Reverse Engineering und Wartung*

Zafer Öztürk
TR 93.31
Swiss Life, General Guisan-Quai 40,
8022 Zürich, zafer@swssai.uu.ch

Institut für Informatik der Universität Zürich,
Winterthurerstr. 190, 8057 Zürich, oeztuerk@ifi.unizh.ch

1. Einleitung

In vielen Betrieben sind heute Applikationen im Einsatz, die über 15 Jahre, einige sogar über 20 und mehr Jahre alt sind. Die meisten solcher Applikationen sind zudem Stützpfeiler in der computergestützten Bearbeitung des Tagesgeschäfts – einer der Hauptgründe, warum sie so alt wurden und warum sie häufig trotz ihres Alters am Leben erhalten werden müssen. Die Wartung hat die Aufgabe, solche Applikationen ständig neuen Anforderungen und neuen Umgebungen (Hardware, Betriebssystem, . . .) anzupassen. Dabei muss sie mit unterschiedlichen Schwierigkeiten kämpfen:

- Veraltete Qualitätsanforderungen
  Mit den Einschränkungen früherer Hardware war es unerlässlich, den Programmtext so kurz wie möglich zu halten (Kampf um jedes Byte), die verwendeten Datenstrukturen zu komprimieren (Kampf um jedes Bit) und die Ausführungszeit zu reduzieren (Kampf um jede Instruktion). Kein Wunder also, dass die Programme wenige oder überhaupt keine Kommentare enthalten, sehr komplexe, meist mehrfach benutzte Datenstrukturen aufweisen und durch Ausnutzung von "Spezialitäten" der Hardware schwer lesbar und auch wenig portabel sind.
- Dokumentation der Applikationen
  Die Dokumentation der Programme, welche vielerorts nicht einmal maschinenlesbar vorliegt, hinkt immer dem aktuellen Stand des Programms hinterher. Es ist nicht auszuschliessen, dass Änderungen nicht oder nur unvollständig nachgetragen werden. Ein Hauptgrund ist sicher der Zeitdruck, unter dem die Anpassungen am Programm gemacht werden müssen. Aber auch die fehlende Einsicht der Entscheidungsträger und die mangelnde Disziplin der Programmierer für diese scheinbar unproduktive und nicht attraktive Arbeit lässt mancherorts die Dokumentation der Programme zur Farce werden.
- Verlust von Know-How
  Durch die Personalfluktuation in den EDV-Abteilungen geht mit den Angestellten auch häufig das Know-How verloren. Zum anderen wurden und werden immer noch wichtige Applikationen, oder zumindest Teile davon, von externen Mitarbeitern implementiert. Was zur Folge hat, dass mit der Beendigung ihrer Arbeit und ihrem Abgang, auch ihr gewonnenes Know-How für die Firma verloren ist.

---

* Dieser Bericht erscheint in identischer Form im Juli ’93 auch als Bericht 4/93 der Informatik-Forschungsgruppe der Rentenanstalt / Swiss Life.

Vor diesem Hintergrund ist es nicht erstaunlich, dass die Motivation in der Forschung im Bereich der Wartung, und dort vorwiegend in der Rückgewinnung von Informationen aus Programmtexten, in den letzten Jahren stetig zugenommen hat.

Im vorliegenden Bericht wird im Kapitel 2 zunächst die Wartungsproblematik aufgezeigt, vor der viele Firmen stehen. In den darauf folgenden Kapiteln werden nacheinander drei verschiedene Lösungsansätze mit ihren Hauptmerkmalen beschrieben. Als erstes wird der Programmer's Apprentice vorgestellt, welcher sich auf Programmuster abstützt (Kap. 3). Als zweites wird der Maintainer's Assistant gezeigt, welcher auf Transformationen am Programmtext basiert (Kap. 4). Als dritten und aktuellsten Ansatz wird das REDO-Projekt beschrieben, in welchem man versuchte, die zu wartenden Programme in die Spezifikationssprache Z zu übersetzen (Kap. 5). Kapitel 6 enthält eine Zusammenfassung und den Ausblick.

2. Die Wartungsproblematik

Viele in einer Organisation eingesetzten Programme sind schon recht alt. Durch Wartungseingriffe während all den Jahren hat sich die zugrundeliegende Architektur des Systems immer mehr verwischt. Zusätzlich werden die Programme immer unstrukturierter und dadurch schwerer lesbar.

Die Eingriffe müssen zudem meist unter Zeitdruck durchgeführt werden, so dass die Änderungen nicht immer sorgfältig genug ausgearbeitet werden. Es kann durchaus vorkommen, dass zusätzliche Fehler durch Unachtsamkeit eingeführt werden. Dies geschieht im Bestreben, eine Anpassung möglichst schnell und mit geringem Aufwand durchzuführen. Die eingeschleppten Fehler machen sich irgendwann später im Betrieb bemerkbar und bedeuten zusätzlichen Aufwand in der Wartung, der hätte vermieden werden können.

Die schwere Lesbarkeit von Programmen rührt auch daher, dass beim Implementieren des Programms und/oder bei dessen Wartung z.T. nichtssagende Namen als Bezeichner gewählt werden (z.B. "ADD C67 TO G3", "PERFORM H01 THRU B56"). Schlecht lesbare Programme sind mit Sicherheit auch schwierig zu verstehen, und ohne Verständnis der Funktionalität eines Programms sind Anpassungen nicht fehlerfrei durchführbar. Wir werden sehen, dass das Verstehen der Funktionalität eigentlich das Hauptproblem in der Wartung ist.

Der Zeitdruck in der Wartung entsteht mehrheitlich dadurch, weil die auftauchenden Fehler nicht vorhersagbar sind, und trotzdem so rasch wie möglich behoben werden müssen. Die Aufwände für die Behebung von Fehlern und von Anpassungen an den Programmen sind nicht leicht bestimmbar. Für die Entscheidungsträger in der Wartung stellt sich somit das Problem, mit nicht planbaren Grössen arbeiten zu müssen.

Eine Möglichkeit, diese "Unplanbarkeit" in den Griff zu bekommen, wäre eine kontinuierliche Wartung der Programme. Dies bedeutet, dass Anpassungen an Programmen vorgenommen werden, ohne dass Fehler im Betrieb aufgetaucht sind. Dieser Art der Wartung wird aber nicht genug Beachtung geschenkt.

Da die Wartung unattraktiv ist und kein Prestige damit verbunden ist, sind die qualifizierten Programmierer eher in der Entwicklung von Neuanwendungen zu finden. So werden in vielen Betrieben die Programme weiterhin durch z.T. unerfahrene Programmierer geflickt und notdürftig in Betrieb gehalten. Das vermindert die Qualität der Programme zusätzlich.

Wie in Abb. 1 zu sehen ist, sind die Aufgaben in der Wartung unterschiedlicher Natur. Mit fast 80% stehen vorwiegend die Adaptionen an neue Umweltbedingungen, neue Benutzerwünsche oder an neue Hard- und Software im Vordergrund.

Analysiert man die Tätigkeiten weiter, sieht man, dass vorwiegend das Verstehen des Programms mit 50% des Aufwandes sehr viel Zeit kostet (vgl. Abb. 2). Vom Rest der Zeit fliessen 25% in die Tests der geänderten Programmteile. Diese nackten Zahlen gewinnen noch mehr an Gewicht, wenn man in Betracht zieht, dass der Aufwand für die Wartung am Gesamtaufwand der EDV-Abteilungen, je nach Autor und befragten Firmen, zwischen 50% und 90% beträgt.

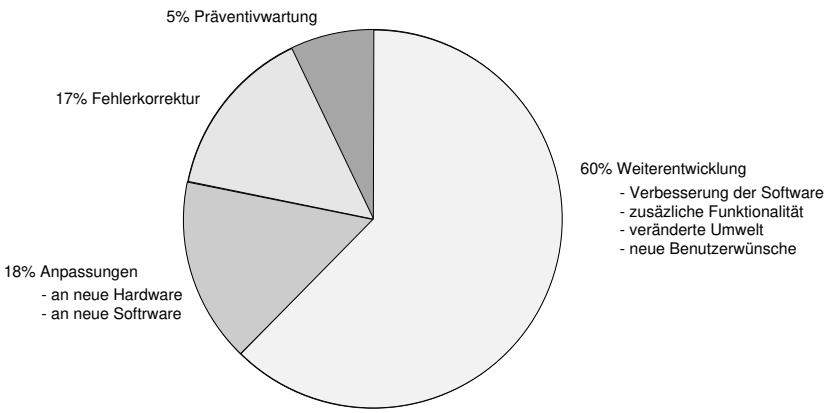


Abbildung 1: Aufgaben in der Software-Wartung (nach [Fra92])

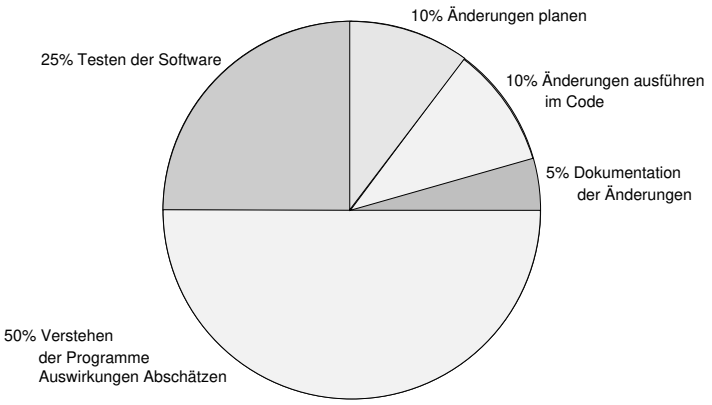


Abbildung 2: Tätigkeiten in der Software-Wartung (nach [Fra92])

Die eben gezeigten Aufwände in der Wartung behindern stark die Neuentwicklung von Software. Die Wartung bindet einen Grossteil der personellen Resourcen in der EDV. Die Folge ist die allgegenwärtige Software-Krise! Die Kapazitäten der EDV-Abteilungen fliessen in die Wartung von überalterten Programmen statt in die Neuentwicklung von Produkten.

Es gibt verschiedene Ansätze, dieses Problem anzugehen. Man kann schon bei der Entwicklung Methoden einsetzen, welche die Wartbarkeit der Programme gewährleisten (siehe Programmer's Apprentice in Kap. 3). Ein weiterer Ansatz ist das Unterstützen des Wartungsprogrammierers beim Verstehen und beim Modifizieren eines Programmes (siehe Maintainer's Assistant in Kap. 4). Oder man versucht aus dem Programm eine Spezifikation herauszuholen, sie in einer dazu geeigneten Notation darzustellen und aus dieser Spezifikation ein neues Programm zu generieren (siehe REDO-Projekt in Kap. 5).

