

Erfahrungen mit der werkzeug-gestützten Änderung von COBOL-Systemen*

Dr. Jürgen Vollmer

– CoCoLab –

vollmer@cocolab.de • www.cocolab.de

Zusammenfassung

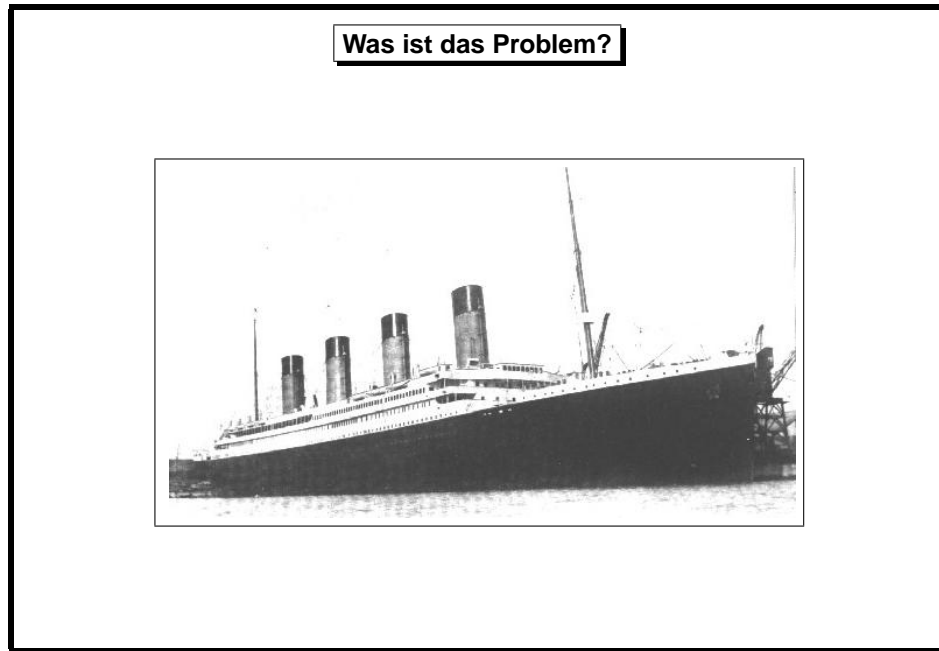
Dieser Vortrag berichtet über Erfahrungen beim Einsatz von Compilerbauwerkzeugen zur Analyse, Umstellung und (Re)dokumentation grosser COBOL-Systeme im Banken- und Versicherungsbereich. Die dazu benutzten Techniken werden kurz gestreift, und es wird skizziert, welche Umstellungen heute automatisch oder halbautomatisch durchgeführt werden können.

Liste der Folien

1	Was ist das Problem?	2
2	CoCoLab = Compiler Compiler Laboratory	3
3	Der CoCoLab-Ansatz	4
4	Interne Programm Darstellung – Abstrakter Syntaxbaum	5
5	Interne Programm Darstellung – Berechnete Information	5
6	Interne Programm Darstellung – Kontrollflußgraph	6
7	Interne Programm Darstellung – Datenflußanalyse	7
8	Anwendung: (Re)-Dokumentation	8
9	Einfache Transformation	9
10	Transformation: Wertpapierkennnummern (WKN)	10
11	Transformation: Wertpapierkennnummern / Programme	10
12	WKN-Umstellungsregeln	12
13	WKN-Umstellung – Technische Probleme	12
14	Aufwand: Wertpapierkennnummern	13
15	Lehren aus den bisherigen Projekten	14
16	These: Vollautomatische Umstellung ist nicht möglich	15
17	Zusammenfassung	16
18	Ausblick	16
19	Zu guter Letzt: Die Rolle von Design Dokumenten ☺	17

*Eingeladener Vortrag, gehalten auf der Fachtagung der *Special Interest Group of the Swiss Informaticians Society (SI) – Software-Engineering*, SI-SE 2003. Thema der Tagung: *Software-Sanierung, Herausforderungen – Ansätze – Erfahrungen*, Universität Zürich-Irchel 13./14. März 2003; <http://www.hsr.ch/weiterbildung/sise2003/index.html>

Folie 1



In James Camerons Kinofilm „Titanic“ (USA 1997) läuft der Schiffskonstrukteur Thomas Andrews nach der Havarie der Titanic mit dem Eisberg zu Kapitän Smith. In der Hand seine Konstruktionspläne, wird er feststellen, daß die Titanic knapp zwei Stunden später sinken wird.

Mir scheint es, als seien die riesigen COBOL-Systeme eine Art Titanic, welche kurz vor einer Havarie, hier in Form von Änderungen der Rahmenbedingungen (Jahr 2000, Euro, gesetzliche Änderungen, usw) stehen. Leider so, daß weder ein einziger Konstrukteur da ist, welcher die ganze Lage überblickt, noch daß es ausreichend und aktuelle Konstruktionspläne gibt. Für COBOL-Systeme sind die Probleme u.A. im organisatorischen und technischen Umfeld zu suchen:

Organisatorisches Umfeld

- Die gesetzlichen, gesellschaftlichen, technischen, usw. Rahmenbedingungen, an welche die Programme angepasst werden müssen, ändern sich.
- COBOL-Programme „leben“ lange, da sie das „Wissen“ des Betriebs darstellen, man kann sie nicht „eben mal so“ austauschen, wie das Betriebssystem eines Arbeitsplatz-PCs.
- Versierte COBOL-Programmierer sind eine aussterbende Spezies!

