berichtsfunktion/Statistik und Buchführung.

Zum Nachweis und zur Rekonstruktion des Testverlaufs werden alle Protokolldateien („logfiles") aufbewahrt. Diese Protokolldateien werden von CANDY jedoch nicht ausgewertet.

Im Gegensatz zu möglichen anderen Ansätzen gibt es bei der Methode der IABG keine „automatische" Wertung von B- oder L-Tests. Diese Idee wurde ausführlich diskutiert, aber dann doch verworfen. Die Tatsache, daß ein Werkzeug entscheidet, ob der Compiler einen B-Test „richtig" behandelt, wird nicht als akzeptabel angesehen. So

steht am Anfang jeder Kette von Dateivergleichen durch CANDY zuerst eine manuelle Bewertung des Testergebnisses.

Eine Validierung läuft immer in zwei Phasen ab, die untereinander verzahnt sein können:

- Ablauf der Testprogramme unter der zu validierenden Implementierung,
- Auswertung der Testergebisse.

Es liegt in der Zuständigkeit des Kunden, die Testergebnisse bereitzustellen. Er bringt die Testprozeduren zum Ablauf, wobei die AVF den Prozeß überwacht. Die Auswertung der Testergebnisse kann in separaten Schritten durchgeführt werden, sobald diese verfügbar sind.

1.3.2 Manuelle Ergebniseinstufung

Testergebnisse müssen immer dann manuell ausgewertet werden, wenn keine Werkzeugunterstützung verfügbar ist. Dies war insbesondere zu Beginn der Validierungsaktivitäten der IABG so. Zu diesem Zweck muß für jedes Testergebnis festgestellt werden, ob es in Übereinstimmung mit den Sprachregeln ist. Dazu muß für

- jeden ausführbaren Test der Ergebnisausdruck,
- jeden B-Test das Compiler Listing,
- jeden L-Test das Binderprotokoll ausgewertet und einer der Wertungskategorien zugeordnet werden.

Jede Ergebnisbewertung und Auffälligkeit ist zur Dokumentation in ein Logbuch einzutragen. Es ist offenkundig, daß die Auswertung eines Validierungsergebnisses eine zeitaufwendige Angelegenheit ist. Die vollständige Auswertung einer Validierung benötigt mindestens fünf Personentage. Dazu kommt noch einmal der Aufwand für die Buchhaltung von ungefähr drei Tagen. Falls auch die Prävalidierung zu berücksichtigen ist, dann verdoppelt sich der Aufwand.

Der beabsichtigte Gewinn durch CANDY liegt nicht primär in der automatischen Ergebniseinstufung. Dies geschieht nur für die ausführbaren Tests. Der wirkliche Vorteil liegt im Vergleichen von B-Test Ergebnissen mit bereits ausgewerteten Testergebnissen. Dadurch wird der manuelle Aufwand bei einer Validierung auf die Einstufung der Tests reduziert, deren Vergleichsergebnis nicht übereinstimmt. CANDY benötigt immer zwei Validierungsergebnisse, die es miteinander vergleichen kann: dies können das Prävalidierungsergebnis und das Validierungsergebnis für dieselbe Plattform sein, oder aber zwei Validierungsergebnisse von verschiedenen Plattformen. Erfahrungsgemäß war der automatische Vergleich zwischen den Validierungsergebnissen verschiedener ACVC-Versionen schwierig: nicht nur daß Tests geändert und hinzugefügt wurden. Oft verbesserten die Hersteller anläßlich des Versionenwechsels auch Phasen oder Pässe des Compilers, so daß die Ergebnisprofile erheblich voneinander abwichen.

Innerhalb der ersten drei Jahre als Validierungsstelle wurden alle Validierungen komplett manuell ausgewertet. Teil der ACVC Distribution war das *ACVC Logbook,* in das alle Ergebniseinstufungen der Tests eingetragen wurden. Das Logbook war der Nachweis des Fortschritts und des Status der Validierung. Für die vollständige manuelle Auswertung einer Validierung werden ungefähr acht Personentage benötigt, einschließlich der Buchführung. Wenn wie im Normalfall eine Prävalidierung dazukommt, dann verdoppelt sich der Aufwand.

Eine vollständige Validierung umfaßt jedoch noch mehr:

- kundenspezifische Aufbereitung und Auslieferung der Testsuite,
- Sonderbehandlung der modifizierten Tests,
- Bearbeitung aller implementierungsspezifischen Sonderfälle, die in der Regel in Test Disputes unter Einschaltung des AVO gelöst werden müssen,
- Überwachung und Begleitung von Installation und Ablauf der Testsuite,
- Erstellung des *Validation Summary Report.*

1.3.3 Werkzeuggestützte Ergebniseinstufung

Die manuelle Verwendung des *ACVC logbook* legte eine werkzeuggestützte Implementierung dieses Verfahrens nahe, zumal die bekannten Unix-Werkzeuge nicht den benötigten Leistungsumfang boten. Die Entwicklung des Werkzeugs CANDY (der Name wurde lautmalerisch nach dem Vergleich von *base* und *candidate results* kreiert) erfolgte in 4 Versionen, vom monolithischen Hostcomputer-Programm über ein portables Tool auf Unix-Basis bis zum Notebook-Tool mit TCP/IP-Anschluß. Die aktuelle Funktionalität ist in [6] ausführlicher dokumentiert. In [5] wurde erstmals ein Gesamtkonzept vorgestellt.

