

HOCHSCHULE ZITTAU/GÖRLITZ (FH)

FACHBEREICH INFORMATIK

**AtoCC - eine Lernumgebung für  
theoretische Informatik**

*Diplomarbeit*

zur Erlangung des Grades eines Diplom-Informatikers (FH)  
des Studienganges Technische Informatik  
an der Hochschule Zittau/Görlitz (FH) – University of Applied Sciences

vorgelegt von

Michael Hielscher

geboren am 1. August 1982 in Zittau.

Referent: Prof.Dr.rer.nat. Christian Wagenknecht

Görlitz, 07.07.2006

# Kurzreferat

Im Informatikunterricht an Universitäten, Hochschulen und Gymnasien werden Inhalte aus der theoretischen Informatik vermittelt. Durch den Einsatz von Software kann der Lernprozess positiv unterstützt werden. Verschiedene theoretische Inhalte können mit Hilfe von Software visualisiert und/oder mit praktischen Übungen am Computer ergänzt werden.

Diese Diplomarbeit schafft eine Lernumgebung für die Teilgebiete Automatentheorie, formale Sprachen und den Compilerbau. Gebildet wird diese Umgebung aus mehreren Softwarewerkzeugen, die jeweils für ein Themengebiet verantwortlich sind. Dabei konzentriert sich diese Arbeit vor allem auf den Entwurf und die Entwicklung der Werkzeuge und deren Zusammenwirken.

In Kapitel 1 wird die Motivation und das Ziel dieser Arbeit beschrieben sowie verwendete Begriffe definiert. Im Folgenden werden die bereits vorhandenen Softwarelösungen analysiert und die Ansprüche, die an eine solche Arbeit bestehen, angesprochen. Davon ausgehend wird die Architektur und die Zielstellung der Lernumgebung AtoCC vorgestellt (Kapitel 2). Kapitel 3 beschreibt eine Auswahl informatischer Konzepte, die bei der Entwicklung verwendet wurden. Die entstandenen Softwarekomponenten von AtoCC werden in Kapitel 4 im Detail beschrieben. Abschließend wird auf die Erprobung einzelner Komponenten in der Praxis und die Ergebnisse dieser Arbeit eingegangen (Kapitel 5).

## **Danksagung**

An dieser Stelle bedanke ich mich bei den Menschen, die mich bei der Erstellung der Diplomarbeit in vielfältiger Weise unterstützt haben. Bei Herrn Prof. Dr. Christian Wagenknecht bedanke ich mich für seinen Ideenreichtum und für die aufgebrauchte Zeit seiner angenehmen Betreuung.

Herzlicher Dank gebührt allen Studenten, die mit Ideen und Kritiken zu diesem Projekt beigetragen haben. Auch meinen Kommilitonen Angela Weickelt und Robert Nittel danke ich für ihre Verbesserungsvorschläge während der schriftlichen Ausarbeitung und ihrem Interesse an dieser Arbeit.

Mein besonderer Dank gebührt meiner Familie und meiner Freundin, die mich finanziell und seelisch in der Zeit meines Studiums unterstützt haben.

# Inhalt

	Seite
Verzeichnis der Abbildungen.....	7
Verzeichnis der Tabellen .....	7
<b>1 EINLEITUNG .....</b>	<b>7</b>
1.1 Motivation.....	7
1.2 Anwendungsgebiete.....	8
1.3 Begriffsdefinitionen .....	9
<b>2 IST / SOLL ANALYSE .....</b>	<b>12</b>
2.1 Vorhandene Softwarelösungen.....	12
2.2 Entwicklungsziel - einheitliche Gesamtlösung.....	13
2.3 AtoCC Architektur.....	14
2.4 AtoCC - technische Umsetzung.....	16
<b>3 FACHWISSENSCHAFTLICHE GRUNDLAGEN.....</b>	<b>17</b>
3.1 XML und XML Schema .....	17
3.2 Data Access Layer .....	18
3.3 Corporate-Design-Konzept.....	19
<b>4 ATOCC – KOMPONENTEN.....</b>	<b>21</b>
4.1 AutoEdit.....	21
4.1.1 Konstruktion, Transformation und Simulation von Automaten.....	21
4.1.2 Qualitativer Export von Transitionsdiagrammen .....	22
4.1.3 Automatenexport als HTML-Seite.....	23
4.2 AutoEdit Workbook.....	24
4.2.1 Client/Server Umsetzung.....	24
4.2.2 Webdatenbank.....	25
4.2.3 Lernaufgaben - Erstellung & Wartung .....	26
4.2.4 Lernaufgaben – Lösungsüberprüfung.....	27
4.3 TDiag .....	29
4.3.1 Entwicklung von T-Diagrammen mit TDiag .....	29
4.3.2 Abhängigkeiten unter Diagrammkomponenten.....	29
4.3.3 Ausführung/Abarbeitung eines T-Diagramms.....	31
4.3.4 Speicherformat von TDiag .....	33
4.4 Visual Compiler Compiler.....	35

4.4.1	Scanner.....	35
4.4.2	Parser.....	37
4.4.3	Übersetzung in einen Compiler Quellcode .....	39
4.4.4	SchemeYACC.....	40
4.4.5	VCC Speicherformat .....	40
4.5	SchemeEdit .....	42
4.5.1	SchemeEdit eine Entwicklungsumgebung für Scheme .....	42
4.5.2	Scheme Interpreter und damit verbundene Probleme.....	43
<b>5</b>	<b>ERPROBUNG, ERGEBNISSE UND AUSBLICK .....</b>	<b>44</b>
5.1	SchemeEdit in der Praxis.....	44
5.2	Erfahrungen mit AutoEdit .....	44
5.3	Einsatz von VCC .....	46
5.4	Ausblick.....	46
	<b>LITERATURVERZEICHNIS .....</b>	<b>47</b>
	<b>ANHANG A: XML SCHEMATA DER KOMPONENTEN.....</b>	<b>50</b>
	AutoEdit / AutoEdit Workbook XML Schema .....	50
	TDiag XML Schema.....	51
	VCC XML Schema.....	52
	<b>ANHANG B: ANWENDUNGSBEISPIELE FÜR ATOCC .....</b>	<b>53</b>
	AutoEdit: Vom Automat zum LaTeX Dokument.....	53
	AutoEdit Workbook: Eine Aufgabe lösen .....	58
	AutoEdit Workbook: Eine Chicken Run Aufgabe .....	61
	TDiag: Entwicklung und Ausführung eines T-Diagramms .....	63
	VCC: Entwicklung eines einfachen Compilers .....	67
	<b>ANHANG C: AUSGEWÄHLTE QUELLCODEBEISPIELE .....</b>	<b>74</b>
	Generieren von Scheme Code für einen DEA .....	74
	Generieren von MS-DOS Batch Dateien.....	76
	TDiag Compiler / Interpreter Presets.....	79
	<b>ANHANG D: DATENTRÄGERBESCHREIBUNG.....</b>	<b>83</b>

## Verzeichnis der Abbildungen

Abbildung 1:	Beispiel Transitionsdiagramm endlicher Automat .....	9
Abbildung 2:	T-Diagramm Compilerbaustein.....	11
Abbildung 3:	Beispiel T-Diagramm .....	11
Abbildung 4:	AtoCC Komponenten und Wechselwirkung .....	15
Abbildung 5:	AtoCC Lernaktivitäten .....	15
Abbildung 6:	DB-Zugriff ohne DAL .....	18
Abbildung 7:	DB-Zugriff mit DAL .....	19
Abbildung 8:	AutoEdit Screenshot.....	20
Abbildung 9:	TDiag Screenshot.....	20
Abbildung 10:	AutoEdit Transitionsdiagramme .....	22
Abbildung 11:	AutoEdit 300 DPI Transitionsdiagramme .....	22
Abbildung 12:	AutoEdit HTML Ausgabe .....	23
Abbildung 13:	AutoEdit Workbook Client / Server Architektur .....	25
Abbildung 14:	AutoEdit Workbook MySQL Tabelle .....	26
Abbildung 15:	TDiag Programmbaustein - Andockpointer .....	30
Abbildung 16:	TDiag Andocken eines Interpreters an ein Programm.....	31
Abbildung 17:	TDiag Diagramm Beispiel.....	31
Abbildung 18:	VCC Scanner Beispiel .....	36
Abbildung 19:	VCC Scanner Arbeitsweise.....	37
Abbildung 20:	VCC Parser Beispiel.....	38
Abbildung 21:	VCC Kompilationsprozess.....	39
Abbildung 22:	SchemeEdit Screenshot.....	43
Abbildung 23:	AutoEdit Downloadstatistik .....	45

## Verzeichnis der Tabellen

Tabelle 1:	Auszug Lehrplan Informatik Jahrgangsstufe 13 .....	8
Tabelle 2:	Klassifikation nach Chomsky.....	10

# 1 Einleitung

Die Begriffe Schule bzw. Unterricht sind im Folgenden als Synonyme für jegliche Art von Bildungseinrichtung und deren Art, ihre Lerninhalte zu vermitteln, anzusehen. Die verwendeten Begriffe für Lehrer/-innen bzw. Schüler/-innen sind im weitesten Sinne (und auch stets in weiblicher Form) zu verstehen.

## 1.1 Motivation

An Universitäten, Hochschulen und Gymnasien werden, neben vielen anderen, informatische Inhalte vermittelt. Dabei werden die Schüler/-innen bei der Behandlung der unterschiedlichen Thematiken, wie etwa dem Programmieren in C++, mit praktischen Aufgaben am Rechner beim Erlernen der theoretischen Grundlagen unterstützt. Dies ist jedoch nicht in allen Teilgebieten der Informatik so einfach möglich. In der theoretischen Informatik werden Konzepte vermittelt, die oftmals nur bedingt mit praktischen Lerninhalten ergänzt werden können. Schülern/-innen fällt es dadurch schwerer, das vermittelte Wissen zu verstehen bzw. zu festigen. Helfen können spezielle Softwarelösungen, in denen das theoretische Wissen interaktiv aufgearbeitet wurde und somit dem Schüler/-in eine Möglichkeit geboten werden kann, auch über die regulären Unterrichtszeiten hinaus, Aufgaben am Computer zu bearbeiten. Des Weiteren können interaktive Lerninhalte zusätzlich motivierend wirken, wie etwa in [Högn 2006] beschrieben:

*„Die Motivation des Lernenden wird gesteigert, wenn er aktiv in den Lernprozess integriert wird, und seinen Lernweg selbst bestimmen kann, d.h. wenn er nicht nur die Rolle eines passiven Beobachters übernimmt. Dies kann durch vielfältige Interaktionsmöglichkeiten realisiert werden.  
[...]interaktive Animationen oder Simulationen, die zum explorativen Arbeiten anregen, können beim Benutzer Neugier wecken[....]“*

Auch Lehrer/-innen können mit Hilfe von Software bei der Ausarbeitung von Lehrmaterialien im Bereich der theoretischen Informatik unterstützt werden. Spezifische Darstellungen und Diagramme lassen sich nur mit hohem Zeitaufwand mit Standardsoftware anfertigen und sind in der Regel anschließend für eine Weiterverarbeitung ungeeignet. Darüber hinaus benötigen Autoren von Fachliteratur diese Darstellungen in hochauflösender Qualität, um gute Druckergebnisse zu erzielen.

## 1.2 Anwendungsgebiete

An Universitäten werden im Fach „Grundlagen der theoretischen Informatik“ formale Sprachen und Automaten behandelt. Unter verschiedenen Bezeichnungen ist dieses Fach im Grundstudium weit verbreitet. Oftmals schließen sich Vorlesungen zu Sprachübersetzern / Compilerbau an. An Fachhochschulen findet man häufig das Fach „Formale Sprachen und Automaten“, welches sich verstärkt auch mit praktischen Aufgaben mit dieser Thematik auseinandersetzt. Auch viele andere höhere Bildungseinrichtungen, wie Berufsakademien, haben ebenfalls diese Themen in ihren Lehrplänen verankert.

An Gymnasien in Deutschland werden Konzepte der theoretischen Informatik, gemäß Lehrplan, unterrichtet. Die konkreten Formulierungen in den Lehrplänen sind dabei von Bundesland zu Bundesland unterschiedlich<sup>1</sup>. Tabelle 1 zeigt einen Auszug des Lehrplans für Informatik an hessischen Gymnasien für die Jahrgangsstufe 13.

**Tabelle 1:** Auszug Lehrplan Informatik Jahrgangsstufe 13 [LP Hessen]

Verbindliche Unterrichtsinhalte/Aufgaben:	
Formale Sprachen und Grammatiken	reguläre und kontextfreie Grammatiken und Sprachen Anwendung mit Syntaxdiagrammen Chomsky-Hierarchie (LK) kontextsensitive Sprachen (LK)
Endliche Automaten	Zustand, Zustandsübergang, Zustandsdiagramm Zeichen, Akzeptor Simulation realer Automaten (z. B. Getränkeautomat) Anwendung endlicher Automaten (z. B. Scanner) deterministische und nicht-deterministische Automaten (LK) reguläre Ausdrücke (LK) Mensch-Maschine-Kommunikation (LK)
Fakultative Unterrichtsinhalte/Aufgaben:	
Übersetzerbau	Scanner, Parser, Interpreter und Compiler z. B. Steuersprache für Roboter, LOGO, Plotter oder miniPASCAL

Eine der Teilkomponenten von AtoCC (AutoEdit) wird in Kürze in einem Schulbuch für Gymnasien erscheinen.

---

<sup>1</sup> Vergleich auch mit [LP Sachsen]

### 1.3 Begriffsdefinitionen

Die *Automatentheorie* beschäftigt sich mit abstrakten Modellrechnern und den mit ihnen lösbaren Problemen. Dabei gibt es verschiedene Automatentypen. Der Begriff endlicher Automat wird z.B. in [Hopcroft 1997] wie folgt definiert:

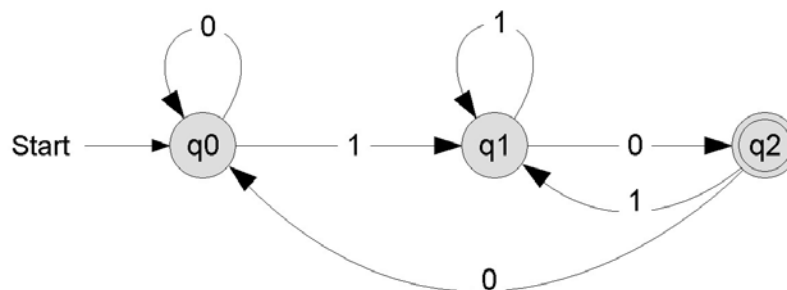
*„Ein endlicher Automat (EA) besteht aus einer endlichen Menge von Zuständen und einer Menge von Transitionen, die auf einem Eingabesymbol aus einem Alphabet  $\Sigma$  arbeiten und einen Zustand in einen anderen überführen. Für jedes Eingabe-Symbol existiert genau ein Zustandsübergang [zutreffend für deterministische endliche Automaten]—möglicherweise zurück in den gleichen Zustand.  
Ein Zustand, der gewöhnlich mit  $q_0$  bezeichnet wird, ist der Anfangszustand, in dem der Automat startet. Einige Zustände sind ausgezeichnet als akzeptierende bzw. Endzustände.“*

Der endliche Automat ist der primitivste Automatentyp und kann als äquivalentes Beschreibungsmittel für reguläre Sprachen verwendet werden<sup>2</sup>.