Technisches Umfeld

- Es gibt trotz Standardisierung viele COBOL-Dialekte.
- Nicht jede Hardwareplattform „spricht“ den gleichen COBOL-Dialekt, dies macht Wechsel der Plattform schwierig bzw. ist, falls nötig, mit großem Aufwand verbunden.
- Jede IT-Abteilung hat eigene Spracherweiterungen und Makropräprozessoren.
- Es gibt sogenannte eingebettete Programmiersprachen, wie CICS, DL/1, ROBOT, SQL usw.
- COBOL ist eine „low-level“ Programmiersprache, die keine Abstraktionsmöglichkeiten bietet, mit entsprechenden Konsequenzen für die Lesbarkeit und Wartbarkeit des Codes.

- COBOL-Programme werden in einem komplexen Umfeld mit vielen Interaktionen zu anderen Systemen (Datenbanken, Transaktionsystem, Dialogsysteme usw.) benutzt.
- Dokumentation ist nicht mehr auffindbar ☹

Zur Unterstützung bei der Änderung von (COBOL)-Programmsystemen bietet CoCoLab einige Lösungen an. CoCoLab ist:

Folie 2

CoCoLab = Compiler Compiler Laboratory

Gegründet: 1994

Mitarbeiter: Dr. Josef Grosch, Dr. Jürgen Vollmer

Ziele:

- Entwicklung & Vertrieb der **Cocktail** Toolbox
Cocktail = Compiler Toolkit Karlsruhe
- Entwicklung von Compilern für Programmiersprachen
- Analyse von Programmsystemen, Reengineering, (Re-)Dokumentation, . . .

Basis:

- Cocktail Toolbox
- Parser z.B. für: COBOL, CICS, Clist, REXX, JCL, DL/1, PL/1, Fortran, SQL (DB2, Oracle, Informix, . . .), C/C++/C#, Java, Natural, . . .
- Programm-Analysen für diese Sprachen

Kunden: Banken, Versicherungen, Telekommunikation, Softwarehäuser, . . .

CoCoLab wurde 1994 von Josef Grosch, der zuvor wissenschaftlicher Mitarbeiter der Forschungsstelle Karlsruhe der GMD (Gesellschaft für Mathematik und Datenverarbeitung) war, gegründet. 1996 stieß Jürgen Vollmer zu CoCoLab (ebenfalls zuvor GMD Karlsruhe und Universität Karlsruhe, Institut Prof. G. Goos).

Geschäftsziele von CoCoLab sind:

- Weiterentwicklung und Vertrieb des **Cocktail** Übersetzerbauwerkzeugkastens, dessen Entwicklung bereits in der Forschungsstelle Karlsruhe der GMD begann.

Die Toolbox besteht aus:

Rex	Scanner Generator
Lark	LR(1) and LALR (2) Parser Generator with backtracking and predicates
Ell	LL(1) Recursive Descent Parser Generator
Ast	Generator for Abstract Syntax Trees
Ag	Generator for Attribute Evaluators
Puma	Transformation of Attributed Trees based on Pattern Matching
Reuse	Library of Reusable Modules
y2l	transformiert Yacc Eingabe in Lark Eingabe
l2r	transformiert Lex Eingabe in Rex Eingabe

Die Toolbox kann Parser in folgenden Ziel-Sprachen erzeugen: C/C++, Java, Modula-2 und Ada.

- CoCoLab entwickelt Übersetzer und Interpreter, sowohl für kundenspezifische Programmiersprachen, als auch für gängige Programmiersprachen und spezielle Hardware.
- CoCoLab bietet Unterstützung im Bereich Reengineering, (Re)-Dokumentation und Analyse von Programmsystemen an. Beispielsweise Jahr 2000, EURO, Wertpapierkennnummerumstellung etc.

Basis für diese Aktivitäten bilden:

- Die Cocktail-Toolbox
- Parser für folgende Sprachen:
Ada, C, C++, C#, CICS, COBOL, OO-COBOL, Clist, Rexx, DL/I, Delphi, Pascal, FORTRAN 77, Fortran 90, HTML, XML, IDL, JCL, Java, Modula-2, NATURAL, OpenUI/OPL, PL/I, Powerbuilder, SQL (ANSI, IBM/DB2, Informix, Oracle, Sybase, Tandem), Tcl, itcl, Visual Basic
- Neben den „eigentlichen“ Parsern, welche einen sogenannten abstrakten Syntaxbaum aufbauen, gibt es auch verschiedene Analysen für Programme dieser Sprachen. z.B. Namensanalysen, Kontroll- und Datenflußanalysen, Toter-Code-Analysen, Slicing, usw.

Zu unseren Kunden zählen Banken, Versicherungen und Softwarehäuser, welche unsere Parser weiter „veredeln“, Firmen aus der Telekommunikationsindustrie, Hersteller von programmierbarer Krypto-Hardware, usw.

Folie 3

Der CoCoLab-Ansatz

- Interne Darstellungsformen eines Programmes:
 1. Liste von Programmzeilen
 - ⇒ einfache Textersetzung möglich
 - ⇒ keine Information über die Struktur ds Programmes
 2. Übersetzerbau (*compiler construction*)
 - ⇒ Strukturinformation
 - ⇒ „Bedeutungs-Information“
 - ⇒ mehr Aufwand
 - ⇒ **CoCoLab & Cocktail**