3. Programmer's Apprentice

3.1. Herkunft

Der Programmer's Apprentice wurde am Massachusetts Institute of Technology (M.I.T) seit ca. 1976 entwickelt. Es wurden bereits einige Prototypen implementiert. Autoren sind unter anderern R.C. Waters und Ch. Rich.

3.2. Grundideen

Das Ziel dieses Projekts ist es, die Arbeit eines erfahrenen Programmierers (Experte) zu verstehen und Teile davon zu automatisieren. Dabei interessieren folgende Fragen:

- Wie spezifiziert man ein Softwaresystem?
- Wie geht man bei dessen Entwicklung vor?
- Wie analysiert man ein Programm und modifiziert es?
- Was sind die Vorgehensweisen beim Verifizieren von Programmen?
- Wie beschreibt man das Programm in der Dokumentation?

Die Liste solcher Fragen ist beliebig lang. Angesichts dieses umfangreichen Fragenkatalogs hat man sich dazu entschlossen, ein System zu entwerfen und zu implementieren, das den Programmierer in allen Stadien der Softwareentwicklung unterstützt. Damit das System (Programmer's Apprentice) bestmöglichst eingesetzt werden kann, muss es kompetente Entscheidungen treffen können, die eine gewisse Intelligenz innerhalb des Systems erfordern.

Man verfolgt vor allem zwei Richtungen in diesem Projekt: zum einen den Assistentenansatz und zum anderen den Inspektionsansatz. Ersteres bedeutet, wie oben schon erwähnt, dass dem Programmierer ein kompetenter Assistent zur Seite gestellt wird. Zweiteres bedeutet, dass dem Programmierer Hilfen zur Verfügung gestellt werden, die es ihm ermöglichen, effizient und einfach in einer Wissensbank abgelegte Programmuster zu inspizieren und ihre Anwendbarkeit für seine Aufgabe zu bestimmen. Auf den Begriff des Programmusters wird später noch näher eingegangen. Zur Realisierung dieser Ansätze stehen auf der technischen Seite zwei Methoden im Vordergrund: der Plan-Calculus und ein hybrides Reasoning-System, genannt Cake.

Der Entscheid, einen intelligenten Assistenten zu bauen, kam dadurch zu stande, weil es noch nicht möglich ist, den Prozess des Programmierens zu automatisieren (ausser in einfachen Fällen). Dieser Assistent ist als ein weiteres Mitglied in einem Projektteam anzusehen, das einem erfahrenen Programmierer unterstellt ist. Im besten Fall würde ein Experte nur diesen einen Assistenten benötigen, der ihm die Routinearbeit abnimmt, damit er sich selber den eigentlichen Problemen widmen kann.

Die wesentlichen Komponenten des Programmer's Apprentice werden im folgenden Abschnitt näher erklärt.

3.3. Komponenten in diesem System

- Inspektionsmethode

  Ein erfahrener Programmierer denkt in Mustern von ganzen Programmen oder Teilen davon, die er aus seiner Erfahrung kennt (siehe [RW88]). Selten wird er ein Programm von Grund auf neu entwickeln. In Anlehnung an diese Erkenntnis hat man sich entschieden, eine Möglichkeit vorzusehen, in einer Wissensbank nach nützlichen Programmustern für eine bestimmte Aufgabe suchen zu können. Dies ist einfach und effizient, aber liefert nicht immer eine Lösung, da die Programmuster in der Wissensbank nicht alle Anforderungen in der Realität abdecken können. Die Methode der Inspektion ist darum auf eine grosse Menge von Programmustern angewiesen, will sie in vielen Fällen erfolgreich sein. Es existieren auch allgemeinere Methoden (deduktive Synthese, Programmtransformation). Sie sind zwar mächtiger, aber schwieriger zu handhaben (siehe [RW88]).
- Plan-Calculus

  Ausführlicher wird der Plan-Calculus in Kapitel 3.4 beschrieben. Hier soll er nur kurz skizziert werden. Im Plan-Calculus werden die Programme und Programmuster formal repräsentiert. Er kann als Kombination von Flussdiagrammen, Datenschemata und abstrakten Datentypen angesehen werden. Der Plan-Calculus abstrahiert von allen programmiersprachenspezifischen Details. Beispiele von solchen Details wären die unterschiedlichen Schreibweisen von Fallunterscheidungen in Lisp, Ada oder C.
- Cake – ein hybrides Reasoning-System

  Cake vereinigt verschiedene Darstellungsformate und zugehörige Inferenzmethoden (siehe Abb. 3). Das hybride Wissensrepräsentations-System Cake besteht aus vier Schichten. Jede Schicht realisiert ihre Funktionalität mit Hilfe der darunter liegenden Schichten.

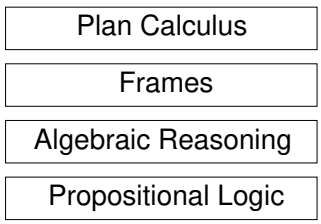


Abbildung 3: Schichten der hybriden Wissensrepräsentation in Cake ([RW88])

1. Die Schicht der *Propositional Logic* (vgl. Abb. 3)
   Die Möglichkeiten dieser Schicht (Aussagenlogik) umfassen das deduktive Schliessen, das Aufzeichnen von Abhängigkeiten, wie es bei einem Truth-Maintenance-System (TMS) üblich ist, und das Aufdecken von Widersprüchen.
2. Die Schicht des *Algebraic Reasoning* (vgl. Abb. 3)
   Die Möglichkeiten dieser Schicht umfassen das Überprüfen auf Kongruenz von Termen durch Substitution (was sehr oft vom Plan-Calculus verwendet wird, um Gleichheit zu prüfen), das Ausnützen von Eigenschaften der algebraischen Operatoren (Kommutativität, Transitivität und Assoziativität), die Behandlung von partiellen Funktionen und das Anwenden der Mengenalgebra.
3. Die Schicht der *Frames* (vgl. Abb. 3)
   Die Möglichkeiten dieser Schicht umfassen die objekt-zentrierte Repräsentation,

die Vererbung von Eigenschaften und das Instanziieren von Objekten. Die Wissensbank der Programmuster verwendet den Vererbungsmechanismus dieser Schicht sehr intensiv.

4. Die Schicht des *Plan Calculus* (vgl. Abb. 3)
   Die Möglichkeiten dieser Schicht umfassen die graphentheoretischen Manipulationen von Programm-Plänen und deren Präsentation als Funktionen. So werden z.B. Namen von Plänen als Prädikatsymbole und Rollennamen/Overlays als Funktionssymbole interpretiert. Entsprechend werden auch die Korrespondenzen, Datenflüsse und die Kontrollflüsse umgesetzt.

3.4. Programm-Pläne und der Plan-Calculus

3.4.1 Begriffe

Der Plan-Calculus ist ein Formalismus zur Repräsentation von Programmen, Programmmustern und Beziehungen zwischen den Programmustern. Programme und Programmuster werden als Pläne dargestellt. Die Relationen zwischen den Programmustern werden durch Overlays dargestellt [Wil90].

Ein *Plan* repräsentiert die Operationen eines Programms, sowie die Datenflüsse, Kontrollflüsse und ihre Beziehungen zueinander. Jeder Plan ist ein gerichteter, azyklischer Graph, in dem die Knoten Operationen, die ausgezogenen Pfeile die Datenflüsse und die gestrichelten Pfeile die Kontrollflüsse darstellen.

Ein *Programmuster* ist eine frameartige Repräsentation. Seine Teile werden in Rollen (roles), die für Attribute stehen, und in Relationen (restrictions) zwischen den Rollen eingeteilt.

Die *Overlays* setzen unterschiedliche Programmuster in Beziehung. Durch sie wird es möglich, unterschiedliche Implementationen einer bestimmten Programmspezifikation zuzuordnen. Dabei steht ein Overlay für eine Beziehung zwischen einer Spezifikation und einer zugehörigen Implementation.

3.4.2 Eigenschaften

Der Plan-Calculus bietet folgende Vorteile (aus [RW90]):

- *Unabhängigkeit von Programmiersprachen:* Die Darstellung von allgemeinen Programmkonstrukten (z.B. Selektion, Iteration) ist für alle Sprachen gleich.
- *Additivität:* Programmuster sind leicht und einfach miteinander kombinierbar. Durch die Wiederverwendung von Programmustern aus einer Datenbank verringert man den Programmieraufwand.
- *Verifizierbarkeit:* Der Plan-Calculus als formale Sprache erleichtert es, Inkonsistenzen in den Definitionen der Pläne zu lokalisieren.
- *Multisichten:* Mit Hilfe von Overlays ist es möglich, verschiedene Abstraktionsebenen darzustellen.

3.4.3 Bausteine und ihre Darstellungen

Pläne werden aus den drei folgenden Primitiven konstruiert [Ric81]:

- *Operation:*
  Mit diesem Primitiv werden die Eingabe- und Ausgabegrössen eines Berechnungsblocks spezifiziert (Input/Output-Spezifikation). Es kann als Blackbox angesehen werden, bei welcher nur die Ein- und Ausgabetypen interessieren (vgl. Abb. 4).
- *Test:*
  Mit diesem Primitiv lassen sich logische Ausdrücke im Programm spezifizieren. Je nach der Eingabebedingung *criterion* und den Eingabewerten *input* wird *success* oder *fail* als Ausgabe geliefert (vgl. Abb. 5).
- *Einfaches Datenobjekt:*
  Mit diesem Primitiv lassen sich Daten (Zahlen, Mengen, . . .) spezifizieren (vgl. Abb. 6).

Zur Beschriftung der Diagrammelemente:

- <Rolle>:<ZulässigerTyp>.
- Fehlt die Angabe <ZulässigerTyp>, ist jeder Typ zugelassen.
- Ausgezogene Pfeile stellen Datenflüsse dar.
- Gestrichelte Pfeile stellen Kontrollflüsse dar.
- “S” steht für *success*, und “F” steht für *fail*.

