Entwicklung und Erprobung eines skriptgesteuerten Nachrichtenverteilers

Diplomarbeit im Studiengang Informatik
an der
Universität Bremen

von Cédrik Duval
Matrikelnummer: 962344

Erstgutachter: Prof. Dr. F. W. Bruns
Zweitgutachter: Prof. Dr. Karl-Heinz Rödiger

Bremen, März 2002

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Das Ziel dieser Arbeit ist die Entwicklung und Erprobung eines Systems zum Austauschen von Nachrichten, dessen Steuerungsmöglichkeiten mit einer Scriptingsprache erweitert oder flexibel gestaltet werden kann. Die Idee zu dieser Arbeit basiert auf der allgemeinen Annahme, daß Softwarekomponenten durch die Erweiterung mit einer Scriptingsprache adaptiver werden. Dadurch hoffe ich, eine Alternative zu schon bestehenden Architekturen zu entwickeln.

1.1 Problembeschreibung und Motivation

In der Softwareentwicklung existieren zwei grundverschiedene Ansätze zum Entwickeln großer Programme. Der einfachere der beiden ist der, alles in ein einziges Programm zu integrieren, was zu monolithischen Strukturen führt. Der zweite Ansatz beruht auf der Idee, daß sich komplexe Programme in Teile zerlegen lassen, die als eigenständige Programme (i.F. Komponenten genannt) angesehen werden können. So könnte in einem Datenbanksystem die Eingabe, Verwaltung und Visualisierung von Datenbeständen jeweils voneinander getrennt werden. Diese Trennung ist nicht nur auf die Ausführung beschränkt, sondern zeigt sich auch schon während der Entwicklung, wo verschiedene Programmierer sich jeweils nur um eine Komponente kümmern und diese im Idealfall vollkommen unabhängig von anderen Komponenten fertig stellen.

Eigenständige Komponenten sind allerdings gezwungen, während ihrer Ausführung auf andere Art miteinander zu kommunizieren, als wenn sie Bestandteile eines monolithischen Systems sind. Dafür existieren ebenfalls zwei Möglichkeiten. Die erste ist der Einsatz von gemeinsamem Speicher, den Komponenten während ihrer Laufzeit nutzen. Dabei nähert sich allerdings ihr Verhalten während der Ausführung wieder den monolithischen Systemen an (vgl. [Tan94]), denn der besondere Vorteil, daß jede Komponente während ihrer Ausführung einen unabhängigen Kontext besitzt, geht dabei verloren. Durch den Einsatz von gemeinsamem Speicher besteht ein besonderer Bedarf an sorgfältig implementierten Synchronisationsmechanismen für die Lese- und Schreibzugriffe.

Die alternative Möglichkeit besteht darin, Komponenten über Nachrichten miteinander kommunizieren zu lassen. Eine Nachricht kann dazu benutzt werden, um bei einer anderen Komponente die Ausführung einer bestimmen Aktion zu veranlassen. Benötigt die empfangene Komponente externe Daten von der sendenden, so werden diese in der Nachricht mitgeliefert. Auf jeden Fall kann keine Komponente direkt auf den Datenbestand einer anderen zugreifen. Dieser Form der Kommunikation liegt ein Protokoll zugrunde, das genau festlegt, wie eine Nachricht aufgebaut werden muß. Das heißt, daß Nachrichten im Normalfall einer strikten Definition unterliegen. Dies wiederum bedeutet, daß eine Komponente nur eine bestimmte Menge von unterschiedlichen Nachrichten zur Verfügung hat, die sie an andere Komponenten verschicken kann, vor allem aber, daß sie nur auf bestimmte Nachrichten

reagieren kann. Eine solche Vorgehensweise ist ausreichend, wenn sich das umschließende System nicht erweitern lassen soll.

Es gibt allerdings Systeme, die darauf ausgelegt sind, daß sie erweitert werden. Darunter fallen speziell verteilte Systeme mit noch offenen Diensten, die ein Server verschiedenen Clients anbietet. Derartig verteilte Systeme finden sich zunehmend im Bereich der Modellierungs- und Simulationssoftware. Ein solches System mit Namen CLEAR[1] wurde im EU-Drittmittelprojekt BREVIE[2] im Forschungsinstitut Arbeit, Umwelt & Technik (ARTEC) der Universität Bremen entwickelt.

Hauptforschungsgebiete bei ARTEC sind das Entwickeln und Untersuchen von Multimediasystemen im Bereich der Produktions- und Automationstechnologie. Eine wichtige Frage in ARTEC ist, inwieweit sich bestehende Technologien mit Mensch-Maschine-Schnittstellen erweitern lassen. Das Ziel ist, eine möglichst enge Kopplung der virtuellen mit der realen Welt zu erreichen, um die Vorteile beider Welten kombiniert nutzen zu können.

BREVIE hatte das Ziel, für Berufsschüler der Mechatronik eine neuartige Lernumgebung zu entwickeln, welche verschiedene Arten des Lernens unterstützt. Den Lehrern und ihren Schülern sollte so ein Maximum an Freiheit gegeben werden, ihre bevorzugte Art zu lehren und zu lernen selbst festzulegen, um die Effizienz zu steigern (vgl. [Bra00]).

CLEAR ist das Ergebnis dieses Projekts. Es kombiniert einen Modelliertisch zum Aufbau realer pneumatischer Schaltungen mit virtuellen Komponenten (Texte, Bilder, Videos, 3D-Darstellung, Schaltungssimulator). Über eine Objekterkennung ist es möglich, die reale Schaltung virtuell abzubilden. Dazu besitzt CLEAR eine Bibliothek pneumatischer Objekte. Mit der Abbildung der Schaltung im Virtuellen kann der Berufsschüler nun Simulationen durchführen, sie modifizieren oder zum späteren Weiterarbeiten abspeichern. Im didaktischen Teil des CLEAR-Systems, steht ihm ein Hilfesystem zur Verfügung, das ihm eine multimediale Beschreibungen zu jedem Objekt liefert.

Obwohl CLEAR auf pneumatische Komponenten beschränkt ist, ist das System prinzipiell auch für andere Einsatzgebiete konzipiert. In dem BREVIE-Nachfolgeprojekt DERIVE (Distributed Real and Virtual Learning Environment for Mechatronics and Teleservice) wurde die Objektbibliothek um elektro-pneumatische Objekte erweitert. Weitere Anwendungen sind denkbar, wobei der Kontext beibehalten (Einführung weiterer elektro-pneumatischer Objekte, die bislang nicht modelliert sind), erweitert (Einführung von hydraulischen Objekten) oder gänzlich verlassen werden kann (Modellierung eines Straßennetzes).

Betrachtet man die zu Beginn dieses Abschnitts angesprochene Problematik, stellt man fest, daß CLEAR leider das Defizit aufweist, sich nicht ohne Schwierigkeiten um neue Clientfunktionen erweitern zu lassen. So ist CLEAR zwar in der Lage, von *Clients* erzeugte

[1] The Constructive Learning Environment

[2] BREVIE (Bridging Reality and Virtuality with a Graspable User-Interface) lief von 1998 bis 2000 in ARTEC und wurde von der EU im Rahmen des IST-Forschungsprogramms finanziert. Dazu kamen noch weitere Partner aus Wirtschaft und Lehre.

Nachrichten zu akzeptieren, aber schon höhere Funktionen, wie die automatische Benachrichtigung, können von neuen *Clients* nur im Rahmen der Vorgaben von CLEAR genutzt werden. Die zur Zeit einzige Lösung besteht darin, die verantwortlichen Komponenten auf der Implementierungsebene zu erweitern. Es stellt sich die Frage, ob es dafür keine Alternative gibt, mit der man diese Einschränkung beseitigen kann.

1.2 Thematische Eingrenzung und Zielsetzung

In meiner Diplomarbeit möchte ich nun versuchen, eine solche Alternative für dieses Problem zu finden. Das Einbinden einer Scriptingsprache ist meiner Meinung nach der geeignetste Ansatz, da Scriptingsprachen zwei wichtige Vorteile bieten. Da es sich um höhere Sprachen handelt, wird ein System, das mit einer Scriptingsprache ausgestattet ist, leichter zu handhaben. Man kann es nun seinen Bedürfnissen entsprechend anpassen. Dieses eröffnet mehr Möglichkeiten als es z.B. eine Lösung mittels Dialogen vermag. Dialoge erlauben nur das Verändern vorgegebener Parameter; Scriptingsprachen ermöglichen dagegen auch das Angeben und Einsetzen neuer Parameter. Das ist eine wichtige Voraussetzung für jeden neuen *Client*, der in ein existierendes System eingebettet werden soll. Desweiteren läßt sich der Sprachumfang gering halten und auf das Wesentliche beschränken. In Kombination mit einer einfachen Syntax ist eine Scriptingsprache auch für ungeübte Benutzer schnell erlern- und einsetzbar.

Im Zuge dieser Arbeit werde ich eine einfache Scriptingsprache entwickeln und einen Prototypen implementieren, auf dem die Sprache läuft. Der Prototyp wird zu Testzwecken in CLEAR integriert. Die dabei gewonnenen Erkenntnisse sollen als Basis für ein allgemeines Konzept dienen. Konkret werde ich die Merkmale von HLA (High Level Architecture) und Jini betrachten und versuchen Ansatzpunkte herauszufinden, die sich mit Scripting erweitern lassen.

1.3 Aufbau der Arbeit

Die Kapitel 2 bis einschließlich 6 sollen als Unterbau für meine Arbeit dienen. Ich werde Konzepte und Techniken betrachten, allerdings noch keine Bewertung vornehmen oder Festlegungen treffen, was ich bevorzuge bzw. ablehne.

In Kapitel 2 werde ich zunächst auf Scriptingsprachen allgemein und deren Einfluß, den sie heutzutage in der Softwareentwicklung haben, eingehen. Ich werde ihre typischen Merkmale herausarbeiten und schließlich aufzeigen, wo ihre Stärken liegen. Dazu werde ich auch versuchen, Scriptingsprachen in verschiedene Untergruppen aufzuteilen.

Das darauffolgende Kapitel befaßt sich mit der Entwicklung von Scriptingsprachen. Grundlegende Unterschiede in ihrer Struktur gegenüber Programmiersprachen verlangen, daß sie anders behandelt werden als diese. Hauptgegenstand sind Übersetzerbau und die dort verwendeten Grammatiken, mit denen Scriptingsprachen erzeugt werden.

In Kapitel 4 befasse ich mich mit dem Einsatzgebiet der Scriptingsprachen innerhalb dieser Arbeit. Ich beleuchte die Kommunikation zwischen Prozessen und die Techniken, die dabei zum Einsatz kommen, wobei ich den Schwerpunkt auf die Kommunikation mittels Nachrichten lege.

Kapitel 5 stellt die beiden Simulationsarchitekturen HLA und Jini vor. Mit dem Wissen aus dem vorherigen Kapitel werde ich mich auf die Bereiche Nachrichtenverwaltung und -versand konzentrieren. Die Vorstellung werde ich bewußt kurz halten. Sie soll nur mit den Konzepten vertraut machen, die hinter den beiden Architekturen stehen, da ich mich in Kapitel 9 darauf beziehen werde.

In Kapitel 6 möchte ich die CLEAR-Umgebung vorstellen. Neben den Kernkomponenten liegt mein Hauptaugenmerk auf jenen Komponenten, mit denen ich durch das Einbinden des Prototypen in Berührung kommen werde.

Das nächste Kapitel beschreibt die verschiedenen Stadien in der Entwicklung des Prototypen. Vorgestellt wird der fertige Prototyp schließlich in Kapitel 8.

Ausgehend von den Erkenntnissen der beiden vorangegangenen Kapitel folgt zum Abschluß eine Betrachtung der Konzepte von HLA und Jini, sowie eine exemplarische Ablaufbeschreibung, wie sich die beiden Architekturen mittels Scripting verbessern lassen könnten.

2 Einführung in die Scriptingsprachen

Dieses Kapitel soll einen Überblick über die Entwicklung von Scriptingsprachen geben. Die breiten Einsatzgebiete von heute, in denen sie zu finden sind, gab es vor zwei Jahrzehnten noch nicht. Heute ist es mitunter irreführend, einfach nur von Scripten zu sprechen. Es gibt vielfältige Unterschiede zwischen einzelnen Gattungen von Scriptingsprachen. In den folgenden Abschnitten werde ich versuchen, diese Gattungen zu klassifizieren. Desweiteren möchte ich die Vergleiche anderer Autoren zwischen einzelnen Scriptingsprachen betrachten und bewerten. Diese Erkenntnisse werden mir später, bei der Entwicklung einer eigenen Sprache, hilfreich sein.

2.1 Die Entwicklung des Scripting

Die ersten Scriptingsprachen überhaupt waren Jobkontrollsprachen, die dazu eingesetzt wurden, um besagte Jobs zu sequenzieren. Eine der ersten dieser Jobkontrollsprachen war JCL (Job Control Language), die im Betriebsystem OS/360 von IBM eingesetzt wurde. Die Jobs selber, die JCL verwaltete, waren in PL/1, Fortran und Assembler geschrieben.

Den nächsten Schritt in ihrer Entwicklung machten die Scriptingsprachen, als sie in Verbindung mit dem Konzept der Pipes auf UNIX-Maschinen eingesetzt wurden. Aus einem Kommandointerpreter wie `sh` und `csh` heraus konnten sie eingesetzt werden, um Programme miteinander interagieren zu lassen. Diese sogenannten Stapelverarbeitungssprachen gelten heute als der Inbegriff der Scriptingsprachen. Allerdings war auch deren Funktionsumfang begrenzt. In den Folgejahren kamen vermehrt spezialisierte Scriptingsprachen auf, wie etwa SED und AWK zum Analysieren und Verändern von Texten.

1986 kam die erste Version von Perl (Practical Extraction and Report Language) heraus. Perl ermöglichte, daß unterschiedlichste Anforderungen (Serverprogrammierung, Systemmanagement, Rapid Prototyping, Textmanipulationen, Datenbankzugriffe) ab sofort mit derselben Sprache gelöst werden konnten, ohne den Aufwand einer Hochsprache in Kauf nehmen zu müssen.

Als Mitte der 90er Jahre die breite Bevölkerung begann, sich für das Internet zu interessieren, drangen schließlich die Scriptingsprachen für das World Wide Web auf den Markt. Sie ermöglichen es, statische Webseiten dynamisch zu machen. JavaScript ist der wichtigste Vertreter dieser Sprachen (vergleiche Abschnitt 2.4.6).

Bis heute hat sich ihr Funktionsumfang beträchtlich erweitert. Damit einher ist aber auch eine Abkehr von traditionellen Eigenschaften zu verzeichnen, wie etwa, daß Scriptingsprachen ausschließlich aus Kommandointerpretern heraus aufgerufen werden.

Inzwischen ist es auch zu einer interessanten Verschmelzung zwischen Programmier- und Scriptingsprachen gekommen. In einigen Hochsprachen wurden Funktionalitäten integriert, die sie den Scriptingsprachen näher bringen. So gibt es beispielsweise in Visual Basic keine

strenge Typisierung, damit alle Variablen gleich behandelt werden können, unabhängig davon, welchen Typ sie repräsentieren und welchen Wert sie speichern.

2.2 Definition

> *„Scripting bezeichnet einen Ansatz der Softwareentwicklung, bei dem Anwendungen aus speziellen vorgefertigten Komponenten erarbeitet werden. Es ist demzufolge ein komponentenorientiertes Vorgehen. [...] Eine Scripting-Sprache [...] liefert eine kompakte Notation zur Erstellung von Skripts. Das Scripting-Modell bestimmt, welche Typen von vorgefertigten Komponenten auf welche Art zusammengefügt werden können. [...] Das Scripting-Modell besteht aus Syntax und Semantik für die Verbindung von Komponenten in einer bestimmten (Anwendungs-)Umgebung.“ (aus [Bau00])*

Wenn man Scripte als eine Folge von Befehlen beschreibt, würde diese Beschreibung auch für eine echte Programmiersprache wie Pascal oder C++ zutreffen. In der Regel ist die Struktur der Scripte allerdings simpler. Der wichtigste Unterschied ist allerdings, daß mit Scriptingsprachen normalerweise keine Programme erzeugt werden, sondern Abläufe von Programmen gesteuert werden (vgl. Abschnitt 2.3). Dieses erreichen sie mit einer Zugriffsschnittstelle sowie einem Interpreter, der die Befehle, die über die Schnittstelle kommen, interpretiert. Allgemein kann man sagen, daß Scriptingsprachen dazu benutzt werden, die Möglichkeiten bestehender Komponenten zu erweitern. Ousterhout nennt sie in diesem Zusammenhang in [Ous98] auch Bindesprachen (*glue languages*) und Systemintegrationssprachen (*system integration languages*).

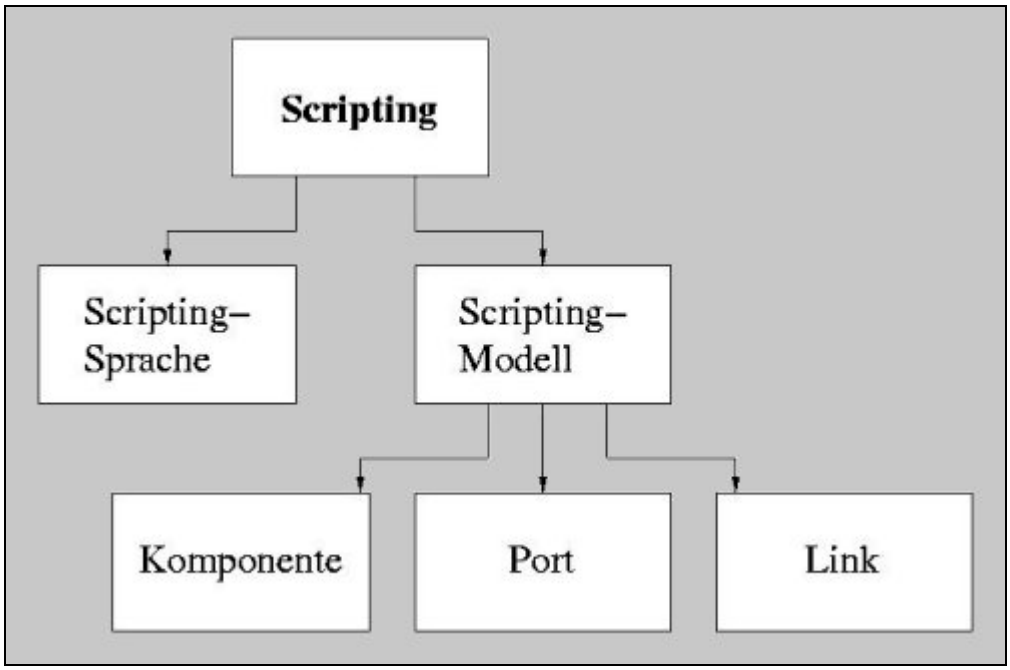

Abbildung 2.1: Scripting

Abbildung 2.1 zeigt nun das hierarchische Modell des Scripting wie es in [Kap89] vorgegeben ist und im Zitat zu Beginn dieses Abschnitts beschrieben wurde. Das Scriptingmodell ist dabei weiter aufgeteilt:

- Komponenten: Hierbei handelt es sich um die Objekte bzw. Bausteine, die durch die Scriptingsprachen miteinander verbunden werden. Im Scriptingmodell ist festgelegt, welche Komponenten von der Scriptingsprache benutzt werden dürfen. Sie können zum Beispiel Applikationen sein.
- Port/Link: Die Schnittstelle der Komponenten und die Art der Übertragung. Das Scriptingmodell legt fest, auf welche Art Komponenten miteinander unter Zuhilfenahme eines Scriptes kommunizieren.

Das Scriptingmodell wird weiter unterschieden, basierend auf den Ein-/Ausgaben, die an den Schnittstellen erwartet werden. Das Datenflußscriptingmodell (*dataflow scripting model*) ist das am häufigsten auftretende. Hier werden an den Ports Daten ein- und ausgegeben. Daneben ist noch das objektorientierte Scriptingmodell (*object oriented scripting model*) erwähnenswert, in dem jede Komponente Dienstleistung über seine Schnittstelle bereitstellt. Eine entsprechende Scriptingsprache ermöglicht es, daß Komponenten auf die Dienste anderer Komponenten zugreifen.

Richter [Ric00] listet weitere Punkte zur Definition von Scriptingsprachen auf, die betrachtet und diskutiert werden sollen. Anstatt um den Aufbau der Scriptingsprachen geht es aber eher um Eigenschaften, die eine Scriptingsprache als solche auszeichnen sollen:

- Programmierung auf hohem Abstraktionsniveau: Scriptingsprachen werden in erster Linie dafür verwendet, existierende Komponenten zu verbinden, anstatt sie selbst zu implementieren.

Dieser Punkt wurde weiter oben schon indirekt behandelt. Die Vorteile liegen klar auf der Hand. Ähnlich dem Prinzip, ein großes Problem in mehrere Teilprobleme zu zerlegen, werden viele kleine Applikationen entwickelt, die am Ende nur noch zusammengefügt werden.

- Erweiterbarkeit und Anpassbarkeit: Applikationen können ausgezeichneten Komponenten innerhalb ihrer Architektur eine Schnittstelle nach außen geben, so daß es möglich wird, vorhandene Funktionalitäten mittels Scripting zu erweitern oder gar neue einzubauen.

Dies ist für mich ein sehr wichtiger Punkt, da diese Diplomarbeit unter anderem auf genau dieser Idee fußt. Genauso, wie mittels Scripting mehrere Applikationen verbunden werden können, ist es auch möglich, verschiedene Komponenten innerhalb einer Applikation nach Belieben durch den Einsatz einer Scriptingsprache zu verknüpfen. Diese Idee ist nicht neu und wird schon eingesetzt (siehe Abschnitt 2.4.7).

- Schnelles Entwickeln von Applikationen und Prototypen: Dadurch, daß Scripte (im Allgemeinen) interpretiert werden, ermöglichen die dazugehörigen Scriptingsprachen während der Implementierung eine höhere Interaktionsmöglichkeit mit dem System.

Damit ist nichts anderes gemeint, als daß man sich zum Einen den Kompilationsvorgang spart und zum anderen Änderungen am Scriptcode während der Ausführung vornehmen kann. Um alle diese Vorteile genießen zu können, sind allerdings zwei Voraussetzungen notwendig:

1. Entwickler müssen eine gewisse Anzahl von fertigen Komponenten einsatzbereit haben, damit sie in bzw. von den Scripten benutzt werden können.
2. Die Umgebung der Scriptingsprachen muß sich nahtlos in die Applikationen bzw. Komponenten integrieren. Ein Umschreiben der Applikationen kommt ebensowenig in Frage, wie ein Verändern der Architektur einer einzelnen Applikation, die mit Scripting-Fähigkeiten erweitert werden soll.

2.3 Unterschiede zwischen Scripting- und Programmiersprachen

Gab es früher nur Scripte, die den Batch-Betrieb automatisierten, so ist das Spektrum heute breiter geworden. Scripte gibt es heute für eine Vielzahl von Anwendungen, einige sind hoch spezialisiert und nur für eine oder sehr wenige Aufgaben zu gebrauchen, andere haben inzwischen Ausmaße und Funktionalitäten von Programmiersprachen erreicht (z.B. Perl).

Dennoch besitzen sie fast alle gewisse Gemeinsamkeiten, die sie als Mitglied der Scriptingsprachen auszeichnen und von den Programmiersprachen trennen. Da das Feld der Scriptingsprachen aber inzwischen breit gefächert ist, gibt es auch Sprachen, die eine oder mehrere der folgenden Kriterien nicht erfüllen aber trotzdem dazugehören.

2.3.1 Interpretersprachen

Scriptingsprachen sind normalerweise Interpretersprachen (vgl. [Ous98]). Das heißt, sie werden nicht kompiliert. Es wird ein externes Programm benötigt, um sie auszuführen. Zuweilen sind diese Programme schon als Module in das benutzte (Betriebs-)System integriert, so daß man deren Scripte ausführen kann wie echte Applikationen.

Einige neuere Versionen von Scriptingsprachen – speziell die sogenannten schnittstellenerzeugenden Sprachen – besitzen einen Compiler, der den Quelltext in einen Bytecode wandelt, der sich schneller interpretieren läßt und dennoch plattformunabhängig (vgl. Abschnitt 2.3.2) bleibt. Als Beispiel seien hier Java und die neueste Version von Tcl erwähnt.

2.3.2 Plattformunabhängigkeit

Durch die einfache Struktur eines Scripts und den Umstand, daß es normalerweise nicht kompiliert wird (vgl. Abschnitt 2.3.1), ist es prädestiniert, ohne syntaktische Veränderung auf verschiedenen Plattformen eingesetzt zu werden. Es bleibt natürlich der Interpreter, der auf die unterschiedlichen Plattformen portiert werden muß.

2.3.3 Fehlerverhalten

Es kann nicht garantiert werden, daß ein Script, obwohl syntaktisch und semantisch korrekt, immer wunschgemäß ausgeführt wird. Durch den Umstand, daß mittels einer Scripting-

sprache Daten zwischen Komponenten ausgetauscht werden, darf man auch die Korrektheit dieser beiden Elemente nicht aus den Augen verlieren.

Abbildung 2.2: Ein MS-DOS Script

Abbildung 2.2 zeigt ein kleines MS-DOS-Script, das den Inhalt eines Verzeichnisses mit Namen „temp“ im Laufwerk C: vollständig ohne Rückfrage löscht (die Option „–n“). Das Script wird dem Anschein nach immer korrekt arbeiten. Tatsächlich tut es das jedoch nur, solange auch ein Verzeichnis mit dem angegebenen Namen existiert. Sollte der Fall eintreten, daß das Verzeichnis nicht vorhanden ist, wird der Befehl *cd* nach dem Abarbeiten der zweiten Zeile eine Fehlermeldung erzeugen, um zu signalisieren, daß ein Verzeichniswechsel nicht statt fand. Trotz dieses Fehlers wird die Stapelverarbeitung nun mit dem Abarbeiten der dritten Zeile fortfahren. Der dortige Löschbefehl kann nun u.U. fatale Folgen haben, da sich das Arbeitsverzeichnis nicht auf das Verzeichnis „temp“ geändert hat, sondern sich irgendwo auf Laufwerk C: befindet (der erste Befehl des Scriptes wechselt zwar auf das Laufwerk C:, aber nicht in das dortige Wurzelverzeichnis). Im schlimmsten Falle könnte der Inhalt des gesamten Laufwerks vernichtet werden.

Um die korrekte Abarbeitung von Scripten garantieren zu können, muß zusätzlich die Korrektheit aller verwendeten Applikationen gelten.