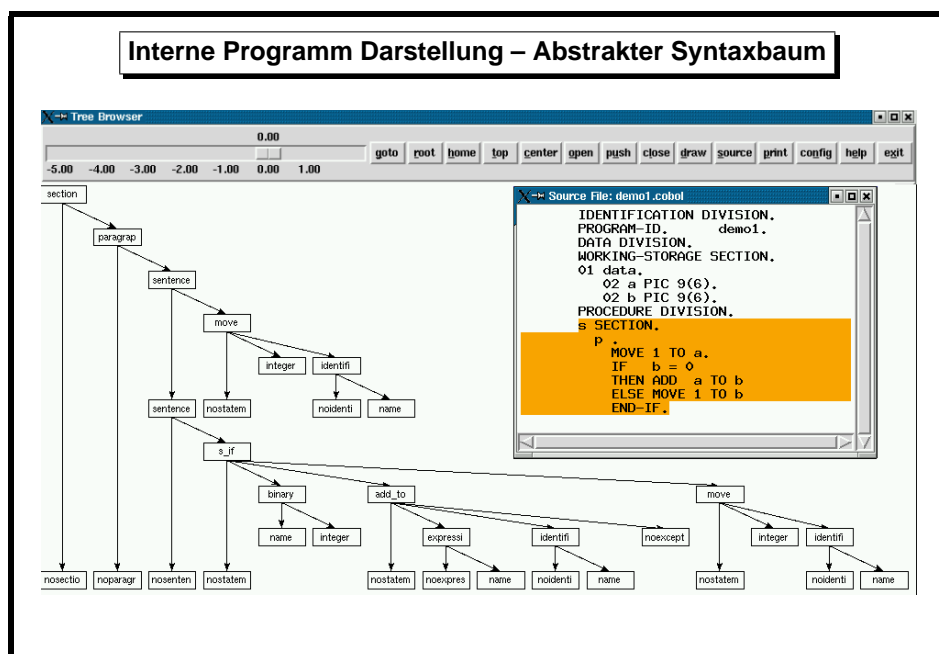
Damit ein Programm transformiert werden kann, muß es im Transformator intern dargestellt werden. Die einfachste Darstellung ist „Liste von Textzeilen“, wie sie in jedem Texteditor benutzt wird. Mit dieser Darstellung kann man schon einige Basistransformationen durchführen: z.B. Textstücke (Variablenamen, Namen von COPY-Strecken, usw.) suchen und ersetzen. Der gravierendste Nachteil dieser einfachen Darstellung ist, daß keinerlei Information über die Struktur des Programmes oder gar Bedeutung des Programmes zur Verfügung steht.

Hier nun setzt die Idee an, Methoden und Werkzeuge aus dem Übersetzerbau (engl. *compiler construction*) und insbesondere auch der Konstruktion von optimierenden Übersetzern, einzusetzen. Der erste Schritt dazu ist es, das Programm zu „zerteilen“ (engl. *to parse*), d.h. in seine syntaktischen Teile zu zergliedern. So wie man es in der deutschen Sprache auch kennt (und in der Schule endlos üben musste): „Ein Satz besteht aus Subjekt, Prädikat gefolgt von einem Objekt“.

Der Preis für diese Technik liegt in der aufwendigeren Erstellung des Transformators und in der u.U. längeren Laufzeit des Transformators (die aber auch im interaktiven Betrieb immer noch vernachlässigbar ist!).

Das erste Analyseergebnis eines Programmes durch einen Übersetzer ist der sogenannte *abstrakte Syntaxbaum* (engl. *abstract syntax tree, AST*), wie er auf der nächsten Folie zu sehen ist.

Folie 4



Diese Folie zeigt einen Ausschnitt aus dem, mittels der Cocktail Werkzeuge *lark* und *ast* erzeugten Syntaxbaumes eines kleinen COBOL-Programmes. Die Darstellung wird automatisch aus der (abstrakten) Grammatik der Programmiersprache und der konkreten Eingabe erzeugt.

Neben dieser strukturellen Darstellung des zu untersuchenden Programmes, werden aber auch noch einige Informationen *über* das Programm *berechnet*. Die wichtigste ist die sogenannte *Definitionstabelle*, welche Informationen über alle vom Programmierer definierten Namen, z.B. Variablen und Paragraphen, darstellt. Die nächste Folie zeigt die Definitionstabelle für die Programmvariablen:

Folie 5

Interne Programm Darstellung – Berechnete Information

The screenshot displays three windows from a COBOL development tool:

- Tree Diagram:** A hierarchical tree showing the structure of variable declarations. A root node 'decl' has two children, each of which is another 'decl' node, representing nested or grouped declarations.
- Source File: demo1.cobol:** Shows the following COBOL code:


```
IDENTIFICATION DIVISION.
PROGRAM-ID.      demo1.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 data.
02 a PIC 9(6).
02 b PIC 9(6).
PROCEDURE DIVISION.
s SECTION.
p .
MOVE 1 TO a.
IF b = 0
THEN ADD a TO b
ELSE MOVE 1 TO b
END-IF.
```
- Attributes of decl @ 08422e14 <2>:** A table of calculated attributes for the variable 'a':

decl	
next	= 08422c5c *
name	= a
position	= "demo1.cobol": 6, 14
end_pos	= "demo1.cobol": 6, 14
e_pos	= "demo1.cobol": 6, 24
level	= 2
fields	= NoTree
lower	= 08422fcc *
object	= 08422eac +
collision	= 00000000 +
kind	= WORKING-STORAGE data item
ContainedInCopy	= 0842371c +
type	= NUMERIC
usage	= none
size	= 6
occurs	= 1
var_size	= 6
offset	= 0
digits	= 6
slack_bytes	= 0
bit_offset	= 0
has_error	= F
ibm_size	= 0
ibm_var_size	= 0
vp_flags	=
vp_memory_bitvector	= <NULL>
vp_pattern	= 00000000 +
vp_nr	= 0