Die wesentliche Leistung von CANDY besteht darin, die Vergleichsresultate von Paaren von Testergebnissen zu ermitteln. Für jeden Test in der Testsuite sind die folgenden Vergleichsergebnisse möglich (siehe Tabelle 4):

CANDY Status	CANDY Report
Basis- und Kandidat-Ergebnis existieren und stimmen überein nach Maßgabe der Vergleichsmuster	match
Basis- und Kandidat-Ergebnis unterscheiden sich in Aspekten, die nicht durch die	mismatch

Vergleichsmuster abgedeckt sind	
Mindestens ein Ergebnis (Datei) fehlt oder ist unvollständig	no base, no candidate, none, incomplete

Tabelle 4

Diese Reports werden von CANDY für alle Arten von Ergebnissen erstellt. Für ausführbare Tests wird zusätzlich die Ergebniseinstufung ausgegeben. Das bedeutet, für alle Testergebnisse, die übereinstimmen (match), wird die Wertung des Basis-Ergebnisses auf das Kandidat-Ergebnis übertragen. Wenn die Ergebnisse übereinstimmen, dann ist also kein manueller Aufwand zur Ermittlung der Ergebniseinstufung notwendig. Die Erfahrungen von mehr als 80 Validierungen nach Beginn der Entwicklung der Werkzeuge haben gezeigt, daß 85% bis 99% aller Testergebnisse einer Validierung übereinstimmen, in Abhängigkeit von der Ähnlichkeit der Testergebnisse.

Eine typische Sitzung mit CANDY läuft folgendermaßen ab:

1. Bereite die erforderlichen Daten für einen CANDY-Lauf vor (Basis und Kandidat-Ergebnisse in Verzeichnisstruktur einordnen);

2. Starte das CANDY-Programm;

3. Setze die Optionen für einen CANDY-Lauf:
   - Pfadangabe zu den Basis- und Kandidat-Ergebnissen,
   - ggf. Einschränkung auf einen Ausschnitt aus der Testsuite,
   - Error- und Ignore-Patterns zur Steuerung der erlaubten Toleranzen;

4. Starte die Vergleichsausführung zum Vergleichen aller ausgewählten Ergebnispaare und zum Erzeugen des resultierenden *validation book*.

5. Bereite durch CANDY Übersichten über die Vergleichsergebnisse und die noch manuell zu bearbeitenden Testergebnisse auf.

6. Sichere und archiviere die Validierungsergebnisse, CANDY's Listen und Protokolldateien und das *validation book*.

Die Grafik in Bild 1 zeigt die grundsätzliche Arbeitsweise von CANDY: für jeden Test aus dem Logbuch wird das entsprechende Basis- und Kandidat-Ergebnis auf die *match*-Bedingung geprüft, in Abhängigkeit von den gesetzten Vergleichsmustern.

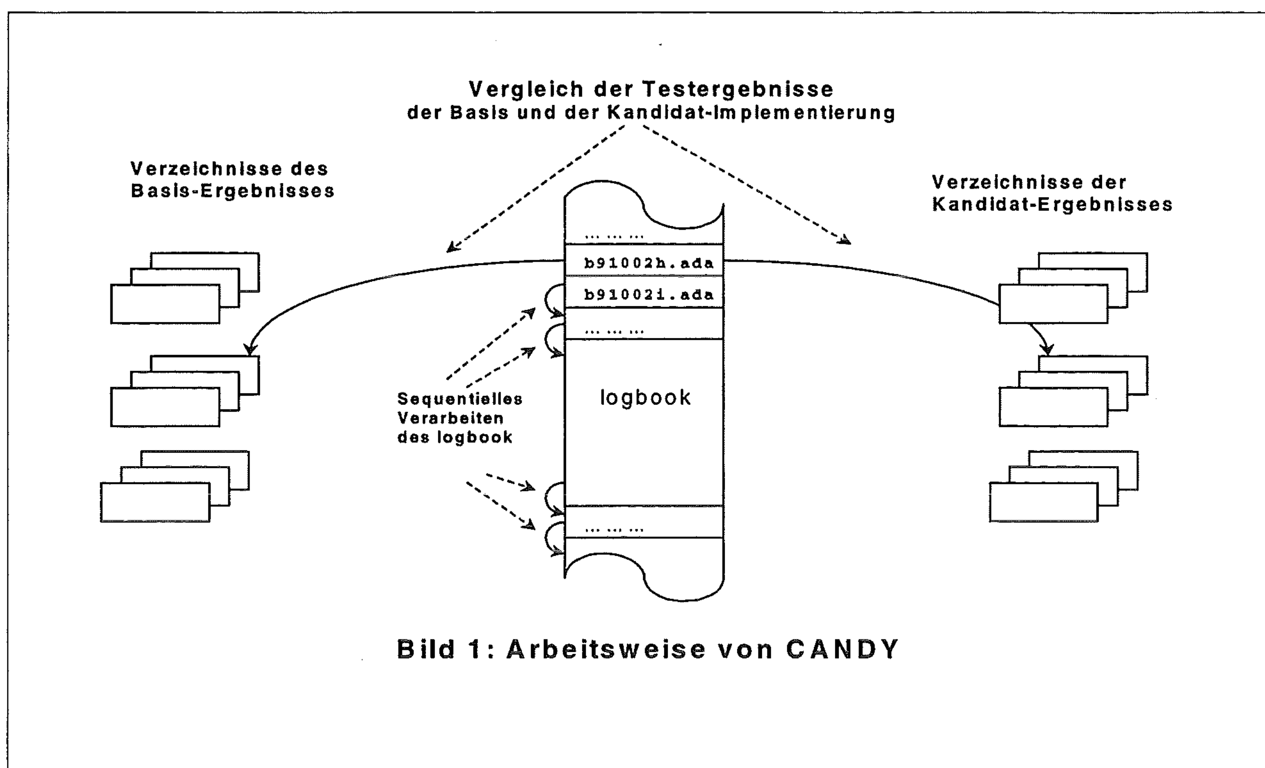


Bild 1: Arbeitsweise von CANDY

An manuellem Aufwand verbleibt die Überprüfung aller Ergebnisse, die nicht übereinstimmten. Darüberhinaus gibt es eine Anzahl von Tests, die immer individuell behandelt werden müssen, da über das reguläre Testergebnis hinaus zusätzliche Kriterien abgeprüft werden müssen (z.B. Informationen aus dem Ablaufprotokoll des Betriebssystems oder spezielle Informationen von Tests zu einem *Specialized Needs Annex*).