2.3.4 Typisierung

Scriptingsprachen benutzen selten unterschiedliche Typen. Der Vorteil ist, daß sich Zahlen, Worte und andere Typen gleich behandeln lassen. Der Interpreter unternimmt intern die notwendigen Umwandlungen, damit Variablen den für eine Operation korrekten Typ haben (Zahl für mathematische Operationen, String für Ausgaben, boolescher Typ für Vergleiche, etc.).

Typisierung zählt damit zu den Merkmalen der High-Level-Programmierung, die im folgenden Abschnitt auf einer allgemeineren Ebene betrachtet wird.

2.3.5 High-Level-Programmierung

Ein einzelner Ausdruck innerhalb einer Scriptingsprache kann bis zu tausend Zeilen (manchmal sogar mehr) Maschinencode entsprechen, wo hingegen Ausdrücke in einer Hochsprache wie Pascal, C++ oder Java im Durchschnitt nur 5 Zeilen Maschinen-instruktionen entsprechen. Dieser krasse Gegensatz läßt sich damit erklären, daß Ausdrücke in

Scriptingsprachen mehr Arbeit leisten. Die nachfolgende Zeile, die aus einem TCL-Script stammt (entnommen aus [Ous98]), soll dieses an einem praktischen Beispiel verdeutlichen:

```
button .b –text Hello! –font {Times 16} –command {puts
hello}
```

Diese Zeile erzeugt einen Button auf dem Bildschirm, der mit dem Ausdruck „Hello!“ im Zeichensatz Times und einer Größe von 16 Punkten beschriftet ist. Beim Drücken des Knopfes wird das Wort „hello“ ausgegeben.

Um einen äquivalenten Knopf mit der selben Funktion in einer Hochsprache zu erzeugen, ist einiges an Mehraufwand nötig. Objekte für den Button und die Schrift müssen erzeugt werden, Parameter wie Position und Größe müssen ermittelt werden, etc. In VisualC++ und den Microsoft Foundation Classes (MFC) wären mindestens 25 Zeilen Code nötig, verteilt auf drei Prozeduren, um einen äquivalenten Button zu erzeugen.

Der Tcl-Interpreter hält für fast alle Parameter Initialwerte bereit. Solange also vom Programmierer des Scripts keine eigenen Angaben kommen, nimmt der Interpreter einfach diese Werte.

Die folgende Tabelle ist ein Auszug aus [Jon96b]. Sie zeigt diverse Sprachen im Vergleich bezüglich ihres Sprachlevel (*language level* / vgl. ebenda). Das Sprachlevel errechnet sich aus der durchschnittlichen Anzahl Codezeilen, die nötig sind, um einen Funktionsknoten (*function points* / [Jon96a]) zu implementieren. Funktionsknoten sind eine Metrik, mit der sich die Produktivität in der Softwareentwicklung messen läßt. Ein Funktionsknoten entspricht dabei einer nicht näher spezifizierten Funktion innerhalb eines Softwarepakets. In der folgenden Tabelle befindet sich die durchschnittliche Anzahl an Ausdrücken, die je nach Sprache nötig ist, um einen Funktionsknoten zu implementieren.

Sprache	**Sprachlevel**	**Ø Anzahl Ausdrücke pro Funktionsknoten**
Assembler	1	320
DOS-Batchscripte	2.5	128
C	3.5	91
Tcl (ohne TK)	5	64
Java	6	53
C++	6	53
Visual Basic 5	11	29
Perl	15	21
UNIX-Shellscripte	15	21

Anhand der Tabelle läßt sich vergleichen, mit wieviel Aufwand man ein vergleichbares Problem mit der jeweiligen Sprache lösen kann. Das Problem muß sich natürlich mit der gewählten Sprache überhaupt lösen lassen.

Man erkennt leicht, wie nah die Hochsprachen an die reinen Scriptingsprachen gerückt sind und sie teilweise sogar schon überholt haben. Dieses läßt sich mit der beginnenden

Verschmelzung der beiden Sprachfamilien erklären, was schon in Abschnitt 2.1 kurz angedeutet wurde.

2.3.6 Vergleich von Aufwand und Leistung

Bei vielen Aufgaben haben Programmierer die Möglichkeit, aus einem großen Fundus an Sprachen die zu wählen, die sie zur Lösung der Aufgabe brauchen. Dieser Abschnitt befaßt sich mit dem Vergleich von verschiedenen Sprachen bei unterschiedlichen Aufgabenstellungen. Der Vergleich beginnt bei der Wahl der Sprache und dem damit verbundenen Aufwand, die Aufgabe zu implementieren.

2.3.6.1 Aufwandsbestimmung

Bei der Frage nach der Wahl der passenden Sprache für ein Problem ist leicht zu ersehen, daß Programmiersprachen am universellsten einsetzbar sind und deshalb für jede Aufgabe in Frage kommen können. Auf der anderen Seite existieren spezialisierte Sprachen, deren Kommandoumfang an gewisse Aufgaben angepaßt ist. John K. Ousterhout stellt in [Ous98] mehrere Vergleiche auf. Gegeben sind komplexere Aufgabenpakete (Datenbankapplikation und –bibliothek, Tabellenkalkulation, Simulator mit GUI, etc.). Ousterhout unterscheidet zwischen der Größe des Quelltextes und der Zeit, die benötigt wurde, die Aufgabe zu lösen. Zwar beschränken sich die meisten Vergleiche auf C/C++ und Tcl/Tk, doch kann man die jeweilige Sprache als Repräsentanten ihrer Familie ansehen.

Bei dem Vergleich wurden natürlich nur solche Aufgaben gewählt, die mit Scriptingsprachen lösbar sind. Entsprechend ist es auch am effizientesten, eine Scriptingsprache einzusetzen. Bedingt durch die mitunter großen Unterschiede in Zeitaufwand und Codegröße, ist die Entscheidung deshalb für komplexe Aufgaben einfach.

Doch auch bei kleineren Aufgaben, die sich mit den meisten Sprachen in einer einzigen Funktion oder gar Zeile implementieren lassen, führen fast immer die Scriptingsprachen. Auf der anderen Seite sind die Ergebnisse fast immer langsamer als ihre Pendants aus den Programmiersprachen. Der nächste Abschnitt befaßt sich damit genauer.

2.3.6.2 Geschwindigkeit

Nicht selten kommt es bei Programmen auf Geschwindigkeit an. Echtzeit wird zwar nicht immer gefordert, trotzdem sollte ein Ergebnis in möglichst kurzer Zeit verfügbar sein. In [Ker98] wird die Performance verschiedener Scriptingsprachen unter die Lupe genommen und mit C als Referenz verglichen. Getestet wurde eine Vielzahl von Mechanismen wie Schleifen, Feldzugriffe, verschachtelte Funktionsaufrufe und *I/O-Streaming*. Dabei wurde immer mit extrem großen Werten gearbeitet, um die Unterschiede deutlich zu machen.

Die jeweilige C-Version ist fast immer am schnellsten; viel wichtiger sind aber die unterschiedlichen Ergebnisse innerhalb der Scriptingsprachen, läßt sich doch so genau erkennen, wo ihre jeweiligen Stärken und Schwächen liegen.

Einen Ansatz gänzlich anderer Art verfolgt Lutz Prechelt in [Pre00]. Anstatt in jeder Sprache eine einzige Lösung für ein vorgegebenes Problem zu implementieren, sammelte er die

Ergebnisse von mehreren unabhängigen Programmierern. Abbildung 2.3 zeigt ein Ergebnis dieser Untersuchung. Die einzelnen Programme sind als kleine Punkte eingetragen (man beachte die logarithmische Zeitachse).

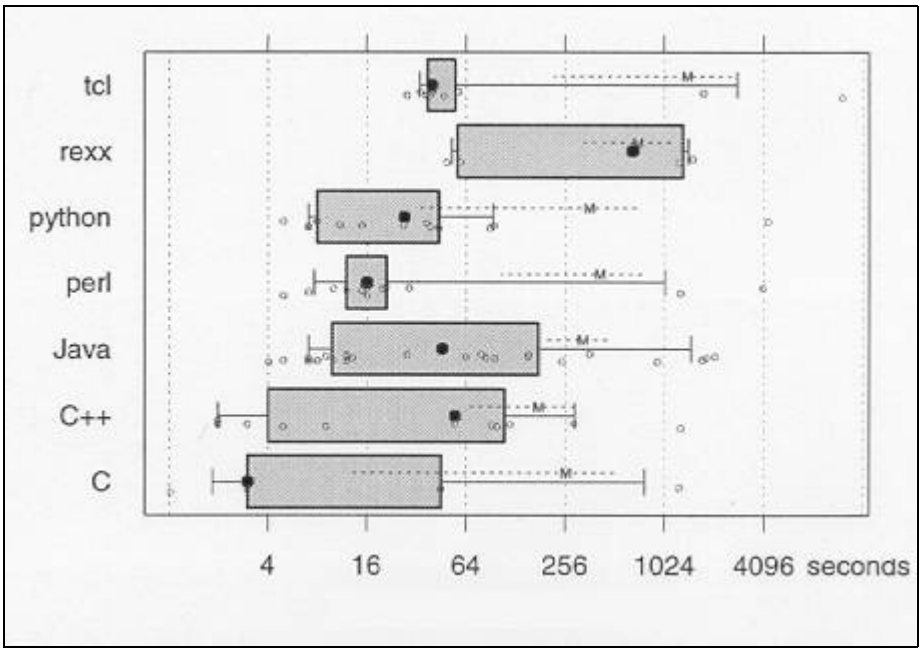

Abbildung 2.3: Vergleich der Laufzeiten

Zwar erkennt man, daß die C- bzw. C++-Programme am schnellsten sind. Andererseits ist zu sehen, wie stark die Qualität der Implementierung das Ergebnis beeinflußt. Neben der Wahl der Sprache ebenfalls ein wichtiges Kriterium für die Performanz.

2.4 Anwendungsgebiete von Scriptingsprachen

Wie im vorigen Abschnitt bereits erwähnt, haben sich im Laufe der Zeit verschiedene Subtypen der Scriptingsprachen entwickelt, die auf verschiedene Anwendungen spezialisiert sind. Sie ähneln damit den frühen Hochsprachen, die ebenfalls für bestimmte Anwendungsgebiete entwickelt wurden. In den folgenden Unterabschnitten versuche ich nun diese Subtypen zu kategorisieren.

2.4.1 Erweiterung von Kommandointerpretern

Dieses Gebiet wurde in den vorhergehenden Abschnitten schon mehrfach erwähnt. Es handelt sich um Scriptingsprachen, die als fester Bestandteil des Betriebssystems Zugriffe darauf ermöglichen. Die Möglichkeiten können extreme Formen annehmen, wie z.B. in Linux, wo sämtliche Funktionen sich komplett über Befehlsscripte steuern lassen.

Ebenfalls bezeichnend ist, daß man Scripte schachteln kann, also sich innerhalb eines Scripts weitere aufrufen lassen. Je nach verwendetem Kommandointerpreter ist die Ausführung der Scripte streng sequentiell (z.B. DOS) oder nebenläufig (z.B. UNIX-Varianten). In MS-DOS und ähnlichen Betriebssystemen muß ein Script, das ein weiteres startete, auf dessen

Termination warten, bevor es mit seiner Ausführung fortfahren kann. Unix und seine Varianten erlauben, daß ein Script nebenläufig zu dem Aufrufenden ausgeführt wird.

Es folgt ein abschließendes Beispiel, in dem drei Applikationen aus Unix miteinander über Pipes verknüpft sind, um die Vorteile dieses Konzeptes zu verdeutlichen:

```
more scripte.txt | grep scripting | wc
```

Das Programm *more* gibt den Inhalt einer angegebenen Datei aus (hier *scripte.txt*). Die Applikation *grep* durchsucht einen Text nach einem Schlüsselwort (*scripting*). Die Anzahl der Zeilen, in denen das Wort vorkommt, wird schließlich mit *wc* gezählt. Natürlich kann man diese Verkettung beliebig weiter führen. Das letzte Ergebnis einer Verknüpfung wird standardmäßig auf dem Bildschirm ausgegeben, könnte aber auch in einer Datei gespeichert oder von einem Statistikprogramm eingelesen werden.

Dieses ist die verbreitetste und bekannteste Form des Scriptings. Zudem auch ein Paradebeispiel für das Datenfluß Scripting Modell (vgl. Abschnitt 2.2).

2.4.2 Temporale Scriptingsprachen

Diese Form des Scripting wird hauptsächlich in der Animation eingesetzt. Komponenten werden mit Zeitangaben verknüpft, um ihre Aktionen zu steuern. Dabei lassen sich relative Zeitangaben (nach 10 Sekunden) oder absolute (um 15:30 Uhr) gleichermaßen verwenden.

Eine bekannte und verbreitete Form der temporalen Scriptingsprachen sind die Makrosprachen aus diversen Anwendungen. Sie erlauben vom Benutzer gemachte Aktionen aufzunehmen und auf Wunsch abzuspielen. Makros ermöglichen es, daß komplexe Handlungen, wie das Aufbauen und Füllen einer Tabelle oder das Neuformatieren eines Textes, vereinfacht werden, da man sie nur einmal vormachen muß. Darüber hinaus haben sie einen hohen Wiederverwendungswert, da sie sich auf ähnliche Objekte anwenden lassen (verschiedene Texte oder Tabellen, die dieselbe Formatierung erhalten).

Ein aufgezeichnetes Makro kann unter Umständen im ASCII-Format vorliegen, so daß es sich theoretisch auch von Hand bearbeiten läßt. Zum Beispiel werden Makros in MS-Office-Applikationen in Visual Basic-Programmcode abgespeichert. Die für gewöhnlich hohe Komplexität der Makros macht eine manuelle Bearbeitung allerdings selten ratsam.

2.4.3 Scripting in der Modellierung und Simulation

Temporales Scripting kann in der Modellierung und Simulation eingesetzt werden. Die Funktion, die es dort übernimmt, geht über das bloße Starten und Beenden von Komponenten hinaus. Das Script übernimmt die Rolle eines Drehbuchs oder Theaterscripts, in dem jede Komponente, wie ein Schauspieler zu verschiedenen Zeitpunkten verschiedene Aktionen ausführt (Bewegen, Sprechen, etc.). Man spricht in diesem Zusammenhang auch von Ereignistabellen (*event tables*). Diese Technik des scriptgesteuerten Simulierens, wird im Deklarativen Modellieren (*declarative modeling*) verwandt, in dem Modelle ähnlich einem

Storyboard beim Film, aus verschiedenen Zuständen zusammengesetzt werden (siehe in diesem Zusammenhang auch [Fis95]).

Ein einfaches Beispiel ist die Positionsangabe von Komponenten zu verschiedenen Zeitpunkten wie in Abbildung 2.4 zu sehen ist. Durch Interpolation des Modells, kann man weitere Positionen berechnen. Hiermit haben wir auch den Unterschied zu gewöhnlichen Temporalscripten geklärt. Während bei letztgenannten Komponenten nur aktiviert bzw. deaktiviert werden, können sie in einer Simulation nicht nur mehrere vorgegebene Zustände annehmen, sondern es lassen sich u.U. während der Simulation neue Zustände herleiten, die vorher nicht definiert waren. In Abbildung 2.4b sieht man das anhand der neuen Positionen, die das Objekt *obj* zum Zeitpunkt 5 und 18 einnimmt. Vorher wurden dem Objekt verschiedene Punkte im zweidimensionalen Raum zugewiesen (Abbildung 2.4a), die es jeweils zu einer bestimmten Zeit einnimmt.

```
# obj ist ein Objekt im zweidimensionalen Raum
SET OBJECT obj
# tl ist ein linearer Zeitstrahl
SET TIMELINE tl
# obj wird mit tl verknüpft
PUT obj tl
# Nun werden Positionen festgelegt
SET AT 0 obj POS 0 0
SET AT 10 obj POS 50 50
SET AT 20 obj POS 25 75
SET AT 30 obj POS 42 0
```

```
: SHOW AT 0 POS obj
0 0
: SHOW AT 5 POS obj
25 25
: SHOW AT 18 POS obj
30 70
: SHOW AT 33 POS obj
42 0
:
```

Abbildung 2.4a+b: Scriptgesteuerte Simulation

Eine solche Technik ist z.B. in der Scriptingsprache Isis (siehe [Aga97]) realisiert worden, mit der man Multimediaprojekte auf verschiedenen Abstraktionsebenen entwickeln kann. Im folgenden Unterabschnitt werde ich auf die Fähigkeiten und Vorteile dieser Technik erneut zu sprechen kommen.

2.4.4 Scripting auf verschiedenen Abstraktionsebenen

Multiebenenabstraktion (*multilevel abstraction*) existiert in Ansätzen schon beim Webscripting, wird dort aber mit verschiedenen Sprachen realisiert (HTML, JavaScript, Java). Das Konzept basiert auf der Idee, für jede Abstraktionsstufe eine Sprache zu nehmen, die den Anforderungen der entsprechenden Stufe genügt (siehe dazu auch Abschnitt 2.4.6).

Die Multiebenenabstraktion versucht diesen Weg über eine einzige Sprache zu gehen. Realisiert worden ist dieses Konzept in der Scriptingsprache Isis. Die Sprache selber besteht aus einfachen Befehlen, mit deren Hilfe komplexe Programmkonstrukte entwickelt werden können, die bei Bedarf in die eigentliche Sprache, ähnlich einer Bibliothek, eingebunden werden können. Die Vorteile liegen klar auf der Hand: Anstatt für jede Aufgabe die passende bzw. vorgesehene Scriptingsprache zu nehmen (die man natürlich beherrschen muß), braucht man nur eine einzige Sprache zu beherrschen, deren Abstraktionsgrad man nach Belieben festlegen kann und das nicht nur in Stufen, sondern fließend.

2.4.5 Schnittstellenerzeugende Sprachen

Schnittstellenerzeugende Sprachen (*interface-building languages*) sind eine Form von Scriptingsprachen, die neben dem einfachen Verbinden von Applikationen auch eine Verbindung zum Benutzer schaffen. Das bekannteste Sprachpaket ist Tk, das zusammen mit Tcl benutzt wird, um Dialoge für die Benutzerinteraktion mit dem System zu erzeugen.

2.4.6 Webscripting

HTML (*Hypertext Markup Language*) ist die Scriptingsprache für das Internet. Sie erlaubt das Erstellen von Hypertexten, die speziell für die Darstellung auf Grafikbildschirmen aufbereitet sind. Die Scripte selber laufen auf sogenannten Browsern, von denen der *Netscape Navigator* und der *Microsoft Internet Explorer* die bekanntesten sind. War HTML zuvor als reine Dokumentenbeschreibungssprache gedacht (ähnlich TeX), so bekam sie mit dem Einführen der Stilvorgaben (*Style Sheets*), Fähigkeiten, die dem eines Desktop Publishing Programms entsprechen. Jüngste Bestrebungen gehen dahin, HTML in XML einfließen zu lassen, um die Sprache mit Indizierungsmöglichkeiten zu erweitern.

Neben den zweidimensionalen HTML-Seiten gibt es eine wachsende Anzahl von dreidimensionalen Welten im Netz, die mit der Sprache *VRML (Virtual Reality Markup Language*) geschrieben wurden. Die Stärke dieser Sprache besteht darin, hierarchische Welten aus mehreren Objekten zu erzeugen. Die Sprache bedient sich dabei gängiger Konzepte der objektorientierten Programmierung (Instanziierung, Vererbung, etc.).

JavaScript und seine Clone/Nachahmer (Jscript, VBScript, etc.) bilden schließlich die dritte Säule der webbasierten Scriptingsprachen. Sie bilden eine Erweiterung der beiden vorherigen und können nicht alleine benutzt werden. Ihre Aktionen sind an die Seite bzw. Welt gebunden, von der sie aufgerufen werden. Ihre Grundfunktionalität besteht dabei auf erweiterter Interaktionsmöglichkeit des Internetsurfers und einer ausgeklügelteren Steuerung der aktuellen Seite. Allgemein ausgedrückt werden Seiten und Welten mit JavaScript dynamischer.

2.4.7 Software-interne Scriptingsprachen

Diese hochspezialisierten Sprachen steuern die Interna eines einzelnen Programms. Sie ähneln den Makrosprachen, nur daß sie nicht so komplex sind und entsprechend von Hand gescripted werden. Häufig ist es sogar so, daß sie die einzige Benutzungsschnittstelle zu dem entsprechenden Programm bieten, wie etwa in Matlab, einem Programm zur Lösung mathematischer Aufgaben. Matlab versteht mathematische Ausdrücke und Funktionen als Eingabe und berechnet diese. Dabei kann der Benutzer auf einen großen Umfang von mathematischen Konstrukten (Vektoren, Matrizen, etc.) und Operationen (Vektorprodukt, Transponierte, etc.) zurückgreifen. [MT97] ist ein Tutorial zu Matlab, das einen Einblick in die Fähigkeiten dieses Programms vermittelt.

Ein weiterer interessanter Ansatz wird von Richter in [Ric00] gezeigt: Die Javaplattform wurde um eine Scriptingsprache (namens Iava) erweitert, die sich in Applikationen

integrieren läßt. Anders als bei Matlab, wo nur ein festes Set an Komponenten (mathematische Operationen) zur Verfügung steht, auf welches der Benutzer zurückgreifen kann, besteht in Iava die Möglichkeit, in die Applikation weitere Komponenten zu integrieren, indem sie einfach gescripted werden. Zwar kann man keine neuen Klassen erzeugen, aber dafür Instanzen bestehender (aus den Java-Paketen) und Methoden. So ist es ein leichtes, Applikationen zu implementieren, die nach Bedarf mit weiterer Funktionalität versehen werden können.

Richter weist darauf hin, daß es am sinnvollsten sei, solchen Scriptingsprachen wie Iava eine ähnliche Syntax wie derjenigen Programmiersprache zu geben, auf die sie aufgesetzt werden. Das ist nicht unbedingt sofort nachvollziehbar. Scriptingsprachen, die zur Erweiterung einer Programmiersprache entwickelt wurden, werden in möglichst simpler Form implementiert, die häufig mit der Struktur der zugrundeliegenden Sprache nicht vereinbar ist. Richters Ansatz erleichtert nicht nur dem Entwickler die Arbeit, da er sich keine weitere Sprache aneignen muß, sondern verringerte auch nach eigener Angabe den Entwicklungsaufwand der Scriptingsprache.

2.4.8 Umgebungsspezifische Sprachen

Umgebungsspezifische Sprachen (*domain specific languages* / DSL), sind keine eigentliche Untergruppe von Scripting Sprachen, sondern existieren parallel zu ihnen. Es gibt allerdings Überschneidungspunkte zwischen ihnen, da einige Scriptingsprachen, wie die UNIX-Shell, als DSL angesehen werden, weswegen ich sie hier kurz aufführen möchte.

Eine DSL ist eine Sprache, deren Einsatzgebiet sich auf einen bestimmten Kontext oder Problembereich beschränkt. Syntax und Semantik der Sprache sind entsprechend optimiert, weswegen sie für andere Aufgaben untauglich ist. Dadurch, daß die spezialisierten Befehle häufig aussagekräftig sind, werden DSL auch Spezifikationssprachen genannt und erheben nebenbei den Anspruch, daß sie auch von ungeübteren Programmierern eingesetzt werden können. Dieses läßt auf ihr Einsatzgebiet schließen, da sie vornehmlich in größeren Softwarearchitekturen als Mechanismus zur Parametrisierung und als Schnittstellenmodell eingesetzt werden (vgl. [Por99]).

Häufig handelt es sich bei den Architekturen um Applikationsgeneratoren. In Abbildung 2.5 (aus [Deu99]) ist eine DSL zu sehen, die aus einer Produktbeschreibung einen Datenbankeintrag, eine Eingabemaske und zwei weitere Module für die Registrierung und das Management des neuen Produkts erzeugt. In [Anl00] wird Beispielhaft eine DSL entwickelt, die Objekte in einen Datenspeicher überführt und dabei gezwungen ist, die Typen der Klasse zu beachten und die Daten eventueller Oberklassen mit zu berücksichtigen.

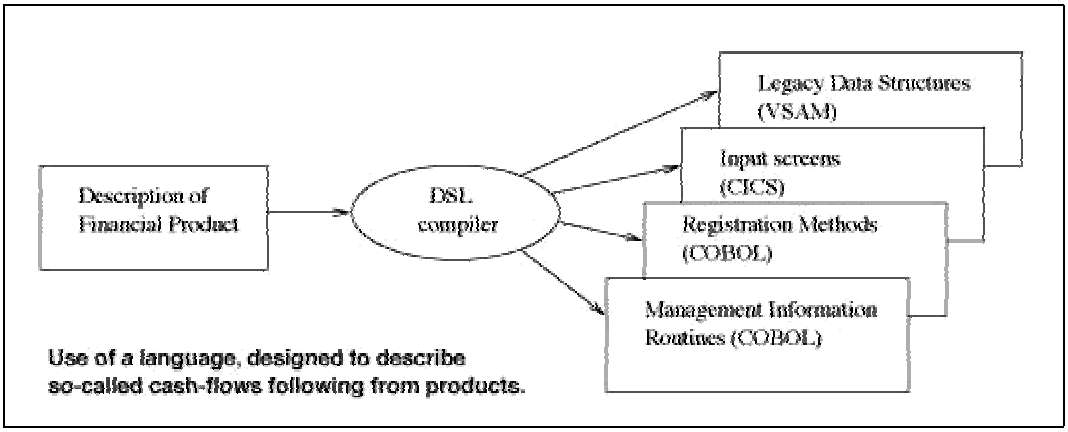

Abbildung 2.5: Ausgaben einer Domain Specific Language

DSL werden also im Zusammenhang mit nichtlinearen Datentransformationen ([Anl00]) eingesetzt, die ihre Funktionalität über die einer schnittstellenerzeugenden Sprache (siehe Abschnitt 2.4.5) erhebt.

2.5 Ausblick

Schon 1975 prognostizierte Frederick P. Broocks, jr[3], daß Batch-Programme nie völlig von interaktiven Systemen verdrängt werden würden. 1995 bestätigte er seine damalige Aussage in [Bro95] ein weiteres Mal.

Die allgemeine Prognose ist dahingehend, daß Scriptingsprachen auch weiterhin eine wichtige Rolle in der Entwicklung spielen werden. Tatsächlich wird ihr Einfluß sogar noch weiter steigen. Dafür sprechen mehrere Faktoren.

Die Scripting-Technologien werden kontinuierlich verbessert. Sprachen wie Perl, die laufend erweitert werden, haben einen gewaltigen Schatz an Ausdrücken und Möglichkeiten. Scriptingsprachen wie diese sind inzwischen so mächtig, daß sich viele Probleme mit ihnen einfacher lösen lassen, als mit herkömmlichen Hochsprachen.

Ein anderer Faktor ist, daß es mehr und mehr Hobbyprogrammierer gibt, die in ihrer Freizeit kleine Programme oder Scripte schreiben. Dieser Trend begann mit der Erschließung des Internet für den Heimbenutzer. Viele selbstgemachte Homepages im World Wide Web, die mit JavaScript oder Perl erweitert wurden, sind ein Beleg dafür.