Für die Darstellung von Automaten werden gewichtete, gerichtete Graphen verwendet, auch Transitionsdiagramm genannt. Diese sind in [Hopcroft 1997] wie folgt beschrieben:

*„Die Knoten im Graph entsprechen den Zuständen des EA. Gibt es bei Eingabe von  $a$  einen Übergang vom Zustand  $q$  in den Zustand  $p$ , dann existiert ein mit  $a$  markierter Pfeil vom Zustand  $q$  in den Zustand  $p$  im Transitionsdiagramm. Der EA akzeptiert eine Zeichenkette  $x$ , wenn die Transitionsfolge, die den Symbolen in  $x$  entspricht, den Anfangszustand in einen akzeptierenden Zustand [Endzustand] überführt.“*

Ein Beispiel für einen solchen Graphen ist in Abbildung 1 dargestellt. Der damit beschriebene Automat  $M = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_2\})$  akzeptiert alle Wörter, die mit dem Teilwort „10“ enden.



**Abbildung 1:** Beispiel Transitionsdiagramm endlicher Automat

<sup>2</sup> Die entsprechenden Beweisführungen findet man ebenfalls in [Hopcroft 1997]

**Formalen Sprachen** liegen formale Grammatiken zugrunde. Tabelle 2 zeigt die Einteilung nach Chomsky<sup>3</sup>. Für die verschiedenen mächtigen Sprachklassen können Automaten, mindestens des entsprechend aufgeführten Typs, als äquivalente Beschreibungsmittel eingesetzt werden. Man erkennt den engen Zusammenhang zwischen formalen Sprachen und abstrakten Automaten.

**Tabelle 2:** Klassifikation nach Chomsky [Sander 1992]

<u>Chomsky-Hierarchie</u>	<u>Grammatiken</u>	<u>Sprachen</u>	<u>Minimaler Automat</u>
Typ-0	Uneingeschränkt	rekursiv	Turingmaschine
Typ-1	kontextsensitiv	kontextsensitiv	linear beschränkt
Typ-2	Kontextfrei	kontextfrei	Kellerautomat
Typ-3	Regulär	regulär	endlicher Automat

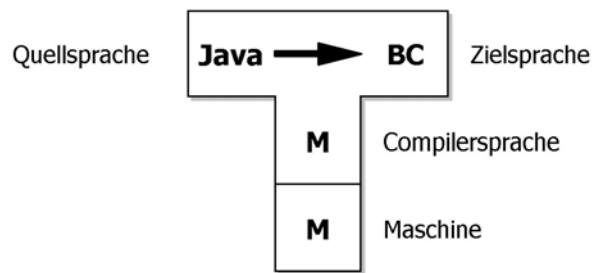
Die Automatentheorie und die formalen Sprachen finden eine praktische Anwendung bei der Entwicklung von Programmen, die einen Text der Quellsprache  $S_1$  in eine Zielsprache  $S_2$  übersetzen. Diese Programme bezeichnet man als Compiler. Ein Teilgebiet der Informatik beschäftigt sich mit der Konstruktion dieser Compiler, dem so genannten **Compilerbau**. Compiler arbeiten in der Regel in vier Phasen<sup>4</sup>:

- *Lexical analysis* – die einzelnen Zeichen des Quelldokuments werden in eine Kette von Symbolen (Token) übersetzt.
- *Syntax analysis* – die Symbolkette wird mit der dem Compiler zu Grunde liegenden Grammatik verglichen und somit die syntaktische Korrektheit des Quelldokuments festgestellt.
- *Type checking* – in modernen Hochsprachen werden Variablen und Objekte nach ihrem Typ klassifiziert. In diesem Schritt muss geprüft werden, ob verschiedene Operatoren auf die erkannten Variabeltypen angewendet werden können.
- *Code generation* – der Quellcode der Zielsprache wird, anhand der Repräsentation aus Schritt 2, erzeugt.

<sup>3</sup> Diese Klassifikation wurde 1956 von Noam Chomsky beschrieben

<sup>4</sup> Vgl. [Wirth 1996] (S. 6-7)

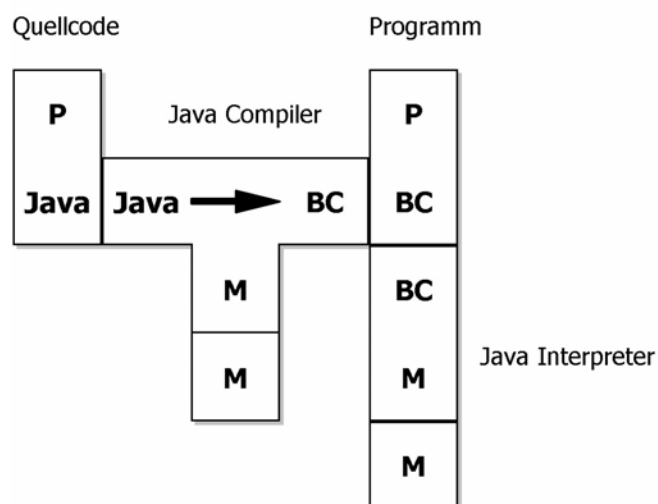
Die Prozesse, die im Compilerbau auftreten, können durch Diagramme dargestellt werden – den so genannten *T-Diagrammen*. Dabei werden Compiler als eine Art „T“ dargestellt<sup>5</sup> (siehe Abbildung 2).



**Abbildung 2:** T-Diagramm Compilerbaustein

Quellsprache und Zielsprache werden symbolisch mit einem Pfeil verbunden, der den Übersetzungsprozess verbildlichen soll. Die Sprache, in der der Compiler selbst verfasst wurde, ist in Abbildung 2 mit „Compilersprache“ beschriftet. Ist der Compiler auf einer Maschine ausführbar, wird dies durch ein optionales Rechteck am unteren Ende dargestellt.

Durch die Kombination dieser T-Bausteine können komplexe Prozesse dargestellt werden. Neben dem Compiler-Baustein werden auch Bausteine für Interpreter, Programme und Ein- / Ausgabe verwendet. In Abbildung 3 ist ein T-Diagramm für einen Java Compiler, mit anschließender Interpretation des erstellten Bytecodes, dargestellt.



**Abbildung 3:** Beispiel T-Diagramm

<sup>5</sup> Vgl. [Wagenknecht 2001] (Seite 12-13)

## 2 Ist / Soll Analyse

### 2.1 Vorhandene Softwarelösungen

Vor allem im Bereich Automatentheorie findet man eine unüberschaubare Anzahl kleinerer Tools und Umgebungen. Die wohl bekanntesten Vertreter sind dabei JFLAP, Kara und Exorciser<sup>6</sup>. JFLAP konzentriert sich auf die Konstruktion und Simulation von verschiedenen Automatenmodellen. Das Projekt, unter der Leitung von Susan Rodger, wurde bereits 1993 veröffentlicht und zählt somit zur Standardsoftware in diesem Bereich. Zu bemängeln ist jedoch die ungenaue Umsetzung zur Theorie, wobei zum Beispiel kein Alphabet definiert werden kann und dennoch suggeriert wird, eine Automatentransformation sei ohne dies möglich. Im Jahr 2000 wurde erstmals Kara vorgestellt. Ein spielerischer Ansatz, um Automatentheorie greifbar zu machen (Eine ähnliche Idee wurde bereits im Jahre 1981 mit „Karel the Robot“ gezeigt.). Aber auch hier sind Kompromisse gemacht worden, die den Schüler etwas irre leiten können. Nichtdeterminismus wurde zum Beispiel in Kara einfach durch die Verwendung von „Random“ implementiert. Bei einer nichtdeterministischen Entscheidung des Automaten, wird per Zufall eine der Möglichkeiten ausgewählt und abgearbeitet, ohne im späteren Verlauf „Backtracking“<sup>7</sup> zu verwenden, falls diese Entscheidung nicht zu einem Endzustand geführt hat (eine mögliche Alternativlösung wird im Abschnitt 4.1.3 beschrieben). 2003 wurde Exorciser vorgestellt, welches sich im Gegensatz zu JFLAP deutlich auf die Bereitstellung von Übungsaufgaben und die automatisierte Lösungsüberprüfung spezialisiert. Eine Untersuchung und Einschätzung weiterer Softwarewerkzeuge für die Automatentheorie wurde von Jan Wielgus im Jahre 2003 vorgenommen<sup>8</sup>. AutoEdit wurde im Rahmen meiner Praxissemesterarbeit entwickelt und hatte die Hauptzielstellung, Automaten (Transitionsdiagramme) erstmals für publizistische Zwecke aufzubereiten<sup>9</sup>. Darüber hinaus bietet AutoEdit, ähnlich zu JFLAP, Transformations- und Simulationsmöglichkeiten für verschiedenste Automatentypen, jedoch mit definitionsgerecht vollständig beschriebenen Automaten.

---

<sup>6</sup> Siehe [JFLAP], [Exorciser], [Kara]

<sup>7</sup> Backtracking ist eine algorithmische Methode, um definitiv eine Lösung zu finden, falls eine oder mehrere existieren.

<sup>8</sup> Siehe [FLAT]

<sup>9</sup> Vgl. [Hielscher 2005]

Im Bereich Formale Sprachen werden selbst kaum Werkzeuge angeboten, sondern vielmehr im Anwendungsfeld Compilerbau. Hier sind immer noch LEX und YACC und deren Weiterentwicklungen Flex und Bison zu nennen. Diese zählen zur absoluten Standardsoftware in diesem Bereich und werden für Lehrzwecke auch heute noch gern eingesetzt. Neuere Entwicklungen wie JavaCC, COCO/R oder ANTLR<sup>10</sup> (seit März 2006 auch mit einer GUI Development Environment ausgestattet) gewinnen aber zunehmend an Bedeutung, was Einsatzstatistiken der jeweiligen Entwicklerteams belegen sollen. Für die bereits in Abschnitt 1.3 angesprochenen T-Diagramme gibt es derzeit noch keine spezielle Softwarelösung. Typischerweise werden diese Diagramme mit Bildbearbeitungssoftware oder Ähnlichem gezeichnet. Dies ist auch in vertretbarer Zeit möglich, da es sich hierbei „nur“ um die Erstellung und Beschriftung von Rechtecken handelt. Ein Ausführen eines solchen Diagramms als Prozess ist damit jedoch nicht möglich.

Zusammenfassend kann gesagt werden, dass verschiedene Lösungen vorhanden sind, die sich jeweils an bestimmten Zielstellungen orientieren und auf einen gezielten Themenkomplex konzentrieren. Für publizistische Zwecke hingegen existieren kaum Softwarelösungen (abgesehen von AutoEdit oder kleinerer codebasierter Plugins für LaTeX).

## **2.2 Entwicklungsziel - einheitliche Gesamtlösung**

Softwarewerkzeuge ohne didaktischen Hintergrund (vor allem im Bereich Compilerbau) bringen jedoch mit ihrer Spezialisierung auch Nachteile im Informatikunterricht. Die Autoren versuchen mit ihren Lösungen ihr Themengebiet möglichst vollständig abzudecken, was zur Folge hat, dass eine Vielzahl von Funktionen in einfachen, für Lehrzwecke ausreichenden, Projekten nicht benötigt werden<sup>11</sup> und der Einstieg somit zusätzlich erschwert werden kann.

Bei der Verwendung vieler kleiner Einzellösungen (für Automatentheorie, Compilerbau, ...) entsteht darüber hinaus ein höherer Administrations- und Installationsaufwand. Schüler müssen sich mit jedem Werkzeug und seiner

---

<sup>10</sup> Siehe [JavaCC], [Coco/R] und [ANTLR]

<sup>11</sup> Bereits für einen einfache Compiler (Zahlen-Addierer) ist mit JavaCC eine Menge Quellcode nötig.

Bedienung neu auseinandersetzen, was einen höheren Zeitaufwand erfordert. Erarbeitungen des Schülers in einem Werkzeug sind in der Regel nicht im thematisch Nachfolgenden weiterverwendbar.

Ziel ist es, eine Gesamtlösung zu entwickeln, die den Schüler ausgehend von der Automatentheorie bis hin zum Compilerbau begleitet. Entscheidend für den Erfolg eines solchen Projekts sind hier:

- Einfache Installation (auch auf privatem Rechner des Schülers) und Zugänglichkeit (z.B.: über Internetdownload).
- Benutzerfreundlichkeit und Selbstbeschreibungsfähigkeit der einzelnen Bestandteile stehen im Vordergrund, um eine kurze Einarbeitungszeit zu gewährleisten.
- Zusammenhänge zwischen den theoretischen Grundlagen sollen auch in der Software durch ein Zusammenspiel der einzelnen Programmkomponenten ersichtlich werden.

## 2.3 AtoCC Architektur

AtoCC (from Automata to Compiler Construction) besteht aus mehreren Einzelkomponenten (AutoEdit, AutoEdit Workbook, T-Diag, VCC und SchemeEdit). Die Verwendung mehrerer kleiner Programme bietet vor allem in der Lehre einige Vorteile. Eine schrittweise Heranführung an die einzelnen theoretischen Themengebiete wird damit unterstützt. Ein Auslassen einzelner Komponenten (oder gar die ausschließliche Verwendung einer einzelnen Komponente) ist denkbar. Der in der Regel damit verbundene Nachteil, immer das entsprechende Werkzeug öffnen zu müssen und Daten auszutauschen, wird in AtoCC dadurch gemindert, dass die Komponenten im Bedarfsfall automatisch gestartet werden und somit der Eindruck entsteht, als würde mit einem Einzelprogramm gearbeitet werden. Abbildung 4 zeigt die einzelnen Bestandteile von AtoCC und deren Wechselwirkungen untereinander. Über die gemeinsame Programmiersprache Scheme<sup>12</sup> (verarbeitet in SchemeEdit) sind die Werkzeuge zusätzlich miteinander verbunden.

