Methoden und Werkzeuge für die Software Migration

Erdmenger, U.; Kaiser, U.; Loos, A.; Uhlig, D.
pro et con Innovative Informatikanwendungen GmbH,
Annaberger Straße 240, 09125 Chemnitz
{uwe.erdmenger, andreas.loos, uwe.kaiser, denis.uhlig}@proetcon.de
www.proetcon.de

Abstract: Die Autoren entwickeln seit 1994 kommerziell Werkzeuge für die Software Migration und setzen diese in praktischen Migrationsprojekten ein. Obwohl diese Werkzeuge wesentlich mit der Compilierung vergleichbare Aufgaben wie Scannen, Parsen und Generierung realisieren, existieren Unterschiede in der Arbeitsweise von Migrationswerkzeugen zu denen klassischer Compiler. Der vorliegende Beitrag vermittelt einen Überblick über Werkzeuge für die Software Migration und beschreibt partiell Unterschiede der integrierten Konvertierungsmethoden zur klassischen Compilierung.

1 Software Migration im Überblick

Aktuell existiert eine Lücke zwischen antiquierter Individualsoftware (Legacy-Systeme) und modernen Technologien der Information und Kommunikation. Die Software Migration hat sich neben dem Einsatz von Standardsoftware und der Neuentwicklung als Alternative etabliert, diese technologische Lücke zu schließen. In [KKW07] wird Software Migration wie folgt definiert: „Software Migration beinhaltet neben der (teil)automatischen Konvertierung von Programmen aus antiquierten Programmiersprachen wie z.B. COBOL oder PL/I in moderne(re) Sprachen wie C++ oder Java unter anderem auch die Integration in neue Betriebssysteme, die Modernisierung der Datenhaltung, der Benutzeroberflächen und der Software Architektur." Diese Definition benennt mit Programmen, Daten und Benutzeroberflächen wesentliche Anwendungsgebiete der Software Migration. Wie andere verbale Definitionen auch ist diese Definition unvollständig im Detail. In Migrationsprojekten sind z.B. komplexe Prozeduren in einer Job Control Language (JCL) zur Steuerung der Batch-Verarbeitung ebenso zu behandeln, die obige Definition fokussiert diese Problematik jedoch nicht. Software Migration ist komplex, zu ihrer Beherrschung müssen Software-Werkzeuge zum Einsatz kommen. In den folgenden Kapiteln werden Werkzeuge für die folgenden Migrationsfelder vorgestellt:

- Programmkonvertierung,
- Datenmigration,
- Modernisierung von Benutzeroberflächen,
- JCL-Konvertierung.

2 Programmkonvertierung und Translatoren

2.1 Architektur von Translatoren

Eine Komponente von Migrationsprojekten ist die Konvertierung von Programmen des Basissystems (Quellprogramme) in Programme des Zielsystems (Zielprogramme). Eine wesentliche Entscheidung dabei ist, ob die Programmiersprache auf dem Zielsystem erhalten werden soll oder nicht. Die Praxis zeigt, daß z.B. COBOL trotz aller Kritiken meist auf dem Zielsystem erhalten bleibt. In diesem Fall muß lediglich eine Dialektanpassung erfolgen. Dafür reichen Mustererkennungen auf der Basis regulärer Ausdrücke aus. [Uh07] beschreibt ein Werkzeug, das verschiedene COBOL-Dialekte ineinander konvertiert. Gesteuert von einer graphischen Eclipse-Oberfläche werden die komplexen Mustererkennungsalgorithmen von Perl genutzt. Bei einem Wechsel der Programmiersprache versagt die Nutzung regulärer Ausdrücke, da es sich um komplexe Übersetzungsvorgänge handelt, welche die Anwendung von Übersetzertechniken erfordern. *Translatoren* sind Werkzeuge, die automatisiert Quellprogramme A in Zielprogramme B konvertieren. Die Praktikabilität von Translatoren wurde in mehreren Migrationsprojekten nachgewiesen, z.B. bei der Entwicklung eines Translators für System Programming Language (SPL) als Quell- und C++ als Zielsprache [Er06]. Abb. 1 zeigt die Architektur eines Translators:

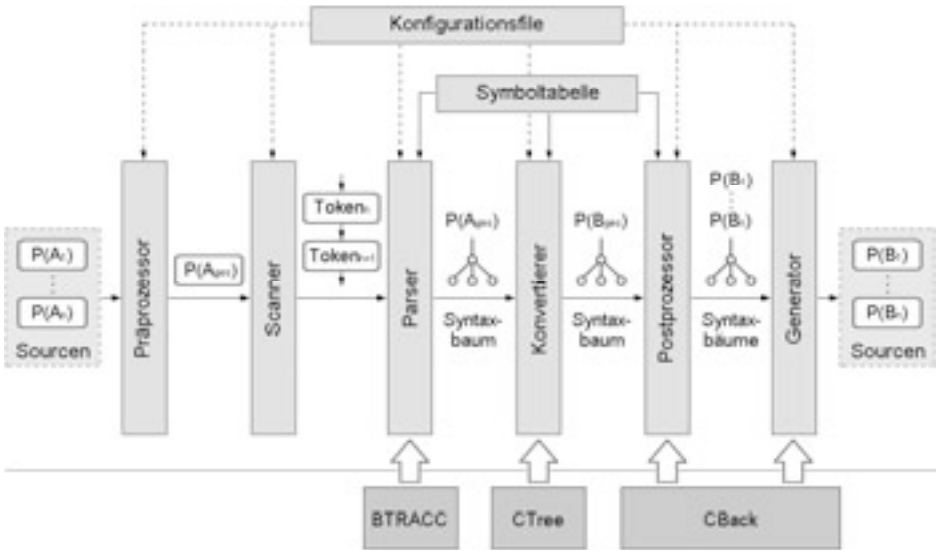


Abbildung 1: Architektur eines Translators

Die Architektur folgt dem klassischen Compilermodell. Scannen, Parsen und Symbolverwaltung sind klassische Compilerfunktionen. Es existieren jedoch auch Unterschiede:

Kommentarerhaltung: Präprozessor und Scanner verwerfen keine Kommentare, diese werden während des gesamten Konvertierungsprozesses erhalten, um sie optional in den späteren Zielcode einzufügen.

