

Forschungszentrum Karlsruhe

Technik und Umwelt

Wissenschaftliche Berichte

FZKA 5981

Erfahrungen mit Ada

**Workshop der Fachgruppe 2.1.5
- Ada Ada-Deutschland –
der Gesellschaft für Informatik**

Hubert B. Keller, Andreas Schwald¹ (Hrsg.)

Institut für Angewandte Informatik

¹München

Forschungszentrum Karlsruhe GmbH, Karlsruhe

1998

Zusammenfassung

Am 17.4.1997 veranstaltete die GI-Fachgruppe „Ada Deutschland“ im Forschungszentrum Karlsruhe den Workshop „Erfahrungen mit Ada“, bei dem Erfahrungen mit der Programmiersprache Ada in der Lehre und in verschiedenen Projekten zur Sprache kamen. Dr. Hubert B. Keller als Gastgeber sowie Dr. Andreas Schwald als Veranstalter konnten über 30 Experten aus Industrie, Forschung und Hochschulen zu dieser Veranstaltung begrüßen. Das Programm umfaßte aktuelle und hochinteressante Vorträge, die verschiedene, sich ergänzende Sichtweisen deutlich machten.

Experiences with the Programming Language Ada
Workshop of the Special Group 2.1.5 Ada
Ada-Deutschland
of the German Society for Computer Science

Abstract

On April, the 17th of 1997, the GI-working group "Ada-Deutschland" held a workshop "Erfahrungen mit Ada" at the Forschungszentrum Karlsruhe dealing with Ada experiences in education and in different projects. More than 30 experts from industry, research and education were welcomed by Dr. Hubert B. Keller as the host and Dr. Andreas Schwald as the organizer. The programm consist of actual and highly interesting talks presenting complementary views of current problems and ongoing work.

Vorwort

Am 17. 4. veranstaltete die GI-Fachgruppe „Ada Deutschland“ im Forschungszentrum Karlsruhe den Workshop „Erfahrungen mit Ada“, bei dem Erfahrungen mit der Programmiersprache Ada in der Lehre und in verschiedenen Projekten zur Sprache kamen. Dr. Hubert B. Keller als Gastgeber sowie Dr. Andreas Schwald als Veranstalter konnten über 30 Experten aus Industrie, Forschung und Hochschulen zu dieser Veranstaltung begrüßen. Das Programm umfaßte aktuelle und hochinteressante Vorträge:

- Prof. Plödereder (Univ. Stuttgart) betonte, daß für Ada das breite Angebot an Konzepten auf hohem Niveau spricht, während für spezifische Anwendungsbereiche Spezialsprachen bessere Unterstützung bieten, und wußte von einer zögerlichen Einstellung zu Ada95 bei manchen Kollegen zu berichten.
- Prof. Tempelmeier (FH Rosenheim) berichtete von der Situation der Programmiersprachenausbildung an seiner Fachhochschule, an der Ada neben C(++) als zweite Sprache für technische Studienrichtungen gelehrt und in der Projektarbeit verwendet wird.
- Prof. Röhrle (FH Sigmaringen-Albstadt) stellte die guten Erfahrungen mit Ada als erster Programmiersprache dar (mit Vertiefung im zweiten Studienabschnitt nach dem Praxissemester). Er wertete die Ada-Ausbildung als Qualitätskriterium für seine FH
- Herr Beck (Bosch Telecom) berichtete über langjährige praktische Projekterfahrung mit Ada im Bereich der Telekommunikation und präsentierte die für diese Anwendungen entwickelte Methodik, die vom V-Modell als organisatorischem Rahmen ausgeht, die Spezifikationssprache SDL zur semiformalen Entwurfsbeschreibung und Ada als Programmiersprache verwendet. Ein eigenentwickeltes Werkzeug („SDL/Ada-Framework“) unterstützt die Umsetzung eines SDL-Entwurfs in Ada sowie die Generierung von Programmtests.
- Herr Keller (Forschungszentrum Karlsruhe) berichtete von der Anwendung von Ada in umwelttechnischen Projekten. Ada hat sich ab ca. 84/85 aus Software-technischen Gründen hervorragend für komplexe Projekte geeignet, die Produktivität ist sehr gut bei gleichzeitig geringer Fehlerrate. Die Wiederverwendbarkeit von Bausteinen aus 1985 ist auch in 1995 gegeben. Allerdings ist die Einbettung von Ada in Umgebungen (Graphik usw.) nicht immer optimal gelöst.
- Herr Thieme (Bundeswehr) stellte die Forderungen an Ada aus Nutzersicht dar, ausgehend von den Erfahrungen mit Ada83 in den DV-Vorhaben der Bundeswehr. Als technische Hauptforderungen nannte er die Bereitstellung von

Produktionscompilern und Entwicklungsumgebungen für Ada95 mit Datenbank- und Grafikanbindungen sowie Fremdsystemschnittstellen, welche auch die Einbindung von kommerziellen Pogrammbibliotheken und -produkten unterstützen.

- Herr Lucas ergänzte diese Ausführungen durch seine Erfahrungen beim „Verkaufen von Ada“, auch in Hinblick auf die Motive und Erwartungen von Entscheidern.
- Herr Lüß stellte einige Aspekte von PL/SQL (prozedurale Erweiterung von SQL) dar, die in vielen Eigenschaften stark an Ada orientiert ist. In der Diskussion zu diesem Vortrag wurden mehrere Produkte erwähnt, welche die Ada-SQL-Einbindung leisten.
- Herr Landwehr (CCI) berichtete von einem „Null-Fehler-Projekt“, einer größeren Programmerweiterung für die Bundeswehr (KWS Fuchs), für die bei der Abnahme und beim Probetrieb (Truppenversuch) zwar eine größere Anzahl von Änderungswünschen formuliert, aber kein Fehler gefunden wurde.
- Herr Ginkel (Eurocopter) skizzierte die Infrastruktur eines Großprojekts zur Entwicklung von Avionik-Software, welches auch häufige entwicklungsbedingte Änderungen realisieren muß, also nicht nur an die Spezifikations- und Programmiermethoden, sondern auch an das Konfigurationsmanagement und die Qualitätssicherung hohe Anforderungen stellt.

„Insgesamt ... gute Erfahrungen mit Ada, das sich hervorragend eignet, Softwareprojekte in dieser Größenordnung zu realisieren und die Wartbarkeit verbessert.“

- Herr Kellogg (Dornier) erläuterte die Anpassungsmaßnahmen (insbesondere für Datenrepräsentationen) in Ada-Programmen, die in Satellitensystemen auf Bord- und Bodenrechnern laufen und Daten austauschen.

In der abschließenden Diskussion zum Thema „Perspektiven für Ada95“, bei der die meisten der 32 Teilnehmer bis abends um halb sieben aushielten, wurden mehrfach die guten Erfahrungen mit den objektorientierten Ausdrucksmitteln von Ada95 (insbesondere auch mit den Hierarchien von Bibliothekseinheiten) berichtet. Andererseits wurden von Seiten der Realzeitanwender Bedenken geäußert, ob diese Möglichkeiten in zeit- und sicherheitskritischen Teilen (in denen auch bei Ada83 nur ein sehr enger und genau kontrollierter Sprachumfang verwendet wird) zweckmäßig seien.

Der vorliegende Bericht des Forschungszentrums Karlsruhe faßt die Vortragsausarbeitungen in schriftlicher Form zusammen.

Andreas Schwald, Hubert B. Keller
FG 2.5.1 Ada

Inhaltsverzeichnis

1	Ada an der Universität Stuttgart	13
1.1	Die Rolle von Ada in der Ausbildung der Dipl.-Informatiker an der Uni Stuttgart	13
1.2	Studentische Akzeptanz von Ada.....	13
1.3	Die Akzeptanz seitens der Fakultät	14
1.4	Die Akzeptanz durch andere Fakultäten.....	14
2	Ada in der Lehre an der Fachhochschule Rosenheim.....	15
2.1	Problematik der programmiersprachlichen Ausbildung an Fachhochschulen	15
2.2	Das Konzept der programmiersprachlichen Ausbildung an der FH Rosenheim.....	16
2.3	Software-Engineering-Projektarbeit an der FH Rosenheim.....	18
2.4	Literatur	18
3	Einsatz von Ada als erste Programmiersprache im Studiengang Technische Informatik der Fachhochschule Albstadt-Sigmaringen	21
3.1	Motivation	21
3.2	Einbettung in das Curriculum des Studienganges Technische Informatik.....	22
3.3	Resümee	23
4	Software - Entwicklung mit SDL und Ada.....	27
4.1	Einleitung.....	27
4.2	Vorstellung der Methodik SDL/Ada.....	31
4.3	Anmerkungen zur Methodik SDL/Ada	37
4.4	Schlußbemerkungen	40
4.5	Literatur	42
5	Ada-Sprachkonzepte für Datenbankanwendungen.....	43

5.1	Einleitung/Motivation: Von der Schnittstellenspezifikation zum Datenmodell.....	43
5.2	Übersicht über PL/SQL.....	44
5.3	Hauptmerkmale von PL/SQL.....	44
5.4	Eigenschaften von PL/SQL.....	45
5.5	Vorteile von PL/SQL.....	49
5.6	Brückenschlag zu Ada.....	49
6	Ada in der Umwelttechnik.....	51
6.1	Einleitung.....	51
6.2	Bisherige Ada-Projekte.....	52
6.3	Zusammenfassung.....	57
7	Was erwartet der Nutzer von Ada und welche Anforderungen hat er?.....	61
8	Wie verkaufe ich Ada ?.....	67
8.1	Wissensbasis ENTSCHIEDER Voreinstellung.....	67
8.2	Wissensbasis PROJEKTLINIENLEITER Voreinstellung.....	68
8.3	Wissensbasis NOVICE Voreinstellung.....	68
8.4	Motivation/Erwartung beim Entscheider.....	69
8.5	Motivation/Erwartung beim Projektlinienleiter.....	69
8.6	Motivation/Erwartung beim Novicen.....	70
8.7	NOVICE.....	70
8.8	PROJEKTLINIENLEITER.....	71
8.9	ENTSCHIEDER.....	71
8.10	Durchsatz.....	72
9	Softwarepflege mit Ada – das Projekt KWS-Fuchs.....	75
10	Erfahrungen mit ADA im Projekt TIGER.....	89
10.1	Avionik SW-Entwicklung bei EUROCOPTER.....	90
10.2	Erfahrungen mit ADA.....	94
10.3	Zusammenfassung der Erfahrungen mit ADA im Projekt TIGER.....	98
10.4	Literatur.....	100
11	Anwendung von Ada in Satellitensystemen.....	101

11.1	Einleitung.....	101
11.2	Die Systeme	101
11.3	Erfahrungen bei der Ada-Programmierung -- "Lessons Learned"	103
11.4	Zusammenfassung und Ausblick.....	117
11.5	Literatur	119
12	Teilnehmer	123
13	Ada Deutschland - GI Fachgruppe 2.1.5 "Ada"	129

I Ada in der Lehre

1 Ada an der Universität Stuttgart

Prof. Dr. Erhard Plödereder

Universität Stuttgart

email: ploedere@informatik.uni-stuttgart.de

1.1 **Die Rolle von Ada in der Ausbildung der Dipl.-Informatiker an der Uni Stuttgart**

Ada im Grundstudium:

- erste Sprache ist nicht festgeschrieben
- in der Vergangenheit: Modula-2, Modula-3, Scheme
- Ada ist als Alternative akzeptabel

Ada im Hauptstudium:

- als primäres Sprachbeispiel, um programmiersprachliche Konzepte zu erläutern
- Kompaktkurse
- als Implementierungssprache in einigen Praktikas, Studien- und Diplomarbeiten

1.2 **Studentische Akzeptanz von Ada**

... rational von den Qualitäten der Sprache überzeugt

- Sicherheit (z.B. frühe Fehlererkennung, Schnittstellenprüfung)
- Vielseitigkeit ("für jedes Grobkonzept etwas da")
- [Wartbarkeit]

... pragmatisch/emotionell "doch lieber C++"

1.3 Die Akzeptanz seitens der Fakultät

... durchaus gegeben, aber....

- mangelnde Detailkenntnisse verhindern breiteren Einsatz
- für spezifische Zwecke sind methodisch engere Sprachen besser geeignet

die "Konkurrenz":

- Modula-2 (Lehrmaterial vorhanden)
- Scheme, Eiffel, Smalltalk (jeweilige Methodik; Sprachumgebung)
- Java (Aktualität; Methodik; Sprachumgebung)

N.B.: C und C++ werden einheitlich abgelehnt

1.4 Die Akzeptanz durch andere Fakultäten

... ist eher gering, allerdings mit nennenswerten Ausnahmen:

- Kompaktkurse für die Luftfahrt
- Vorlesung und Seminar zur Prozessautomatisierung (Elektrotechnik)

verbreitet ist der Wunsch nach Fortran und C/C++

2 Ada in der Lehre an der Fachhochschule Rosenheim

Theodor Tempelmeier¹

Fachhochschule Rosenheim

In diesem Beitrag wird die Stellung der Programmiersprache Ada in der Informatikausbildung an Fachhochschulen, insbesondere an der Fachhochschule Rosenheim, beleuchtet.

2.1 *Problematik der programmiersprachlichen Ausbildung an Fachhochschulen*

Jede Ausbildung im Programmieren ist mit der Auswahl einer Programmiersprache verbunden. Die Entscheidung für oder gegen eine bestimmte Sprache ist seit jeher Gegenstand von Kontroversen. Jeder Dozent bevorzugt aus einer Vielzahl von Gründen eine oder mehrere bestimmte Sprachen: Der eine legt Wert auf klare, durchdachte Konzepte, der andere bevorzugt Sprachen, die nur geringen Tippaufwand erfordern, wieder ein anderer mag am Stellen und Lösen von Rätseln sein Freude haben [1].²

Der Autor des vorliegenden Beitrags hat eine Präferenz für die Sprache Ada, nicht zuletzt wegen persönlicher Erfahrungen im Bereich sicherheitskritischer Software, also von Software, von deren Funktionieren direkt Menschenleben abhängen können.

Bei der Auswahl einer Sprache stehen die didaktischen Forderungen der Lehre (Vermittlung klarer, langfristig tragfähiger Konzepte) im Wettstreit mit den Forderungen der Industrie³ („Praxisbezug“). Dies erweist sich bei der Sprachauswahl als besonders problematisch, da die sauber durchdachten Sprachen Ada, Eiffel, Modula, Oberon, etc. in der Industriepraxis kaum eingesetzt werden. Umgekehrt werden in der Industrie Sprachen wie C, C++, COBOL, PL/I bevorzugt, die in vielerlei Hinsicht für die Lehre (und auch für die industrielle Anwendung!) ungeeignet erscheinen. Als Kritiker der Sprachen C bzw. C++ seien hier nur [3-5] aufgeführt, denen sich der Autor vorbehaltlos

¹ Prof. Dr. Theodor Tempelmeier, Fachbereich Informatik, Fachhochschule, Marienberger Str. 26, 83024 Rosenheim, Telefon 08031-805-510, Fax -105, E-Mail tt@extern.lrz-muenchen.de, World Wide Web <http://www.fh-rosenheim.de>.

² Die treffende Formulierung wurde von L. Frevert [2] übernommen.

³ Der Begriff „Industrie“ stehe hier allgemein für alle Organisationen, die Informatikabsolventen aufnehmen, also auch für Behörden, Körperschaften, Rechenzentren, Banken, Versicherungen, etc.

anschließt. Dem steht beispielsweise die platte Forderung aus der Industrie gegenüber, „Für Informatiker sollte C++ zum Ausbildungsstandard gehören.“ [6]. Es bleibt abzuwarten, inwieweit die für die Zukunft geplanten bzw. bereits bestehenden Hochschulbeiräte aus Industrievertretern hier sinnvollere Beiträge liefern können.

Für die Fachhochschulen mit ihrer starken Betonung des Praxisbezugs spitzt sich die Problematik „C, C++ vs. Ada, Eiffel, Oberon, Modula“ weiter zu:

- „Dürfen“ Fachhochschulen überhaupt aus didaktischen Gründen etwas lehren, was in der Praxis nur in relativ geringem Maße eingesetzt wird?
- Soll man unter dem Deckmantel des ach so geheiligten Praxisbezugs den Rückfall auf den „Stand der 60er Jahre“ [3] mitvollziehen?
- Welche Sprache und welche Konzepte sollen die Studenten in ihrem ersten Studienjahr lernen, damit Sie sich im anschließenden Praxissemester in einem Industriebetrieb bewähren können?

Jede einzelne Fachhochschule, jeder Fachbereich muß sich diesen Fragen stellen. Dabei haben sich manche für fortschrittliche Sprachen wie Ada entschieden, andere wählen nach pragmatischen Gesichtspunkten aus. Wünschenswert wäre nach Meinung des Autors eine bundesweite, allgemeine Diskussion dieser Fragen - bezogen auf die spezifische Situation der Fachhochschulen, eventuell im Rahmen des jährlichen Fachbereichstags Informatik.

2.2 Das Konzept der programmiersprachlichen Ausbildung an der FH Rosenheim

An der Fachhochschule Rosenheim wurde im Zuge des Aufbaus des Studiengangs Informatik ein Konzept für den Programmierspracheneinsatz erstellt, welches auf Pascal als erster Lehrsprache beruhte [7]. Etwa zum zehnten Jahrestag der Informatik in Rosenheim [8] wurde das Konzept pragmatisch an die Industriesituation angepaßt und Pascal durch „C+“ als erste Programmiersprache ersetzt [9]. Gleichzeitig wurde aber das gründliche Erlernen einer zweiten Sprache de facto zur Pflicht gemacht, so daß sich insgesamt folgende Situation ergibt.

- Grundausbildung im Programmieren in „C+“

Mit „C+“ wird dabei eine Teilmenge von C++ bezeichnet, die sich etwa durch die Schlagworte „C++ als besseres C“ und „modulares Paccsal in C-Syntax“ charakterisieren läßt. Durch die Modularisierung können einerseits die grundlegenden Ideen der Objektorientierung (Kapselung, abstrakte Datentypen) vermittelt werden, andererseits sind sich die Lehrenden durchaus der Schwächen von C und C++ im Bereich der Modularisierung bewußt. Objektorientierung im eigentlichen Sinne (Vererbung, Polymorphie) wird erst nach der der Grundausbildung in einer eigenen Lehrveranstaltung „Objektorientierte Programmierung“ angeboten.

- Ausbildung in einer zweiten Programmiersprache, z.B. Ada

Ab dem vierten Semester beginnt für die Studenten die Ausbildung in einer zweiten Programmiersprache. Selbstverständlich geht es dabei nicht um das bloße Erlernen einer neuen Sprache. Vielmehr werden weitere wichtige Programmier Techniken wie Generizität, Überladen, Typerweiterung, Ausnahmebehandlung, Ansätze der Parallelprogrammierung, etc. hinzuerlernt. Außerdem wird - ebenso wichtig - ständig ein Vergleich zu anderen Programmiersprachen gezogen, und die Gründe und Konsequenzen der Sprachunterschiede werden diskutiert.



Abb. 1: Integration des GNU Ada Übersetzers in WinEdit und Einfügen von Standard-Codesequenzen [10]

Diese Veranstaltung wird in zwei Ausprägungen abgehalten, wobei die Ausprägung für die Studienrichtung „Technik“ mit dem Schwerpunkt Ada vom Autor dieses Beitrag durchgeführt wird. Aufgrund ihrer Sauberkeit und Vollständigkeit erweist sich dabei Ada als ideale Lehrsprache. Weitere Gründe für den Ada-Einsatz sind die Vorbereitung auf die Vorlesung Echtzeitsysteme sowie die Ähnlichkeit von Ada zu FORTRAN 90 und zur Hardwarebeschreibungssprache VHDL. In die vergleichende Sprachbetrachtung werden u.a. C, C++, COBOL, FORTRAN, und Pascal einbezogen. Praktische Übungen werden jedoch ausschließlich in Ada durchgeführt, um die Verwirrung bei den Studierenden in Grenzen zu halten. Für die Übungen wird mit sehr guten Ergebnissen GNU Ada mit einer selbst konfigurierten Oberfläche für den kommerziellen Editor WinEdit unter Windows 95/NT eingesetzt ([10] und Abb. 1).

2.3 Software-Engineering-Projektarbeit an der FH Rosenheim

Über das übliche Angebot an Lehrveranstaltungen zum Thema Software-Engineering hinaus wird an der Fachhochschule Rosenheim für das Abschlußsemester der Studienrichtung Technik ein großes Projekt(spiel) im Umfang von sechs Semesterwochenstunden durchgeführt. In der Vergangenheit wurde dabei mehrmals beispielhaft die Steuerung eines fahrerlosen Transportsystems (FTS) als Thema gewählt. Der technische Prozeß, d.h. die im Fabrikgelände umher fahrenden Fahrzeuge, etc, ist dabei für Testzwecke zu simulieren. Das Projektspiel beginnt üblicherweise mit dem Systementwurf, also der Festlegung der Struktur des Gesamtsystems einschließlich aller Rechner und deren Organisation. Beispielsweise muß hier die Weichenstellung in Richtung „zentrale oder dezentrale Organisation der Rechner und Fahrzeuge“ erfolgen. Die eigentliche Realisierung der Software erfolgt dann in Ada, wobei das gesamte Sprachspektrum einschließlich „Tasking“ und „Protected Objects“ zum Einsatz kommt. Der Schwerpunkt liegt dabei auf der Software-Entwurfsphase. Der Software-Entwurf erfolgt direkt in Ada (Programmieren im Großen). Für die Entwurfsdiskussion und -dokumentation haben sich dabei grafische Darstellungen sehr bewährt, die jedoch isomorph⁴ zum Programmcode sein müssen. Es werden im wesentlichen modifizierte Buhr-Diagramme verwendet. Abstraktere und weniger ausdrucksstarke Darstellungen, z.B. die Unified Modeling Language [11] und ihre Vorläufer, können nicht unmittelbar die Einheiten des Programmcodes widerspiegeln. Sie haben sich für die Entwurfsphase als weniger hilfreich erwiesen. Üblicherweise diskutieren die Studierenden die Entwürfe anhand von Anwendungsfällen („Use Cases“), ohne daß es dazu einer Einführung dieses Begriffs bedarf. Als systematische Vorgehensweise für das Auffinden des Entwurfs wird in die Methode von Nielsen und Shumate (z.B. [12]) verwendet.

2.4 Literatur

- [1] Feuer, A.R.: Das C-Puzzlebuch. C Programmier-Training. Carl Hanser Verlag, München Wien 1985.

⁴ Der Begriff isomorph wird hier nicht im strengen mathematischen Sinn, sondern allgemein in der Bedeutung „gleichgestaltig“ verwendet.

- [2] Frevert, L.: PEARL 90 in der Lehre - Erfahrungsbericht. In: Peter Holleczeck (Hrsg.), PEARL 96, Workshop über Realzeitsysteme, Fachtagung der GI-Fachgruppe 4.4.2 Echtzeitprogrammierung, PEARL, Boppard, 28./29. November 1996. Springer Verlag, Berlin, Heidelberg, New York 1996.
- [3] Wirth, N.: Gedanken zur Software-Explosion. Informatik-Spektrum 1994, 17, 5-10.
- [4] Ludewig, J.: Sprachen für das Software-Engineering. Informatik-Spektrum, 1993 16, 286-294.
Leserbriefe und Erwiderung zu diesem Artikel in Informatik-Spektrum 1994, 17, 59-60, 186-187.
- [5] Hug, K.: Diskussionsbeitrag zum Artikel „C++ im Nebenfachstudium: Konzepte und Erfahrungen“. Informatik-Spektrum 1996, 19, 338-341.
- [6] Letters, F.: Wirklichkeitsferne Hochschulausbildung. Computer Zeitung Nr. 18, 2.Mai 1997, S. 6.
- [7] Frank, L., Siedersleben, J.: Programmiersprachen im Informatik-Studium. PIK (Praxis der Informationsverarbeitung und Kommunikation), 1992, 15, 106-111.
- [8] 10 Jahre Informatik in Rosenheim. Sondernummer der Rosenheimer Hochschulhefte, Mai 1996.
- [9] Studienführer 1997. Fachhochschule Rosenheim, Mai 1997.
- [10] Integration des GNU Ada Übersetzers in WinEdit und Erzeugung von Standard-Codesequenzen. Information und Bezugsmöglichkeit über [ftp.fh-rosenheim.de](ftp://ftp.fh-rosenheim.de), Verzeichnis /pub/languages/ada/win95.
- [11] Unified Modeling Language 1.0. Jan 1997. Information und Bezugsmöglichkeit: <http://www.rational.com>.
- [12] Nielsen, K., Shumate, K.: Designing Large Real-Time Systems with Ada. Intertext Publications & McGraw-Hill, New York 1988. Siehe auch spätere Veröffentlichungen dieser Autoren.

Die in diesem Beitrag verwendeten Warenzeichen sind nicht jeweils gesondert gekennzeichnet. Bei einer Verwendung dieser Warenzeichen sind die entsprechenden Vorschriften und Einschränkungen zu beachten.

3 Einsatz von Ada als erste Programmiersprache im Studiengang Technische Informatik der Fachhochschule Albstadt-Sigmaringen

Prof. Dr. Jörg Röhrle
Technische Informatik
Fachhochschule Albstadt-Sigmaringen

3.1 Motivation

Der Wunsch zum Einsatz von Ada als erste Programmiersprache im Studiengang *Technische Informatik* entstand mit der curricularen Zusammenfassung der beiden Erst- bzw. Zweitsemesterveranstaltungen "Programmiersprachen I und II" zur Jahresveranstaltung "Programmentwicklung". Mit diesem Schritt sollte der bis dahin auf der Kodierung liegende Schwerpunkt ausgedehnt werden auf alle Aspekte des Software-Entwicklungsprozesses. Zur Realisierung dieses Konzeptes sollte eine Programmiersprache zugrundegelegt werden, die dem derzeitigen Entwicklungsstand des Software-Engineerings Rechnung trägt. Dabei fiel die Entscheidung zugunsten der Programmiersprache Ada aufgrund folgender Überlegungen:

3.1.1 Einsatz eines validierten Sprachkonzeptes

Die Verwendung einer validierten Programmiersprache ist hervorragend geeignet zur Darlegung der Qualitätsprobleme bei der Entwicklung systemunabhängiger Softwaresysteme.

3.1.2 Berücksichtigung von Qualitätskriterien

Gerade bei Anfängern fördert beispielsweise die Strenge des in Ada realisierten *Typkonzeptes* und die damit verbundene Möglichkeit zur *Attributierung* das Bewußtsein für die Einhaltung von Qualitätskriterien und bewirkt damit gleichsam die Entwicklung einer gewissen *Programmierkultur*. Dies macht sich vor allem dann bemerkbar, wenn im unmittelbaren Vergleich zu anderen Programmiersprachen eine Verhaltensspezifikation des Laufzeitsystems schmerzlich vermißt wird.

3.1.3 Realisierung von Sicherheitsaspekten

Da Ada ursprünglich für die Entwicklung großer, auf verschiedenen Systemumgebungen verteilte Software konzipiert wurde, lassen sich verstärkt auch Sicherheitsaspekte behandeln, beispielsweise anhand des *Kopiermodells bei der Parameterübergabe* von Unterprogrammen sowie der Möglichkeit zur *Ausnahmebehandlung*.