---

<sup>12</sup> Programmiersprache (LISP-Dialekt, unterstützt Paradigma der funktionalen Programmierung)

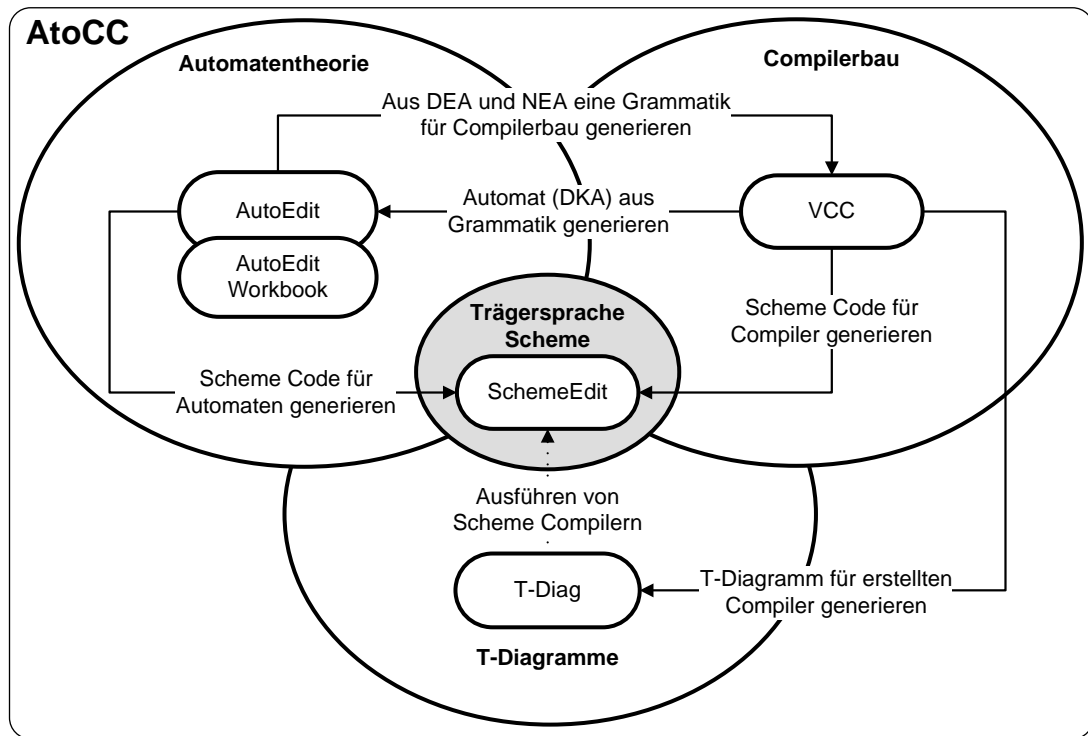


Abbildung 4: AtoCC Komponenten und Wechselwirkung

AtoCC strebt das Ziel an, dem Schüler bei den verschiedenen geistigen Aktivitäten im Lernprozess zu unterstützen. In Abbildung 5 werden diese Stufen anhand der Einzelkomponenten von AtoCC dargestellt<sup>13</sup>.

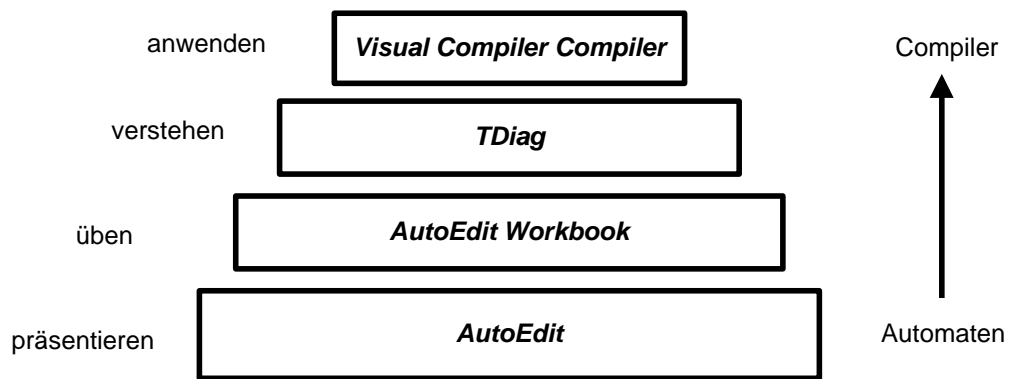


Abbildung 5: AtoCC Lernaktivitäten

<sup>13</sup> Aus [Hielscher 06]

## **2.4 AtoCC - technische Umsetzung**

Für die Entwicklung von AtoCC wurde die objektorientierte Programmiersprache Delphi 5 von Borland verwendet. Ausschlaggebend für diese Wahl waren die geringen Anforderungen und hohe Kompatibilität, die Delphi-Programme an den Zielrechner stellen (kein Interpreter oder zusätzliche Bibliotheken nötig). Voraussetzung für AtoCC ist dadurch eine Microsoft Windows Plattform. Konzipiert und getestet wurde AtoCC unter Windows XP, ist aber auch auf älteren Windows-Versionen lauffähig. Für XML Zugriffe wurde die MS-XML Bibliothek verwendet, die auf älteren Systemen (vor .NET Framework) noch nicht standardmäßig installiert ist.

Auf eine plattformunabhängige Lösung, wie etwa mit Java, wurde verzichtet, da erfahrungsgemäß bei komplexen graphischen Benutzungsoberflächen nur bedingt die korrekte Darstellung auf anderen Plattformen sichergestellt werden kann. Eine Integration des systemeigenen Webbrowsers in die Anwendung (wie in AtoCC Komponenten verwendet), ist unter dem Aspekt der Plattformunabhängigkeit nicht möglich.

Für die Anfertigung von Endanwenderinstallationen wird das Freeware-Werkzeug „Inno Setup“ eingesetzt. Damit kann eine dem Windowsstandard gerechte Installation und Deinstallation von AtoCC gewährleistet werden.

Die AtoCC Quelldateien werden dieser Arbeit als Datenträger (siehe Anhang D) beigelegt. Auszugsweise werden aber auch Quellcodebeispiele im Anhang C angegeben.

## 3 Fachwissenschaftliche Grundlagen

### 3.1 XML und XML Schema

Die Extensible Markup Language (engl. für „erweiterbare Auszeichnungssprache“) gilt zurzeit als Standard, um Daten für Mensch und Maschine gleichermaßen lesbar, in Form eines Baumes, abzuspeichern. Nachfolgend ein Beispiel:

```
<Kunde>  
  <Name>Mustermann</Name>  
  <Vorname>Hans</Vorname>  
  <Adresse>Musterstrasse 123, 0815 Musterhausen</Adresse>  
</Kunde>
```

Definiert wurde dieser Standard durch das W3C<sup>14</sup>. Die syntaktische Struktur eines XML-Dokuments wird im klassischen Fall durch die DTD (Dokumenttypdefinition) beschrieben. Als Weiterentwicklung der DTD findet man das XML Schema. Ein Schema ist selbst in XML verfasst und beschreibt genau wie die DTD den Aufbau des Zieldokuments. Die Definitionsmöglichkeiten sind im Schema jedoch deutlich höher und lassen komplexere Datentypen zu. Die vielfältige Literatur zum Thema XML und XML Schema beschreibt dies eingehender (zum Beispiel: [XML 2006]).

Moderne Programmiersprachen bieten bereits fertige XML Parser/Writer in ihren Standardbibliotheken und ermöglichen damit dem Softwareentwickler ohne zusätzlichen Aufwand Anwendungsdaten im XML Format zu lesen bzw. zu speichern.

Für ein Projekt wie AtoCC ist die Verwendung von XML in Voraussicht auf zukünftige Projekte sinnvoll. Somit kann gewährleistet werden, dass andere/zukünftige Software die erstellten Daten mit geringem Aufwand weiterverarbeiten kann. Beispiele für AtoCC XML Dateien, zur Definition vom Automaten, T-Diagrammen oder Compilern, folgen in den Beschreibungen der einzelnen Werkzeuge in Kapitel 4.

---

<sup>14</sup> Siehe: <http://www.w3.org/>

## 3.2 Data Access Layer

Nahezu jede Applikation schreibt und liest Daten aus verschiedensten Quellen (Dateien, Datenbanken,...). Auf dem Markt befinden sich unterschiedlichste Datenbanksysteme (Oracle, MS-SQL, MySQL...), die alle ihre eigenen Verhaltensweisen und Befehle aufweisen. Eine Anwendung, die eine Verbindung zu einem Datenbankserver herstellt und mit diesem kommuniziert, verwendet speziell auf diese Datenbank angepasste Funktionen, welche die Kommunikation steuern. Im einfachsten Fall wird von den Entwicklern ein Datenbanksystem gewählt und auf dessen Basis die Anwendersoftware entwickelt. Der Nutzer ist jedoch anschließend verpflichtet, das gleiche System zu verwenden. Um dem entgegenzuwirken, verwendet man einen Data Access Layer (DAL). In Abbildung 6 und 7 werden die unterschiedlichen Zugriffsarten dargestellt. Die Anwendung greift nicht mehr direkt auf die Datenbank zu, sondern kommuniziert ausschließlich über den DAL. Dieser stellt wiederum die Verbindung mit der Datenbank her und überträgt die Befehle und Antworten von bzw. zur Anwendung. Der Nachteil ist, dass der Vorgang verlangsamt wird, da erst der DAL die Verbindung herstellen muss. Bedeutender Vorteil ist aber, dass bei der Verwendung eines anderen Datenbanksystems ausschließlich der DAL angepasst werden muss, die Anwendung selbst bleibt dabei unberührt.

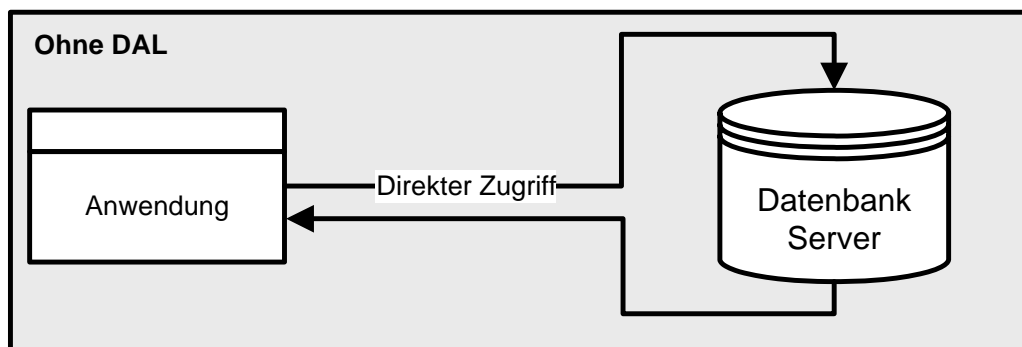
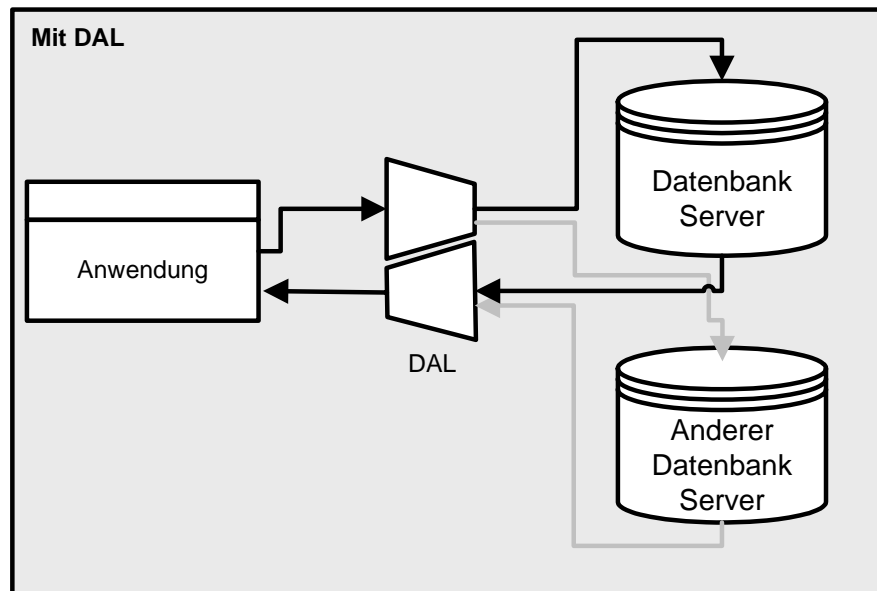


Abbildung 6: DB-Zugriff ohne DAL



**Abbildung 7:** DB-Zugriff mit DAL

Für AutoEdit Workbook und SchemeEdit werden Webdatenbanken verwendet. Da das Datenbanksystem My-SQL von den meisten öffentlichen Webhostern angeboten wird, wurde es für AtoCC verwendet. Dies ist aber durch den Einsatz eines Data Access Layer nicht zwingend nötig. Änderungen an der Datenbank sind somit nicht mit einer zwangsläufigen Änderung des Clienten (AutoEdit Workbook / SchemeEdit) verbunden, wodurch zusätzliche Softwareupdates für den Anwender entfallen.

### 3.3 Corporate-Design-Konzept

Als Erfinder des Corporate Designs gilt Peter Behrens. Er war zwischen 1907 und 1914 künstlerischer Berater für die AEG und entwickelte zum ersten Mal ein einheitliches Unternehmens-Erscheinungsbild<sup>15</sup>. Kommerzielle Firmen setzen heute bei der Softwareentwicklung auf ein einheitliches Design (optisch – Farben, Logos, Grafiken) und eine einheitliche Bedienoberfläche (gleichartige Anordnung von Steuerelementen). Der damit entstehende Wiedererkennungseffekt soll dem Anwender ein Gefühl der Vertrautheit vermitteln. Natürlich spielt im professionellen Bereich vor allem die eigene Vermarktung (Werbung) eine große Rolle. Als Nebeneffekt fällt es dem Nutzer aber auch leichter, sich in unterschiedlicher Software desselben Anbieters einzuarbeiten<sup>16</sup>.

<sup>15</sup> Selbst die Bundesregierung besitzt heute eigene Corporate-Design-Richtlinien Vgl. [CDBund]

<sup>16</sup> Vgl. [Abdullah 2002]

Die Firma Microsoft zeigt in ihren Produkten wie Windows, Office oder Visual Studio ein einheitliches Design, welches sich mit jeder Version in der gesamten Produktpalette widerspiegelt.