Erhaltung von Präprozessorinformationen: Wie beim klassischen Compiler werden Präprozessorbefehle vor dem Scannen ausgeführt. Diese Informationen sind im Konvertierungsprozeß ebenfalls zu erhalten. Quellprogramm-Makros z.B. müssen optional als Makros im Zielprogramm erscheinen und die Include-Befehle dienen in der Phase Generierung der Aufteilung des Zielprogrammes auf verschiedene Files. Praxiserfahrungen besagen, daß es aufgrund der o.g. Funktionalitäten nicht sinnvoll ist, Präprozessoren, Scanner etc. von existierenden Compilern für eine Translatorentwicklung zu verwenden. Sie sind prinzipiell neu zu entwickeln.

Schnittstelle zwischen Quell- und Zielrepräsentation: Es existiert eine strikte Trennung. Der Parser liefert einen attributierten Syntaxbaum des Quellprogrammes. Der Konvertierer realisiert daraus einen attributierten Syntaxbaum des Zielprogrammes.

Postprozessor: Diese Komponente ist neu gegenüber dem klassischen Compilermodell. Wesentliche Aufgaben sind die Zerteilung des vollständigen, attributierten Syntaxbaumes des Zielprogrammes in Teilsyntaxbäume, das Einfügen von Präprozessoranweisungen der Zielsprache und die optionale Integration der Kommentare des Basisprogrammes. Die Aktionen erfolgen auf der Basis von Syntaxbäumen und nicht auf der Basis von Files.

Generator: Der Generator schreibt physisch Sourcecode durch Traversieren einzelner Syntax(teil)bäume. Es entstehen bei einer Konvertierung nach z.B. C++ mehrere .h-Files und ein main-Programm. Da die Forderung nach Wartbarkeit existiert, Wartbarkeit aber unterschiedlich interpretiert wird, kann die formatierte Ausgabe entsprechend den Vorstellungen der Nutzer konfiguriert werden (Konfigurationsfile).

Abstrahierend aus der Vielzahl technischer Details soll die Arbeitsweise des Postprozessors expliziert werden: Die Aufgabe eines Präprozessors im klassischen Compiler ist es, Präprozessoranweisungen auszuführen, so daß im Ergebnis ein einziges Quellcodefile ohne Präprozessoranweisungen existiert. Präprozessoranweisungen beinhalten wichtige Informationen, z.B., aus welchen Includefiles sich das Quellprogramm zusammensetzt, welche Makros an welcher Position in den Includefiles auftreten und welche Kommentare an welcher Position vorkommen. Im Gegensatz zur klassischen Compilierung werden diese Informationen während der Translation erhalten. Präprozessor und Scanner fügen dazu neue Token in den Tokenstrom ein, welche der Parser überliest. Nach dem Parsen werden sie anhand der Quelltextposition den Knoten des Syntaxbaumes des Quellprogrammes zugeordnet. Der Konvertierer plaziert diese an die korrespondierenden Knoten des Zielsyntaxbaumes. Diese Informationen verwendet der Postprozessor für folgende Aufgaben:

Zerteilung des Zielsyntaxbaumes: Es entsteht für jedes Includefile ein Teilsyntaxbaum mit semantisch äquivalentem Inhalt. Dazu sind aus dem Zielcodebaum Teilbäume für Includefiles abzuspalten und an deren Stelle im Zielcodebaum Syntaxteilbäume für Include-Anweisungen der Zielsprache einzufügen:

```
... (SPL-Main-File) ...     ... (C++-Main-File) ...
 DCL 1 PERSON,              struct {
       %INCLUDE BSP;        #include "bsp.hpp"
                            };
... (SPL-Includefile) ...   ... (C++-Includefile) ...
  2 NAME CHAR(10),          TFixString<10>  name;
  2 ALTER                   short alter;
    BINARY FIXED(15);
```

Plazieren von Makros: Im Zielsyntaxbaum sind alle Teilbäume, welche aus Makros im Quellprogramm hervorgegangen sind, annotiert. Der Postprozessor vergleicht diese Teilbäume, definiert ein Makro der Zielsprache und ersetzt die Syntaxbäume durch Syntaxbäume der entsprechenden Makroaufrufe. Der Teilsyntaxbaum des Makros wird an die entsprechende Stelle des Syntaxbaumes eingefügt.

Kommentarerhaltung: Die Kommentare werden als Vor- und Nachkommentar den einzelnen Teilbäumen des Zielprogrammes heuristisch zugeordnet. Vom Generator werden sie dann vor oder nach dem entsprechenden Zielcodefragment plaziert.

Bei der Entwicklung von Translatoren und weiteren Reengineering-Werkzeugen werden von den Autoren folgende, eigenentwickelte Generierungstools (Meta-Tools) eingesetzt, sie sind ebenfalls in Abbildung 1 dokumentiert:

BTRACC: Backtracking Compiler Compiler, ein Parsergenerator auf Basis des Backtracking-Verfahrens.

CTree: Ein Werkzeug zur deklarativen Beschreibung von Syntaxbaummodellen.

CBack: Ein Werkzeug zur Generierung strukturierten C/C++-Codes aus Syntaxbäumen. Die Formatierung des Codes ist optional einstellbar.

Erfahrungen besagen, daß der Entwicklungsaufwand für Translatoren ca. 3 bis 3.5 Mannjahre beträgt. Er reduziert sich bei Nutzung von Meta-Tools um ca. 25 %.

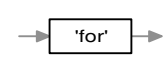
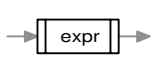
2.2 BTRACC - Parsergenerator auf Backtracking-Basis

Im weiteren soll die Funktionalität des Parsergenerators BTRACC diskutiert werden. Folgende Anforderungen existieren bei der Translatorentwicklung an den Parsergenerator:

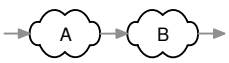
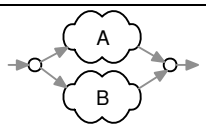
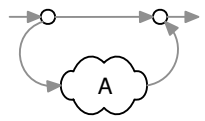
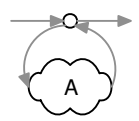
- Verarbeitung komplexer Grammatiken (insbesondere bei antiquierten Quellsprachen wie z.B. COBOL und PL1),
- Verwendung der in Dokumentationen vorgegebenen Grammatik ohne aufwendige Umstellung zur Erreichung einer LL(n)- oder LR(n)-Eigenschaft,
- Verarbeitung mehrerer Dialekte in einem Werkzeug,
- Einfache Integration neuer, syntaktischer Konstruktionen ohne Umbau der existierenden Grammatik.