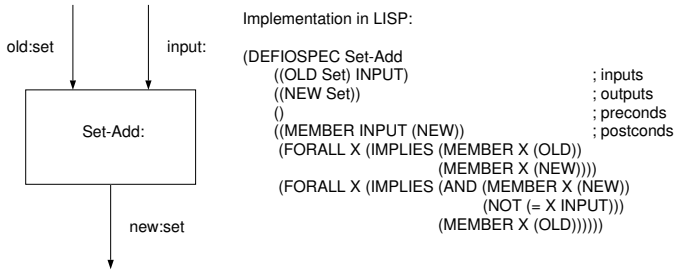


Abbildung 4: Der Plan eines Programmusters, das für eine Operation steht

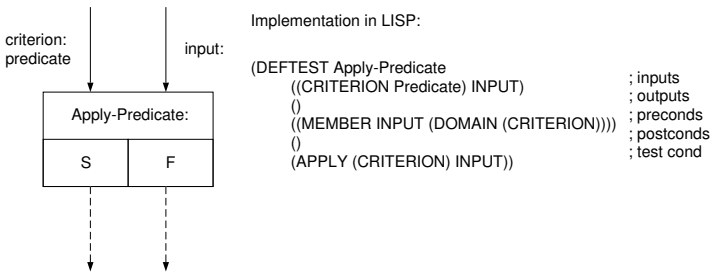


Abbildung 5: Der Plan eines Programmusters, das für einen Test/Verzweigung steht

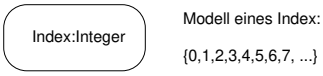
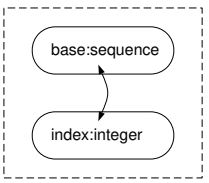


Abbildung 6: Der Plan eines Programmusters für ein einfaches Datenobjekt

3.4.4 Zusammengesetze Pläne

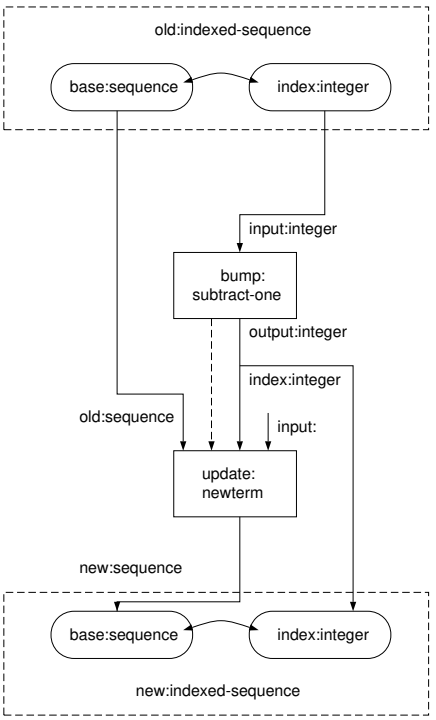
Es lassen sich zwei Klassen von Plänen unterscheiden. Zum einen die *Daten-Pläne* (vgl. Abb. 7), welche die passiven Teile eines Programms darstellen (z.B. Datenstrukturen), und zum anderen die *temporalen Pläne* (vgl. Abb. 8), welche die aktiven Teile eines Programms darstellen (z.B. Anweisungen, Verzweigungen, Prozeduren). Die Rollen inerhalb von *Daten-Plänen* enthalten ausschliesslich einfache Datenobjekte oder andere Daten-Pläne. Die Rollen der *temporalen Pläne* können Pläne unterschiedlicher Typen enthalten (Operationen, Tests, einfache Datenobjekte). Teil-Pläne werden in den Abbildungen als gestrichelte Kästchen, Relationen mit Doppelpfeilen dargestellt.



Implementation in LISP:

```
(DEFDATAPLAN Indexed-Sequence
      ((BASE Sequence) (INDEX Integer))     ; roles
      ((GT (INDEX) 0)                       ; constraints
       (LE (INDEX) (LENGTH (BASE)))))
```

Abbildung 7: Ein Daten-Plan, aufgebaut mit Programmustern von einfachen Datenobjekten



Implementation in LISP:

```
(DEFPLAN Bump+Update
      ; roles
      ((OLD Indexed-Sequence) (NEW Indexed-Sequnce)
       (BUMP Subtract-One) (UPDATE Newterm))
      ; constraints
      ((CFLOW (OUT (BUMP)) (IN (UPDATE)))
       (= ((INDEX (OLD))) ((INPUT (BUMP))))
       (= ((BASE (OLD))) ((OLD (UPDATE))))
       (= ((OUTPUT (BUMP))) ((INDEX (UPDATE))))
       (= ((OUTPUT (UPDATE))) ((BASE (NEW))))
       (= ((OUTPUT (BUMP))) ((INDEX (NEW))))))


(DEFIOSPEC Subtract-One
      ((INPUT Integer))
      ((OUTPUT Integer))
      ()
      ((= (OUTPUT) (- (INPUT) 1))))

(DEFIOSPEC Newterm
      ((OLD Sequence) (INDEX Integer) INPUT)
      ((NEW Sequence))
      ((GT (INDEX) 0) (LE (INDEX) (LENGTH (OLD))))
      ((= (LENGTH (NEW)) (LENGTH (OLD)))
       (= (TERM (NEW) (INDEX)) INPUT)
       (FORALL I
         (IMPLIES (NOT (= I (INDEX)))
           (= (TERM (OLD) I) (TERM (NEW) I))))))
```

Abbildung 8: Ein temporaler Programm-Plan (Bump+Update), aufgebaut aus unterschiedlichen Programm-Plänen

3.4.5 Beziehungen zwischen Plänen

Pläne können auf unterschiedliche Weise in gegenseitiger Beziehung stehen. Durch Hinzufügen weiterer Restriktionen zu einem bestehenden Plan entsteht ein neuer Plan, der zum alten in einer *Spezialisierungsbeziehung* steht. Ein Plan gilt als *Erweiterung* eines anderen, wenn er zusätzliche Rollen (und evtl. weitere Restriktionen) zu denen des anderen aufweist. Ferner besteht zwischen dem Plan und den in ihm enthaltenen Plänen die *Aggregationsbeziehung*. Der wichtigste Beziehungstyp zwischen Plänen ist der *Overlay*. Dabei werden zwei unterschiedliche Sichten auf ein gleiches Programm, welche durch

unterschiedliche Pläne repräsentiert werden, miteinander in Beziehung gesetzt (Abb. 9). Die eine Sicht ist z.B. auf der Designebene und die andere eine mögliche Realisierung auf der Implementationsebene. Ein Overlay ist somit eine Relation (1–1 oder 1–m) zwischen verschiedenen Plänen, wo die jeweils miteinander korrespondierenden Teil-Pläne einander zugeordnet sind (in Abb. 9 durch gestrichelte Doppelpfeile dargestellt).

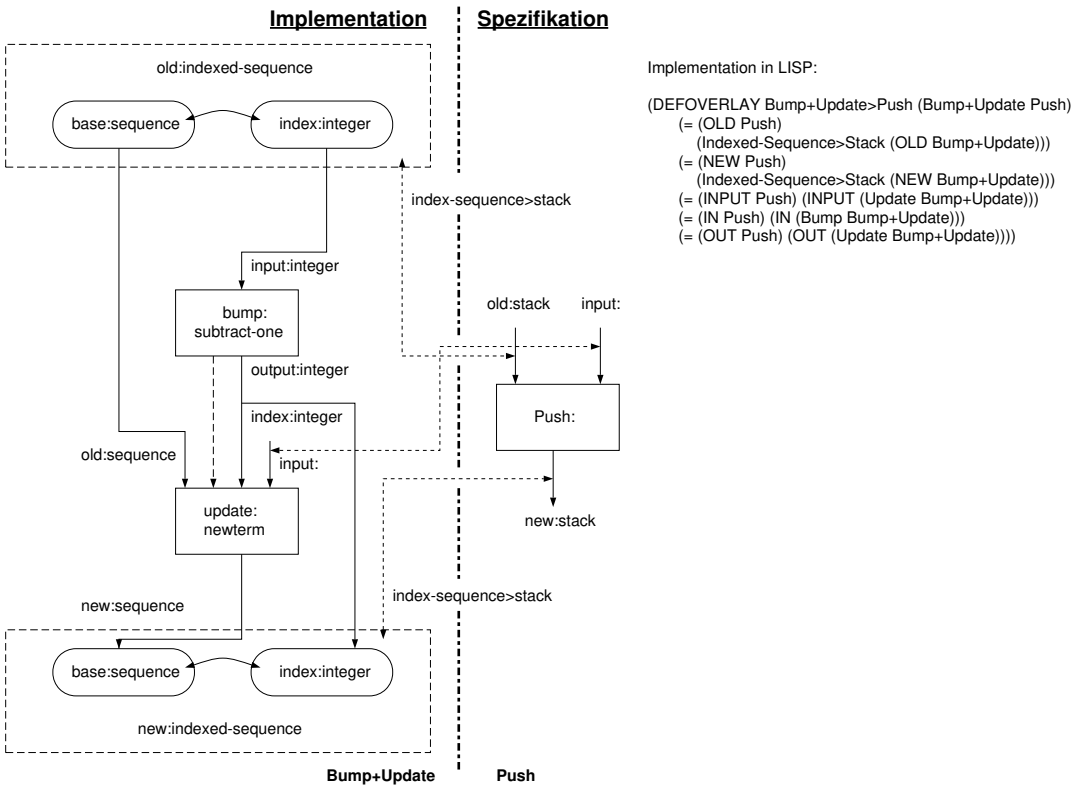


Abbildung 9: Overlay zwischen zwei Programm-Plänen

3.4.6 Nachteile des Plan-Calculus

- Es ist eine umfangreiche Datenbank von Programmustern nötig, um ein realistisches Programm repräsentieren zu können. Schon für kleine Prozeduren braucht man mehr als zehn verschiedene Programmuster.
- Die fehlende Effizienz dieses Ansatzes kommt daher, dass bei der Anwendung des Plan-Calculus sehr viele Programmuster betrachtet und getestet werden müssen.
- Der nicht-lokale Kontrollfluss (typischerweise jede Fehlerbehandlung) ist schlecht darstellbar.
- Programme, die über eine gemeinsame, globale Datenbank kommunizieren, können schlecht repräsentiert werden, weil ihr Datenfluss nicht gut im Plan-Calculus zerlegt werden kann.

3.5. Möglichkeiten des Programmer's Apprentice

Die folgenden Punkte charakterisieren den Ansatz des Programmer's Apprentice in Stichworten.