Als letzter und wichtigster Grund sei (noch einmal) auf die Fähigkeit der Scriptingsprachen eingegangen, Applikationen miteinander zu kombinieren. Große Softwarepakete bestehen heute selten aus einer großen Applikation, sondern aus vielen kleinen. Neue Aufgaben werden durch neue Applikationen realisiert, die in die bestehenden Pakete eingebunden werden. Schnittstellenerzeugende Sprachen sind hierbei sehr hilfreich, da sie die für den Benutzer nötige grafische Schnittstelle realisieren.

---

[3] im „The Mythical Man-Month“

2.5.1 Visuelles Scripting

Visuelles Scripting ist eine Idee, die schon seit ein paar Jahren diskutiert wird und bei der es darum geht, die Möglichkeiten der visuellen Programmierung auf das Scripting zu übertragen. Hierbei werden die Scripte aus vorgefertigten Komponenten, wie bei einem Baukasten, zusammengesetzt. Die Komponenten selber können dann vom Programmierer mit relevanten Daten beschrieben werden, wie in Abbildung 2.6 zu sehen ist.

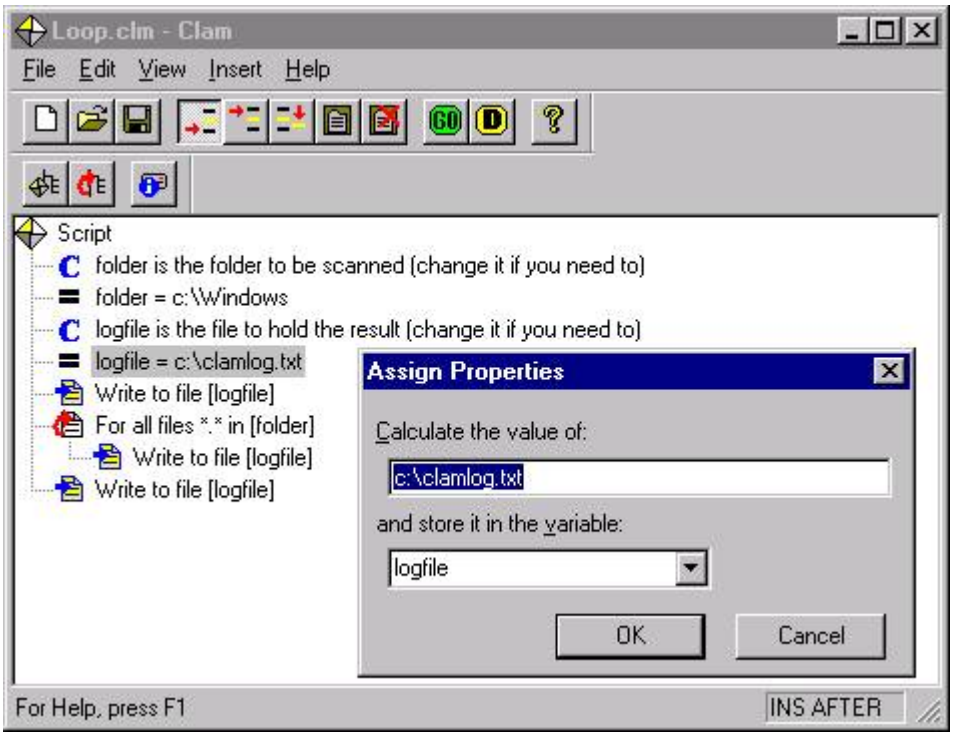

Abbildung 2.6: Ein visueller Scripteditor

Visuelles Scripting sollte nach Bauer [Bau00] folgende Möglichkeiten als Erweiterung zum konventionellen Scripting bieten:

- *grafische Repräsentation von Komponenten*
- *Darstellung von Ports an Komponenten (ungebundene Parameter)*
- *grafische Verbindungen zwischen Ports*
- *grafisches Editieren*
- *Kapselung von Scripts als wiederverwendbare Komponenten*
- *Möglichkeit der direkten Ausführung von Scripts*

Zur Zeit gibt es mehrere Applikationen, die verschiedene dieser Möglichkeiten realisiert haben, doch der Durchbruch des Visuellen Scriptings steht noch aus. Ein Blick über den Tellerrand hinaus mag jedoch sinnvoll und lohnenswert sein, da das Gebiet der Visuellen

Programmierung einige Teilgebiete besitzt, die sich auf das Visuelle Scripting übertragen lassen. Diese werden in Abbildung 2.7 (Quelle: [Pos96]) dargestellt.

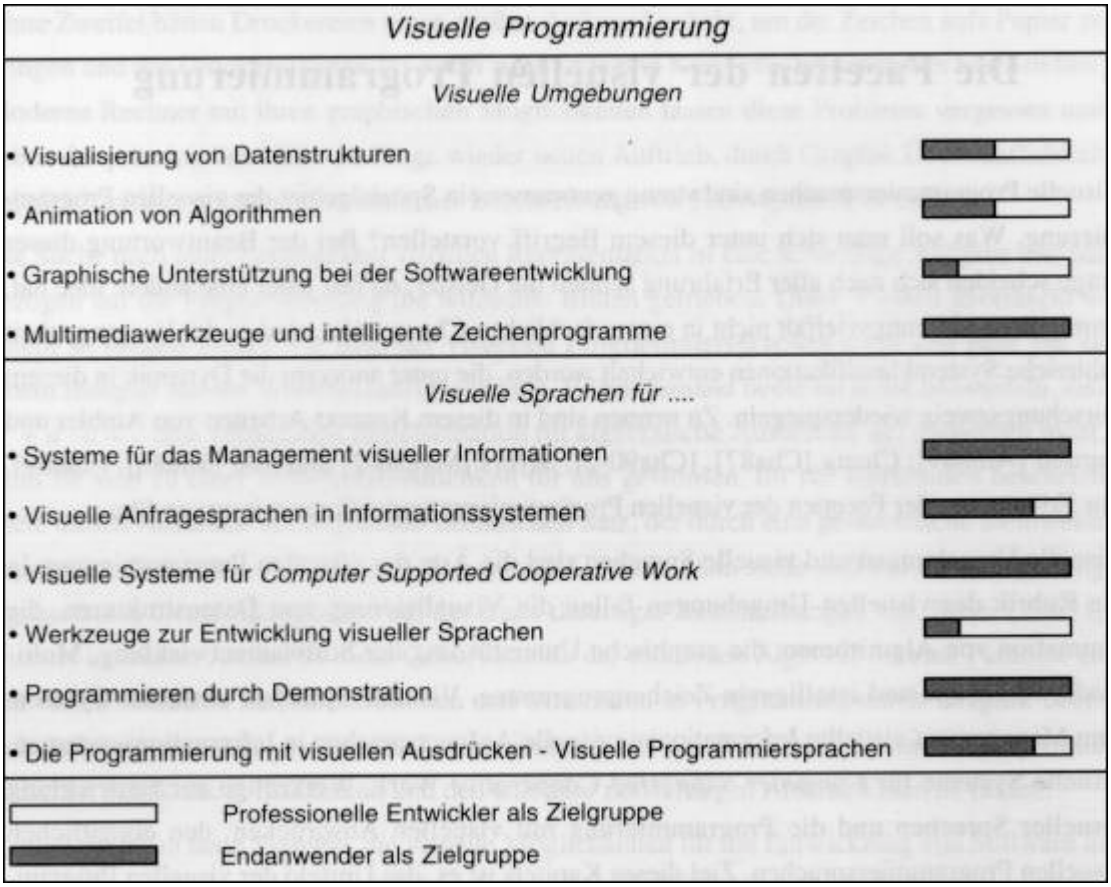

Abbildung 2.7: Visuelle Programmierung

Vergleicht man die einzelnen Punkte mit der Aufzählung, die weiter oben die Fähigkeiten einer Visuellen Scriptingsprache definieren soll, so fallen sofort Ähnlichkeiten auf. Es zeigt sich also, daß die visuellen Ableger der beiden Sprachfamilien sogar mehr Berührungspunkte haben, als die textorientierten, nichtvisuellen Vertreter.

2.5.2 Kombination von Programmier- und Scriptingsprachen

Es ist abzusehen, daß sich die beiden Sprachtypen weiter annähern werden. Es muß dabei nicht zwingend auf eine Verschmelzung hinauslaufen, wie sie im Ansatz schon in VisualBasic zu sehen ist. Viel eher läßt sich vermuten, daß moderne Programmiersprachen (etwa Visual C++) einen scriptbaren Aufsatz mit eingeschränktem Befehlssatz bekommen, um z.B. schnell kleine Prototypen zu entwickeln (*rapid Prototyping*).

2.5.3 Fortführende Literatur

Weitere Informationen über die Zukunft der Scriptingsprachen sind in [Ous98] und [Bau00] zu finden. Speziell zum Thema Visuelles Scripting ist vor allem [Pos96] erwähnenswert, sowie weitere Literatur, die sich mit Visueller Programmierung befaßt. Denn so, wie viele

Scriptingsprachen aus Programmiersprachen hervor gegangen sind, kann man prognostizieren, daß die Erkenntnisse der visuellen Programmierung dazu dienen werden, das Gebiet des visuellen Scriptings zu erweitern und zu bereichern.

3 Parsertechniken

In diesem Kapitel befasse ich mich mit Scriptingsprachen aus der Sicht der Entwickler. Eine Scriptingsprache zu entwickeln, ist ein völlig anderes Unterfangen, als ein Programm mit der selben dialoggesteuerten Funktionalität zu entwerfen. Allerdings werde ich mich nicht auf diese Unterschiede, sondern auf die reine Entwicklung einer Sprache konzentrieren. Nachfolgend werde ich die einzelnen Entwicklungsschritte aufführen, die mit der Formulierung der Sprache beginnen. Die nächsten Schritte beschreiben dann, wie diese Formulierung Schritt für Schritt umgewandelt wird, bis ein Interpreter vorliegt, der ein Script einliest und in Befehle umwandelt. Abschließend erfolgt ein kurzer Vergleich der zu verwendenden Werkzeuge.

Dieses Kapitel dient nicht als Anleitung zum eigenen Entwickeln einer Scripting- oder Programmiersprache, sondern soll nur einen Überblick über das vorliegende Thema vermitteln. Der geneigte Leser findet unter [Nie99] eine schrittweise Einführung dazu.

3.1 Entwicklungsschritte einer Scriptingsprache

Wie wir im vorigen Kapitel gesehen haben, können Scriptingsprachen einen Umfang besitzen, der von wenigen hochspezialisierten Befehlen, bis hin zu einer kompletten Programmiersprache reicht. Dennoch ist allen Sprachen gemein, daß sie von einem Interpreter gelesen, geprüft und in ausführbare Befehle umgewandelt werden.

Diese Schritte sind auch beim Interpretieren bzw. Kompilieren von Programmiersprachen nötig, um ausführbaren Code zu erzeugen. Wir werden im Folgenden sehen, daß Scripting- und Programmiersprachen bei ihrer Entwicklung bis zu einem bestimmten Punkt gleich behandelt werden.

Die folgende Abbildung aus [ASU86] zeigt die nötigen Schritte, um aus einem Quelltext ein lauffähiges Programm zu erzeugen. Scriptingsprachen gehen normalerweise bis zum dritten, zuweilen auch bis zum vierten Schritt.

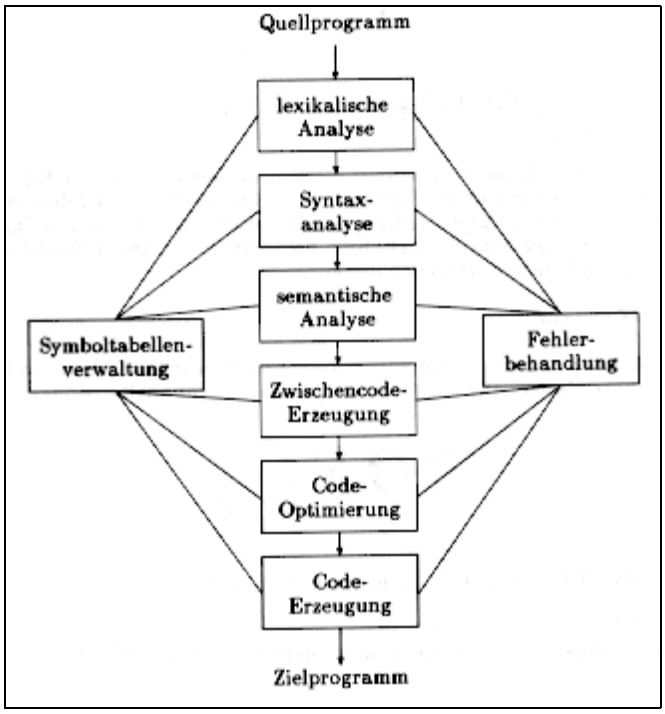

Abbildung 3.1: Phasen eines Compilers

3.2 Sprache und Grammatik

Der erste Schritt besteht darin, eine Scriptingsprache als formale Sprache zu definieren. Es müssen ihre Syntax und ihre Semantik festgelegt werden. Eine derart definierte Sprache bedeutet ebenfalls, daß nun eine Grammatik existiert, mit deren Produktionsregeln diese Sprache erzeugt wird.

3.2.1 Grammatiken

Eine Grammatik besteht aus folgenden Komponenten:

- Eine Menge von terminalen Symbolen;
- Eine Menge von nichtterminalen Symbolen, die disjunkt zur Menge der terminalen Symbole ist;
- Eine Menge von Produktionsregeln, die einer Menge von Symbolen (es muß mindestens ein Nichtterminal vorhanden sein) ein oder mehrere terminale und nichtterminale Symbole in beliebiger Kombination zuordnet;
- Ein Startsymbol, daß aus der Menge der nichtterminalen Symbole stammen muß.

Um terminale von nichtterminalen Symbolen zu unterscheiden, werden im Folgenden terminale Symbole in einfache Anführungszeichen eingeschlossen.

Es gibt mehrere Typen von Grammatiken, die kurz betrachtet werden. Typ 0-Grammatiken bilden die Basis der formalen Sprachen, während jeder weitere Typ eine Spezialisierung des vorherigen ist [Vos00].

- Typ 0 (rekursiv aufzählbare Grammatiken) – Dieses ist die Oberklasse aller formalen Sprachen. Die Produktionsregeln von Typ 0-Grammatiken lassen sich rekursiv benutzen, um so die dazugehörigen Typ 0-Sprachen zu erzeugen. Es gibt ansonsten keinerlei Einschränkungen.
- Typ 1 (kontextsensitive Grammatiken) – Typ 1-Grammatiken verhalten sich ähnlich wie Typ 0, mit dem Unterschied, daß bei den Produktionsregeln die Anzahl der zu ersetzenden Elemente in einem Wort, nicht verringert werden darf. Nach Anwendung einer Produktionsregel verfügt das Wort also über mindestens genauso viele Symbole wie vorher.
- Typ 2 (kontextfreie Grammatiken) – Dieser Grammatiktyp geht einen Schritt weiter und verlangt, daß in einem Wort nur exakt ein nichtterminales Symbol auf einmal gegen andere Symbole ausgetauscht werden darf. Es darf keine Produktionsregel geben, die mehrere nichtterminale auf einmal ersetzt.
- Typ 3 (reguläre Grammatiken) – In einer regulären Grammatik wird schließlich noch gefordert, daß eine Produktionsregel maximal ein terminales Symbol erzeugen darf.

Für die weitere Betrachtung sind nur kontextfreie Grammatiken von Interesse, da Programmiersprachen sich mit diesen am besten darstellen lassen. Eine kontextfreie Grammatik wird häufig in Backus-Naur Form (BNF) ([Gar98]) dargestellt, mit der sie mathematisch korrekt beschrieben werden kann. Die Abbildung 3.2 zeigt eine BNF, mit der man sämtliche rationalen Zahlen darstellen kann.

```
ZAHL            := '-' ABSZAHL | ABSZAHL;
ABSZAHL         := ZIFFERNFOLGE NACHKOMMA;
NACHKOMMA       := '.' ZIFFERNFOLGE | @;
ZIFFERNFOLGE    := ZIFFER | ZIFFER ZIFFERNFOLGE;
ZIFFER          := '0' | '1' | '2' | '3' | '4' |
                   '5' | '6' | '7' | '8' | '9';
```

Abbildung 3.2: Die rationalen Zahlen als Backus-Naur Form

In einer BNF werden terminale Symbole in einfache Anführungszeichen eingeschlossen. Das „|“ trennt Alternativen voneinander und das „@“ steht für das leere Symbol.

3.2.2 Kontextfreie Grammatiken

Kontextfreie Grammatiken werden deswegen so genannt, weil sie durch ihr einziges nichtterminales Symbol auf der linken Seite völlig unabhängig in der Wahl ihrer Alternativen sind. In Kontextsensitiven Grammatiken können auf der linken Seite nichtterminale und terminale Symbole gemeinsam stehen, um so die Menge der Alternativen einzuschränken. Diese Einschränkung ist für die meisten Programmiersprachen nicht tragbar.

Kontextfreie Grammatiken lassen sich nach unterschiedlichen Regeln beschreiben, weswegen sie weiter unterschieden werden. Die zwei bekanntesten Vertreter sind die LL- und die LR-Grammatiken, die ich in Abschnitt 3.4 wieder zur Sprache bringen werde. Zuerst ist jedoch die Syntax der Sprache von Interesse.

3.3 Lexikalische Analyse

Eine formalisierte Grammatik besteht aus mehreren Symbolen (*tokens*). Typische terminale Symbole sind Zahlen, Schlüsselwörter, mathematische Operatoren oder auch die sogenannten *white-spaces* (Leerzeichen, Tabulator und Zeilenumbruch).

Nachdem eine Sprache formalisiert und in BNF niedergeschrieben wurde, müssen zuerst sämtliche Symbole der Sprache identifiziert werden. Dieses geschieht mit einem lexikalischen Scanner, der Programme einliest und die gelesenen Daten den Symbolen zuordnet.

Scanner lassen sich mit moderatem Aufwand von Hand implementieren. Beliebter sind allerdings Scannergeneratoren, die aus einer Vorlage einen Scanner erzeugen. Das heißt, zu einer Sprache wird eine Vorlage aufgebaut, die beschreibt, welche Symbole in der Sprache vorkommen und wie diese aussehen. Das nachfolgende Bild zeigt die Beschreibung einiger typischer Symbole, wie sie in jeder Sprache vorkommen können. Die Notation orientiert sich dabei an dem Scannergenerator *flex* (siehe [Pax95]).

```
DIGIT    [0-9]
ID       [a-zA-Z][a-zA-Z0-9]*
NUMBER   (-?(({DIGIT}+)|({DIGIT}*\.?{DIGIT}+)([eE][+\-]?{DIGIT}+)?))
```

Abbildung 3.3: Lexikalische Symbole

`DIGIT` steht für eine Ziffer (‚0' bis ‚9'). `ID` beschreibt einen Identifikator, der mit einem Buchstaben (groß oder klein) beginnen muß. Danach können weitere Buchstaben oder Ziffern folgen. Mit dem komplexen Symbol `NUMBER` schließlich ist die Beschreibung von Zahlen möglich (z.B. ‚1', ‚3.141', ‚-.8E+9').

Ein Scannergenerator liest diese Vorlage ein und erzeugt daraus einen Scanner, der in der Lage ist, aus sämtlichen Programmen, die in der entsprechenden Sprache geschrieben wurden, die Symbole auszulesen. Die wichtigen Symbole werden zum Generator weitergereicht. Die restlichen (Leerzeichen, Zeilenumbrüche, Kommentare, etc.) werden ignoriert.

Im Parser wird geprüft, ob sich die Verkettung der Symbole auf die Regeln der Grammatik reduzieren läßt.

3.4 Grammatikalisches Parsen

Parser beinhalten die Produktionsregeln einer Grammatik. Die Symbolfolgen, die der Scanner schickt, werden mit diesen Regeln verglichen. Bei Ungleichheit ist das zugrundeliegende Script fehlerhaft und kann nicht vom Parser verarbeitet werden. Andernfalls können nun bestimmte Aktionen ausgeführt werden, die mit den Produktionsregeln verknüpft sind.

In Abschnitt 3.2.2 bin ich kurz darauf eingegangen, daß es mehrere Formen von Grammatiken gibt, die sich in gewissen Regeln bezüglich ihrer Schreibweise unterscheiden. Ich werde hier nun anhand von Beispielen Unterschiede und Gemeinsamkeiten hervorheben.

3.4.1 Das Problem des Vorausschauens

Grammatiken, die geparst werden, haben häufig das Problem ähnlicher Produktionen. Das folgende Beispiel veranschaulicht dieses (Terminale sind fett):

```
s := if e then s
   | if e then s else s
```

Eine typische If-Abfrage kann einen alternativen Teil (**`else`**) besitzen. Allerdings können typische Parser dies am Anfang der Produktion noch nicht wissen, da sie meistens nur ein Symbol vorausschauen (*look ahead*). Um aber die Beispielgramamtik zu parsen, muß der Parser die ersten fünf Symbole kennen, damit er eine Entscheidung fällen kann.

Diese Beschränkung ist historisch bedingt: Je mehr Symbole der Parser vorausschauen muß, bevor er sich für eine Produktion entscheidet, desto höher steigt der Aufwand und damit die Rechenzeit. Vor 10 bis 20 Jahren fiel dieser Zeitverlust noch merklich ins Gewicht, weswegen die Parserentwickler sich implizit darauf einigten, daß ein Parser maximal ein Symbol vorausschauen soll. Zudem wurden fast ausschließlich LR-Parser benutzt (vgl. Abschnitt 3.4.2), für die eine Vorausschau von einem Symbol ausreichend war. Kritische Produktionen wie die obige wurden umgeschrieben:

```
s := if e then s t
t := else s | @
```

Der Nachteil war, daß Grammatiken dadurch Einbußen in ihrer Lesbarkeit hatten. Mitte der 90er Jahre mehrten sich die Stimmen aus dem Entwicklerbereich wieder, die eine größere Symbolvorausschau für Parser favorisieren und ihre Vorzüge verteidigen. In [PQ96] entkräftet Terence J. Parr die Nachteile, die dagegen sprechen und zeigt weitere Vorteile auf.

3.4.2 LR- und LALR-Grammatiken

Das Parsen einer LR-Grammatik beginnt mit den untersten Ausdrücken. Ein Eingabesymbol wird direkt mit einem Ausdruck verglichen. Bei Gleichheit wird geprüft, ob der Ausdruck Bestandteil einer Produktion ist. In der Beispielgrammatik aus Bild 3.3 wird die Eingabe von „-4.56“ zuerst dazu führen, daß das Minuszeichen der obersten Produktion zugeordnet wird. Da diese noch nicht weiter reduziert werden kann, wird die „4“ genommen und zu einer `ZIFFER` und diese sofort zu einer `ZIFFERNFOLGE` reduziert. Da zum jetzigen Zeitpunkt, noch keine Informationen über den Ausdruck `NACHKOMMA` bekannt sind, wird nun der Punkt eingelesen. Am Ende ist die gesamte Zahl zum Ausdruck `ZAHL` reduziert.

Durch die umgekehrte Abarbeitung der LR-Grammatiken gelten für sie, bezüglich ihrer Schreibweise, nicht die Beschränkungen der LL-Grammatiken (siehe Abschnitt 3.4.3). Trotzdem haben sie einen Nachteil, der sie nicht uneingeschränkt benutzbar für Scriptingsprachen macht. Das folgende Beispiel aus [DS95] wird dieses verdeutlichen:

```
def           := param_spec return_spec ','
param_spec    := type | name_list ':' type
return_spec   := type | name ':' type
```

```
name_list       := name | name ',' name_list
type            := ID
name            := ID
```

`ID` ist ein terminales Symbol, das Identifikatoren beschreibt. Die Grammatik muß ein Symbol vorausschauen, ist also LR(1). Durch die Ähnlichkeit der beiden Produktionen `param_spec` und `return_spec` entsteht in den meisten Parsergeneratoren ein Konflikt, da angenommen wird, sie seien äquivalent. Sie werden zu einem Parserschritt zusammengefaßt. Die meisten Parsergeneratoren verstehen deshalb nur LALR(1)-Grammatiken, welche obige Konstrukte verbieten.

Es ist zwar möglich, Parsergeneratoren zu schreiben, die auch LR(1)-Grammatiken verstehen, doch ist dieses sehr schwer und die von ihnen erzeugten Parser sind sehr groß, was sie langsamer macht.

Weitere LR-Grammatiken sind z.B. LR(0), *Simple* LR(k) und *Generalized* LR(k), die jedoch für eine eingehende Betrachtung uninteressant sind, da es sich um weitere Untergruppen und Sonderfälle handelt. In einigen Fällen sind sie auch nicht mächtig genug, um Programmiersprachen zu beschreiben. Regeln der Form

```
A := A B | A B C | C E F
```

können z.B. von LR(0)-Grammatiken nicht bearbeitet werden, da sie nach der Eingabe von „ABC“ nicht beurteilen können, ob es sich um die zweite Alternative oder die erste in Kombination mit der dritten handelt. Es fehlt das nächste Symbol, das Aufklärung bringt.

3.4.3 LL-Grammatiken

LL-Grammatiken beginnen mit ihrer obersten Produktion und versuchen durch eindeutige Auswahl, die nächste Produktion zu bestimmen. Sollte das nicht möglich sein, wird es mit der folgenden Produktion erneut versucht. In Abbildung 3.2 etwa wird dieses erreicht, indem der Parser prüft, ob die Eingabe (z.B. „-4.56“) mit einem Minuszeichen beginnt. Auf diese Weise arbeitet er sich durch die Grammatik, bis er die terminalen Symbole erreicht hat (die Ziffern ‚0‘ bis ‚9‘).

Dadurch haben LL-Grammatiken den Nachteil, daß sie extrem abhängig von der Anzahl der vorausschauenden Symbole sind. Erschwerend kommt hinzu, daß es Produktionen gibt, bei denen die Zahl der voraus zuschauenden Symbole unbekannt ist. In diesem Beispiel kann das Symbol `B` beliebig oft vorkommen, bevor ein `C` oder `D` kommt:

```
A := (B)* C | (B)* D
```

Ein weiteres Problem ergibt sich durch das Abarbeiten von oben nach unten. Linksrekursive Produktionen der Form

```
A := A B
```

lassen die Parser in eine Endlosrekursion laufen.

3.4.4 Erweiterung von LL-Grammatiken

Aus den beiden vorangegangenen Abschnitten erkennen wir, daß sich (LA)LR-Grammatiken allgemein leichter schreiben lassen, da sie weniger Sonderfälle zu berücksichtigen haben. Entsprechend werden fast ausschließlich diese im Parser- und Compilerbau eingesetzt. Auf der anderen Seite besitzen LL-Grammatiken allerdings leistungsfähigere Parser, die Entwickler besser unterstützen (siehe unten).