Als Open Source verfügbare Parsergeneratoren (YACC, COCO/R, PCCTS,...) werden diesen Anforderungen nicht vollständig gerecht. Insbesondere die Forderung, die in Dokumentationen verfügbare Syntax ohne Umstellungen zu verwenden und Erweiterungen komfortabel integrieren zu können, ist nicht erfüllt. Eine Erweiterung einer komplexen YACC-Grammatik z.B. führt in der Regel zu einer Reihe von Konflikten, welche beseitigt werden müssen. Diese Argumente waren die Motivation für die Entwicklung eines neuen Parsergenerators BTRACC:

BTRACC akzeptiert Grammatiken in Erweiterter Backus-Naur-Form (EBNF). Diese ist gemäß unserer Erfahrungen leicht aus gegebenen Sprachdokumentationen zu extrahieren. BTRACC liest diese Grammatikbeschreibung ein, prüft ihre syntaktische Korrektheit und baut eine interne Darstellung auf. Diese kann als Graph mit verschiedenen Knotenarten interpretiert werden:

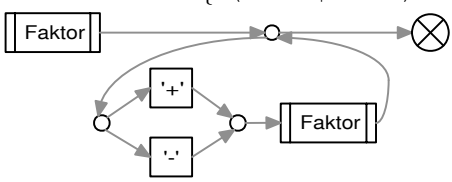
Terminal	Nichtterminal	Regelende	Verzweigung und Sammelknoten
			

Aus diesen Grundbausteinen wird zu jeder Regel ein Graph aufgebaut, welcher die rechte Regelseite widerspiegelt. Diese Regeln werden, sortiert nach ihrem Namen, in einer Hash-Liste verwaltet, welche einen schnellen Zugriff über den Regelnamen (Nichtterminal) gestattet.
Die Strukturen der Backus-Naur-Form werden wie folgt zusammengesetzt:

Sequenz `A B`	Alternative `A|B`	Option `[A]`	Wiederholung {`A`}
			

Dabei stehen `A` und `B` für Terminalsymbol-Knoten, Nichtterminalsymbol-Knoten oder komplette Strukturen (rekursive Definition). Die Verbindungen zwischen den Knoten (Kanten) besitzen keine Attribute und sind daher einfach als C-Zeiger implementiert. Mit diesen Festlegungen ergibt sich die folgende, interne Darstellung:

```
Term = Faktor { ( '+' | '-' ) Faktor }.
```



Zur Generierung des Parsers wird der Graph in „depth first order"- Richtung durchlaufen und zu jedem Knoten wird ein Befehl für eine virtuelle Maschine generiert. Diese Befehle werden als Feld von Strukturen abgebildet, wobei jeder Befehl eine **Befehlsart**, einen oder (bei Entscheidungsknoten) zwei **Nachfolgebefehle** (Indizes anderer Befehle im Feld) und (bei Terminalsymbolen) ein zu konsumierendes Token besitzt. Wichtige Befehle sind:

BRANCH: Der Befehl entsteht aus Verzweigungsknoten. Er hat zwei Nachfolger und wird abgearbeitet, indem zuerst der Index des zweiten Nachfolgers als *Entscheidungspunkt* auf einem Stack abgelegt wird und dann mit dem ersten Nachfolger fortgefahren wird.

TEST: Der Befehl entsteht aus Terminalknoten. Das nächste Token im Eingabestrom wird mit dem aktuellen Token verglichen. Bei Gleichheit wird mit dem (einzigen) Nachfolger fortgesetzt, bei Ungleichheit mit dem Befehl, dessen Index im letzten Entscheidungspunkt

auf dem Stack steht. Dieser wird dabei vom Stack entfernt (Backtracking). Ist kein Entscheidungspunkt mehr auf dem Stack, wird die Abarbeitung mit negativem Ergebnis (Eingabe nicht erkannt) beendet.

CALL: Er entsteht aus Nichtterminalen. Bei der Abarbeitung wird ein *Callframe* mit dem zweiten Nachfolger auf dem Stack abgelegt und mit dem ersten fortgesetzt.

RETURN: Der Befehl entsteht aus Endknoten. Er wird abgearbeitet, indem der Index des Nachfolgers aus dem obersten, aktiven Callframe ermittelt wird. Der Callframe bleibt auf dem Stack erhalten, aber ist nun inaktiv, damit er beim nächsten RETURN nicht wieder verwendet wird. Die Abarbeitung endet mit positivem Ergebnis (Eingabe geparst), wenn kein aktiver Callframe mehr vorhanden ist.

Während des Parsens erfolgt keine Attributverarbeitung. Grund dafür ist, daß die Attributberechnung (welche z.B. Syntaxbäume aufbaut) im Backtracking-Fall nur mit großem Aufwand rückgängig gemacht werden kann. Daher erfolgt die Attributberechnung in einem separaten Durchlauf nach erfolgreichem Parsen. Es ist zu beachten, daß in diesem zweiten Durchlauf noch der Stackinhalt zur Verfügung steht, der im ersten Durchlauf aufgebaut wurde. Dieser wird nun als „roter Faden“ verwendet, so daß der Durchlauf deterministisch erfolgt.

Zum Zweck der Attributierung generiert BTRACC zu jeder Regel eine Funktion. Die Regelparameter sind die Parameter der Funktion und die Berechnung der Attribute geschieht innerhalb des Funktionsblocks. Die Funktionen rufen sich gegenseitig auf (analog zum Verfahren des rekursiven Abstiegs, welches auch von LL(1)-Parsergeneratoren verwendet wird).