- Der Programmer's Apprentice kann z.B. Modifikationen des Programmierers am Programm erledigen, unter Einhaltung der Konsistenz solcher Modifikationen (z.B. Umbenennen einer Variablen im ganzen Programmtext).
- Der Programmer's Apprentice begründet seine Entscheidungen und beschreibt, wie er zu einer bestimmten Entscheidung gelangt ist, wenn der Programmierer dies wünscht.
- Der Programmierer kann die zu erstellende Software auf der Algorithmusebene beschreiben, ohne ihn ausprogrammieren zu müssen. Er spezifiziert statt implementiert.
- Die Interaktionen zwischen dem Programmierer und dem Programmer's Apprentice finden auf der Design- und Requirementsebene statt, und nicht auf der Implementationsebene.
- Der Programmer's Apprentice ist unabhängig von der Programmiersprache, in der die Software schlussendlich implementiert werden soll.
- Der Programmierer hat die Möglichkeit, mit anderen Tools auf den Programmtext direkt zuzugreifen und diesen zu verändern.
- Einfaches Erweitern der Bibliothek von Programmustern durch den Programmierer ist möglich.
- Die Entwicklung von Software ist schneller und zuverlässiger, da man schon vorhandene Programmuster oder Teile leichter wiederverwenden kann.
- Die Fähigkeit zum Schlussfolgern anhand vorgegebener Bedingungen und Einschränkungen zwischen Teilen von Programmustern spart unnötige Rückfragen des Programmer's Apprentice an den Programmierer.
- Der Programmer's Apprentice kann einfache Programmdokumentationen, basierend auf den verwendeten Programmustern, automatisch generieren.
- Der Programmer's Apprentice hat die Möglichkeit, Widersprüche und eventuell Fehler in den Eingaben des Programmierers zu lokalisieren und kann Hinweise geben, was nicht korrekt ist.
- Das Repertoire von Begriffen aus dem Software-Engineering, das Benutzer und Programmer's Apprentice teilen, trägt dazu bei, Missverständnisse zu umgehen.

3.6. Realisierung der Ideen

Es wurde ein Prototyp entwickelt, um die Brauchbarkeit der Ideen zu zeigen [Wat85]. Er wurde als eine Erweiterung von Emacs implementiert. Daher kommt auch sein Name KBEmacs (Knowledge-Based Editor in Emacs).

Der Programmer's Apprentice kann Mutationen nur auf dem Plan des Programms ausführen. Dagegen kann der Programmierer sowohl auf dem Text als auch auf dem Plan des Programms seine Mutationen anbringen.

Immer wenn der Programm-Plan vom Programmierer oder dem Programmer's Apprentice geändert wird, erzeugt der *Coder* (Abb. 10) den entsprechenden Programmtext dazu. Diese Transformation ist relativ einfach. Der grössere Aufwand entsteht dort, wo der Text gut lesbar dargestellt werden muss.

Immer wenn der Programmtext vom Programmierer direkt geändert wurde, versucht der Analyzer, den entsprechenden Programm-Plan dazu zu generieren. Dabei können Schwierigkeiten auftreten. Der *Analyzer* (Abb. 10) kann evtl. nicht mehr alle Programmuster bestimmen, die für die Repräsentation des Programms nötig wären. Sei es, weil ein neues, für den Programmer's Apprentice unbekanntes Programmuster verwendet wurde, oder sei es, weil die direkte Manipulation des Programmtexts ein verwendetes Programmuster so verfremdet hat, dass es für den Programmer's Apprentice nicht erkennbar ist. Damit ist die Abbildung des ganzen Programmtexts in Programmuster in Frage gestellt. Zusätzlich wird die Möglichkeit des Programmer's Apprentice, selbständige Aktionen auszuführen und den Programmierer zu beraten, eingeschränkt.

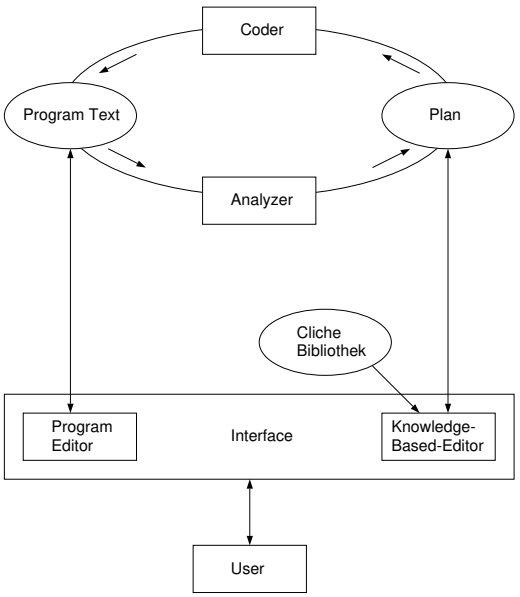


Abbildung 10: KBEmacs als ein Prototyp des Programmer's Apprentice, der einige Möglichkeiten dieses Ansatzes aufzeigt [Wat85].

3.7. Nutzen und Aussichten

Der Programmer's Apprentice wurde auf verschiedene Aufgaben des Software-Engineering angewandt und seine Tauglichkeit aufgezeigt.

- So wurde an einem bestehenden Programm mit Hilfe einer Bibliothek von Programmustern und des Plan-Calculus ein Reverse Engineering durchgeführt, um eine Spezifikation des untersuchten Programms zu erhalten [Wil90]. Da die Abbildung eines Programms in eine Spezifikation nicht eindeutig ist, kann man nur eine plausible Spezifikation und nicht die ursprüngliche Spezifikation aus dem Progammtext ableiten.
- Es wurde ein auf dem Plan-Calculus basierendes, intelligentes Debuggingtool entwickelt, das dem Programmierer hilft, Fehlerquellen im Programmtext schneller zu finden [Kup88].
- Es existieren mittlerweile kommerzielle Tools von *Bachman Information Systems* und *Sanders Associates*, die auf den Ideen des Programmer's Apprentice aufgebaut sind.

Man ist nun daran, einen Requirements Apprentice und einen Design Apprentice zu entwickeln, so dass der gesamte SW-Entwicklungsprozess von "intelligenten Assistenten" begleitet werden kann.

4. Maintainer's Assistant

4.1. Herkunft

Der Maintainer's Assistant [War93] ist ein Werkzeug im Rahmen des Projekts REFORM (Reverse Engineering Using Formal Methods). Die Projektpartner sind die University of Durham, das Centre for Software Maintenance und die IBM (UK). Das Projekt läuft seit ca. 1988. Daran beteiligte Personen sind unter anderen M. Ward, M. Munro, F.W. Callis, H. Yang und T.Bull.

4.2. Grundidee

Der Maintainer's Assistant ist als ein interaktives Werkzeug zur Wartung von Programmen konzipiert worden. Unter Ausnützung formaler Methoden [CKMW88, WCM89] kann dieses Werkzeug mithelfen, alten Quellcode zu analysieren und zu modifizieren. Dabei spielt das Neustrukturieren von Sourcen eine untergeordnete Rolle gegenüber dem Verstehen der Funktionsweise des Programms. Die Dokumentation der Programme, die die Funktion beschreiben sollte, ist in vielen Fällen nicht vorhanden oder wurde seit langem nicht mehr nachgeführt, so dass sie den aktuellen Stand der Software nicht mehr widerspiegelt. Aus diesem Grund stützt sich der Maintainer's Assistant direkt und ausschliesslich auf den Quellcode ab [Bul90, Yan91].

4.3. Möglichkeiten des Maintainer's Assistant

Der Maintainer's Assistant ist sprachunabhängig konzipiert, so dass Programme in unterschiedlichen Programmiersprachen behandelt werden können. Erreicht wird diese Unabhängigkeit durch die Abbildung des Programmtexts in eine Zwischensprache WSL (Wide-Spectrum-Language) [Yan91]. Für jede Programmiersprache, die man behandelt haben will, müssen Übersetzer in WSL existieren. Auf diese Weise konnten die Operationen zur Modifikation und Synthese, zur Adaption und Verifikation von Programmen allgemein definiert werden, ohne die Details der jeweiligen Programmiersprache betrachten zu müssen.

Der Maintainer's Assistant verwendet Transformationen, um ein Programm in ein neues Programm zu überführen (Abbildung eines Programms in ein anderes). Transformationen können einfache Operationen sein (Löschen von Text, Umstellen von Anweisungen innerhalb eines Programmes). Die Korrektheit solcher einfachen Transformationen müssen zunächst formal bewiesen weden, was z.T. sehr schwierig sein kann. Sie stellen dann die Grundbausteine für weitere, komplexere Transformationen dar [Bul90]. Da die komplexen Transformationen aus korrekten, schon bewiesenen Transformationen bestehen, ist der Beweis ihrer Korrektheit weniger aufwendig, aber immer noch sehr anspruchsvoll. Jede Transformation, die als korrekt bewiesen wurde, kann wiederum zum Aufbau von weiteren, noch komplexeren Transformationen verwendet werden.

Die Erweiterung einer Datenbank von Tranformationen aus schon vorhandenen Transformationen ist möglich. Durch Anwendung von Transformationen aus dieser Datenbank kann die semantische Äquivalenz eines transformierten Programms gegenüber dem ursprünglichen Programm garantiert werden. Darum ist es mit Hilfe des Maintainer's Assistant

möglich, grosse Programme umzustrukturieren, ohne zusätzliche Fehler durch die Transformationen einzuschleppen.

Weitere Möglichkeiten des Maintainer's Assistant sind (aus [WCM89]):

- Ableiten einer Spezifikation aus einem Programmtext
- Bearbeiten von separaten Teilen aus dem Programmtext
- Aufzeigen des Kontroll- und Informationsflusses
- Aufdecken und Darstellen der Programmstruktur
- Bestimmen des Gültigkeitsbereichs von Variablen und ihrer Verwendung
- Nachdokumentieren von Programmen [CKMW88]

4.4. Komponenten des Systems

Die Bearbeitung eines Programmtexts geschieht in den drei Phasen *Translation*, *Transformation* und *Integration*.

Phase 1: Translation
Zunächst wird der Programmtext in einfache WSL-Konstrukte übersetzt (Abb. 11). Der Wartungsprogrammierer hat durch den *Browser* die Möglichkeit, auf das System Einfluss zu nehmen. Mit Hilfe des *Program-Slicers* wird die Repräsentation des Programmtexts in kleinere Einheiten zerlegt. Am Ende dieser Phase werden die Daten in der Datenbank abgelegt.