Für AtoCC steht der Nebeneffekt der Vertrautheit im Vordergrund und soll das gestellte Ziel, die Minimierung der Einarbeitungszeit, begünstigen. Dazu werden Menüs und Toolbars gleichartig gestaltet und Schaltflächen möglichst identisch angeordnet. In Abbildung 8 und 9 werden Bildschirmfotos von AutoEdit und TDiag dargestellt, um dies zu verdeutlichen.

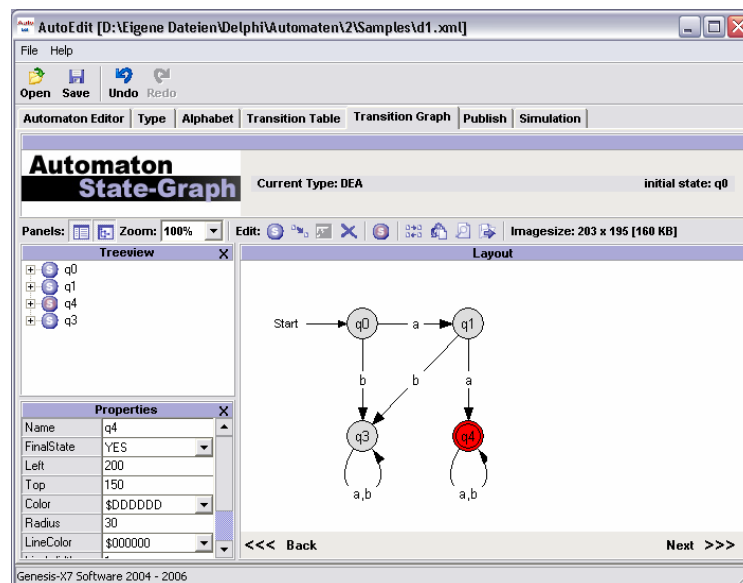


Abbildung 8: AutoEdit Screenshot

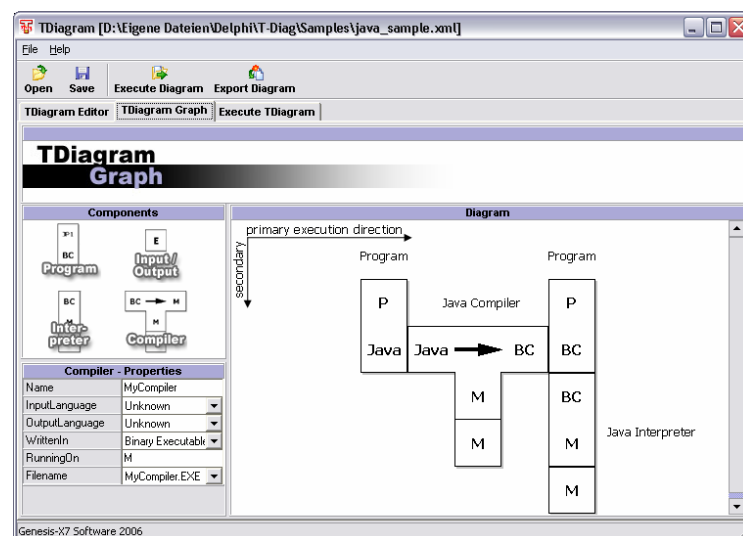


Abbildung 9: TDiag Screenshot

## 4 AtoCC – Komponenten

In diesem Kapitel werden die einzelnen Softwarebausteine von AtoCC im Detail beschrieben. Im Anhang B ist zu jeder Komponente ein konkretes Anwendungsbeispiel mit Bildschirmfotos zu finden. Anhang C beschreibt darüber hinaus ausgewählte Funktionen anhand von Quellcodeauszügen.

### 4.1 AutoEdit

AutoEdit richtet sich an zwei verschiedene Zielgruppen. Zum einen an Lehrende und Autoren, die Transitionsdiagramme für Unterrichtsmaterialien und Fachliteratur benötigen und zum anderen an Schüler, die mit Hilfe von AutoEdit Automaten erstellen, transformieren und simulieren können.

#### 4.1.1 *Konstruktion, Transformation und Simulation von Automaten*

Mit AutoEdit können abstrakte Automaten verschiedenen Typs konstruiert werden. Die notwendigen Schritte der Erstellung werden in AutoEdit, in aufeinander folgenden Formularen abgearbeitet. Der Anwender wird beginnend mit der Typwahl des Automaten, über die Bestimmung des Alphabets, bis hin zur Konstruktion des Transitionsdiagramms (graphisch oder tabellarisch) geleitet. Eine Suche von Diagrammeinstellungen in Menüpunkten entfällt dabei gänzlich (wie etwa in JFLAP). Der entwickelte Automat kann anschließend mit beliebigen Eingabewörtern simuliert werden. Transformationen, von einem Automatentypen in einen anderen, werden ebenfalls von AutoEdit unterstützt.

Die Simulation von Automaten kann, neben einer animierten Darstellung innerhalb von AutoEdit, auch über den Export in einen äquivalenten Scheme-Quellcode erfolgen. Dies erlaubt beispielsweise das Einarbeiten zusätzlicher Ausgabeinformationen (Debug-Informationen) in den generierten Code, um noch detaillierter die Arbeitsweise des Automaten zu verfolgen und zu verstehen.

### 4.1.2 Qualitativer Export von Transitionsdiagrammen

Neben den bereits beschriebenen Funktionen ist der Export von Transitionsdiagrammen, als hochauflösende Bitmap- oder Vektorgraphiken, ein Schwerpunkt von AutoEdit. Für publizistische Zwecke ist AutoEdit mit dieser Funktionalität derzeit einzigartig. Aus diesem Grund bietet AutoEdit im Vergleich zu anderen Automaten-Tools eine Vielzahl von Manipulationsmöglichkeiten des Diagramms (Farben, Linienstärken, Schriftgrößen, ...). In Abbildung 10 sind einige Beispieldiagramme mit verschiedenen Formatierungsmöglichkeiten gezeigt.

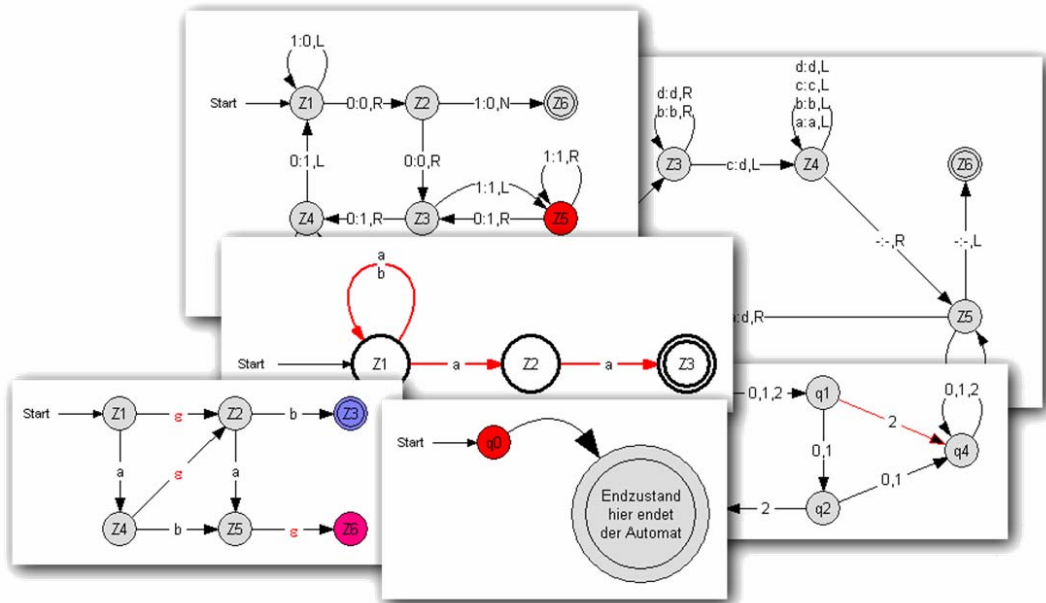


Abbildung 10: AutoEdit Transitionsdiagramme

Im Vergleich zu den in Abbildung 10 verwendeten Bildschirmfotos von Automaten (was in etwa der Qualität von anderen Werkzeugen wie JFLAP entspricht) ist in Abbildung 11 ein hochauflösendes Diagramm dargestellt.

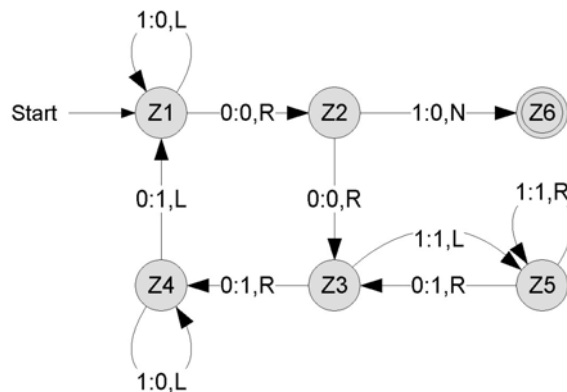


Abbildung 11: AutoEdit 300 DPI Transitionsdiagramme

### 4.1.3 Automatenexport als HTML-Seite

Zur Publikation von Übungsaufgaben wird neben der Papierform auch das Internet verwendet. AutoEdit bietet die Möglichkeit einen Automaten in eine HTML-Seite zu transformieren. Dabei werden Informationen wie Zustandsübergangstabelle, Transitionsdiagramm und Grammatik als JPEG Graphiken in HTML eingebettet. Abbildung 12 zeigt einen solche HTML-Ausgabe für einen NEA. Eine solche generierte Seite kann nun mit Aufgabenstellungen ergänzt werden.

**Automaton**

Type: NEA

Transition Graph:

```
graph LR
    Start((Start)) --> q0((q0))
    q0 -- "a,b" --> q0
    q0 -- "b" --> q1((q1))
    q0 -- "a" --> q3((q3))
    q1 -- "b" --> q2(((q2)))
    q2 -- "a,b" --> q2
    q3 -- "a" --> q4(((q4)))
    q4 -- "a,b" --> q4
```

Definition:  $M = (\{q_0, q_1, q_2, q_3, q_4\}, \{a, b\}, \delta, q_0, \{q_2, q_4\})$

Transition Table:

	a	b
q0	{q0, q3}	{q0, q1}
q1	{}	{q2}
q2	{q2}	{q2}
q3	{q4}	{}
q4	{q4}	{q4}

Grammar:  $G = (N, T, P, s)$

$G = (\{q_0, q_1, q_2, q_3, q_4\}, \{a, b\}, P, q_0)$

$P = \{$   
q0 -> aq0 | bq0 | aq3 | bq1  
q1 -> bq2 | b  
q2 -> aq2 | a | bq2 | b  
q3 -> aq4 | a  
q4 -> aq4 | a | bq4 | b  
 $\}$

Abbildung 12: AutoEdit HTML Ausgabe

## 4.2 AutoEdit Workbook

Wie bereits in Abbildung 4 (Kapitel 2.3 - AtoCC Komponenten und Wechselwirkung) dargestellt, sind AutoEdit und AutoEdit Workbook eng miteinander verbunden. Durch einen Schalter im Programm kann jederzeit zwischen den beiden Programmen hin und her gewechselt werden.

AutoEdit Workbook stellt sich zum Ziel, dem Schüler eine Möglichkeit zu geben eigenständig zu üben und zu experimentieren. Ein Aufgabenkatalog wird über einen Webserver zur Verfügung gestellt und ist jederzeit erweiterbar. Die vom Schüler erstellten Lösungen können direkt vom Programm überprüft werden. Ein zusätzliches Eingreifen der Lehrenden, welcher ohnehin in der Regel mit Aufgabenkontrollen zeitlich überfordert ist, wird nicht benötigt.

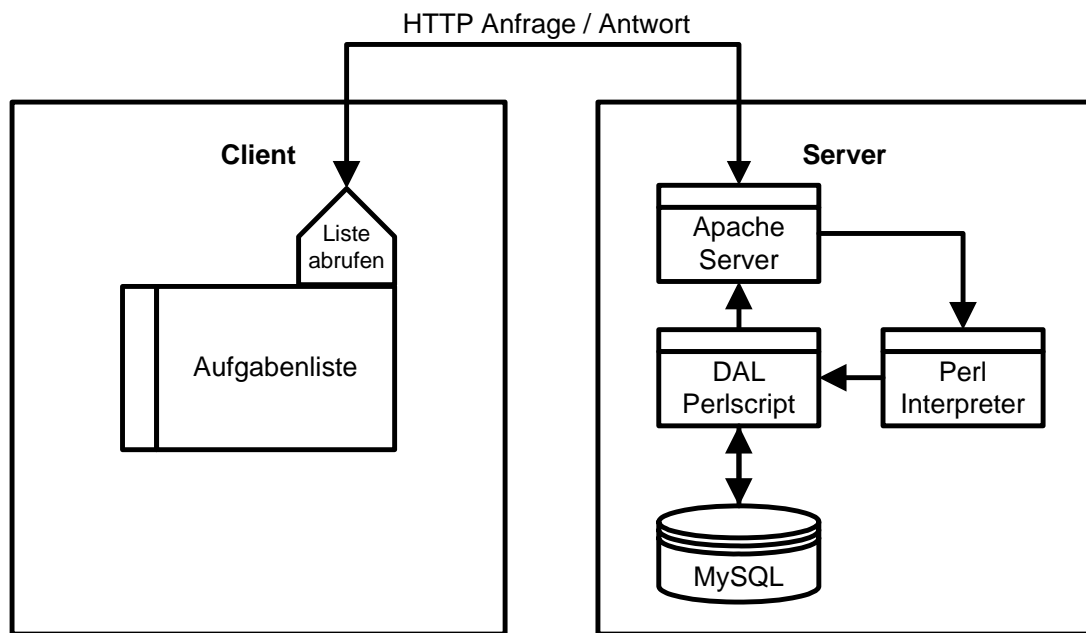
### 4.2.1 Client/Server Umsetzung

Da in AutoEdit Workbook die Möglichkeit bestehen soll neue Aufgaben hinzuzufügen, aber dem Schüler ein ständiges Herunterladen von Übungsdateien erspart werden soll, wurde eine Client / Server Lösung mit Webdatenbank gewählt. In Abbildung 13 werden schematisch die beteiligten Komponenten, am Beispiel des Abrufs der Aufgabenliste, gezeigt. Das in Kapitel 3.2 beschriebene Konzept des Data Access Layers (DAL) wird in Form eines Perlscripts<sup>17</sup> verwirklicht. Der Client stellt eine CGI<sup>18</sup> HTTP Anfrage an den Webserver. Der Apache Server startet den Perlinterpretierer und übergibt diesem das DAL Perlscript. Dieses verarbeitet die Anfrage des Client und greift stellvertretend auf die Datenbank des Webservers zu (hier MySQL). Die Ergebnisse werden über den Apache Server zurück zum Client als HTTP-Antwort gesendet und können nun von AutoEdit verarbeitet werden.