1.4 Zusammenfassung und Ausblick

1.4.1 Bewertung des erreichten Stands

Ada wurde u.a. als Sprache für sicherheitskritische Anwendungsbereiche entworfen. Deswegen werden an den generierten Code besondere Qualitätsanforderungen gestellt, wodurch sich wiederum hohe Anforderungen an den Compiler ergeben. Ein Ada-Compiler ist ein komplexes Produkt, für das parallel zur Standardisierung der Sprache mit großem Aufwand ein Validierungssystem entwickelt wurde. Wenn eine Validierung manuell durchgeführt wird, dann besteht die Gefahr, daß sich Fehler bei der Auswertung einschleichen. Außerdem ist das Verfahren personalintensiv und damit teuer. Die Erfahrungen der letzten zehn Jahre mit Ada Validierungen haben gezeigt, daß der Validierungsprozeß unter Zuhilfenahme von Werkzeugen wesentlich verbessert werden kann. Es verbleibt die Implementierung von Verwaltungsfunktionen (z.B. für Kundeninformationen), die aber den Validierungsprozeß selbst nicht beeinflussen.

Die Erfahrungen der letzten zehn Jahre belegen die Fortschritte: es konnten zuletzt im Juni 1997 zehn Validierungen in sechs Arbeitstagen erledigt werden. Vorausgegangen

war eine reguläre Prävalidierung. Der Bearbeiteraufwand belief sich auf acht Tage für die Prävalidierung, sechs Tage für die Prävalidierung zuzüglich drei für Verwaltungsaufgaben. Dies ergibt eine Summe von 17 Tagen. Ohne Werkzeugunterstützung hätte man für die zehn Validierungen zuzüglich der Prävalidierung 11*8, das heißt 88 Bearbeitertage benötigt. Dies bedeutet in der Summe eine Reduzierung des Aufwands um ca. 80% gegenüber dem rein manuellen Verfahren. Diese Kalkulation berücksichtigt nur Zeitaufwände, noch überzeugender sind natürlich die Kosteneinsparungen. Die durchschnittlichen Gebühren für eine Validierung liegen inzwischen unter 10% der Gebühren zu Beginn der Validierungstätigkeiten. Dies ist abhängig von der Anzahl der Validierungen, die parallel durchgeführt werden können.

Erwähnenswert ist auch, daß die Aktivitäten der IABG niemals in irgend einer Form subventioniert wurden. Alle Arbeiten an den Werkzeugen trugen sich intern aus den Gebühren für die Validierung. IABG würdigt hiermit das ausgezeichnete Engagement der Studenten von Staffordshire University, G.B., aus den letzten zehn Jahren. Sie haben einen wesentlichen Beitrag zur Fortentwicklung des Werkzeugs CANDY und des Validierungsverfahrens geleistet.

Unsere Erfahrungen haben gezeigt, daß die Validierung von Ada Compilern optimiert und ohne Reibungsverluste in den Entwicklungszyklus der Hersteller integriert werden kann.

1.4.2 Ausblick

Seit 1995 ist der fortgeschriebene Standard Ada 95 [2] in Kraft. Für die Gestaltung des Übergangs von Ada 83 zu Ada 95 legte man sich im AJPO auf einen Zwei-Jahres-Zeitraum von März 1995 bis März 1997 fest. Dieser Zeitraum wurde noch einmal bis zum 30. Juni 1997 verlängert. In dieser Zeit waren die ACVC-Versionen 2.0 und 2.0.1 in Kraft, wobei zum Erlangen eines Zertifikats nicht die Implementierung des vollen Sprachumfangs nachgewiesen werden mußte. Zum Ende März 1998 schließlich verloren alle Zertifikate nach ACVC 1.11 (Ada 83) und nach ACVC 2.0.(1) (Ada 95 eingeschränkt) ihre Gültigkeit. Damit existierten am Ende der Übergangszeit nach Ada 95 am 31. März 1998 genau drei Implementierungen, die gemäß ACVC 2.1 validiert sind (Green Hills Software).

Im November 1997 kündigte das US-Verteidigungsministerium an, daß es das Ada Joint Program Office spätestens im September 1998 schließen werde. Dieser Termin wurde mittlerweile auf den 30. Juni 1998 vorverlegt. Die Konsequenzen der Umsetzung dieser Ankündigung sind weitreichend:

- keine direkte Finanzierung einer Ada Infrastruktur,
- keine zentrale Beeinflussung der Entwicklung von Ada,
- keine Kontrolle der Validierung, wie bisher durch die “Ada Validation Organisation” (AVO) gegeben,
- Unsicherheit über die Zukunft der Informationspolitik, wie sie bisher durch das Ada Information Clearinghouse (AdaIC) sehr erfolgreich ausgeübt wurde.

Wohin sich die Ada Validierung bis zum Ende des Jahres 1998 entwickelt haben wird, ist derzeit nur schwer abschätzbar. Die vorherrschende Meinung ist, daß die Validierung ein wesentliches Element der Sprache Ada bleiben sollte. Und dies sehen sogar die meisten der Ada Hersteller so.