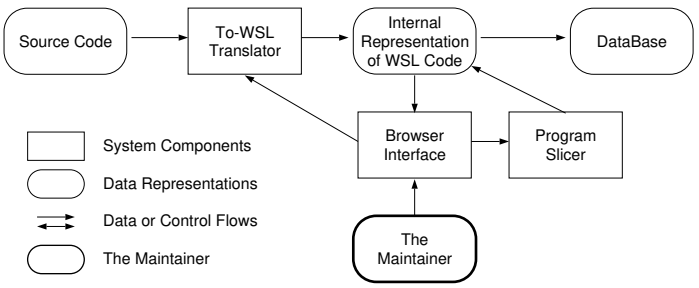


Abbildung 11: Vom Programmtext zu einfachen WSL-Konstrukten (nach [Yan91])

Phase 2: Transformation
Zur Bearbeitung eines Programmteils wird dieser aus der Datenbank geholt (vgl. Abb. 12). Durch den *Structure Editor* kann der Wartungsprogrammierer erkannte Fehler direkt im Programm korrigieren, allerdings mit der Gefahr, neue Fehler einzuführen, da keine als korrekt bewiesene Transformationen aus der Datenbank von Transformationen verwendet werden. Der *Program Transformer* ist interaktiv. Er zeigt den Programmtext an und bietet dem Benutzer alle darauf anwendbaren Transformationen an, aus denen dieser eine zur Ausführung bringen kann. Der Program Transformer greift auf die internen Werkzeuge, unter anderen auf die *History/Future-Database* (Verwaltung der Versionen) und den *General Simplifier* (Vereinfachen von logischen und mathematischen Ausdrücken) zu. Das umgewandelte Programmstück wird anschliessend in die Datenbank zurückgeschrieben.

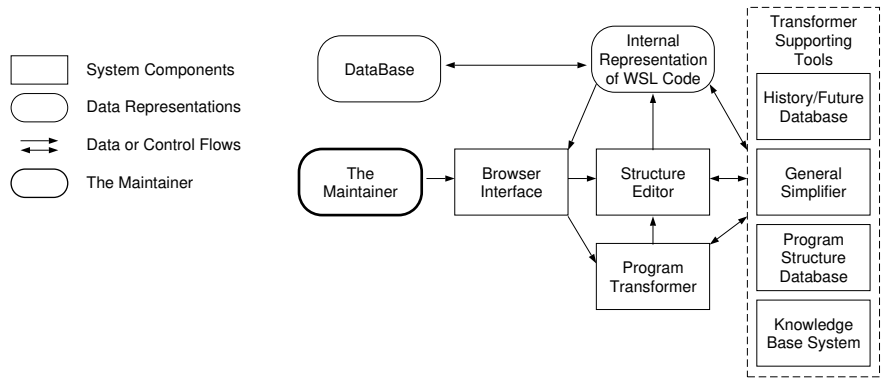


Abbildung 12: Von einfachen WSL-Konstrukten zu komplexeren WSL-Konstrukten (nach [Yan91])

Phase 3: Integration
Die dritte Phase beginnt, wenn alle Programmstücke in der Datenbank bearbeitet worden sind (vgl. Abb. 13). Sie werden zu einem Programmtext mit komplexeren WSL-Konstrukten zusammengefügt, der dann in eine Spezifikationssprache (z.B. Z) überführt wird.

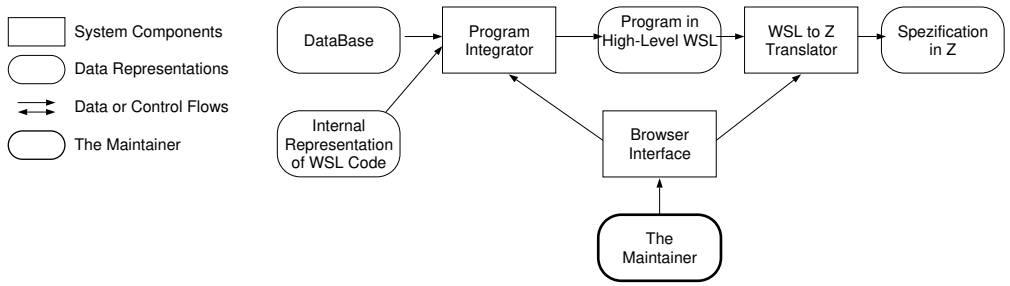


Abbildung 13: Von komplexeren WSL-Konstrukten zu Spezifikationen in Z (nach [Yan91])

4.5. Transformationen im Maintainer's Assistant

4.5.1 Begriffe

Eine Vorbedingung eines Programms ist die Bedingung, welche vor dem Ausführen dieses Programms gelten muss. Eine Nachbedingung eines Programms ist die Bedingung, welche nach dem Ausführen dieses Programms gelten muss, falls das Programm terminiert. Eine schwächste Vorbedingung eines Programms ist die schwächste Bedingung aller Vorbedingungen dieses Programms, welche vor dem Ausführen dieses Programms gerade noch gelten muss, damit die vorgegebene Nachbedingung erfüllt werden kann. So ist z.B. die Bedingung (x>10) schwächer als die Bedingung ((x>15) $\wedge$ (x<20)).

Zwei iterative Programme können nur dann semantisch äquivalent sein, falls sie die gleiche schwächste Vorbedingung haben. Dieser Satz gilt auch für rekursive Programme. Um die Korrektheit eines Programms zu zeigen, fängt man mit der Nachbedingung an und arbeitet sich vom Ende des Programms aus gegen den Beginn vor. Dabei bestimmt man bei jeder Anweisung die schwächste Vorbedingung, die gelten muss, um auf die gewünschte Nachbedingung zu gelangen. Im nächsten Schritt ist dann die neu bestimmte schwächste Vorbedingung die Nachbedingung der vorangehenden Anweisung. So fährt man fort, bis das ganze Programm bearbeitet ist. Auf diese Weise findet man die schwächste

Vorbedingung, die vor dem Ausführen des Programms gelten muss. Falls sie erfüllbar ist, liefert das Programm auf die erlaubten Eingaben einen Endzustand, der die vorgegebene Nachbedingung erfüllt. Dies gilt aber nur, falls das Programm terminiert.

Aus der Spezifikation des Programms (vor der Implementation erstellt) seien die schwächste Vorbedingung und die Nachbedingung bekannt, wie sie gelten sollten. Stimmen die schwächsten Vorbedingungen, einerseits die aus der Spezifikation des Programms und andererseits die aus dem Programm hergeleitete (die Nachbedingung der Spezifikation von hinten durch das Program propagiert) überein, so haben die Spezifiakation und ihre Implementation die gleiche Bedeutung. Beide Darstellungen sind dann semantisch äquivalent.

4.5.2 Eigenschaften

Alle Transformationen auf der WSL-Darstellung der Programme sind so gestaltet, dass die schwächste Vorbedingung des gesamten Programms oder von seinen Teilen vor und nach ihrer Anwendung identisch bleiben. So ist die Korrektheit des Programms nach jeder angewandten Transformation garantiert [WCM89].

Mit WSL lassen sich sowohl Programme, als auch Spezifikationen von Programmen darstellen. Dies erlaubt den stufenweisen Übergang von der Implementationsebene in die Spezifikationsebene durch Transformationen, ohne einen Wechsel der Darstellungssprache.

Die Transformationen selber werden in Meta-WSL dargestellt, welches eine geringfügige Erweiterung von WSL ist [Bul90].

4.5.3 Typen von Transformationen und ihre Anwendung

- *Syntaktische Transformationen*
  Bei diesen Transformationen wird eine Source-To-Source Abbildung vorgenommen, ohne dass dabei der Kontrollfluss tangiert würde. Beispiele wären das Eliminieren von unbenutzten Variablen oder das Vereinfachen von logischen Ausdrücken in bedingten Anweisungen.
- *Auflösen von Rekursionen*
  Parameterlose Prozeduren, welche in gegenseitiger Rekursion stehen, sprich GOTOs simulieren, können durch Transformationen in einfachere Ablaufstrukturen, wie Iterationen, umgewandelt werden.
- *Semantische Transformationen*
  Diese sind wiederum Source-To-Source Abbildungen, welche aber den Kontrollfluss ändern. Beispiele wären das Eliminieren von Ästen in Case-Anweisungen oder das Mischen von Zuweisungen.
- *Editieren*
  Diese Transformation geschieht ohne Kontrolle des Maintainer's Assistant. Darin liegt auch die Gefahr: Vom Benutzer getätigte Änderungen können Fehler enthalten.

Die obengennanten Transformationen können benutzt werden, um lokale oder globale Restrukturierungen am Programmtext vorzunehmen. Man kann aber auch genauso gut nur Teile des Programmtexts in eine höhere Abstraktionsstufe überführen, um die Lesbarkeit und Verständlichkeit zu erhöhen. Im weiteren ist es möglich, ein Programm mit neuen Datenstrukturen zu versehen, um die Effizienz zu erhöhen oder um neue Anforderungen zu erfüllen.

4.5.4 Vorteile von WSL gegenüber Höheren Programmiersprachen

Die Syntax von WSL ist viel einfacher als die von höheren Programmiersprachen. In WSL lassen sich sowohl maschinennahe Formulierungen (Implementation) als auch Spezifikationen von Programmen ausdrücken [WCM89, Bul90], wogegen die höheren Programmiersprachen ersteres abdecken, aber mit zweiterem grosse Mühe bekunden. Ein weiterer Vorteil von WSL ist ihre Unabhängigkeit von der Implementationssprache des Programmtexts, was die Portierung in eine neue Sprache erleichtert oder gar erst möglich macht. Jede höhere Programmiersprache hat nämlich ihre Stärken und weist dadurch Merkmale auf, die in anderen Sprachen nicht zur Verfügung stehen oder nur mühsam zu realisieren sind.

Neben den Programmen auf Implementations- und Spezifikationsebene kann man auch die Transformationen selber in einer Erweiterung von WSL, in Meta-WSL, ausdrücken. Es erübrigt sich damit das Lernen einer neuen Sprache. Die Realisierung von Meta-WSL war einfach, da nur geringe Erweiterungen nötig waren. Alle Werkzeuge von WSL sind auch in Meta-WSL verwendbar (Compiler, Browser, . . .). Ein weiterer Vorteil von Meta-WSL liegt darin, dass damit die Auto-Maintenance möglich wird. Das bedeutet, das WSL zum Warten der WSL-Implementation (Transformationen) eingesetzt werden kann.

4.5.5 Nachteile des Transformationsansatzes

Um einen Programmtext umzuformen, muss man sehr viele Transformationen anwenden. Neben dem Problem, welche Transformationen überhaupt in der Datenbank der Transformationen vorhanden sind, muss man auch die Reihenfolge ihrer Anwendung bestimmen. Dies bleibt trotz der Unterstützung des Maintainer's Assistant immer noch aufwendig und mühsam.

Ein weiterer Nachteil ist, dass die Transformationen auf prozess-orientierte Programme abgestimmt sind, und darum daten-intensive Programme nicht behandelt werden können. Viele der heute im Einsatz stehenden kommerziellen Applikationen in der Industrie sind aber gerade daten-intensive Programme.