---

<sup>17</sup> Perl ist eine Scriptsprache die häufig auf Webservern eingesetzt wird.

<sup>18</sup> Common Gateway Interface




**Abbildung 13:** AutoEdit Workbook Client / Server Architektur

Die Gründe, warum gerade HTTP als Übertragungsweg genutzt wird, sind technischer Natur. Zum einen ist die Wahl des Ports 80 günstig, da dieser in der Regel an Schulen nicht durch Firewalls oder Ähnliches geblockt wird und zum anderen auf Grund der Restriktion öffentlicher Webhoster<sup>19</sup>, eigene Programme ausschließlich als CGI Befehle ausführen zu dürfen. In Folge dessen ist die entwickelte Lösung auf nahezu allen Webservern ohne großen Aufwand installierbar (Datenbank vorausgesetzt).

#### 4.2.2 Webdatenbank

Das von Webhostern meist angebotene MySQL Datenbanksystem wird in der Regel auf eine Datenbank pro Nutzer beschränkt. Somit müssen alle Tabellen, auch gegebenenfalls unterschiedlicher Projekte, nebeneinander kooperieren. Bei umfangreicher Nutzung der Datenbank wird diese durch die Vielzahl von Tabellen schnell unübersichtlich. Aus diesem Grund und der geringen Datenmenge wurde auf eine vollständige Normalisierung der für AutoEdit Workbook benötigten Tabelle verzichtet. Abbildung 14 zeigt die verwendete Einzeltabelle für die Aufgabenstellungen.

<sup>19</sup> Webhoster sind Internetdiensteanbieter, die Webserver zur Unterbringung von Webseiten anbieten.

AutoLessons		
	ID	<i>Primärschlüssel</i>
	Author	<i>Autor der Aufgabe</i>
	Email	<i>Email des Autors</i>
	CreateDate	<i>Erstellungsdatum der Aufgabe</i>
	Title	<i>Titel der Aufgabe</i>
	Group	<i>Gruppen für Baumdarstellung</i>
	Description	<i>Aufgabenstellung in HTML</i>
	Automaton	<i>Lösungsautomat in XML</i>
	Visible	<i>Legt fest ob diese Aufgabe sichtbar ist</i>

**Abbildung 14:** AutoEdit Workbook MySQL Tabelle

### 4.2.3 Lernaufgaben - Erstellung & Wartung

Zielstellung für AutoEdit Workbook soll eine sofortige und zuverlässige Überprüfung der vom Schüler erstellten Lösung sein. Um die Schülerlösung und die Musterlösung miteinander vergleichen zu können, müssen beide Automaten zunächst minimiert und anschließend die minimierten Automaten verglichen werden. In AutoEdit Workbook können nur Aufgabenstellungen für endliche Automaten erstellt werden. Für andere Automatentypen ist ein Test auf Äquivalenz nicht möglich. Eine Ausweitung auf Kellerautomaten oder Turingmaschinen ist aus diesem Grund nicht geplant. Aus der Musterlösung, die der Aufgabenstellung zu Grunde liegt, werden dem Schüler bei der Bearbeitung Automatentyp und das zu verwendende Alphabet vorgegeben.

Sowohl Lehrer als auch Schüler können mit AutoEdit Workbook Aufgaben erstellen und auf dem Server bereitstellen lassen. Dazu wird ein integriertes Formular verwendet, in dem die relevanten Daten wie Autor, Aufgabenstellung und Musterlösung angegeben werden müssen. Eine Aufgabenstellung kann in HTML verfasst werden und somit Hyperlinks und Bilder enthalten, was verschiedenste Gestaltungsmöglichkeiten erlaubt. Die Musterlösungen werden in Form eines Automaten (als XML Datei) angegeben und mit auf den Server geladen. Um Musterlösungen zu erstellen ist AutoEdit (nicht Workbook) notwendig, denn nur hier können Automaten mit eigenem Alphabet erstellt und gespeichert werden.

Neben den traditionellen endlichen Automaten wurde eine weitere Aufgabenart implementiert. Die so genannten „Chicken Runs“ bieten einen spielerischen Zugang zur Theorie. Vor allem das Experimentieren steht hierbei im Vordergrund. Ein oder mehrere Hühner werden mit Hilfe des konstruierten Automaten auf einem Spielbrett bewegt. An Stelle von Alphabetszeichen an den Übergängen des Automaten, werden Bedingungen wie „Baum voraus“ oder „auf Futter stehen“ zur Steuerung des Automaten verwendet. Das Konzept ist stark an Kara<sup>20</sup> angelehnt. Die bereits in Kapitel 2.2 angesprochene ungünstige Umsetzung des Nichtdeterminismus von Kara, wird bei Chicken Runs mit Hilfe von Cloning berichtigt. Gibt es mehr als eine Entscheidungsmöglichkeit, entstehen mehrere „Hühner“, die als eigenständige Automaten parallel weiter abgearbeitet werden. Chicken-Run-Aufgaben können derzeit nicht über AutoEdit Workbook erstellt werden, sondern müssen ausschließlich manuell vom Entwickler in die Datenbank gebracht werden, da sie eine zusätzliche Information für das Spielbrett benötigen.

#### **4.2.4 Lernaufgaben – Lösungsüberprüfung**

Um die Lösungen des Schülers mit der Referenzlösung des Aufgabenautors zu vergleichen, sind zunächst Transformationen nötig. Im folgenden Codebeispiel wird diese Vorbereitung gezeigt. Später soll Automat A und B miteinander verglichen werden:

```
function TAutomaton.isEqualDEA (Auto : TAutomaton) : Boolean;
var A,B : TAutomaton; s : TMemoryStream; r : Boolean; i,z,w : Integer;
begin
  Auto.XML.SaveToStream(s); A.XML.LoadFromStream(s);
  A.Transformation.TransfromNEAtoDEA;
  A.Transformation.TransfromMinimalDEA;
  This.XML.SaveToStream(s); B.XML.LoadFromStream(s);
  B.Transformation.TransfromNEAtoDEA;
  B.Transformation.TransfromMinimalDEA;
```

---

<sup>20</sup> Siehe [Kara]

Beide Automaten werden zunächst in einen DEA und anschließend in den Minimal-Automat transformiert. Für den Vergleich werden nun alle Zustände, Übergänge und Labels abgelaufen und miteinander verglichen. Wird eine Unstimmigkeit gefunden, wird der Vorgang abgebrochen und ein „false“ zurückgegeben, da die Automaten nicht äquivalent sind. Die Zustandsnamen spielen bei diesem Vergleich keine Rolle, womit der Anwender frei über die Bezeichnung seiner Zustände bestimmen kann.

```
// Anzahl der Zustände muss übereinstimmen
if high(A.Automaton.Statements) <> high(B.Automaton.Statements) then begin
  result := false; A.Free; B.Free; exit;
end;
for i := 0 to high(A.Automaton.Statements) do begin
// Anzahl der Übergänge am Zustand I muss übereinstimmen
  if high(A.Automaton.Statements[i].Transitions) <>
    high(B.Automaton.Statements[i].Transitions) then begin
    result := false; A.Free; B.Free; exit;
  end;
  for z := 0 to high(A.Automaton.Statements[i].Transitions) do begin
// Anzahl der Labels am Übergang Z muss übereinstimmen
    if high(A.Automaton.Statements[i].Transitions[z].Conditions) <>
      high(B.Automaton.Statements[i].Transitions[z].Conditions) then begin
      result := false; A.Free; B.Free; exit;
    end;
    if A.Automaton.Statements[i].Transitions[z].Statement <>
      B.Automaton.Statements[i].Transitions[z].Statement then begin
      result := false; A.Free; B.Free; exit;
    end;
    for w := 0 to high(A.Automaton.Statements[i].Transitions[z].Conditions) do
begin
// Label Inhalt muss übereinstimmen wenn die Anzahl stimmt
      if A.Automaton.Statements[i].Transitions[z].Conditions[w].Value <>
        B.Automaton.Statements[i].Transitions[z].Conditions[w].Value then begin
        result := false; A.Free; B.Free; exit;
      end;
    end; end; end; result := true; A.Free; B.Free; end; // sonst True
```

## **4.3 TDiag**

TDiag wendet sich wie AutoEdit an zwei Zielgruppen. Zum einen an Schüler, denen am Beispiel von T-Diagrammen (siehe Kapitel 1.3) die Arbeitsweise von Compilern veranschaulicht werden kann und zum anderen an Autoren von Lehrmaterialien oder Fachliteratur, die diese T-Diagramme benötigen.

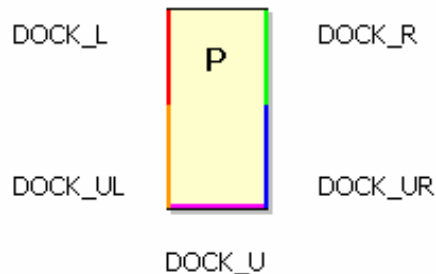
### ***4.3.1 Entwicklung von T-Diagrammen mit TDiag***

T-Diagramme werden in TDiag mittels Drag&Drop zusammengefügt. Von vier verschiedenen Bausteintypen kann in einer Liste gewählt werden (Programm, Ein/Ausgabe, Interpreter und Compiler). Nach dem Platzieren auf dem virtuellen Arbeitsblatt müssen die Eigenschaften des Bausteins vom Benutzer eingestellt oder eine fertige Voreinstellung (bei Interpretern und Compilern) ausgewählt werden. Anschließend können die platzierten Bausteine frei über die Arbeitsfläche verschoben werden, um das gewünschte Diagramm zu erhalten. Unter bestimmten Bedingungen haften Bausteine aneinander (näheres dazu in Kapitel 4.3.2). Im Anhang B ist die Erstellung eines einfachen Diagramms anhand von Bildschirmfotos beschrieben.

### ***4.3.2 Abhängigkeiten unter Diagrammkomponenten***

Ein T-Diagramm hat einen einfachen graphischen Aufbau und kann deshalb auch mit jeder Bildverarbeitung gezeichnet werden. Unberücksichtigt bleiben dort jedoch die vielen Abhängigkeiten der Komponenten, wie diese aneinander passen. Eine Vielzahl von Regeln muss eingehalten werden. Ein C# Compiler kann beispielsweise nur einen C# Quellcode übersetzen, oder ein Java-Interpreter kann keinen Pascalcode interpretieren. Diese und viele weitere Regeln werden in TDiag angewendet, um zu gewährleisten, dass nur korrekte Diagramme vom Nutzer erstellt werden können. Dem Schüler werden somit sofort Fehler ersichtlich. Visuell wird ein Fehler durch Rotfärbung eines Bausteins, oder durch nicht „andocken“ an andere Bausteine angezeigt.

Diagrammbausteine besitzen (wie in Abbildung 15 gezeigt) Andockstellen, an denen sich andere Bausteine anlagern können. Technisch werden diese durch Pointer auf den Basistyp „Baustein“ verwirklicht und durch RTTI (Runtime Type Information) jeweils der tatsächlich anliegende Bausteintyp bestimmt.



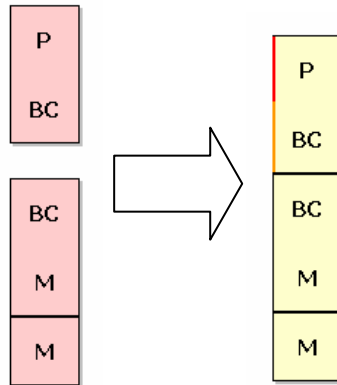
**Abbildung 15:** TDiag Programmbaustein - Andockpointer

Unter der Annahme, dass der in Abbildung 15 gezeigte Programmbaustein nur mit Hilfe eines Interpreters ausgeführt werden kann, wird an DOCK\_U ausschließlich ein Interpreterbaustein andocken können. Andere Komponenten ergeben an dieser Position dann keinen Sinn.

Wie diese Regeln des Andockverhaltens umgesetzt wurden, soll stellvertretend am folgenden Codebeispiel gezeigt werden. Dargestellt ist die Regel für das Dockverhalten zwischen Interpretern und Programmen:

```
for i := 0 to high(Diagram.Interpreters) do begin // Iteriere alle Interpreter
if Diagram.Interpreters[i].DOCK_O=nil then begin // noch freier Interpreter
for z := 0 to high(Diagram.Programs) do // mit allen Programmen prüfen
if (Diagram.Programs[z].DOCK_U = nil)and // noch kein Anschluß unten
(Diagram.Programs[z].RunningOn= "")and // kein lauffähiges Programm
(Diagram.Programs[z].Layout.x = Diagram.Interpreters[i].Layout.x)and
(Diagram.Programs[z].Layout.y = Diagram.Interpreters[i].Layout.y
-2*BlockDefSize)and // Position stimmt
(Diagram.Programs[z].Language= Diagram.Interpreters[i].InputLanguage )
then begin // Andocken wenn alle Bedingungen übereinstimmen
Diagram.Interpreters[i].DOCK_O := Diagram.Programs[z];
Diagram.Programs[z].DOCK_U := Diagram.Interpreters[i];
```

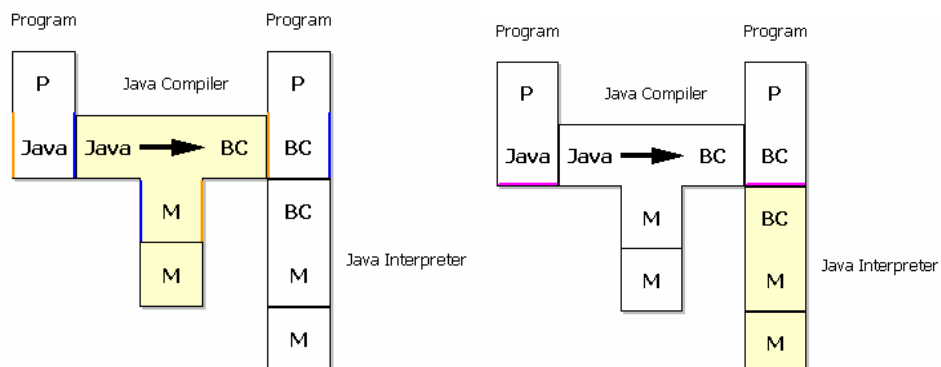
Der Codeausschnitt ist für den in Abbildung 16 gezeigten Fall verantwortlich. Durch Aneinanderschieben der beiden rot gefärbten Komponenten entsteht ein lauffähiges Diagramm. Das in Javabytecode (BC) vorliegende Programm „P“ wird mit Hilfe des Javainterpreters ausgeführt.