Die Fragen, die für die Zeit nach der Aufgabe der Verantwortung der Ada Validierung durch das US-DoD gelöst werden müssen, sind vielfältig:

- Wer bestimmt die “Sprachenpolitik”? (Die Frage “Wer” soll sich allgemein auf Insititutionen, Organisationen oder Gremien beziehen, nicht auf Einzelpersonen.)
- Wer überwacht die Weiterentwicklung der ACVC Testsuite im Sinne eines “Test Method Control Board”?
- Wer pflegt die ACVC Testsuite im Sinne eines “ACVC Team” weiter?
- Wer koordiniert die Arbeit der Ada Validierungsstellen (AVFs) – falls es auch in Zukunft mehr als eine geben wird?
- Wer hat die Autorität, im Zusammenhang mit Validierungen Entscheidungen zu treffen bezügl. Sprachinterpretation, Grenzfällen etc. (Umsetzung von Resolutionen der Ada Rapporteur Group)?
- Wer braucht ein Zertifikat zur Bestätigung einer erfolgreichen Validierung?
- Was sind die Kosten der einzelnen Aktivitäten?
- Wer kommt für die Kosten auf?
- Wie funktioniert die Ada Öffentlichkeitsarbeit in Zukunft, was kann man vom DoD und von DISA erwarten?
- Was ist die Rolle der Ada Resource Association (ARA) und was kann man von ihr erwarten?

Eine Diskussion über diese Fragen wurde auf der TRI-Ada Tagung 1997 in den USA initiiert. Dabei hat sich zwar das Augenmerk auf die ARA konzentriert. Auch wurden erste Dokumente im Sinne der Fortschreibung *der Ada Validation Procedures* mit ARA-Focus erarbeitet. Ganz offenbar sind die Ada-Hersteller in den USA aber nicht bereit, für die zu erwartenden Kosten aufzukommen. Auch kann von der ARA nicht eine solche Distanz zu den Herstellern erwartet werden, wie diese durch das DoD gegeben war.

Nun blickt man aus den USA mit gewissen Erwartungen nach Europa. Da in mehreren europäischen Ländern und in der NATO Ada weiterhin Standard ist, argumentiert man dort mit einigem Recht, daß Europa/NATO sich dann auch mehr um die Validierungs-Infrastruktur kümmern sollten. Schließlich habe man fast 15 Jahre lang in den USA das Validierungssystem aufgebaut und die Kosten dafür aufgebracht.

Die US-Validierungsstelle bei EDS mit Phil Brashear wird vermutlich auch weiterhin für Validierungen zur Verfügung stehen. Es wird jedoch ab dem Zeitpunkt der Schließung des AJPO in den USA kein von einer Regierungsstelle autorisiertes

Validierungszertifikat mehr geben.

Bis Juni 1998 haben sich weitere Punkte in der Diskussion um die Zukunft der Ada Validierung herauskristallisiert:

- Die ARA wird zwar Anlaufstelle für Ada Validierungen sein. Es wird jedoch eine unabhängige Autorität geben (AVA; *Ada Validation Authority*), die nur ISO-Gremien (ARG, WG9) verpflichtet ist. Die Ada Hersteller werden somit keinen direkten Einfluß auf die Regeln der Ada Validierung haben. Die AVA fungiert somit als Nachfolgerin der Ada Validation Authority.

- Für die Finanzierung der AVA sind Übergangsregelungen unter Einbeziehung des US-DoD im Gespräch.

- Die ACVC Testsuite wird bis auf Substanzerhaltung in absehbarer Zeit nicht weiterentwickelt.

Egal, welche Konstellation sich nach der Schließung des Ada Joint Program Office abzeichnet: Am Ende dieses Prozesses wird eine Ada Validierungsstelle eine größere Kompetenz haben und autonom bei der Ausstellung von Validierungszertifikaten sein.

Was Deutschland angeht, so ist man hier ohne Organisationsänderungen in der Lage, den Betrieb der Validierungsstelle bei der IABG fortzusetzen und vom DIN bestätigte Validierungszertifikate für ACVC 2.1 auszustellen. Darüber hinausgehende europäische Initiativen zur Unterstützung der Ada Validierung sind derzeit noch nicht bekannt.

13.1 Literatur

Zur Beachtung: Die URLs der Dokumente [1] und [3] stammen vom Februar 1998 und können sich im Laufe der Zeit ändern.

[1] Ada Compiler Validation Procedures, Version 5.0, 18 November 1997, Ada Joint Program Office , Center for Computer Systems Engineering, Defense Information Systems Agency (*http://www.sw-eng.falls-church.va.us /AdaIC/compilers/val-proc/val5_0fin.shtml*).