Reine LL-Grammatiken sind jedoch keine Konkurenz zu LR-Grammatiken. Deshalb wurden weitere Untergruppen der LL-Grammatiken entwickelt, die ihre durch Aufbau und Abarbeitung entstandenen Beschränkungen aufheben sollen.

3.4.4.1 Dynamische LL-Grammatiken

Die Grundidee bei der Entwicklung von dynamischen LL-Grammatiken (dyn-LL(k)) war es, Mehrdeutigkeiten und andere Probleme zu eliminieren, die im vorigen Abschnitt aufgezeigt wurden. Die dynamischen LL-Grammatiken beschreiben dabei den Ansatz von Arnd Rußmann [Ruß97], wonach sich die Menge der Produktionen während des Parsens dynamisch ändern kann.

Dieses wird erreicht, indem jeder Produktionsregel ein Etikett (*label*) zugeordnet wird. Eine Produktion kann nur dann benutzt werden, wenn das dazugehörige Etikett zu dem aktuellen Zeitpunkt aktiv ist. Entsprechend benötigt eine dynamische Grammatik neben dem Startsymbol nun auch ein Startetikett. Die Menge der Produktionsregeln wird dahingehend erweitert, daß mit ihnen auch Zustandsübergänge ermöglicht werden (also dem Deaktivieren des aktuellen Etiketts und dem Aktivieren eines anderen). Desweiteren gibt es eine Menge, in der die jeweils gerade aktiven Produktionsregeln stehen.

Mit der Dynamisierung von LL-Grammatiken sind nun auch Regeln wie die folgende einfacher zu lösen (aus [Par93]):

```
element := ID '(' expression_list ')' // Arrayreferenz
         | ID '(' expression_list ')' // Funktionsaufruf
```

Obwohl syntaktisch völlig identisch, besteht zwischen den beiden Alternativen ein semantischer Unterschied, der vom Programmierer, normalerweise durch mehrere Abfragen in den Aktionen, die zu den Regeln gehören, erreicht werden muß. Er muß zum Beispiel prüfen, ob `ID` ein Array oder eine Funktion ist. Mit Etiketten wird dieses Problem umgangen. Je nach vorher gesetztem Etikett weiß der Programmierer ganz genau, welche der beiden Regeln gerade aktiv ist.

```
qa: element := ID '(' expression_list ')'
qf: element := ID '(' expression_list ')'
```

3.4.4.2 Prädikatenbehaftete LL-Grammatiken

Mit einer weiteren Methode zur Erweiterung von LL-Grammatiken, die mächtiger, aber auch aufwendiger ist, befassen sich Terence J. Parr und Russell W. Quong in [PQ95]. In ihrem

Ansatz werden Produktionsregeln mit syntaktischen bzw. semantischen Prädikaten erweitert (pred-LL(k)), um so zu entscheiden, welche Regel im Endeffekt vom Parser ausgewählt wird.

```
A := ((A)* B)? (A)* B | (A)* C
```

Das vorletzte Beispiel aus 3.4.3 wurde mit dem syntaktischen Prädikat `(A)* B` belegt. Die Regel prüft also zuerst, ob sie sich zu dieser Produktion umformen läßt. Gelingt dies, wird die erste Produktion genommen, ansonsten die zweite.

3.4.5 Gemeinsamkeiten von LL- und LR-Grammatiken

Von der Familie der dyn-LL(k) und pred-LL(k) einmal abgesehen, verändert das Umschreiben eine Grammatik nicht in ihrer Struktur. Deswegen ist es möglich, eine Grammatik eines Typs in eine andere zu konvertieren. Typische Regeln dazu finden sich in [ASU86] bzw. [Sta96], wo das Problem der Umwandlung expliziter betrachtet wird.

Es wäre allerdings falsch, davon auszugehen, daß eine Grammatik immer nur von einem Typ sein kann. Das folgende einfache Beispiel aus [Par93] zeigt eine Grammatik die gleichzeitig LR(0) und LL(2) ist:

```
A := B C | B D
```

4 Message Passing

In diesem Kapitel befasse ich mich mit der Kommunikation zwischen Prozessen, die mittels verschiedener Techniken realisiert werden kann. Durch das Thema meiner Arbeit liegt das Hauptaugenmerk auf dem *Message Passing*. Ich werde erklären, worum es sich dabei handelt, wofür es eingesetzt wird und warum man es anderen Techniken vorzieht. Zuletzt werde ich gängige Modelle des *Message Passing* genauer betrachten und vergleichen.

4.1 Was ist Message Passing?

Wenn Prozesse miteinander kommunizieren sollen, gibt es dafür verschiedene Möglichkeiten. Eine ist das gemeinsame Nutzen derselben Speicherbereiche (*shared memory*), in denen mehrere Prozesse nacheinander schreiben bzw. gleichzeitig lesen können. Diese Möglichkeit hat allerdings Nachteile. Tanenbaum listet in diesem Zusammenhang in [Tan94] einige auf, die entweder durch programmiersprachliche Defizite oder implementatorische Ungenauigkeiten eintreten. Gerade wenn mehrere Prozesse schreibend und lesend auf einen gemeinsamen Speicherbereich zugreifen, besteht die Gefahr von Synchronisationsfehlern, was in fehlerhaften Daten resultiert.

Der wichtigste von Tanenbaum angeführte Nachteil liegt jedoch in der Plattformgebundenheit, die gemeinsam genutzter Speicher mit sich bringt. Man kann einen Prozeß nicht auf eine andere Plattform auslagern, um so z.B. Performanzgewinne zu erzielen.

Eine ähnliche Technik beruht auf dem Schreiben in Dateien, die allerdings an den selben Problemen scheitert.

Hier kommt das Konzept des *Message Passing* ins Spiel: Prozesse kommunizieren miteinander, in dem sie sich Nachrichten schicken.

4.1.1 Übertragungsarten

Die einfachste Form des *Message Passing* geschieht von einem Prozeß zu einem anderen (*point-to-point*). Daneben gibt es noch die Möglichkeiten der Mehrfachsendung (*multicast*) und der Rundsendung (*broadcast*). Während die Mehrfachsendung benutzt wird, um dieselbe Nachricht an mehrere ausgewählte Prozesse zu versenden, benutzt man die Rundsendung, wenn alle Prozesse die Nachricht empfangen sollen.

Bei der Nachrichtenübermittlung wird zwischen blockierendem und nichtblockierendem Senden unterschieden. Blockierendes Senden tritt ein, wenn der sendende Prozeß auf eine Bestätigung des empfangenen Prozesses warten muß. Diese Form wird auch synchrones Senden genannt, da so zwei Prozesse ihren gemeinsamen Ablauf neu ausrichten. Synchrones Senden wird häufig im Zusammenhang mit aktionsorientiertem *Message Passing* benutzt (vgl. Abschnitt 4.1.2).

Nichtblockierendes oder asynchrones Senden läßt den sendenden Prozeß nicht warten, sondern puffert die Nachricht solange zwischen, bis der empfangende Prozeß bereit ist, sie anzunehmen. Beide Sendetypen sind austauschbar, da sie mit den Mitteln des jeweils anderen Typs realisiert werden können (vgl. [HH89]).

4.1.2 Aufbau und Typen von Nachrichten

Um eine Nachricht erfolgreich an einen anderen Prozeß zu versenden, sind einige Daten notwendig. Der Prozeß muß die Adresse seines Zieles kennen. Unter Umständen befindet sich der Zielprozeß nicht auf dem selben Rechner. Desweiteren muß die Nachricht über eine Identifikation verfügen, damit der empfangene Prozeß weiß, worum es sich bei der Nachricht handelt. Abhängig von der Identifikation der Nachricht, fallen auch die Daten aus, die sie transportiert. Einige Nachrichtentypen verlangen zudem eine Angabe der übermittelten Datentypen und der Gesamtlänge der Nachricht (vgl. [Sta00]). Nachrichten können zudem einen Zeitstempel erhalten, um den Zeitpunkt des Abschickens genau festzuhalten (vgl. Abschnitt 4.1.3).

Eine Nachricht wird, ausgehend von ihrer primären Bestimmung, in eine von zwei Kategorien unterteilt. Die Nachricht ist datenorientiert, wenn sie Nutzdaten mit sich trägt, die der empfangene Prozeß weiter verarbeitet. Ist es statt dessen ihre Bestimmung, den empfangenen Prozeß zu einer Aktivität zu veranlassen, so ist sie aktionsorientiert. In Kombination mit synchronem Senden, werden datenorientierte Nachrichten benutzt, um einen wartenden Prozeß zur Weiterarbeit zu bewegen (vgl. [HH89]).

4.1.3 Andere Eigenschaften von Nachrichten

Nachrichten haben oftmals eine zeitlich begrenzte Lebensdauer. Dieses kommt besonders in Systemen vor, in denen Prozesse ihre Nachrichten asynchron senden bzw. wo in kurzer Folge wiederholt dieselben Datensätze verschickt werden (z.B. Druck- oder Temperaturmeß-systeme).

Wartet eine Nachricht dabei zu lange darauf, daß der Empfänger bereit ist sie anzunehmen, kann es sein, daß sie ihre Gültigkeit verliert. Die Daten einer Radaranlage eines Flughafens z.B. werden in schneller Folge aktualisiert, damit die Fluglotsen einen genauen Überblick behalten, welche Maschine sich wo im Luftraum befindet. Ältere Daten sind nicht einfach nur obsolet geworden, sondern unter Umständen auch gefährlich.

Daraus ergibt sich ein ähnliches Problem bei Systemen mit vielen Prozessen und hoher Netzbelastung infolge von regem Nachrichtenaustausch. Hier kann es vorkommen, daß Nachrichten nicht in der Reihenfolge beim Zielprozeß eintreffen, in der sie abgeschickt wurden. Dabei ist es unerheblich, ob es sich dabei um einen oder mehrere sendende Prozesse handelt. Um dieses Problem zu umgehen, müssen Nachrichten mit einem Zeitstempel versehen werden, damit sich verfolgen läßt, wann sie abgesetzt wurden. Eingehende Nachrichten können so anstatt nach dem Zeitpunkt des Eintreffens (*receive order / RO*), nach dem Zeitpunkt des Abschickens *(time stamp order / TSO*) sortiert werden. Mit *TSO* bleibt gewähr-

leistet, daß die Nachrichten in der korrekten Reihenfolge bearbeitet werden. In dem Beispiel mit der Radaranlage bleibt so gewährleistet, daß die Fluglotsen nicht mit falschen Radarbildern konfrontiert werden (vgl. [HLA98]).

4.2 Verknüpfung von Ereignissen und Nachrichten

Ereignisse (*Events*) besitzen ähnliche Eigenschaften wie Nachrichten. Sie unterbrechen oder verändern den normalen Lauf eines Prozesses und zwingen ihn, darauf zu reagieren. Anders als Nachrichten, sind Ereignisse allerdings auf einen Prozeß beschränkt und werden nicht zwischen mehreren Prozessen verschickt. Sollten es die Umstände allerdings erforderlich machen, daß ein anderer Prozeß von einem Ereignis erfährt, kann man sich dafür den Nachrichtenaustausch zunutze machen. Unter Umständen kann ein Ereignis auch verworfen werden, wenn es als unwichtig erkannt wurde und es nicht synchron ist (siehe unten).

Nach Herrtwich/Hommel [HH89] werden Ereignisse nach synchronen und asynchronen unterschieden. Synchrone Ereignisse sind interne Ereignisse, die direkt im Ablauf des Prozesses auftreten. Sie werden auch Ausnahmen (*Exceptions*) genannt. Eine typische Ausnahme ist der Versuch, einen nicht alloziierten Speicherbereich zu referenzieren. Moderne Programmiersprachen bieten heutzutage die Möglichkeit, auf bestimmte Ausnahmen (z.B. Mausklick) zu prüfen und sie gegebenenfalls abzufangen, damit der Prozeß seinen Lauf nicht abbricht.

Asynchrone Ereignisse werden von externen Quellen erzeugt. Auch sie zwingen den gerade laufenden Prozeß zu einer Unterbrechung seiner Arbeit, bis das Ereignis bearbeitet ist. Asynchrone Ereignisse werden auch Unterbrechungen (*Interrupts*) genannt. Anders als bei Ausnahmen, muß es sich bei einer Unterbrechung nicht zwingend um einen Fehler handeln. Ein Interrupt ist ein Signal, dessen Bearbeitung oberste Priorität genießt. Typische Interrupts kommen von Peripheriegeräten wie Drucker oder Maus.

4.3 Message Passing Standards

In diesem Abschnitt möchte ich die bekanntesten Standards des *Message Passing* (PVM und MPI) näher betrachten und ihre Vor- und Nachteile aufzeigen. Es existieren noch weitere Konzepte und Implementierungen, jedoch handelt es bei diesen häufig um spezielle Lösungen, die von Firmen und Herstellern verteilter und paralleler Systeme entwickelt wurden. Diese Lösungen haben meistens den Nachteil, daß sie an eine Plattform gebunden sind und deshalb für die weitere Betrachtung uninteressant sind. Darüber hinaus gibt es eine allgemeine Aussage, die auch von Al Geist[4] vertreten wird:

[4] Geist arbeitete jeweils an der Entwicklung von PVM und MPI mit.

„PVM is the existing de facto standard for distributed computing and MPI is being touted as the future message passing standard.“ (aus [Gei96])

Dieses läßt erkennen, daß in MPI große Erwartungen gesetzt sind, die bisher noch nicht vollständig erfüllt wurden.

4.3.1 PVM: The Parallel Virtual Machine

Die Entwicklung von PVM [Gei94] begann im Sommer 1989 am Oak Ridge National Laboratory. Man beschäftigte sich dort im Zuge eines Forschungsprojektes mit den Möglichkeiten des *distributed heterogenous computing*. Um ein Grundgerüst für die weitere Forschung zu haben, wurde PVM entwickelt, das in seiner zweiten Version (1991) publik gemacht und schon bald weltweit eingesetzt wurde. PVM liegt zur Zeit in der Version 3.4 vor.

PVM ist eine Softwareumgebung für heterogene Netzwerke, mit der man eine Gruppe von Rechnern wie einen einzelnen Mehrprozessorrechner benutzen kann. Alle Plattformen zusammen werden als sog. virtuelle Maschine angesehen. Beim Informationsaustausch zwischen Prozessen wurde ein einfaches *Message Passing*-Modell implementiert, damit einerseits Portabilität zwischen den unterschiedlichen Plattformen gewährleistet bleibt und andererseits gesendete Nachrichten schnell beim Zielprozeß ankommen.

4.3.2 MPI: Der Message Passing Interface Standard

Im Jahr 1992 versammelten sich Mitglieder und Vertreter von Instituten, Laboratorien und Hardwareherstellern, um mit MPI [MPI94] einen neuen Standard zu schaffen. Mit dem Wissen und der Erfahrung über *Message Passing* sollte ein Standard geschaffen werden, der einerseits hochperformant ist und sich andererseits auf viele Plattformen übertragen läßt. Wie PVM ist auch MPI offen und Hersteller von Parallelrechnern werden ermutigt, ihre Hardware an die Schnittstellen anzupassen (vgl. [WAL95]). Von MPI existiert seit 1997 die Version 2.0 (MPI2 / siehe [MPI97]).

4.3.3 Vergleich von PVM und MPI

Alle *Message Passing*-Modelle besitzen einige gemeinsame Charakteristika. Jeder Prozeß läuft separat (allerdings nicht zwangsläufig unabhängig, siehe unten) von den anderen und kontrolliert seinen eigenen, privaten Speicherbereich. Die verschiedenen Prozesse kommunizieren und synchronisieren sich über Bibliotheksfunktionen, die das Modell zur Verfügung stellt (siehe [KEC99]). Im folgenden möchte ich spezifische Unterschiede und Gemeinsamkeiten zwischen PVM und MPI in wichtigen Bereichen behandeln.

Beide Standards erlauben, daß ein Programm, welches auf einer Architektur implementiert wurde, sich ohne Veränderung auf eine andere übertragen, dort kompilieren und schließlich ausführen läßt. Der PVM Standard geht allerdings dahingegehend weiter, daß die beiden Programme untereinander kommunizieren können, was in MPI nicht spezifiziert ist. Dafür bietet PVM Funktionen zur Konvertierung von numerischen Formaten zwischen unterschiedlichen Plattformen an ([KEC99]).

Auch sonst besitzt PVM Festlegungen, auf die MPI verzichtet, wie etwa die Kommunikation von Prozessen, deren Programme in unterschiedlichen Sprachen implementiert wurden oder die Einbindung von anderen Arten der Interprozeßkommunikation (z.B. *Shared Memory* / vgl. [MPI94]).

Dieses hängt vor allem mit den Konzepten zusammen, auf denen die beiden Standards beruhen. PVM behandelt mehrere Rechner wie einen Mehrprozeßrechner, die sog. virtuelle Maschine (*virtual machine*), während MPI sich ausschließlich auf die Nachrichtenverwaltung konzentriert. Das Konzept der virtuellen Maschine sowie das dazugehörige Resourcenmanagement existiert in MPI überhaupt nicht.

Auf der anderen Seite bietet MPI durch die Konzentration auf die Nachrichtenverwaltung eine umfangreichere Schnittstelle für Programmierer. Das Anlegen von getrennten Topologien und Bildung von Prozeßgruppen läßt sich in MPI mit entsprechenden Funktionen realisieren, die in PVM nicht vorhanden sind.

In [Gei96] sind drei Unterschiede vorgestellt, die im folgenden betrachtet werden.

4.3.3.1 Laufzeitumgebung

In der virtuellen Maschine von PVM existiert der sogenannte PVM-*Demon* (*pvmd*), der zu Beginn alle weiteren Prozesse startet. Er hat auch die Kontrolle über sämtliche Prozesse, inklusive derjenigen, die später hinzukommen.

Der Standard stellt eine breite Palette von Funktionen zur Verfügung, um sich diverse Informationen über diese Prozesse geben zu lassen. Weitere Funktionen ermöglichen die Einflußnahme auf ihren Ablauf. Es ist in den PVM-Umgebungen jederzeit möglich, Prozesse nach Belieben zu starten bzw. zu terminieren. Ermöglicht wird dieses durch sog. Ereignisnachrichten (*event messages*), die andere Prozesse über Änderungen informieren. Damit ist es möglich, daß die Ressourcen (Prozessoren, Hauptspeicher, Auslagerungsspeicher) neu verteilt werden, um den optimalen Lauf des Gesamtsystems weiterhin zu garantieren. Das Senden dieser Ereignisnachrichten entspricht dem aktionsorientierten Nachrichtenaustausch (siehe Abschnitt 4.1.2).

Obwohl man in MPI inzwischen auch Prozesse dynamisch zur Laufzeit erzeugen oder beenden kann, verzichtet MPI in seiner Spezifikation auf eine explizite Ressourcenverwaltung (siehe oben). Jeder Prozeß ist für sich eigenständig und nur sich selbst verantwortlich. Welche Auswirkungen das bei Fehlverhalten hat, wird im nächsten Abschnitt erläutert.

4.3.3.2 Fehlertoleranz und -notifikation

Beide Standards garantieren einen sicheren Nachrichtenaustausch. Das bedeutet, daß Nachrichten solange gesendet werden, bis sie am Ziel eingetroffen sind und daß ihre Reihenfolge erhalten bleibt. Trotzdem können die Prozesse, die die Nachrichten verschicken bzw. empfangen, fehlerhaft arbeiten.

In PVM muß ein Prozeß, der die Dienste anderer beansprucht, darauf vorbereitet sein, daß ein Dienst nicht erbracht werden kann, weil der Prozeß, der ihn ausführen soll, abstürzt. Auch

hier sind die Ereignisnachrichten hilfreich. Der PVM-*Demon* (siehe oben), teilt dem wartenden Prozeß mit, daß der andere Prozeß, auf den er wartet, nicht mehr existent ist. Anstatt daß der erste Prozeß nun für den Rest seiner Laufzeit in einer Warteschleife hängt, kann er nun Gegenmaßnahmen ergreifen, z.B. auf den Dienst verzichten, einen anderen Prozeß mit der Aufgabe betrauen oder sogar einen neuen Prozeß erzeugen, der den Dienst übertragen bekommt.

In MPI wird bei einem Fehler in einem Prozeß normalerweise das gesamte System beendet. Ein Programmierer kann dieses nur umgehen, wenn er eigene Fehlerbehandlungsroutinen implementiert, um das System robuster zu gestalten. Das größte Problem ist aber noch immer, daß MPI nicht in der Lage ist, das vorzeitige Terminieren eines Prozesses auszugleichen. Dieses hängt mit den Kommunikatoren (*Communicators*) zusammen, die MPI benutzt, um mehrere Prozesse in einem Kontext zusammenzufassen. Beim Starten werden sämtliche Prozesse zuerst in einem Hauptkommunikator gesammelt. Dadurch, daß er statisch ist und nachträglich keine Prozesse mehr aufnehmen oder entlassen kann, wird er durch das Fehlen eines Prozesses ungültig. Das Verhalten des restlichen Systems ist damit nicht mehr vorhersagbar.

4.3.3.3 Nachrichten

Nachrichten sind in MPI und PVM ziemlich unterschiedlich. Es existieren natürlich Gemeinsamkeiten, wie etwa, daß sie keine vorgeschriebene Länge benötigen oder Variablen als formatierte Datenströme (*contiguous data*) versenden können. Doch während PVM nur Basistypen versenden kann, ist MPI in der Lage, verschiedene Komposittypen zu erzeugen und diese zu versenden, was vor allem den Programmierer entlastet, da er nicht mehr gezwungen ist, die Nachricht aus einzelnen Typen zusammenzubauen.

Die größten Unterschiede liegen aber in den Sende- und Empfangsoperationen für die Nachrichten. PVM unterstützt ausschließlich nichtblockierendes Senden. Die Laufzeitumgebung stellt einen Puffer bereit, der die Nachrichten so lange aufnimmt, bis sie vom Zielprozeß empfangen werden können. Auf der Empfängerseite existieren andererseits mehrere Funktionen zum Empfangen der Nachrichten durch direkte Abfrage (*polling*), warten für eine bestimmte Zeit (*timeout*), unendlich lange (blockierend) oder überhaupt nicht (nichtblockierend) [Gei94].

MPI legt den Schwerpunkt auf die Sendeoperationen und bietet hier verschiedene Funktionen an, die sich jeweils blockierend oder nichtblockierend benutzen lassen (dieses gilt auch für das Empfangen weiter unten). Neben dem normalen Senden erlaubt MPI auch Nachrichten erst zu senden, wenn der empfangene Prozeß bereit ist. Eine ähnliche Funktion beendet die Sendeoperation erst, wenn sie die Mitteilung erhalten hat, daß die Nachricht korrekt eingetroffen ist (synchrones senden). Natürlich kann man die Nachrichten auch an einen bestimmten Puffer senden, der sich dann um die Zustellung zum Zielprozeß kümmert. Im Gegensatz dazu gibt es nur eine einfache Funktion, die den Nachrichtenempfang regelt [DW96].

Beide Umgebungen ermöglichen es einem Entwickler, sich eigene Nachrichten zu definieren. Die Behandlung dieser Nachrichten (*message handling*) muß vom Entwickler selber implementiert werden. MPI und PVI stellen dafür spezielle Funktionen zur Verfügung, damit die selbst erzeugten Nachrichten sich nahtlos in das Gesamtsystem einfügen.

PVM setzt allerdings voraus, daß es nur auf Nachrichten reagiert, die von anderen Prozessen aus der PVM-Umgebung gesendet wurden. Nachrichten, die von außerhalb gesendet wurden, werden durch die *Message Handler* ignoriert und müssen mit den üblichen programmiersprachlichen Mitteln behandelt werden [Gei97].

In PVM 3.4 wurden sogenannte *Message Mailboxes* implementiert, die dazu eingesetzt werden können, um Nachrichten anzunehmen und zu speichern. Anders als bei einem Zwischenpuffer, der die Nachricht automatisch weitergibt, werden hier die Nachrichten in eine Liste geschrieben. Ein anderer Prozeß kann nun in einer *Mailbox* nach einer bestimmten Nachricht fragen (er muß wissen, wie sie heißt) und bekommt diese dann zugeschickt, wenn sie vorhanden ist.

5 HLA und Jini

Mit HLA (*High Level Architecture*) und Jini liegen zwei mächtige Architekturen aus dem Bereich der verteilten Systeme vor. Sie sind deshalb so interessant, da bei ihnen andere Aspekte in den Vordergrund gestellt wurden und sie sich so von den ‚traditionellen' verteilten Systemen (vgl. PVM und MPI in Kapitel 4) unterscheiden.

In diesem Kapitel möchte ich kurz den Aufbau dieser beiden Architekturen vorstellen. Er wird in Kapitel 9 interessant, wo sie als Beispiele für die allgemeine Vorgehensweise der Integration von Scriptingsprachen dienen sollen.

5.1 Die High Level Architecture

Die High Level Architecture (HLA) [HLA98a, HLA98b, HLA98c] wurde zu dem Zweck entwickelt, Interoperabilität und Wiederverwendbarkeit einzelner Simulationskomponenten zu ermöglichen. Damit hebt sie sich von gängigen Architekturen für verteilte Simulation ab, deren Hauptanliegen die Effizienzsteigerung paralleler Prozesse ist. Ihre Entwicklung wurde im März 1995 vom Department of Defense in Auftrag gegeben und liegt seit 1998 in der Version 1.3 vor.

5.1.1 Grundstruktur von HLA

Die wichtigste Eigenschaft von HLA ist, daß sie nur eine allgemeine Schnittstelle liefert. Es wird keine Entwicklungsumgebung oder Implementierung vorgeschrieben. Abbildung 5.1 aus [DKW98] zeigt einen Überblick über die Struktur eines typischen Simulationsaufbaus in HLA.