Die Funktionen ändern die Variablen, z.B. die Stackgröße, manipulieren aber nicht den Stackinhalt. Im Falle einer Verzweigung (es existiert ein alternativer Weg), wird verglichen, ob dieser Weg im „stackobersten“ Entscheidungspunkt steht und die dort angegebene Position in der Tokenliste der aktuellen Position entspricht. Ist dies der Fall, so führte in der Parsing-Phase der erste Weg zum Ziel. Dieser ist nun auch hier zu wählen. Ist es nicht der Fall, so ist der alternative Weg der korrekte.

Im Backtracking-Fall muß auch auf das entsprechende Token im Tokenstrom zurückgesetzt werden. Entsprechende Informationen werden in den Entscheidungspunkten auf dem Stack vermerkt. Die bereits verarbeiteten Token werden in einer Liste zu diesem Zweck zwischengespeichert.

Zusammenfassend ist zu sagen, daß sich dieses Verfahren bewährt hat. Die nichtdeterministische Arbeitsweise führt nur zu geringem Overhead, gemessen an der Gesamtlaufzeit eines Translators. Durch Berücksichtigung der FIRST-Mengen der Nachfolger eines BRANCH-Befehls, die statisch ermittelt werden können, und deren Vergleich mit dem aktuellen Token, werden überflüssige „Irrwege“ vermieden und die Performance verbessert.

Mit BTRACC wurden Parser für COBOL, PL1, SPL, TAL, C, SQL, NATURAL und JAVA entwickelt und in kommerzielle Migrations- und Reengineering-Werkzeuge integriert.

3 Ausgewählte Aspekte der Datenmigration

Die Datenmigration ist wichtiger Bestandteil eines jeden Migrationsprojektes. Es müssen zwei Aspekte betrachtet werden: Die Migration der Datenverwaltungssysteme und die Migration der Datenstrukturen in den Programmen. Ziel ist eine vollautomatische, hochperformante Migration der Daten mit folgenden Aspekten:

- Sie kann zwar in der Migrationsphase zu Testzwecken regelmäßig erfolgen, zum Zeitpunkt der Übernahme in den produktiven Betrieb muß eine „Big Bang"-Migration durchgeführt werden, da während der Datenmigration ein produktiver Betrieb im allgemeinen nicht möglich ist.
- Ein Fallback-Szenario, bei dem die Daten vom Ziel- auf das Basissystem zurückmigriert werden, ist möglich, aber mit enormen Zeitverlusten verbunden. Aus diesem Grund muß die Datenmigration umfassenden Tests unterworfen werden.
- Werkzeuge zur Datenmigration werden nur während eines kurzen Zeitraumes und von IT-Experten genutzt. Aus diesem Grund sind aufwendige GUIs, Dokumentationen etc. nicht notwendig.

Die genannten Aspekte schließen eine manuelle Datenmigration aus.

3.1 Migration der Datenstrukturen

Während für die Programmkonvertierung i.d.R. eine kontextfreie Grammatik als formale Beschreibungsform genutzt wird, ist das für Datenstrukturen nicht möglich. In Legacy-Systemen liegen Datenbestände einerseits in Datenbanken vor (hier existieren i.d.R. DDL-Statements, aus denen die Datenstrukturierung ermittelt wird), andererseits in Files. Files sind dabei im klassischen Sinne als „Folge von Bytes" zu verstehen. Wie einzelne Bytefolgen im Sinne der Datensatzstruktur zu interpretieren sind (textuelles Zeichen, Festkommazahl, Gleitkommazahl, ...), kann selten aus Dokumentationen ermittelt werden. Eine andere Quelle böte das systematisierte Know-how der Entwickler. Auch diese Quelle versagt oft, da die Entwickler nicht mehr zur Verfügung stehen. Deshalb bildet die feingranulare, werkzeugunterstützte Quellprogrammanalyse in Bezug auf die Schnittstellen zu den Daten eine wesentliche Quelle zur Bestimmung der Datensatzstruktur. Programmierer komplexer COBOL- oder PL1-Programme nutzen häufig ein simples Schema:

- Ein Datensatz wird in einen Speicherbereich geladen.
- Im Programm wird eine Datenstruktur definiert, die über diesem Puffer positioniert wird (Überlagerungen). Anhand des an einer bestimmten Byteposition liegenden Datentyps wird die zugehörige Bytefolge interpretiert.

Die Bestimmung des Datensatzaufbaus durch statische Analysen der Quellprogramme birgt Risiken, da Überlagerungen auch dynamisch sein können. Deshalb wird als zusätzliche

Informationsquelle eine heuristische Analyse der Files herangezogen. Dabei wird das File als $m \times n$-Matrix interpretiert. m beschreibt die Zahl der Datensätze, n die Maximallänge eines Datensatzes im File. Jeder Datensatz D kann mit $D = (d_{k1}, d_{k2}, d_{k3}, ..., d_{kn})$ als Zeilenvektor interpretiert werden.

Ein Spaltenvektor S mit $S = (s_{1j}, s_{2j}, ..., s_{mj})$ beschreibt alle Bytes an einer bestimmten Byteposition. Über die Analyse eines Spaltenvektors (Häufigkeit des Auftretens einzelner Bytes) kann heuristisch darauf geschlossen werden, ob es sich um Text- oder Binärzeichen handelt. Diese Kenntnis ist insbesondere dann notwendig, wenn mit der Datenmigration eine Migration der Zeichenkodierung (z.B. von EBCDIC nach ASCII) verbunden ist. Während textuelle Zeichen konvertiert werden müssen, können binäre Zeichen unverändert bleiben, wenn sie bspw. als Ganzzahl interpretiert werden. Voraussetzung ist, daß Quell- und Zielarchitektur über identische Zahlendarstellungen verfügen und identische Byte-Reihenfolge aufweisen. Ist eines der Systeme als Big- und das andere als Little-Endian-Architektur ausgelegt, können Daten nicht unverändert migriert werden. Das beschriebene Analyseverfahren wurde erfolgreich angewendet, es existiert dafür ein Analysewerkzeug [Lo05].