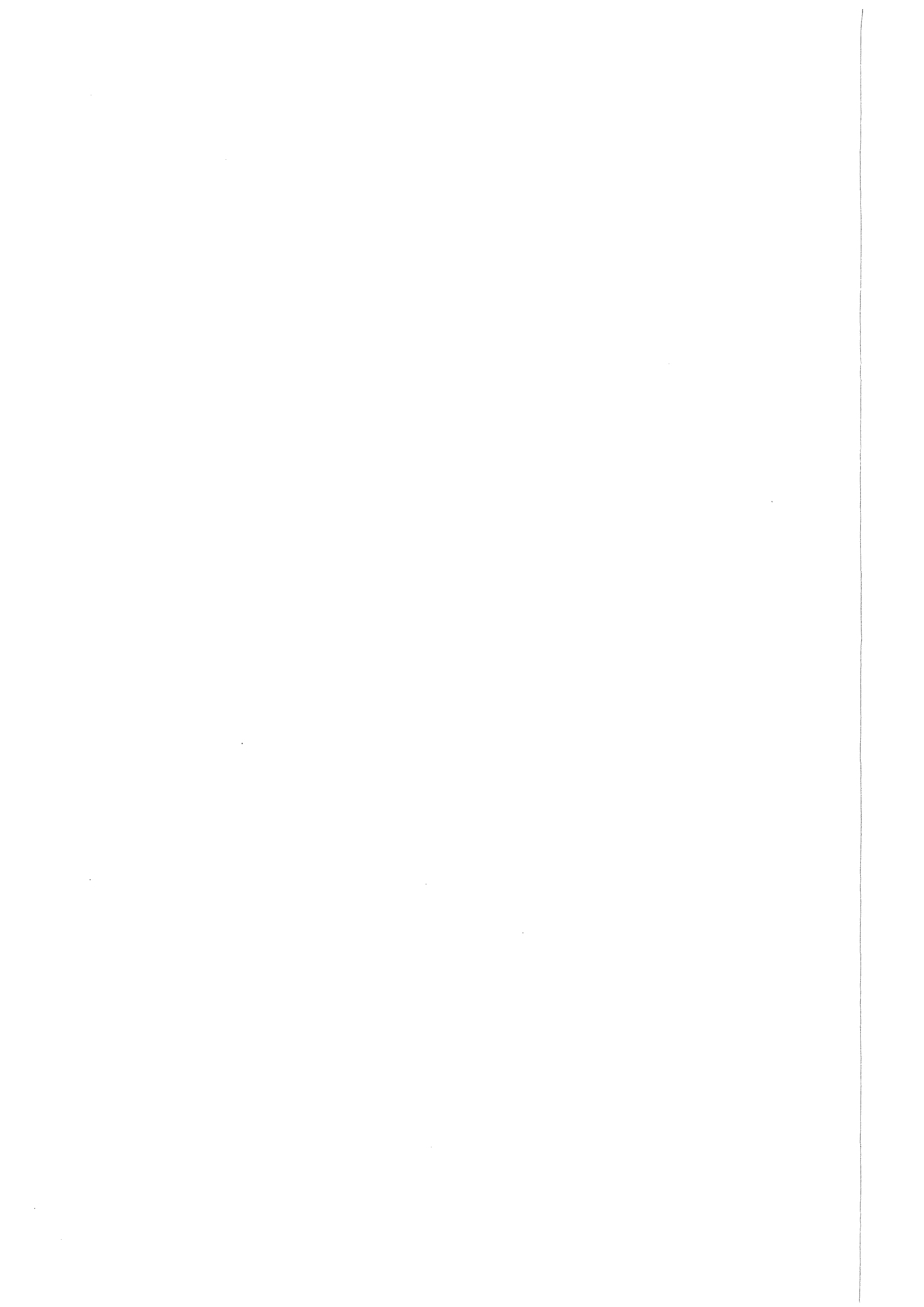
[2] Reference Manual for the Ada Programming Language, ANSI/ISO/IEC 8652:1995, FIPS PUB 119-1.

[3] The Ada Compiler Validation Capability (ACVC) Version 2.1 User's Guide, CTA Incorporated, Dayton, OH 45431, March 1997 (*http://www.sw-eng.falls-church.va.us/AdaIC/compilers/acvc/95acvc/acvc2_1/index*).

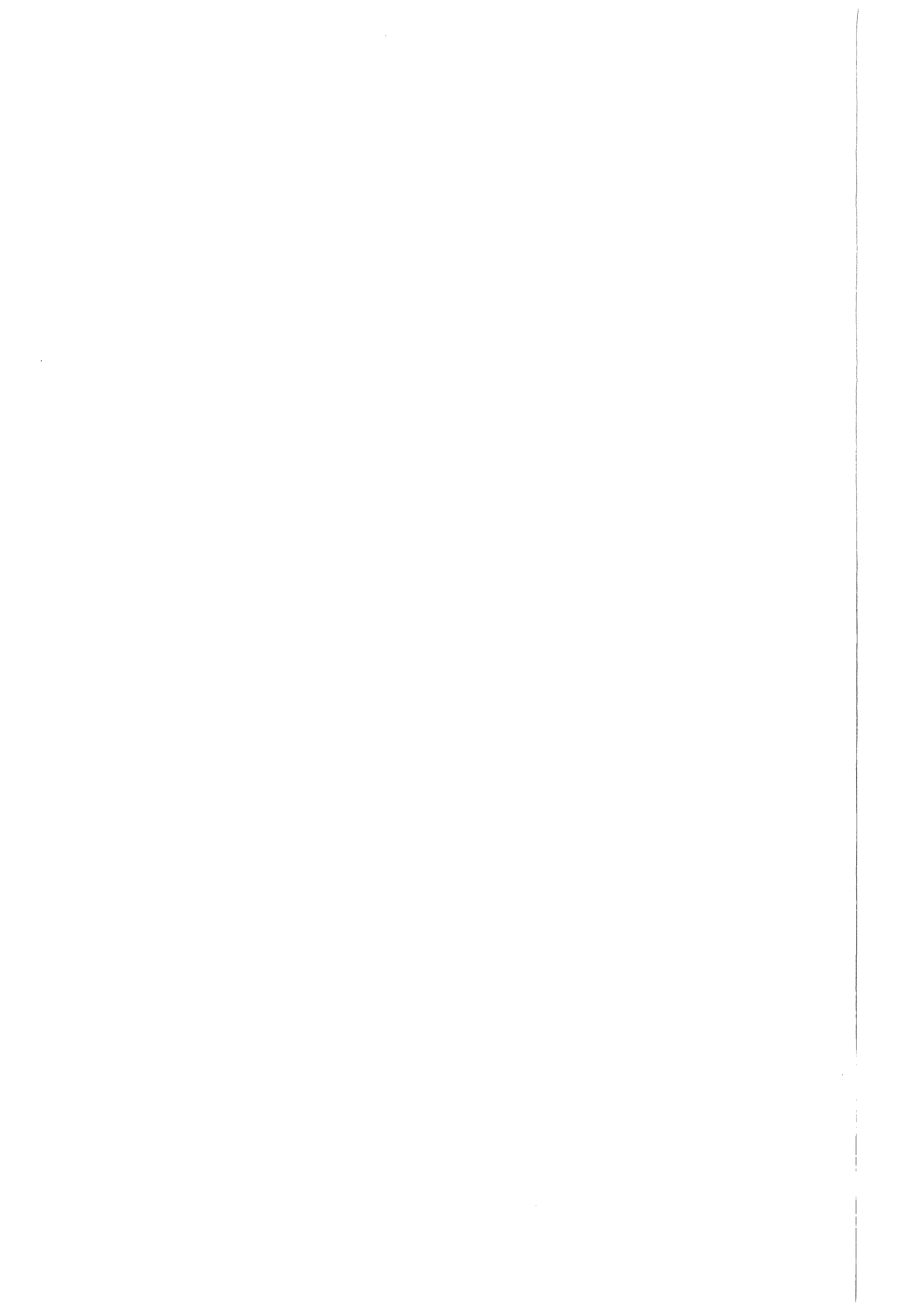
[4] An Efficient Compiler Validation Method for Ada 9X, Michael Tonndorf, Proceedings of the Ada Europe '93 Conference, Springer Verlag LNCS 688, Berlin Heidelberg, 1993.