3.1.4 Behandlung nebenläufiger Prozesse

Neben der Programmierung eines sequentiellen Ablaufs bietet Ada eine bereits *im Sprachkonzept formulierte Möglichkeit* zur Implementierung nebenläufiger Prozesse und eignet sich damit sehr gut zur Darlegung des Client-Server-Prinzips, auf das in der Veranstaltung *Betriebssysteme* dann weiter eingegangen wird.

3.1.5 Berücksichtigung objektorientierter Entwicklungskonzepte

Spätestens mit der Einführung des Ada-95-Standards wurde das objektorientierte Programmierparadigma hinreichend abgebildet. Die Möglichkeit zur Definition *abstrakter Klassen in Pakethierarchien* macht auch die in der Praxis eher als kritisch erachteten Konzepte, namentlich das der Mehrfachvererbung, im Vergleich zu anderen Programmiersprachen weitaus besser beherrschbar.

3.1.6 Schnittstelle zu anderen Programmiersprachen

Die Programmentwicklung im Umfeld der Technischen Informatik erfordert bisweilen die Implementierung systemnaher Software, beispielsweise im Bereich der *eingebetteten Systeme* oder der *Robotik*. Diese mittlerweile als klassisch geltenden Anwendungsgebiete, die sich vor allem durch die Programmiersprache C eröffnen, werden durch Aspekte aus dem Bereich des *Netzwerk-Computing* auf Basis der Programmiersprache *Java* ergänzt.

3.1.7 Einsatz von Ada zur Implementierung wiederverwendbarer Software-Bausteine

Mit der Betonung der frühen Phasen des Software-Lifecycles, namentlich der Analyse- und Designphase, verlagert sich die Problematik auf das "Programmieren im Großen", also auf die Fragestellung, welche Anwendungsobjekte existieren und auf welche Weise sie beschrieben werden können. Die aus der theoretischen Informatik bekannten formalen Sprachnotationen haben den für die praxisorientierte Lehre gravierenden Nachteil der fehlenden Übertragbarkeit auf reale Ausführungsmodelle. Als Alternative zu algebraischen Notationen kann die Syntax des Paketkonzeptes von Ada herangezogen werden, das sich aufgrund seiner Ausdrucksmöglichkeiten besonders zur Spezifikation von Anwendungsobjekten eignet. Im Mittelpunkt steht hierbei die Möglichkeit zur Definition *Abstrakter Datentypen*, die sich durch das Konzept der generischen Pakete so formulieren lassen, daß einerseits die gesamte Zugriffslogik gegenüber dem Benutzer verborgen bleibt, während andererseits durch Privatisierung eine klare Trennung der von außen sichtbaren Datenstrukturen möglich ist.

3.2 Einbettung in das Curriculum des Studienganges Technische Informatik

Die in der Anfängerveranstaltung "Programmentwicklung" gelegten Grundkenntnisse werden in den nachfolgenden Semestern vertieft und ausgebaut:

- Die auf der Basis der Programmiersprache C++ basierende Veranstaltung "**Softwarekonstruktion**" im vierten, bzw. im siebten und achten Semester profitiert hierbei von dem von Anfang an praktizierten "Programmieren im Grossen", vor allem im Umgang mit der Definition Abstrakter Datentypen und dem damit einhergehenden objektorientierten Programmierparadigma.

- Durch Bezug auf die sehr stark an Ada angelehnte Sprache PL/SQL des Datenbanksystems ORACLE können im Rahmen der Fünftsemesterveranstaltung "**Datenbanksysteme**" Programme implementiert werden, die auf bereits bekannten Konzepten basieren.
- Ein Schwerpunkt der **Betriebssystemveranstaltung** bezieht sich auf die Implementierung von Client-Server-Anwendungen, welches in der Veranstaltung *Programmentwicklung* vorgestellte Task-Konzept zumindest ansatzweise vorgestellt wurde.

3.3 *Resümee*

Die Jahresveranstaltung *Programmentwicklung* wurde im Studiengang *Technische Informatik* der Fachhochschule Albstadt-Sigmaringen zum Wintersemester 1994/95 eingeführt. Die Erfahrungen, die seither in bezug auf Ada als erster Programmiersprache im Rahmen einer auf den Prinzipien des Software-Engineering beruhenden Veranstaltung gemacht wurden, sind überwiegend positiv! Im Gegensatz zu der ansonsten praktizierten Vorgehensweise, Ada erst in höheren Semestern und damit meist nur im Rahmen von Zusatzveranstaltung anzubieten, ermöglicht eine konzentrierte Behandlung von Sprachkonzepten, die ansonsten über mehrere Veranstaltungen und Programmiersprachen verteilt gelehrt werden. Gerade bei Anfängern läßt sich beobachten, daß die durch Ada aufgenommene Programmierdisziplin beim Übergang zu anderen Programmiersprachen selbstverständlich übertragen wird. Darüber hinaus werden die in Ada gelehrt Konzepte dort meist schmerzlich vermißt und als mehr oder weniger große Einschränkung empfunden. Die Erfahrung des zurückliegenden Zeitraums lehrt, daß diese Art des Übergangs wesentlich leichter fällt als die ansonsten übliche Vorgehensweise des langsamen "Vortastens" durch die Lehre typischer Anfängersprachen.

Als besonders motivierend wird weiter auch empfunden, daß die am Anfang des Studiums dargelegten Konzepte in nachfolgenden Vertiefungsveranstaltungen, wie Softwarekonstruktion, Betriebssysteme oder Datenbanksysteme wiederholt angewandt und somit verinnerlicht werden. Hierbei konnte eine erhebliche Qualitätsverbesserung der von Studentenseite implementierten Programme festgestellt werden. Mit der zunehmenden Verbreitung des Ada-Sprachkonzeptes bzw. von direkt daraus abgeleiteten Elementen, wird dann auch dem öfter vorgebrachten Pragmatismus Rechnung getragen, der angeblich durch die Lehre industriell stärker eingesetzter Programmiersprachen zum Ausdruck kommen sollte.

II Projekte 1

4 Software - Entwicklung mit SDL und Ada

Gerhard Beck
Bosch Telecom GmbH, Backnang
gerhard_beck@bk.bosch.de

Zusammenfassung

Die Integration ‚standardisierter Sprachen‘ für Projektorganisation, SW-Entwurf und Implementierung wird als praxiserprobter methodischer Ansatz zur SW-Entwicklung beschrieben. Der gezielte Einsatz phasenspezifischer Ausdrucksmittel, die Überwindung der Sprachgrenzen und zukünftige Entwicklungspotentiale werden aufgezeigt.

Stichworte: Methodik SW-Entwicklung, V-Modell, SDL, Ada

4.1 Einleitung

4.1.1 Motivation

Der vorliegende Artikel entstand anlässlich eines Workshops ‚Erfahrungen mit Ada‘ der GI-Fachgruppe 2.1.5 Ada am 17. April 97 im Forschungszentrum Karlsruhe -Technik und Umwelt.

Der Artikel berichtet über langjährige praktische Projekterfahrungen mit der Programmiersprache Ada bei Bosch Telecom. Insbesondere wird über eine im Telekommunikationsumfeld entwickelte Methodik berichtet, die durch Verknüpfung der Spezifikationssprache SDL (Specification and Description Language) und der Programmiersprache Ada geprägt ist. Das V-Modell liefert hierbei den organisatorischen Rahmen und das Basis-Vokabular zur Projektabwicklung.

4.1.2 Technisches Umfeld

Folgende Kenngrößen charakterisieren das technische Umfeld:

- Applikation: Server für Key Distribution (online)
- Embedded Computer System (ECS)
- Entwicklungsumgebung:
 - SPARC/Solaris (native)
 - VMEbus, 68xxx, pSOS tm (cross)
- ca. 150.000 SLOC

ist. Signalverbindungen und Kanäle transportieren Signale (Signal s_i) einschließlich optionaler Signalparameter. Die zum Signal-Transport erforderliche Zeit ist unbestimmt, die Signalreihenfolge bleibt erhalten. Bild 2 zeigt beispielhaft Elemente eines Prozeßdiagramms zur Beschreibung des *Systemverhaltens*.

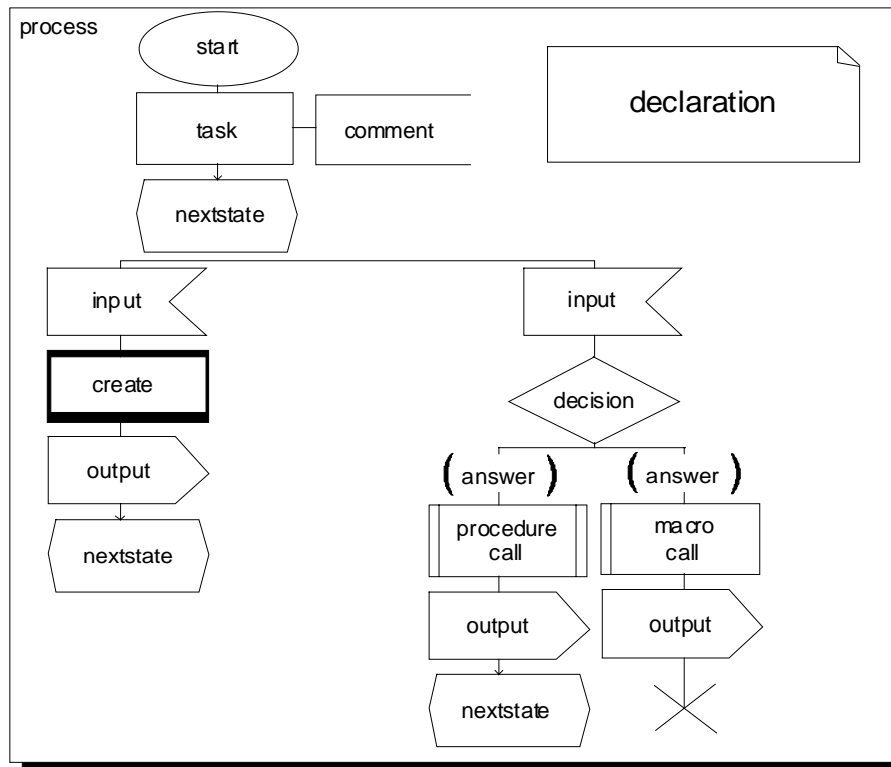


Bild 2: SDL Prozeßbeschreibung (behaviour)

Das Datentypkonzept von SDL basiert auf abstrakten Datentypen (abstract data type, ADT). Abstrakte Datentypen beschreiben das funktionale Verhalten von Objekten durch Spezifikation von Datentypen (sorts) und Operatoren (operators). Durch zusätzliche Angabe von Gleichungen (equations) können die Eigenschaften von Operatoren spezifiziert werden.

```

newtype Bool

literals true, false;

operators

not: Bool -> Bool;

axioms

not (true) == false;

not (false) == true;

endnewtype Bool;
    
```

Bild 3: Beispiel einer einfachen ADT-Deklaration

Ada83 ⁷, Ada95 ⁸

Standardisierte (ANSI, ISO) Programmiersprache, die neben modernen Konzepten der SW-Erstellung die Programmierung großer, eingebetter Systeme in Realzeitumgebungen unterstützt.

Spracheigenschaften Ada83

strenges Typkonzept

Datenabstraktion

Nebenläufigkeit

getrennte Übersetzbarkeit

Programmschablonen

Ausnahmebehandlung

Spracherweiterungen Ada95

Objektorientierung

hierarchische Bibliotheken

Realzeitprogrammierung

⁷ Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A-1983

⁸ Information technology-Programming languages-Ada, International Standard ISO/IEC-8652:1995(E)

4.2 Vorstellung der Methodik SDL/Ada

4.2.1 Software-Entwicklungsprozeß

Jeder SW-Entwicklungsprozeß bedarf zur Koordinierung einzelner Tätigkeiten eines organisatorischen Rahmens, der Entwurf und Implementierung des Produkts ‚Software‘ ermöglicht.

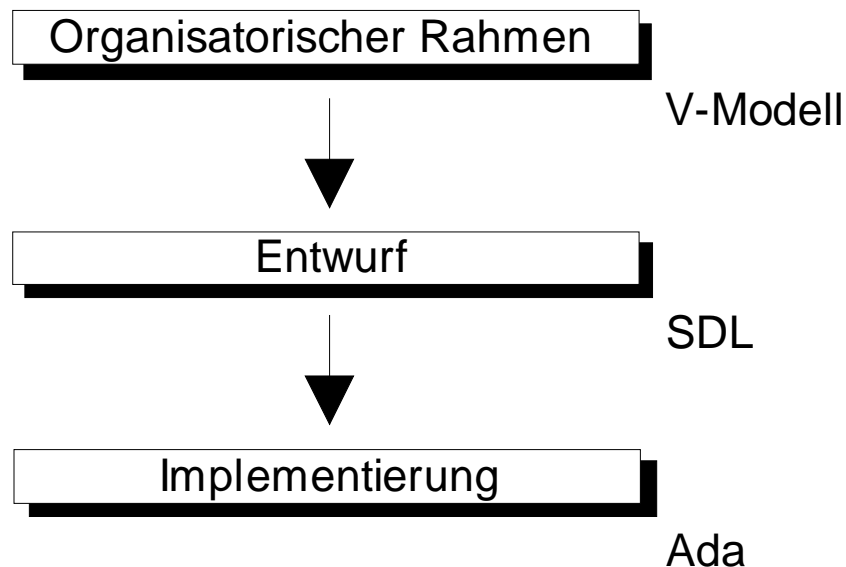


Bild 4: Basisstruktur SW-Entwicklung, Abbildung

Das V-Modell stellt einen solchen Rahmen zur Verfügung, indem mittels eines projektspezifischen Tailorings sämtliche zu erstellenden Entwicklungsergebnisse auf informaler Ebene definiert werden. Die Bedeutung eines solchen Rahmens erwächst aus der Bereitstellung eines standardisierten Vokabulars, auf dessen Basis die Kommunikation zwischen Auftraggeber und Auftragnehmer aber auch zwischen den beteiligten Entwicklern erfolgt.

4.2.1.1 Beschreibungsebenen

Die Beschreibungsebenen innerhalb des V-Modell Submodells ‚SW-Entwicklung‘ können in drei Kategorien unterschieden werden:

Beschreibungen in natürlicher Sprache stellen eine *informale Beschreibungsebene* dar, die zur Kommunikation der beteiligten Personen untereinander genutzt wird.

Entwicklungsergebnisse in Form ausführbarer Software stellen eine *formale Beschreibungsebene* dar, in der sämtliche Details der Implementierung in einer vom Rechner verständlichen Programmiersprache formuliert vorliegen.

Zwischen beiden Ebenen wird eine *semiformale Beschreibungsebene* vorgesehen, innerhalb derer der Entwurf der gewählten SW-Architektur beschrieben wird.

Der Architekturentwurf sollte gut kommunizierbar sein und gleichzeitig als Basis einer automatischen Codegenerierung genutzt werden können. Die zuletzt genannte Anforderung macht eine formale Definition der Spezifikationssprache selbst notwendig.

Die Lage der beiden Grenzlinien kann variiert werden. Hierbei ist wegen des Interpretationsspielraums eine Minimierung des informalen Anteils anzustreben. Die Lage der unteren Grenzlinie wird über den gewählten Detaillierungsgrad der Spezifikation bestimmt.

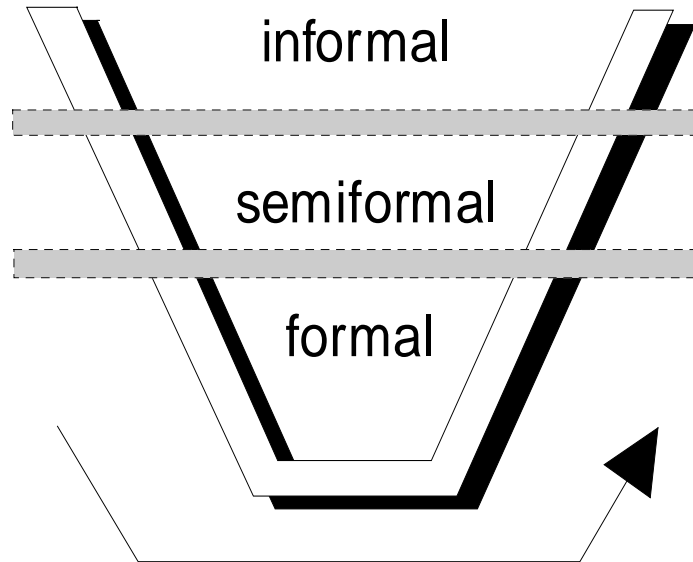


Bild 5: V-Modell, Beschreibungs- / Integrationsebenen

4.2.1.1.1 Informale Ebene

Zu Beginn jeder SW-Entwicklung stehen Beschreibungen wie Konzeptpapier, Lastenheft und Anforderungspapiere. Kennzeichnend für diese Beschreibungsebene sind natürlichsprachliche Texte, oft ergänzt um beliebige graphische Elemente, die das 'WAS' der Applikation beschreiben. Die Differenzierung zwischen *System*, *Segmenten* und *SW-Komponenten* innerhalb des V-Modells impliziert eine erste Strukturierung der Aufgabenstellung nach definierten Gesichtspunkten.

4.2.1.1.2 Semiformale Ebene

Auf Basis der informalen Beschreibungen entstehen zur Beschreibung einer geeigneten SW-Architektur Entwurfsunterlagen, die das 'WIE' zur Erfüllung der gestellten Anforderungen beschreiben (global design). Typisch für diese Beschreibungsebene ist der Bedarf nach möglichst graphischen Ausdrucksmitteln mit formal definierter Semantik, die den Entwurf der SW-Architektur anschaulich machen und eine Diskussion derselben innerhalb der Entwicklergruppe ermöglichen. Die Zulassung einer gewissen Unschärfe in der Beschreibung ist wegen der zu diesem Zeitpunkt unklaren Details oder zur Abstraktion eine essentielle Forderung an die Beschreibungssprache. Im angesprochenen Telekommunikationsumfeld hat sich die Spezifikationssprache SDL zur Beschreibung von Architektur und Verhalten des zu entwerfenden Systems verbreitet durchgesetzt.

4.2.1.1.3 Formale Ebene

Innerhalb der formalen Beschreibungsebene findet die Implementierung der mit SDL beschriebenen Architektur statt. Es liegt nahe, in SDL formulierte Entwürfe automatisch in einen Coderahmen zu transformieren. Die manuelle Vervollständigung der nicht in

SDL formulierten Details (detailed design) entspricht der Ergänzung von Algorithmen und Datentypen (ADTs). Der Umfang der direkt zu erstellenden Software kann in einem gewissen Rahmen durch angemessene Entwurfsentscheidungen gesteuert werden. Die Programmiersprache Ada bietet durch die konzeptionelle Nähe zu SDL (gemeinsames Anwendungsumfeld) ideale Voraussetzungen sowohl zur automatischen Codegenerierung (Beispiel tasking-Konzept) als auch zur manuellen Programmierung der Details (Beispiel Paket-Konzept).

4.2.1.2 Integrationsebenen

Den drei Beschreibungsebenen des Submodells stehen drei Integrationsebenen (Modul, Prozeß und Prozessorsystem(e)) gegenüber, deren Existenz den Bedarf an unterschiedlichen Testumgebungen motiviert.

4.2.1.2.1 Modul

Auf Modul-Ebene werden die üblichen Test- und Integrationswerkzeuge eingesetzt, die in der Regel als Bestandteil einer Programmierumgebung vorzufinden sind. Mittels High Level Debugger können auch in Cross-Umgebungen Kontrollfluß und Speicherinhalte in allen Details (Bitebene) beobachtet werden.

4.2.1.2.2 Prozeß

Auf Prozeß-Ebene gestaltet sich wegen der parallelen Kontrollflüsse die Beobachtung der Abläufe mit herkömmlichen Mitteln aufwendig. An dieser Stelle hat sich die Konzentration auf Prozeßzustand und Nachrichtenfluß zwischen den Prozessen bei gleichzeitiger Verbergung von Details bewährt. Da herkömmliche Debugger eine solche Abstraktion nur bedingt unterstützen, ist ein maßgeschneidertes Testwerkzeug korrespondierend zur Spezifikation vorteilhaft. Darüber hinaus besteht während der Entwicklung oft der Bedarf, fehlende Systemteile auf Spezifikationsniveau zu simulieren. Auf Basis der SDL-Spezifikation kann geeignete Software zur Test- und Simulationsunterstützung automatisch generiert werden.

4.2.1.2.3 Prozessorsystem(e)

Die Integration eines Prozessorsystems erfolgt üblicherweise durch Stimulierung und Beobachtung der realen Schnittstellen des Integrationsobjektes. In diesem Bereich ist oft der Einsatz von Protokollanalytoren oder speziellen Werkzeugen (Beispiel Lastsimulation) erforderlich. Die Vielfalt von Schnittstellentypen, Koordinierungsprobleme verteilter Architekturen und zulässige Ablaufvarianz stehen Automatisierungsanforderungen im Sinne von Regressionstests entgegen. In der Praxis sind deshalb aufwendige manuelle ‚Abnahmen‘ auf Basis von Lastenheft und Anforderungspapieren üblich, die das korrekte Verhalten des Integrationsobjektes in typischen und Grenzbereichen überprüft.

4.2.2 Software-Architekturen

4.2.2.1 Entwicklungsumgebung

Der graphische Entwurf der SDL-Spezifikation erfolgt mittels eines kommerziellen Werkzeugs. Als Ergebnis liegt ein dokumentiertes Systemmodell vor, das die

Architektur und das *Verhalten* des zu entwerfenden Systems auf Spezifikationsniveau beschreibt. Das Werkzeug überprüft hierbei die Konsistenz der Spezifikation und gestattet unter Vernachlässigung realzeitlicher Gesichtspunkte die Simulation des Entwurfs auf dem Entwicklungsrechner.

Mittels eines eigenentwickelten SDL/Ada-Frameworks wird der SDL-Entwurf in Ada-Code transformiert, der um die nicht in SDL spezifizierten Details (Datentypen und Funktionen) durch manuelle Programmierung zu ergänzen ist. Das vollständige Programm stellt eine Ada-Implementierung der SDL-Spezifikation dar, die auf Basis eines SDL/Ada-Laufzeitsystems ausführbar ist.

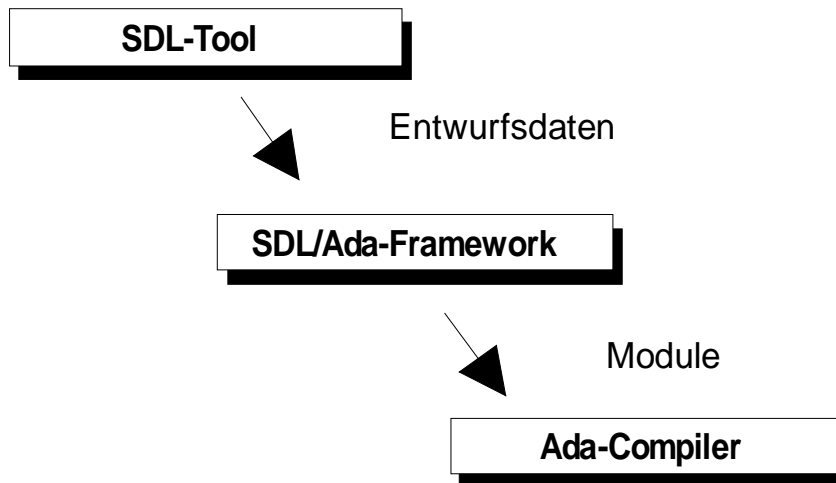


Bild 6: Werkzeugkette

Die in SDL formulierte Spezifikation erlaubt darüberhinaus Codegenerierung zur Testunterstützung indem auf das standardisierte SDL-Kommunikationsmodell aufgesetzt wird. Die Datentypdeklarationen der Signalparameter sind hierzu in Ada-Paketen bereitzustellen.

Spezielle Testwerkzeuge erlauben interaktiv oder programmgesteuert Kommunikationsverhalten und Zustände der Prozesse selektiv zu protokollieren. Durch Eingabe von Signalen über die Testschnittstelle können Prozesse stimuliert werden. Die Test-Ein-/Ausgaben erfolgen in Klartext über einen ausgezeichneten Metaprozeß auf dem Entwicklungs- oder Zielrechner.

Komponenten SDL/Ada-Framework

- SDL/Ada-Tool
 - Konsistenzüberprüfung
 - Dokumentation
 - Generierung
 - Schnittstellenpakete
 - interne Systemdarstellung

- Testcode
- SDL/Ada-Runtime
 - SDL/Ada-Interface
 - starten / beenden von Prozessen
 - Prozeßidentifikation
 - Nachrichtenverarbeitung
 - Timer
 - Testfunktionen
 - SDL/Ada-Kernel
 - Abbildung der statischen Systemstruktur
 - Verwaltung von Prozeßinstanzen
 - Fehlerbehandlung
 - Monitorfunktionen
 - Testfunktionen
- SDL/Ada-Test
 - interaktive Testschnittstelle
 - Test - Interpreter

Bild 7 faßt den methodischen Ansatz auf die Realisierungsphasen bezogen noch einmal zusammen. Die SDL-Spezifikation stellt als Ausgangs- und Endpunkt den stabilisierenden Rahmen der Ada-Implementierung dar.

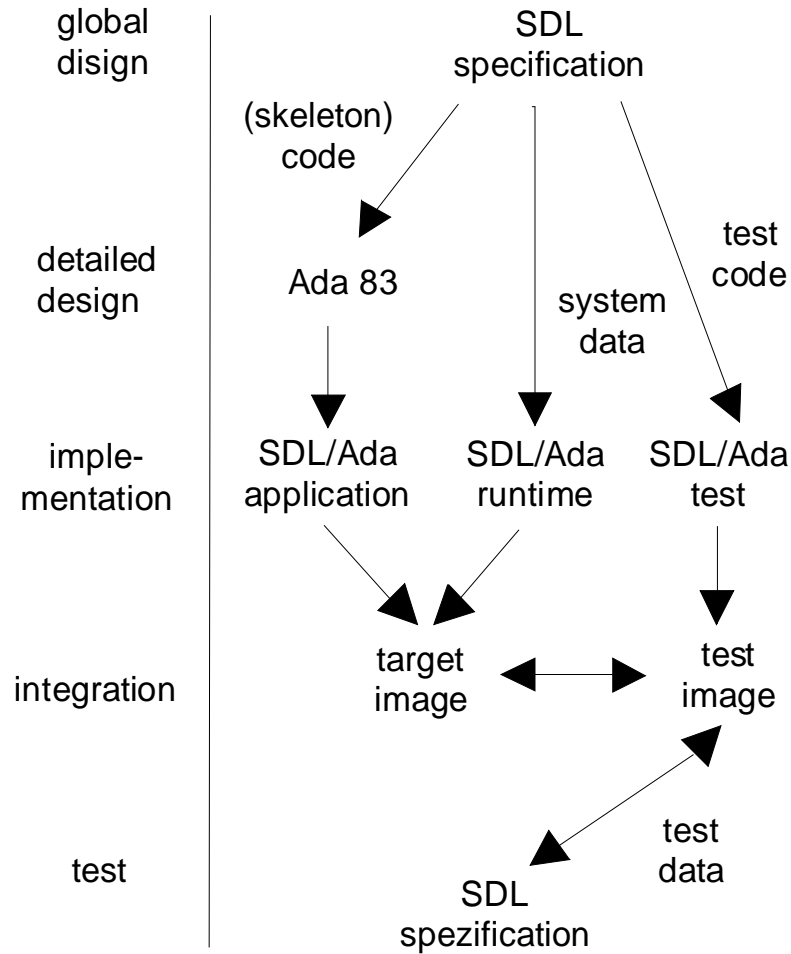


Bild 7: Entwicklungsprozeß

4.2.2.2 Zielsystem

Bild 8 zeigt ideale und praktische Zielarchitekturen SDL/Ada-basierter Implementierungen. Zwischen Betriebssystem und SDL-Applikation sind die beiden Laufzeitsysteme für Ada und SDL/Ada angesiedelt. Die geschichtete Architektur macht die SDL-Applikation prinzipiell auf jeder Ada-Plattform lauffähig. In der Praxis ist jedoch häufig die Portabilität der idealen SW-Architektur durch direkte Sichtbarkeit der tiefer liegenden Schichten eingeschränkt.