3.2 Migration der Datenverwaltungssysteme

Die Datenverwaltung in Zielsystemen fokussiert meist auf die Nutzung von Datenbasismanagementsystemen (DBMS). Bei Neuentwicklungen basieren diese auf relationalen Entwürfen, wobei die redundanzfreie Speicherung der Daten im Vordergrund steht. Dabei stellt das Datenmodell eine Abstraktion der Geschäftsprozesse dar. Legacy-Systemen liegt selten ein relationales Modell zugrunde, so daß sich für die Datenmigration in erster Näherung ein relationales Redesign empfiehlt, welches aber Migrationsrisiken beinhaltet:

- Mit einem Redesign entstehen neue Anforderungen an die Programm-Schnittstelle. Häufig verwenden Programme Datenstrukturen, die eine genaue Kenntnis der persistenten Abspeicherung voraussetzen. Ändert sich die Struktur, sind Modifikationen im Sourcecode notwendig, welche eine automatische Konvertierung der Programme erschweren.
- Der Aufwand für ein relationales Redesign ist identisch mit dem eines neuen Entwurfs, welcher alle Stufen der Qualitätssicherung durchlaufen muß, um Fehler zu minimieren. Das Legacy-Datenverwaltungssystem hingegen stellt ein über Jahre ausgetestetes System dar und arbeitet weitgehend fehlerfrei.

Ob ein Redesign oder eine strukturelle 1:1-Migration sinnvoll ist, muß für jedes Szenario neu entschieden werden. Hier spielen auch Aspekte wie Projektdauer und -budget eine Rolle. In einem aktuellen Migrationsprojekt (Vgl. [Te07]) wurde die 1:1-Migration des Datenverwaltungssystems mit folgenden Prämissen gewählt:

- Die Dateien des Basissystems sind satzorientiert. Jeder Datensatz enthält einen textuellen oder numerischen Schlüssel.

- Das Zielsystem nutzt ein relationales DBMS. Jeder Datei des Basissystems wird im Zielsystem eine Datenbanktabelle zugeordnet. Sie enthält neben Verwaltungsdaten zwei Attribute: Den Datensatz aus der Datei des Basissystems und den Schlüssel, welcher aus dem Datensatz extrahiert und in einem separaten Attribut gespeichert wird.
- Die Schlüsselattribute werden mit einem Index versehen, um effiziente Zugriffe zu garantieren. Hier sind in jedem Fall Analysen vorzunehmen, welche das Verhältnis zwischen lesenden und modifizierenden Statements dokumentieren, da bei überwiegendem Update-Anteil der Overhead für die Reorganisation der Indizes steigt.

Dieses Datenmodell stellt somit kein Redesign der Geschäftsprozesse dar, sondern entspricht dem Datenmodell des Basissystems. Dieses Manko wird durch minimierte Risiken in der Programmmigration ausgeglichen.

3.3 Performance

Legacy-Systeme verfügen häufig über ein hochoptimiertes Filemanagement, welches auf die individuellen Geschäftsprozesse zugeschnitten ist. Inwieweit eine DBMS-Lösung im Zielsystem für transaktionsorientierte, interaktive Anwendungen die notwendige Performance liefern kann, muß durch ausgedehnte Lasttests nachgewiesen werden. I.d.R. liegt keine zeitliche Gleichverteilung der Transaktionszahlen vor, so daß ausgehend von Spitzenzeiten „Worst Case“-Szenarien entwickelt werden müssen, welche die Grundlage dieser Lasttests darstellen. Im genannten Projekt konnte nachgewiesen werden, daß mit reinen Datenbankmitteln der geforderte Durchsatz nicht erreicht werden konnte, obwohl alle Möglichkeiten der Leistungsverbesserung, die datenbank- und hardwareseitig zur Verfügung standen, ausgeschöpft wurden. Aus diesem Grund wurde für häufig genutzte Tabellen ein zusätzlicher Shared Memory Pool implementiert, der Daten im Hauptspeicher puffert und damit Performanceverluste, die hauptsächlich aus I/O-Operationen des DBMS resultieren, kompensiert. Da parallele Prozesse an die Datenhaltung gekoppelt sind, müssen geeignete Mechanismen zur Prozeßsynchronisation genutzt werden.

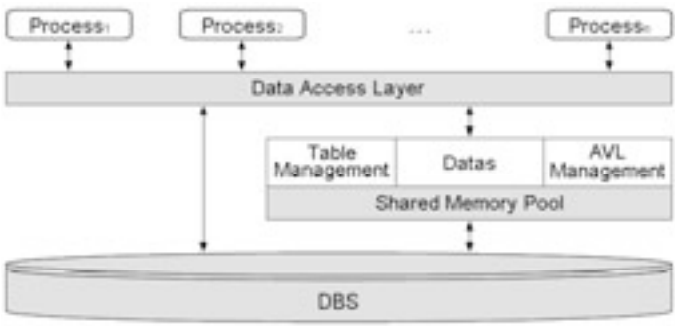


Abbildung 2: Architektur des Datenverwaltungssystems

In Abbildung 2 nutzen alle Prozesse eine definierte Zugriffsschicht (Data Access Layer), die ihrerseits direkt oder indirekt über den Shared Memory Pool mit dem Datenbanksystem (DBS) kommuniziert. Beim Entwurf des Shared Memory Pools waren die Aspekte umfassender Transaktionssicherheit und Entwicklungsaufwand gegeneinander abzuwägen. Dabei wurden folgende Designentscheidungen getroffen: Der Shared Memory Pool wird über Semaphore geschützt. Konkurrierende Prozesse erhalten so auf kritische Abschnitte einen exklusiven Zugriff. Die effiziente Lokalisierung einzelner Datensätze wird über AVL-Bäume realisiert.

Mit dem Start des ersten Prozesses werden alle notwendigen Tabellen aus dem DBS in den Shared Memory Pool geladen. Die daraus resultierenden Performanceverluste können toleriert werden, da dieser Vorgang einmalig ist und weitere Prozesse einen gefüllten Pool vorfinden.

Es erfolgt kein implizites Rückschreiben der Daten in das DBS, etwa beim Transaktionsende. Stattdessen werden Methoden definiert, die zu gewissen Zeitpunkten explizit die Synchronisation zwischen DBS und Shared Memory Pool vornehmen. Diese Festlegung impliziert, daß Tabellen, deren Daten im Shared Memory Pool enthalten sind, zu keinem Zeitpunkt direkt im DBS manipuliert werden dürfen.