4.6. Nutzen und Aussichten

Mit Hife des Maintainer's Assistant sind lokale Änderungen des Quellcodes möglich, ohne sich um deren Auswirkungen auf andere Teile des Textes zu kümmern, da der Maintainer's Assistant die konsistente Ausführung von Modifikationen sicherstellt. Der Programmierer hat zwei Möglichkeiten, mit dem Maintainer's Assistant zu arbeiten. Er kann als erstes aus der Datenbank von Transformationen selber Transformationen aussuchen, zusammenstellen und zur Ausführung bringen. Falls er aber Transformationen anwenden will, die im aktuellen Kontext zu Inkonsistenzen führen können oder welche nicht erlaubt sind, so werden diese vom Maintainer's Assistant zurückgewiesen. Die zweite Möglichkeit des Programmierers besteht darin, den Maintainer's Assistant anzuweisen, ihm Vorschläge von Transformationen zu geben, welche der Maintainer's Assistant im aktuellen Kontext anwenden würde und welche erlaubt sind. Aus diesen Transformationen wählt der Programmierer dann aus.

Dadurch, dass der Programmierer Teile von Programmen bearbeiten kann (Umstrukturieren, nicht erreichbare Textstellen entfernen, . . .), ohne das ganze Programm anzupassen, kommt er schrittweise zu einer Spezifikation des vorliegenden Programms.

Der Ansatz im Maintainer's Assistant ist prozess-orientiert und kann darum für datenintensive Applikationen, wie sie im kommerziellen Bereich häufig sind, schlecht eingesetzt werden. Eine weitere Schwierigkeit mit dem Maintainer's Assistant entsteht, falls der Benutzer direkt auf dem Quellcode Änderungen vornimmt. Der Maintainer's Assistant kann dann die Äquivalenz zwischen dem neuen und dem alten Programm nicht mehr garantieren.

Im Rahmen des Maintainer's Assistant stehen noch offene Fragen im Raum, die weiter verfolgt werden. Hier nur einige davon:

- Bestimmen von Programmstellen, an denen es von Vorteil wäre, Transformationen anzuwenden (z.B. besonders "unstrukturierte Programmteile").
- Automatisieren der Anwendung von Transformationen, d.h. der Maintainer's Assistant soll nach vorgegebener Strategie oder nach Optimalitätskriterien selber Transformationen anwenden können.

5. Das REDO Projekt

5.1. Herkunft

REDO ist ein Projekt (Nr. 2487) im Rahmen des ESPRIT-Programms. Das Akronym steht für "*RE*verse Engineering, *DO*cumentation and Validation of Systems" [Zuy93]. Projektmitglieder sind Firmen aus der Privatwirtschaft, staatliche Instutionen und Universitäten, darunter die University of Oxford. Dieses Projekt dauerte von 1989 bis Frühjahr 1992 und darin involviert waren unter anderen auch K. Lano, P.T. Breuer und H. Haughton.

5.2. Grundidee

Das angestrebte Ziel besteht darin, existierende COBOL-Programme in eine Spezifikation zu überführen. Eine solche Repräsentation ist sehr kompakt und leserlich. Dieses Ziel wird in drei Schritten erreicht. Zunächst wird der Programmtext in UNIFORM abgebildet, eine Untermenge von COBOL-Verben, danach in eine funktionale Zwischensprache übersetzt und anschliessend in eine Spezifikationssprache überführt. Die gewünschten Sichtweisen auf das Programm (z.B. Datenflussdiagramme, Entity/Relationship-Diagramme, Call-Graphen) werden ausschliesslich aus dem Programmtext extrahiert, weil die Dokumentation der Programme fehlerhaft oder unvollständig sein kann.

Die abgeleiteten Spezifikationen werden dann mit den vorgegebenen Spezifikationen verglichen, um ihre Übereinstimmung resp. Abweichung zu überprüfen.

5.3. Vorgehen bei der Analyse

In objekt-orientierten Umgebungen sind immer einige Grundklassen vordefiniert, von denen neue Objekte abgeleitet werden können. So werden beim Ansatz in REDO z.B die Standard-COBOL-Konzepte für Files und Sequenzen als Basisklassen genommen. Für jeden Filetyp und jeden Zugriffsmodus wird eine eigene Klasse gebildet. Wie aus der

Typ des Files	**Modus des Zugriffs**
SEQUENTIAL	SEQUENTIAL
RELATIVE	SEQUENTIAL
RELATIVE	RANDOM
RELATIVE	DYNAMIC ACCESS
INDEXED	SEQUENTIAL
INDEXED	RANDOM
INDEXED	DYNAMIC ACCESS

Abbildung 14: Tabelle der File-Zugriffarten

Abb. 14 ersichtlich ist, gibt es sieben Basisklassen von Files. Alle diese Klassen haben die Zugriffsmethoden SEND und RECEIVE, die spezifisch für ihre Klasse geschrieben sind. Auch der REPORT WRITER in COBOL wird als Klasse definiert, mit den Zugriffsmethoden INITIATE, GENERATE und TERMINATE.

Die oben erwähnten Objektklassen dienen als Gerüst für die Übersetzung von COBOL-Programmtexten in eine objekt-orientierte Repräsentation. Diese Umsetzung wird in drei Schritten erreicht, die im folgenden genauer beschrieben werden (aus [LBH91, LBH93]).

A. Schritt 1: *Übersetzen in UNIFORM* ('clean')

Der COBOL-Programmtext wird in die Zwischensprache UNIFORM übersetzt. Der neue Programmtext ist semantisch äquivalent zum ursprünglichen Programm. Die Übersetzung eliminiert lediglich redundante Konstrukte aus COBOL. Die folgende Liste zeigt einige solcher Konstrukte.

- Die PERFORM UNTIL Anweisungen werden durch DO UNTIL Schleifen ersetzt. Durch Einführen einer zusätzlichen Variablen können auch PERFORM TIMES und PERFORM THROUGH ersetzt werden.
- Die DEPENDING ON Klausel in einer GO TO Anweisung wird durch ein explizites IF THEN ELSE ersetzt.
- Die arithmetischen Anweisungen (ADD, SUBTRACT, MULTIPLY, . . .) werden in Zuweisungen übersetzt (X := Y + 1). Falls die ON SIZE ERROR Klausel verwendet wurde, werden sie durch bedingte Zuweisungen ersetzt (IF Y<MAX THEN X := Y + 1).
- Die MOVE TO Anweisung kann nicht in eine einfache Zuweisung überführt werden, da die Operanden unterschiedliche Längen haben können. In COBOL wird die Umformung der einzelnen Operanden automatisch gemacht. In UNIFORM müssen diese Umformungsfunktionen explizit angegeben werden.

Das Ziel des ersten Schritts ist die Vereinfachung des Programmtexts. Die impliziten Angaben in den einzelnen DIVISIONs werden dort, wo sie verwendet werden, explizit dargestellt. So müssen bei den nachfolgenden Analysen nicht jedesmal alle DIVISIONs betrachtet werden, um die Bedeutung eines Ausdrucks zu verstehen.

B. Schritt 2: *Weitere Abstraktion* ('specify')

Aus der FILE SECTION und der ENVIRONMENT DIVISION ist bestimmbar, welche Arten von Files oder Reports verwendet werden. Diese werden als Hauptvariablen des Programms definiert und als Objekte im objekt-orientierten Sinne repräsentiert. Ebenso werden im Programm verwendete indizierte Arrays als Objekte dargestellt.

Anhand des Datenflussdiagramms werden ihre Abhängigkeiten untereinander bestimmt und formuliert. Mit Hilfe des Datenflussdiagramms ist relativ einfach zu bestimmen, welche Variable (Flag, Counter, . . .) auf welches Objekt Bezug nimmt. Sie wird dem entsprechenden Objekt als ein neues Attribut zugefügt. Neben dem Datenflussdiagramm werden auch sogenannte Phasen und Programm-Slices des Programms herangezogen, um Interaktionen zwischen den Objekten zu entdecken. Phasen sind Programmteile zwischen einer OPEN- und der zugehörigen CLOSE-Operation, in welchem der Zugriffsmodus des betreffenden Files nicht verändert wird. Programm-Slices sind ausführbare Auszüge aus dem Programmtext, welche bezüglich bestimmter Variablen das gleiche Verhalten aufweisen, wie das gesamte Programm (vgl. Abb. 15).

$$slice_f \begin{pmatrix} x := y + 10; \\ z := x * x \\ SEND\ x\ TO\ f; \end{pmatrix} = \begin{pmatrix} x := y + 10; \\ SEND\ x\ TO\ f; \end{pmatrix}$$

Abbildung 15: Beispiel eises Slices (aus [Zuy93] S.234)

In der PROCEDURE DIVISION, wo die eigentliche Bearbeitung im Programm stattfindet, werden die Paragraphen und die Schleifen in Funktionen überführt und die GOTOs und PERFORMs in Funktionsaufrufe.

C. Schritt 3: *Umsetzen in Spezifikationen* ('simplify')

Die aus dem Programmtext abgeleiteten Funktionen werden in die Spezifikationssprache Z übertragen. Dies ist leicht zu realisieren, da diese Spezifikationssprache selber auch einen funktionalen Ansatz als Grundlage hat, wie später gezeigt wird. In den Objekten, welche aus dem Programmtext extrahiert wurden, werden die Funktionen ebenfalls in der Spezifikationssprache Z formuliert. So erhält man eine objekt-orientiert Darstellung des COBOL-Programms in Z++, welches eine objekt-orientiert Erweiterung von Z darstellt.

5.4. Die Spezifikationssprache Z

5.4.1 Ursprung und Zweck

Die Spezifikationssprache Z basiert auf den Ideen von J.R. Abrial aus den frühen 80er Jahren. Sie wurden von B.A. Sufrin aufgegriffen und weiterentwickelt. Später hat J.M. Spivey, gemeinsam mit anderen aus der *Programming Research Group* der *University of Oxford* daran weiter gearbeitet.

5.4.2 Begriffe

Für alle formalen Spezifikationen gilt, dass sie das WAS beschreiben, ohne zu sagen, WIE man dies dann realisieren kann. Somit kann man über die Essenz des spezifizierten Sytems sprechen. Die formalen Spezifikationen erfüllen unterschiedliche Anforderungen aus der Praxis. Der Designer kann seine Ideen besser mit dem Anwender besprechen. Der Programmierer kann sich auf die Eindeutigkeit der formalen Spezifikation verlassen (aber evtl. nicht auf das, was damit spezifiziert wurde). Die formalen Spezifikationen sind mitunter ein zentrales Element in der Kommunikation zwischen verschiedenen Gruppen mit unterschiedlichen Anforderungen an die Beschreibung eines Systems und können mitunter als Dokumentation herangezogen werden. Sie sind präzise und eindeutig.