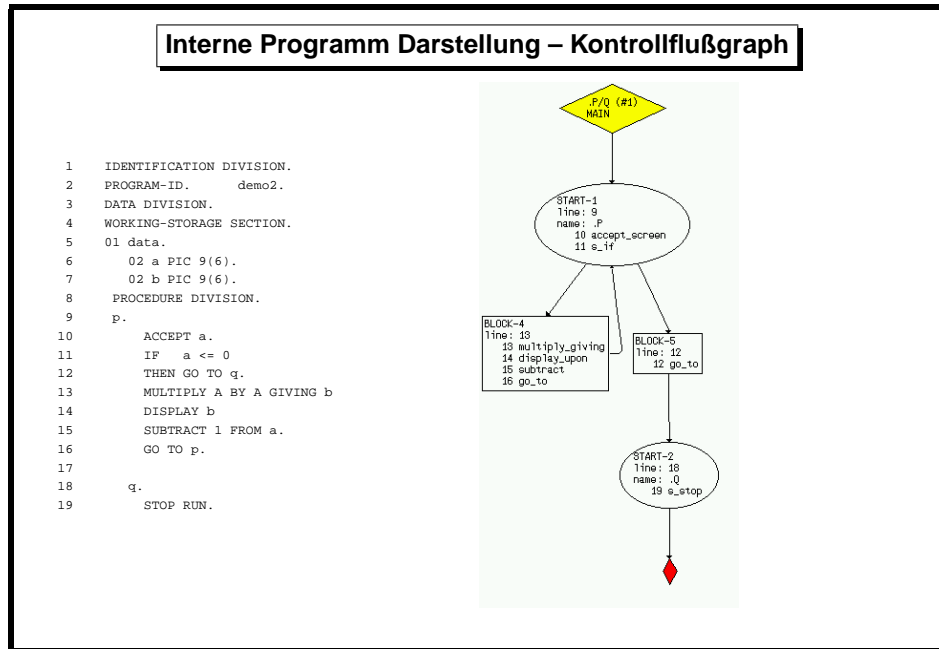
Auch hier wird nicht einfach eine Liste von Variablenamen „gespeichert“, sondern es werden auch Beziehungen zwischen den Variablen repräsentiert: Die Variable `data` ist eine Gruppe mit zwei Feldern `a` und `b`. Für `a` sind ein Teil der gesammelten Information dargestellt: die Stufennummer (`level`); die Tatsache, daß `a` keine Gruppe sondern ein Feld ist (`fields = NoTree`; für `data` steht hier ein Zeiger auf die Deklaration von `a`); von welcher Art und Typ `a` ist (`kind` und `type`); die Anzahl der zur Speicherung benötigten Bytes `size` und die, bezgl. der umgebenden 01-Stufe, relativen Adresse `offset`; ob durch eine `SYNCHRONIZED`-Klausel `slack bytes` nötig werden, bzw. wenn ja wieviele; usw.

Die meisten Knoten im abstrakten Syntaxbaum sowie der Definitionstabelle enthalten Positionsangaben (`position`), welche genau beschreiben, wo das durch diesen Knoten repräsentierte Programmwort oder Programmkonstrukt in der Programmquelle steht. Dazu werden der Dateiname, die Zeile und die Spalte gespeichert. Diese Positionsangaben sind für die Durchführung von Transformationen essentiell wichtig.

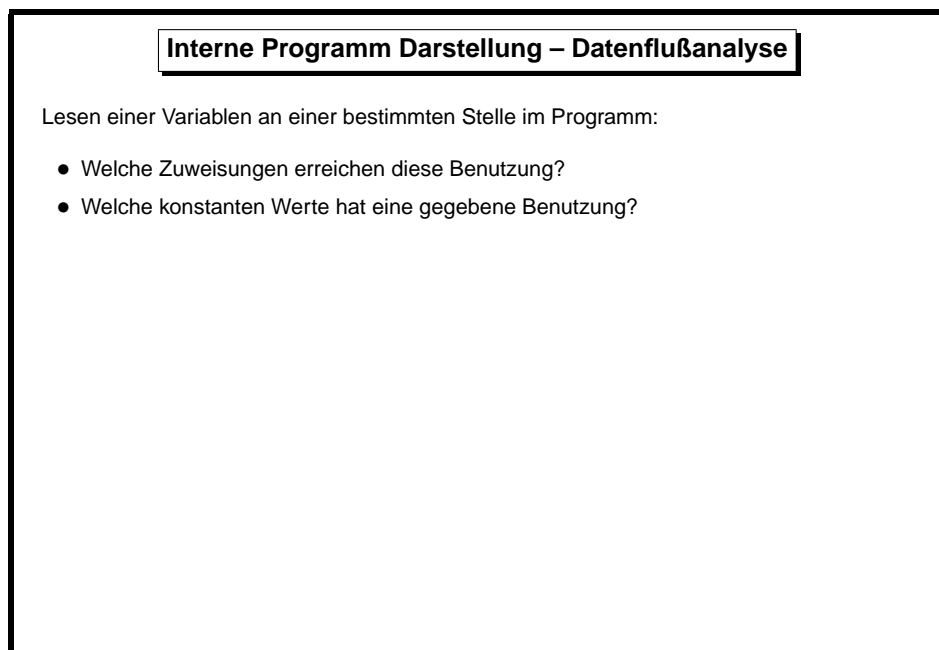
Die sogenannte *Namensanalyse* berechnet nun für jedes Auftreten eines Namens im Programm, welche Deklaration diesem Auftreten zugeordnet ist. Konkret bedeutet dies: in jedem `name` Knoten auf Folie 4 wird ein Zeiger auf die entsprechende Deklarationsinformation gespeichert. Zur Auflösung von Mehrdeutigkeiten (eine Name ist mehrfach als Feld in verschiedenen 01-Gruppen definiert) werden die gleichen Regeln benutzt, die ein COBOL-Übersetzer verwendet.

Eine andere interne Darstellung des Programmes ist der Kontrollflußgraph (engl. *control flow graph*), welche das Ergebnis der Kontrollflußanalyse ist. Er beschreibt in welcher Reihenfolge die Anweisungen ausgeführt werden. Dabei werden die Anweisungen, welche den Programmablauf verändern, beachtet. Als direktes Ergebnis kann sogenannter toter Code gefunden werden, d.h. Anweisungen die nie ausgeführt werden:

Folie 6



Folie 7



Das folgende Bild zeigt alle Datenflüsse auf. Die Variable **a** in Zeile 11 erhält ihren Wert sowohl von der ACCEPT-Anweisung in Zeile 10, als auch von der Subtraktion in Zeile 15.

```

1 IDENTIFICATION DIVISION.
2 PROGRAM-ID. demo2.
3 DATA DIVISION.
4 WORKING-STORAGE SECTION.
5 01 data.
6 02 a PIC 9(6).
7 02 b PIC 9(6).
8 PROCEDURE DIVISION.
9 p.
10 ACCEPT a.
11 IF a <= 0
12 THEN GO TO q.
13 MULTIPLY A BY A GIVING b
14 DISPLAY b
15 SUBTRACT 1 FROM a.
16 GO TO p.
17
18 q.
18 STOP RUN.