[5] An Integrated Tool Environment for Ada Compiler Validations, Michael Tonndorf, Proceedings of the TRI-Ada '93 Conference, ACM SIGAda, New York, 1993.

[6] Ten Years of Tool Based Ada Compiler Validations: An Experience Report. Michael Tonndorf, Proceedings of the Ada Europe '98 Conference on Reliable Software Technologies, Springer Verlag LNCS 1411, Berlin Heidelberg, 1998.



Ada, adé ?
- 10 Jahre Erfahrungen mit Ada - ein persönliches Resumée



14 Ada, adé ? - 10 Jahre Erfahrungen mit Ada - ein persönliches Resumée

Dr. Uwe Kühne

Dornier GmbH (VE5 A7)

14.1 Übersicht

- Kenngrößen unserer Ada-Projekte
- Erfolgreich abgeschlossene Ada-Projekte
- Erfolgsfaktoren / Wertschöpfung
- Derzeitige Projekte / Herausforderungen
- Gedanken über die Zukunft von Ada

14.2 Zunächst einmal:

Ich bin ein Ada - Fan !

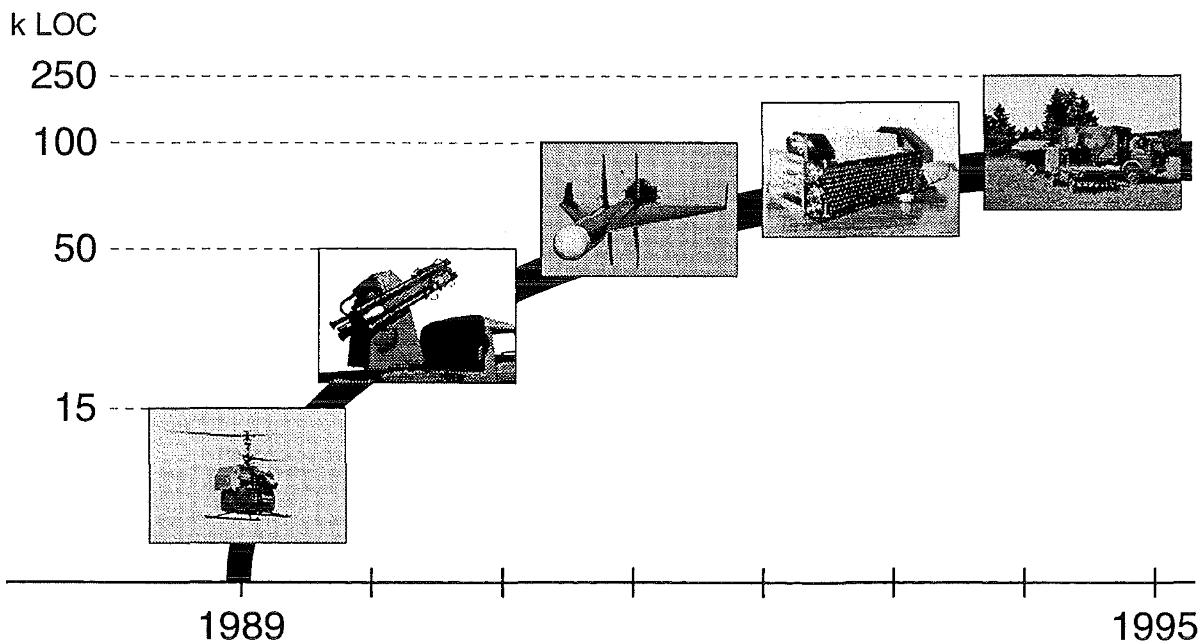
Aber:

Die Qualität einer Software-Entwicklung hängt nicht allein vom Einsatz einer bestimmten Programmiersprache ab !

14.3 Kenngrößen unserer Ada-Projekte

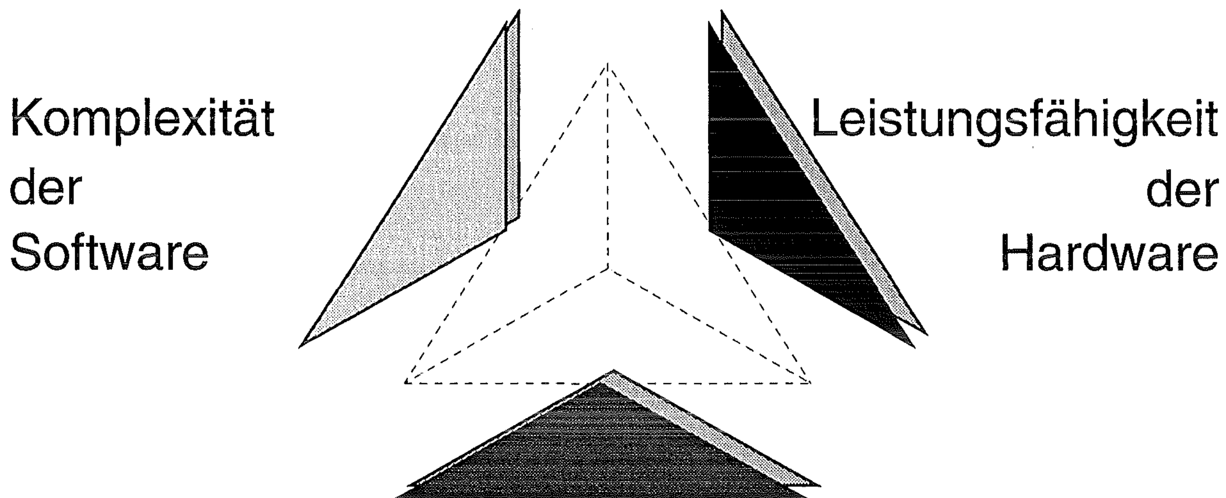
- Regelungs- u. Führungssysteme für Drohnen und Fahrzeuge
- Eingebettete Echtzeit-Systeme (harte Echtzeit-Bedingungen)
- Typisches SW-Entwicklungsteam: 5 .. 10 MA
- Typische Projektlaufzeit: 2 .. 3 Jahre
- Operationelle Software: 1×10^5 .. 5×10^5 LOC