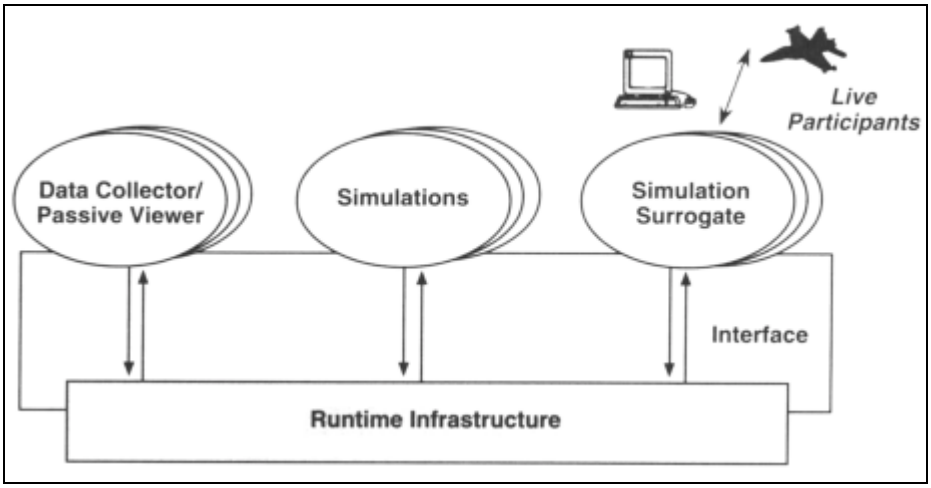

Abbildung 5.1: High Level Architecture

Jegliche Applikationen, die eine Schnittstelle zu HLA besitzen, gelten als *Federates*. Zusammen mit der *Runtime Infrastructure* (RTI) bilden sie eine Simulationsumgebung (genannt *Federation*). Grob gesagt, kann man eine *Federation* als verteilte Applikation

ansehen, deren Komponenten die einzelnen *Federates* darstellen. Innerhalb der *Federates* wird eine besondere Untergruppe unterschieden, die *Simulations*, die während ihrer Laufzeit Werte verändern oder Ereignisse auslösen. Die restlichen *Federates* können aus Datenbanken und passiven Beobachtern bestehen. Den *Federations* stellt die RTI Dienste zur Koordination und Kommunikation zur Verfügung.

5.1.2 Kommunikation in HLA

Die Kommunikation zwischen *Federates* geschieht ausschließlich über die RTI. Sie sorgt für Koordination, Synchronisation und einen reibungslosen Datenaustausch. Für die Korrektheit der Daten ist sie allerdings nicht zuständig. Diese Aufgabe übernimmt das *Federation Object Model* (FOM).

Im FOM steht formal beschrieben, welche Attribute aus den einzelnen *Federates* zum Austausch freigegeben sind und welche notwendigen Voraussetzung dafür erfüllt sein müssen. Dieser Formalismus ist ein grundlegender Bestandteil von HLA im Kontext der Wiederverwendbarkeit von *Federates*. Letztgenannte können nur diejenigen Attribute abonnieren, die in der FOM aufgeführt sind.

Simulation Object Models (SOM) auf der anderen Seite existieren für jede *Simulation*. In einem SOM werden Klassen und die *Interactions* einer *Simulation* aufgelistet.

Interactions sind Nachrichten bzw. Ereignisse, die während einer Simulation auftreten, wenn eine Klasse in einem *Federate* dessen Zustand ändert. Sie lassen sich hierarchisch organisieren, so daß sich Untergruppen von *Interactions* bilden lassen, die Spezialisierungen darstellen. Auch *Interactions* können von *Federates* abonniert werden. Wird eine *Interaction* abonniert, die über Spezialisierungen verfügt, so werden diese implizit mit abonniert.

Datenaustausch ist in HLA ereignisgesteuert. Ändert ein Attribut seinen Wert entsprechend den Vorgaben aus der FOM (z.B. wenn eine Zustandsänderung einen gewissen Schwellwert überschreitet), werden die *Federates*, die es abonniert haben automatisch durch das RTI benachrichtigt. Das Auslösen einer *Interaction* wird analog behandelt.

Um eine unnötige Datenflut zu reduzieren, bietet HLA die Möglichkeit, beim Abonnieren von Attributen genaue Angaben zu machen, unter welchen Voraussetzungen eine Zustandsänderung jeweils mitgeteilt werden soll (*routing spaces*). Es besteht so die Möglichkeit, aus dem Wertebereich eines Attributes einen relevanten Ausschnitt zu wählen, der betrachtet wird. Solange das Attribut Werte außerhalb des Ausschnitts annimmt, werden keine Änderungen an abonnierende *Federates* geschickt.

5.2 Jini

Hinter einem auf Jini [JIN99] basierendem System steht die Idee, Benutzer und ihre benötigten Ressourcen zu vereinen. Das Ziel von Jini ist, das Netz flexibel und einfacher administrierbar zumachen, damit vorhandene Ressourcen leicht zugänglich sind. Jini ist von

SUN Microsystems auf Basis der Java Virtual Machine entwickelt worden und wurde im Januar 1999 in seiner aktuellen Version vorgestellt.

5.2.1 Aufbau des Jini-Systems

Das Jini-System läßt sich laut den Entwicklern nicht als eine Ansammlung von Clients und Servern betrachten, sondern eher von einer Gruppe von Dienstleistungen (*Services*). Das System läßt sich in drei Bestandteile zerlegen, von denen die *Services* einer sind.

5.2.1.1 Services

Services sind Dienste, die in einem Jini-System zur Verfügung gestellt werden. Jeder *Service* übernimmt eine bestimmte Aufgabe (ein Dokument drucken, Konvertierung von einem Textformat in ein anderes). Zur Laufzeit können *Services* dynamisch in eine *Federation* integriert bzw. wieder aus ihr entfernt werden (siehe Abschnitt 5.2.2). Sie lassen sich auch kombinieren, um eine Folge von Abläufen zu bilden; eine sogenannte *Transaction*. Auch ein einzelner Service, der eine Tätigkeit wiederholt ausführt, wird zu einer *Transaction*. Alle Services zusammen bilden die *Federation*.

Um ein *Service* implementieren zu können, muß sich der Entwickler an das Programmiermodel halten, das unter anderem die Programmiersprache (Java) und einige Basisschnittstellen vorgibt.

5.2.1.2 Programmiermodel

Das Programmiermodel (*programming model*) bietet ein Set von Schnittstellen an, die jeder neu zu implementierende *Service* benötigt, um in einer *Federation* teilnehmen zu können (siehe Abschnitt 5.2.2).

5.2.1.3 Infrastruktur

Die Infrastruktur bildet den Kern von Jini, da sie die Basisfunktionen und -komponenten zur Verfügung stellt.

Die *Lookup Services* (LS) sind eine dieser Funktionen. Sie stellen einen Kontaktpunkt zwischen dem System und dessen Benutzern dar. Sie sind hierarchisch organisiert, so daß ein LS weitere, spezialisierte LS besitzen kann.

Die Infrastruktur stellt außerdem ein Sicherheitssystem zur Verfügung, die das vorhandene des RMI-Modells (*remote method invocation*) für die Anwendung auf verteilten Systemen erweitert.

5.2.2 Kommunikation

Jini erlaubt *Services*, Benutzern und anderen Komponenten (den Clients) nach Belieben in eine *Federation* einzutreten oder diese zu verlassen. Häufig ist dieser Vorgang sogar automatisch, etwa wenn ein *Service* Dienste zur Verfügung stellt, die nur für eine bestimmte Zeit benötigt werden. Eine Applikation zum Auswerten von Simulationsergebnissen muß nicht zwangsläufig von Anfang an in der *Federation* sein. Es genügt, wenn ein *Service* die Daten

protokolliert und erst nach Beendigung der Simulation die Funktionen der Auswertungsapplikation aufruft.

Der Infrastruktur müssen alle vorhandenen *Services* und deren Schnittstellen bekannt sein. Damit die Schnittstellen dem System aber zugänglich gemacht werden können, muß ein neuer *Service* zunächst einen passenden LS finden, dem er dann mitteilt, welche Dienste er bereitstellt. Wenn die vorgegebenen LS von Jini nicht ausreichen, steht es einem Programmierer hier frei, neue zu schreiben, um die Infrastruktur zu erweitern.

Das Abonnieren von Diensten läuft auf Basis des *Leasings* ab. Zwei Services schließen quasi ein Abkommen. Der Benutzer gibt an, wie lange er gewisse Dienste benötigt und der Anbieter legt fest, wie lange der Benutzer die gewünschten Dienste im Endeffekt nutzen darf und ob der Zugriff auf sie exklusiv erfolgt oder nicht.

Für die weitere Kommunikation zwischen den *Services* sind die LS nicht mehr nötig. Die Diensterbringung läuft als RMI ab, der Javavariante des RPC (*remote procedure call*).

Events dagegen werden wie in HLA behandelt. Sie können von jedem *Service* abonniert werden. Wird ein *Event* ausgelöst, werden alle *Services* benachrichtigt, die es abonniert haben.

6 Die CLEAR-Architektur

Ich habe mir als Plattform für meinen Prototypen das bei artec entwickelte System CLEAR ausgesucht. Die Aufgaben, für die CLEAR entwickelt wurde, sind schon in Abschnitt 1.1 kurz beschrieben worden. In diesem Kapitel möchte ich mich auf die technischen Aspekte des Systems konzentrieren. In CLEAR kommt eine eigene *Message Passing* Architektur zum Einsatz, die, bei genauerer Betrachtung, einige der Konzepte beinhaltet, die in den vorherigen Kapiteln besprochenen wurden. Das wichtigste Modul ist der Server ROMAN (*Real Object Manager*). Er hält die aktuelle Szene im Speicher und synchronisiert sie mit den angeschlossenen Clients.

6.1 Der Real Object Manager (ROMAN)

In der Architektur von CLEAR ist der ROMAN eine zentrale Komponente. Synchronisation der Szene und Verwaltung der Clientdienste werden ausschließlich von ihm vorgenommen. Direkte Kommunikation zwischen Clients ist nicht erlaubt. Entsprechend erhält ein Client auch nur über den ROMAN seine Nachrichten. Abbildung 6.1, das aus [Ern00] entnommen wurde, zeigt das Gesamtsystem.

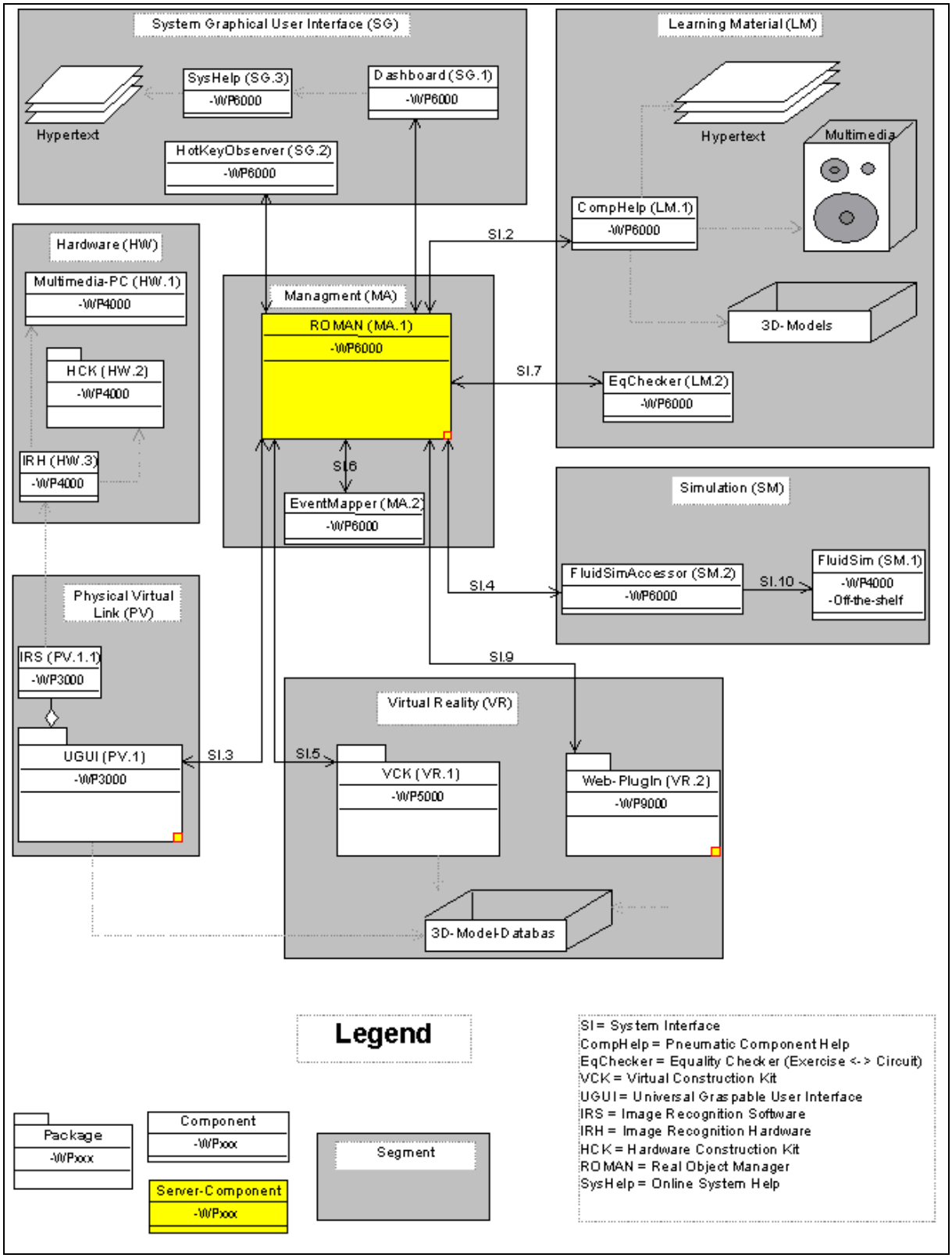

Abbildung 6.1: CLEAR-Architektur

Zur Kommunikation abonnieren Clients die für sie wichtigen Nachrichten im ROMAN. Letzterer fungiert als Verteiler, der dafür sorgt, daß Nachrichten nur an Abonnenten weitergegeben werden. Unter Umständen kann der Inhalt einer Nachricht das Absetzen einer

weiteren Nachricht (z.B. einer Antwort) nötig machen. Zu diesem Zweck existiert ein spezieller Client, der *EventMapper*.

6.2 Der EventMapper-Client

Die wichtigste Funktion des *EventMappers* ist das Beantworten eines bestimmten Nachrichtentyps (siehe Abschnitt 6.3), die ihm vom ROMAN gesendet werden. Dazu meldet er sich bei seiner Initialisierung wie ein gewöhnlicher Client am ROMAN an und abonniert diese Nachricht. In Abbildung 6.2 ist zu sehen, wie eine Nachricht, angefangen beim aufrufenden Benutzer (gleichzeitige Betätigung der Tasten ‚Ctrl' und ‚F2'), durch das System zum *EventMapper* geschickt wird. Aus seiner internen Zuordnungstabelle liest der *EventMapper* die entsprechende Antwortnachricht (RECOGNISE_WORLD_REQUEST) aus und sendet diese zurück an den ROMAN, der sie an die Clients weiterleitet, die die Nachricht abonniert haben.

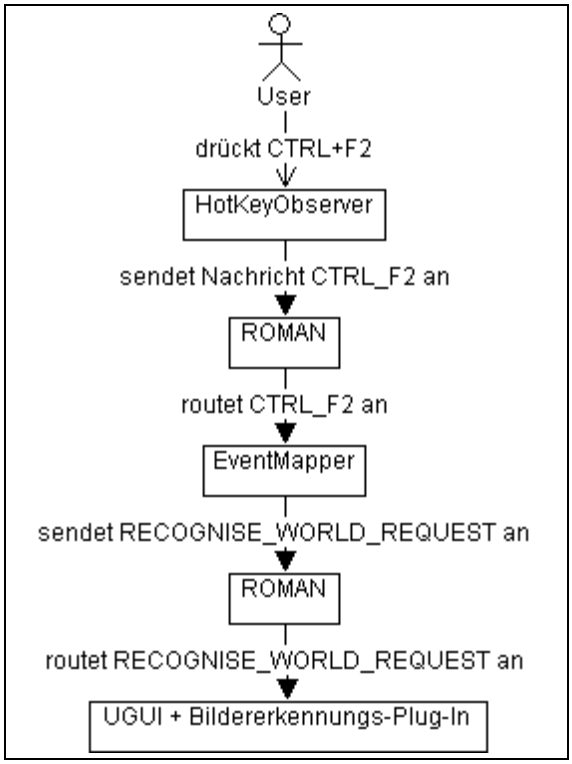

Abbildung 6.2: Nachrichten in CLEAR

Der *EventMapper* ist mein wichtigster Ansatzpunkt, da seine Funktionalität fest implementiert ist. Der Versuch, diese Funktionalität komplett in eine Scriptingsprache zu übertragen, würde mit Sicherheit den für diese Arbeit vorgesehenen Rahmen sprengen, weswegen ich nur einige Funktionen auswählen werde, die ich in Abschnitt 6.4 besprechen werde.

6.3 Aufbau von Nachrichten

Sämtliche Nachrichten in CLEAR sind nach einem bestimmten Schema aufgebaut.

Byte 1	Byte 2-5	Byte 6 – (6+ Länge des Datenstring)
'$'	Länge des Datenstring in network byte order	Der Datenstring

Der Datenstring enthält den Bezeichner und eventuelle Parameter der Nachricht. Der Aufbau ist recht einfach gehalten, da sämtliche Datentypen vor dem Versenden ebenfalls in Strings konvertiert werden müssen. Der Datenstring baut sich folgendermaßen auf:

```
<Nachrichtenbezeichner>
<Parameter_1>
...
<Parameter_n>
```

Ein einzeiliger Parameter wird folgendermaßen in den String eingetragen:

```
<PARAMETER_NAME> <PARAMETER_WERT>
```

Für Mehrzeilige gilt diese Schreibweise:

```
BEGIN <PARAMETER_NAME>
<PARAMETER_WERT>
END <PARAMETER_NAME>
```

Einzeilige Parameter lassen sich alternativ als mehrzeilige eintragen. Es existieren keine weiteren Symbole, die ein Ende oder einen bestimmten Abschnitt der Nachricht markieren.

Für die vorliegende Arbeit sind zudem die Nachrichten, welche für die Arbeit mit Ereignissen (*Events*) implementiert sind, von besonderem Interesse. In Abschnitt 6.3.2 wird angegeben, welche Parameter Ereignisse jeweils benötigen.

6.3.1 Erzeugen und Abonnieren von Events

Um ein Event zu erzeugen, sendet ein Client eine Nachricht mit dem Bezeichner CREATE EVENT. Als Parameter werden Name und Gruppenzugehörigkeit benötigt. ROMAN trägt das Event in eine interne Liste ein. Mit SUBSCRIBE EVENT kann der Client das soeben erzeugte Event abonnieren. Der Eventname ist dafür ausreichend. ROMAN sendet eine Antwortnachricht (SUBSCRIBE RESPONSE EVENT) um dem Client das Abonnement zu bestätigen.

Jedes Event ist Mitglied einer Gruppe. Damit ein Client alle Events einer Gruppe abonniert, sendet er eine Nachricht vom Typ SUBSCRIBE GROUP mit dem entsprechenden Gruppennamen. Die Antwort vom ROMAN fällt entsprechend aus (SUBSCRIBE RESPONSE GROUP). Soll ein neues Event zu einer Gruppe gehören, die noch nicht existiert, so wird diese automatisch erzeugt.

Ein Abonnement kann jederzeit wieder gekündigt werden, wobei auch hier zwischen einzelnen Events (UNSUBSCRIBE EVENT) und ganzen Gruppen (UNSUBSCRIBE GROUP) unterschieden wird.

6.3.2 Auslösen von Events

Clients bzw. ROMAN können jederzeit ein Ereignis per Nachricht verschicken. Der Bezeichner einer solchen Nachricht lautet RAISE EVENT. Die Nachricht selber beinhaltet neben dem Ereignisnamen (Parameter NAME) weitere Parameter. Um den Ursprung der Nachricht bestimmen zu können, wird der Name des Clients (Parameter CLIENTNAME) übermittelt. Einige Ereignisse führen darüber hinaus noch Nutzdaten (Parameter TEXT) mit sich. Dieses ist auch der Nachrichtentyp, der vom *EventMapper* abonniert wird.

6.4 Auswertung der Ist-Analyse

Nach der Ist-Analyse der vorherigen Abschnitte sind folgende Defizite des EventMappers erkennbar:

1. Verknüpfungen sind auf Ereignisse beschränkt.
2. Durch die festgelegten Verknüpfungen ist der Einsatz von selbst definierten Ereignissen eingeschränkt.
3. Ändern lassen sich diese Verknüpfung nur durch das Umschreiben des Programmcodes.

Auf Grundlage dieser Analyse kann ich nun damit beginnen, eine auf Scripting basierende Erweiterung zu entwickeln.

7 Scriptbasierter Nachrichtenaustausch

Mit den Erkenntnissen, die ich über den Aufbau von Scriptingsprachen und *Message Passing* Systemen in den vorangegangenen Kapiteln gesammelt habe, möchte ich nun eine eigene Sprache entwickeln und sie in das CLEAR-System integrieren. In diesem Kapitel dokumentiere ich den theoretischen Entwicklungsprozeß.

7.1 Anforderungsdefinition der Nachrichtenschnittstelle

Nach der Untersuchung des CLEAR-Systems in Kapitel 6 können mehrere Anforderungen konkretisiert werden, die in die Scriptingsprache eingearbeitet werden sollen:

1. Verknüpfen von beliebigen Nachrichten:

   Bisher können nur Nachrichten vom Typ RAISE EVENT miteinander verknüpft werden (siehe Abschnitt 6.2). Durch die Erweiterung sollen alle Arten von Nachrichten verknüpft werden können; anders ausgedrückt, auf jede Art von eingehender Nachricht kann nun mit dem Absetzen einer anderen (Antwort-) Nachricht reagiert werden. Der EventMapper wird zum MessageMapper erweitert.

2. Erzeugen und Abonnieren von selbst definierten Ereignissen:

   Diese Funktion wird bereits durch die Architektur mit den Nachrichten CREATE EVENT und SUBSCRIBE EVENT unterstützt. Ereignisgruppen (SUBSCRIBE GROUP) erlauben es, vom Kontext her zusammengehörige Ereignisse zu verbinden. Dadurch wird es ermöglicht, mit dem Abonnieren einer Gruppe implizit sämtliche dazugehörigen Ereignisse zu abonnieren. Das Erzeugen und Abonnieren geschieht bisher aus dem Programmcode der Clients heraus. Es wäre sinnvoll, Teile davon in die Scriptingsprache auszulagern, wo sie zentral an einer Stelle mit der restlichen Funktionalität zusammengefaßt sind.

3. Fallunterscheidungen:

   Eine Nachricht muß auf ihren Namen und ihre evtl. vorhandenen Parameter geprüft werden können. Die Kriterien, nach denen abgefragt wird, ergeben sich aus dem Inhalt einer soeben empfangenen Nachricht. Die Fallunterscheidungen sollen dazu benutzt werden, zu bestimmen, ob die geprüfte Verknüpfung gültig ist, um dann die verknüpfte Nachricht abzuschicken.

4. Variablen definieren:

   Die Möglichkeit eigene Variablen zu benutzen, erlaubt das Setzen von Schaltern. Diese können in den Fallunterscheidungen aus 3. abgefragt werden. Dieses erlaubt einem, stärkeren Einfluß auf das Absetzen von Nachrichten zu nehmen.

Mit diesen Vorgaben ist es nun möglich, die Scriptingsprache zu spezifizieren und eine neue Struktur für den MessageMapper zu entwickeln.

7.2 Grobspezifikation

Anhand der Spezifikation läßt sich nun die Grobstruktur für die Scriptingsprache und den Parser erstellen. Die ersten vier Abschnitte gehen dabei direkt auf die Punkte aus Abschnitt 7.1 ein.

7.2.1 Verknüpfung von beliebigen Nachrichten

Wie mehrfach erwähnt, wird im EventMapper nur die Nachricht RAISE EVENT verknüpft. Der Typ der Nachricht wird dabei ignoriert, da er bekannt ist und deshalb nicht benötigt wird. Verknüpfungen finden auf der Ebene der Ereignisnamen statt. Mit der Erweiterung auf beliebige Nachrichten wird der Nachrichtentyp unerläßlich. Die Ereignisnamen sind aber ebenfalls weiterhin wichtig, da sonst alle bisherigen Ereignisverknüpfungen zu wenig aussagekräftigen RAISE EVENT → RAISE EVENT Verknüpfungen werden. Die Nachrichtenverknüpfungen müssen deshalb Vorbedingungen erhalten, in denen die Nachrichtenparameter auf bestimmte Werte geprüft werden können. Im Fall der ehemaligen Ereignisverknüpfungen ist dies der Ereignisname. Dieses entspricht der Anforderung von Punkt 3 aus dem vorherigen Abschnitt.

Die Verknüpfung von Nachrichten ist ein Problem, da der MessageMapper-Applikation der Aufbau einer Nachricht nicht bekannt ist. Deshalb muß für jede Nachricht, die verknüpft wird, eine Liste mit den Parametern (vgl. Abschnitt 6.3) zur Verfügung stehen.

Zur Zeit benutzen alle Clients dasselbe initiale Set an Verknüpfungen. Gerade im Hinblick auf zukünftige Erweiterungen möchte man aber die Möglichkeit haben, daß verschiedene Clients ihr jeweils eigenes Set besitzen. Dabei muß berücksichtigt werden, ob der MessageMapper beim jeweiligen Client ausschließlich auf dessen Set zugreift oder zusätzlich das allgemeingültige Initialset berücksichtigen darf. Für den aktuellen Stand sollte der letzte Fall ausreichend sein, wobei allerdings festgelegt wird, daß clientspezifische Sets eine höhere Präzedenz vor dem Initialset bekommen.

Durch die Erweiterung des Parsers auf mehrere Sets von Nachrichtenverknüpfungen besteht die Gefahr von Überschneidungen, wenn zwei Clients die gleiche Nachricht mit jeweils einer anderen verknüpfen. Eine logische Trennung der einzelnen Sets ist deshalb erforderlich.

7.2.2 Erzeugen von Ereignissen

Die Fähigkeit, Ereignisse und Gruppen selbst zu definieren, muß in der Scriptingsprache verfügbar werden, damit Clients davon Gebrauch machen können. Der MessageMapper liest die Definitionen von Ereignissen und Gruppen ein und setzt die entsprechenden Nachrichten ab, die den ROMAN veranlassen, sie zu erzeugen. Sobald dem Client mitgeteilt wird, daß diese vorhanden sind, kann er sie abonnieren.

Zu beachten ist, daß normalerweise jedes Ereignis einer Gruppe zugeordnet ist. Es stellt sich für spätere Erweiterungen die Frage, ob die Notwendigkeit dazu besteht. Auf der anderen Seite läßt sich ein einzelnes Ereignis auch als einziger Vertreter seiner Gruppe ansehen. In dieser Arbeit werde ich die Möglichkeit bereit stellen, Ereignisse zu definieren, die keiner Gruppe zugeordnet sind.

7.2.3 Fallunterscheidungen

Die Möglichkeit, einzelne Parameter einer Nachricht zu prüfen, wird in Form von Vorbedingungen in die Scriptingsprache eingefügt. Die Parameterwerte innerhalb einer Nachricht, für die eine Verknüpfung gesucht wird, müssen vorher aus der Nachricht extrahiert werden.