```

Hat man diese und noch weitere Analysen, kann man anfangen ein Programm zu (re)-dokumentieren:

Anwendung: (Re)-Dokumentation

Aufgabenstellung:

Welche Bedeutung hat ein CALL?

```
CALL "handle-file" USING param, args.
```

Beispiel:

Öffnen/Lesen/Schreiben/Schliessen einer Datei.

Erwünschte Ausgabe:

CALL in Zeile 100 öffnet Datei abc zum Lesen im Mode xyz.

Aufgabe des Benutzers:

Spezifikation der Tatsache:

Feld von 1. USING Argument auf Offset 5 legt Aktion fest

Feld von 1. USING Argument auf Offset 6 die Art der Datei

Feld von 1. USING Argument auf Offset 7-20 den Namen der Datei

Zulösende Probleme:

Finde alle Zuweisungen an relevante Parameter, bestimme deren Werte
(Datenflußanalyse + Konstantenpropagation)

Folie 8

Einer unserer Kunden stand vor der Aufgabe, seine vorhandenen Altsoftware vor einer grösseren Umstrukturierung besser zu dokumentieren. Eine grosse „Unbekannte“ waren die CALL Anweisungen und deren Bedeutung. Es sollte für jede CALL-Anweisung im Programm herausgefunden werden, was sie an dieser Stelle bedeutet und dies in einer verständlichen Form ausgegeben werden. Dazu hat der Kunde spezifiziert, welche Parameter mit welcher Wertbelegung welche Bedeutung haben. Dazu wurde eine kleine Spezifikationssprache entwickelt, die zusammen mit dem Programm eingelesen wurde. Technisch gesehen war es nötig den Datenfluß und die Werte der Parameter zu

bestimmen. Dies ist natürlich nicht immer möglich, aber für diese konkrete Anwendung war es meist möglich. Es kam dabei vor, daß eine CALL-Anweisung mehrere Bedeutungen umfasste, da die Parameter in IF-Anweisungen in Abhängigkeit von verschiedenen Bedingungen gesetzt wurden. Mit diesen Analyseergebnissen lassen sich auch ohne weiteres einfache Textersetzungen durchführen:

Folie 9

Einfache Transformation

Beispiele:

- Einfügen/Ersetzen syntaktischer Klauseln, z.B. END-IF
- Systematisches Ersetzen von bestimmten Bezeichnern:


```

01 data1.
  02 part1.
    10 x PIC 9. unverändert
01 data2.
  02 x PIC 9. ⇒ 01 y PIC 9.
MOVE x OF data1 TO x OF data2. ⇒
MOVE x OF data1 TO y OF data2.
```
- Modifikation von Programmnamen von aufgerufenen Programmen:


```

01 prog X(10).
MOVE "A" to prog. ⇒ MOVE "B" to prog.
CALL prog.
CALL "A" ⇒ CALL "B"
```

Für das Ersetzen eines Punktes nach der IF-Anweisung benötigt man Kenntnis, wo die IF-Anweisung „aufhört“. Diese Information ist aber genau im abstrakten Syntaxbaum dargestellt.

Sollen z.B. Feld/Gruppen-Bezeichner einer bestimmten 01-Stufe ersetzt werden, dann benötigt man die Ergebnisse der Namensanalyse.

Für die Ersetzung des Programmnamens benötigt man die Datenflußanalyse, um festzustellen, welchen Wert die Variable `prog` bei einer `CALL` Anweisung hat. Es sollen ja erstens nicht alle Aufrufe verändert werden, sondern nur die von "A"; und zweitens, die Variable kann ja auch noch für andere Zwecke benutzt werden, für diese soll der Wert natürlich nicht verändert werden. Danach muß man alle „gültigen“ Zuweisungen finden und diese ersetzen.

Komplexere Transformationen benötigen weitere Analysen:

Folie 10

Transformation: Wertpapierkennnummern (WKN)

Problemstellung:
numerisch 6 Stellen ⇒ alphanumerisch 6 Stellen

Technische Aufgaben:

- Umstellung der Datenbanken
- Umstellung der Dateien
- Umstellung der COBOL-Programme
- Compile & Link & Test

Organisatorische Aufgaben:

- (Programm)-Clusterbildung
- Reihenfolge der Umstellung
- Verwaltung der Quellen
- Fachliche Änderungen während Umstellung
- Parallelbetrieb Alt/Neu

Die Transformation von Wertpapierkennnummern (WKN) ist notwendig, da sie internationalisiert werden und nun *ISIN* Nummern heissen. WKN's sind numerische sechstellige Daten. Die *ISIN* wird im „Endausbau“ 12 stellig sein und aus alphanumerischen Zeichen bestehen. Dieses Projekt wurde mit einer großen deutschen Bank durchgeführt. In einem ersten Schritt sollten nur 6-stellige alphanumerische *ISIN*'s (der WKN-Anteil) benutzt werden.

Das Projekt umfasste die Umstellung der Datenbanken, Dateien, JCL's und COBOL-Programme. CoCoLab wurde beauftragt ein Werkzeug für die Umstellung der Programme zu erstellen. Die eigentliche Umstellung der COBOL-Programme, Datenbanken usw. wurde durch Mitarbeiter der Bank und anderen Fremdfirmen durchgeführt.