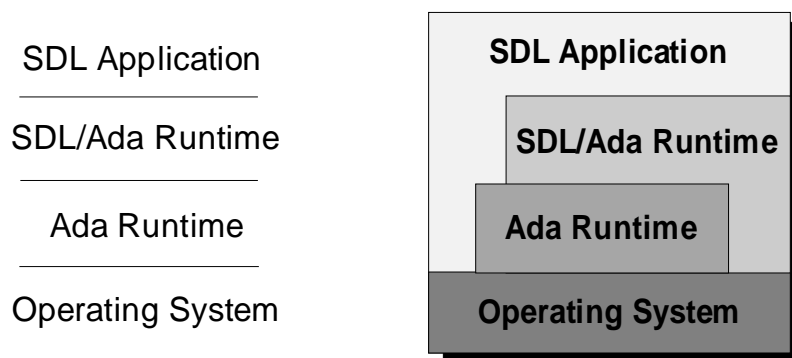


Bild 8: ideale / praktische SW-Architektur

Ein wesentlicher Grund hierfür liegt in der Bereitstellung effizienter asynchroner Kommunikationsmechanismen, die i.d.R. vom Betriebssystem, nicht jedoch von Ada83 bereitgestellt werden. Spracherweiterungen von Ada95 wie z.B. 'protected types' bieten interessante Ansätze, die die ideale Architektur in greifbare Nähe rücken lassen.

4.3 Anmerkungen zur Methodik SDL/Ada

4.3.1 Charakterisierung

Der im vorliegenden Artikel beschriebene methodische Ansatz ist durch die Verknüpfung von Standards geprägt. Die Integration der drei genannten Standards kann problemlos vollzogen werden. Die Verwendung einer Spezifikationssprache in der mittleren Ebene des V-Modells liefert eine scharfe Abgrenzung von Anforderungsbeschreibung, Architektur- Entwurf und Implementierung. Die unterschiedlichen Sprachebenen erzwingen eine saubere Differenzierung der durchzuführenden Entwicklungsarbeiten wobei der Übergang von der Grob- zur Detailspezifikation bzw. Implementierung rechnergestützt und somit konsistent vollzogen wird. Eine wesentliche Eigenschaft der vorgestellten Methodik ist die Skalierbarkeit, die durch angemessene Abstraktion auf Spezifikationsebene erreicht wird.

4.3.2 Spezifikationssprache vs. Implementierungssprache

Die Verwendung einer formal definierten Spezifikationssprache dient der präzisen Beschreibung eines geforderten Systemverhaltens bzw. dessen Struktur, die weit über informale und oft mißverständliche Papiere hinausgeht. Die 'high-level'-Sprachmittel der Spezifikationssprache liefern einen anwendungsunabhängigen jedoch Domainenspezifischen Baukasten erprobter SW-Komponenten. Der Freiheitsgrad einer direkten Programmierung wird auf Architekturebene gezielt eingeschränkt, wodurch eine Transparenz der Software auf Architekturebene sichergestellt und standardisierte Testschnittstellen ermöglicht werden. Die Spezifikation stellt ein simulierbares Systemmodell dar, das als Analyse- und Implementierungsbasis genutzt werden kann. Durch Einsatz eines Codegenerators kann das Programmgerüst der Implementierung direkt aus der Spezifikation abgeleitet werden. Die Überwindung der Sprachgrenzen (semantic gap) bereitet wegen der konzeptionellen Verwandtschaft beider Sprachen keine Probleme. Bild 9 zeigt die realisierte Ada-Implementierung elementarer SDL-Konstrukte.

4.3.3 Ergebnisse

4.3.3.1 Projekt-Erfahrungen

Der auf die drei Standards abgestützte SW-Entwicklungsprozeß wird durch die vorgestellte Methodik klar strukturiert. Darüberhinaus wird durch die eingesetzten Sprachen eine globale Projektkultur gefördert, die die Grundlage nachvollziehbarer Entwicklungsergebnisse bildet. Die von kommerziellen Werkzeugen bereitgestellte Simulationsfähigkeit des SDL-Entwurfs auf dem Entwicklungsrechner verliert mit der Verfügbarkeit des SDL/Ada-Lauzeitsystems auf der gleichen Rechnerplattform an Bedeutung. In der Wartungsphase hat sich die Verfügbarkeit der Testmittel auf Spezifikationsniveau im Zielsystem für Fehleranalysen als hilfreich erwiesen. Defizite

bzgl. Anwendung und Leistungsfähigkeit der drei Standards behindern singular und in verstärktem Maße in der Kombination die initiale Akzeptanz der Vorgehensweise.

SDL	Ada83
architecture	data structure
behaviour	program structure
system	program(s)
block	package
process	task
channel	[routing mechanism]

Bild 9: Abbildung SDL -> Ada83

4.3.3.2 Produkt-Qualität

Die homogene, visualisierbare Modellierung der Software im Sinne von SDL macht die Implementierung in weiten Teilen nachvollziehbar (Programmgerüst). Der 'Umweg' über die Spezifikationsprache zahlt sich durch die graphische Dokumentation der SW-Architektur und Verfügbarkeit adäquater Testmittel spätestens in der Wartungsphase aus. Die Kombination mit den bekannten qualitätsfördernden Eigenschaften der Programmiersprache Ada macht die Methodik auch gegenüber kommerziellen 'C'-Code generierenden SDL-Werkzeugen überlegen. Die Standardisierung schützt die eigene Wertschöpfung durch Portierbarkeit und Unabhängigkeit von bestimmten Werkzeugen, Plattformen und Herstellern. Diesem Aspekt wird insbesondere bei langlebigen Produkten größte Bedeutung zugemessen.

4.3.4 Entwicklungspotentiale

4.3.4.1 Standards

Die aktuell verfügbare SDL/Ada-Entwicklungsumgebung entstand im Rahmen eines Projektes und ist aus nachvollziehbaren Gründen bzgl. ihrer Funktionalität auf Basismechanismen (SDL-Subset) beschränkt. Die Einschränkungen beziehen sich auf verzichtbare SDL-Sprachmittel und Coderahmengenerierung für die Prozeßimplementierung, deren Vervollständigung wünschenswert erscheint. Portierungen des SDL/Ada-Frameworks auf weitere Plattformen (z.B. POSIX, Ada95 bare) werden durch die geschichtete Architektur begünstigt. Die Protokollierung der Prozesskommunikation mit Hilfe der SDL/ Ada-Testwerkzeuge liefert derzeit Testdaten, die in einer speziellen Textform vorliegen. Eine Weiterentwicklung in Richtung standardisierter Nachrichten-

darstellung (message sequence charts⁹, MSC) würde die Integration graphischer Komponente kommerzieller SDL-Werkzeuge ermöglichen. Die SDL/Ada-Testwerkzeuge bieten eine interaktive und eine programmierbare textorientierte Benutzerschnittstelle. In diesem Bereich wäre eine Annäherung an bestehende Standards (Tree and Tabular Combined Notation¹⁰, TTCN) im Sinne des Artikels vorteilhaft.

4.3.4.2 Architekturen

Die im Kapitel Zielsysteme angesprochenen SW-Architekturen basieren auf realen Implementierungen. Stichworte wie ‚Multiprogramming‘, ‚Multiprocessing‘, ‚distributed Processing‘, ‚distributed SDL‘ und ‚Ada Partitions‘ bieten vielfältige Aspekte über weitere sinnvolle Architekturen nachzudenken.

4.3.4.3 Objektorientierung

Nach [3] werden *Systemstruktur* und *Systemverhalten*, die Basiselemente jeder SDL-Spezifikation, als zwei unabhängige Dimensionen zur Beschreibung der statischen und dynamischen Aspekte eines Systemmodells betrachtet. *Vererbung* als kennzeichnende Größe objektorientierter Ansätze zur Abstraktion und Wiederverwendung von Software wird, wie in Bild 10 dargestellt, als dritte, unabhängige Dimension beschrieben.

Objektorientierte Spracherweiterungen von SDL92 und Ada95 fordern Klarheit bzgl. einer evolutionären Weiterentwicklung der beschriebenen Methodik SDL/Ada.

Unter der in Bild 11 stilisierten *horizontalen* Integration von OO und SDL sei ein zunächst objektorientierter Ansatz verstanden, der in der Implementierung mit SDL weitergeführt wird. Semantische Unterschiede beider Welten lassen diese Integrationsform fragwürdig erscheinen.

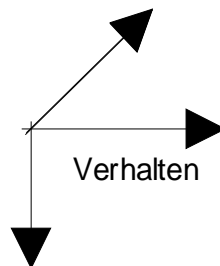


Bild 10: Dimensionen System-Modell

Die *vertikale* Integration bearbeitet getrennte SW-Bereiche mit beiden Ansätzen parallel. Die Ergebnisse werden erst zur Bindezeit zusammengeführt und stellen somit keine echte Integration dar.

⁹ New Recommendation Z.120, CCITT, Message Sequence Charts (MSC), 1992

¹⁰ Draft Recommendation X.292, CCITT, OSI Conformance Testing Methodology and Framework for Protocol Recommendations for CCITT Applications - The Tree and Tabular Combined Notation (TTCN), 1992

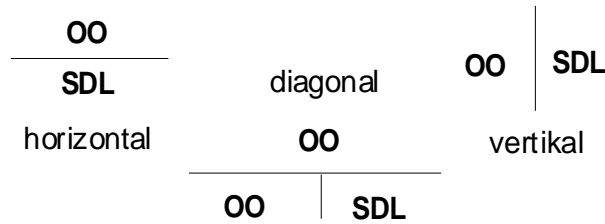


Bild 11: OO / SDL Integrationsstrategien

Bei der vorgeschlagenen *diagonalen* Integration wird Objektorientierung zunächst als übergeordnete Methodik nur im Bereich der Analyse (OOA) einsetzt. Nach Identifikation aktiver und passiver Objekte werden im Grobentwurf die aktiven Objekte wie bisher als SDL-System modelliert. Die passiven Objekte entsprechen aus SDL-Sicht den abstrakten Datentypen (ADTs), die auch im Feinentwurf objektorientiert modelliert werden können.

Bild 12 zeigt den ‘diagonalen’ Migrationspfad zur Integration von OO und SDL im Ada-Kontext. Die Sprachmittel von Ada95 gestatten die Methoden- Integration im Feinentwurf.

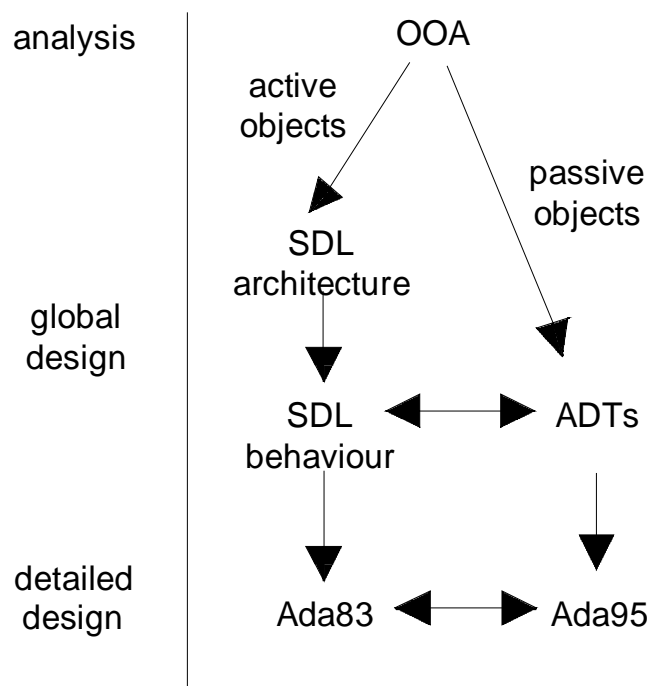


Bild 12: OO / SDL / Ada Integration

4.4 **Schlußbemerkungen**

Die Verknüpfung einer anwendungsspezifischen Spezifikationssprache mit einer leistungsfähigen Programmiersprache hat sich in einem realen Projekt als methodischer Ansatz voll bewährt. Die Standardisierung beider Sprachen und des praktizierten organisatorischen Rahmens dient der langfristigen Sicherung der eigenen Wertschöpfung. Die Aussagen sind im Wesentlichen auf folgende Erfahrungen gestützt:

Sprachen

- Die praktizierte SW-Entwicklungskultur wird weitgehend durch die zur Verfügung stehende(n) Sprache(n) geprägt und somit positiv beeinflusst.
- Die Sprachen SDL und Ada haben durch ihren gemeinsamen Entstehungscharakter (Reviews) Ähnlichkeiten bzgl. Qualität und Leistungsfähigkeit.
- Die Abgrenzung von Grob- und Feinentwurf bzw. Implementierung wird ohne semantischen Bruch durch zwei Sprachebenen erzwungen.
- Zum Entwurf stehen formal definierte high-level Sprachelemente zur Verfügung, die ein nahezu intuitiv verständliches Prozeß- und Kommunikationsmodell verwenden.
- Die Abbildung des Entwurfs auf die Programmiersprache kann automatisiert werden.
- Integration und Test kann durch die homogene Kommunikationsinfrastruktur auf Spezifikationsniveau durchgeführt werden.
- Der Detaillierungsgrad des Grobentwurf's (Abstraktion) kann nach Bedarf individuell gesteuert werden.

Standards

- Die Standardisierung ist Grundlage der Verfügbarkeit kompatibler, kommerzieller Werkzeuge.
- Das projektweite Implementierungsmodell bleibt über die Lebensdauer des Produkts konstant.
- Kommunikationsprobleme zwischen den beteiligten Personen werden durch das ‚standardisierte Kommunikationsprotokoll‘ weitgehend vermieden.
- Stabilität und Verbreitung der Standards vergrößern das in Frage kommende Mitarbeiterpotential, Reibungsverluste durch Mitarbeiterwechsel werden minimiert.
- Architektur und Prozeßverhalten können mittels kommerzieller Werkzeuge entworfen und visualisiert werden.
- Unabhängigkeit von Plattform und Hersteller kann für einen Systemwechsel z.B. aus wirtschaftlichen Überlegungen genutzt werden.
- Der durch den Entwicklungsstandard geprägte organisatorische Rahmen stellt das erforderliche informale Vokabular auch zur projektübergreifenden Kommunikation bereit.

4.5 Literatur

- [1] F. Belina, D. Hogrefe, A. Sarma (1991) „SDL with Applications from Protocol Specification“, Prentice Hall
- [2] R. Braek, O. Haugen (1993) „Engineering Real Time Systems, An objectoriented methodology using SDL“, Prentice Hall
- [3] B. Selic, G. Gullekson, P. Ward (1994) „Real-Time Object-Oriented Modeling“, John Wiley & Sons, Inc.

5 Ada-Sprachkonzepte für Datenbankanwendungen

Frank-Peter Lüß
Helsinkistr. 70
24109 Kiel

Zusammenfassung

PL/SQL ist eine prozedurale Erweiterung von SQL, der nichtprozeduralen Datenmanipulationssprache für relationale Datenbanken. PL/SQL ist in vielen Eigenschaften stark an Ada orientiert.

Durch die Integration mit dem ORACLE-Server, den 4GL-Werkzeugen und dem Case-Designer von ORACLE ist PL/SQL die state-of-the-art-Sprache, um 3GL-Anteile in Datenbankanwendungen zu erstellen.

5.1 Einleitung/Motivation: Von der Schnittstellenspezifikation zum Datenmodell

Erfahrungshintergrund insbesondere 2 Entwicklungsstufen desselben DB-Anwendungssystems:

1. Stufe (1988-90) mit DG/SQL (relationales DB-System von Data General), SGU (einfacher Maskengenerator), Ada und FORTRAN77 (Übergang zur DB und zu den Masken via FORTRAN).
2. Stufe (1994-97) mit ORACLE7 mit FORMS 4.0 (4GL-MMI-Generator), REPORTS 2.0 (4GL-Berichte-Generator), GRAPHICS 2.0 und PL/SQL u.a. Werkzeugen wie IMP, EXP, Loader; C(++) für graphische DB-Anwendung.

Projekterfahrungen:

- Weitverbreitete 4GL-Tools einsetzen
- Bei Anforderungen auf die Möglichkeiten der 4GL-Tools Rücksicht nehmen
- MMI klar konzipieren und durch Prototyping Akzeptanz erzielen
- Datenmodell als Kernstück frühzeitig beginnen und sauber ausführen
- SW-Entwurf trotz 4GL und/oder hoher 3GL-Werkzeuge ausführlich gestalten
- Auf den Einsatz von C und C++ verzichten

Bei prozedural orientierten SW-Entwurfsansätzen sind die Schnittstellen zwischen den verschiedenen Modulen von besonderer Wichtigkeit. Sie bilden bei größeren Projekten

auch die Kommunikationsschnittstellen zwischen verschiedenen Entwicklern. Unzulänglichkeiten im Schnittstellentwurf können daher teuer werden.

Je weiter sich die Moduln dynamisch entkoppeln lassen, um so einfacher lassen sie sich testen und integrieren. Komplexere Datenschnittstellen sollten daher konsequenterweise als Teil eines Datenmodells gesehen werden und die Schnittstellenspezifikation auf die Beschreibung der Parameter bei Moduln reduziert werden, die in einer Aufrufbeziehung zueinander stehen. Bei DB-Anwendungen bildet das E/R-Datenmodell/die Datenstruktur das tragende Gerüst und den Integrationsrahmen für die übrigen SW-Anteile.

5.2 Übersicht über PL/SQL

PL/SQL als querschnittliche Komponente im ORACLE-Tool-Set

PL/SQL steht im Zusammenhang mit den folgenden Werkzeugen/Komponenten von ORACLE zur Verfügung:

- ORACLE-Server (speichert DB-Trigger, stored Program Units (Functions, Procedures, packages (getrennt nach Spec und Body))), speichert alle anderen DB-Objekte wie Tabellen, Sichten (Views), Sequencer, Indices und natürlich die Datensätze).
- SQL*DBA und SQL*Plus (Umgebungen für die direkte Durchführung von SQL-DDL- und -DML-Befehlen, aber auch Ausführung von PL/SQL-Blöcken)
- Procedure Builder (Bearbeitung von Bibliotheken, Bearbeitung von Program Units)
- FORMS (4GL-Tool zur Generierung interaktiver DB-Anwendungen, wird nach Anfrage vom 18.04.1997 bei der ORACLE-Niederlassung Hamburg von ORACLE als strategisch wichtiges Produkt gesehen, an dessen Einstellung keinesfalls gedacht wird, sondern das permanent auf dem Stand der Technik gehalten wird und z.Z. Web-fähig gemacht wird)
- GRAPHICS (4GL-Tool zur Generierung von Graphiken)
- REPORTS (4GL-Tool zur Generierung von komplexen DB-Reports)
- CASE*Designer (Meta-Werkzeug zum Entwurf von DB-Anwendungen, zum Generieren von Program Units, FORMS- und REPORTS-Moduln)

FORMS, GRAPHICS und REPORTS haben enge Schnittstellen zueinander. PL/SQL-Blöcke können auch in Wirtssprachen (Host languages) wie C oder Ada eingebettet werden. Prozedurale Anteile werden generell durch Ereignisse ausgelöst (Trigger).

5.3 Hauptmerkmale von PL/SQL

PL/SQL ist eine prozedurale Spracherweiterung von SQL. Konzeptionell ist PL/SQL stark an Ada orientiert.

Schlüsselwörter, Blockstruktur, Unterstützung von SQL, Kontrollstrukturen (Konditionen, Iterationen, Ablaufkontrolle), einfaches Cursor-Handling durch Cursor-For-Loops, Fehlerbehandlung durch Exception Handling, Modularisierung durch Unterprogramm- und Paket-Technik, Information Hiding durch Einkapselung (Sichtbarkeitsregeln) sind gegeben. In Client-Server-Umgebungen ist eine höhere Performance durch Netzentlastung bei RPC und Stored Program Units (Server leistungsfähiger) gegeben. DB-Anwendungen sind soweit portierbar, wie ORACLE auf den Plattformen zur Verfügung steht.

Im Vergleich zu Ada bietet PL/SQL keine Generik und einfachere, weniger restriktive Datentypen.

5.4 *Eigenschaften von PL/SQL*

5.4.1 Datentypen und Datentyp-Konvertierung

Auf den zugrunde liegenden Zeichensatz, die lexikalische Konstanten, Trennzeichen, Namensvergabe bei Identifikatoren etc. wird nicht näher eingegangen. Es gibt Multi- und Single-Line-Kommentare.

Skalare Datentypen sind:

BINARY_INTEGER -2147483647 .. +2147483647

BINARY_INTEGER subtypes:

NATURAL 0 .. +2147483647

POSITIVE 0 .. +2147483647

NUMBER[(precision, scale)] precision <=38, -84 <= scale <= 127

NUMBER subtypes: DEC, DECIMAL, DOUBLE_PRECISION, FLOAT, INTEGER, INT, NUMERIC, REAL, SMALLINT (Kompatibilität zu ANS/ISO, IBM)

CHAR[(maximum_length)] maximum_length < 32767

CHAR subtypes: CHARACTER, STRING (Kompatibilität zu ANS/ISO, IBM)

VARCHAR[2](maximum_length) maximum_length < 32767

LONG bis zu 2 GB Text

RAW Binärdaten bis zu 32767 Bytes

LONG RAW Binärdaten bis zu 2 GB (durch die OLE-Fähigkeit von FORMS lassen sich mit diesem Datentyp in einfacher Weise OLE-Container definieren, sodaß z.B. eine direkte Integration mit Word, Excel etc. gegeben ist)

BOOLEAN

DATE

ROWID Pseudocolumn zur internen Identifikation eines DB-Datensatzes

MLSLABEL (nur Trusted ORACLE) Access Control Daten

Bei der Deklaration können skalare Typen aus der DB durch das Attribute %TYPE 'geerbt' werden.

Die wichtigsten expliziten Konvertierungsfunktionen sind:

- TO_CHAR von NUMBER, DATE
- TO_DATE von NUMBER, CHAR
- TO_NUMBER von CHAR

Es gibt auch die Möglichkeit zur impliziten Konvertierung, die ich persönlich nicht schätze. Ein weiterer Kritikpunkt ist, daß die Wertebereiche der Datentypen zwischen PL/SQL und dem Server (d.h. den Datenbanktabellen) z.T. differieren.

5.4.2 Deklarationen, Namenskonventionen, Reichweite und Sichtbarkeit

Die Deklaration von Konstanten, Typen, Variablen, Records und PL/SQL-Tables (Arrays) ist möglich. Bei der Deklaration von Variablen ist eine Default-Vorbelegung möglich. Wie im Beispiel gezeigt können Records und PL/SQL-Tables definiert werden. Mit PL/SQL-Tables kann eine Array-Verarbeitung in PL/SQL durchgeführt werden. Nützlich sind auch die Typ-Deklarationen unter Benutzung der Attribute %TYPE und %ROWTYPE, sodaß DB-Konforme Typdeklarationen möglich sind.

Die Reichweiten und Sichtbarkeiten gehorchen innerhalb von PL/SQL-Program-Units den Ada-Regeln. Außerhalb der Units wird das Lokalitätsprinzip befolgt: referenzierte PL/SQL-Programm-Units werden zunächst im gleichen Modul gesucht, danach wird nach stored Program Units geguckt. Bei Benutzung einer stored Program-Unit aus einem anderen Schema sollte ein public Synonym vorhanden sein. Bei der Referenz von DB-Objekten (Tabellen, Views) wird ohne Angabe des Schemas unterstellt, daß die betreffenden Objekte im Schema des Übersetzenden/Ausführenden vorliegen. Bei benutzung von DB-Links kann direkt auf Tabellen einer Remote-DB zugegriffen werden, wobei der Zugriff automatisch (z.B. via TCP/IP bei Rechnernetzen oder ISDN bei verteilten Anwendungen) durch die unterliegende Netzwerksoftware TNS erfolgt.

5.4.3 Zuweisungen, Ausdrücke, Vergleiche und eingebaute Funktionen

Bei Operatoren, Zuweisungen, Ausdrücken, Vergleichen gelten ebenfalls die Ada-Regeln. Interessant ist, daß alle Prozeduren und Funktionen, die in SQL zur Verfügung stehen, ebenfalls in PL/SQL zur Verfügung stehen:

allgemeine Funktionen: DECODE, NVL

Fehlerberichts-funktionen: SQLCODE, SQLERRM

Numerik-Funktionen: ABS, CEIL, COS, COSH, EXP, FLOOR, LN, LOG, MOD, POWER, ROUND, SIGN, SIN, SINH, SQRT, TAN, TANH, TRUNC

Buchstabenfunktionen: ASCII, CHR, CONCAT (||), INICAP, INSTR, INSTRB, LENGTH, LENGTHB, LOWER, LPAD, LTRIM, NLS_INITCAP, NLS_LOWER, NLS_UPPER, NLSSORT, REPLACE, RPAD, RTRIM, SOUNDEX, SUBSTR, SUBSTRB, TRANSLATE, UPPER

Konvertierungsfunktionen: CHARTOROWID, CONVERT, HEXTORAW, RAWTOHEX, ROWIDTOCHAR, TO_CHAR 3x, TO_DATE, TO_LABEL, TO_MULTI_BYTE, TO_NUMBER, TO_SINGLE_BYTE

Datumsfunktionen: ADD_MONTHS, LAST_DAY, MONTH_BETWEEN, NEW_TIME, NEXT_DAY, ROUND, SYSDATE, TRUNC

5.4.4 PL/SQL-Tabellen und Datensätze (Records)

PL/SQL-Tabellen sind indizierte Felder. Records können auch geschachtelt werden.

5.4.5 Kontrollstrukturen

Selektion:

```
if      <condition> then <statement>;
{elsif <condition> then <statement>;}
{else          <statement>;}
end if;
```

Iteration:

Es gibt einfache Schleifen, Ausgang mit exit-Statement. Es gibt While- und For-Schleifen. Besonders hilfreich für eine aussagekräftige Programmierung: Namen von Schleifen.

Eine besondere Form der Schleifen ist die Cursor-For-Schleife, bei der in besonders eleganter und direkter Weise mit den Abfrageergebnissen gearbeitet werden kann (siehe Beispiel).

Sequence:

NULL und GOTO.

5.4.6 SQL-Unterstützung, Cursor-Management und Transaktionskontrolle

Möglichkeit für dynamisches SQL (per Aufruf von vordefinierten Standard-Funktionen), auch DDL möglich. Einfache SQL-Ausdrücke (INSERT, UPDATE, DELETE) können unverändert direkt eingefügt werden, wobei alle sichtbaren Parameter direkt (ohne den Umweg über Host-Variablen) direkt zur Verfügung stehen.