7.2.4 Definition von Variablen

Mit dem Einbringen von Variablen in die Scriptingsprache ist es sinnvoll, die Parameter der Nachrichten ebenfalls als Variablen anzusehen. Damit können sie gleich behandelt werden, wenn auf ihnen gelesen bzw. geschrieben werden soll. Das Auslesen einer Variable geschieht in den Vorbedingungen der Nachrichtenverknüpfung. Zusätzlich werden Nachbedingungen in die Sprache integriert, mit denen sich Variablen modifizieren lassen. Variablen, die zugleich Nachrichtenparameter sind, lassen sich damit verändern, bevor sie mit der Nachricht verschickt werden.

7.2.5 Die Scriptingsprache

Ausgehend von den letzten vier Abschnitten, fasse ich noch einmal die benötigten Elemente meiner Scriptingsprache zusammen:

1. Deklaration von Variablen;
2. Definition neuer Nachrichten und zugehöriger Parameter, Ereignisse und Ereignisgruppen;
3. Erzeugung von Nachrichtenverknüpfungen mit Vor- und Nachbedingungen.

Dabei habe ich die Elemente auch schon in eine notwendige Reihenfolge gebracht. Es können weder Nachrichtenparameter definiert, noch Bedingungen eingesetzt werden ohne vorherige Variablendeklaration. Außerdem müssen die Nachrichten zuerst definiert werden, damit sie in Verknüpfungen benutzt werden können. Zuletzt werden Scripte übersichtlicher, wenn sämtliche Definitionen in einem Block konzentriert sind.

7.2.6 Der Scriptinginterpreter

In Abbildung 7.1 sind die logischen Bestandteile zu sehen, aus denen der MessageMapper bestehen wird. Die soliden Komponenten innerhalb des MessageMappers existieren zwar bereits; sie müssen allerdings angepaßt werden. Die gestrichelt dargestellten Komponenten müssen gänzlich neu entwickelt werden. Die einzelnen Komponenten werden im Folgenden genauer betrachtet.

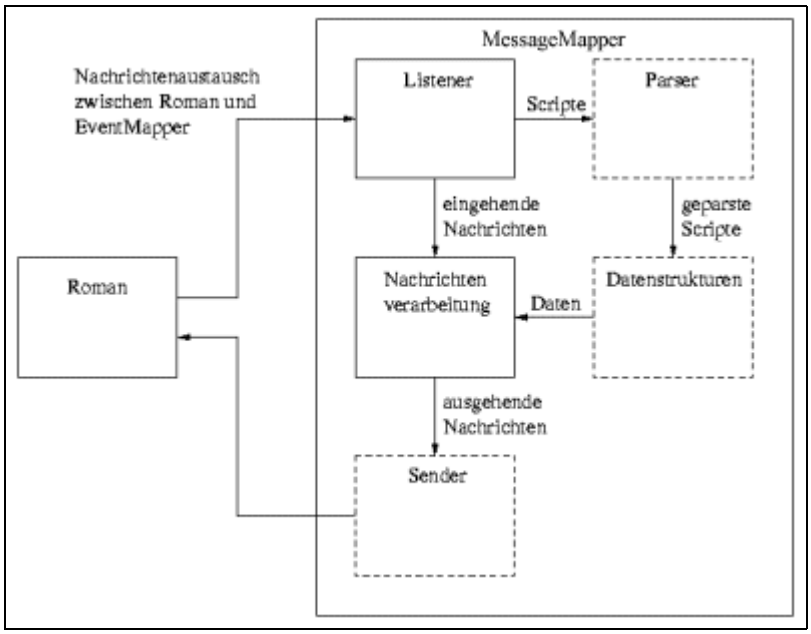

Abbildung 7.1: Grobstruktur des EventMappers

7.2.6.1 Listener

Seine ursprüngliche Aufgabe bestand darin, auf ankommende Nachrichten zu warten. Seine Funktionalität wird nun dahingehend erweitert, daß er von Clients verschickte Scripte (in Form von Nachrichten) an den Parser weiterleitet.

7.2.6.2 Parser

Zum Programmstart wird hier das Initialscript geparst. Kommen später über den Listener weitere Scripte an, so werden sie hier ebenfalls geparst. Die extrahierten Daten werden in die Datenstrukturen eingefügt.

7.2.6.3 Datenstrukturen

In ihnen werden die Daten der geparsten Scripte abgelegt. Die Nachrichtenverarbeitung greift auf sie zu.

7.2.6.4 Nachrichtenverarbeitung

Die Nachrichtenverarbeitung erhält Zugriffsmöglichkeiten auf die Datenstrukturen, um bei Bedarf auf die geparsten Daten zugreifen zu können.

7.2.6.5 Sender

Der Sender erhält ebenfalls Veränderungen. Er verschickt nun beliebige Nachrichten, die in der Komponente Nachrichtenverarbeitung zusammengestellt wurden, statt wie bisher nur die Nachricht vom Typ `RAISE EVENT`.

7.3 Feinspezifikation

Eine vollständige Spezifikation der Klassen befindet sich in Anhang A. Hier werden nur die wichtigsten Punkte besprochen. Ich beginne mit der Syntax der Scriptingsprache. Die Reihenfolge der Sprachenelemente orientiert sich dabei am logischen Aufbau der Scripte (siehe Abschnitt 7.2.5).

7.3.1 Variablen

Variablen werden mit dem Befehl `var` deklariert bzw. definiert:

```
var NAME, TEXT;
var TRUE = 1, FALSE = 0;
```

Eine Definition ist nicht erforderlich. Handelt es sich bei einer Variable um einen Parameter, wird sie mit der ersten Nachricht, die ihn enthält, automatisch mit einem Wert beschrieben. Solange eine Variable nur deklariert ist, ist sie typenlos.

7.3.2 Nachrichten

Nachrichten und ihre Parameter lassen sich mit `define message` bzw. `define parameter` erzeugen:

```
define message {
      define parameter NAME;
      define parameter TEXT;
} RAISE~EVENT;
```

Der Aufbau der Nachricht muß dem Sender (siehe 7.2.6.5) bekannt sein, damit er in der Lage ist, die Nachricht mit den richtigen Parametern zu versehen.

Die Tilde in `RAISE~EVENT` dient als Platzhalter für das Leerzeichen (*space*). Der Umstand, daß einige Nachrichten aus mehreren Worten bestehen, machte diese Einführung notwendig.

7.3.3 Ereignisse und Ereignisgruppen

Um ein Event zu definieren existiert der Befehl `define event`:

```
define event SIM_STATUS;
```

Entsprechend werden Eventgruppen mit `define group` erzeugt:

```
define group {
      define event SIM_START;
      define event SIM_STOP;
} SIMULATION;
```

7.3.4 Nachrichtenverknüpfungen

Vollständige Verknüpfungen sehen folgendermaßen aus:

```
associate <Nachrichtentyp_0> with
      { <Vorbedingung_1> } <Nachrichtentyp_1> { <Nachbedingung_1> },
      ...
      { <Vorbedingung_n> } <Nachrichtentyp_n> { <Nachbedingung_n> };
```

Beispiel:

```
associate RAISE~EVENT with
      { (NAME == "SIM_START") }          // Vorbedingung
            RAISE~EVENT                  // verknüpftes Ereignis
      { (NAME="SIM_STATUS"; TEXT=TRUE;}, // Nachbedingung

      { (NAME == "SIM_STOP") }           // Vorbedingung
            RAISE~EVENT                  // verknüpftes Ereignis
      { (NAME="SIM_STATUS"; TEXT=FALSE;}; // Nachbedingung
```

Beide Bedingungsblöcke einer vollständigen Verknüpfung können weggelassen werden. So ergibt sich bei Auslassung der Vorbedingung eine Verknüpfung, die immer gültig ist, wenn die entsprechende Nachricht vom MessageMapper empfangen wird. Das Weglassen der Nachbedingung bedeutet, daß den Parametern der verknüpften Nachricht keine neuen Werte zugewiesen werden. Passiert dies zu einem Zeitpunkt, wo einige Variablen noch überhaupt keinen Wert besitzen, weil sie weder am Anfang des Scriptes definiert wurden noch von einer eingehenden Nachricht einen Wert zugewiesen bekamen, könnte dieses zu einem Fehler führen. In Kapitel 8 werde ich auf die Fehlertoleranz des fertigen Parsers noch genauer zu sprechen kommen.

7.3.5 Die Scriptingsprache

Aus Kapitel 3 (speziell Abschnitt 3.4.4.2) schließe ich, daß eine prädikatenbehaftete LL-Grammatik die beste Alternative für mich ist, um die Sprache zu entwickeln. Der Parser der Sprache wird mit dem LL(k)-Compiler-Compiler PCCTS[5] in der Version 1.33MR22 entwickelt.

7.3.6 Der Scriptinginterpreter

Entsprechend Abschnitt 7.2.6 bleibt die Struktur des ursprünglichen EventMappers weitestgehend erhalten. Einzige Änderung wird die Integration der neuen Komponenten sein.

7.3.6.1 Parser

Der Parser wird mittels der Werkzeuge aus PCCTS erzeugt. Der Kompiliervorgang besteht aus zwei Teilen. Im ersten Schritt werden aus der Grammatik *grammar.g* mittels des Parsers ANTLR folgende Dateien erzeugt:

- tokens.h – Eine Aufzählung aller terminalen Symbole (vgl. 3.2.1)
- Parser.cpp/Parser.h – In diesen Dateien sind die Funktionen und Variablen zu finden, die direkt im Parser definiert wurden. Im konkreten Fall befinden sich hier die Zugriffs-funktionen auf die Datenstrukturen.
- grammar.cpp – Im eigentlichen Parser werden die Produktionsregeln der nichtterminalen Symbole abgelegt.

[5] Kostenloses Herunterladen der neuesten Version ist unter www.antlr.org möglich.

- parser.dlg

Mit dem Scannergenerator DLG wird aus der Eingabedatei *parser.dlg* der lexikalische Scanner kompiliert: DLGLexer.cpp/h (vgl. 3.3).

7.3.6.2 Datenstrukturen

Für die Datenstrukturen werden eine Reihe von Containerklassen benötigt:

- MessageClass – In einer Instanz befindet sich der Name einer Nachricht, sowie eine Liste ihrer Parameter.
- VariableClass – Jede Instanz dieser Klasse speichert sämtliche Informationen einer Variable: Namen, Typ und aktueller Wert.
- EventClass – Für jedes Event werden Name und evtl. Gruppenzugehörigkeit in einer Instanz abgelegt.
- GroupClass – Jede Instanz entspricht einer Gruppe. An Informationen besitzt sie allerdings nur ihren Namen.
- Association – Eine Instanz entspricht einem Nachrichtentyp, dem etwas zugeordnet werden soll. Weitere Klassen, auf die Association zugreift, erzeugen einen Baum für die Elemente der Vorbedingung bzw. eine verkettete Liste für die Zuweisungen der Nachbedingung. In der Instanz selber werden der Nachrichtentyp und der Clientname abgelegt.

Weiterhin stellt jede Klasse entsprechende Zugriffsfunktionen zum Schreiben und Lesen ihrer Elemente bereit. Die Klasse *Association* bietet darüber hinaus noch die Möglichkeit, jederzeit den Wahrheitsgehalt der Vorbedingung zu evaluieren.

7.3.7 Fehlertoleranz und Robustheit

Von Hand geschriebene Scripte können Fehler enthalten, die vom Parser abgefangen werden müssen, damit er zumindest einen Teil des Scripts verarbeiten kann bzw. nicht einfach abstürzt. PCCTS bietet hierfür zwei Vorgehensweisen. Die erste ist eine einfache Fehlerbehandlung, mit der man nach dem Auftreten eines Fehlers aufräumen bzw. versuchen kann, den Parser an eine Stelle zu führen, an der er weiter arbeiten kann (erneutes synchronisieren). Die zweite Vorgehensweise besteht im Abfangen von Ausnahmen (siehe Abschnitt 4.2). Diese unterscheiden differenzierter in der Art des Fehlers und erlauben so eine gezielte Behandlung.

Davon abgesehen macht es Sinn, einen Scripttester zu implementieren, den man aufruft, um ein soeben geschriebenes Script auf syntaktische Korrektheit zu prüfen.

7.3.8 Entwicklungsumgebung

Durch die Erweiterung bestehender Software bin ich in der Wahl meiner Mittel eingeschränkt. Der EventMapper wurde auf einer Windowsplattform unter Microsoft Visual C++ 5.0/6.0 implementiert. Für die Benutzung des PCCTS-Compilers existiert eine Unterstützung für Visual C++, so daß es hier kaum zu Schwierigkeiten kommen sollte.

8 Funktionalität des scriptgesteuerten Nachrichtenverteilers

Dieses Kapitel dient dazu, Scriptingsprache und Parser kurz vorzustellen, sowie ihren Einsatz zu demonstrieren. Ich werde eine neue Ist-Analyse durchführen und einige Probleme besprechen, die während der Implementierung und Integration des Parsers zu lösen waren. Abschließen werde ich das Kapitel mit einem Ausblick auf zukünftige Erweiterungsmöglichkeiten.

8.1 Vorstellung der Scriptingsprache

Die Sprachelemente sind schon in den Abschnitten 7.3.1 bis 7.3.4 vorgestellt worden. Hier möchte ich ihre Grammatik vorstellen, die ich in erweiterter Backus-Naur Form (siehe [Gar98]) spezifiziert habe. Der besseren Lesbarkeit wegen wurde nicht explizit darauf geachtet, ob sie LR(1) bzw. LL(1) (vgl. Abschnitt 3.4) ist. Zuvor möchte ich kurz die verwendeten Symbole und Schreibweisen erklären:

- (...)* – Der geklammerte Block kann beliebig häufig hintereinander vorkommen oder überhaupt nicht.
- (...)+ – Der geklammerte Block kann beliebig häufig hintereinander vorkommen, aber mindestens einmal.
- {...} – Der geklammerte Block kommt maximal einmal vor.
- p1 | p2 – Es kommt entweder p1 oder p2 vor (Alternative).
- (z1–z2) – Jedes Ascii-Zeichen von z1 bis einschließlich z2 (Zusammenfassung).
- bla – Ein nichtterminales Symbol.
- **bla** – Ein terminales Symbol.
- **BLA** – Terminale Satzzeichen und ähnliche Symbole werden der besseren Lesbarkeit wegen ausgeschrieben.

```
script          := variables definitions associations
variables       := (var decl (COMMA decl)* SEMICOLON)*
decl            := ID {EQUALS variable}
definitions     := (msgdefinition | groupdefinition |
                   eventdefinition)*
msgdefinition   := define message { LCURLY
                   (paramdefinition)* RCURLY } ID
                   SEMICOLON
paramdefinition := define parameter ID SEMICOLON
groupdefinition := define group LCURLY (eventdefinition)*
                   RCURLY ID SEMICOLON
eventdefinition := define event ID SEMICOLON
associations    := (association)*
```

```
association     := associate ID with assoctoken (COMMA
                   assoctoken)* SEMICOLON
assoctoken      := {expr} ID {assignment}
expr            := LCURLY <logischer Ausdruck> RCURLY
assignment      := LCURLY (ID EQUALS variable SEMICOLON)*
                   RCURLY
variable        := ID | INTEGER | REAL | STRING
```

Die folgenden vier Symbole sind Platzhalter für Gruppen von terminalen Symbolen.

```
ID              := (A-Za-z)(A-Za-z0-9_:?!~)*
INTEGER         := {MINUS}(0-9)+
REAL            := {MINUS}(0-9)*{COLON(0-9)+}
STRING          := DOUBLEQUOTES <Text> DOUBLEQUOTES
```

Es folgt ein Auszug aus dem Script, das während der Implementation entstanden ist. Dieses Script wird vom *MessageMapper* während seiner Initialisierung geladen und verarbeitet. Die Beschreibung der einzelnen Befehle sind der Feinspezifikation (Abschnitt 7.3) zu entnehmen.

```
var NAME, TEXT, CLIENT;
var FALSE = 0, TRUE = 1;
var SIM = FALSE;

define group {
      define event SPEECH_IN;
      define event SPEECH_OUT;
      define event SET_SPEECH_COMMANDS;
} SPEECH;

[…]

define event SIM_START;
define event SIM_STOP;

[…]

associate RAISE_EVENT with
    { (NAME == "SPEECH_IN") && (TEXT == "Start") }
      RAISE_EVENT
    { NAME = "SIM_START"; SIM = TRUE; },
    { (NAME == "SPEECH_IN") && (TEXT == "Stop") }
      RAISE_EVENT
    { NAME = "SIM_STOP"; SIM = FALSE; };

[…]
```

8.2 Beschreibung des Parsers

Der Parser – und damit der MessageMapper – liegt in seiner endgültigen Version vor. Er ist in CLEAR integriert und hat die Aufgaben des alten EventMappers übernommen. Ich habe

darüber hinaus einen Client namens „TrafficLight“ geschrieben, der die Fähigkeiten des MessageMappers anschaulich demonstrieren soll. Er wird ebenfalls in diesem Kapitel vorgestellt werden.

8.2.1 Aufruf und Einsatz des Parsers

Als Bestandteil des früheren EventMappers, wird der Parser automatisch mit dem MessageMapper gestartet. Sofort nach dem Aufruf wird die Scriptdatei *initialscript.txt* geladen, die sich im Programmverzeichnis von CLEAR befinden muß. Erweiternde Scripte können dem MessageMapper über die Nachrichtenschnittstelle von CLEAR zugeschickt werden. Dafür gibt es das Ereignis SEND_SCRIPT, welches als Parameter das Script mit sich führt. Wenn das Script geparst wurde, wird vom MessageMapper ein ACK_SCRIPT gesendet, dessen Parameter der Name des jeweiligen Clients ist, dessen Script zuletzt verarbeitet wurde. Sollte während des Parsens ein Fehler entdeckt worden sein, wird im Parameter TEXT eine entsprechende Fehlermeldung mitgeliefert. Beide Events sind in der Gruppe SCRIPT zusammengefaßt, die jeder Client abonnieren muß. Was zusätzlich auf der Clientseite zu beachten ist, wird exemplarisch am Client TrafficLight in Abschnitt 8.3 gezeigt.

8.2.2 Vergleich mit den Vorgaben

Hier möchte ich einige Punkte diskutieren, die während der Programmierung aufkamen und näher untersucht werden mußten. Außerdem war es notwendig, mit der Fertigstellung des Parser zu prüfen, ob er die Vorgaben aus Abschnitt 7.1 erfüllt.

- Für die Abfrage der Vorbedingung stehen zum jetzigen Zeitpunkt nur die logischen Operatoren && (AND / Konjunktion), || (OR / Disjunktion) und ! (NOT / Negation) zur Verfügung. Dennoch läßt sich mit diesen Operatoren jede andere logische Operation nachbilden (Beweis siehe z.B. [Coy92] S. 37). Beispiele:
  - x NAND y = NOT (x AND y)
  - x XOR y = (x AND (NOT y)) OR ((NOT x) AND y)
- Eine kleine Einschränkung findet sich in dem Umstand, daß nur genau zwei Ausdrücke mit einem logischen Operator verknüpft werden können. Eine Verknüpfung mehrerer Ausdrücke funktioniert nur mit entsprechender Klammerung. Beispiel: (a AND b) AND (x AND y).
- Scriptvariablen können nur STRING und NUMBER sein. Zwar existiert im Parser eine Trennung von Ganz- und Fließkommazahlen, doch wurde das nur der Performanz wegen eingerichtet.
- Scriptvariablen können außerhalb des Parsers gelesen und beschrieben werden. Zur Zeit wird diese Möglichkeit genutzt, um während der Initialisierung des MessageMappers den Inhalt einer Variable in einer Nachricht zu verschicken.
- Die Fehlerbehandlung ist rudimentär (siehe auch folgenden Abschnitt).

- Für eine Scriptingsprache normalerweise untypisch ist die feste Deklaration von Variablen (vgl. Abschnitt 2.3.4). Es existiert zwar keine feste Typisierung, doch müssen sämtliche im Script verwendeten Variablen (wozu auch die Parameter in den Nachrichten-definitionen zählen) am Anfang deklariert werden. Dieses soll dazu dienen, die Übersicht über die benutzten Variablen zu behalten. Durch einen Schreibfehler könnte sonst eine neue Variable erzeugt werden, die nie benutzt wird.
- Es ist bisher nicht möglich eigene Nachrichten zu definieren. Zwar wird dieses durch die Sprache nicht verboten, wird aber keinen Effekt auf den Ablauf haben. Die gültigen Nachrichten werden durch den ROMAN bestimmt. Ohne Änderungen an seinem Code ist man auf die Nachrichten beschränkt, die er zur Verfügung stellt. Siehe dazu auch Abschnitt 8.5.4.
- Benutzte Nachrichten, müssen in jedem Script definiert werden (siehe auch Beispielscript in Abschnitt 8.3.1). Praktisch muß jedes Script, in dem Ereignisse verknüpft werden, die Nachricht RAISE EVENT definiert werden. Solange garantiert werden kann, daß die Nachricht in einem anderen Script bereits vom MessageMapper eingelesen wurde (z.B. initialscript.txt), genügt es, wenn die dazugehörigen Parameter in der Variablen-deklaration am Anfang des Scripts erscheinen. Der MessageMapper filtert doppelte Nachrichtendefinitionen nachträglich heraus.

  Beispiel:

  ```
  var NAME, TEXT;
  define message {} RAISE~EVENT;
  ```

- Neben den bisherigen Startoptionen, die aus dem EventMapper übernommen wurden, wurden dem MessageMapper noch zwei weitere hinzu gefügt:

  1. –scriptfile: Dieser Schalter erlaubt es, ein alternatives Script als *initialscript.txt* anzugeben.
  2. –globalscript: Mit Setzen dieses Schalters bestimmt man, ob das initiale Script auch von denjenigen Clients benutzt werden soll, die ein eigenes Script an den MessageMapper schicken (TRUE) oder diese ausschließlich auf den Inhalt ihres eigenen Scripts zurückgreifen dürfen (FALSE). Per Voreinstellung innerhalb des MessageMappers gilt der erste Fall.

8.2.3 Fehlerbehandlung

Der Parser kann an verschiedenen Stellen in einem zu parsenden Script auf syntaktische Fehler stoßen:

1. Eine oder mehrere Variablen werden nicht definiert.
2. Ein Nachricht wird nicht definiert.
3. Ein Parameter innerhalb einer Nachricht wird nicht definiert.

4. Eine Ereignisgruppe wird nicht definiert.
5. Ein Ereignis wird nicht definiert. Dabei ist es unerheblich, ob es zu einer Gruppe gehört.
6. Eine Nachrichtenverknüpfung bekommt eine fehlerhafte Vor- oder Nachbedingung.
7. Eine Nachrichtenverknüpfung wird überhaupt nicht definiert.

Alle Fehler haben gemeinsam, daß dem Parser Informationen bei seiner Arbeit fehlen, was unweigerlich zu ungewollten Ergebnissen führt. Das folgende Beispielscript verdeutlicht das:

```
var NAME, TEXT, SIM;

define group {
     define evrnt SPEECH_IN;
     define event SPEECH_OUT;
} SPEECH;

associate RAISE_EVENT with
     { (NAME == "SPEECH_IN") && (TEXT == "Start") }
          RAISE_EVENT
     { NAME = "SIM_START"; SIM = TRUE; },
```

Durch die fehlerhafte Definition des Ereignisses SPEECH_IN wird es bei der Initialisierung von CLEAR nicht vom MessageMapper abonniert. Folglich wird ihm das Auftreten auch nicht vom ROMAN gemeldet. Die Verknüpfung, die das Ereignis SIM_START verschicken soll, wird niemals benutzt. Aus diesem Grunde wird beim Eintreten eines Fehlers das Parsen vollständig gestoppt und eine Fehlermeldung zurückgegeben. Befindet sich der Fehler im initialen Script, wird die Meldung direkt auf dem Bildschirm ausgegeben. Ansonsten schickt der MessageMapper sie an den Client, von dem das Script stammt.

Ein anderer Typ von Fehlermeldungen, der mehr als Warnung dient, soll den Scriptautor darauf hinweisen, ob er vergessen hat, eine Variable zu deklarieren. Obwohl das Script dadurch nicht mehr akzeptiert wird, unterbricht ein Auftreten dieses Fehlers den Parservorgang nicht. Es können also unter Umständen noch weitere Fehler entdeckt werden. Ein Beispiel für eine Fehlermeldung zeigt die folgende Abbildung. Hier wurde vergessen, die Variable `SIM` zu deklarieren. Außerdem wurde in Zeile 236 des Scripts ein Leerzeichen in die Variable `NAME` eingefügt.

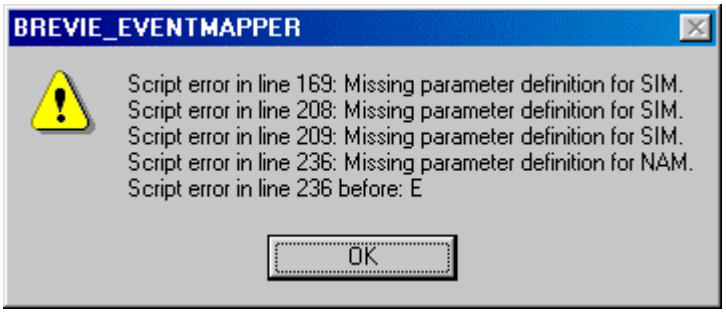

Abbildung 8.1: Typische Fehlermeldung des MessageMappers

8.3 Vorstellung des Clients TrafficLight

Der Client TrafficLight soll dazu dienen, die Arbeitsweise des MessageMappers anschaulich zu demonstrieren.

8.3.1 Initialisierung

Mit dem Start des Client abonniert dieser sofort die Gruppe SCRIPT und schickt dem MessageMapper das folgende Script:

```
var RED = 0, GREEN = 1;
var TRAFFIC_LIGHT_CONDITION = RED;
var NAME, TEXT;

define message {
      define parameter NAME;
      define parameter TEXT;
} RAISE~EVENT;

define group {
      define event GREEN_PHASE_OVER;
      define event BUTTON_PRESSED;
      define event SET_LIGHT;
} CHANGE_TRAFFIC_LIGHT;

define group {
      define event ASK_LIGHT;
      define event GET_LIGHT;
} INFO_TRAFFIC_LIGHT;

associate RAISE~EVENT with

 { (NAME == "SET_LIGHT" && TEXT == "GREEN") }
      RAISE~EVENT
 { NAME="GET_LIGHT"; TEXT="GREEN"; TRAFFIC_LIGHT_CONDITION=GREEN; },

 { (NAME == "SET_LIGHT" && TEXT == "RED") }
      RAISE~EVENT
 { NAME="GET_LIGHT"; TEXT="RED"; TRAFFIC_LIGHT_CONDITION = RED; },

 { ((NAME == "BUTTON_PRESSED")&&(TRAFFIC_LIGHT_CONDITION != GREEN)) }
      RAISE~EVENT
 { NAME="GET_LIGHT"; TEXT="GREEN"; TRAFFIC_LIGHT_CONDITION=GREEN; },

 { (NAME == "GREEN_PHASE_OVER") }
      RAISE~EVENT
 { NAME="GET_LIGHT"; TEXT="RED"; TRAFFIC_LIGHT_CONDITION=RED; },

 { (NAME == "ASK_LIGHT") }
      RAISE~EVENT
 { NAME="GET_LIGHT"; TEXT=TRAFFIC_LIGHT_CONDITION; }
;
```

Durch das vorhergehende Abonnement bekommt der Client nun mit, ob das Script erfolgreich verarbeitet wurde und kann seine Initialisierung fortsetzen, indem es die beiden Eventgruppen abonniert. Damit ist der Client bereit.