3.4 Abschließende Bemerkungen

Für die Datenmigration wurde von den Autoren ein Satz von Werkzeugen entwickelt, eine detaillierte Beschreibung liefert [Lo05].

4 Konvertierung von Host-Masken in moderne Benutzeroberflächen

Die Konvertierung ASCII-orientierter Host-Masken in browserbasierte Benutzeroberflächen soll am Beispiel von SCREEN COBOL expliziert werden. Mit diesem COBOL-Dialekt werden Masken z.B. in HP NonStop-Systemen (Tandem) entwickelt. Die Konvertierungstechnologie läßt sich jedoch vom konkreten Basissystem abstrahieren und verallgemeinern. Abbildung 3 dokumentiert das Basissystem:

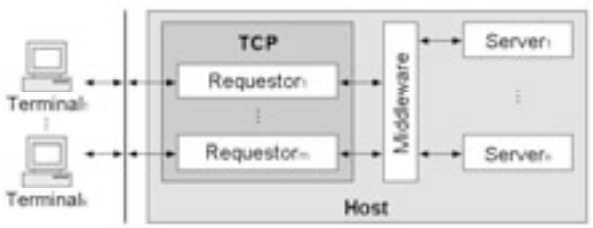


Abbildung 3: HP NonStop-Basissystem

Die in SCREEN COBOL geschriebenen Maskenprogramme (sogenannte Requestoren) laufen unter Steuerung eines Terminal-Control-Prozesses (TCP) und zeigen an den über eine Terminalemulation angeschlossenen Terminals die Maskeninhalte an. Erfolgen in den Masken Eingaben, werden diese an den Requestor übertragen. Unter Einbindung einer Middleware wird daraufhin dem Server (ein transaktionsorientiertes Programm) eine Message gesendet. Der Server realisiert die Verarbeitung und sendet eine Antwortmessage. Diese wird vom Requestor verarbeitet und als Ergebnis werden neue Daten in der aktuellen Maske angezeigt oder eine andere Maske aufgeschaltet etc. SCREEN COBOL-Programme beinhalten zwei für den Konvertierungsprozeß wichtige Informationen:

- Die Anordnung der Ein- und Ausgabefelder in der Maske (das „Layout“) einschließlich ihrer Eigenschaften.
- Die COBOL-Strukturen, in welchen die Ein- und Ausgabeinformationen gespeichert werden.

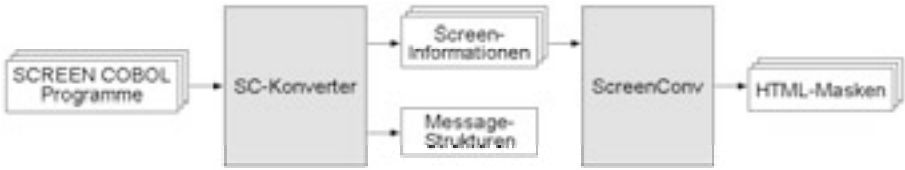


Abbildung 4: Technologie der Maskenkonvertierung

Für die Konvertierung sind diese Informationen wesentlich, sie werden automatisiert aus den SCREEN COBOL-Programmen extrahiert. Abbildung 4 verdeutlicht das: Es existiert ein Werkzeug *SC-Konverter*, Hauptbestandteil ist ein front-end für SCREEN COBOL-Programme. SC-Konverter liefert zwei Ergebnisfiles:

- *Message-Strukturen* beinhalten in aufbereiteter Form die o.g. COBOL-Strukturen.
- *Screen-Informationen* beinhalten Informationen zum Maskenlayout. Daraus generiert ein Werkzeug *ScreenConv* HTML-Code für die Masken im Zielsystem (*HTML-Masken*).

Message-Strukturen und Screen-Informationen werden im Zielsystem genutzt.

In einem konkreten Migrationsprojekt existierten für die Gestaltung der Benutzerkommunikation im Zielsystem die folgenden Nutzerforderungen:

- Der Maskenaufbau sollte erhalten bleiben. Der Nutzer motivierte das mit einer kurzen Einarbeitungszeit.
- Die Lauffähigkeit im Webbrowser im Zielsystem sollte ohne zusätzliche Installation auf dem Clienten gegeben sein. Hier spielten reduzierter Administrationsaufwand und Einsparung von Lizenzkosten eine Rolle.

In Abbildung 5 wird die Zielsystemarchitektur des Migrationsprojektes dokumentiert.

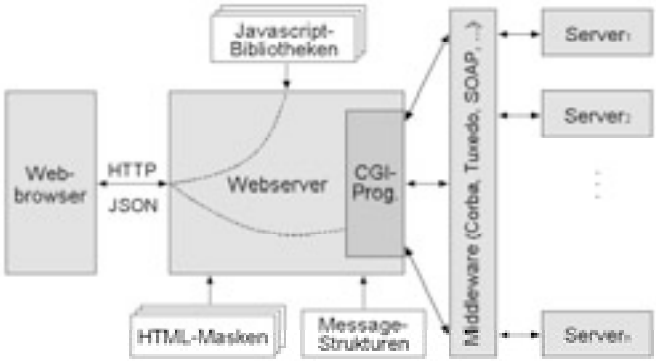


Abbildung 5: Architektur des Zielsystems

Für die Kommunikation zwischen Webbrowser und Webserver wird AJAX genutzt. Das verhindert unnötige Reloads von kompletten Seiten und den Eintrag in die History des Browsers, die im Basissystem auch nicht vorhanden ist.