Das SELECT-Statement wird zu SELECT .. INTO, wobei hier zwei typische Ausnahmesituationen vorliegen können: 'no_data_found' und 'too_many_rows'.

Ist eine Transaktion (Folge von aus logischen Gründen untrennbaren DB-Operationen, die die DB von einem logisch konsistenten in einen anderen logisch konsistenten Zustand überführt) erfolgreich durchgeführt worden, wird sie durch ein COMMIT {WORK} abgeschlossen und damit der neue DB-Zustand für alle DB-Nutzer wirksam und sichtbar, wobei evtl. temporäre Sperrungen aufgehoben werden.

Schlägt eine Operation einer Transaktion fehl, wird durch eine ROLLBACK-Operation der vorherige konsistente DB-Zustand auch für den diese Transaktion auslösenden Nutzer wieder wirksam.

COMMIT und ROLLBACK stehen daher als PL/SQL-Kommandos zur Verfügung. Ein COMMIT ist typischerweise nach einer Folge (z.B. als letztes Kommando am Ende eines Blockes) von PL/SQL-Kommandos zu finden, in der die DB-Inhalte verändert werden.

Ein ROLLBACK ist typischerweise im Exception-Handler zu finden.

5.4.7 Fehlerbehandlung (Exception Handling)

Vordefinierte Exceptions gibt es für die Standard-Ausnahmesituationen. Darüber hinaus gibt es nutzerdefinierte Exceptions, die deklariert und durch ein RAISE aufgerufen werden können. Eine Zuordnung einer selbstdefinierter Exception zu einem ORACLE-Fehlercode wird durch das Pragma

```
EXCEPTION_INIT(<exception_name>, <ORACLE_error_number>)
```

durchgeführt. Eine solche Exception wird durch den entsprechenden ORACLE-Fehler geraist.

Im Zusammenhang mit der Fehlerbehandlung sind die PL/SQL-Funktionen SQLCODE und SQLERRM von besonderer Bedeutung. Die aktuelle Exception kann am Ende des zugehörigen Exception-Handler-Abschnitts wieder durch das PL/SQL-Kommando RAISE erneut geraist werden.

Die Übernahme des Exception-Handling-Konzepts von Ada ist von großem Vorteil zur Generierung von Programmen, die

- sehr robust laufen (die Kontrolle wird auf die nächsthöhere Ebene zurückgegeben),
- die Auswirkungen von Laufzeitfehlern auf ein Minimum beschränken (DB wird konsistent gehalten),

- die Analyse von Fehlersituation im Rahmen der SWPÄ unterstützen (aufgetretener Aufrufreihenfolge kann gespeichert werden zusammen mit Zustandsinformationen der betroffenen DB-Objekte).

5.5 Vorteile von PL/SQL

DB ist gleichzeitig Entwicklungs-, Integrations- und SWPÄ-Umgebung (durch die Objekt-Navigatoren übersichtlich unterstützt), wobei mit entsprechend vielen verschiedenen DB-Instanzen (Entwicklung, Test und Produktion) gearbeitet wird. Die Sprachkonstrukte erlauben die Entwicklungen strukturierter und gut lesbarer SW.

Das Exception-Handling erlaubt die Generierung robuster und leicht pflegbarer Programme. Die Einbettung von SQL erfolgt direkt, d.h. keine EXEC SQL-Einschübe, keine zusätzlichen HOST-Variablen etc. Die fehlende Generik wird durch die Möglichkeit, intelligente, die aktuelle Laufzeitumgebung (bei FORMS) auswertende Trigger zu generieren, ausgeglichen.

Es stehen eine Standardpakete zur Verfügung, die die einfache Nutzung der DB-Funktionalität erlauben (z.B. die Job-Steuerung). Aber auch dynamisches SQL (DML und auch DDL) ist durch Nutzung von Paketfunktionen möglich.

Die Lastverteilung in verteilten Systemen kann besser gesteuert werden.

5.6 Brückenschlag zu Ada

Da ORACLE mit Pro*Ada auch eine Schnittstelle zu Ada bietet, können Ada-Anwendungen ebenfalls direkt im ORACLE-Umfeld integriert werden. Die Übereinstimmung von Ada und PL/SQL bietet dabei die Möglichkeit, dieselben Programmierrichtlinien und SWE-Vorgehensweisen in beiden Sprachumfeldern anzuwenden und gleichzeitig die hohe Produktivität der ORACLE-Anwendungsgeneratoren zu nutzen. Das durchgängige Sprachkonzept erhöht in derartigen Projekten die Lesbarkeit und damit die Wartbarkeit. Durch die Verbreitung von ORACLE ist eine hohe Portabilität gegeben.

6 Ada in der Umwelttechnik

Dr. Hubert B. Keller
Forschungszentrum Karlsruhe - Technik und Umwelt
Institut für Angewandte Informatik
Postfach 3640, D-76021 Karlsruhe

6.1 Einleitung

Informationen über das Forschungszentrum Karlsruhe:

F&E-Schwerpunkte (je ca. 1/3 Anteil)

- Energie
- Umwelt
- Mikrosystemtechnik

ca. 3.000 Mitarbeiter

Einsatz von Ada am Institut für Angewandte Informatik:

- verschiedene Ada-Projekte seit 1984
- komplexe Anwendungen unter Echtzeitbedingungen mit Integrationscharakter bzgl. eingesetzter Methoden(prototypischer Modelcharakter)
- Entwicklung/Einsatz auf WS und PC
 - Vax
 - Alpha
 - Intel-PC

Projektziele:

Umweltgerechte, intelligente Führung komplexer verfahrenstechnischer Prozesse am Beispiel der thermischen Abfallbehandlung sowie Innovatives Umweltmonitoring mit intelligenten Sensoren

Einsatz innovativer Verfahren und Werkzeuge (Neuronale Netze, Machine Learning, Bildverarbeitung, verteilte Kommunikation, Prozeßankopplungen, Zeitreihenverwaltung, Regelverarbeitung, Animation)

6.2 Bisherige Ada-Projekte

- Modellierungs- und Echtzeitsimulationssystem K_advice
- graphische Systeme (UI)
 - GAP
 - XGAP
 - XM_View
- Simulation und Prognose von Prozeßverhalten mit neuronalen Netzen
- Feuerleistungsreglung auf Basis IR-Kamera
- C³R-System zur automatische Modellierung von Prozeßzusammenhängen (Struktur und funktionale Abhängigkeiten -> maschinelles Lernen)
- X_Executive
Verwaltungs- und Ausführungssystem für verteilte Applikationen
- Tool für Entwicklung von Bildverarbeitungsanwendungen (Kommando-Script-gesteuert)
- Animationstool mit Bildeinblendung
- Verteilte Multisensorsystem-Meßnetze

6.2.3 X-Executive

X(=Applikation)-Executive (>250 packages, >2000 units)

- Applikationen unter Echtzeitbedingungen
- Datenerfassung / Prozeßankopplung, Bildverarbeitung, . . .
- verteilte Kommunikationsstruktur
- graphische Benutzerschnittstelle
- Zeitreihenverwaltungssystem
- . . .

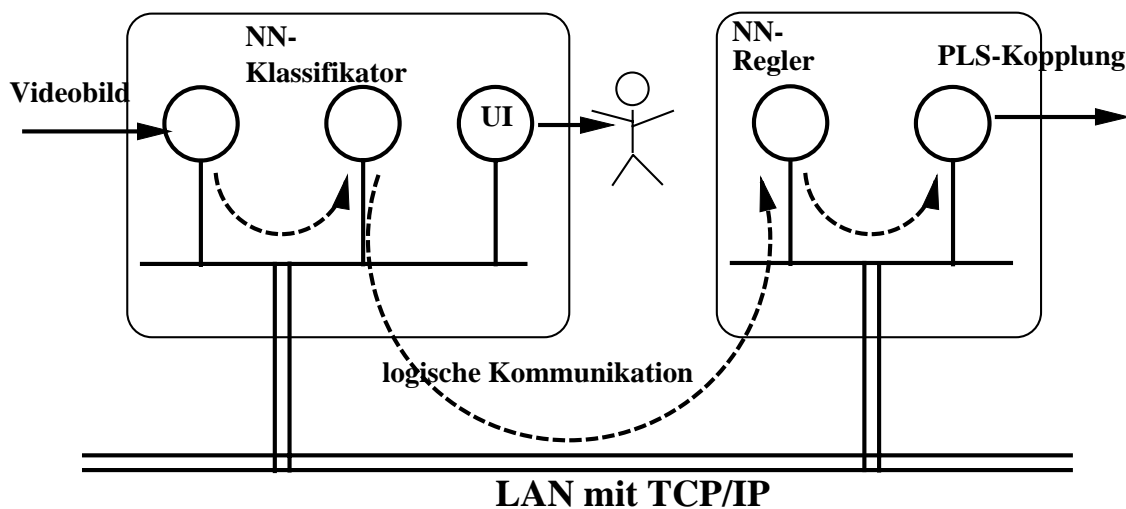
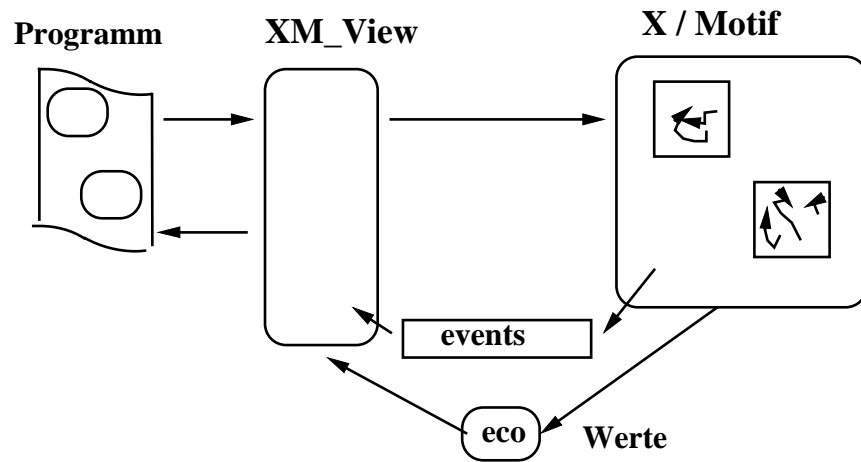


Bild: Systemstruktur

6.2.4 XM_View

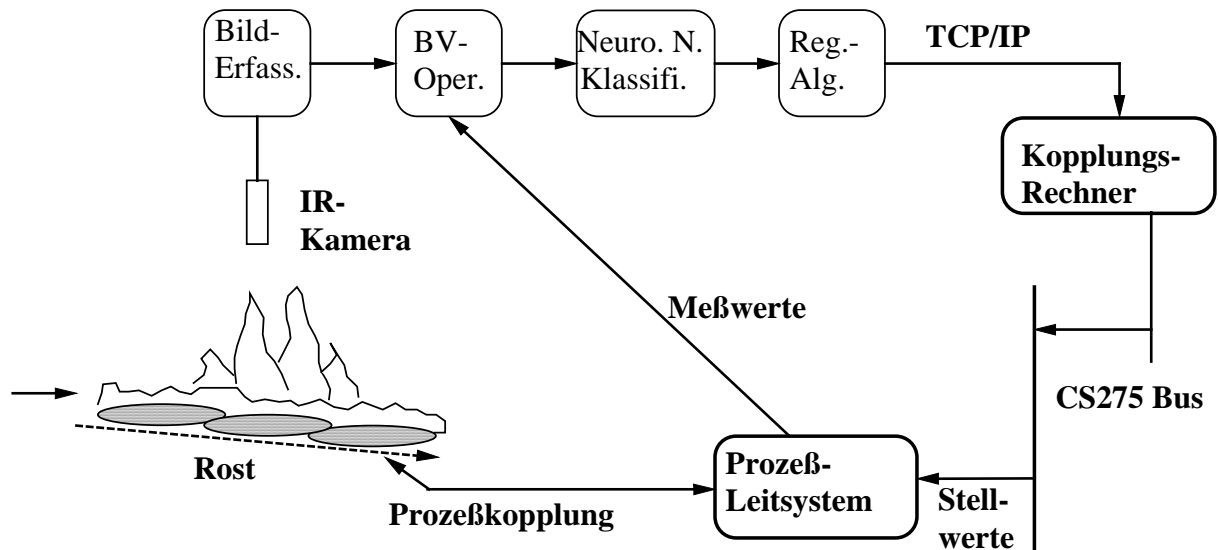
- Werkzeug zur Erzeugung graphischer Oberflächen
- Anwenderprogramm steuert den Ablauf
- direkte Benutzerschnittstelle für Anwenderprogramm oder Laufzeitsystemkomponente (default)
- ereignisorientierte Verarbeitung (klassenspezifische events für Fenster mit / ohne Vorgabe)
- synchrone / asynchrone Kommunikation (direkte Schnittstelle)

- über 100 Pakete, über 600 units (ohne Basisbausteine und Animation)



Struktur der Kommunikation

6.2.5 Feuerleistungsregelung in der thermischen Abfallbehandlung



Feuerleistungsregelung

6.2.6 C³R-System zur Maschinellen Modellierung

Ziel:

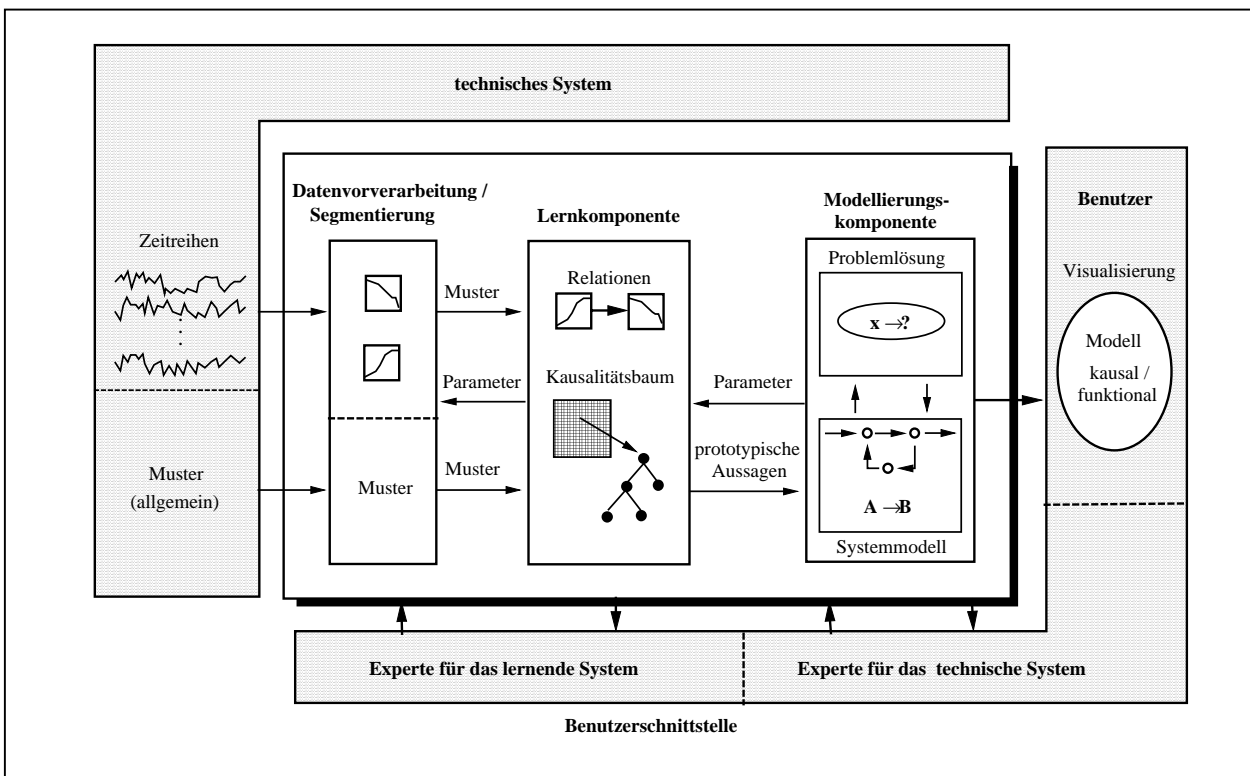
Steuerung/Kontrolle komplexer dynamischer techn. Systeme

Ansatz:

Nachbildung des Lernverhaltens menschlicher Bediener bzgl. Ursache-Wirkungs-Relationen und funktionaler Beziehung

Anwendungen:

Pilotanlage zur thermischen Müllbehandlung
kognitionspsychologisch orientiertes System

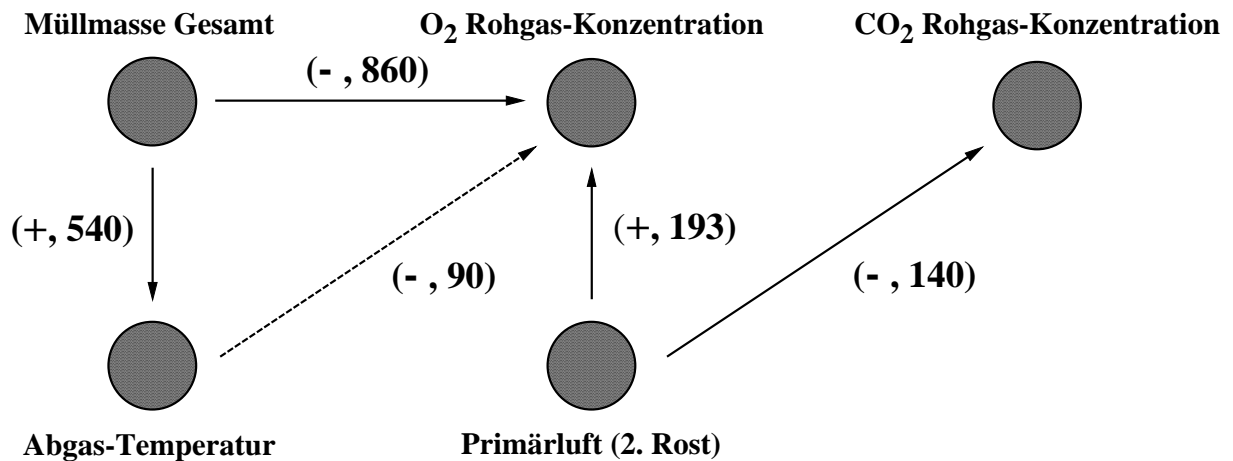


Architektur des C³R-Systems

Anwendung an der FZK-Versuchsanlage zur Müllverbrennung TAMARA

Automatisch abgeleitete Regel: Müllmasse Gesamt → Abgas Temperatur (T151):

WENN Müllmasse_Gesamt in [253,5; 270,5]
UND Veränderung=Abnahme nach [235,4; 252,2]
innerhalb von 50.6 ± 20,5 Sekunden
UND T151 in [941,7; 988,7]
DANN Veränderung=Abnahme nach [930,1; 971,4]
mit Verzögerung von 540.0 ± 24,5 Sekunden,
innerhalb von 335,0 ± 26.9 Sekunden.
 Vertrauensgrad: 0,667.




Abgeleitete Kausalstruktur


6.3 Zusammenfassung


- Ada hat sich ab ca. 84/85 aus Software-technischen Gründen hervorragend für komplexe Projekte (auch im Umweltbereich) geeignet
- Produktivität ist sehr gut bei gleichzeitig geringer Fehlerrate
- Debugging von Tasks nicht optimal
- Tasking z. T. schlecht abgebildet (blockierend bei Externschnittstelle)
- Browsing von Bibliotheken fehlte
- einfache Konzepte wie Datenstruktur-Hierarchisierung in Pakethierarchie abbilden (Semantik)
- Typentkopplung über access-Einbindung
- record als private, access als limited private zur echten Kapselung
- Trennung in deskriptive und Ausführungsstrukturen (quasi OO)
- Bausteine aus 1985 auch in 1995 wiederverwendet
- Ada95 mit Vererbung und weiteren features bietet noch bessere Unterstützung (Stabilität?)
- Einbettung von Ada in Umgebungen (Graphik usw.) noch nicht optimal (Portabilitätsprobleme)


III Allgemeine Themen


7 Was erwartet der Nutzer von Ada und welche Anforderungen hat er?


		<p>IT III 3</p>
<p><u>Thema:</u></p> <p style="text-align: center;">Was erwartet der Nutzer von Ada[®] und welche Anforderungen hat er ?</p> <p><u>Gliederung:</u></p> <ol style="list-style-type: none"> <i>1. Typisierung der DV - Vorhaben der Bundeswehr</i> <i>2. "Ada - Vergangenheit "bei der Bundeswehr</i> <i>3. Wo wird Ada derzeit eingesetzt?</i> <i>4. Probleme in der Vergangenheit, Erfahrungen</i> <i>5. Forderungen an Ada aus Nutzersicht</i> 		
<p>01.04.97</p>	<p>Axel Thiemann</p>	

		IT III 3
<p style="text-align: center;">Typisierung der DV- Vorhaben</p> <ul style="list-style-type: none">* Waffeneinsatzsysteme (WES)<ul style="list-style-type: none">- <i>typ. Prozessdatenverarbeitung</i>- <i>Realzeit</i>- <i>sicherheitskritisch</i> * Führungsinformationssysteme (FüInfoSys)<ul style="list-style-type: none">- <i>große Datenmengen</i>- <i>interaktive Grafik</i>- <i>formatierte u. unformatierte Textverarbeitung</i>- <i>Anbindung an Datenbanken</i>- <i>Kommunikation über Draht und Funk</i> * Fachinformationssysteme<ul style="list-style-type: none">- <i>kommerzielle DV</i> (<i>Logistik, Personal, Gehaltsabrechnung</i>) * Simulation<ul style="list-style-type: none">- <i>Entwicklung</i>- <i>Ausbildung</i>- <i>Taktik (Operations Research)</i>		
01.04.97	Axel Thiemann	

		IT III 3
<p>"Ada -Vergangenheit" bei der Bw</p> <p>Erfahrungen: Software - Entwicklung u.Pflege</p> <ul style="list-style-type: none">- <i>zu teuer</i>- <i>zu lange</i>- <i>zu fehleranfällig</i> <p>Konsequenz: Ada</p> <ul style="list-style-type: none">- <i>Vorhaben Sperber (Karlsruher Ada Kompiler)</i>- <i>Ada - Erlass des BMVg (1989)</i>- <i>Unterstützung verschiedener Ada- Aktivitäten (z.B. Bindings)</i>		
01.04.97	Axel Thiemann	

		IT III 3
<p>Probleme in der Vergangenheit, Erfahrungen</p> <p>Probleme</p> <ul style="list-style-type: none">- <i>Kompilierqualität</i>- <i>Realzeitverhalten</i>- <i>Datenbank- und Grafikanbindung</i>- <i>Schnittstelle zu Sprachen- und Fremdsystemen</i>- <i>Akzeptanz in der Industrie</i> <p>Erfahrungen</p> <ul style="list-style-type: none">- <i>positiv</i>		
01.04.97	Axel Thiemann	

		IT III 3
<p>Einsatz von Ada</p> <p>* Waffeneinsatzsysteme (WES) - <i>Ada häufig eingesetzt, insbesondere in Luftfahrtsvorhaben</i></p> <p>* Führungsinformationssysteme (FüInfoSys) - <i>Ada praktisch nicht eingesetzt</i></p> <p>- <i>BMVg - Weisung: Verwendung von COTS Produkten</i></p> <p>- <i>Chance für Ada als COTS Produkt</i></p> <p>* Fachinformationssysteme - <i>praktisch keine Anwendung</i></p> <p>* Simulation - <i>Entwicklung</i> - <i>Ausbildung --- häufig</i> - <i>Taktik --- steigend</i> <i>(Operations Research)</i></p>		
01.04.97	Axel Thiemann	

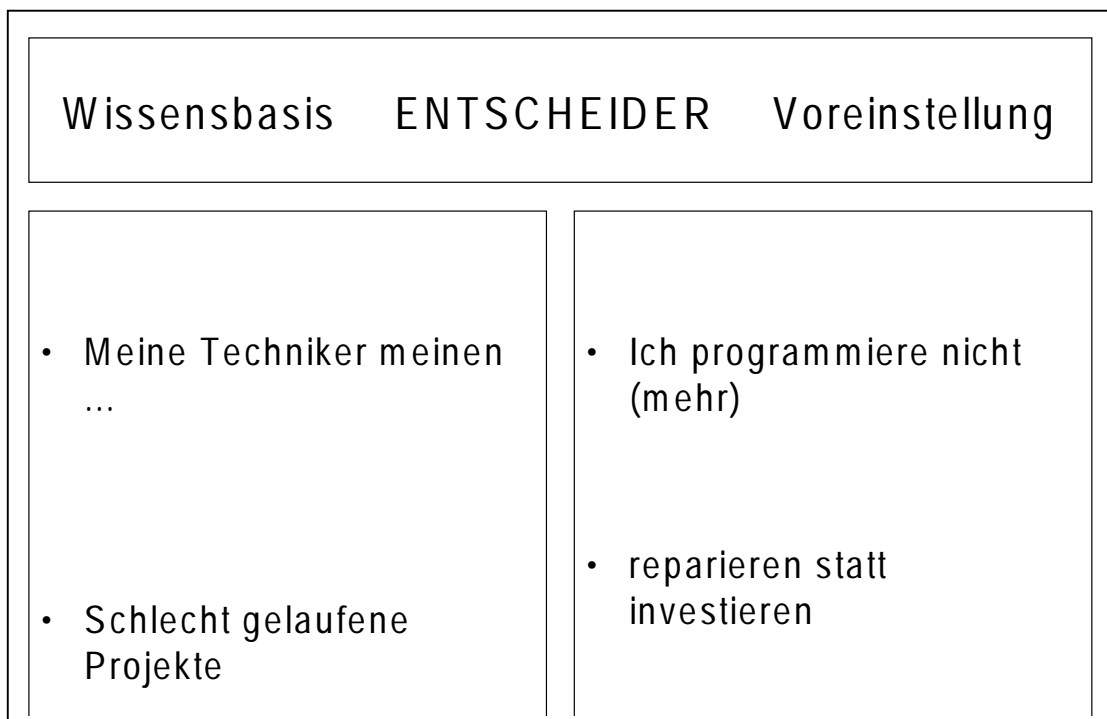
		IT III 3
<p><u>Forderungen:</u></p> <p>technisch:</p> <ul style="list-style-type: none">- <i>Produktionscompiler u. Entwicklungsumgebung für Ada 95</i>- <i>Datenbankanbindung</i>- <i>Grafikanbindung (Toolboxen)</i>- <i>Fremdsystemschnittstellen</i>- <i>Gemeinsame Hardware - Plattform für Entwicklung und Einsatz</i>- <i>Betriebssystemanbindung</i>- <i>Einbindung von "comerial of the sheft Produktion" (COTS)</i> <p>organisatorisch:</p> <ul style="list-style-type: none">- <i>Ausbildung an Hochschule</i>- <i>Motivation des Firmenmanagemantes</i>		
01.04.97	Axel Thiemann	