8.3.2 Arbeitsweise

Wie aus der Abbildung ersichtlich, besitzt er neben den zwei Ampelfeldern zwei Knöpfe. Der ‚Press'-Knopf dient dazu, die Ampel zu aktivieren. Normalerweise befindet sie sich in einer kontinuierlichen Rotphase. Durch Aktivierung wird eine Nachricht vom Typ RAISE EVENT mit Inhalt BUTTON_PRESSED an den ROMAN geschickt. Der MessageMapper sendet daraufhin die Bestätigung mit der neuen Farbe (GET_LIGHT).

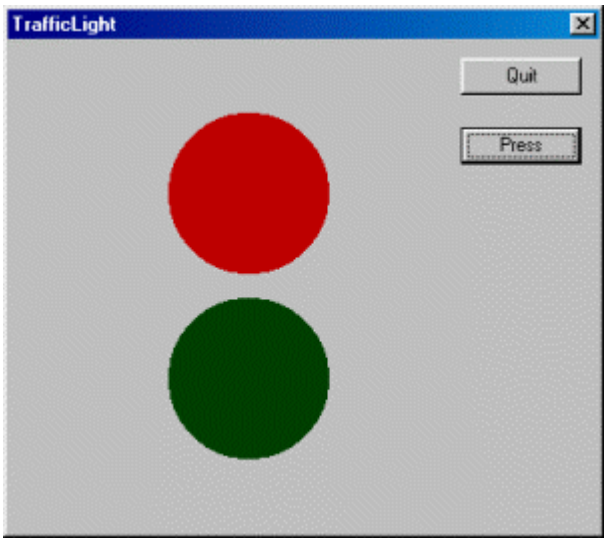

Abbildung 8.2: TrafficLight

Die Ampel verbleibt für eine gewisse Zeit in der Grünphase. Weiteres Drücken des ‚Press'-Buttons verlängert diese Phase allerdings nicht. Nach Ablauf der Zeit wird ROMAN das Event GREEN_PHASE_OVER vom TrafficLight-Client übermittelt, auf welches der MessageMapper mit SET_RED antwortet. Der Button ‚Press' kann nun erneut betätigt werden. Mit dem ‚Quit'-Button wird das Programm beendet.

8.4 Abnahme

Anhand des Beispiels im vorangegangenen Abschnitt kann man sehen, daß uns die Scripting-schnittstelle eine Vielzahl neuer Möglichkeiten beschert. Die Ampelsteuerung ist nur ein typisches Beispiel dafür. Mit folgender Verknüpfung etwa hätte man schon den Grundstein für einen Chatclient gelegt:

```
associate RAISE~EVENT with

     { (NAME == "SEND_MSG") }
          RAISE~EVENT
     { NAME = "RECEIVE_MSG"; };
```

`SEND_MSG` und `RECEIVE_MSG` sind Ereignisse zweier Ereignisgruppen. Ein Chatclient abonniert `RECEIVE_MSG` und verschickt eigene Beiträge mit `SEND_MSG`.

Mit der Möglichkeit Variablen zu definieren, sind wir in der Lage, bestimmte Zustände zu sichern. Damit lassen sich Schalter einsetzen, die in den Vorbedingungen geprüft werden können.

```
associate RAISE~EVENT with
     { (NAME == "SIM_START") }
          RAISE~EVENT
     { NAME = "SIM_CONDITION"; NAME = TRUE; SIM = TRUE; },

     { (NAME == "SIM_STEP" && SIM == TRUE) }
          RAISE~EVENT
     { NAME = "SIM_CONDITION"; };
```

Eine große Schwäche des alten EventMappers bestand darin, daß er nur Nachrichten vom Typ RAISE EVENT versenden konnte. Die zu verschickenden Informationen mußten in den Feldern `NAME` und `TEXT` untergebracht werden. Mit dieser Erweiterung, in der nun alle Nachrichten, die dem ROMAN bekannt sind, verknüpft werden können, existiert jetzt eine viel größere Auswahl an Möglichkeiten. In diesem Beispiel sind die Nachrichten `RAISE EVENT` (Ereignis `DELETE_WORLD`) und `REPLACE WORLD` miteinander verknüpft, um im ROMAN eine leere Szene zu erzeugen.

```
associate RAISE~EVENT with
     { (NAME == "DELETE_WORLD") && SIM==FALSE }
          REPLACE~WORLD
     { DEFINITION = ""; }; // Erzeugung einer leeren Szene
```

8.5 Möglichkeiten der Erweiterung

Die Vorgaben wurden zwar erfüllt, dennoch bietet der MessageMapper weitere Möglichkeiten für Erweiterungen und Modifikationen an. Ich weise allerdings darauf hin, daß einige dieser Erweiterungen über den Kontext des MessageMappers hinaus gehen. Das erklärt, warum ich sie als zukünftige Erweiterungsmöglichkeiten ansehe, da sie nicht mehr direkt in den Bereich fallen, den ich in dieser Diplomarbeit abgesteckt habe.

8.5.1 Weitere Variablen

Die Struktur der Variablen innerhalb des MessageMapper ist recht einfach gehalten. Weitere Variablen machen deshalb erst Sinn, wenn die Klasse *VariableClass* (siehe Anhang A) erweitert wird. Durch Ableitung lassen sich Unterklassen bilden, die unterschiedliche Basistypen repräsentieren. In diese Unterklassen lassen sich dann spezialisierte Funktionen (Abfragen, Typkonversionen, etc.) einfügen.

8.5.2 Weitere logische Operatoren in der Vorbedingung

Mit den Operatoren AND, OR und NOT lassen sich zwar schon alle Fälle behandeln, jedoch geht dieses teilweise auf Kosten der Lesbarkeit in den Scripten. Daraus können semantische

Fehler resultieren, die vom Parser nicht entdeckt werden und sich erst zur Laufzeit bemerkbar machen. Der Einbau weiterer Operatoren würde dem entgegenwirken.

8.5.3 Erweiterung der Funktionalität

Die Sprache allgemein kann um weitere Funktionen erweitert werden. Eine vorhergehende Analyse ist dafür nötig, um zu klären was für Erweiterungen sinnvoll sind. Denkbar sind Schleifen und Fallunterscheidungen (als Erweiterung der einfachen Vor- und Nachbedingungen).

8.5.4 Angeben eigener Nachrichten

Mit dieser Erweiterung werden Clients in die Lage versetzt, eigene Nachrichten zu definieren. Dieses bedeutet allerdings eine Erweiterung des ROMAN, der alle Nachrichten verwaltet. In der Scriptingsprache sind entsprechende Befehle zum Definieren von Nachrichten und den dazugehörigen Parametern schon vorhanden, die auch ohne weitere Modifikation benutzt werden können.

8.5.5 Entfernung von Redundanten Informationen

Zum jetzigen Zeitpunkt wird für jedes neue Script eine neue Instanz des Parsers erzeugt. Reduntante Informationen können erst nachträglich heraus gefiltert werden (siehe beispielsweise vorletzten Eintrag in Abschnitt 8.2.2). Würde man den Parser dahingehend erweitern, daß er schon beim Parsen erkennt, ob Informationen doppelt vorhanden sind, könnten diese sofort heraus gefiltert werden. Allerdings entsteht hierbei das Problem, daß in unterschiedlichen Scripten Variablen bzw. Ereignisse gleichen Namens auftreten könnten, aber jeweils in einem anderen Kontext stehen. Die Scriptingsprache müßte so erweitert werden, daß sich Informationen nach globalen und kontextspezifischen (nur zum Script gehörig) trennen lassen.

Auf der anderen Seite gehen durch diese Erweiterung viele Informationen im Script verloren. In einem jetzigen Script kann man schnell einsehen, welche Variablen, Ereignisse und Nachrichten vorkommen.

8.5.6 Fehlerminimierung beim Schreiben der Scripte

Für dieses Problem bieten sich zwei Lösungen an. Die erste wäre, einen Syntaxchecker zu implementieren, den man die geschrieben Scripte parsen läßt. Mit entsprechend hilfreichen Fehlermeldungen sollte ein Script leicht korrigiert werden können. Der Checker kann diese Aufgabe allerdings ebenfalls übernehmen. Der Vorteil dieser Lösung besteht darin, daß im Grunde derselbe Parser benutzt werden kann, der im MessageMapper zum Einsatz kommt.

Eine aufwendigere Lösung besteht darin, einen visuellen Scriptingeditor (siehe Abschnitt 2.5.1) zu schreiben. Der Vorteil hier ist, daß man zumindest keine syntaktischen Fehler mehr verursachen kann. Somit können auch von reinen Anwendern leicht Scripte erzeugt werden; allerdings sollte die Einarbeitung in den Editor leichter sein, als das Erlernen der Sprache.

9 Konzeptuelles Vorgehen zur Integration von Scriptingsprachen

In diesem Kapitel möchte ich darlegen, wie meiner Meinung nach ein Entwickler vorgehen sollte, wenn er eine Scriptingsprache in ein bestehendes System integrieren möchte. Ich werde mich dabei vornehmlich auf meine Erkenntnisse aus den Kapiteln 7 und 8, sowie den Fakten aus Kapitel 2 stützen. Die konzeptionelle Vorgehensweise wird dabei anhand von Beispielen unterstützt, die sich schwerpunktmäßig mit HLA und Jini befassen.

9.1 Ist-Analyse

Im Normalfall ist davon auszugehen, daß ein Entwickler genau weiß, an welcher Stelle er ein Programm bzw. eine Komponente mit einer Scriptingsprache erweitern möchte. Im konkreten Beispiel soll es um das Problem der Integration eines neuen Clients in ein etabliertes System gehen und ob sich dieses mittels Scripting vereinfachen läßt. Der Aufwand wird davon bestimmt, wie leicht sich ein typischer Client anpassen läßt und wie entgegenkommend das System ist, in das der Client integriert werden soll.

HLA und Jini ist gemeinsam, daß sie eine Anpassung der Schnittstelle des Clients erfordern, so daß er in der Lage ist, Nachrichten so aufzubauen, wie sie von der Architektur vorgegeben sind. Die Schnittstelle von HLA erweist sich in den meisten Fällen als aufwendig zu implementieren, da sie viele Anforderungen stellt, die in Form von Funktionsaufrufen erbracht werden müssen. Desweiteren benötigen HLA-*Federates* zwingend eine objektorientierte Darstellung der Modelle in Form der SOM. Ein anzupassender Client muß im Prinzip neu spezifiziert werden, um die SOM aufzubauen. Sobald die SOM steht, müssen daraus die Funktionen abgeleitet werden, die später vom RTI aus zur Verfügung stehen. In [YSB98] wird auf theoretischer Ebene diskutiert, was man als Entwickler alles zu beachten hat, wenn man eine Simulation für HLA anpassen will. Die Probleme die dabei auftreten können, lassen sich exemplarisch in [SK98] betrachten. Dort beschreiben S. Straßburger und U. Klein ihre Vorgehensweise bei der Einbindung des Simulators SLX in eine RTI. Ihre Lösung fußt auf dem Entwickeln einer DLL, die SLX Datentypen und Kommandos in eine dem RTI verständliche Form bzw. umgekehrt konvertiert. Der Konverter kommt dabei auf eine Größe von etwa 5000 Codezeilen. Zwar schreiben die Autoren, daß sie nun ein allgemeingültiges Konzept parat hätten, um „einen Simulator über eine Zwischen- bzw. Wrapper-Bibliothek HLA-fähig zu machen“, weisen aber gleichzeitig darauf hin, daß bei den Konverterfunktionen der Bibliothek noch Anpassungen an das entsprechende Simulationswerkzeug vorgenommen werden müssen. Überhaupt läßt sich diese Lösung nur dann benutzen, wenn die interne Datenrepräsentation des Simulationswerkzeugs bekannt ist.
Im Gegensatz zu HLA, wo der Entwickler in der Wahl seiner Programmiersprache ungebunden ist, verlangt Jini von jedem Client, daß er in Java-Bytecode vorhanden ist oder zumindest konvertiert werden kann, da seine Schnittstellen vollständig über RMI angesprochen werden. Muß man auf eine andere Sprache ausweichen, etwa bei einem *Embedded*

System mit geringem Speicherplatz, bleibt einem hier nur ein Umweg über eine Javaschnittstelle, die auf einer weiteren Plattform die Kommunikation zwischen Client und *Lookup Services* steuert. In den *Lookup Services* liegt wiederum die Stärke von Jini, da sie sich beliebig erweitern lassen, um so Clients mit neuen Funktionalitäten anderen zugänglich zu machen. Wenn allerdings kein passender LS vorhanden ist, muß ein neuer implementiert und in die Infrastruktur von Jini integriert werden, bevor der neue *Service* zugänglich wird.

HLA und Jini haben beide den Vorteil, daß sie in der Lage sind, verschiedene abstrakte Datentypen direkt zu versenden und sich nicht darauf beschränken müssen, diese vor dem Versenden via Nachricht in ihre Bestandteile zu zerlegen. Probleme können auftreten, wenn gewisse Basistypen nicht vorhanden sind und deswegen substituiert werden müssen (z.B. einen Boolean als Ganzahl oder String darstellen).

Ein ähnliches Problem, das in [SK98] angerissen wird, ist die korrekte Interpretation der Basistypen bzw. eine übereinstimmende Bytegröße. Zum Beispiel kennt HLA für Gleitkommazahlen nur einen Float mit einer Größe von 4 Byte.

Aus diesen Vorgaben lassen sich nun zwei Ansätze formulieren. In HLA soll eine Wrapperbibliothek ähnlich der aus [SK98] entwickelt werden, die mittels Scripting dem Anwender einen Teil der notwendigen Schnittstellenanpassungen vereinfacht. Damit soll es ermöglicht werden, auch fremde Simulatoren in HLA einzufügen.

In Jini soll eine Scriptingsprache entwickelt werden, die in einen LS integriert werden kann, um so eine variable Schnittstelle zu bieten. Damit erhalten wir die Möglichkeit, Änderungen an den Schnittstellen des Service vorzunehmen und nur das dazugehörige Script ändern zu müssen. Der LS wird automatisch aktualisiert. Wir nehmen hier allerdings den Kompromiß in Kauf, daß der Client in Java vorliegt.

9.2 Anforderungsdefinition der Scriptingsprache

Mit der Definition der Sprache wird festgelegt, was sie alles an Funktionen erfüllen muß. Die HLA Wrapperbibliothek hat folgende Aufgaben:

- Datentypenkonversion – Funktionen, die eine Variable als einen anderen Datentyp zurückgeben (z.B. Double zu Float). Wegen der möglichen Typenbeschränkungen, kommt man nicht umhin, auch seltenere Konverterfunktionen zu schreiben (z.B. Integer mit 16 und 32 Bit Feldgröße).
- Funktions-Abbildungen – Schnittstellenfunktionen zum RMI bzw. zum jeweiligen Simulator.
- Maßanpassungen – Einfache Konverterfunktionen, die z.B. Längenmaße (inch auf cm, dm auf m) umrechnen.

Die Scriptingsprache in Jini soll die Funktion erfüllen, dem LS via Script mitzuteilen, über welche Schnittstellen ein bestimmter *Service* verfügt. Der Interpreter muß daraus dann den entsprechenden Programmcode erzeugen um den LS zu erweitern.

9.3 Aufwandsbestimmung

Die Frage nach dem Aufwand ist nicht zu unterschätzen. Bei allen Vorteilen, die ein gescriptetes System mit sich bringen mag, muß man sich stets die Frage stellen, ob sich der Aufwand lohnt, den man dafür in die Entwicklung steckt.

Die Erweiterung des ROMAN EventMappers zum MessageMapper ging verhältnismäßig schnell. Der Code war offen und bot die entsprechenden Stellen, an denen er sich erweitern ließ. Desweiteren waren Anforderungen und Ziel klar umrissen. Dennoch sind trotz aller Vorteile manche Lösungen schneller und effizienter im Quellcode untergebracht.

Den HLA-Wrapper zu implementieren bedeutet in der Tat einen enormen Aufwand. Für jeden neuen Simulator, der mit dem Wrapper kombiniert werden soll, müssen die entsprechenden Schnittstellen vorhanden sein. Anders ausgedrückt bedeutet dies, daß der Entwickler jedesmal den Quellcode erweitern muß, solange der Simulator nicht eine bereits integrierte Schnittstelle unterstützt.

Auch die Probleme in Jini sind nicht zu unterschätzen. Es muß eine Scriptingsprache entwickelt werden, die beliebige Anpassungen zur Laufzeit vornehmen kann. Eine solch komplexe Sprache zu definieren, bedeutet einen derartigen Aufwand, der in diesem Zusammenhang normalerweise nicht gerechtfertigt werden kann.

Wir haben es in beiden Fällen mit unterschiedlichen Problemen zu tun. In Jini bedeutet die Entwicklung der Scriptingsprache einen enormen Arbeitsaufwand, während in HLA viel Arbeit neben der Entwicklung der Sprache vorgenommen werden muß.

9.4 Entwicklung und Integration der Sprache

Entscheiden wir uns, trotz der schlechten Ergebnisse aus dem letzten Abschnitt, die jeweiligen Sprachen zu entwickeln, sollte folgendes berücksichtigt werden. Die Grammatik der Sprachen sollte vornehmlich in BNF (siehe Abschnitt 3.2.1) niedergeschrieben werden. Das ermöglicht Anpassungen an LALR- bzw. LL-Grammatiken (Abschnitte 3.4.2 und 3.4.3) schon in der Theorie vorzunehmen, bevor man beginnt, die Grammatik in einer für den Parser verständlichen Form aufzuschreiben.

In HLA haben wir es mit einer Bindesprache zu tun, wie wir sie bereits aus Abschnitt 2.2 kennen. Ihre Aufgabe besteht darin, es dem Scriptprogrammierer zu ermöglichen, die Ausgaben von Funktionen als Eingaben anderer Funktionen zu verwenden. Die Funktionen von der RTI-Seite aus werden voraussichtlich nur einmal implementiert werden müssen und können dann benutzt werden. Auf der Seite der zu integrierenden Simulatoren haben wir

schon in Abschnitt 9.3 erkannt, daß hier für jeden neuen Simulator die Schnittstellen u.U. eingefügt werden müssen.

Für Jini bietet sich möglicherweise eine einfache Lösung an, die den erwarteten Aufwand erheblich minimiert. Aus Abschnitt 2.4.7 kennen wir die Scriptingsprache Iava, die es ermöglicht, neue Instanzen bestehender Klassen zur Laufzeit aufzurufen. Es muß geprüft werden, ob diese Sprache den Anforderungen genügt. Wenn die Klassen eines *Services* als Java-Package abgelegt sind, reicht es, einen *Lookup Service* mit einer Iava-Schnittstelle zu versehen. Über ein ankommendes Skript muß ihm mitgeteilt werden, wo die Packages liegen, so daß der Scriptinginterpreter von Iava die Instanzen erzeugen und in den LS integrieren kann.

9.5 Fazit

Wir haben exemplarisch gesehen, auf welche unterschiedlichen Gegebenheiten man als Entwickler achten muß, wenn man eine Scriptingsprache in eine Applikation integrieren möchte. Auf der einen Seite übersieht man leicht, daß der Aufwand schnell den Nutzen übersteigt, auf der anderen Seite kann für ein gegebenes Problem schon eine Lösung existieren. Des weiteren bietet eine einmal integrierte Sprache eine Basis für zukünftige Erweiterungen, die sich nun einfacher einbauen lassen sollten. Darüber hinaus darf man nicht vergessen, daß eine gescriptete Applikation eine Versalität besitzt, die weder durch zusätzlichen Programmcode noch durch Einbau von Dialogen erreicht werden kann.

10 Schlußbemerkung

Scriptingsprachen sind aus heutigen Applikationen nicht mehr wegzudenken. Zwar hat ihr Siegeszug relativ spät begonnen, doch hat diese lange Zeit ihnen nur zum Vorteil gereicht, da sie nun in vielfältigen Bereichen eingesetzt werden. Mit der vorliegende Diplomarbeit hatte ich mir das Ziel gesetzt, Vorteile und Einsatzmöglichkeiten der Scriptingsprachen zu verdeutlichen. Ich habe einerseits versucht, ihre Verwandtschaft, andererseits ihre Unabhängigkeit zu der Familie der Programmiersprachen deutlich zu machen (Kapitel 2 und 3). An der Erweiterung des EventMappers zum MessageMappers zeigte ich, wie sich mit wenig Aufwand viel erreichen läßt (Kapitel 7 und 8). Eine allgemeine Vorgehensweise zum Integrieren einer Scriptingsprache bzw. was man als Entwickler dennoch beachten muß und ob der Aufwand daß Ergebnis lohnt, wurde in Kapitel 9 vermittelt. Damit, so hoffe ich, sollte anderen Entwicklern genügend Inspiration vermittelt worden sein, um sich selbst in die Welt der Scriptingsprachen zu begeben.

Abschließend möchte ich noch sagen, daß ich völlig mit John K. Ousterhouts Hoffnung (in [Ous98]) übereinstimme, daß Programmierer in Zukunft verstärkt darauf achten werden, ob für ein gegebenes Problem eine Programmiersprache mit starker Typisierung und schneller Ausführungsgeschwindigkeit, oder eine Scriptingsprache mit höherer Produktivität und Wiederverwendbarkeit in Frage kommt.

Anhang A: Programmdokumentation

Die Klassen sind in der Spezifikationssprache UML [EP98] dargestellt.

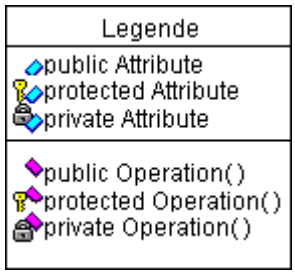

Abbildung A.1: Legende

Containerklassen

Diese Klassen zeichnen sich dadurch aus, daß jede Instanz für ein logisches Element innerhalb des Parsers steht.

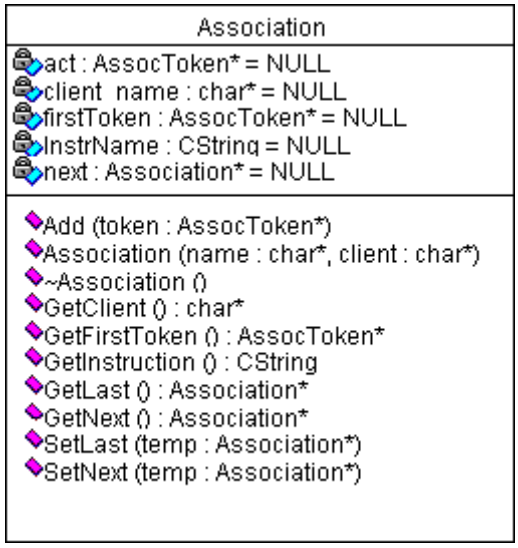

Abbildung A.2: Klasse Association

Die Klasse `Association` stellt die nötigen Operationen zur Verfügung, um eine Nachricht auf mögliche Verknüpfungen zu prüfen. Verknüpfungen sind in Instanzen von `AssocToken` abgelegt. Der Name der Nachricht befindet sich im Attribut `InstrName`.

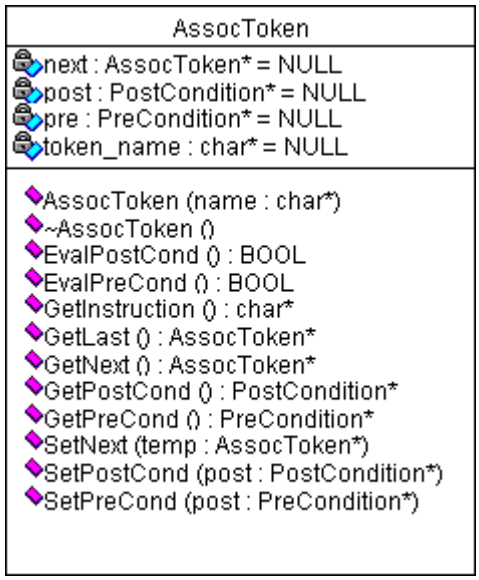

Abbildung A.3: Klasse AssocToken

Diese Klasse ist das Aggregat für die jeweilige Vor- und Nachbedingung einer Verknüpfung. Ihre Operationen erlauben das Setzen, Abfragen und Evaluieren die Bedingungen.

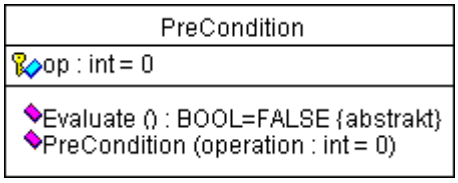

Abbildung A.4: Klasse PreCondition

Es handelt sich hierbei um eine abstrakte Metaklasse. Ihre Unterklassen dienen dazu, zwischen logischen und relationalen Operationen zu unterscheiden, sowie einen binären Baum aufzubauen, dessen Wahrheitswert bei einer Evaluation von den Blättern aus berechnet wird.

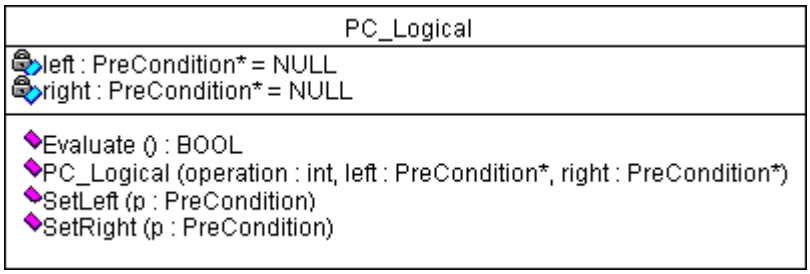

Abbildung A.5: Klasse PC_Logical

`PC_Logical` regelt die Evaluation der logischen Operationen AND, OR und NOT mit maximal zwei Ausdrücken (`left` und `right`).