Neben den rein technischen Fragestellungen gab es auch viele organisatorische Dinge zu klären. Es begann damit, die Programmquellen vollständig zu besorgen (die Umstellung wurde i.d.R. nicht von den Fachabteilungen vorgenommen sondern zentral). Da es absehbar war, daß viele Komponenten umgestellt werden müssen, war eine Umstellung zu einem Termin nicht erwünscht. Statt dessen sollten „Cluster“ aus zusammenhängenden Programmen gebildet werden, die als Ganzes dann umgestellt werden. Dies bedingt natürlich eine Coexistenz von Alt- und Neusoftware, mit entsprechenden Problemen. Es wurden ca. 50 Cluster gefunden, die aus 1 bis 400 Programmen bestehen. Zum Auffinden der Cluster wurde die *CALL*-Analyse des Werkzeuges benutzt.

Erschwerend kam hinzu, daß es kein „Codefreeze“ in den Fachabteilungen gab, d.h. selbst während des Umstellungsprozesses an einem Cluster wurden weiter fachliche Änderungen an den Programmen vorgenommen. Eine gute Versionsverwaltung der Quellen war hierzu notwendig.

Folie 11

Transformation: Wertpapierkennnummern / Programme

Umstellung der COBOL-Programme:

- Festlegung der Umstellungsregeln
 - Festlegung des Initialwertes (SPACE oder ZERO)
 - Umstellung der PICTURE Klausel
 - Umstellung der Anweisungen
- Welche Variablen enthalten WKN?
 - Finden der „initialen Treffer“
 - ⇒ Suchliste der Art *WKN* oder W6N21
 - Propagation der Eigenschaft „enthält WKN“
 - ⇒ `MOVE variable TO WKN-variable`
 - ⇒ `MOVE WKN-variable TO variable`

Die Umstellungsregeln wurden von einem versierten COBOL-Kenner einer Fremdfirma entwickelt. Sie lagen als informelles Dokument vor und beschreiben die Transformationen und Bedingungen, unter denen sie angewendet werden können. Es wurden nicht alle Regeln implementiert, da für manche der Implementierungsaufwand in keinem Verhältnis zur Häufigkeit der Anwendbarkeit in den konkreten Quellen stand. Das Werkzeug unterstützte hier vorab in der Abschätzung, ob es sich „lohnt“, eine Transformation zu implementieren.

Wie findet man aber nun die WKN-relevanten Variablen und Anweisungen? Da WKN's in einem Programm nicht direkt zu erkennen sind (es gibt keine COBOL-Funktionen, die WKN's „behandeln“), musste die Spezifikation manuell erfolgen. Von Vorteil erwies sich, daß die Variablen i.d.R. einen gemeinsamen Namensanteil hatten, so daß Suchlisten mit Variablennamen spezifiziert werden konnten. Für das WKN-Projekt wurden 480 verschiedene COBOL-Namen und Namensanteile und zusätzlich 101 verschiedene DB2-Spaltennamen, die WKN-relevant waren, angegeben.

Aber nicht nur diese „initialen Treffer“ sind WKN-relevante Variablen, sondern auch

- alle Variablen, die Werte dieser initialen Treffer aufnehmen und
- alle Variablen, die zur Berechnung der WKN-relevanten Variablen benutzt werden.

Dieser Prozess des Auffindens weiterer Variablen wurde *Propagation* genannt. Der Transformator berechnet dazu für Speicherstelle einer Variablen die Eigenschaft „ist WKN-relavent“ bzw. „ist nicht WKN-relavent“. Es wurden bis zu 800.000 Anweisungen und Variablendeklarationen gefunden. Diese Anzahl konnte durch einige Heuristiken auf ca. 300.000 Stellen¹ eingeschränkt werden.

Bei diesen Zahlen wird klar, daß es sich nun um ein *Massenproblem* handelt. Nach anfänglichen Versuchen diese Stellen von Hand zu sichten, wurde bald darauf verzichtet (auch Dank der Weiterentwicklung des Werkzeuges).

¹z.B. wurden alle MOVE-Anweisungen nicht mehr angezeigt, deren Quelle und Ziel gleich lang waren; es wurden nur noch Variablen betrachtet, die mindestens 6 aber nicht mehr als 20 Ziffern umfassten, usw.

Beispielhaft seien nun zwei Umstellungsregeln vorgestellt:

Folie 12

WKN-Umstellungsregeln
<pre> level name PIC 9(count) ⇒ level name PIC X(count) 01 src PIC 9(d) ⇒ PIC X(d) 01 dst PIC 9(s) ⇒ PIC X(s) MOVE src TO dst ⇒ s = d: keine Änderung s < d: MOVE ZERO TO dst MOVE src TO dst(d-s+1:s) s > d: MOVE src(s-d+1:d) TO dst </pre> <hr/> <pre> level name PIC ±9(count) ⇒ level name level+1 name-SIGN PIC X(1) VALUE SPACE. level+1 name-VALUE PIC X(count). 01 src PIC 9(d) ⇒ ... 01 dst PIC +9(s) ⇒ ... MOVE src TO dst ⇒ s = d: MOVE src TO dst-VALUE MOVE SPACE TO dst-SIGN s < d: MOVE ZERO TO dst-VALUE MOVE src TO dst-VALUE(d-s+1:s) MOVE SPACE TO dst-SIGN s > d: MOVE src(s-d+1:d) TO dst-VALUE MOVE SPACE TO dst-SIGN </pre>

Neben den Umstellungsregeln für die PICTURE Klauseln, müssen natürlich auch die MOVE-Anweisungen umgestellt werden, da mit ZERO nicht der Standardwert SPACE für alphanumerische Variablen für WKN's benutzt wird.

In den Regeln stehen *count*, *d* und *s* für ganze Zahlen, die den Wiederholungsfaktor in der PICTURE-Zeichkette angeben.