8 Wie verkaufe ich Ada ?

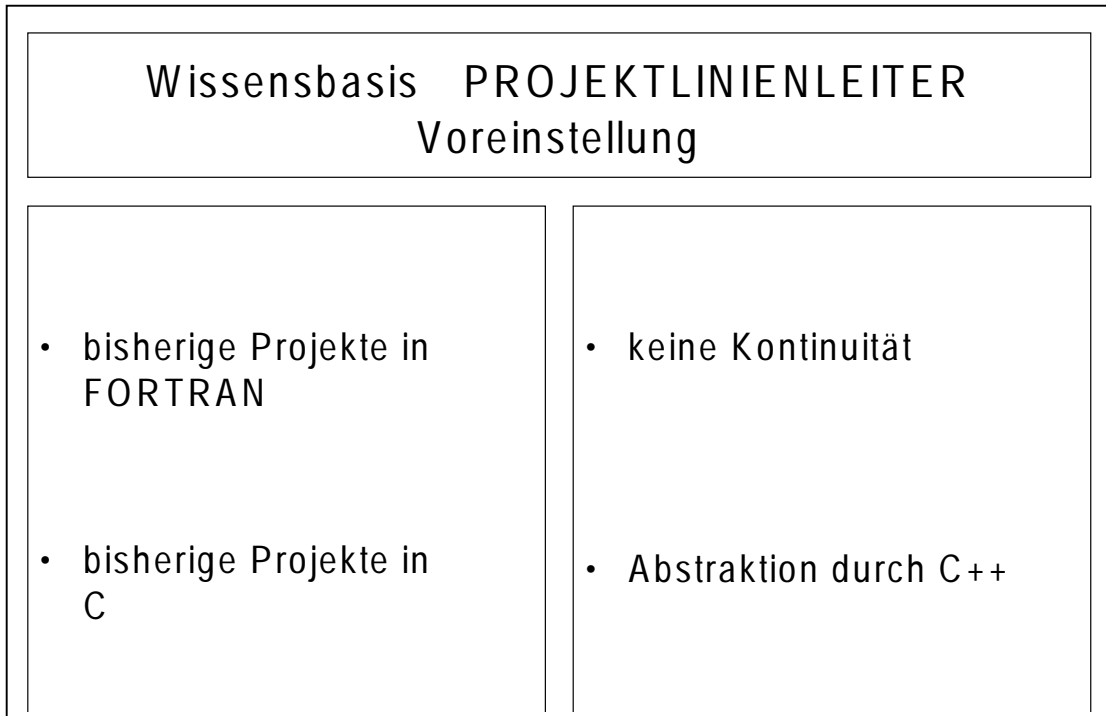
Kai Lucas
Software Consultant
Olchung-Neuesting
e-mail: 100564.2277@COMPUSERVE.COM

- Was liegt vor
- Was ist zu wecken
- Was muß geschehen

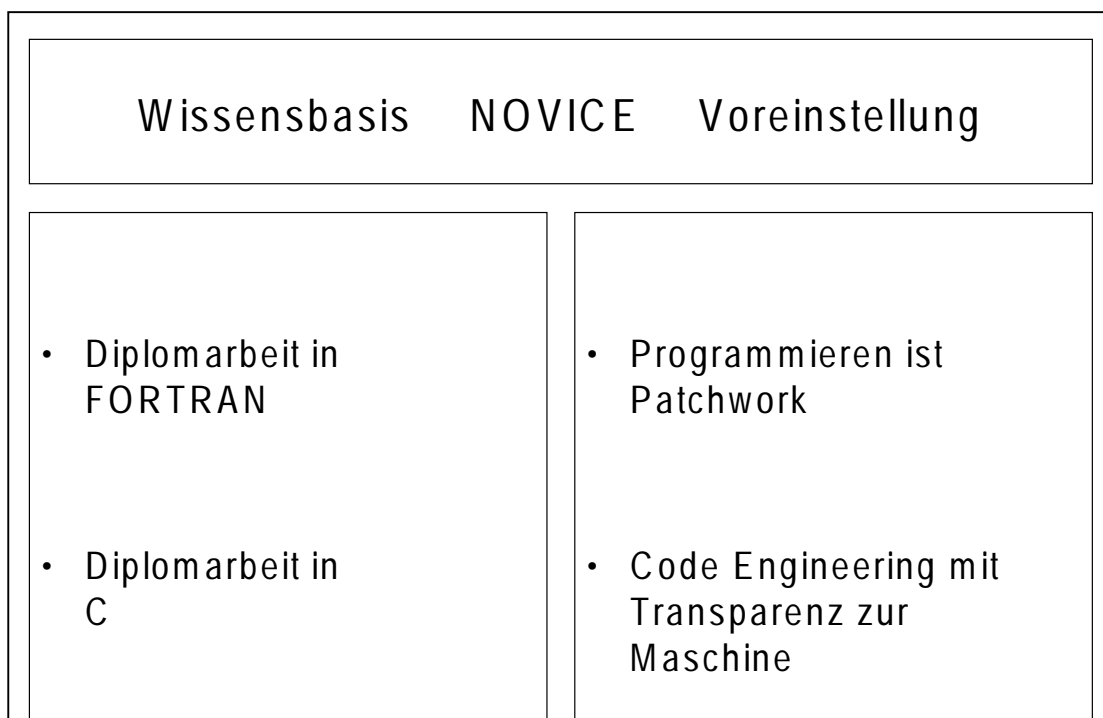
8.1 Wissensbasis ENTSCHEIDER Voreinstellung



8.2 Wissensbasis PROJEKTLINIENLEITER Voreinstellung



8.3 Wissensbasis NOVICE Voreinstellung



8.4 Motivation/Erwartung beim Entscheider

auszurichtende Motivation	ENTSCHEIDER	aufzubauende Erwartung
<ul style="list-style-type: none">• kein Vergleich mit anderen		<ul style="list-style-type: none">• strategische Entscheidung durch eigene Meinung

8.5 Motivation/Erwartung beim Projektleiter

auszurichtende Motivation	PROJEKTLINIENLEITER	aufzubauende Erwartung
<ul style="list-style-type: none">• Software Engineering<ul style="list-style-type: none">- im Projekt- in der Projektfolge		<ul style="list-style-type: none">• kostengünstig<ul style="list-style-type: none">- in der Erstellung- in der Pflege- in der Übertragbarkeit

8.6 Motivation/Erwartung beim Novicen

auszurichtende Motivation	NOVICE	aufzubauende Erwartung
<ul style="list-style-type: none">• Software Engineering im Programm		<ul style="list-style-type: none">• Transparenz zum Problem

8.7 NOVICE

zu überwinden sind	NOVICE	durchzuführende Aktionen
<ul style="list-style-type: none">• Diskrepanz: „militär“ - nein und „high tech“ - ja• Vorurteile: ineffizient, komplex		<ul style="list-style-type: none">• Ada vermitteln Seminar: „SWE mit Ada“• projektbegleitende Beratung: „gemeinsam umsetzen“

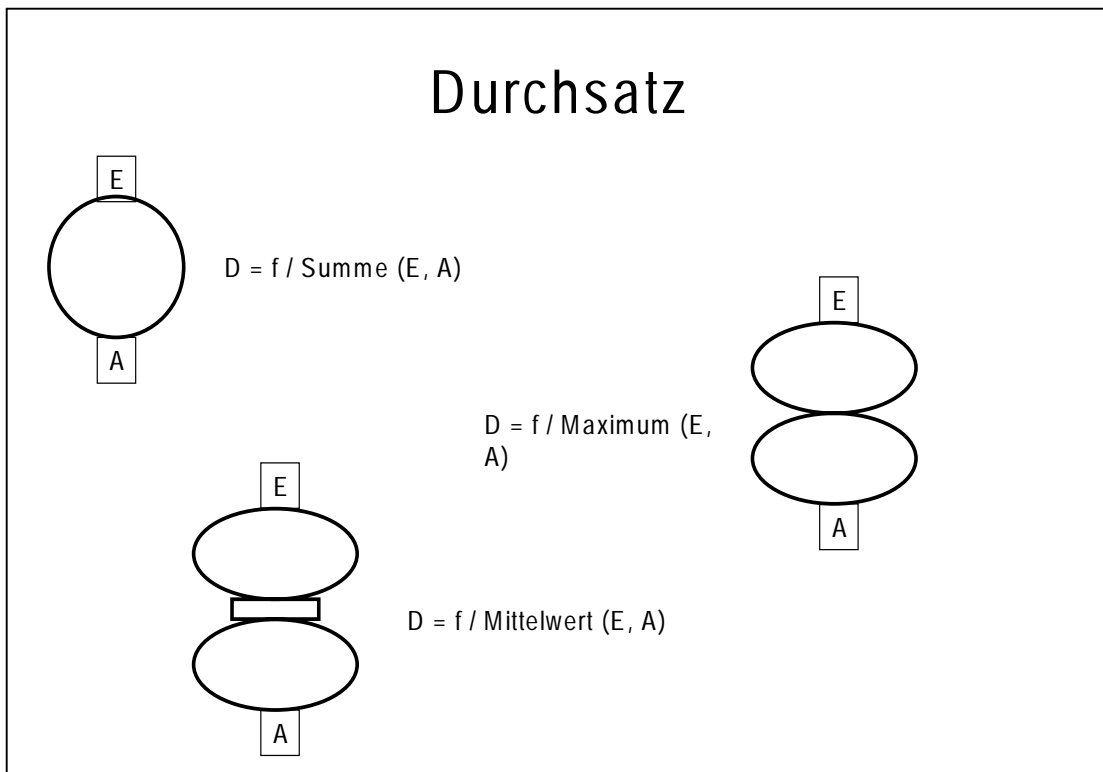
8.8 PROJEKTLINIENLEITER

zu überwinden sind	PROJEKTLINIENLEITER	durchzuführende Aktionen
<ul style="list-style-type: none"> • Lemming Engineering: C++ ist objektorientiert • Vererbung und Polymorphismus gestalten einfacher 		<ul style="list-style-type: none"> • Erfahrungen fortschreiben Seminar: „Realtime Engineering mit Ada“ • projektbegleitende Beratung: „review“ • Einbringen von FORTRAN

8.9 ENTSCHEIDER

zu überwinden sind	ENTSCHEIDER	durchzuführende Aktionen
<ul style="list-style-type: none"> • oneway Software Entwicklung „V-Modell“ 		<ul style="list-style-type: none"> • Motivations Seminar: „Ada für Entscheider“ • Projekt auflegen • evolutionäre Software Entwicklung „Spiralmodell“

8.10 Durchsatz

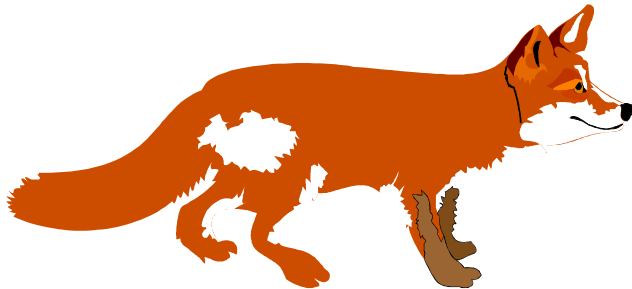


IV Projekte 2

9 Softwarepflege mit Ada – das Projekt KWS-Fuchs



Softwarepflege mit Ada: das Projekt KWS-FUCHS



Rudolf Landwehr

Competence Center Informatik GmbH
Lohberg 10, 49716 Meppen

Tel. 05931-805-450, Fax 05931-805-100
Email landwehr@cci.de

“Just in time”
Softwarepflege und -änderung
mit

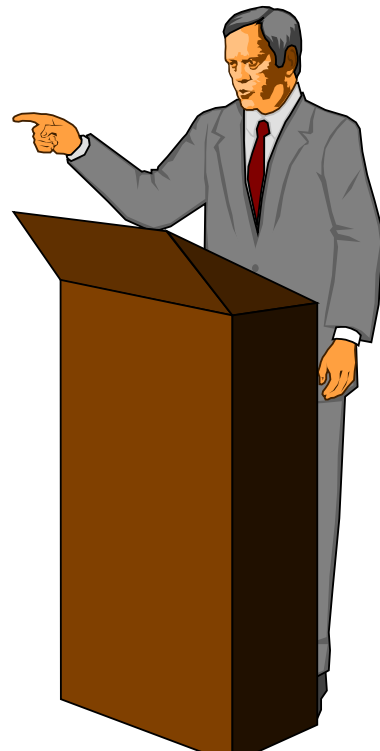
Ada



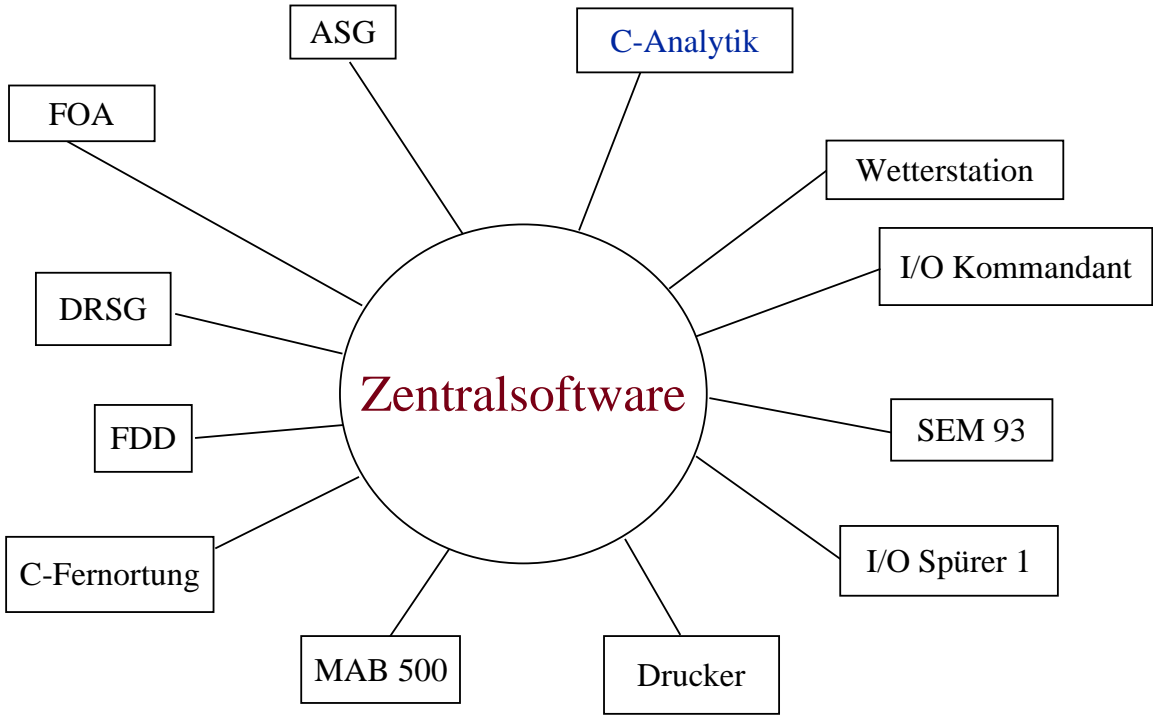
Übersicht

- ❑ Gestern - Entwicklung
 - Anforderungen an die SWKE
 - Schnittstellen und Prozesse
 - Zeitlicher Ablauf
 - Truppenversuch

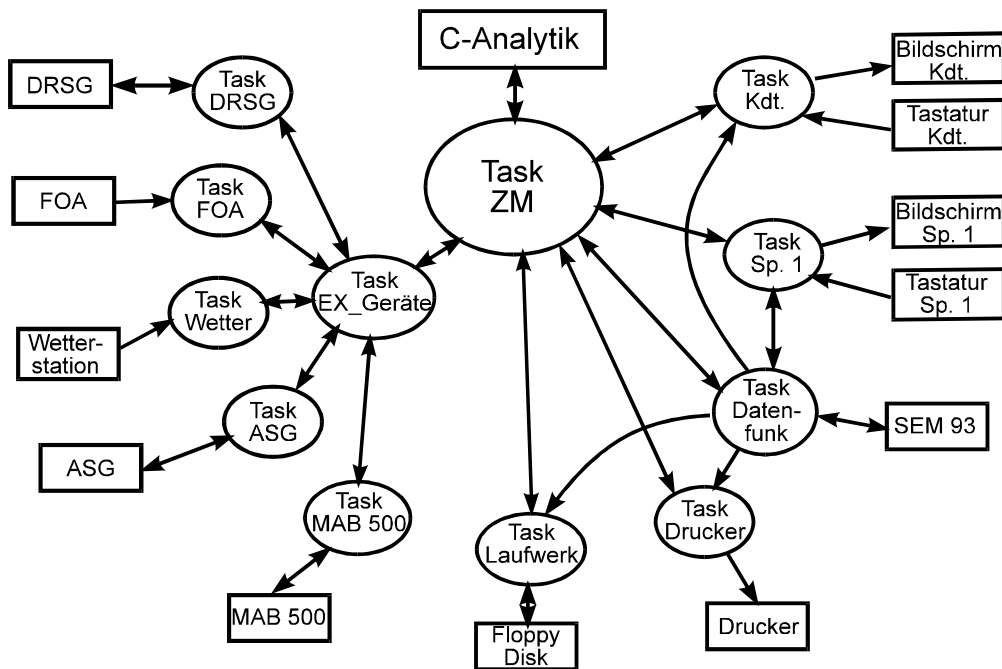
- ❑ Heute - SWPÄ
 - Vertragliche Regelungen
 - Änderungsumfang
 - Erfahrungen mit Ada
 - Fazit und Status



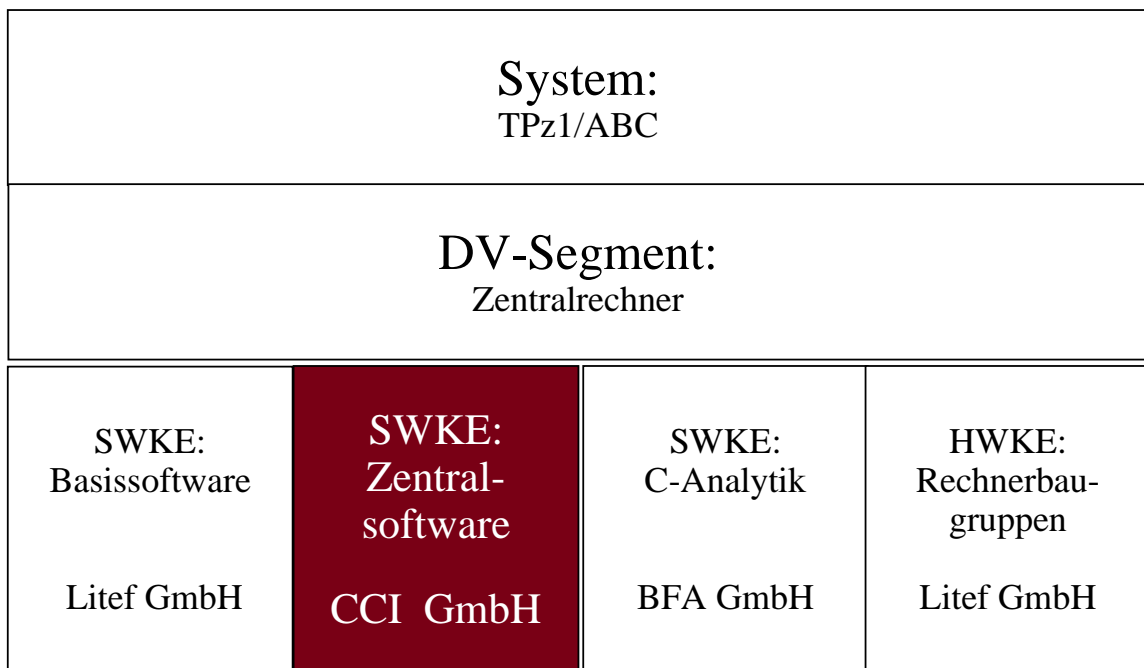
Externe Schnittstellen



Software Prozeßmodell



Einbettung nach V-Modell



Integration



Abschluß Truppenversuch

Ergebnisse Truppenversuch

☺ **Kein Fehler in der Ada-Software aufgetreten**

- Auftraggeber wünscht Änderungen im Programmablauf
- Auftraggeber wünscht funktionale Erweiterungen

Es wurden keine Gewährleistungsansprüche geltend gemacht

Änderungen werden realisiert im Rahmen der SWPÄ

Zeitlicher Ablauf

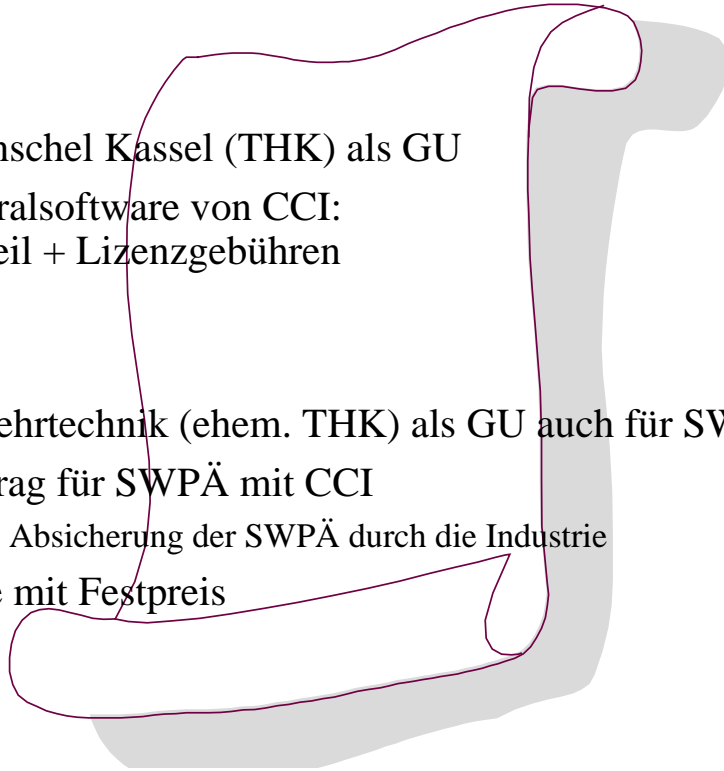
- 21.01.1992 Angebotsabgabe
- 01.02.1993 Entwicklungsbeginn
- 24.05.1993 Abnahme Prototyp durch den ÖAG
- 10.09.1993 Vertragsabschluß
- 20.10.1993 Beginn der SW-Integration
- 23.11.1993 THK-QS Abnahme
- 06.12.1993 Übergabe an den ÖAG

- 03.11.1994 Abschluß Truppenversuch
- 1995 Relative Ruhe
- 14.03.1996 SWPÄ, Beginn Serienreifmachung
- 30.11.1996 Teilabnahme SW, Probleme bei HW
- April 1997 Systemintegration

1992						
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

Vertragliche Regelungen

- ❑ Entwicklung
 - Thyssen Henschel Kassel (THK) als GU
 - SWKE Zentralsoftware von CCI:
Festpreisanteil + Lizenzgebühren
- ❑ SWPÄ
 - Henschel Wehrtechnik (ehem. THK) als GU auch für SWPÄ
 - Rahmenvertrag für SWPÄ mit CCI
 - langfristige Absicherung der SWPÄ durch die Industrie
 - Einzelabrufe mit Festpreis



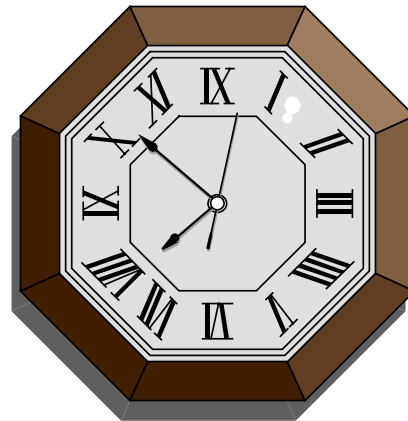
Änderungswünsche des ÖAG

- ❑ Echtzeituhr
- ❑ MAB500 (Dosisleistungsmessung)
- ❑ A-Spüren
- ❑ Wetterstation
- ❑ Dialogsteuerung
- ❑ Meldungswesen
- ❑ Probenarchivierung
- ❑ Kommunikation über Modem
- ❑ Sonstiges



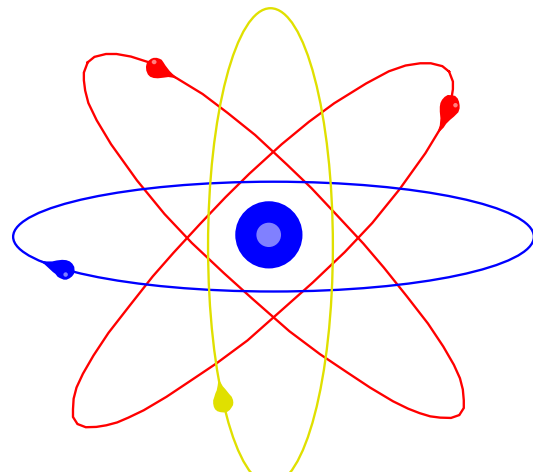
Echtzeituhr

- ❑ Integration einer Echtzeituhr
- ❑ Erweiterung des Menütitels "Optionen" um die Menüoption "Echtzeituhr"
- ❑ Änderungen der Prozesse
 - Task ZM, Task Kommandant und Task Spürer 1



Änderungen MAB500

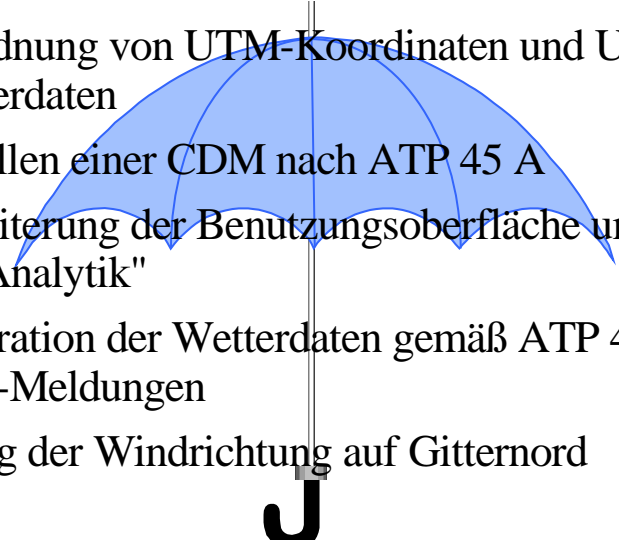
- ❑ Anzeige der aktuellen Dosisleistung auf den Bildschirmen von Kommandant und Spürer 1
- ❑ Darstellung der Trendanzeigen auf den Bildschirmen von Kommandant und Spürer 1
- ❑ Änderungen der Prozesse
 - Task Datenfunk, Task Drucken, Task ZM, Task Kommandant und Task Spürer 1



Änderung A-Spüren

- ❑ Automatische Erstellung von NBC 4 Nuklear Meldungen unter dem Menüpunkt "A-Spüren"
- ❑ Meßdatenerfassung auch mit MAB500
- ❑ Automatische Erstellung von NBC 4 Nuklear Meldungen entsprechend
 - vorgegebenen Algorithmen zur Rastermessung
 - vorgegebenen Wegstrecken
 - vorgegebenem Algorithmus für Konturlinien
- ❑ Änderungen der Prozesse
 - Task ZM, Task Datenfunk, Task Drucken, Task Laufwerk, Task Kommandant und Task Spürer 1