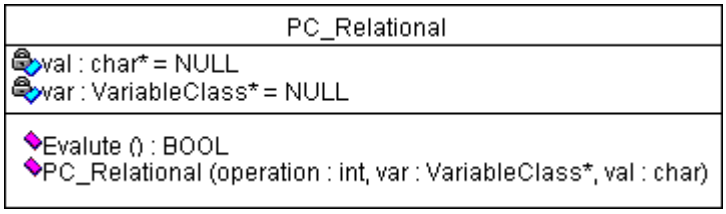

Abbildung A.6: Klasse PC_Relational

Die Klasse `PC_Relational` wird für die relationalen Operationen <, <=, ==, !=, => und > benutzt.

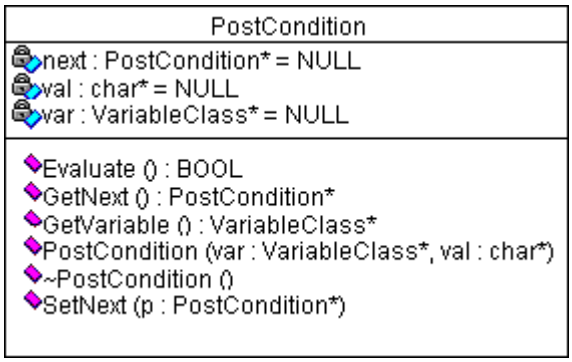

Abbildung A.7: Klasse PostCondition

Mit dieser Klasse werden die Zuweisungen einer Nachbedingung realisiert. Mehrere Zuweisungen bilden eine einfach verkettete Liste, die ihren Ursprung in der `AssocToken`-Instanz hat.

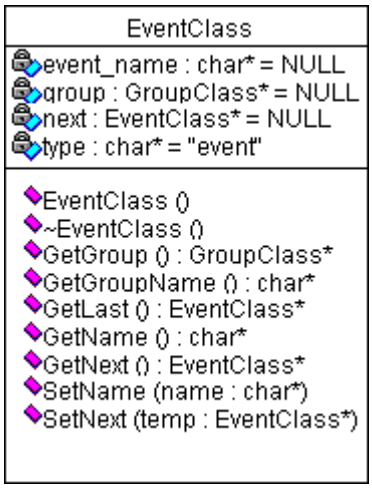

Abbildung A.8: Klasse EventClass

Jede Instanz stellt den Namen eines Ereignisses bereit. Ist es Mitglied einer Gruppe, lassen sich mit den gegebenen Funktionen auch Informationen über diese erfragen.

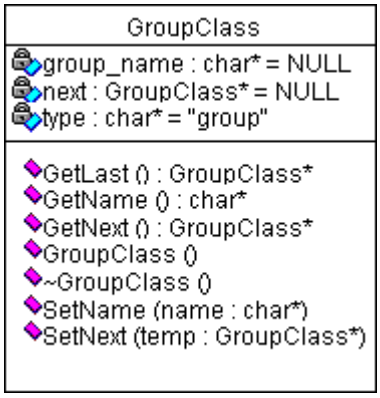

Abbildung A.9: Klasse GroupClass

Die Klasse `GroupClass` ist für den Parser von eher untergeordneter Bedeutung. Sie wird ausschließlich benutzt, um eine Menge von zusammengehörigen Ereignissen auf einmal zu abonnieren. Deswegen ist der Funktionsumfang minimal. Es existieren z.B. keine Funktionen die bestimmen, welche Ereignisse Bestandteil der jeweiligen Gruppe sind, da dieses durch den gegebenen Ablauf von Erzeugung und Abonnement unnötig ist (siehe Abschnitt 6.3.1).

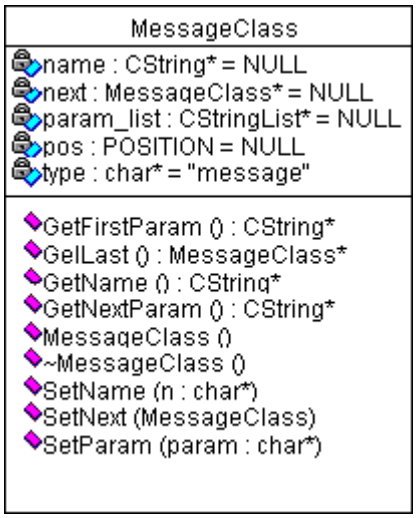

Abbildung A.10: Klasse MessageClass

Die wichtigste Funktion dieser Klasse ist das Speichern der verwendeten Parameter einer Nachricht. Diese werden benötigt, wenn eine Verknüpfung gefunden und die verknüpfte Nachricht mit ihren Parametern gesendet werden soll. Ein Parameter, der in einer Instanz abgelegt werden soll, muß als `VariableKlass`-Objekt vorhanden sein. Es existiert allerdings keine Verknüpfung zu dieser Instanz; der Name des Parameters wird in eine Stringliste (`param_list`) eingefügt.

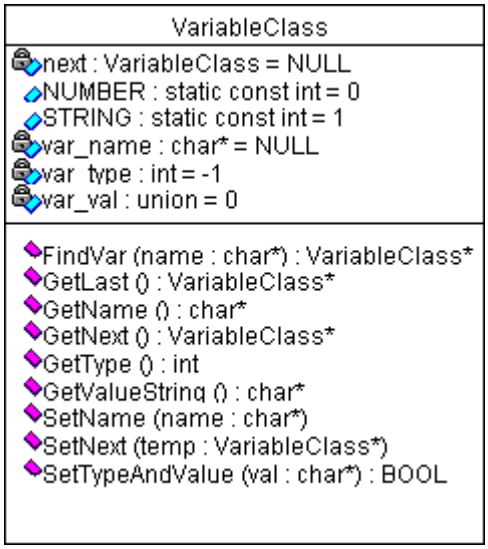

Abbildung A.11: Klasse VariableClass

Diese Klasse stellt den Container für Scriptvariablen zur Verfügung. Da zur Zeit nur die Datentypen STRING und NUMBER (sowie ein theoretischer Sonderfall „UNDEF“, für undefinierte Variablen) existieren, wurde auf weitere Unterklassen verzichtet. Der Wert einer Variable kann nur als Zeichenkette ausgelesen werden.

Die folgende Abbildung zeigt alle Klassen und in welcher Abhängigkeit sie zueinander stehen.

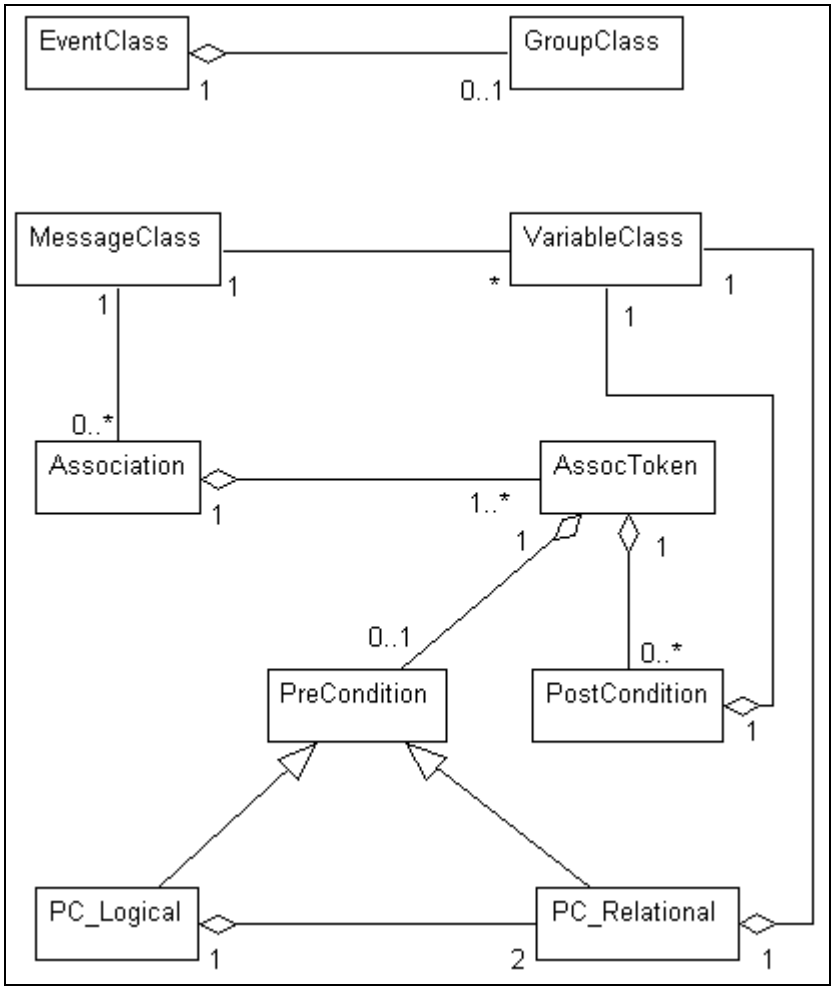

Abbildung A.12: Klassendiagramm der Containerklassen

Hauptklassen

Diese Klassen stellen die Hauptfunktionalität des Parsers bereit. Von Interesse ist allerdings nur die Klasse `Parser`. Sie stellt die Schnittstelle zwischen dem MessageMapper und dem eigentlichen Parser bereit. Eine genaue Definition der Klasse `ANTLRParser` existiert in [Par93], weswegen ich hier darauf verzichte.

Die Funktionen von `Parser` dienen in erster Linie dazu, geparste Elemente in verkettete Listen einzuhängen (`AddNewXXX`) bzw. besagte Listen an den Teil zurückzugeben, der den Parser startete (`GetXXX`). Mit `GetError` lassen sich vorhandene Fehler auslesen. Die Funktion `SetClient` macht den Client dem Parser bekannt, der dieses Script geschickt hat. Damit werden die im Anschluß geparsten Ereignisse und Verknüpfungen später nur von diesem Client genutzt. Das InitialScript wird per Festlegung als „global“ definiert und ist damit auch für andere Clients zugänglich. Schließlich existieren noch eine Reihe von Suchfunktionen(`FindXYZ`), mit denen man gezielt prüfen kann, ob eine Instanz mit gegebenen Namen existiert.

Für jeden Durchlauf des Parsers wird eine neue Instanz von `Parser` erzeugt.

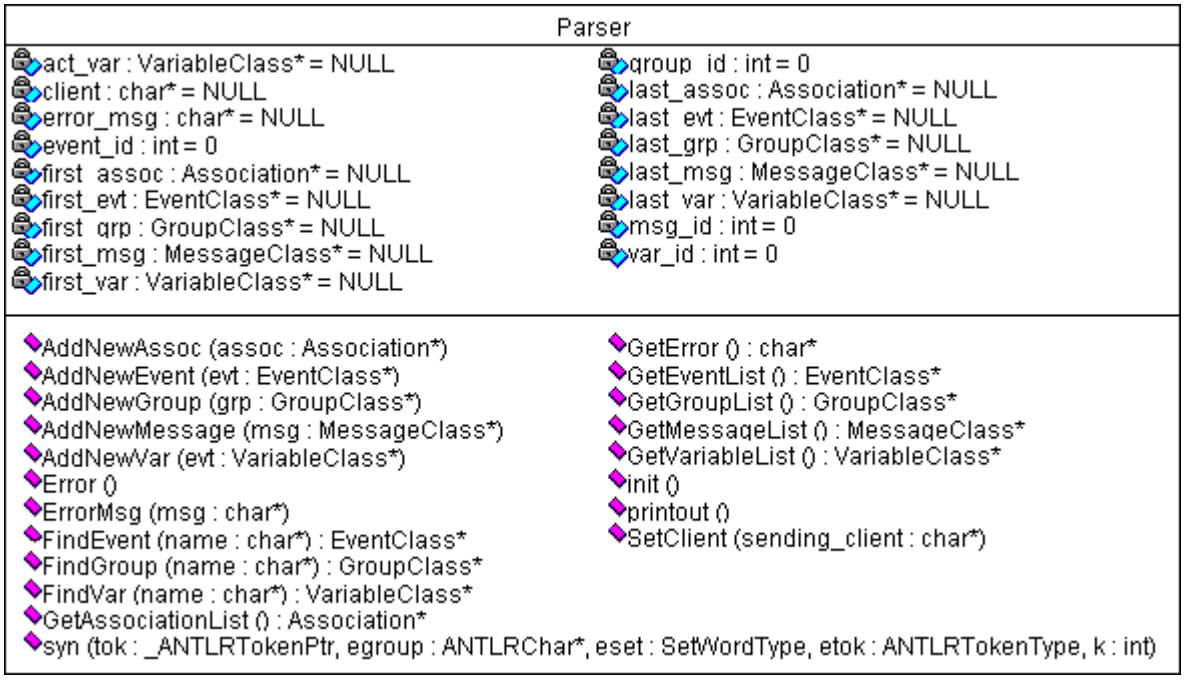

Abbildung A.13: Klasse Parser

In der letzten Abbildung werden die Verbindungen zwischen Haupt- und Containerklassen explizit dargestellt.

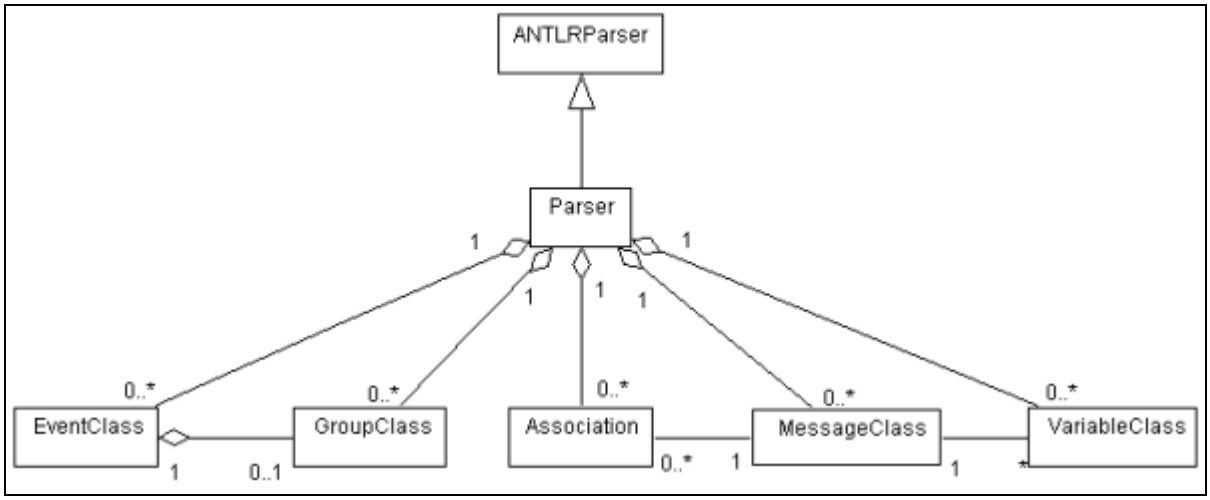

Abbildung A.14 : Assoziationen der Haupt- und Containerklassen

Anhang B: Glossar

Client

Ein Client nimmt Dienste in Anspruch, die ein →Server zur Verfügung stellt.

DLL

Eine Dynamik Link Library (DLL) ist eine Bibliothek auf MS Windows Plattformen, die sich dadurch auszeichnen, daß sie in ein Programm zur Laufzeit eingebunden werden können, um auf deren Funktionalitäten zuzugreifen. Andere Betriebssysteme verfügen über ähnliche Mechanismen unter anderem Namen.

Embedded System

Es handelt sich dabei um Systeme, die ein Set von wohldefinierten, festspezifizierten Funktionen ausführen. Benutzer können auf diese Funktionen nur über spezielle Schnittstellen (etwa Knöpfe, Schalter oder Drehregler) zugreifen. Typische Embedded Systems sind Handys, CD-Player und Synthesizer.

Interoperabilität

Interoperabilität bezeichnet, wie gut sich eine Komponente in ein →Verteiltes System einfügt. Wird eine Komponente zwischen zwei Systemen ausgetauscht, die derselben Architektur zugrunde liegen, ist die Interoperabilität normalerweise sehr hoch.

Prozeß

Ein in Ausführung befindliches Programm.

Server

Ein Server stellt Dienste zur Verfügung (z.B. Email), die von →Clients in Anspruch genommen werden können.

Verteiltes System

Ein verteiltes System besteht aus mehreren Komponenten, die zusammen arbeiten. Hierbei kann es sich um einen softwarebasierten Ansatz handeln (mehrere Programme auf einem Rechner) oder auch um physikalische Verteiltheit (mehrere Rechner im Verbund).

Wiederverwendbarkeit

Beschreibt den Grad, wie gut sich eine Komponente für verschiedene Zwecke einsetzen läßt. Wiederverwendbarkeit kommt häufig in der Simulation vor, wo dieselben Modelle in verschiedenen Umgebungen eingesetzt werden.

Anhang C: Literaturverzeichnis

[Aga97] Agamanolis, Stefan / Bove, V. Michael jr.; *Multilevel Scripting for Responsive Multimedia*; in: IEEE MultiMedia; 4(4): 40 - 50 Oct - Dec 1997

[Anl00] Anlauff, Matthias et al; *Using Domain-Specific Languages for the Realization of Component Composition*; in: Lecture notes in computer science; 1783: 112 - 126 2000

[ASU86] Aho, Alfred V. / Sethi, Revi / Ullman, Jeffrey D.; *Compilers: Principles, Techniques, and Tools*; Addison-Wesley Publishing Company; Reading, Massachusetts 1986

[Bau00] Bauer, Günther; *Bausteinbasierte Software: Eine Einführung in die modernen Konzepte des Software-Engineering*; Friedrich Vieweg & Sohn Verlagsgesellschaft; Braunschweig/Wiesbaden 2000

[BB00] Brauer, Volker / Bruns, F. W. et al; *Final Report – MM1002 BREVIE: Bridging Reality and Virtuality with a Graspable User Interface*; artec-paper 81; Universität Bremen, Forschungszentrum Arbeit – Umwelt – Technik (artec) 2000

[Bro95] Brooks, Frederick P. jr.; *The Mythical Man-Month: Essays on Software Engineering – Anniversary Edition*; Addison-Wesley Longman Inc.; Reading, Massachusetts 1995

[Coy92] Coy, Wolfgang; *Aufbau und Arbeitsweise von Rechenanlagen* (2. Auflage); Friedrich Vieweg & Sohn Verlagsgesellschaft mbH; Braunschweig/Wiesbaden 1992 (1. Auflage von 1987)

[Deu99] Deursen, Arie van; *Using a Domain-Specific Language for Financial Engineering*; in: ERCIM News; 38: 21 - 21 Jul 1999; WWW: http:// www.ercim.org/publication/Ercim_News/enw38/EN38.pdf (Stand: 18.03.2002)

[DKW98] Dahmann, Judith S. / Kuhl, Frederick / Weatherly, Richard; *Standards for Simulation: As Simple As Possible But Not Simpler – The High Level Architecture For Simulation*; in: Simulation 71(6): 378 - 387 Dec 1998

[DS95] Donnelly, Charles / Stallman, Richard; *BISON – The YACC-compatible Parser Generator*; WWW: http://www.mathematik.uni-kassel.de/docu/bison-1.25/ (Stand: 18.03.2002)

[DW96] Dongarra, J. J. / Walker, D. W.; *MPI: A Standard Message Passing Interface*; in: Supercomputer 12(1): 56 - 68 Jan 1996

[EP98] Eriksson, Hans-Erik / Penker, Magnus; *UML Toolkit*; John Wiley & Sons, Inc.; New York/Chichester/Weinheim/Brisbane/Singapore/Toronto 1998

[Ern00] Ernst, Hauke et. al.; *WP6400 – Complex Construction Kit Design Specification*; Unveröffentlichtes Projektpapier; Arbeitspapier D64 des Projektes „BREVIE“; Bremen 2000

[Fis95] Fishwick, Paul A.; *Simulation model design and execution: building digital worlds*; Prentice Hall, London 1995

[Gar98] Garshol, Lars Marius; *BNF and EBNF: What are they and how do they work?*; WWW: http://www.garshol.priv.no/download/text/bnf.html (Stand: 18.03.2002)

[Gei94] Geist, G. Al et al; *PVM: Parallel Virtual Machine*; MIT Press, Cambridge (Masachusettes) 1994; WWW: http://www.netlib.org/pvm3/book/node1.html (Stand: 18.03.2002)

[Gei96] Geist, G. Al et al; *PVM and MPI: a comparison of features*; WWW: http://www.epm.ornl.gov/pvm/PVMvsMPI.ps (Stand: 18.03.2002)

[Gei97] Geist, G. Al; *Advanced Tutorial on PVM 3.4*; WWW: http://www.csm.ornl.gov/pvm/EuroPVM97/ (Stand: 18.03.2002)

[HH89] Herrtwich, Ralf Guido / Hommel, Günter; *Nebenläufige Programme*; Springer-Verlag; Berlin-Heidelberg-New York; 1989 (2. Auflage 1994)

[HLA98a] *High Level Architekture Rules Version 1.3*; WWW: http://www.dmso.mil/public/library/projects/hla/specifications/rules1-3dod (Stand: 18.03.2002)

[HLA98b] *High Level Architekture Object Model Template Specification Version 1.3*; WWW: http://www.dmso.mil/public/library/projects/hla/specifications/omt1-3dod (Stand: 18.03.2002)

[HLA98c] *High Level Architekture Interface Specification Version 1.3*; WWW: http://www.dmso.mil/public/library/projects/hla/specifications/main_body (Stand: 18.03.2002)

[JIN99] *Jini ™ Architectural Overview – Technical White Paper*; WWW: http://www.sun.com/jini/whitepapers/architecture.pdf (Stand: 18.03.2002)

[Jon96a] Jones, Capers; *What Are Function Points?*; WWW: http://psaweb.pisa.iol.it/archweb/develop/software/0funcmet.htm (Stand: 18.03.2002)

[Jon96b] Jones, Capers; *What Is A Language Level?*; WWW: http://psaweb.pisa.iol.it/archweb/develop/software/0langtbl.htm (Stand: 18.03.2002)

[Kap89] Kappel, G. et al; *An Object-Based Visual Scripting Environment*; in: Tsichritzis, D. (Herausgeber); *Object Composition*; Université de Geneve 1989

[KEC99] Koniges, Alice E. / Eder, David C. / Cahir, Margarat; *Parallel Computing Architectures*; in: *Industrial Strength Parallel Computing: Programming Massively Parallel Computers*; Koniges, Alice E. (Ed.); Morgan Kaufmann Publishers; Silicon Valley 1994

[Ker98] Kernighan, Brian W.; *Timing Trials, or Trials of Timing: Experiments with Scripting and User-Interface Languages*; in: Software – Practice and Experience; 28(8): 819 - 843 Jul 1998

[MPI94] Otto, Steve (Ed.) et al.; *MPI: A Message Passing Standard*; University of Tennessee; Knoxville, Tennessee 1994; WWW: http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html (Stand: 18.03.2002)

[MPI97] Huss-Lederman, Steve (Ed.) et al.; *MPI-2: Extensions to the Message-Passing Interface*; University of Tennessee; Knoxville, Tennessee 1997; WWW: http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html (Stand: 18.03.2002)

[MT97] Messner, Bill / Tilbury, Dawn; *Control Tutorials for Matlab: Matlab Basics Tutorial*; WWW: http://www.engin.umich.edu/group/ctm/basic/basic.html (Stand: 18.03.2002)

[Nie99] Niestadt, Jan; *flipcode – Programming Tutorials: Scripting Language Tutorial Series*; WWW: http://www.flipcode.com/tutorials/ (Stand: 18.03.2002)

[Ous98] Ousterhout, John K.; *Scripting: Higher-Level Programming for the 21st Century*; in: Computer: innovative technology for computer professionals; 31(3): 23 - 30 Mar 1998

[Par93] Parr, Terence John; *Language Translation Using PCCTS and C++*; Automata Publishing Company; San Jose, Kalifornien 1993; WWW: http://www.antlr.org/papers/pcctsbk.pdf (Stand: 18.03.2002)

[Pax95] Paxson, Vern; *FLEX, version 2.5 – A fast scanner generator*; WWW: http://www.mathematik.uni-kassel.de/docu/flex-2.5.4/ (Stand: 18.03.2002)

[Por99] Porto-Barreto, Luciano; *Domain-Specific Languages - An Overview*; WWW: http://compose.labri.u-bordeaux.fr/documentation/dsl/ (Stand: 18.03.2002)

[Pos96] Poswig, Jörg; *Visuelle Programmierung – Computerprogramme auf grafischem Wege erstellen*; Carl Hanser Verlag; München, Wien 1996

[PQ95] Parr, Terence John / Quong, R. W.; *ANTLR: A Predicated-LL(k) Parser Generator*; in: Software – Practice and Experience; 25(7): 789 - 810 Jul 1995

[PQ96] Parr, Terence John / Quong, Russell W.; *LL and LR Translators Need k>1 Lookahead*; in ACM SIGPLAN Notices, 31(2): 27 – 34 Feb 1996

[Pre00] Prechelt, Lutz; *An empircial comparison of C, C++, Java, Perl, Python, Rexx and Tcl*; in: IEEE Computer 33(10): 23 - 29 Oct 2000

[Ric00] Richter, Mathias W.; *Iava: yet another interpreter for scripting within Jaya platform*; in: Software – Practice and Experience; 30(2): 81 - 106 Feb 2000

[Rußß97] Rußmann, Arnd; *Dynamic LL(k) parsing*; in: Acta Informatica; 34, 267 - 289 1997

[SK98] Straßburger, Steffen / Klein, Ulrich; *Integration des Simulators SLX in die High Level Architecture*; in: Preim, B. / Lorenz, P. (Ed.); Proceedings der Tagung Simulation und Visualisierung pp. 32 - 40; SCS European Publishing House; Magdeburg 1998

[Sta96] Stanchfield, Scott; *Converting a grammar from LALR to LL*; WWW: http://javadude.com/articles/lalrtoll.html (Stand: 18.03.2002)

[Sta00] Stancovic, Nenad / Zhang, Kang; *An Evaluation of Java implementations of message-passing*; in: Software – Practice and Experience; 30(7): 741 – 763 Jun 2000

[Tan94] Tanenbaum, Andrew S.; *Moderne Betriebssysteme*, Carl Hanser Verlag München Wien / Prentice-Hall International Inc., München / Wien / London 1994

[Vos00] Vossen, Gottfried / Witt, Kurt-Ulrich; *Grundlagen der Theoretischen Informatik mit Anwendungen*; Friedrich Vieweg & Sohn Verlagsgesellschaft mbH; Braunschweig/Wiesbaden 2000

[YSB98] Yu, Lois C. / Steinman, Jeffrey S. / Blank, Gary E.; *Adapting Your Simulation for HLA*; in: Simulation 71(6): 410 - 420 Dec 1998

Anhang D: Eidesstattliche Erklärung

Ich versichere hiermit, daß ich die vorliegende Arbeit selbständig und nur unter Verwendung der angeführten Literatur angefertigt habe.

Bremen, den 26.03.02