Die Kommunikation stellt sich im Zielsystem dann wie folgt dar:
Eine Maske wird im Browser angezeigt. Ihr HTML-Code wurde auf Basis der SCREEN COBOL-Programme des Basissystems (teil)automatisch generiert und steht als HTML-File zur Verfügung. Die Maske nutzt eine Javascript-Bibliothek für die Kommunikation mit dem Server und eine maskenspezifische Javascript-Bibliothek für Aktionen wie das Auslesen von Eingaben, das Setzen von neuen Werten in der Maske sowie die Steuerung. Beide Bibliotheken sind in Zielsystemen obiger Architektur wiederholt einsetzbar. Werden Eingaben in der Maske getätigt und mit einer dem Basissystem entlehnten „Datenfreigabe“ (ENTER-Taste) beendet, so werden die Maskendaten über einen AJAX-HTTP-Request an den Webserver gesendet. Für diese Kommunikation wird JavaScript Object Notation (JSON) als Alternative zu XML verwendet. Der Webserver übergibt die Daten an ein CGI-Programm, welches die Message vom JSON-Format in eine COBOL-Message konvertiert. Eine COBOL-Message ist vereinfacht eine mit Daten gefüllte COBOL-Struktur. Die für das Füllen der COBOL-Struktur notwendigen Informationen (Offset, Länge, Typ der einzelnen Datenfelder) stehen in Message-Strukturen zur Verfügung. Sie wurden im Konvertierungsprozeß aus den SCREEN COBOL-Programmen gewonnen. Die mit Inhalten gefüllte COBOL-Struktur wird über eine Middleware (CORBA, TUXEDO,...) an den COBOL-Server gesendet. In der Referenzimplementierung wird anstelle einer kommerziellen Middleware das handliche SOAP-Protokoll verwendet. Der Server verarbeitet die Message, erzeugt eine Antwortstruktur und sendet diese zurück zum CGI-Programm. Dieses erzeugt aus den Daten wieder eine JSON-Struktur und schickt sie mit HTTP-Request an den Browser. Über Javascript-Funktionen werden die entsprechenden Daten zur Anzeige gebracht. Soll auf Grund der Serverantwort eine neue Maske (HTML-Seite) angezeigt werden, wird diese vorher durch einen weiteren HTTP-Request geladen und angezeigt.

Erfahrungen mit dieser Zielarchitektur besagen, daß die Konvertierung von SCREEN COBOL nach HTML und Javascript möglich ist und eine Reihe von Vorteilen (Unabhängigkeit von proprietären, kostenpflichtigen Maskentools bzw. Emulatoren, Einsatz von Standardwerkzeugen und -techniken, . . .) bietet.

5 Konvertierung von JCL-Prozeduren

5.1 Eigenschaften von Job Control Sprachen

Migrationslösungen für Job Control Languages (JCLs) werden in Veröffentlichungen stiefmütterlich behandelt. Dabei sind die in einer JCL formulierten und auf der untersten Ebene des Basissystems angesiedelten Prozeduren und Jobs für eine Reihe wesentlicher Aufgaben verantwortlich. Dazu gehören u.a. die Ablauforganisation und Protokollierung von Programmen sowie die Sicherung, Archivierung und Verteilung von Daten. Aufgrund des breiten Aufgabenspektrums haben Job-Netze einen erheblichen Umfang. Das motiviert eine nähere Betrachtung aus der Sicht der Software Migration. JCLs zeichnen sich durch folgende Eigenschaften aus:

Syntaktische Komplexität: JCLs werden interpretativ abgearbeitet. Es wird ein befehlsorientierter Ansatz verfolgt. Daraus resultiert eine im Vergleich zu Programmiersprachen geringere Komplexität auf Befehlsebene, obwohl das Spektrum unterschiedlicher Befehle für JCLs breit ist. Des weiteren bieten JCLs oft nur simple Datentypen wie Skalare und Listen.

Abhängigkeit vom Betriebssystem: In JCLs wird ungekapselt auf Betriebssystem-Kommandos zugegriffen. Der Einsatz von sogenannten Jobvariablen zur Interprozeßkommunikation ist ein Beispiel dafür. Die Jobs sind damit schwerer vom Betriebssystem zu trennen als Programme, die meist über Bibliotheken auf Betriebssystemfunktionen zugreifen.

Einbettung externer Programme: Jobs werden u.a. dazu benutzt, Programmabläufe zu automatisieren. Dazu existiert die Möglichkeit, Programmaufrufe in die Jobs einzubetten. Beispiele dafür sind Sortierprogramme oder Editoren. Daraus ergibt sich eine Kopplung der Jobs an das Betriebssystem selbst und an die genutzten Programme.

5.2 S2P - SDF to Perl Translator

In einem Migrationsprojekt [EU07] galt es, die im Betriebssystem BS2000 (Basissystem) eingesetzte System Dialog Facility (SDF) im UNIX-Zielsystem geeignet abzulösen. SDF entspricht den o.g. Eigenschaften einer JCL. Unter UNIX wurde eine Sprache benötigt, welche die wesentlichen Eigenschaften von SDF abbilden kann und darüber hinaus Möglichkeiten bietet, die in UNIX nicht unterstützten SDF-Funktionen zu emulieren. Die Entscheidung fiel auf Perl, da Perl allen o.g. Anforderungen entspricht und mit dem „Comprehensive Perl Archive Network (CPAN)“ eine internationale Plattform für den Austausch von Technologien bietet.

Für die Konvertierung von SDF-Prozeduren in semantisch äquivalente Perl-Scripts wurde ein Werkzeug *SDF To Perl Translator* (S2P) entwickelt. SDF bietet zwar eine Vielfalt unterschiedlicher Befehle, besitzt aber eine einfache, befehlsorientierte, syntaktische Struktur. S2P wurde mit dem in Perl verfügbaren Parsergenerator *Parse::RecDescent* entwickelt, da BTRACC in der aktuellen Version kein Perl als Zielsprache unterstützt.

Parse::RecDescent generiert aus einer attributierten, in einer EBNF-ähnlichen Sprache formulierten Grammatik einen Parser als Perl-Modul. Abbildung 6 zeigt die Architektur von S2P:

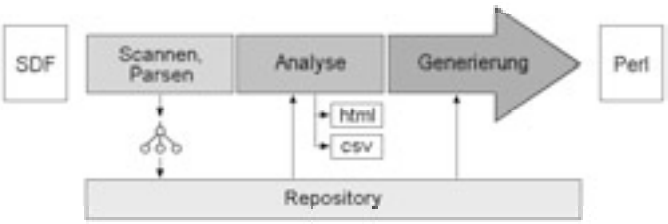


Abbildung 6: Architektur von S2P

Die Verarbeitung beginnt mit der lexikalen Analyse der SDF-Prozedur. S2P realisiert das in Analogie zum SDF-Interpreter befehlsweise. Anschließend wird jeder Befehl syntaktisch analysiert. Es wird ein Syntaxbaum des SDF-Befehls in Form von Perl-Arrays und -Hashes aufgebaut. Diese werden anschließend in einem Repository abgelegt.