Änderungen Wetterstation

- ❑ Zuordnung von UTM-Koordinaten und Uhrzeit zu den Wetterdaten
 - ❑ Erstellen einer CDM nach ATP 45 A
 - ❑ Erweiterung der Benutzungsoberfläche um den Menüpunkt "W-Analytik"
 - ❑ Integration der Wetterdaten gemäß ATP 45 A in NBC-Meldungen
 - ❑ Bezug der Windrichtung auf Gitternord
 - ❑ Änderungen der Prozesse
 - Task ZM, Task Datenfunk, Task Drucken, Task Laufwerk, Task Kommandant und Task Spürer 1
- 

Änderungen in der Dialogsteuerung 1

- ❑ Menüpunkte "Löschen" und "Beenden" mit Sicherheitsabfrage
- ❑ Beim Speichern, Drucken, Senden oder Löschen
 - Anzeigen der Meldung
 - Erzeugen einer Dialogbox "Wirklich"
- ❑ Dialogbox für "Keine Diskette im Laufwerk"
- ❑ Beim Bearbeiten einer vorhandenen Meldung Dialogbox erzeugen mit Inhalt "Überschreiben oder aufbewahren"
- ❑ Beim Beenden eines Untermenüpunktes soll der zuletzt ausgewählte Hauptmenüpunkt automatisch angewählt werden

Änderungen in der Dialogsteuerung 2

- ❑ Nach Beenden einer Aktion im Message Browser soll nur zur nächsthöheren Ebene verzweigt werden
- ❑ Nach Anzeige des letzten Gerätes im Menü "Selbsttest" Anzeige von "Letztes Gerät"

- ❑ Änderungen der Prozesse
 - Task Kommandant, Task Spürer 1 und Task Laufwerk

Änderungen Meldungswesen

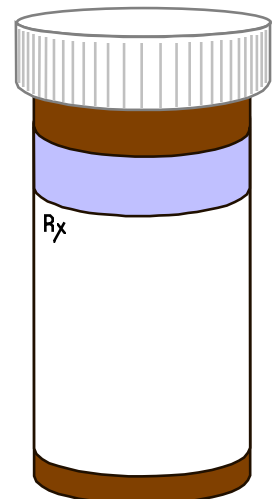
- ❑ Erhöhung der zu verwaltenden Meldungen auf 50
- ❑ Aufbau der Meldungen in Anlehnung an ATP 45

- ❑ Änderungen der Prozesse
 - Task Kommandant, Task Spürer 1 und Task Datenfunk

Zusatz Probenarchivierung

- ❑ Erweiterung des Menütitels "Optionen" um die Menüoptionen "Probennahme" und "Probentabelle"
- ❑ Realisierung einer Bildschirmmaske für die Bodenprobenarchivierung
- ❑ Speicherung der Bodenprobeninformationen beim Wechseln des Probenmagazins

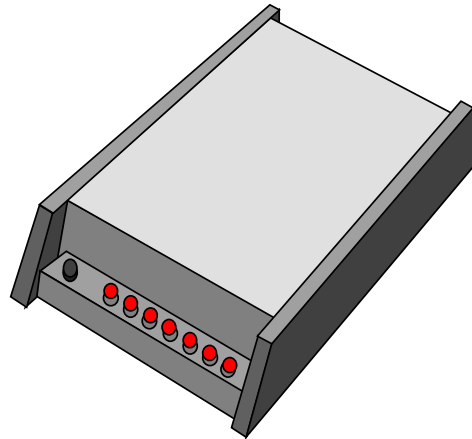
- ❑ Änderungen der Prozesse
 - Task ZM, Task Drucken, Task Laufwerk, Task Kommandant und Task Spürer 1



Zusatz Modemkommunikation

- ❑ Erweiterung der Grundeinstellungen um einen Toggle "Modem- oder Datenfunkkommunikation"
- ❑ Realisierung der Modemkommunikation

- ❑ Änderungen der Prozesse
 - Task ZM, Task Kommandant, Task Spürer 1 und Task Datenfunk



Sonstige Änderungen

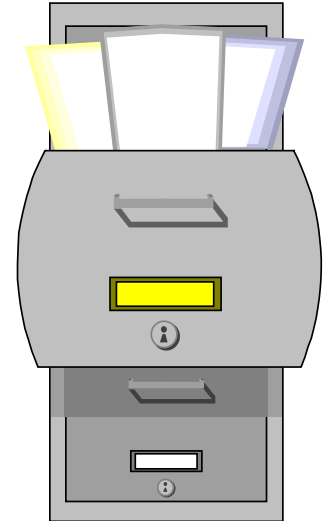
- ❑ Die letzten Eintragungen aus dem Startbildschirm sollen gespeichert werden
- ❑ Realisierung einer "Hardcopy"-Funktion für die Zentralsoftware

- ❑ Änderungen der Prozesse
 - Task ZM, Task Drucken, Task Kommandant und Task Spürer 1

Änderungsumfang

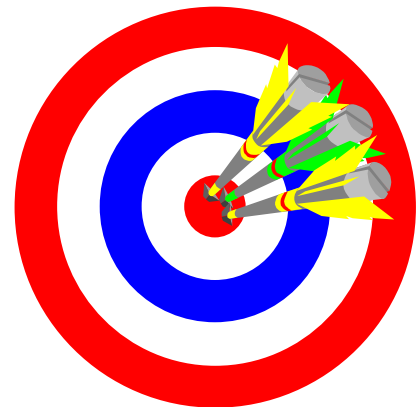
- ❑ Viele Detailänderungen
 - Wenig grundsätzlich Neues
 - Fast alle Prozesse betroffen

- ❑ Viele Änderungen beim Bedienablauf
 - Erfahrungen aus der praktischen Nutzung
 - Viele Dienststellen hatten Verbesserungsvorschläge
 - Gegenbewegung zur großen Disziplin während der Entwicklung



Ziel erreicht? Bei der Software ja!

- ❑ Fast alle Änderungswünsche für die Zentralsoftware im Budgetrahmen realisierbar
- ❑ Softwarearchitektur robust gegen Änderungen
- ❑ Dokumentation nach V-Modell
 - Auch noch nach zwei Jahren "lesbar"
 - Überarbeitung der Dokumente zeitgerecht und kostengünstig
- ❑ Strenges Konfigurationsmanagement mit PCMS
- ❑ Gebrauch von Ada in der SWPÄ verringert Aufwand



Probleme

- ❑ Die Hardware (was sonst ;-)
 - Zu spät
 - Treiber instabil
 - ...
- ❑ Mittlerweile 4 Monate Verspätung bei Endabnahme
- ❑ Erhebliche Mehrkosten durch wiederholte Anläufe zu Systemintegration/Test



10 Erfahrungen mit ADA im Projekt TIGER

A. Ginkel / H. Seidel
Eurocopter Deutschland

Zusammenfassung des Vortrags:

Seit 1989 wird bei EUROCOPTER für die drei Varianten des bilateralen TIGER Programmes operationelle Avionik-Software in ADA für embedded Mehrprozessor Computer entwickelt. Das Systemdesign sieht eine mit MIL-STD 1553B vernetzte, redundant ausgelegte Rechnerarchitektur vor.

Ein entscheidender Anteil der Software realisiert die Benutzerschnittstelle des Piloten zum Waffensystem Hubschrauber; darüber hinaus werden große Datenmengen zur Ansteuerung und Überwachung der Avionik-Komponenten in Echtzeit verarbeitet. Die Software wird gemäß DoD-STD-2167A entwickelt und mit ADA-83 realisiert.

Die Evolution der heute bei EUROCOPTER eingesetzten Werkzeuge, wie z.B. SUN-Workstation, APEX und Testmate, demonstriert die gestiegenen Anforderungen, die an die Entwicklungsumgebung gestellt werden. Das Profil der Softwarespezialisten, das sich aus den unterschiedlichen Profilen an die Entwicklungsmanschaft für embedded Echtzeitanwendungen dieser Größenordnung ableitet, hat entscheidenden Einfluß auf das Entwicklungsergebnis. Es erweist sich darüber hinaus als schwierig, gut ausgebildetes ADA-Personal auf dem Stellenmarkt zu finden.

Die Wechselwirkung zwischen den Zielvorgaben, wie Zeit, Budget, Funktionsumfang, Requirements und Software-Kritikalität sowie den technischen Randbedingungen wie z.B. verfügbarer Speicher und maximale CPU-Belastung und die resultierenden Vorschriften für die Verwendung der Programmiersprache ADA, muß gemäß unserer Erfahrung permanent und mit den richtigen Werkzeugen überwacht werden, um das geforderte Entwicklungsergebnis zu erreichen.

Aufgrund der entwicklungsbedingten häufigen Änderungen, die durch Erprobungsergebnisse, Problem-Reports und nachträgliche Anforderungen in die Software einfließen, werden zweckmäßigerweise auch ADA-Codegeneratoren zur Automatisierung der SW-Änderungen eingesetzt. Wichtig sind spezialisierte Werkzeuge, um ein zuverlässiges Änderungskontrollverfahren, sowie die Umsetzung von einzuarbeitenden Änderungen sicherzustellen.

Insgesamt wurden gute Erfahrungen mit dem Einsatz von ADA gemacht, das sich hervorragend eignet, SW-Projekte dieser Größenordnung zu realisieren und die Wartbarkeit verbessert. ADA ist im Gesamtprozeß der Softwareentwicklung für Avionik Geräte ein wichtiger Garant für die Realisierung eines hohen Qualitätsniveaus, das allerdings nur durch zusätzliche, spezifisch abgestimmte Verfahren erreicht werden kann.

10.1 Avionik SW-Entwicklung bei EUROCOPTER

10.1.1 Von der Mechanik zur Digitaltechnik

EUROCOPTER entwickelt zur Zeit neue Hubschraubersysteme mit digitaler Cockpit-Technologie. Wie bei Flugzeugen der Airbusflotte werden Hubschrauber für den zivilen Einsatz (EC135) für die Heeresfliegertruppen und die Marine (NH90/TIGER) mit sogenannten gläsernen Cockpits ausgestattet, bei denen die Instrumente zur Flugführung, Logistik, Maintenance und Waffenbedienung mit moderner Computertechnologie ausgeführt sind.

Die Hubschrauberfliegertruppen der Bundeswehr werden damit ohne Zwischenschritt aus der Technologie der siebziger Jahre von rein elektromechanischen Systemen, wie sie bisher betrieben werden (z.B. BO105), zu Systemen mit vollständig digitaler Technik übergehen.

10.1.2 Das Projekt TIGER

Frankreich und Deutschland entwickeln gemeinsam den mittelschweren Kampfhubschrauber „TIGER“. Er soll im Jahr 2001 in Dienst gestellt werden. Er wird in drei Versionen für die Rollen Begleitschutz-, Panzerabwehr- und als Unterstützungshubschrauber gebaut werden. Die beiden ersten Versionen werden von der französischen Armee angeschafft, die dritte von der Bundeswehr (bekannt unter dem Kürzel UHT).

Im letzten James Bond-Film „Golden Eye“ spielte ein TIGER-Prototyp die wichtigste Nebenrolle.

10.1.3 ADA als Programmiersprache

Es war von vornherein klar, daß über der langen Nutzungsdauer der Waffensysteme mit Operationeller Software eine intensive Softwarepflege und -änderung (SWPÄ) absehbar war. Die Software mußte also hinsichtlich ihrer Wartbarkeit und der langfristigen Verfügbarkeit von Compilern und Tools, letztlich also der Reduzierung der Life-Cycle-Costs, hohen Ansprüchen genügen. Der Einsatz in fliegenden Waffensystemen erforderte konsequenterweise einen hohen Sicherheitsstandard und eine besonders zuverlässige Software.

Im TIGER Programm war schon früh klar, daß die Aufgabe, das avionische System vollständig in Software zu realisieren, ein relativ umfangreiches Softwaresystem ergeben würde, das nur von einem großen und kompetenten Team von Entwicklern zu implementieren war. (Das System wurde dann noch erheblich größer, als ursprünglich abgeschätzt).

Nahezu gleichzeitig wurde vom Bundesverteidigungsministerium der bekannte „Erlaß über die Verwendung von Programmiersprachen in der Bundeswehr“ herausgegeben, der die Programmiersprache ADA für fliegende Waffensysteme festschrieb.

So entschied man sich auch zu Beginn der Softwareentwicklung der beiden militärischen Programme NH90 und TIGER ADA einzusetzen, nachdem zunächst PASCAL mit Real-Time-Extension vorgesehen war.

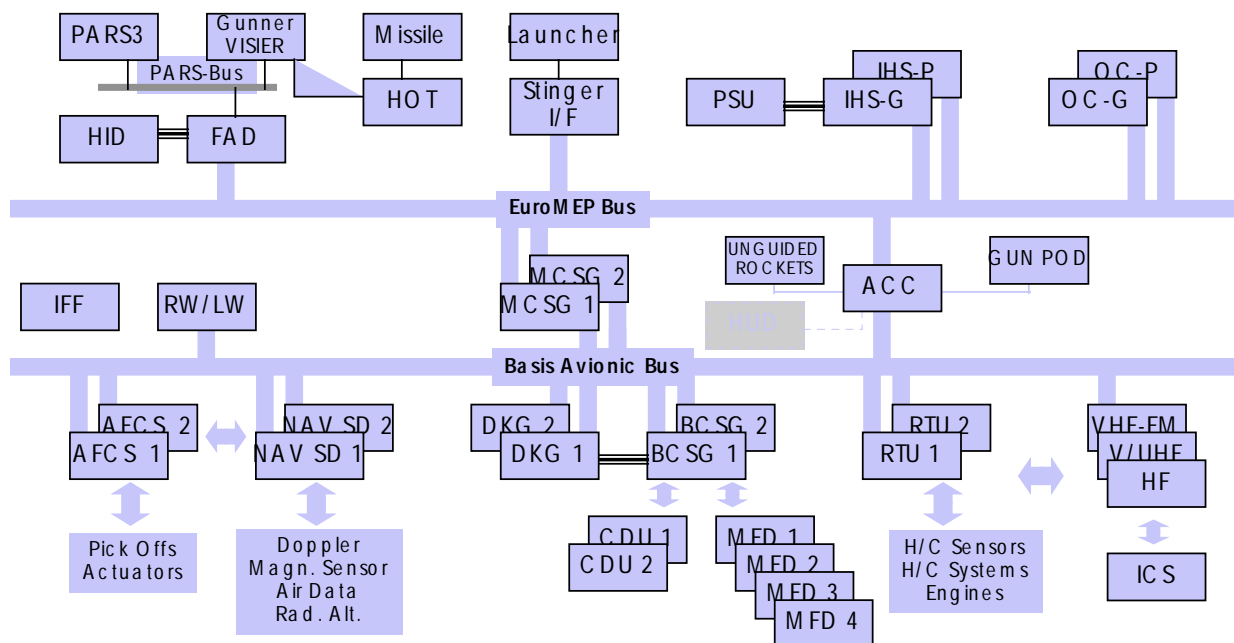
Im Programm TIGER wird nunmehr seit 1989 sowohl die Operationelle Flight Software für die Basis Avionik, als auch das Missionssystem in ADA entwickelt. Der Projektstart für die Software des NH90 begann 1995, wobei Prozesse, Verfahren und Softwaremodule aus dem TIGER Programm wiederverwendet werden.

10.1.4 Avionik-Systemarchitektur des UHT

Das Systemdesign sieht eine mit MIL-STD 1553B vernetzte, redundant ausgelegte Rechnerarchitektur vor. Es teilt sich in die zwei Komponenten Basis Avionik und Missionssystem. Die embedded Computer sind weitestgehend Mehrprozessorsysteme, die mit Motorola 680x0 Mikroprozessoren bestückt sind.

Die **Basis Avionik** stellt alle Informationen zur Verfügung, die ein moderner Kampfhubschrauber benötigt, um sicher und mit hoher Präzision seine Mission zu erfüllen.

Der Hubschrauberpilot wird mit allen zur Flugführung notwendigen Anzeigen digital versorgt: Aircraft Management, Navigation (Strapdown Laser, Doppler, Radarhöhenmesser, Wegpunkt bezogene Funktionen, wie Routen, Direct-To, From-To



erzeugen im Navigationssystem Daten, die dem Autopiloten zur Verfügung gestellt werden), Autopilot (Höhe, Kurs, Vertikale Geschwindigkeit, Doppler Geschwindigkeit, Zielverfolgung auf Sichtlinie, Rückstoßkompensation), Kommunikation, Elektronische Gegenmaßnahmen (Radar/ Laser Warnung, IFF) und taktisches Management (Digitale Karte, Data Link, Speicherung von Video Bildern). Daten von nicht-avionischen Systemen (z.B.: Turbine) werden gesammelt, um diese zu überwachen und sie den Wartungsfunktionen zur Auswertung zur Verfügung zu stellen.

Dem Piloten stehen zur Anzeige der digitalen Informationen im Cockpit Display-Systeme zur Verfügung. (MFD und CDU).

Das **Missionsystem** realisiert die für den Kampfhubschrauber typischen Missions- und Waffenschnittstellen (z.B.: Waffenbedienung, verschiedene Pilot- und Gunner-Sicht Systeme mit TV-Kameras, infrarot Sensoren, Head-In Display).

ADA wird in den als operationell signifikant eingestuften Softwareanteilen neben C, PASCAL und Assembler hauptsächlich eingesetzt. Als Beispiel diene die „Operational Flight Resident Software“ (OFRS) des Zentralcomputers „Bus Controller Symbol Generator“ (BCSG), die komplett in ADA geschrieben wurde (ca. 600.000 Lines of Code).

Rechner mit operationeller Software:		
Rechner	Größe [KLoC]	Sprache:
BCSG - OFRS	600.000	ADA
BCSG - EQSW	50.000	ADA / Assembler / C
MCSG - OFRS	130.000	ADA
MCSG - EQSW	50.000	ADA / Assembler / C
CDU	20.000	C
Operator Controls	10.000	Pascal
RTU	30.000	Assembler
AFCS	70.000	C

10.1.5 Operationelle Flight Software des Zentralcomputers

Die OFRS für den Zentralcomputer (BCSG) wird seit 1989 bei EUROCOPTER mit der Fa. ESG in einer Arbeitsgemeinschaft (ARGE) zusammen mit Unterauftragnehmern (i&, Butler) entwickelt. Zum heutigen Zeitpunkt sind 220 Mannjahre Entwicklung in die Phasen Preliminary Design bis Formal Qualification Test an der Software- Test-Bench (STB) eingeflossen.

Die operationelle Flight Software der Basis Avionik im BCSG realisiert die Benutzerschnittstelle des Piloten zum Waffensystem Hubschrauber über vier „Multi Funktions Displays“ (MFD) und 2 „Control und Display Units“ (CDU). Darüber hinaus werden große Datenmengen zur Ansteuerung und Überwachung (System-Monitoring

und Maintenance) der Avionik-Komponenten wie Sprech- und Datenfunk, Digitale Karte, Navigationssystem, Autopilot sowie Missionsystem über den Milbus (MIL-STD 1553B) in Echtzeit verarbeitet.

Die Entwicklungsphasen sahen zunächst einen Software Prototypen vor (V1). Die hieraus gewonnenen Erfahrungen führten zum heutigen endgültigen Software Design, das in den weiteren Hauptvarianten V2 und V3 nicht mehr verändert wird.

Der BCSG ist ein Multiprozessorsystem, das sich aus 5 Prozessoren zusammensetzt. Um eine exakte Prozessorlast und festgelegte Reaktionszeiten sicherzustellen, werden die Tasks bzw. Prozesse bereits im Software Design auf die Prozessoren verteilt und stehen damit zur Compilezeit fest. Die Equipment-Software, die den Zugriff der Applikation (OFRS) auf die Hardware ermöglicht, stellt zusätzlich Prozeduren wie Mailboxen und Systemqueues zur Verfügung, die eine Kommunikation über Prozessorgrenzen erlaubt.

10.1.6 Der Entwicklungsprozess

Die Software wird gemäß DoD-STD-2167A entwickelt und zur Zeit mit ADA-83 realisiert. Ein Übergang zu ADA-95 ist in der laufenden Entwicklungsphase noch nicht vorgesehen.

In mehreren auch die Entwicklung begleitenden Simulationsphasen werden MMI und prinzipielle operationelle Abläufe in einem speziell hierfür entwickelten Cockpit-Simulator (SimCo) definiert und mit dem Kunden abgestimmt. Auf Silicon Graphics Workstations werden hierzu die Displays in dem virtuell flugfähigem Cockpit simuliert.

In der Phase SW-Requirements Analyse wird Teamwork mit SA/RT eingesetzt. Die SW Requirement Spezifikation enthält derzeit ca. 1000 Requirements für die operationellen Abläufe, die um die Requirements aus den MMI Formatspezifikationen, die sich aus der Simulationsphase ergeben, ergänzt werden. Datenbanken mit Schnittstellen- sowie Logistik-Definitionen dienen als Input für die Codegeneratoren während der späteren Softwareentwicklung.

Für das Preliminary Design wird die Methode HOOD mit dem Design-Tool STOOD eingesetzt. Detailed Design und Code wird mit ADA und speziellen Annotations in den ADA Specs erstellt. Die Einhaltung der Coding Standards werden mit einem in APEX integrierten Tool vor den Design und Code Walkthroughs überprüft.

Nach der Hazardous-Analyse ist die Software als missionskritisch eingestuft und ist nach Level 2A zu testen. Dies bedeutet unter anderem das im Unittest eine Statement- und Decision-Coverage nachzuweisen ist. Im Projekt TIGER wird hierzu Testmate bzw. eigenentwickelten Instrumenter verwendet.

In der abschließenden Software Entwicklungsphase, während des Formal Qualification Testing werden mit In-Circuit-Emulatoren und Loral-Milbus Simulatoren die Requirements auf der Software Testbench (STB) nachgewiesen.

In weiteren Schritten werden die Rechner in den RIGS und auf den Hubschrauber-Prototypen integriert und getestet. Waren die B- und C- Tests erfolgreich kommt es zur Flugfreigabe.

10.2 Erfahrungen mit ADA

10.2.1 Entwicklungsumgebung

Die ersten ADA Compiler wurden 1990 für die OFRS im Projekt TIGER ausgewählt. Zunächst entschied man sich für die Beschaffung eines Rational R1000 Host Systems mit ADA Compiler und integrierter Entwicklungsumgebung. Drei weitere dieser Entwicklungsumgebungen wurden in den nächsten vier Jahren beschafft, um dem gestiegenen SW-Umfang gerecht zu werden. Da es sich um stand alone Maschinen handelte, war man nicht in der Lage, ein gemeinsames Dateisystem zu verwalten. Physikalische Kopien des ADA Quell-Codes waren auf jeder Maschine erforderlich, damit die Entwickler compilieren und testen konnten. Demzufolge war das Konfiguration Management sehr aufwendig, um die Dateikonsistenz zu gewährleisten, insbesondere bei sich häufig ändernden Systemanforderungen.

Beim Zielsystem entschied man sich für den Alsys ADA Compiler (Alsys 5.1.7), der auf Hewlett Packard Workstations zur Verfügung stand. Dies bedeutete, daß der Quell-Code nach abgeschlossener Host-Entwicklung auf die HP-Workstations transferiert werden mußte. Hier wurde der Code cross-compiliert, in den Zielrechner geladen und mit HP-Emulatoren getestet. Dieses zusätzliche Halten der Sourcen auf den HP-Workstations, die auch hier einem Änderungsprozess unterworfen waren, bedeutete ein weiteres Konfigurations-Management Problem.

Mit dem Start der NH90-Entwicklung in 1995 entschied man sich im Projekt TIGER, auf die neue APEX ADA-Entwicklungsumgebung von Rational umzusteigen. Der Hauptunterschied lag darin, daß APEX auf SUN-Workstations mit UNIX-Betriebssystem lauffähig ist. Dies hat den Vorteil, daß der Quellcode innerhalb der einzelnen Developmentviews nur noch einmal vorhanden sein muß.

Auch der Alsys Compiler übersetzt die gleichen Quellen für das Target-System direkt aus der APEX-Umgebung.

Zukünftig wird die Zielhardware für den Tiger und den NH90 auf die Verwendung des Motorola 68040 Mikroprozessors standardisiert. Hierfür ist ein neuer Targetcompiler 5.5.8 von Aonix (früher Alsys) im Zulauf. Die neue Hardware/Compiler Konfiguration muß insgesamt um den Faktor 5 schneller sein, als die aktuelle Konfiguration, um die Lauffähigkeit der OFRS auf einem statt auf zwei Prozessoren sicherzustellen und um genügend Reserve (Memory/CPU) für die Serie zur Verfügung zu stellen.

10.2.2 ADA Personal

Das Profil der Softwarespezialisten, das sich aus den unterschiedlichen Anforderungen an die Entwicklungsmannschaft für embedded Echtzeitanwendungen dieser Größenordnung ableitet, hat entscheidenden Einfluß auf das Entwicklungsergebnis. Es erweist sich als schwierig, gut ausgebildetes ADA-Personal auf dem Stellenmarkt zu finden.

In den Ausbildungsplänen der Informatik-Fakultäten deutscher Universitäten und Hochschulen wird die Sprache ADA nur selten angeboten. Selbst der Fachbereich Informatik der Universität der Bundeswehr bietet diese, gemäß Rahmenerlaß der Bundeswehr vorgeschriebene Sprache nicht an.

EUROCOPTER hat gute Erfahrungen mit Studenten und Absolventen der Staffordshire University/ England gewonnen. Hier liegt eine gute Mischung aus Theorie und Praxis im Bereich ADA vor. Die Entwickler sind zudem relativ jung, wenn sie mit der Ausbildung fertig sind.

Da es sich um multinationale Programme handelt, setzt sich auch das ADA Know How international zusammen: Entwickler aus Deutschland, dem europäischen Ausland und den USA bilden die zeitweise bis zu 40 Personen starken Softwareteams je Projekt.

Die Softwareabteilung bei EUROCOPTER verfügt über ein gutes Mischung aus jungen, gut ausgebildeten Entwicklern und erfahrenen ADA Experten mit ausgezeichneten Kenntnissen im Bereich SW-Engineering. Fremdfirmen und Unterauftragnehmer runden das Bild ab. Geringe Fluktuation sorgen für ein gleichmäßig hohes Qualitätsniveau im Projekt.

10.2.3 Einfluß der Entwicklungsprioritäten auf das Ergebnis

In Barry W. Boehms Buch „Software Engineering Economics“ [BOE81] wird auch das Thema der konfliktären Entwicklungsziele behandelt. Als Beispiel dafür erwähnt er das *Weinberg Experiment*. Dabei wurden fünf Entwicklungsteams die gleiche Aufgabe aber mit unterschiedlichen Entwicklungsprioritäten gestellt. Nach der Fertigstellung wurden die Ergebnisse anhand der fünf Entwicklungsprioritäten bewertet. Das Resultat beim Vergleich der Entwicklungsergebnisse zeigte folgendes:

- jedes Team landete mit ihrem vorgegebenen Entwicklungsziel auf Platz eins (ein Team auf dem zweiten)
- keines der Teams bot gute Leistungen bei allen Entwicklungszielen.

Vor diesem Hintergrund ist auch das BCSG Team untersucht worden, und es zeigten sich ähnliche Effekte. Durch den hohen Zeitdruck zu Beginn des Projektes war der Speicherverbrauch, die CPU Belastung und die Fehlerhäufigkeit höher als ursprünglich geplant. Nach der Optimierung des Entwicklungsprozesses und der damit erreichten Steigerung der Qualität der Software konnten die Nachteile bei der CPU Belastung und dem Speicherverbrauch entscheidend verbessert werden.