**Abbildung 16:** TDiag Andocken eines Interpreters an ein Programm

### 4.3.3 Ausführung/Abarbeitung eines T-Diagramms

T-Diagramme werden in TDiag nicht nur „gezeichnet“. Das Programm kennt die Abhängigkeiten der einzelnen Komponenten, wodurch es möglich wird, ein Diagramm in eine MS-DOS Batchdatei zu transformieren und diese anschließend auszuführen. Dies bietet eine zusätzliche Kontrollmöglichkeit für den Schüler, indem praktisch getestet werden kann, ob das erstellte T-Diagramm tatsächlich den gewünschten Compilerprozess wiedergibt. In Abbildung 17 ist ein Diagramm dargestellt, welches einen Javaquellcode in Javabytecode übersetzt und das Ergebnis anschließend interpretiert.



**Abbildung 17:** TDiag Diagramm Beispiel

Dabei ergeben sich zwei Kommandos für die Batchdatei – zunächst kompilieren und anschließend interpretieren mit Javac.exe bzw. Java.exe. Bereits an diesem kleinen Beispiel wird ersichtlich, dass die Reihenfolge der Abarbeitung eine entscheidende Rolle spielt. TDiag verwendet aus diesem Grund, ähnlich des Lesens der deutschen Sprache, die Regel von links nach rechts bzw. von oben nach unten.

Für das obige Diagramm (Abbildung 17) wird die folgende Batch Datei erzeugt:

```
@ECHO OFF
REM This file was automaticly created by T-Diag
ECHO ;Starting batch file that will execute the diagram.
ECHO ;=====
ECHO Starting: javac.exe testprogram.java
ECHO ...
javac.exe testprogram.java
IF ERRORLEVEL 1 GOTO ERROR
ECHO Done with: javac.exe testprogram.java
ECHO ;=====

ECHO Starting: java.exe -cp "d:\\ samples\\" testprogram
ECHO ...
java.exe -cp "d:\\ samples\\" testprogram
IF ERRORLEVEL 1 GOTO ERROR
ECHO Done with: java.exe -cp "d:\\ samples\\" testprogram
ECHO ;=====

ECHO ;Diagram successfuly executed.
GOTO END

:ERROR
ECHO .
ECHO ;=====
ECHO ;There was an error while executing the diagram
:END
```

Die technische Umsetzung der Batchdateitransformation ist im Anhang C genauer beschrieben. Die Batchdatei wird im selben Verzeichnis und mit gleichem Namen wie die Diagrammdatei abgelegt, aber trägt die Endung „.bat“. Diese Batchdatei ist anschließend auch ohne TDiag verwendbar und kann jederzeit eigenständig gestartet werden, um erneut den Compilerprozess abzuarbeiten. Die korrekte Fehlerbehandlung innerhalb der Batchdatei basiert auf das ERRORLEVEL der ausgeführten Anwendung. Standardsoftware wie Java oder C# Compiler halten sich an die Regelung, eine „1“ bei Fehler zurückzugeben (bei eigenen Programmen ist dies der Wert, der von der Funktion „int main (...)“ zurückgegeben wird). Einige Programme hingegen verhalten sich hier nicht so (z.B.: Chez Scheme). In Folge dessen wird die Batchdatei auch bei einem Fehler nicht an der betreffenden Stelle abgebrochen. Dies sollte bei der Verwendung der generierten Dateien berücksichtigt werden, wenn die ECHO-Ausgaben als Debuginformationen genutzt werden.

#### **4.3.4 Speicherformat von TDiag**

Die Struktur einer TDiag XML Datei ist mit einer Liste zu vergleichen. Zunächst werden alle verwendeten Sprachen eingetragen, anschließend die Programmbausteine, Interpreter, Compiler, Ein/Ausgabe und schlussendlich die Kommentare. Jeder Baustein besitzt eine eindeutige ID (Nummer). Wie in Abschnitt 4.3.2 gezeigt, sind die Andockvariablen DOCK\_U, DOCK\_R, ... Pointer auf andere Bausteine. Diese können mit Hilfe der ID und einer eindeutigen Zuordnung (Baustein A kennt die Beziehung zu B und B umgekehrt auch die Beziehung zu A) wiedergegeben werden. Das verwendete XML Schema ist im Anhang A zu finden. Eine stark gekürzte Beispieldatei für ein T-Diagramm (in Fett wurden die ID's und deren Querverweise hervorgehoben), welches ein Java Programm und einen Java Compiler enthält:

```
<TDIAGRAM>
  <LANGUAGES>
    <LANGUAGE value="Java" label="Java" defaulttextextension=".java"/>
    <LANGUAGE value="Java Bytecode" label="BC" defaulttextextension=".class"/>
    <LANGUAGE value="Binary Executables" label="M"
defaulttextextension=".exe;.com"/>
  </LANGUAGES>
  <PROGRAMS>
    <PROGRAM value="1">
      <FILENAME value="testprogram.java"/>
      <RUNNINGON value=""/>
      <LABELTEXT value="P"/>
      <LANGUAGE value="Java"/>
      <LAYOUT><X value="132"/><Y value="60"/></LAYOUT>
      <DOCK_UR value="0"/>
    </PROGRAM>
    ...
  </PROGRAMS>
  <INTERPRETERS>
    ...
  </INTERPRETERS>
  <COMPILERS>
    <COMPILER value="0">
      <NAME value="Java Compiler"/>
      <FILENAME value="javac.exe %InFile"/>
      <RUNNINGON value="M"/>
      <OUTPUTFORMAT value="%InFileNoExt.class"/>
      <INPUTLANGUAGE value="Java"/>
      <OUTPUTLANGUAGE value="Java Bytecode"/>
      <WRITTENIN value="Binary Executables"/>
      <LAYOUT><X value="180"/><Y value="108"/></LAYOUT>
      <DOCK_L value="1"/>
      <DOCK_R value="2"/>
    </COMPILER>
  </COMPILERS>
  <EAS/>
  <COMMENTS/>
</TDIAGRAM>
```

## 4.4 Visual Compiler Compiler

Die Entwicklung des Visual Compiler Compiler (kurz VCC) wurde bereits 2002 im Rahmen einer Projektarbeit im Fach Sprachübersetzer begonnen. Das ursprüngliche Ziel lag in der Schaffung einer einheitlichen, graphisch ansprechenden, Softwarelösung für Scanner- und Parsergenerator für die Zielsprache Scheme. Über die vergangenen drei Jahre wurde VCC weiterentwickelt und bietet nun auch die alternative Zielsprache C# an. Die von VCC erstellten Compiler streben dabei weniger eine hohe Effizienz, sondern vielmehr eine für den Schüler einfache Entwicklung an.

### 4.4.1 Scanner

Wie bereits in Kapitel 1.3 beschrieben, besteht die erste Arbeitsphase eines Compilers in der lexikalischen Analyse des Eingabetextes. Durch die Verwendung von Pattern (Schablonen) werden Gruppen von Zeichen zu so genannten Token zusammengefasst. Ein Beispiel für Eingabetext und einige Tokenpattern (als reguläre Ausdrücke):

Eingabetext:

(43 + 12) \* 3

Token:

[0-9]+ → Zahl

[\+\-\\*\^/] → Rechenzeichen ; „\“ als Escapezeichen für Sondersymbole

( → KlammerAuf

) → KlammerZu

Aus dem Eingabetext werden nun Token-Value-Paare erstellt, die später im Parser für die Syntaxanalyse verwendet werden. Für obiges Beispiel könnte das Ergebnis (als Scheme Paar-Liste) wie folgt aussehen:

```
( (KlammerAuf . "(") (Zahl . "43") (Rechenzeichen . "+")  
  (Zahl . "12") (KlammerZu . ")") (Rechenzeichen . "*") (Zahl . "3") )
```

Die Verwendung von regulären Ausdrücken hat sich als besonders günstig erwiesen, da mit diesen bereits komplexe Konstruktionen von Zeichenklassen oder Alternativen möglich sind. In der ersten Version von VCC wurde noch eine eigene Patternsprache verwendet, die aber ausschließlich für primitive Token angewendet werden konnte. Bereits mit der zweiten Version wurde diese durch reguläre Ausdrücke ersetzt, womit sich VCC noch mehr an die Funktionsweise von LEX annäherte.

VCC bietet die Möglichkeit den Scanner in einer visuellen Oberfläche zu entwerfen. Funktionen, wie das Einfärben von Token in verschiedene Farben, können später bei der Konstruktion des Parsers zur besseren Lesbarkeit beitragen. In Abbildung 18 ist ein Screenshot von VCC (Scanneransicht) gezeigt.

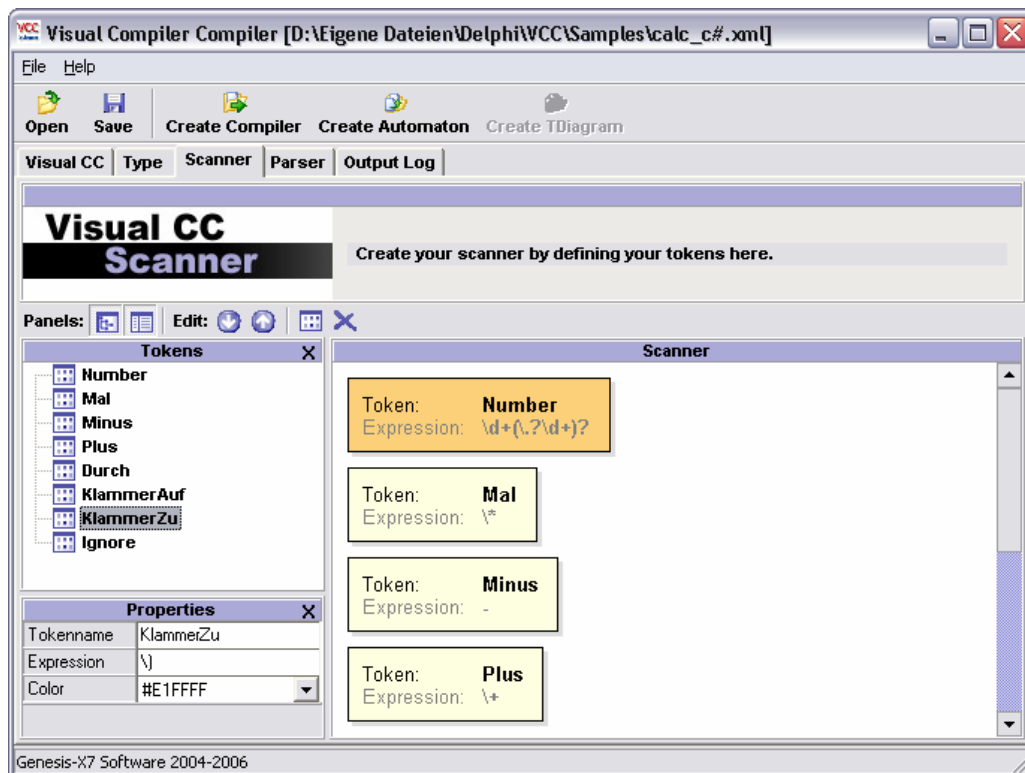
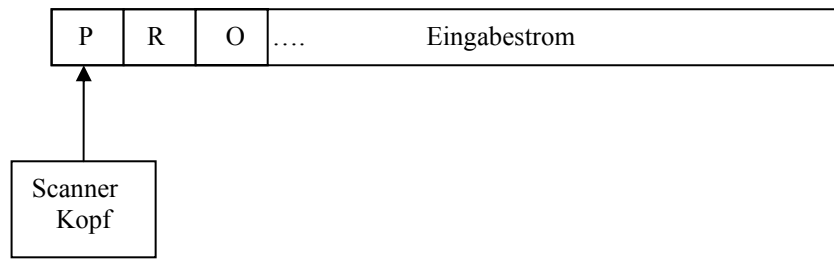


Abbildung 18: VCC Scanner Beispiel

Die Arbeitsweise der lexikalischen Analyse ist nahezu äquivalent zu LEX. Mit wenigen Regeln wird bestimmt, welches Token ausgewählt wird. Der Eingabetext kann als langes Band verstanden werden, auf dem die einzelnen Zeichen geschrieben stehen. Dies wird in Abbildung 19 schematisch dargestellt.



**Abbildung 19:** VCC Scanner Arbeitsweise

In jedem Arbeitsschritt werden alle Token-Pattern auf das Band (ausgehend von der Scannerkopfposition) angewendet und geprüft, ob diese passen. Im Verlauf dieser Überprüfung werden alle als passend gefundenen Pattern in eine Liste aufgenommen und jeweils deren erkannte Wortlänge vermerkt. Anschließend werden die Token herausgesucht, die die maximale Länge aufweisen. Sollte mehr als ein Token diesem Kriterium entsprechen, wird sich für das Token entschieden, welches in der Tokenliste (in VCC in der Baumdarstellung links zu erkennen) am höchsten steht.

#### 4.4.2 *Parser*

Der Parser übernimmt in VCC die Arbeitsschritte: Syntaxanalyse und Codegeneration. Bei der Entwicklung eines Parsers sollte zunächst eine Grammatik erstellt werden, die anschließend in VCC übertragen werden kann. Eine Grammatik enthält Terminale und Nichtterminale. Die Terminale sind äquivalent zu den bereits in 4.4.1 beschriebenen Token. Die Nichtterminale werden in VCC durch Regeln wiedergegeben, die wiederum der BNF Form<sup>21</sup> der Grammatik entsprechen. Ein Beispiel für eine Grammatik in dieser Form:

Eingabe	→	Eingabe Ausdruck   $\epsilon$
Ausdruck	→	Ausdruck + Ausdruck   Ausdruck - Ausdruck   Ausdruck * Ausdruck   Ausdruck / Ausdruck   ( Ausdruck )   Zahl
Zahl	→	Zahl Ziffer   Ziffer
Ziffer	→	1   2   3   4   5   6   7   8   9   0

<sup>21</sup> Backus-Naur-Form - eine Notationsform kontextfreier Grammatiken

Durch die Verwendung von Zeichenklassen im Scanner können bereits Nichtterminale wie Zahl / Ziffer durch ein Token mit „[0-9]+“ als Pattern zu Terminalen aufgelöst werden. Dadurch vereinfacht sich die Parserkonstruktion, da weniger Regeln nötig sind.

In VCC wird der Parser graphisch entworfen. Abbildung 20 zeigt in etwa die Umsetzung der obigen Beispielgrammatik.