Die folgende Verarbeitungsphase dient dem Reverse Engineering und der Redokumentation des existierenden Bestandes an SDF-Prozeduren. Dabei können Einzel- und Batchanalysen über dem Repository durchgeführt werden. Die Ergebnisaufbereitung erfolgt in Form von CSV- und HTML-Dateien. Ein Beispiel für eine Batchanalyse ist das Lokalisieren „ähnlicher“ Jobs - sogenannter Clones. Die Analyse erfolgt über alle im Repository enthaltenen Prozeduren. Die Cloneanalyse basiert auf einem befehlsweisen Vergleich der SDF-Prozeduren unter Ausschluß der Kommentare. Das CPAN-Modul *Algorithm::Diff* dient als Grundlage des Algorithmus. Die Analyse findet identische Befehlsfolgen in zu vergleichenden SDF-Prozeduren. Der Grad der Übereinstimmung wird mittels Parameter gesteuert. Weiter werden die Beziehungen zwischen Clones analysiert, um größere Clone-Gruppen zu ermitteln. Hintergrund der Cloneanalyse ist das Auffinden von Redundanzen im SDF-Bestand, welche vor der eigentlichen Konvertierung beseitigt werden können.

Der letzte Verarbeitungsschritt realisiert die Konvertierung der SDF-Prozeduren nach Perl. Im Ergebnis entsteht ein semantisch äquivalenter Perl-Job. Bei der Konvertierung der einzelnen Befehle werden zwei Typen unterschieden: SDF-Befehle, die ein Äquivalent in Perl besitzen, werden in dieses konvertiert. So wird z.B. der SDF-Befehl

`SKIP-COMMANDS TO-LABEL=ENDE` in Perl als `goto ENDE;` abgebildet.

Existiert keine Entsprechung in Perl, wird für den Befehl ein Funktionsaufruf generiert:

```
SDF: / ADD_FILE_LINK LINK-NAME=OUTFILE, FILE_NAME=I.DTA.OUT

Perl: add_file_link(LINK_NAME=>'OUTFILE', FILE_NAME=>'I.DTA.OUT');
```

5.3 Die Laufzeitbibliothek im Zielsystem

Im Zielsystem wird eine Laufzeitbibliothek als Schnittstelle zwischen Betriebssystem und konvertierten Perl-Scripts installiert. Diese folgt einem 2-Schichten-Modell.

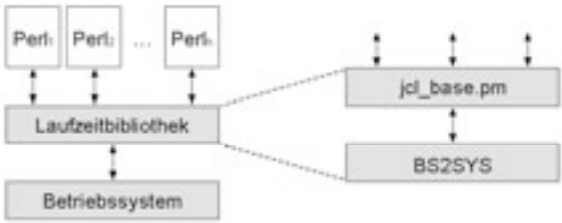


Abbildung 7: 2-Schichten-Modell der Laufzeitbibliothek

Systemnahe Komponenten (z. B. die Emulation von Jobvariablen) sind in *BS2SYS* enthalten. BS2SYS wurde aus Performancegründen in C implementiert. Das Perl-Modul *jcl_base.pm* stellt die Perl-Implementationen der SDF-Befehle bereit, welche nicht direkt in einen Perl-Befehl konvertiert werden können (z.B. `IF ...`). $Perl_i$ beschreiben die im Ergebnis der Konvertierung entstandenen Perl-Scripts.
S2P wurde erfolgreich in mehreren BS2000-Migrationsprojekten eingesetzt.

6 Fazit

Der Artikel beschreibt ausgewählte Aspekte der Software Migration, einige Ansätze wie z.B. die architekturbasierte Migration wurden nicht behandelt. Die beschriebenen Werkzeuge werden in kommerziellen Projekten, insbesondere für die Plattformen BS2000 und HP NonStop, erfolgreich eingesetzt [Te07], [Su05]. Zukünftige Arbeiten verfolgen das Ziel, auf Basis der vorgestellten Technologien Werkzeuge zur Unterstützung weiterer Plattformen zu entwickeln und sie in eine Software-Reengineering-Architektur zu integrieren.

Literatur

[Er06] Erdmenger, U.: SPL-Sprachkonvertierung im Rahmen einer BS2000-Migration. GI - Softwaretechnik-Trends, Band 26, Heft 2, ISSN 0720-8928, Mai 2006.

[EU07] Erdmenger, U., Uhlig, D.: Konvertierung der Jobsteuerung am Beispiel einer BS2000-Migration. GI - Softwaretechnik-Trends, Band 27, Heft 2, ISSN 0720-8928, Mai 2007.

[KKW07] Kaiser, U., Kroha, P., Winter, A. (Hrsg.): GI - Softwaretechnik-Trends, Band 27, Heft 1, ISSN 0720-8928, Februar 2007.

[Lo05] Loos, A.: Migration von Host-Dateien in relationale Datenbanksysteme am Beispiel einer BS2000-Migration. Fachberichte Informatik, Universität Koblenz-Landau, 15/2005.

[Su05] Sum, R.: Das Migrations-Projekt „Harmonisierung und Integration von Datenbanken bei MAN/TDB“. Fachberichte Informatik, Universität Koblenz-Landau, 15/2005.

[Te07] Teppe, W.: ARNO: Migration von Mainframe Transaktionssystemen nach UNIX. GI - Softwaretechnik-Trends, Band 27, Heft 1, ISSN 0720-8928, Februar 2007.

[Uh07] Uhlig,D. : Eine Entwicklungsumgebung (IDE) für die BS2000-Migration auf Eclipse-Basis. GI - Softwaretechnik-Trends, Band 27, Heft 1, ISSN 0720-8928, Februar 2007.