Folie 13

WKN-Umstellung – Technische Probleme

- Umgang mit REDEFINES (statische Aliase)

```
01 WKN PIC 9(6).
01 WKN-X REDEFINES WKN X(6).
01 WKN-PART REDEFINES WKN.
  02 WKN-1-3 PIC 9(3).
  02 WKN-2-6 PIC 9(3).
```

```
01 memory PIC X(100).
01 DATA1 REDEFIENS memory.
  10 WKN PIC 9(6). ...
01 DATA2 REDEFIENS memory.
  10 XYZ PIC 9(6). ...
```

- Mehrdeutig benutzte Variablen

- Umgang mit falsch positiv erkannten WKN-Variablen

```
⇒ Ausschußliste der Art: *-WPK-SA
⇒ IGNORE-ALIAS: XYZ OF DATA2
⇒ Heuristiken (z.B. WKN hat zwischen 6 und 20 Bytes),
   aber: 175 verschiedene PICTURE Klauseln für WKN gefunden
```

- „Richtiges“ Einfügen der Anweisungen (terminierender Punkt).

- „Schönes“ Layout der erzeugten Texte.

Als Hauptproblem bei der Umstellung stellte sich der Umgang mit den sogenannten statischen Aliasen dar, d.h. mit Variablennamen, welche eine Speicherstelle mit unterschiedlichen Namen ansprechen. In COBOL wird dies durch die REDEFINES-Klausel ausgedrückt. Dieses Problem tritt natürlich nicht nur bei der WKN-Umstellung auf, sondern bei allen Transformationen, die etwas mit den Werten von Variablen zu tun haben.

Ursache des Problemes ist die Bedeutung des Aliases, er kann

1. dazu dienen, verschiedene „Sichten“ auf die Werte einer Variable zu bieten oder aber auch
2. dazu dienen, Speicherplatz zu sparen.

Im ersten Fall müssen alle Sichten als WKN-relevant betrachtet werden. Im zweiten Fall sollte es unterbleiben, ansonsten werden u.U. Variablen fälschlicherweise transformiert.

Uns ist keine Analyse bekannt, die diese beiden Fälle sicher unterscheiden kann. Aus diesem Grund waren die Transformationen nur erlaubt, wenn eine Variable keine Aliase hatte.

Weitere Probleme kamen durch „falsche“ Propagation zustande. Es wurde häufig eine Variable DRUCKZEILE benutzt, die zur Ausgabeformatierung diente. Die Propagation sorgte nun dafür, daß alle Variablen, welche die gleichen Speicherbereiche dieser Variablen beschrieben, als WKN-relevant markiert wurde. Dies war natürlich nicht immer erwünscht.

Als Lösung bot sich nur eine manuell erstellte Ausschlußliste an. Im Projekt wurden so ca. 3.400 COBOL-Namen, alle Variablen von 10 COPY-Strecken und ca. 420 statische Aliase ausgeschlossen.

Ein COBOL-spezifisches Problem ist die Handhabung des Satzende-Punktes. Falls der ursprünglichen Anweisung ein Punkt folgt, müssen die neuen Anweisungen unmittelbar vor dem Punkt eingefügt werden. Dabei soll natürlich ein „schönes“ Layout des Programmtextes erhalten bleiben.

Folie 14

Aufwand: Wertpapierkennnummern

Anzahl Programme:	15.000 (gesamt) 1.000 (WKN-relevant)
Anzahl COPY-Strecken:	30.000 (gesamt) 1.500 (WKN-relevant)
Anzahl Fundstellen:	800.000 (ohne Heuristiken) 300.000 (mit H.)
Lines Of Code:	2.300.000 (Programme), 300.000 (COPY's)
Ergebnisse:	- ca. 4.050 Umstellungen - keine Transformationsregel für ca. 5.900 Stellen - ca. 24.700 Warnungen
Geschätzter Aufwand:	vollständig manuelle Umstellung: 1,5 Tage / Programm 0,1 Tag / COPY-Strecke = ca. 1.700 Tage
Realer Aufwand:	halbautomatische Umstellung: 8 Personen, 5,5 Monate á 20 Tage + Werkzeug 200 Tage = 1080 Tage
Ersparnis:	⇒ ca. 600 Tage, d.h. ca 30% schneller/billiger

Folie 15

Lehren aus den bisherigen Projekten

- Nur ungenaue Spezifikation des Problems durch den Kunden.
- Spezifikationen ändern sich.
- Kunde: „Das Feature XYZ wird bei uns nicht benutzt.“
Erfahrung & Analyse: doch!
- Kundenspezifische Anpassung der Tools ist enorm wichtig.
- Verständigungsproblem: Compilerbauer ↔ COBOL-Programmierer

Wie wahrscheinlich in vielen anderen Projekten auch, ist zu Projektbeginn vieles unklar. Die Erstellung eines Prototyps hat im WKN-Projekt auch nur die Machbarkeit des automatischen Anteils der Umstellung aufgezeigt. Die eigentlichen technischen Probleme treten erst mit der Zeit auf¹. Auf der anderen Seite zeigte der Prototyp bereits den Projektverantwortlichen auf, daß Annahmen wie „es werden keine arithmetischen Operationen mit WKN's durchgeführt“ nicht stimmen. Es gab Programme, welche die WKN mit 100 multiplizierten und dann eine 2-stellige sogenannte „Sonderausstattungsnummer“ dazu addierten. Als Konsequenz daraus wurde eine „defensive“ Implementierung gewählt, getreu dem Motto: „alles was nicht explizit erlaubt ist, ist verboten“. Unter dieser Überschrift lassen sich auch die ca. 24.000 Warnungen verstehen, die das Umstellungswerkzeug erzeugt.

Im Laufe des Projektes wurden Teile des Werkzeuges immer wieder geändert, da sich mit zunehmender Erfahrung und Einsatz des noch nicht fertigen Werkzeuges, neue Anforderungen ergaben. Die schnelle Reaktion auf solche Änderungen war durch eine Vorort-Präsenz während der Entwicklung garantiert.