14.4 Erfolgreich abgeschlossene Ada-Projekte



14.5 Erfolgsfaktoren dieser Projekte

- Ausgewogenheit / Verträglichkeit zwischen



- "Stimmigkeit" der Auftraggeber-Anforderungen

14.6 Erfolgsfaktoren (Forts.)

- Einsatz von Ada als Zielsprache
    - ↳ *Ada-Highlights ... Ada-Highlights ... Ada-Highlights*
    - ↳ Konzepte unterstützen Arbeit in großen Teams
    - ↳ Design (fast) ohne Work-arounds umsetzbar
(Vor. für den Einsatz von Code-Buildern)
    - ↳ (SW-) Integration in Rekordzeit
    - ↳ Entwicklung von Konzepten (nicht nur Lösungen)

14.7 Wertschöpfung aus diesen Projekten

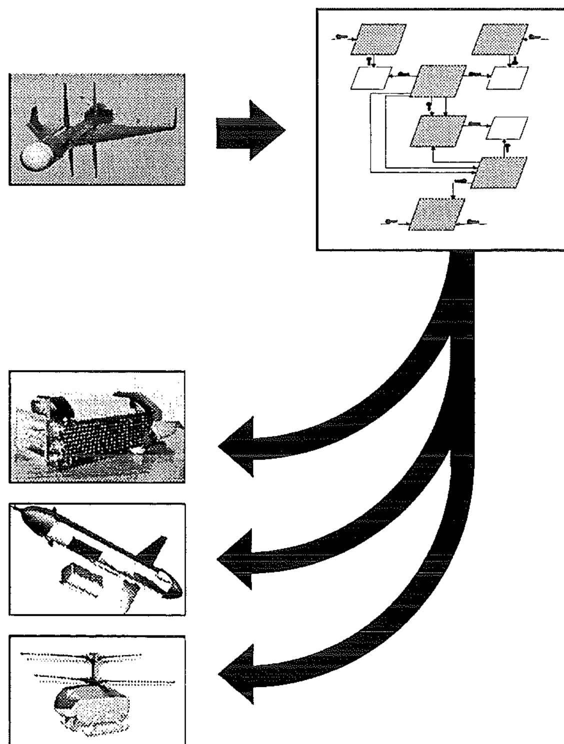
- Ausgehend von Ada als Zielsprache
  - ↳ Einführung von Design-Methoden
  - ↳ Einführung von CASE-Tools
  - ↳ Einführung von Analyse-Methoden

  zu einem "State of the Art" Software Entwicklungs*prozeß*
- Strategie für SW-Engineering (über Projektgrenzen hinaus)

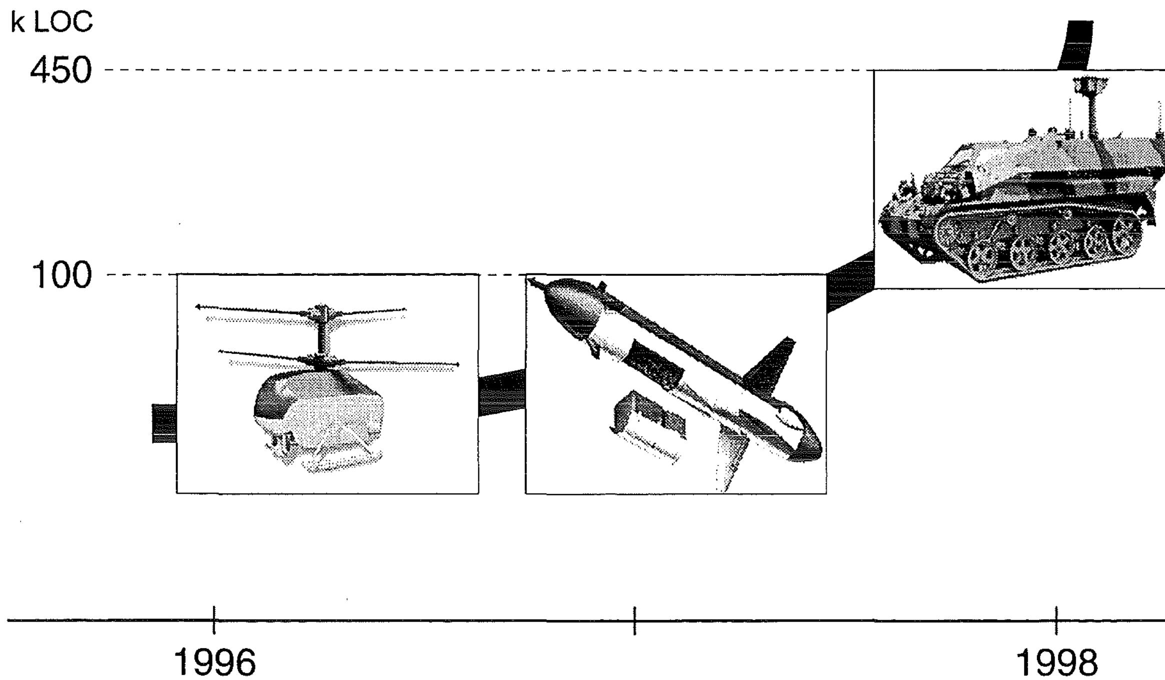
14.8 Wertschöpfung (Forts.)