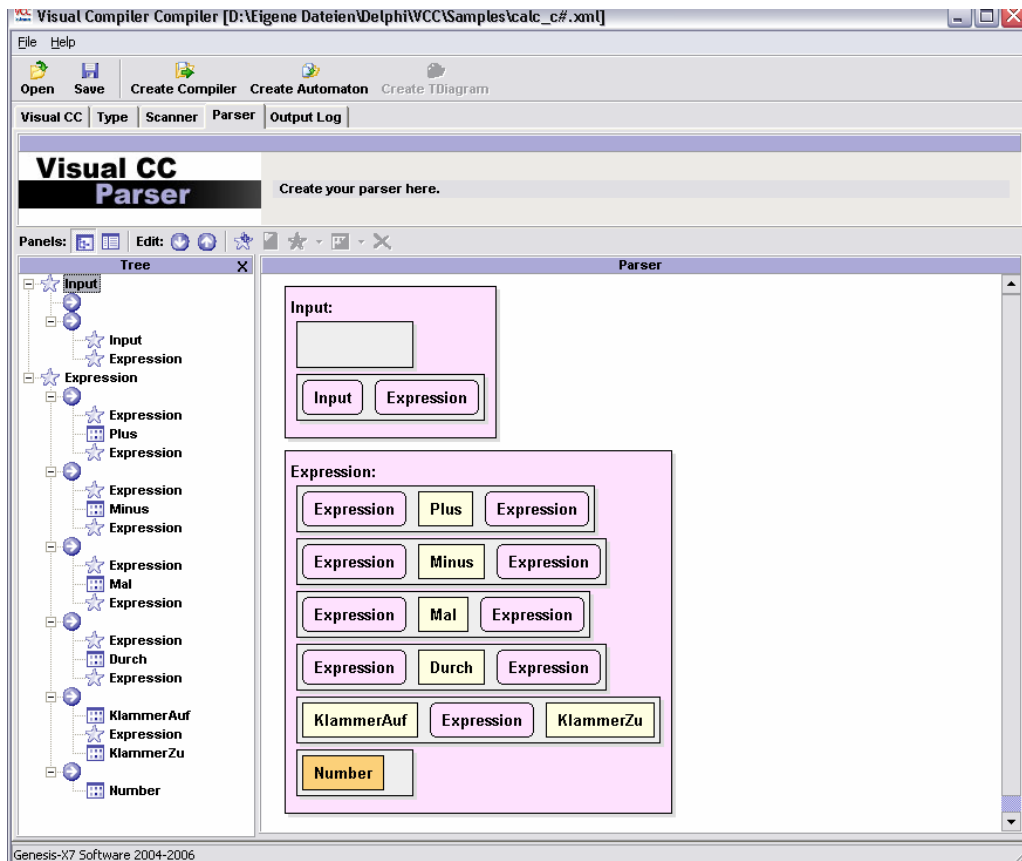


Abbildung 20: VCC Parser Beispiel

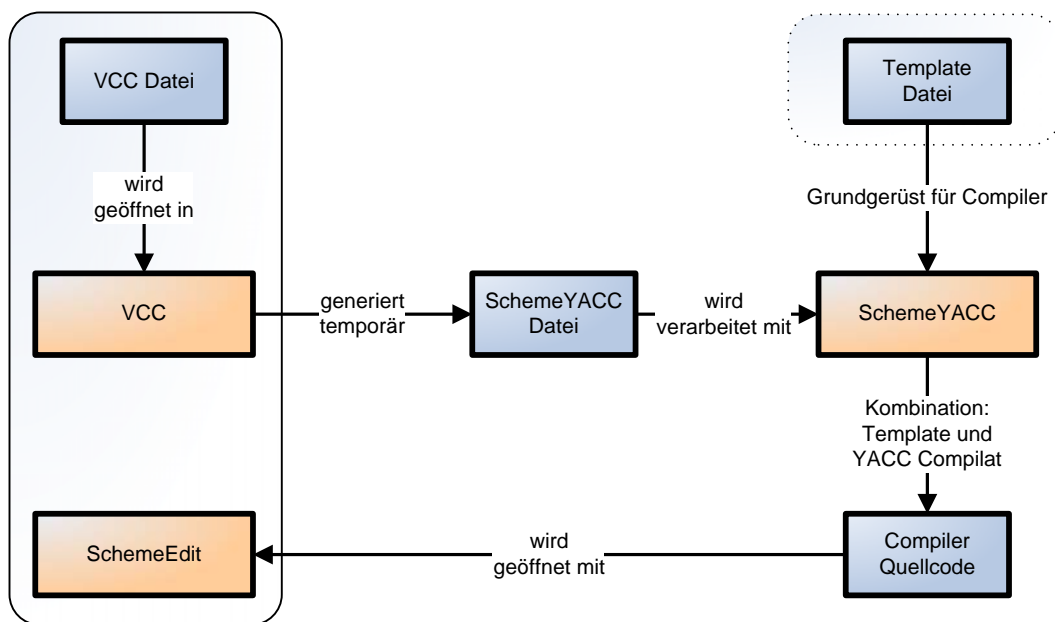
Die Regeln werden dabei abgerundet und die Token als Rechtecke dargestellt. Die bei der Scannerentwicklung vergebenen Farben der Token (wie hier das Token „Number“) werden bei dieser Darstellung wieder aufgegriffen.

Gegenüber der herkömmlichen Erstellung der Parserregeln in einer Textdatei (z.B.: YACC) können in VCC keine Schreibfehler entstehen, da sowohl Token als auch Regeln ausschließlich durch Menüs ausgewählt werden können und somit die manuell Eingabe entfällt. Auf Grund der Fehleranfälligkeit wurde in VCC auf eine Copy & Paste Möglichkeit bei der Erstellung des Parsers verzichtet. Gerade die

durch Copy & Paste entstehenden Fehler können in größeren Projekten nur schwer gefunden und ausgebessert werden.

#### 4.4.3 Übersetzung in einen Compiler Quellcode

Der eigentliche Übersetzungsvorgang von einer VCC Quelldatei (in XML) zum fertigen Compiler Quellcode erfolgt in mehreren Teilschritten. In Abbildung 21 ist dieser Prozess dargestellt.



**Abbildung 21:** VCC Kompilationsprozess

Der Großteil dieses Prozesses findet im Hintergrund statt. Der Anwender sieht deshalb nur den linken, umrandeten Teil (Abbildung 21). Eine Ausnahme bildet das „Template“. Diese Datei beinhaltet Scheme-Code für den späteren Compiler. Während der Abarbeitung von SchemeYACC wird der generierte Parser-Kellerautomat und das Template zu einem fertigen Compiler zusammengeführt. Im Template findet man beispielsweise Hilfsfunktionen, die für jeden Compiler gleichermaßen verwendet werden. Im Normalfall kann man die von VCC mitgelieferte Template-Datei einsetzen. Besteht der Wunsch bestimmte Bibliotheken oder zusätzliche Hilfsfunktionen standardmäßig aufzunehmen, kann diese Datei entsprechend editiert werden.

#### 4.4.4 *SchemeYACC*

Wie bereits im vorangehenden Abschnitt in Abbildung 21 gezeigt, wird im Hintergrund das Hilfsprogramm SchemeYACC verwendet. Es handelt sich dabei um eine eigens für VCC adaptierte Variante von TP-YACC<sup>22</sup>. Die ursprüngliche Ausgabesprache Turbo Pascal wurde durch Scheme bzw. C# ersetzt. Die SchemeYACC Quelldateien (welche temporär erzeugt werden) besitzen den gleichen syntaktischen Aufbau, wie die des original YACC Programms von 1979.

SchemeYACC ist ein Kommandozeilenprogramm und kann durch einen zusätzlichen Parameter auch C# Quellcode erzeugen, wobei aber ein anderes Template verwendet wird, welches in C# geschrieben ist. Diese Funktionalität ist erst zu einem späteren Zeitpunkt in das VCC Projekt aufgenommen worden, weshalb dennoch das Programm den Titel „SchemeYACC“ behalten hat.

#### 4.4.5 *VCC Speicherformat*

Im Gegensatz zu LEX und YACC wird in VCC die Definition von Scanner und Parser in einer Datei gespeichert. Damit entfällt das Anpassen der YACC-Datei, wenn in der LEX-Datei Änderungen vorgenommen werden. VCC speichert, wie die anderen Komponenten von AtoCC, seine Daten in einer XML-Datei ab. Diese enthält am Anfang die Tokenliste mit den entsprechenden Pattern des Scanners. Gefolgt wird diese vom globalen Quellcode, der mit in den vorliegenden Ausgabecompiler aufgenommen werden soll (etwa für Hilfsfunktionen oder Variabeldefinitionen). Am Ende schließen sich die Regeln und der Aktionsquellcode des Parsers an<sup>23</sup>.

Der nachfolgende Auszug aus einer solchen XML-Datei zeigt den beschriebenen Aufbau. Es handelt sich dabei um einen einfachen Taschenrechner-Compiler. Die in diesem Fragment enthaltene Regel steht für die Addition zweier Ausdrücke. Ebenfalls gut zu erkennen sind die Pattern für den Scanner als reguläre Ausdrücke (entsprechend mit Sonderzeichenbehandlung durch Backslashes).

---

<sup>22</sup> Turbo Pascal YACC von Albert Graef (April 2000)

<sup>23</sup> Vgl. XML Schema in Anhang A

```
<VCC>
  <CODETYPE value="Scheme"/>
  <SCANNER>
    <READ value="\d+(\.?\d+)?" token="Number" color="#E1FFFF"/>
    <READ value="\*" token="Mal" color="#E1FFFF"/>
    <READ value="-" token="Minus" color="#E1FFFF"/>
    <READ value="\+" token="Plus" color="#E1FFFF"/>
    <READ value="/" token="Durch" color="#E1FFFF"/>
    <READ value="\(" token="KlammerAuf" color="#E1FFFF"/>
    <READ value="\)" token="KlammerZu" color="#E1FFFF"/>
    <READ value="\s|\n|\r|\t" token="Ignore" color="#E1FFFF"/>
  </SCANNER>
  <GLOBALCODE>
    (define Variable1 0) ; Beispiel für globalen Quellcode
  </GLOBALCODE>
  <RULES>
    ...
    <RULE name="Expression" color="#FFE1FF">
      <RIGHTSIDE>
        <RULE name="Expression"/>
        <TOKEN name="Plus"/>
        <RULE name="Expression"/>
        <CODE>
          (set! $$ (number->String
            (+ (string->number $1) (string->number $3))))
        </CODE>
      </RIGHTSIDE>
    </RULE>
    ...
  </RULES>
</VCC>
```

## 4.5 SchemeEdit

Für die in AtoCC verwendete Programmiersprache Scheme gibt es eine Vielzahl von Interpretern. Der Großteil von diesen wird als Freeware angeboten, was den Einsatz an Bildungseinrichtungen begünstigt. Nachteil ist jedoch, dass die einzelnen Scheme- Implementationen nicht immer 100% kompatibel zueinander sind. Für AtoCC fiel die Wahl auf Petite Chez Scheme<sup>24</sup>, da dies an der Hochschule Zittau / Görlitz schon seit längerem erfolgreich eingesetzt wird.

### 4.5.1 *SchemeEdit eine Entwicklungsumgebung für Scheme*

Petite Chez Scheme ist ein Kommandozeileninterpreter. Eine einfache graphische Entwicklungsumgebung, zum Editieren und Ausführen von Scheme-Quellcode, ist im Chez Scheme Softwarepaket bereits vorhanden – die SWL (**S**cheme **W**idget **L**ibrary). Die SWL bietet jedoch nicht mehr als einen einfachen Texteditor und ist auf Grund der Plattformunabhängigkeit in Tastaturbelegung und Steuerelementausrichtungen nicht dem Windows Standard angepasst (etwa Scrollleiste auf der linken Seite oder statt Strg+C für Copy die Tastenkombination Alt+C), was aus eigener Erfahrung schnell frustriert. Aus diesem Grund wurde SchemeEdit entwickelt (Abbildung 22 zeigt einen Screenshot). Funktionen wie Syntax Highlighting, Klammer Highlighting, Zeilennummern oder ein Funktionsbaum erleichtern das Lesen und Navigieren im Quellcode.

Ähnlich wie bei AutoEdit Workbook wird auch für SchemeEdit eine Webdatenbank verwendet. Ebenfalls über einen DAL angesprochen, bietet diese Datenbank Scheme Quellcodefragmente an, welche vom Benutzer in sein aktuelles Projekt aufgenommen werden können. Ziel einer solchen Repository ist es, häufig anfallenden Quellcode der gesamten Benutzergemeinde online zur Verfügung zu stellen.

---

<sup>24</sup> Kostenlos unter [www.scheme.com](http://www.scheme.com) (derzeit Version 7.0a).

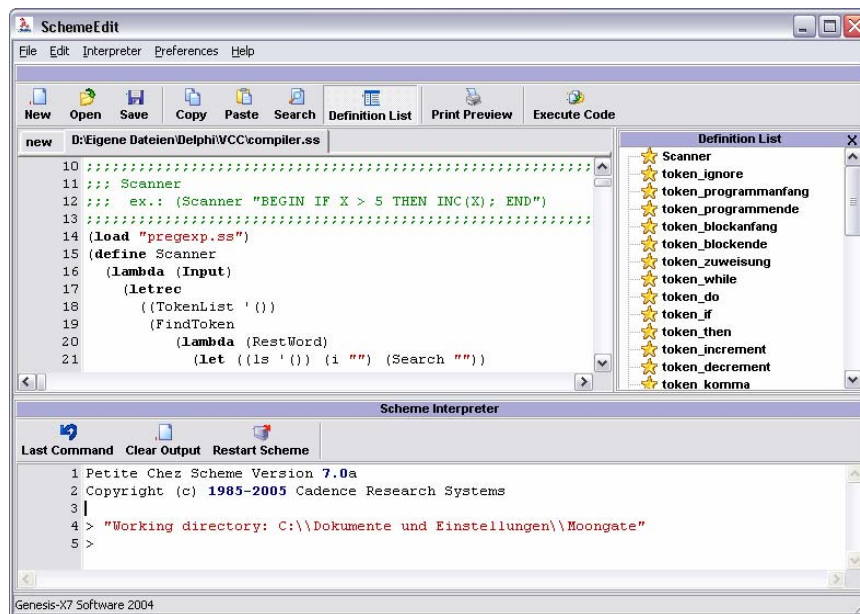


Abbildung 22: SchemeEdit Screenshot

#### 4.5.2 *Scheme Interpreter und damit verbundene Probleme*

Leider sind auch Probleme mit Petite Chez Scheme zu nennen, die bei der Entwicklung von SchemeEdit nicht vorhersehbar waren. Im Laufe der Weiterentwicklung von Chez Scheme änderten sich mit jeder Version die Startparameter des Interpreters und deren Windows Registry Einträge. Somit muss bis heute jede neue Chez Scheme Version an SchemeEdit angepasst werden. Das geschieht in der Regel durch den Entwickler in einem Update von SchemeEdit oder bei Bedarf durch den Nutzer über das Interpretereinstellungsmenü.