Auch über einige Verständigungsprobleme sollte man sich im Klaren sein. Für einen Compilerbauer ist ein Programmtext etwas gänzlich anderes als für einen COBOL-Programmierer. Ersterer denkt in abstrakten Syntaxbäumen, und was man damit machen kann, letzterer in Funktionalitäten, kennt seine E605-Datensätze in- und auswendig, und sieht COBOL-Programmzeilen. Von beiden bedarf es Lernaufwand, sich besser zu verstehen.

These: Vollautomatische Umstellung ist nicht möglich

Halbautomatische Umstellung bedeutet:

- Werkzeug liefert Analysedaten für manuelle Umstellung.
- Werkzeug transformiert in eindeutigen „Situationen“.
- Werkzeug markiert problematische Stellen zur manuellen Nachbearbeitung.

Korrolar:

Vollautomatische Umstellung ist für triviale Probleme möglich.

Folie 16

¹Dies ist sicherlich keine neue Erkenntnis.

Folie 17

Zusammenfassung

- Erfolg mit Techniken aus dem (optimierenden) Compilerbau.
- Halbautomatische Umstellung ist möglich.
- Vollautomatische Umstellung nur für triviale Probleme.
- Erstellung von Transformation-Werkzeug lohnt sich.

Folie 18

Ausblick

Aufgaben:

- (bessere) Erklärungskomponente für Analyseergebnisse
- (bessere) Visualisierung der Analyseergebnisse
- Programmübergreifende Analyse
- Entwicklung von Heuristiken, um komplexere Trafo's zu erlauben

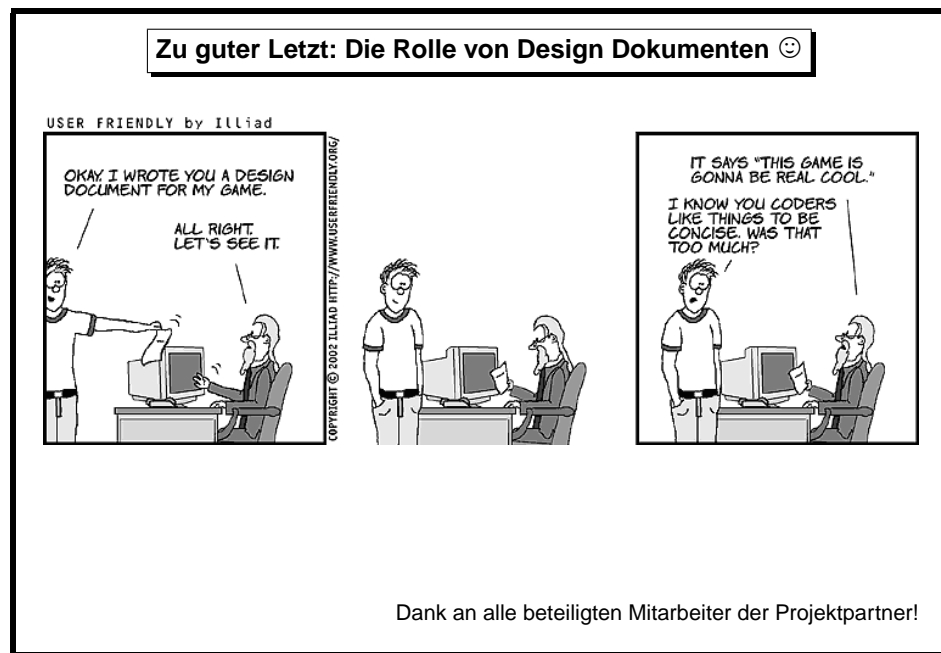
Während des Einsatzes des WKN-Umstellungswerkzeuges wurde ziemlich schnell klar, daß die Analyseergebnisse den Benutzern erklärt werden mussten. Die Benutzer waren nicht bereit, jedes Ergebnis ohne Grund zu akzeptieren¹. Deshalb wurde eine Erklärungskomponente erstellt, welche zu jeder als WKN-relevant markierten Variablen den Grund nannte, warum sie markiert wurde. Aus Zeitgründen wurde aber nur ein einfaches Programm geschrieben, welches aus der Debug-Ausgabe des Transformators eine HTML-Datei erzeugt. Diese Erklärungen waren manchmal sehr gut, manchmal aber zu umfangreich und umständlich (manuell liessen sich bessere Erklärungen finden).

Manche Analyseergebnisse lassen sich auch graphisch darstellen. Z.B. der CALL-Graph (welches Programm ruft welches andere auf). Einige Experimente mit einem Teil der 15.000 Hauptprogramme, die zur Verfügung standen, zeigten aber ziemlich schnell die Grenzen des benutzten Graph-visualisierers auf: der Graph war zu komplex (Laufzeit des Visualisierers) und nicht übersichtlich genug.

Die CALL-Graph-Analyse zeigt die Richtung: man möchte auch über die Programm-, Datei- und Datenbankgrenzen hinweg analysieren. Im WKN-Projekt wurde dies auch ansatzweise durchgeführt. Die Eigenschaft „Variable enthält WKN“ wurde auch über die CALL-Parameter hinweg propagiert. Noch weitergehende Analysen würden Dateien, Datenbanken, JCL usw. einbeziehen.

Mancher z.Z. vom Umsteller vorzugebender Eintrag in der Ausschlußliste liesse sich vielleicht auch über Heuristiken herleiten. Z.B. ob ein REDEFINES ignoriert werden soll, in dem man den sogenannten Programm-Slice für das Subjekt und Objekt des REDEFINES und deren Überschneidung bestimmt. Leider fehlte dazu die Zeit im Projekt.

Folie 19



¹Während der Entwicklung des Werkzeuges manchmal auch zurecht, denn nichts ist fehlerfrei.