- Standard-Architektur mit
  - ↳ Task-Komm.-Mechanismen
  - ↳ Fehlerbehandlungskonzepten
  - ↳ Test-Mechanismen

  für
  - ⇨ langsam fliegende Drohnen
  - ⇨ schnell fliegende Drohnen
  - ⇨ Hubschrauber-Drohnen



14.9 Derzeit laufende Ada-Projekte



14.10 Herausforderungen

- Eingebettete Systeme mit mehr als 30 Prozessoren
- Einsatz von Echtzeit-Betriebssystemen
- Einsatz von neuen Zielsysteme (PPC)
- Einsatz von neuen Entwicklungsumgebungen
- Einsatz von Ada '95 für Client-Server (ab April '98)
- SWPÄ über lange Zeiträume

14.11 Skepsis ist angebracht, denn

- Ada hat (nach wie vor) eine geringe Verbreitung
  - ↳ Fokus liegt auf anderen Sprachen
  - ↳ keine COTS-Produkte verfügbar
  - ↳ oftmals "Erstentdecker" von Problemen / Fehlern
- Es fehlt der "Druck des Marktes"
- Die "politische" Unterstützung für Ada schwindet

14.12 Prinzipiell ist zu hinterfragen

- Kann eine Sprache (und damit eine zugehörige Entwicklungsumgebung) alle Aspekte der Sprache (OO und RT) gleich gut unterstützen? ("Problem der Generalisierung")
- Konkret: Sollte Ada auf den Bereich der sicherheitskritischen Echtzeit-Anwendungen fokussieren?
  - ↳ Laufzeit-optimierende Code-Generatoren
  - ↳ minimale / effiziente / zertifizierte Echtzeit-Kerne
  - ↳ COTS-Produkte (speziell für diesen Bereich)

14.13 Hat Ada eine Zukunft?

- *Nicht* aufgrund der Möglichkeiten der Sprache und der Schönheit der Konzepte.
- *Nur* durch den Nachweis, daß Projekte mit Zielsprache Ada erfolgreicher sind (in Bezug auf Termine und Kosten) als andere Projekte.
- Voraussetzung ist:
  - ↳ derzeitige Ada-Anwender zu halten
  - ↳ neue Anwender zu gewinnen

Teilnehmer

15 Teilnehmer

Nachname	Vorname	Ort
Barr	Volkert	Oldenburg
Baumann	Wolfram	Freiburg
Behrens	Thorsten	Bremen
Behrmann	H.-M.	
Brömel	Peter	Creuzburg
Bühler	Gerhard	Wachtberg-Werthhoven
Closhen	Patrick	Darmstadt
Dencker	Peter	Karlsruhe
Ecke	Frank	Sondershausen
Etrich	Matthias	Darmstadt
Fachet	Ralf	Karlsruhe
Fesche	D.	
Fey	Ines	Berlin
Fischer	Dietrich	Taufkirchen
Gärtner-Frank	Marianne	
Geyer	Lothar	Kaiserhammer
Gonser	Peter	Nürnberg
Greger	Ralf	
Gronowski	Matthias	
große Osterhues	Bernhard	München
Hagedorn	Jörg	Offenbach
Harm	Jörg	Rostock
Häussler	Uli	Koblenz
Helfert	Stefan	Heidelberg
Helmke	Alexander	Weinstadt
Hepke	Günter	Karlsruhe
Heppner	Christian	Kassel
Hohm	Dietmar	Immenstaad

Holländer	Udo	Fürstenfeldbruck
Hoyng	Jürgen	
Hülsing	Thomas	Bremen
Jäger	Bernhard	Kassel
Jungclaus	Heiko	Kiel
Keller	Hubert	Karlsruhe
Kraas	Ewald	Bremen
Krauss	Stefan	Stuttgart
Kreiß	Günter	München
Kruse	Franz	Bremen
Kühne	Uwe	Friedrichshafen
Kutzik	Andre	Adendorf
Landwehr	Rudolf	
Le Marquand	Freddy	Fonteany le Fleury
Lohmann	Wolfgang	Rostock
Lorang	Gerald	Darmstadt
Lucas	Kai	Olching / Neu-Esting
Lux	Peter	
Mangold	Karl-Otto	Konstanz
Müller	Rolf	Adendorf
Munch	Poul	Lyngby
Murrach	Winfried	
Nivelnkötter	Thomas	Bremen
Norrenbrock	H.	
Oppenheimer	Frank	Oldenburg
Plödereder	Erhard	Stuttgart
Romanski	Georg	Burlington
Röttges	Thomas	Bad Zwischenahn
Sauermann	Gerd	Oberhaching
Schlitt	Manfred	Obertshausen
Schlüter	Michael	Kiel

Schröer	Joachim	Ranshofen, Austria
Schumacher	Guido	Oldenburg
Schwald	Andreas	
Seelhorst	Christoph	Achim bei Bremen
Siebert	Ingo	Siegburg
Sieper	Walter	Wilhelmshaven
Stäcker	Wilfried	
Strauss	Michael	Vaterstetten/München
Suilmann	Matthias	Meppen
Sukatsch	Carsten	
Tiedemann	Michael	Ulm
Tonndorf	Michael	Ottobrunn
Voges	Udo	Karlsruhe
Wachsmuth	Klaus	Karlsruhe
Weber-Wulff	Debora	Berlin
Wegner	Frank	Obertshausen
Wenz	Oliver	Offenbach