10.2.4 Umsetzung der Entwicklungsziele mit ADA

Ein sehr wichtiges Ziel der BCSG Softwareentwicklung im Projekt TIGER ist ein zuverlässiges System zu erhalten, das sich an seinen Schnittstellen gegenüber unerwarteten Einflüssen äußerst robust verhält. Es wurden folgende Eigenschaften als dazu notwendig identifiziert:

- Vorhersagbarkeit des Zeitverhaltens
- Verhindern der zeitlichen Degradation des Zeitverhaltens
- Verhindern der situationsbedingten Degradation

10.2.4.1 *Vorhersagbares Zeitverhalten mit Rate Monotonic Scheduling (RMS)*

Das Software Design basiert auf der Rate Monotonic Theorie (RMS) (mehrfach veröffentlicht, u.a.: [DDC-1, SIL89, SHA90]) und sieht nur harmonische und periodische nicht-wechselwirksame Tasks vor, die nicht über das ADA-Rendezvous kommunizieren.

Bei wechselwirkenden Tasks, die den Rendez-Vous Mechanismus nutzen, ergibt sich die Gefahr der Task-Blockade, wenn gleichzeitig mehrere Tasks auf geschützte physikalische Ressourcen zugreifen. Dadurch wird es sehr schwierig ein deterministisches System zu erhalten.

Für ein statisches Ensemble von nicht-wechsel-wirkenden Tasks läßt sich zeigen, daß alle Tasks ihre deadlines erreichen, wenn die Taskprioritäten monoton mit der geforderten Frequenz der Tasks steigen und die Prozessorlast ein bestimmtes Limit nicht überschreitet. Dieser Ansatz definiert eine einfache Task Struktur zusammen mit, in definierten Grenzen, vorhersagbarem Zeitverhalten.

Um ein statisches Ensemble nicht-wechselwirkender Tasks zu erhalten, ist die Benutzung von ADA Tasking eingeschränkt auf die Definition der statischen Tasks.

10.2.4.2 *Einschränkung des Gebrauchs von dynamischen Objekten*

Die uneingeschränkte Benutzung von dynamischen Objekten, d.h. Objekte, die während der Laufzeit eines Programmes erzeugt und wieder vernichtet werden, ist problematisch, da der Heap fragmentieren kann, was sowohl zu einem Heap-Überlauf als auch zu einer erhöhten Prozessorlast führen kann, welches wiederum die Vorhersagbarkeit des Programmverhaltens einschränkt.

Bei dem im Projekt TIGER verwendeten Cross-Compilern wird Heap-Speicher sowohl bei der Benutzung des Allokationsoperators „new“ als auch bei der Deklaration von lokalen Objekten, deren Größe eine gewisse Grenze (1024 Bytes) überschreitet, allokiert. Eine der im Projekt gültigen Programmierrichtlinien ist, daß Heap-Allokationen nur während der Initialisierungsphase des ADA-Programmes zulässig sind, die Heap-Aufteilung muß vom Anfang des Programmstartes an statisch sein.

10.2.4.3 *Einschränkung der Benutzung von Exceptions*

Die Bearbeitung von Exceptions ist in Echtzeitsystemen und insbesondere bei den in unserem Projekt verwendeten Cross-Compilern sehr zeitaufwendig. Der Aufwand steigt monoton mit der Anzahl der Hierarchieebenen, über die eine Exception weitergegeben wird, bis sie endgültig abgearbeitet ist. Auftretende Exceptions, die, wie der Name sagt, an das Auftreten gewisser Ausnahmesituationen gebunden sein sollen, können also eine Degradation des Zeitverhaltens bis hin zu einem Systemzusammenbruch, bewirken. Deshalb ist die Benutzung und Behandlung von Exceptions in unserem Projekt (trotz eingeschalteten RunTime Checks) unter genaue Kontrolle gestellt. Defensives Programmieren ist deshalb Vorschrift (z.B. Polstellen wie Division durch 0 dürfen nicht durch Exceptions abgefangen werden, sondern nur durch explizite Abfrage). Dies führt zu einem höheren Grad der Robustheit der OFRS.

Exceptions treten deshalb nur aus zwei Gründen auf:

- Bugs
- Inkorrekte Daten an Schnittstellen zum Computer

Bugs kann man niemals verhindern („Es ist grundsätzlich immer noch ein Fehler vorhanden...“), aber ihre Auswirkungen müssen weitestgehend kontrollierbar sein. Deshalb wurden Orte in der Aufrufhierarchie des Programmes bestimmt, an denen bestimmte Exceptions abgearbeitet werden müssen. Die bisherigen Testphasen haben das Vorgehen als sehr wirksam bestätigt.

10.2.4.4 Coding Standards

Die oben genannten Spracheinschränkungen sind zusammen mit insgesamt ca. 200 weiteren Regeln, die die Qualität der entwickelten Software sicherstellen sollen, in den Projekt Coding Standards festgelegt. Diese sind in drei Klassen eingeteilt:

- **„Mandatory“**: Diese Regeln sind unbedingt bindend für alle Entwickler im Projekt.
- **„Required“**: Diese Regeln sind bindend, können aber in Absprache mit dem Projektleiter bzw. der Qualitätssicherung umgangen werden.
- **„Recommended“**: Diese Regeln sind bindende Hinweise, die für ein einheitliches Aussehen des Programmcodes sorgen sollen.

Die Coding Standards werden mit einem in APEX eingebundenen Programm, das mit den projektspezifischen Code Regeln instrumentiert wird, überprüft. Die Ergebnisse dienen als Eingabe für die regelmäßigen Design- und Code- Walkthroughs.

Diese im Software Entwicklungs Plan vorgeschriebenen Walkthroughs erweisen sich als wichtiges qualitätssicherndes Instrument während der Entwicklungsphase: sie dienen der Überprüfung, ob die Coding-Standards eingehalten werden, und der Verifikation, daß Entwicklungsziele wie Anforderungen an die Performance, Memory, Zuverlässigkeit und Sicherheit erreicht werden.

10.2.5 Einsatz von Tools zur Codegenerierung

MMI-Bereiche sowie Interfaceteile, die datenorientiert sind und einer hohen Änderungsrate unterliegen, werden mit Codegeneratoren erstellt.

Generatoren stellen insgesamt einen entscheidenden Faktor zur Produktivitäts- und Qualitätssteigerung in den Projekten TIGER und NH90 dar. Vorgegebene Termine wären sonst bei den auftretenden Änderungsraten nicht haltbar.

Für die Entwicklung der Codegeneratoren werden sowohl ADA selbst, als auch state of the art Tools, wie Perl verwendet. Perl eignet sich als Implementierungssprache für Programmgeneratoren besser als ADA, dank seiner bereits vordefinierten, mächtigen Standardoperationen zur Verarbeitung von Zeichenketten. Diese Operationen müssten sonst erst in ADA aufwendig implementiert werden.

Im Projekt TIGER sind 2 Klassen von Generatoren zur Generierung von ADA Code im Einsatz:

- Spezielle Generatoren (Milbus Generator)
- Allgemeine wiederverwendbare Generatoren (z.B.: CDU Generatoren)

Die allgemeinen, wiederverwendbaren Generatoren erzeugen im wesentlichen vollständige ADA Programm-einheiten, die konstante Relationen darstellen (Alphabete, Abbildungen mit diskreten Definitionsbereichen und beliebigen Wertebereichen im Rahmen des ADA Typenkonzepts). Generiert werden unter anderem jeweils umfangreiche Alphabete (Namensräume), Automatentafeln, Folgen von Interpreter-Instruktionen, Entscheidungstabellen, Tabellen zur Definition von MMI Maskenformaten und Inhalten. Als Eingaben werden Datenbestände aus einer Spezifikationsdatenbank und von Hand erstellte Tabellen benutzt.

Die eingeschlagene Richtung, wiederverwendbare Generatoren zur Generierung von Abstraktionsklassen zu benutzen, ergänzt und erweitert ADA's Möglichkeiten der statischen Programm- Parametrisierung mittels „Generischer Einheiten“ (generics).

10.2.6 Änderungs-Management

Ein Garant für den Erfolg des Projektes ist nicht nur der verwendete Compiler, die Entwicklungsumgebung und erfahrenes Personal, sondern auch ganz entscheidend der verwendete Entwicklungsprozess und der Umgang mit ständig auftretenden permanenten Änderungsanforderungen. Gemeint sind dabei die Änderungen der Spezifikation, Defect-Reports der Avionik Geräte und hierdurch erforderliche „Workarounds“, SW-Änderungen zur Fehlerbehebung und Verbesserungen.

Die in den Entwicklungstools eingebauten Methoden (Check in/ Check out, SCCS) sind hierzu nicht ausreichend, man benötigt darüber hinaus ausgereifte Tools, die eine Änderungskontrolle ermöglichen. Das bei EUROCOPTER eingesetzte Werkzeug, mit dem dieser Änderungsprozess von den Spezifikationen bis zum Flugversuch kontrolliert wird, wurde im Laufe der Entwicklung selbst entwickelt. Es basiert auf einem Datenbanksystem von ORACLE, um Änderungen und die unterschiedlichen Zustände von Problemreports, Engineering Change Requests und Software Change Notes zu erfassen. Mit einer monatlichen Statistik wird das Management über den aktuellen Bearbeitungsstand informiert. Dies hat zu einer erhöhten Transparenz der Vorgänge und einem vertrauensvollen Umgang miteinander geführt.

10.3 Zusammenfassung der Erfahrungen mit ADA im Projekt TIGER

Insgesamt wurden gute Erfahrungen mit dem Einsatz von ADA gemacht. Die Sprache eignet sich als Standardprogrammiersprache gut, um komplexe Echtzeitprojekte dieser Größenordnung, die eine lange Laufzeit haben und an denen viele Entwickler arbeiten, zu realisieren.

Der Einsatz von ADA sorgt für eine ausgezeichnete Wartbarkeit nicht nur für zukünftige Wartungsaktivitäten während der Software Pflege und Änderung (SWPÄ) beim Kunden, sondern auch bereits während der Entwicklung.

Die klare und übersichtliche Sprache, das strenge Typenkonzept, das Prinzip des Exception-Handlings und das integrierte Tasking stellen sich als Vorteile gegenüber anderen vergleichbaren Sprachen heraus.

ADA ist im Gesamtprozeß der Softwareentwicklung für Avionik Geräte ein wichtiger Garant für die Realisierung eines hohen Qualitätsniveaus.

Dennoch: der Einsatz von ADA im embedded Echtzeitumfeld mit hohen Anforderungen an Vorhersagbarkeit und Stabilität bedingt Einschränkungen des Sprachumfangs, die in unserem Projekt in ca. 200 Coding-Standards und in Software-Guidelines dokumentiert sind.

ADA als HOL stellt den Anspruch eine sichere und für militärische Anwendungen besonders geeignete, plattformunabhängige Sprache zu sein. Dennoch sind eine Reihe von Fehlern möglich, die erst zur Laufzeit erkannt werden. Als Beispiel diene das strong typing Konzept: fehlerhafte Zuweisungen von verschiedenen Subtypes eines Datentypes werden nicht zur Compilezeit erkannt, sondern erzeugen Laufzeitfehler (Exceptions). Besondere Aufmerksamkeit muß während der Code Walkthroughs auf bestimmte Konstrukte (z.B.: address clauses) gelegt werden, die versteckte Zeigerstrukturen enthalten.

Vorteile von ADA:

- Klare, übersichtliche Sprache (Lesbarkeit, Wartbarkeit, geeignet für das Groß-Projekt)
- Validation der Compiler (Verfügbarkeit, Zuverlässigkeit)
- strenges Typkonzept, Modularität (Information Hiding)
- Exception-Handling
- integriertes Tasking

Einschränkungen:

- Wechselwirkende Tasks mit Task Rendez-Vous sind für komplexe, kritische Realzeitprojekte nicht geeignet. (Nachweis für Zeitverhalten schwierig)
- Fehlerträchtige ADA Konstrukte sind möglich
- Typkonzept verspricht Sicherheit, hat aber Lücken
- Laufzeitintensives Exception-Handling nur eingeschränkt nutzbar

Nachteile:

- kaum COTS verfügbar
- kaum ADA-Experten verfügbar

Fazit:

Im Projekt TIGER wurde mit ADA gute Erfahrungen gemacht. Für die Gesamtqualität des Produktes sind allerdings neben ADA auch Erfahrungen und professionelle Kenntnisse in den übrigen Bereichen des SW-Engineering von entscheidender Bedeutung: Software Entwicklungsmethoden, Anbindung und Ausrichtung der Softwareentwicklungsstrategie auf die zugrunde liegende Hardware und ein konsequentes Projekt- und Änderungs-Management, um den Entwicklungsprozess zu kontrollieren.

10.4 Literatur

- [BOE81] B. Boehm, „Software Engineering Economics“, Prentice-Hall, 1981
- [DDC-1] Lee Silverthorn, DDC, „Managing Large Aerospace Software Projects in ADA using RMS“
- [SIL89] L. Silverthorn, „Rate-Monotonic Scheduling Ensures Tasks Meet Deadlines“, EDN, Oct.26, 1989
- [SHA90] L. Sha and J. Goodenough, „Real-Time Scheduling Theory and Ada“, IEEE Computer, April 1990, pp53-62

11 Anwendung von Ada in Satellitensystemen

O. Kellogg
Abteilung RST13, Informatik
Daimler-Benz Aerospace AG
Dornier Satellitensysteme GmbH, Ottobrunn

11.1 Einleitung

Dornier tritt als Anbieter von Komplettlösungen am Satellitenmarkt auf. Dies bedeutet, daß sowohl der Satellit selber, als auch die nötige Testumgebung zum Austesten des Satelliten vor dem Start am Boden, sowie die Boden-Kontrollstation produziert werden. In beiden Anwendungsumgebungen (Bord und Boden) wird am Standort Ottobrunn seit 1988 Ada eingesetzt.

Es wird zunächst eine Übersicht über die Systeme, in denen wir Ada verwenden, gegeben. Dann werden konkret einige Erfahrungen dargestellt, die wir in der Ada-Programmierung gesammelt haben.

11.2 Die Systeme

11.2.1 Das Bordsystem

Heutige Satelliten bewältigen eine Vielzahl von Aufgaben größtenteils selbständig. Sie halten die richtige Lage, d.h. Lageabweichungen müssen autonom korrigiert werden. Sie können zudem von der Bodenstation ferngesteuert werden, d.h. erhalten per Radiowellen sogenannte Telekommandos, welche bordseitige Aktionen auslösen können (z.B. Feuern einer Düse zur Lageänderung). Weiterhin sammeln sie autonom verschiedene Daten über ihren Zustand, die als Telemetrie zur Bodenstation gesendet werden. Um dies alles zu bewältigen, werden ein oder mehrere Digitalrechner eingesetzt.

Die Lageregelung bildet einen wesentlichen Teil der Onboard-Software. Die hierfür verantwortlichen Lageregelungsspezialisten schreiben ihre Algorithmen in Ada und testen diese in einer Hostrechner-basierten Simulationsumgebung aus. Sie liefern ihre AOCS (Attitude and Orbit Control Software) an die Systemsoftware-Abteilung ab, welche die Software fürs Zielsystem kompiliert (Crosscompiler), auf den Bordrechner bringt, und im Gesamtsystem testet.

Der Satellit ist ständig kosmischer Strahlung ausgesetzt, durch welche normale Halbleiterbauelemente zerstört würden. Die Satelliten-Elektronik muß deshalb strahlungsfest sein. Die Auswahl der CPU für den Bordrechner ist hierdurch stark eingeschränkt - nur wenige Prozessortypen sind auch in strahlungsfester Variante erhältlich. Bislang werden 16-Bit-Prozessoren mit MIL-STD-1750A Befehlssatz verwendet. Raumfahrttaugliche RAM-Bausteine sind sehr teuer (wie die CPU selbst) und stehen nur

in kleinen Speichergrößen zur Verfügung, der Bordrechner muß daher mit 128 bis 384 Kilobyte auskommen.

Für die Bordsoftware bedeutet dies, daß sehr genau auf den Speicherbedarf des übersetzten Codes geachtet werden muß.

Es wird nicht der volle Ada-Sprachumfang, sondern die Safe-Ada Untermenge eingesetzt (insbes. keine dynamische Speicher-Allokierung). Es wurde ermittelt, daß der Crosscompiler generische Pakete nicht sonderlich effizient übersetzt, daher mußte auf deren Verwendung verzichtet werden.

Anstatt des Ada-Tasking wird ein In-House entwickeltes Echtzeit / Multitasking-Betriebssystem verwendet. Dies hat folgende Gründe:

1. Es wird ein absolut deterministisches Tasking-Verhalten verlangt und man wollte sich nicht auf die Eigenheiten eines bestimmten Compilers verlassen.
1. Das Tasking-Laufzeitsystem des zu Beginn eingesetzten Ada83-Crosscompilers war zu groß. (Dieser Punkt bedarf allerdings der Reevaluierung).

11.2.2 Das Bodensystem

Wie bereits unter S1. erwähnt, nimmt die Lageregelung (Attitude and Orbit Control System = AOCS) den wesentlichen Teil der Bordsoftware in Anspruch.

Der Satellit verfügt über verschiedene Sensoren zur Lagebestimmung, wie Erd- und Sonnensensor (temperaturbasiert), Erd-Magnetfeldsensor, Drehgeschwindigkeitsmesser (Gyrometer), und andere. Weiterhin ist er mit Aktuatoren (Düsen, Schwungrädern od. a.) zur Lagekorrektur ausgestattet. Die Aufgabe der Lageregelung ist es, aus den Meßwerten der Sensoren die Lage des Satelliten zu ermitteln, diese mit der berechneten Soll-Lage zu vergleichen, und bei Abweichungen z.B. die entsprechenden Düsen zum Feuern zu veranlassen.

Von der bordseitigen zur Lageregelung benutzten Hardware, also dem Bordrechner und den verschiedenen Sensoren und Aktuatoren, stehen während der SW-Entwicklungsphase nur wenige Exemplare (wenn überhaupt) zur Verfügung, und die Debugging-Möglichkeiten auf diesen sind meist sehr eingeschränkt. Aus diesem Grund erfolgen verschiedene per Software realisierte Simulationen.

Die Regelungsingenieure, welche die AOCS-Software erstellen und testen, arbeiten in ihrer gewohnten bequemen Hostrechner-Softwareumgebung. Beim Entwurf der AOCS wurde darauf geachtet, daß die Lageregelungssoftware durch Austausch weniger Ada-Pakete zwischen Entwicklungs- und Zielsystem portabel ist.

Es wurde weiterhin darauf geachtet, daß Daten, die in irgendeiner Weise extern sichtbar sind, auf beiden Systemen gleich repräsentiert sind. Ausgenommen von dieser Gleichheit der Datenrepräsentation sind bislang die Fließkommazahlen, da deren Formate bei Host- und Target-Computer verschieden sind. Das Bestreben ist, zwischen Entwicklungs- und Zielrechner die gleichen Bedingungen für das Auftreten von Overflow und sonstigen Ausnahmeständen zu schaffen.

Die beim Übergang vom Entwicklungs- zum Zielrechner auszutauschenden Pakete sind:

1. Paket, in dem Basistypen wie 16-Bit-Integer, 32-Bit-Integer etc. sowie Operationen zu deren Bitmanipulation in compilerunabhängiger Form zur Verfügung gestellt werden
2. Paket für mathematische Funktionen
3. Paket für Aufrufchnittstelle zum Echtzeit-Betriebssystem (beim Entwicklungsrechner besteht dessen Body größtenteils aus Stubs).

Das Austesten der AOCS auf dem Hostrechner bedeutet, daß auch die Satelliten-Umwelt (Satellitenbewegung, Störgrößen, Erde, Sonne, Mond etc.) simuliert werden muß. Es werden also die Meßwerte, welche in der Wirklichkeit von den Sensoren des Satelliten gemessen werden, durch Simulationsmodelle künstlich erzeugt und in die Lageregelung eingespeist.

Während die Lageregelung (AOCS) in Ada geschrieben ist, ist die Umweltsimulation ein über viele Jahre weiterentwickeltes Fortran77-Programm.

Bei der Kopplung zwischen Lageregelung und Simulation beschritten die Projekte verschiedene Wege. Bei einem Projekt sind diese als getrennte Prozesse auf dem Hostrechner realisiert, die über "Shared Memory" Daten austauschen: Der AOCS-Prozeß holt sich die vom Simulationsprozeß erzeugten Sensor-Meßwerte aus dem gemeinsamen Datenbereich. Beim Wechsel von der Simulation auf die echte Satelliten-Hardware wird der Umweg über Shared Memory durch die echte Sensormessung ersetzt.

Ein anderes Projekt koppelte die beiden Software-Teile direkt durch Anwendung von Pragmen für Daten-Export/Import. Beide Wege erwiesen sich als machbar. Der etwas umständlichere Weg über Shared Memory ermöglicht, daß sich unabhängige Analyse- oder Auswertungsprogramme zur Laufzeit in den gemeinsamen Speicher einklinken können.

Wesentliches Design-Merkmal der Software ist, daß die Simulationen problemlos durch entsprechende echte Hardware ersetzt werden können. Die Simulationsmodelle können bereits lange vor Zur-Verfügungstehen der realen Sensorhardware erstellt werden, und zwar anhand der Datenblätter/Spezifikationen der Hersteller. Diese Möglichkeit, den realen Sensor anstelle der Simulation einzusetzen, ist sehr wichtig. Es muß letztlich immer mit realer Hardware getestet werden -- und es kam bereits vor, daß die Herstellerangaben zur Hardware inkorrekt waren! In dem Fall muß das Software-Modell entsprechend abgeändert werden. Großer Vorteil ist dennoch, bereits die Lageregelung im "geschlossenen Kreis" (Closed Loop) zwischen Satellit und Umwelt entwickeln und austesten zu können, lange bevor die reale Hardware zur Verfügung steht.

11.3 Erfahrungen bei der Ada-Programmierung -- "Lessons Learned"

Die einzelnen Programmierhinweise sind mit deren Relevanz gekennzeichnet (Bord- und/oder Boden-Systemprogrammierung).

11.3.1 Bord/Boden: Zuviel der Typen

a)

Beim ersten Ada-Projekt, welches zugleich als "Ada-Lernprojekt" diente, gab es die Tendenz, selbst dann Untertypen zu deklarieren, wenn nur ein oder zwei Variablen des Untertyps ausgeprägt wurden, etwa:

```
-- In einer Paketspezifikation:

subtype Thrust_Level_Type is Short_Int;

subtype Magnetometer_Value_Type is Short_Int;

-- In einem anderen Paket:

Thrust_Level : Thrust_Level_Type;

Magnetometer_Value : Magnetometer_Value_Type;
```

In diesem kurzen Beispiel machen die zusätzlichen Typdeklarationen das Programm zwar besser lesbar. Wenn es allerdings sehr viele solcher Untertypdeklarationen gibt ("Typen-Dschungel"), wird die Les- und Wartbarkeit des Programm eher beeinträchtigt als gefördert: Es muß immer im deklarierenden Paket nachgeschaut werden, nur um herauszufinden, daß es sich letztlich um eine einfache Integer-Variable handelt.

b)

Bisweilen wurde das Konzept der Gleichheit der Datenrepräsentation zwischen Entwicklungs- und Zielrechner übertrieben. So hat zum Beispiel die Deklaration

```
type BOO is new BOOLEAN;

for BOO'size use 16;
```

kaum praktischen Vorteil gegenüber dem vordefinierten Ada-Typ BOOLEAN. Diese Neuvereinbarung ist zwar nicht direkt schädlich, nur überflüssig, und sie wurde hauptsächlich deswegen aus dem Code genommen, weil der Crosscompiler mit dem abgeleiteten Typ Probleme in logischen Vergleichen hatte.

c)

Wirklich nachteilig war der einstige Ansatz, die Integer-Größen der Telemetrievariablen auch bodenseitig vollständig "durchzuziehen". Mit anderen Worten, wenn es in der Telemetrie Variablen der Typen INTEGER_8, INTEGER_16, INTEGER_32,

UNSIGNED_8, etc. gab, dann zogen sich auch exakt diese Typen durch die gesamte bodenseitige Verarbeitung der Telemetrievariablen. Das hatte den entspr. Bodencode stark aufgebläht, führte z. B. zu großen typabhängigen CASE-Statements, um auch nur einfachste Arithmetik mit solchen Variablen zu machen. Der richtige Weg war natürlich, die verschiedenen Variablen in der bodenseitigen Eingangsverarbeitung nur einmal auf eine oder zwei Standardgrößen zu bringen (z.B. nur INTEGER_32, maximal noch UNSIGNED_32), und dann mit diesen vereinheitlichten Größen weiterzuarbeiten.

11.3.2 Boden: Zeiger für Kurzformen statt "rename"

Eine der grundsätzlichen Programmier-Erfahrungen die wir machten, lautet: Wähle prägnante, aber aussagekräftige Variablennamen. Zu kurze Namen sind schwer dechiffrierbar, aber auch das andere Extrem schadet der Lesbarkeit, insbesondere wenn überlange Variablennamen in verschachtelten Records verwendet werden. Es gibt Fälle, wo eine einzige Variablenbenennung über mehrere Zeilen geht. Da muß die Frage erlaubt sein: Ist das wirklich nötig?

Auch bei Beachtung dieser Regel ist es oft praktisch, tief verschachtelte oder sonstwie längliche Datenreferenzen in einem lokalen Block (etwa in einer Schleife) mit einer Kurzschreibweise zu benennen. Insbesondere bei Programmierern mit C-Background, wo dies üblicherweise per Zeiger erledigt wird, konnten solche Kurzformen etwa so aussehen:

```

declare
    -- Es sei HK_DEF.HK_DESCRIPTOR der letztlich interessierende Typ
    type HK_Access is access HK_Def.HK_Descriptor;

    function Reference is new Unchecked_Conversion (System.Address,
                                                    HK_Access);

    HK : HK_Access := Reference
        (HK_Def.HK_Data.HK(Descr.HK_Index)'address);

begin
    -- ...

    -- ... Arbeiten mit HK statt HK_Def.HK_Data.HK(Descr.HK_Index)

    -- ...

end;
```

Unnötig kompliziert und zudem (durch die `Unchecked_Conversion`) nicht immer portabel und laufzeitoptimal. Das war natürlich vor der "Entdeckung" der `renames` Anweisung:

```
declare
    HK : HK_Def.HK_Descriptor renames
        HK_Def.HK_Data.HK(Descr.HK_Index);
begin
    [...]
end;
```

11.3.3 Boden: Zeigertypen, nur um Variablen von diskriminierten Records dynamisch auszuprägen

Die Diskriminanten von Records mit variantem Teil dürfen bekanntlich nicht so einfach verändert werden. Will man einem solchen Record mitten im Code eine neue Diskriminante zuweisen, darf die neue Diskriminante erstens keine Variable sein, und zweitens müssen alle Komponenten des der neuen Diskriminanten entsprechenden Variantenteils schon gleich bei der Zuweisung aufgeführt sein. Das ist oft lästig, zumal wenn die konkreten Werte der einzelnen Komponenten am Punkt der Neuausprägung noch nicht alle bekannt sind. Die Verwendung von Zeigern und Ausprägung per "new" bringen zwar die gesuchte Abhilfe, doch ist dies nicht einzusehen, wenn deren einziger Existenzgrund in solcherlei "Workaround" besteht.