5.4.3 Eigenschaften und Syntax

Durch Verwenden von mathematischen Strukturen (Mengen, Relationen, Funktionen, . . .) zur Spezifikation von Programmen bleibt man implementationsunabhängig. Durch ihren formalen Charakter können bestimmte Eigenschaften eines spezifizierten Programms hergeleitet oder bewiesen werden. Ein gutes Beispiel einer solchen Spezifikationssprache ist Z.

Das zentrale Konzept in Z ist das Schema, welches Teile einer Spezifikation repräsentiert. So können Teile separat spezifiziert werden und später durch den *Schema Calculus* von Z (siehe Kap. 5.4.4) miteinander in Beziehung gesetzt werden. Durch diese Eigenschaft von Z wird die Wiederverwendung von Teilen einer Spezifikation erst möglich. Sie führt zur Reduktion des Aufwands für eine Spezifikation, da man sich auf solche Teile beziehen kann.

Die Schemata können sowohl statische wie auch dynamische Sachverhalte eines Systems ausdrücken. Statische Sachverhalte sind z.B. die Systemzustände, die möglich sind, und die Invarianten, die bei jedem Zustandswechsel eingehalten werden. Die dynamischen sind z.B. zugelassene Operationen, Beziehungen zwischen den Ein- und Ausgängen von Operationen oder die Änderungen des Systemzustands.

Die Spezifikationssprache Z besitzt eine formal definierte Semantik, welche es erlaubt, gewisse Konsistenzprüfungen schon bei der Spezifikation des System automatisch durchzuführen. So können viele Fehler und Ungenauigkeiten schon früh erkannt werden, und nicht erst bei der Implementation, wo sie dann sehr hohe Kosten verursachen.

5.4.4 Beispiele von Schemata

Die folgenden Beispiele sind aus [Spi92] entnommen. Es werden einige Aspekte von Z anhand eines Buches mit Geburtstagseinträgen (BirthdayBook) gezeigt. Jeder Eintrag bestehe aus einem Namen und einem Datum. Wobei der Name vom Typ NAME, welcher alle zulässigen Namen enthält, sei und das Datum vom Typ DATE, welcher alle zulässigen Daten enthält. Nun kann der Zustandsraum eines BirthdayBook mit einem Schema beschrieben werden (siehe Abb. 16). Das Schema hat den Namen *BirthdayBook* und besteht aus zwei

BirthdayBook

$$known : \mathbb{P}\, NAME$$
$$birthday : NAME \pfun DATE$$
$$known = \mathrm{dom}\, birthday$$

Abbildung 16: Schema *BirthdayBook* aus [Spi92]

Teilen. Der eine ist oberhalb der Mittellinie und der zweite ist unterhalb der Mittellinie. Im oberen Teil werden die verwendeten Grössen beschrieben und im unteren Teil werden die Beziehungen zwischen ihnen festgehalten.

- *known* ist die Menge aller dem System bekannter Namen, also ein Element der Potenzmenge aller Namen.
- *birthday* ist eine Funktion, die jedem Namen ein Datum zuordnet.
- *known* = dom *birthday* gibt an, dass der Definitionsbereich der Funktion *birthday* gerade der Menge der bekannten Namen des Systems (*known*) entspricht.

Der Ausdruck *known* = dom *birthday* stellt eine Invariante im BirthdayBook dar, die vor und nach jeder Operation auf BirthdayBook immer erfüllt bleibt. Ein möglicher Zustand, den das BirthdayBook einnehmen kann, könnte wie folgt aussehen:

known	=	{	John, Mike, Susan		}
birthday	=	{	John	$\mapsto$	25-Mar,
			Mike	$\mapsto$	20-Dec,
			Susan	$\mapsto$	20-Dec }

Man sieht auch, dass die Invariante *known* = dom *birthday* erfüllt ist, denn *known* = {John, Mike, Susan} und dom *birthday* = {John, Mike, Susan}.
Nun können auch Operationen über diesem Zustandsraum definiert werden. In Abb. 17 wird eine Operation zum Erweitern von BirthdayBook definiert. Der Ausdruck Δ*BirthdayBook*

AddBirthday

$\Delta BirthdayBook$
$name? : NAME$
$date? : DATE$

$name? \notin known$
$birthday' = birthday \cup \{name? \mapsto date?\}$

Abbildung 17: Schema *AddBirthday* aus [Spi92]

besagt, dass die Operation, welche durch dieses Schema beschrieben wird, den aktuellen Zustand von *BirthdayBook* ändert. Ein *?* am Ende eines Variablennamens zeigt an, dass diese Grösse eine Eingabegrösse ist. Im Falle des Schemas *AddBirthday* sind dies *name?* vom Typ NAME und *date?* vom Typ DATE. Die erste Zeile unterhalb der Mittellinie ist eine Vorbedingung, die erfüllt sein muss, um die beschriebene Operation erfolgreich durchzuführen. Sie besagt, dass der eingegebene Name nicht bereits im Definitionsbereich von BirthdayBook enthalten sein darf. Die zweite Zeile besagt, dass die Funktion *birthday* erweitert wird, falls die Vorbedingung erfüllt ist.

Intuitiv nimmt man an, dass nach der Operation *AddBirthday* die Menge der Namen ebenfalls erweitert worden ist, nämlich zu *known'* = *known* $\cup$ *{name?}*. Es kann hergeleitet werden, dass dies zutrifft:

$$
\begin{aligned}
known' &= \text{dom } birthday' \\
&= \text{dom } (birthday \cup \{name? \mapsto date?\}) \\
&= \text{dom } birthday \cup \text{dom } \{name? \mapsto date?\} \\
&= \text{dom } birthday \cup \{name?\} \\
&= known \ \cup \{name?\}
\end{aligned}
$$

Eine weitere Operation auf dem BirthdayBook kann die Abfrage eines Eintrages sein (siehe Abb. 18). Der Ausdruck Ξ*BirthdayBook* in Abb. 18 sagt uns, dass der Zustand

FindBirthday

$\Xi BirthdayBook$

$name? : NAME$

$date! : DATE$

$name? \in known$

$date! = birthday(name?)$

Abbildung 18: Schema *FindBirthday* aus [Spi92]

des Schemas bei dieser Operation unverändert bleibt. Sie hat die gleiche Bedeutung, wie Δ*BirthdayBook* zusammen mit den beiden Gleichungen *known'* = *known* und *birthday'* = *birthday*.

Ein *!* am Ende einer Variablen zeigt an, dass diese Grösse eine Output-Grösse ist. Im Schema *FindBirthday* ist dies *date!* vom Typ DATE. Die erste Zeile unterhalb der Mittellinie ist wiederum die Vorbedingung für diese Operation. Sie besagt, dass der eingegebene Name bereits im BirthdayBook eingetragen sein muss, um die Operation *FindBirthday* erfolgreich ausführen zu können. In der zweiten Zeile ist die Wirkung der Operation festgelegt. Der Outputgrösse *date!* wird der Wert der Funktion *birthday* an der Stelle *name?* übergeben, falls die Vorbedingung erfüllt ist.

Die Wirkungen aller Operationen in der Spezifikationssprache Z werden als Beziehungen (Relationen) zwischen ihren Ein- und Ausgabegrössen beschrieben, ohne vorzugeben, WIE sie realisiert werden sollen.

Bis hierhin haben wir nichts darüber ausgesagt, was passieren soll, falls die geforderten Vorbedingungen der einzelnen Operationen nicht erfüllt sein sollten. Das System kann abstürzen oder fehlerhaft weiterarbeiten, ohne die gegebenen Spezifikationen zu verletzen. Um diesen Mangel zu beheben, führen wir einen neuen Datentyp ein:

$$RESULT = \quad ok \quad | \quad already_known \quad | \quad not_known$$

Wir definieren nun eine Operation *Success*, die nichts anderes macht, als eine Ausgabe *ok* zu liefern (siehe Abb. 19). Mit dem Konjunktionsoperator $\wedge$ des Schema-Calculus

Success

$result! : REPORT$

$result! = ok$

Abbildung 19: Schema *Success* aus [Spi92]

[Spi92] können wir dieses Schema mit dem früher definierten Schema *AddBirthday* verknüpfen.

$$AddBirthday \quad \wedge \quad Success$$

Diese Verknüpfung bedeutet, dass bei erfolgreichem Ausführen der Operation *AddBirthday* ein Resultat *ok* zurückgeliefert wird. Sie gilt als Bestätigung, dass die Operation korrekt ablief, was man in der ursprünglichen Version nicht hatte. Um eine vollständige Spezifikation zu erhalten, muss man für jede Situation, wo Fehler auftauchen können, diesen Sachverhalt

festhalten. Das nächste Schema drückt aus, dass ein eingegebener Name bereits im BirthdayBook eingetragen ist (siehe Abb. 20). Mit Hilfe der eben aufgezeigten Operation kann

AlreadyKnown

$$\Xi BirthdayBook$$
$$name? : NAME$$
$$result! : REPORT$$

$$name? \in known$$
$$result! = already_known$$

Abbildung 20: Schema *AlreadyKnown* aus [Spi92]

man eine robuste Version von *AddBirthday* definieren, nämlich:

$$RAddBirthday = (AddBirthday \ \wedge \ Success) \ \vee \ AlreadyKnown$$

Dadurch, dass *AddBirthday* und *AlreadyKnown* Vorbedingungen aufweisen, die komplementär zueinander sind, wird entweder ein neues Element in das BirthdayBook eingetragen und das Resultat *ok* geliefert, oder aber das BirthdayBook bleibt unverändert, da das Element bereits existiert, und das Resultat *already_known* wird gemeldet.

Wir haben gezeigt, dass es möglich ist Teile eines Systems separat zu spezifizieren und sie nachträglich mit Hilfe des Schema-Calculus zu kombinieren. Die zusammengesetzte Operation *RAddBirthday* hätte auch zu Beginn in einem Schema definiert werden können (Abb. 21). Die Interpretation des Schemas und die Bedeutung der einzelnen Zeilen seien dem Leser überlassen.

RAddBirthday

$$\Delta BirthdayBook$$
$$name? : NAME$$
$$date? : DATE$$
$$result! = REPORT$$

$$(name? \notin known \wedge$$
$$\quad birthday' = birthday \cup \{name? \mapsto date?\} \wedge$$
$$\quad result! = ok) \vee$$

$$(name? \in known \wedge$$
$$\quad birthday' = birthday \wedge$$
$$\quad result! = already_known)$$

Abbildung 21: Schema *RAddBirthday* aus [Spi92]

5.4.5 Vorteile / Nachteile

Dadurch, dass der Spezifikationssprache Z ein funktionaler Ansatz zu Grunde liegt, können vor allem die funktionalen Anforderungen an ein System einfach erfasst und ausgedrückt werden. Ein Beispiel wäre die Summation einer Kolonne aus einer Tabelle.