Die bessere Lösung dieses Problems erfordert nur einmalig einen Mehraufwand, nämlich im Schreiben einer Funktion, die den Diskriminantentyp als Parameter hat und ein entsprechend ausgeprägtes Record (mit mit Default- oder Nullwerten angefüllten Komponenten) als Resultat liefert. Hier ein auszugsweises Beispiel:

```
type Operation_Element_Kind is (op, bif, func);
type Operation_Element (Kind : Operation_Element_Kind := op) is
    record
        Common_Var_1, Common_Var_2 : Natural;
    case Kind is
        when op =>
            Operation : Operation_Type;
        when bif =>
            Builtin : Intrinsic_Function;
```

```
when func =>
    Func_Index : Natural;
    Param_Array : Expression_Array;
end case;
end record;

-- Baue dynamisch ein "Skelett" für OPERATION_ELEMENT:
function Make_Operation_Element (Kind : Operation_Element_Kind;
                                Common_Val_1 : Natural := 0;
                                Common_Val_2 : Natural := 0)
    return Operation_Element is
begin
    case Kind is
        when op =>
            return Operation_Element'(Kind => op,
                                      Common_Var_1 => Common_Val_1,
                                      Common_Var_2 => Common_Val_2,
                                      Operation => nop);
        when bif =>
            return Operation_Element'(Kind => bif,
                                      Common_Var_1 => Common_Val_1,
                                      Common_Var_2 => Common_Val_2,
                                      Builtin => dummy);
    end case;
end function;
```

```
when func =>
    return Operation_Element'(Kind => func,
                               Common_Var_1 => Common_Val_1,
                               Common_Var_2 => Common_Val_2,
                               Func_Index => 0,
                               Param_Array => (others => null));

end case;

end Make_Operation_Element;
```

Anwendungsbeispiel:

Ausprägen mit einer Variablen als Diskriminante und bevor alle Komponentenwerte bekannt sind. Das kann nützlich sein, wenn einige Komponenten sofort zugewiesen werden sollen, um auf sie schon lesend zugreifen zu können, während andere Komponenten erst noch berechnet werden müssen. Der Vorteil der Initialisierungsfunktion wird erst deutlich, wenn der Record sehr viele Komponenten enthält: Der Zwang zur Auflistung aller Initialisierungswerte bei jeder Ausprägung entfällt.

```
Elem_Type : Operation_Element_Kind;

... weitere Deklarationen, Code zur Berechnung von "Elem_Type"
...

declare
    Op_Rec : Operation_Element := Make_Operation_Element (Elem_Type);
begin
    -- ... gemeinsam benutzter Code zur Berechnung der Common_Var_...
    -- Berechnungen für den Fall (Elem_Type = func) :
    if Elem_Type = func then
        declare
            Function_Index : Natural;
            Parameters : Expression_Array;
```

```

begin
    -- ... Berechnung von Function_Index ...

    Op_Rec.Func_Index := Function_Index;

    -- ... Berechnung von Parameters ...

    Op_Rec.Param_Array := Parameters;

end;

-- ... sonstiger Code, z.B. weitere Fallunterscheidungen ...

end if;

-- ... sonstiger Code ...

end;
```

11.3.4 Bord: Portierbarkeit von Repräsentationsklauseln (MIL-STD-1750 spezifisch)

Der militärische Standard des verwendeten Mikroprozessors schreibt eine Namensgebung der Bits in einem Wort vor, welche invers zum sonst üblichen verläuft. Das höchstwertige, am meisten links stehende Bit wird mit "Bit 0" bezeichnet, "Bit 15" ist also das niederwertigste Bit eines 16-Bit-Wortes. Der Ada-Crosscompiler hält sich bei der Zählung der Bitpositionen in Record-Repräsentationsklauseln an den MIL-STD. Um aber die gleichen Repräsentationen auch auf Entwicklungs-Hostrechnern zu erhalten, mußte ein längliches Hilfspaket erstellt werden:

```

with System;

package Bitranges is      -- MIL-STD-1750A version

    Unit : constant := System.Storage_Unit;

    subtype Bit_00_00 is Natural range 15..15;
    subtype Bit_00_01 is Natural range 14..15;
    subtype Bit_00_02 is Natural range 13..15;
    subtype Bit_00_03 is Natural range 12..15;
    subtype Bit_00_04 is Natural range 11..15;
    subtype Bit_00_05 is Natural range 10..15;
    subtype Bit_00_06 is Natural range 09..15;
```

```
subtype Bit_00_07 is Natural range 08..15;
subtype Bit_00_08 is Natural range 07..15;
subtype Bit_00_09 is Natural range 06..15;
subtype Bit_00_10 is Natural range 05..15;
subtype Bit_00_11 is Natural range 04..15;
subtype Bit_00_12 is Natural range 03..15;
subtype Bit_00_13 is Natural range 02..15;
subtype Bit_00_14 is Natural range 01..15;
subtype Bit_00_15 is Natural range 00..15;

subtype Bit_01_01 is Natural range 14..14;
subtype Bit_01_02 is Natural range 13..14;
subtype Bit_01_03 is Natural range 12..14;
subtype Bit_01_04 is Natural range 11..14;
[...]
subtype Bit_01_14 is Natural range 01..14;
subtype Bit_01_15 is Natural range 00..14;
subtype Bit_02_02 is Natural range 13..13;
subtype Bit_02_03 is Natural range 12..13;
[...]
subtype Bit_02_15 is Natural range 00..13;

[...]
subtype Bit_14_14 is Natural range 01..01;
subtype Bit_14_15 is Natural range 00..01;
subtype Bit_15_15 is Natural range 00..00;
end Bitranges;
```

Ein weiteres Paket, Long_Bitranges, stellt entsprechende Deklarationen für Long_Integer, Bit-Bereich 0..31, zur Verfügung.

In der Hostrechner-Version sieht das Paket "normaler" aus:

```
with System;

package Bitranges is      -- host computer native version

    Unit : constant := System.Storage_Unit;

    subtype Bit_00_00 is Natural range 00..00;
    subtype Bit_00_01 is Natural range 00..01;
    subtype Bit_00_02 is Natural range 00..02;
    [...]
    subtype Bit_15_15 is Natural range 15..15;
end Bitranges;
```

Repräsentationsklauseln müssen dann wie folgt geschrieben werden:

```
with Bitranges; use Bitranges;

package Example is

    type My_Rec is record
        A : Short_Integer;
        B : Short_Integer;
    end My_Rec;

    for My_Rec use record
        A at 0*Unit range Bit_00_01'first..Bit_00_01'last;
        B at 0*Unit range Bit_02_15'first..Bit_02_15'last;
    end record;
end Example;
```

11.3.5 Boden: Vereinfachung durch Abstraktion (Fallbeispiel)

Dieses Beispiel zeigt, wie durch Ändern der sichtbaren Schnittstelle die Handhabung eines angebotenen Dienstes vereinfacht werden kann. Es handelt sich um eine Programmierschnittstelle zum "Shared Memory" des Betriebssystems. Die erste Variante war wie folgt:

```
with System;

procedure Create_Global_Section (Name           : String;
                                Size           : Natural;
                                Section_Address : out System.Address;
                                Created        : out Boolean);
```

Die Anwendung dieses Dienstes mußte mehrstufig erfolgen, d.h. man war zu einer bestimmten Aufteilung in Hilfspaket und -Funktion gezwungen. Zunächst die Paketspezifikation mit der Deklaration des Datentyps, dessen Instanzen ins Shared Memory gelegt werden soll:

```
package Data_Type_Pkg is
    type Shared_Data_Type is record
        X, Y, Z : Float;
    end record;
end Data_Type_Pkg;
```

Dann eine Hilfsfunktion, die die Adresse des erzeugten Shared-Memory Segments zurückliefert:

```
with System;
with Data_Type_Pkg, Create_Global_Section;

function Create_Shared_Data return System.Address is
    Section_Address : System.Address;
    Created         : Boolean;
```



```
begin
    Create_Global_Section (Name => "my_shared_data",
                          Size => Shared_Data_Type'size/8,
                          Section_Address => Section_Address,
                          Created => Created);

    return Section_Address;
end Create_Shared_Data;
```

Schließlich die Ausprägung:

```
with System, Unchecked_Conversion;
with Data_Type_Pkg, Create_Shared_Data;
pragma Elaborate (Create_Shared_Data);
package Glob_Sect_Demo is

    type Data_Access is access Data_Type_Pkg.Shared_Data_Type;

    function To_Data_Access is new Unchecked_Conversion (System.Address,
                                                         Data_Access);

    Shared_Var : Data_Access := To_Data_Access (Create_Shared_Data);
end Glob_Sect_Demo;
```

Wie kam es zu diesem scheinbar komplizierten Aufbau? Das Erzeugen der sogenannten "Global Sections" unter VMS ist nicht ganz einfach. Nach einigen erfolglosen Versuchen hat man sich genau an das Beispielprogramm gehalten, welches hierzu im VAX-Ada Runtime Reference Manual steht. Nach dem Motto "never touch a running system" wurde auch die sichtbare Schnittstelle des DEC-Beispiels eins zu eins übernommen. Erst bei der Portierung der Software auf das Unix-Betriebssystem kam die folgende Überarbeitung zustande:

```
generic

  type Object is private;

  type Object_Pointer is access Object;

  Max_Instances : Positive := 32;

  -- Max_Instances determines how many times Acquire can be called
  -- per instantiation of Shared_Memory.

  Verbose : Boolean := false;

  -- When (Verbose = true), a message is written to the standard
output
  -- upon each call to Acquire or Release.

package Shared_Memory is

  function Acquire (Name : String) return Object_Pointer;

  -- Returns the access value for the object allocated, or a null
  -- access value in case of error.

  function Release (Name : String) return Boolean;

  -- Returns TRUE on successfully releasing the named memory segment,
  -- or FALSE in case of error.

end Shared_Memory;
```

Hier die Anwendung des Pakets Shared_Memory auf den Shared_Data_Type des vorigen Beispiels:

```
with Data_Type_Pkg, Shared_Memory;

package Shared_Mem_Demo is

    type Data_Access is access Data_Type_Pkg.Shared_Data_Type;

    package Shared_Data_Pkg is new Shared_Memory
        (Data_Type_Pkg.Shared_Data_Type, Data_Access);

    Shared_Var : Data_Access := Shared_Data_Pkg.Acquire
        ("my_shared_data");

end Shared_Mem_Demo;
```

Die zweite Variante unterscheidet sich in drei Punkten von der ersten:

- a) Namensgebung -- der Dienst ist nicht schon durch die Namensgebung an eine Eigenheit des jeweiligen Betriebssystems gekoppelt (z.B. "Create_Global_Section")
- b) Sichtbarkeit von System.Address -- Größen wie System.Address liegen im Allgemeinen außerhalb der Applikationsdomäne und sollten nach Möglichkeit nicht Package Spec-sichtbar sein.
- c) Eliminierung von Zwischenschritten bei der Anwendung -- die Wiederverwendung der Prozedur Create_Global_Section erzwang das Schreiben einer Hilfsfunktion. Durch relativ einfache Änderungen des sichtbaren Interfaces hat sich dieser Zwischenschritt erübrigt. Erst im Package Body unterscheidet sich die VMS- von der UNIX-Version.

11.3.6 Bord: In Assembler oder in Ada?

Die Entscheidung, ob ein Programmteil in Assembler oder in Ada geschrieben wird, wird primär durch dessen Grad an Hardwareabhängigkeit bestimmt. Beispiel: eine Interrupt Service-Routine wird normalerweise in Assembler geschrieben, da diese mit einem speziellen Assemblerbefehl verlassen werden muss. Weniger zeitkritische Folgeaktionen, die von der Service-Routine nur angestoßen werden, können hingegen in Ada geschrieben sein.

Wenn der Programmteil stark an das verwendete Echtzeit-Betriebssystem gekoppelt ist (etwa: Gerätetreiber), wird zumeist ebenso in Assembler kodiert. Hier gibt es jedoch Ausnahmen. So

benötigte z.B. ein bestimmtes Gerät einen Pufferbereich für die Einspeisung der Daten. Der Puffer sollte sich aber wie ein Ringpuffer verhalten (also ein eindimensionales Array mit Schreib- und Lesezeiger sowie Füllstandanzeigen). Es wurde erkannt, daß es sich dabei um einen allgemein verwendbaren Mechanismus handelt. Die Verwaltung des Ringpuffers konnte in Ada realisiert werden, da es sich zeigte, daß der vom Crosscompiler erzeugte Code hinreichend speichereffizient war. Die Formulierung eines generischen Pakets, welches den Ringpuffer-Mechanismus verallgemeinert, war andererseits (für die Onboard-Software) nicht möglich, da der resultierende Code sich als zu speicherintensiv erwies.

Es gibt auch Programmteile, bei denen die Entscheidung Assembler versus Ada weniger eindeutig ist. Dazu zählt die Telemetrie- und Telekommando-Verarbeitung. Aufgabe der Telemetrieverwaltung ist es, verschiedene Daten aus der Lageregelungssoftware in vordefinierte Daten-*Packets* zusammenzupacken, welche an die Bodenstation gesendet werden. Das Gegenstück dazu, die Telekommandoverarbeitung, dekodiert Kommandodaten-Packets, welche von der Bodenstation empfangen wurden.

Traditionell sind diese Programmteile in das Onboard-Betriebssystem eingebettet und in Assembler kodiert. Dennoch wurde in einem neueren Projekt z.B. die Telemetrie-Verarbeitung in Ada realisiert. Das war deswegen möglich, weil die zu telemetrierenden Daten zum Großteil ihrerseits aus in Ada geschriebener Software stammen. Die gesamte Ada-Software wurde so entworfen, daß alle zu telemetrierenden Variablen *package spec*-sichtbar sind.

Allerdings ist der Code für die benötigten Datenpaketaufbau-Tabellen etwas gewöhnungsbedürftig. Es wimmelt dort von Attributen wie `'address`, `'size` und `'length`. Es bedarf außerdem einiger Hilfskonstruktionen, da z.B. außerhalb der Ada-Welt liegende, zu telemetrierende Werte erst der Telemetrieverwaltung (in Ada per `pragma Import` etc.) bekannt gemacht werden müssen.

Weiterhin hat der Telemetry-Handler dadurch, daß er Zugriff auf sämtliche zu verschickende Daten haben muß, eine Kopplung zu sehr vielen anderen Ada-Paketen. Wie eine Krake zieht das TM-Handling sämtliche Packages, die TM-relevante Variablen enthalten, per `with` an.

Der große Vorteil der Ada-Version liegt darin, daß diese auf dem Entwicklungsrechner vorgetestet werden kann - bequemer und zeitsparender als auf dem Zielrechner. Über die Wartbarkeit der Ada-Version gegenüber der Assembler-Version liegen allerdings noch keine Erfahrungen vor.

11.3.7 Wer verdreht hier die Bytes?

Sobald man Software an Hardware zu koppeln hat, geht die Frage los: in welcher Reihenfolge kommen die Bytes an? Höherwertiges oder niederwertiges Byte zuerst? Wir müssen eingestehen, daß auch die Verwendung von Ada uns in diesem Punkt keine definitive Lösung gebracht hat. Selbst wenn die Hardware-Beschreibung diese Information enthält, kann dennoch irgendwo ein zusätzlicher Dreher passieren (etwa im Betriebssystem oder Gerätetreiber). Es muß also letztlich immer erst der Praxistest zeigen, ob z.B. die Langworte oder Fließkommazahlen richtig wieder zusammengefügt wurden. Zur Not muß die Software dann bei Bedarf noch mal "drehen", wenn's nicht paßt!

Wenn es sich um In-House gebaute Hardware handelt, ist das Problem in den Griff zu bekommen: es ist darauf zu bestehen, daß die Geräte ihre Daten in IP-Netzwerkordnung aufs Kabel legen. Dazu gibt es in der BSD- (Unix-) Sockets-Programmiersbibliothek nützliche Funktionen mit Namen wie "Host to Network Long" (htons) oder "Network to Host Short" (ntohs), die die entsprechenden Wandlungen für 16- und 32-Bit-Worte zur Verfügung stellen.

11.4 Zusammenfassung und Ausblick

Ada hat sich als Programmiersprache bei uns - fernab jeden Mandats - fest etabliert und bewährt. Als besonders vorteilhaft hat sich die strenge Prüfung der Modulschnittstellen erwiesen. Allerdings hat es einiger Jahre bedurft, die Möglichkeiten von Ada voll (bzw. *angemessen*) auszuschöpfen, also einerseits schon beim Entwurf von Systemen Ada einzubeziehen (z.B. Definition günstiger und sauberer Modulschnittstellen durch Spec/Body-Trennung, Verwendung von generischen Moduln), andererseits auch die Ada-Sprachmittel adäquat einzusetzen und nicht "C-in-Ada", "Fortran-in-Ada" usw. zu programmieren. Die Bordsystemprogrammierung unterliegt harten Grenzen hinsichtlich des Speicherplatzbedarfs, deswegen kann dort nicht volles Ada eingesetzt werden. In der Test- und Bodensystemprogrammierung wird Ada uneingeschränkt eingesetzt und ist von dort kaum mehr wegzudenken.

Die wesentlichen Ausblicke für die Bordsoftware-Entwicklung sind:

- * Übergang zu Ada95, möglichst in Verbindung mit dem Übergang zu einem leistungsfähigeren (32-Bit, strahlungsfesten) Prozessor.
- * Evaluierung der Tasking-Möglichkeiten von Ada95 in Verbindung mit der Evaluierung des Zielrechner-Laufzeitsystems beim konkreten Ada95-Crosscompiler. Wenn möglich, Übergang zu vollem Ada95 mit Tasking anstelle des jetzigen dedizierten Echtzeit-Betriebssystems.

Die Tendenzen in der Test- und Bodensystemprogrammierung sind folgende:

Von	Zu
-----	-----
Separaten Anfertigungen der Test-, Simulations- und Kontrollstations- SW	Gemeinsamem Software-Kern für Test, Simulation, und Bodenstation
Monolithischen Systemen, Kommunikation per direktem Unterprogrammaufruf oder Task-Rendezvous	Auf Prozeßebene modularisierten Systemen mit standardisierten externen Kommunika- tionsmechanismen (Queues, Sockets)
Software ist schon beim Entwurf fest an ein Betriebssystem/eine Rechner-Hardware gekoppelt	Man ist sich der BS- u. Rechner-Abhängig- keiten bewußt und es wird schon bei der Systemdefinition auf Portabilität geachtet
Binäre Datenformate der Telemetrie und Telekommandos in der Bodensoftware per Record-Repräsentationsklauseln fest- kodierte (bei Änderung der Formate ist Rekompilation nötig)	Datenformatbeschreibungen werden bei Programmstart aus ASCII-Dateien eingelesen. Keine Rekompilation bei Änderungen in den Formaten. Zudem können die entsprechenden bordseitigen Tabellen z.B. als Sourcecode automatisch generiert werden (Vermeidung von Fehlern bei der Parallelführung von Bord- und Bodensoftware)
Einzelne Utilities wie Parser, Graphik-, Plot-, Datenauswertungs-, oder Konverter- programme in Ada (od.a. 3GL) schreiben	Zunächst nachschauen, ob es das Tool bereits als Freeware gibt und wenn nicht, dann in einer Scripting-Sprache schreiben (s. REF-1)
GUI-Programmierung in X-Windows/OSF- Motif mit Ada-Interface	pTk (Perl/Tk, s. REF-2) mit TASH (Tk/Ada-Kopplung, s. REF-3) wo nötig

11.5 Literatur

REF-1

John K. Ousterhout:

"Scripting: Higher Level Programming for the 21st Century",

<http://www.sunlabs.com/people/John.Ousterhout/scripting.html>

REF-2

Scott Raney:

"The Scripting Revolution (Graphical Perl)",

The X Journal, November 1996

<http://www.sigs.com/publications/docs/txjr/9611/txjr9611.raney.html>

REF-3

Terry J. Westley:

"TASH: Tcl Ada SHell, An Ada/Tcl Binding",

<http://www.ocsystems.com/xada/tash/>

V Teilnehmer

12 Teilnehmer

NAME	VORNAME	FIRMA	ANSCHRIFT	E-MAIL
Beck	Gerhard	Bosch Telecom	Gerberstr. 33, 71520 Backnang	gerhard.beck@ bk.bosch.de
Bühler	Gerhard	FFM	Neuenahr Str. 20, Wachtberg	buehler@fgau.de
Burgbacher	Horst	EUROCONTRO L Karlsruhe	Rintheimer Querallee 6, 76131 Kh'e	
Dencker	Peter		Steinäckerstr. 25, 76275 Ettlingen	dencker@aonix.d e
Fick	Andreas	FZ Karlsruhe		fick@IAI.FZK.DE
Ginkel	Axel	Eurocopter, 81663 München	Wasserburger Landstr. 250A, 81827 München	ginkel@nebplace. de
Gliss	Bernd	MPIe Stuttgart	Hirsenbergstr. 1, 70569 Stuttgart	gliss@edv.mpi- stuttgart.mpg.de
Hermann	Peter	Uni Stuttgart	Alte Dorfstr.11, 71229 Leonberg- Geb.	ph@csv.ica.uni- stuttgart.de
Keller	Hubert B.	FZK Karlsruhe		Keller@iai.fzk.de
Kellogg	Oliver	DSS (Dornier Satelliten- systeme)		oliver.kellogg@ Space.otn.dasa.d e
Kramer	Henry	Alcatel SEL	Ostendstr.3, 75175 Pforzheim	
Landwehr	Rudolf	CCI GmbH	Lohberg 10, 49716 Meppen	landwehr@cci.de
Lucas	Kai		Senserstr. 4, 82140 Olching	
Lüß	F.-P.		Kiel	

Mangold	Karl Otto	ATM Konstanz		mangold@ atmcomputer.de
Möhlmann	Bert	debis Systemhaus	Lademannbogen 21/23, 53343 Hamburg	
Neumann	Horst		Burgmaierstr. 10b, 85521 Ottobrunn	horst.neumann@ t-online.de
Osterhues	Bernhard, große	FZK, Karlsruhe		osterhues@ iai.fzk.de
Paus	Michael		Kauzenhecke 12, 70597 Stuttgart	michael@ifr. Luftfahrt.uni- stuttgart.de
Plödereder	Erhard	Univ. Stuttgart, Inst. f. Informatik	Breitwieserstr. 20- 22, 70565 Stuttgart	ploedere@ informatik.wi- stuttgart.de
Ritter	Andreas	NORTEL DASA NETWORK SYSTEMS		ritter@ comsys.dofn.de
Röhrle	Jörg	Fachhochschule Albstadt- Sigmaringen	Johannesstr. 3, 72458 Albstadt	roehrle@ fh-albsig.de
Sauermann	Gerd	Concurrent Comp. GmbH	Planegg	gerd.sauermann @clur.de
Schwald	Andreas		Guardinistr. 73, 81375 München	ASCHWALD@ compuserve.com
Seidel	Helmut	Eurocopter,	81663 München	
Siara	Reinhard		Rusebyyer Str. 5, 24340 Eckernförde	
Tempelmeier	Theodor	Fachhochschule	Marienberger Str. 26, 83024 Rosenheim	tt@extern.lrz- muenchen.de
Thiemann	A.	BWB IT III3	Koblenz	
Tonndorf	Michael	IABG	Einsteinstr. 20, 85521 Ottobrunn	tonndorf@iabg.de

Trieffurth	Thomas	HEITEC Industriepl. Karlsruhe	Daimerlerstr. 10, 76185 Karlsruhe	
Weinert	Annette		Karlsruhe	
Zeh	Albrecht	SEKAS GmbH	Perchtinger Str.3, 81379 München	zeh@SEKAS.de

VI Ada Deutschland

13 Ada Deutschland - GI Fachgruppe 2.1.5 "Ada"

Ada Deutschland als GI Fachgruppe 2.1.5 "Ada" unterhält unter der domain www.ada-deutschland.de die offiziellen Web Seiten. Nachfolgend wird die Hauptseite von Ada Deutschland per 23.7.1997 beispielhaft dargestellt.



Ada-Deutschland

Dies ist die offizielle Home Page von **Ada-Deutschland**, der Fachgruppe 2.1.5 "Ada" der [Gesellschaft für Informatik](http://www.gi-informatik.de).



Ada - The Language for a Complex World

- [Aktuelles](#)
- [Top Tip zu Ada, C++, Java](#)
- [Die Fachgruppe 2.1.5 "Ada"](#)
- [Ada und Software Engineering](#)
- [Ada Arbeitskreise](#)
- [Softwaretechnik-Trends](#)

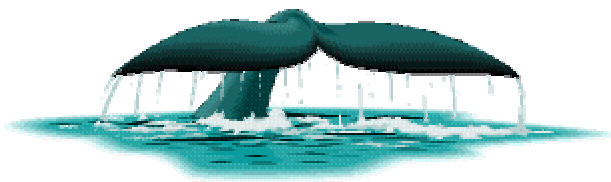
- [Ada Stellenangebote](#)
** vom 11.7.1997 **
- [Ada Literatur](#)
- [Ada Projekte](#)
- [Ada an Hochschulen](#)
- Ada Experten und Produkte/Anbieter
 - [Experten zu OOA, OOD und Ada](#)
 - [Produkte, Anbieter zu Ada](#)

Haben Sie Interesse sich oder Ihr Produkt hier zu plazieren, vielleicht mit einer eigenständigen Web Seite, so melden Sie sich doch bitte beim Ada-Deutschland Server.

- [Ada World Wide](#)
-

Der Top Tip

- [Java and Ada / Programming the Internet in Ada 95](#)
- [Ada, C, C++, and Java](#)
- **Von C/C++ zu Ada95**



[Ein Englisch-sprachiges Tutorial, das die hervorragenden Möglichkeiten von Ada95 darstellt und vor allem auch den Weg von C/C++ zu guter Ada95 Programmierung beschreibt.](#)

- [Comparison of Ada and C++ Features, Eigenschaften von Ada und C++ im Vergleich](#)
-

- **Links der GI**

[Gesellschaft für Informatik - GI](#)

[Fachausschuß 2.1 Softwaretechnik und Programmiersprachen](#)

[Softwaretechnik \(FG 2.1.1\)](#)

[Requirements Engineering \(FG 2.1.6\)](#)

[Test, Analyse und Verifikation von Software \(FG 2.1.7\)](#)

[Software-Entwicklungsumgebungen \(FG 2.1.8\)](#)

[Objektorientierte Software-Entwicklung \(FG 2.1.9\)](#)

[Arbeitskreis "Generative und Komponentenbasierte Softwareentwicklung" der FG 2.1.9](#)

Anträge auf Aufnahme in die GI oder die Fachgruppen sind an die
Geschäftsstelle der GI zu richten:

Gesellschaft für Informatik e. V., Wissenschaftszentrum, Ahrstr. 45, 53175 Bonn,
Tel. 0228-302145)

Die Erstellung dieser Home Page von Ada-Deutschland konnte mit der freundlichen Unterstützung von [Aonix](#) auf den Servern des [Forschungszentrums Karlsruhe](#) realisiert werden.

Für Anregungen oder interessante Verweise auf wichtige, hier aber noch nicht vorhandene Einträge oder Web-Seiten, bitte eine Nachricht an keller@iai.fzk.de schicken.