Anforderungen, wie beispielsweise, dass ein Fenster auf dem Bildschirm einen roten Hintergrund und blaue Knöpfe haben soll, oder dass die Antwortzeit einer Operation erträglich sein soll, können auch in Z formuliert werden, obwohl sie in der Regel nicht viel mit der Funktionalität eines Systems zu tun haben.

Es besteht die Möglichkeit in Z generische Funktionen zu formulieren, die mit unterschiedlichen Datentypen verwendet werden können. Als Beispiel sei die Funktion *member* erwähnt, die in einer Liste von Elementen mit beliebigem Typ das Vorkommen eines bestimmten Elements testet.

Z hat auch eine statische Typprüfung, die es erlaubt, aus den Interaktionen zwischen den verwendeten Argumenten den Typ des Funktionsresultats herzuleiten. Damit kann sichergestellt werden, dass in einer Spezifikation keine Typfehler auftreten.

5.5. Möglichkeiten des REDO-Ansatzes

Während der Analysephase gehen bei dem REDO-Ansatz Details in den Programmen verloren, z.B. weiss man nicht mehr, ob ein Funktionsaufruf mit einem GOTO oder mit einem PERFORM implementiert war. Trotzdem wird durch die objekt-orientierte Repräsentation das Programm für den Programmierer viel verständlicher, da die wichtigen Angaben zum Verstehen des Programms in den Objekten konzentriert sind. Auch die Lesbarkeit des Programms ist viel besser und dadurch auch die Wartbarkeit. Aus der objekt-orientierten Darstellung lassen sich Dokumentationen des zugehörigen Programms generieren.

- Aus der Hirarchie der Objekte lassen sich hat-teil/ist-teil Graphen ableiten, welche die Zusammengehänge unter den Objekten aufzeigen.
- Aus den Methoden der einzelnen Objekte lassen sich die Datenflüsse ableiten.
- Auf die Repräsentationen der Datenflüsse lassen sich Metriken anwenden, um die Komplexität von Interaktionen zwischen den Objekten zu bestimmen. Je grösser die Vernetzung eines Objekts ist, desto aufwendiger sind Änderungen an diesem Objekt.

5.6. Vorteile dieses Ansatzes

Durch die Übersetzung in die Zwischensprache UNIFORM ist der Ansatz von REDO sprachunabhängig, es wird lediglich ein Übersetzer von der betreffenden Sprache in UNIFORM benötigt.

Auch ein eingeschränktes Re-Engineering ist mit dem REDO-Ansatz möglich. Durch das Reverse-Engineering erhält man eine abstrakte Beschreibung der Funktionalität eines Programms. Das Forward-Engineering liefert dann den verbesserten Programmtext. Dabei kommt man z.B. zu einem Programm, das ohne GOTOs auskommt, da jede Funktion als ein PARAGRAPH, und jeder Funktionsaufruf als ein PERFORM generiert werden kann.

6. Zusammenfassung und Ausblick

Die beschriebenen Ansätze haben ihre Stärken an unterschiedlichen Stellen. So unterstützt der Programmer's Apprentice den Entwicklungsprozess von Programmen, um das Wartungsproblem zu mildern. Mit seiner Hilfe ist es möglich, vordefinierte Programmuster wiederzuverwenden. Dadurch können die Produktionskosten von Programmen reduziert werden. Die wiederverwendbaren Programmuster können ihrerseits aus alten Programmen extrahiert worden sein. Beim Programmer's Apprentice steht also Wiederverwendbarkeit statt Reengineering im Vordergrund.

Beim Ansatz des Maintainer's Assistant wird das Programm zuerst in die Zwischensprache WSL übersetzt. Auf dieser Zwischenrepräsentation können dann Transformationen durchgeführt werden. Sie bilden das ursprüngliche Programm in ein semantisch äquivalentes Programm ab. Die Transformationen sind in einer leichten Erweiterung von WSL geschrieben. Beim Ansatz des Maintainer's Assistant ist die Programmiersprachunabhängigkeit durch die Zwischenrepräsentation und die Möglichkeit der Bearbeitung von Teilen aus Programmen erwähnenswert.

Im REDO-Projekt wird das Programm (in COBOL) zuerst in UNIFORM, welches einem reduziertem COBOL entspricht, übersetzt. Als nächstes wird diese Repräsentation in eine funktionale Sprache abgebildet und schliesslich in eine Spezifikation des Programms übersetzt. Die Spezifikationssprache im REDO-Projekt ist Z. Der grosse Vorteil dieses Ansatzes ist, dass aus einem Programm eine Spezifikation extrahiert werden kann. Das Reverse Engineering und die Wiederverwendbarkeit ist im REDO-Projekt sogar auf der Spezifikationsebene möglich, im Gegensatz zum Programmer's Apprentice, wo nur auf der Implementationsebene Wiederverwendung möglich ist. So kann man aus einem bestehenden Programm eine Spezifikation ableiten, und diese als Ausgangspunkt für eine Neuentwicklung nehmen. Dies wiederum führt zur Kostenreduktion in der Entwicklung.

Ein Hauptziel bei allen Ansätzen ist es, dem Programmierer das Verstehen eines Programmes zu erleichtern. Eine Möglichkeit ist die Dokumentation zu den Programmen aktuell zu halten. Dies bedingt, dass alle Modifikationen an der Software kontrolliert werden und in der Dokumentation, sei es im Programmtext oder separat, nachgeführt werden. Dem Programmierer können auch Werkzeuge von Nutzen sein, welche im Programm implizit vorhandene Informationen extrahieren. Solche Werkzeuge können z.B. ein Crossreferenzer, ein Call-Graphgenerator, Daten- und Kontrollflussgeneratoren oder ein sprachsensitiver Editor sein. Mit ihrer Hilfe ist es leichter möglich, die Funktionalität eines Programms zu verstehen und die geeigneten Änderungen daran durchzuführen.

Zu erwähnen ist, dass seit Ende 1992 am Institut für Informatik der Universität Zürich in enger Zusammenarbeit mit der Rentenanstalt / Swiss Life am Projekt AEMES[†] (*A*n *E*xtensible *M*aintenance *E*ngineering *S*ystem) gearbeitet wird, welches sich mit dieser Problematik auseinandersetzt [FKO92]. Es geht dabei um das Extrahieren und das Präsentieren von Informationen aus dem Programmtext und deren geeignete und effiziente interne Repräsentation.

[†] Das Projekt AEMES wird durch den KWF (Kommission zur Förderung der wissenschaftlichen Forschung), der Firma Bull und der Schweizerischen Lebensversicherungs- und Rentenanstalt / Swiss Life unterstützt. Das Projekt wird geleitet von Prof. L. Richter vom Institut für Informatik der Universität Zürich.

7. Danksagung

Ich danke Ulrich Reimer (Rentenanstalt/Swiss Life), der massgeblich am Gelingen dieses Berichts beteiligt war, und Peter Baumann (Universität Zürich) für ihre Anregungen und für die Diskussionen zum Thema, die ich mit ihnen führen durfte. Mein Dank geht auch an Herrn L. Richter für seine Unterstützung bei der Erstellung dieses Berichts. Weiter bedanke ich mich bei meinen Kollegen Jürg Fässler und Markus Kiser für ihr Korrekturlesen.

Bibliographie

[Bul90] T. Bull. An Introduction to WSL Program Transformer. In *Conference on Software Maintenance 1990*, pages 242–250, Washington, D.C., November 1990. IEEE Computer Society Press.

[CKMW88] F.W. Calliss, M. Khalil, M. Munro, and M. Ward. A Knowledge-Based System for Software Maintenance. In *Conference on Software Maintenance 1988*, pages 319–324, Washington, D.C., October 1988. IEEE Computer Society, IEEE Computer Society Press.

[FKO92] J. Faessler, M. Kiser, and Z. Oeztuerk. AEMES - First Specification. internal report KWF2326.1(AEMES)-D01-SPEC V1.0 (12.92), Institut fuer Informatik, University of Zurich, Winterthurerstrasse 190, 8021 Zurich, December 1992.

[Fra92] A. Frazer. *Software Reuse and Reverse Engineering in Practice*, volume 12 of *UNICOM Applied Information Technology*, chapter 10, pages 209–243. Chapman and Hall, London, 1992. Reverse Engineering - Hype, Hope or Here?

[Kup88] R.I. Kuper. Automated Techniques for the Location of Software Bugs. Report 904, MIT, Artificial Intelligence Lab, MIT, May 1988.

[LBH91] Kevin Lano, P. T. Breuer, and Haughton H. Applying Formal Methods to the Reverse-Engineering of COBOL. Technical Note ?, Oxford University Computing Laboratory, Programming Research Group, 11 Keble Road, Oxford OX1 3QD, August 1991.

[LBH93] K. Lano, P.T. Breuer, and H. Haughton. Reverse-Engineering COBOL via Formal Methods. *Journal of Software Maintenance / Research and Practice*, 5(1):13–35, March 1993.

[Ric81] Charles Rich. A Formal Representation For Plans In The Programmer's Apprentice. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pages 491–506, August 1981.

[RW88] Charles Rich and Richard C. Waters. The Programmer's Apprentice: A Research Overview. *IEEE Computer*, 21(11):10–25, November 1988.

[RW90] Charles Rich and Richard C. Waters. *The Programmer's Apprentice*. ACM Press New York, NY, 1990.

[Spi92] J.M. Spivey. *The Z notation: A Reference Manual*. Prentice Hall, New York, second edition, 1992.

[War93] M. Ward. Abstracting a Specification from Code. *Journal of Software Maintenance / Research and Practice*, 5(2):101–122, June 1993.

[Wat85] Richard C. Waters. The Programmer's Apprentice: A Session with KBEmacs. *IEEE Transactions on Software Engineering*, 11(11):1296–1320, August 1985.

[WCM89] M. Ward, F.W. Calliss, and M. Munro. The Maintainer's Assistant. In *Conference on Software Maintenance 1989*, pages 192–199, Washington, D.C., October 1989. IEEE Computer Society Press.

[Wil90] Linda Mary Wills. Automated Program Recognition: A Feasibility Demonstration. *Artificial Intelligence*, 45:113–171, 1990.

[Yan91] Hongij Yang. The Support Environment for a Reverse Engineering System - The Maintainer's Assistant. In *Conference on Software Maintenance 1991*, pages 13–22, Washington, D.C., October 1991. IEEE Computer Society Press.

[Zuy93] van H.J. Zuylen, editor. *The REDO Compendium, Reverse Engineering for Software Maintenance*. John Wiley & Sons, 1993.