Ein anderes Problem ist die Dateiverarbeitung von Chez Scheme. Umlaute und Sonderzeichen können hier zu Problemen führen, was sich vor allem bei der Entwicklung von Compilern (mit VCC entwickelte Scanner) negativ auswirken kann. Ein Umformatieren der Eingabedateien in Unicode Text schafft teilweise Abhilfe. Schüler suchen aber erfahrungsgemäß diesen Fehler im eigenen Quellcode und nicht im Chez Scheme Interpreter und dessen Dateiverarbeitung.

## **5 Erprobung, Ergebnisse und Ausblick**

In diesem Kapitel sollen die bisherigen Erfahrungen mit AtoCC (seinen Komponenten) aufgezeigt werden. Den hier vorgestellten Ergebnissen liegen bislang noch keine konkreten empirischen Studien zu Grunde. Diese sollen mit dem nächsten Semester folgen.

### **5.1 SchemeEdit in der Praxis**

SchemeEdit ist die älteste Komponente von AtoCC und wird bereits seit über zwei Jahren eingesetzt. Die Informatikstudenten an der Hochschule Zittau / Görlitz haben in diesem Zeitraum zur Verbesserung des Projekts beigetragen, indem konstruktive Kritik gegeben oder auch Programmfehler aufgezeigt werden konnten. Die in SchemeEdit enthaltene Online Repository wurde jedoch kaum verwendet und auch nicht durch Schüler erweitert. Der Grund hierfür ist vermutlich, dass noch ein anderes webbasiertes Scheme Repository Projekt parallel entstand und dies vorwiegend verwendet wurde.

SchemeEdit hat sich erfolgreich als Alternative zur SWL bewährt. Für bestimmte Aufgabenstellungen wurde aber auch eine andere Scheme-Implementation (DrScheme) verwendet, welche ebenfalls einen ausgereiften Editor besitzt.

### **5.2 Erfahrungen mit AutoEdit**

AutoEdit wurde im Verlauf des letzten Jahres immer wieder verbessert (neue Funktionen hinzugefügt, Fehler entfernt). Im September 2005 wurde das Projekt erstmals auf der INFOS'05<sup>25</sup> der Öffentlichkeit vorgestellt. Durch den eingetragenen Querverweis in Wikipedia.de wurde AutoEdit inzwischen auch von vielen Studenten in ganz Deutschland und darüber hinaus heruntergeladen (ersichtlich durch Statistiken des Serverprogramms Webalizer, welches auf dem Downloadserver eingesetzt wird).

---

<sup>25</sup> Vgl. [Hielscher 2005]

Durch Emails von Nutzern mit Anregungen und Danksagungen wurde auch aufgezeigt, dass Studenten mit den Simulationswerkzeugen für Automaten, die an ihren Schulen / Universitäten eingesetzt werden, weniger zufrieden sind.

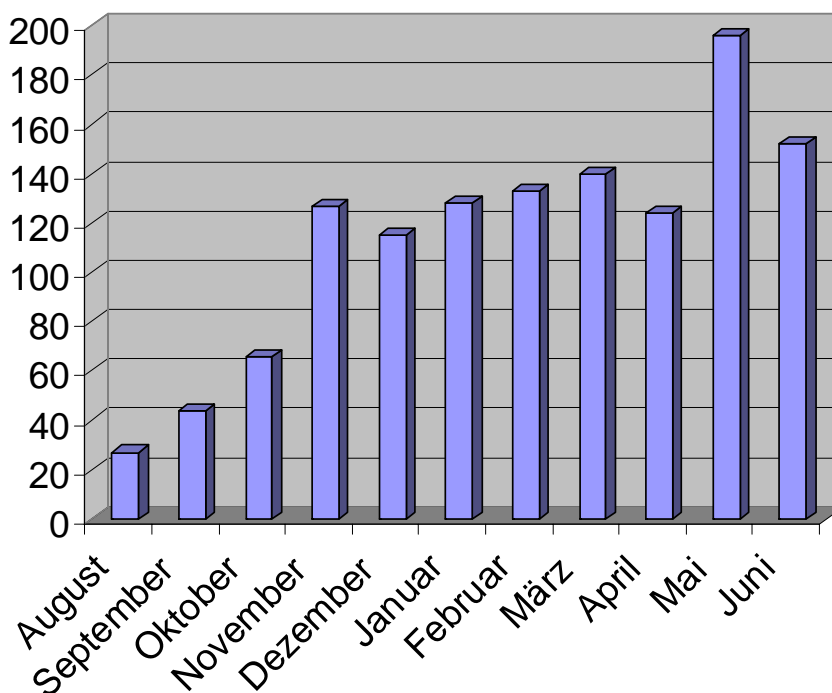
Beispiel Zitat:

*„... wir hier an der Fachhochschule verwenden das Programm "Automaten-IDE" von Jan Brückner (zum Teil wesentlich komplizierter). Da finde ich die Drag'n'Drop-Variante von euch wesentlich besser. Wirklich ein Spitzenprogramm!“  
[Auszug aus einer Email von Marcus Hula – Informatik Student an der Fachhochschule Schmalkalden]*

Eine einfache Handhabung und intuitive Bedienung wird von befragten Anwendern immer wieder hervorgehoben. Dies ist besonders erfreulich, da genau dies ein Hauptziel von AtoCC darstellt.

Für AutoEdit war es möglich, über das vergangene Jahr hinweg, eine Downloadstatistik zu führen (siehe Abbildung 23). Gerade im Vergleich zu den Anfangsmonaten sind die Zahlen stetig gewachsen.

#### Downloads AutoEdit



**Abbildung 23:** AutoEdit Downloadstatistik

AutoEdit Workbook ist bislang noch unerprobt und es liegen noch keine Erfahrungsberichte vor.

### 5.3 Einsatz von VCC

VCC wurde an der Hochschule Zittau / Görlitz bereits von zwei Matrikeln in ihren Belegarbeiten im Fach Sprachübersetzer verwendet. Dabei wurden alle Arbeiten bis auf eine Ausnahme mit der Zielsprache Scheme angefertigt. Studenten fiel dabei die langsame Arbeitsweise von regulären Ausdrücken im Scanner negativ auf, die gerade bei längeren Eingabedateien mehrere Minuten in Anspruch nehmen kann. Verantwortlich hierfür ist die verwendete Bibliothek „Portable regular expressions for Scheme“ von Dorai Sitaram. Leider gibt es derzeit keine alternative Implementation für Chez Scheme mit vergleichbarem Funktionsumfang.

Abgesehen von diesem Problem wurde VCC aber durchweg positiv bewertet (in den Belegarbeiten wurde um eine kurze Einschätzung zu VCC gebeten). Eine Gruppe bildete eine Ausnahme. Hier wurde die Komplexität der Thematik verkannt und ohne theoretische Überlegungen (aufstellen einer Grammatik) versucht, einen graphischen Entwurf mit VCC zu erstellen. Die Folge war ein sehr unübersichtliche Grammatik. Dies wurde von der Gruppe auf das Programm VCC geschoben. Aus diesem Grund wurde ein umfangreiches Tutorial erstellt, welches den Einstieg in den Compilerbau mit VCC erleichtern soll und die Schritte der Entwicklung genauer beschreibt. Dieses Tutorial (30 Seiten PDF + Quelldateien) wird mit VCC ausgeliefert.

### 5.4 Ausblick

Das Gesamtprojekt AtoCC wurde auf der ITiCSE'06 vorgestellt<sup>26</sup> und ist seit einem Monat über das Internet erhältlich. Im Wintersemester 2006 wird AtoCC die Studenten im Fach „Formale Sprachen und Automaten“ begleiten. Dabei wird mit einer Evaluation versucht werden, den Erfolg des Projekts besser einzuschätzen. Umfangreichere Anleitungen und weitere Tutorials sollen ebenfalls noch im Verlauf der nächsten Monate entstehen, um den Studenten zusätzlich den Einstieg zu erleichtern. Ziel für AtoCC bleibt auch in Zukunft, die Qualität des Projektes bei Bedarf mit regelmäßigen Patches zu sichern.

---

<sup>26</sup> Vgl. [Hielscher 2006]

## Literaturverzeichnis

- [Abdullah 2002]** ABDULLAH RAYAN: *Corporate Design. Kosten und Nutzen.*  
Mainz: Schmidt, 2002
- [ANTLR]** *ANother Tool for Language Recognition*  
URL: <http://www.antlr.org/> (2006-05-19)
- [Coco/R]** *The Compiler Generator Coco/R*  
URL: <http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/> (2006-05-19)
- [CDBund]** *Corporate Design Dokumentation der Bundesregierung*  
URL: <http://styleguide.bundesregierung.de> (2006-05-26)
- [Ding-Zhu 2001]** DING-ZHU DU, KER-I KO: *Problem solving in Automata, Languages, and Complexity.*  
New York: John Wiley & Sons Inc, 2001
- [Exorciser]** *Exorciser - Automatisches Generieren und Korrigieren von strukturierten Übungen zur theoretischen Informatik* – URL: <http://www.swisseduc.ch/informatik/exorciser/index.html>
- [FLAT]** *FLAT-Werkzeuge* - URL: <http://www.inf.hs-zigr.de/~wagenkn/TI/Automaten/FLAT-Software> (2006-05-19)
- [Hielscher 2006]** HIELSCHER M., WAGENKNECHT CHR.: *AtoCC - Learning Environment for Teaching Theory of Automata and Formal Languages.* ITiCSE'06, June 26-28, 2006, Bologna, Italy.  
ACM 1-59593-055-8/06/0006

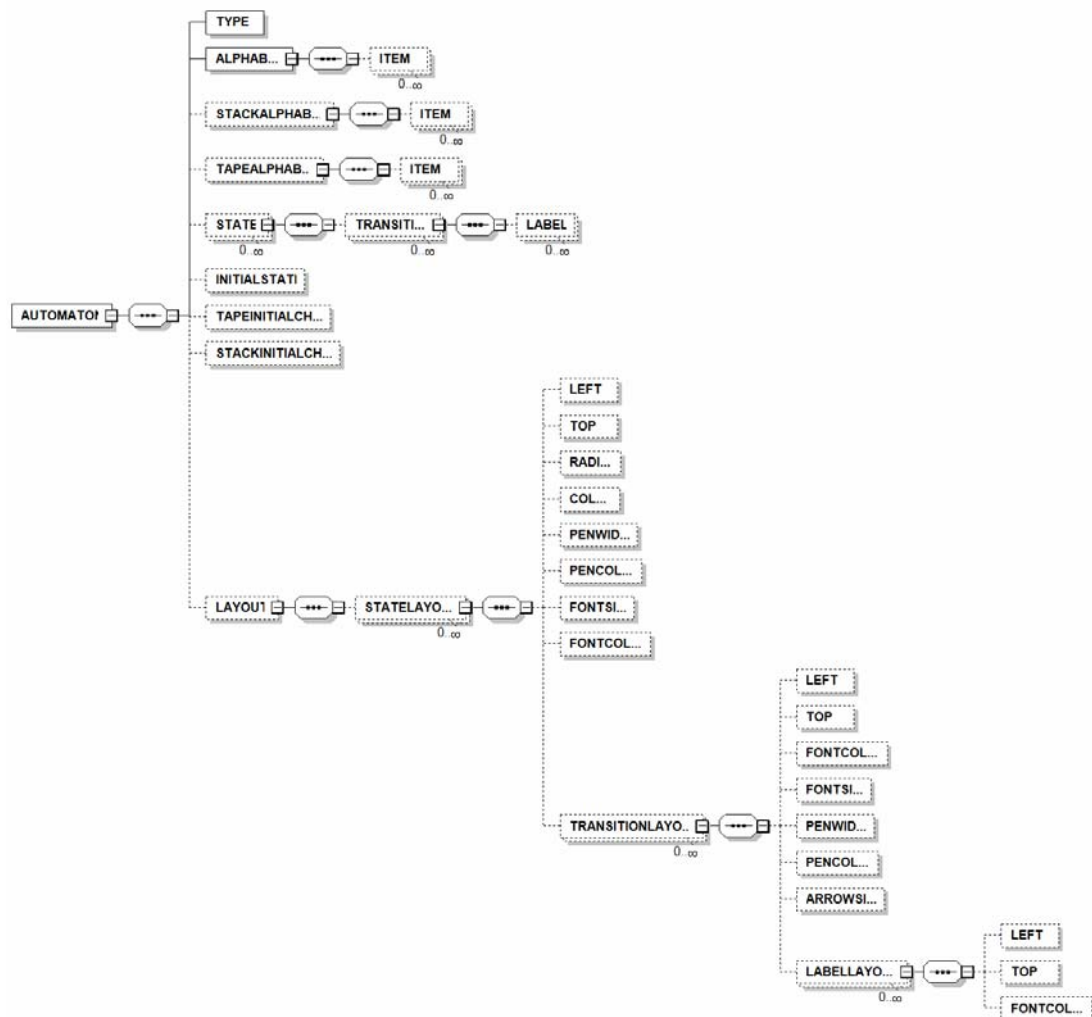
- [Hielscher 2005]** HIELSCHER M., WAGENKNECHT CHR.: *AutoEdit - ein Werkzeug zum Editieren, Simulieren, Transformieren und Publizieren abstrakter Automaten*. In: Unterrichtskonzepte für informatische Bildung INFOS'05 28.-30.09.05 in Dresden - Praxisband (Hrsg.: H. Rohland), S. 25-27
- [Hopcroft 1997]** HOPCROFT JOHN E., ULLMAN JEFFREY D.: *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*. 3. Auflage, München : Oldenbourg, 1997
- [Hopcroft 2001]** HOPCROFT JOHN E., MOTWANI RAJEEV, ULLMAN JEFFREY D.: *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley Publishing Company 2001
- [Högn 2006]** HÖGN OLIVER: *Medienadäquate Aufbereitung und Rezeption von Lerninhalten*. URL: <http://www.unikat.at/enzi/lernen/hoegn2.html> (2006-05-19)
- [JavaCC]** *JavaCC - a parser/scanner generator for java*  
URL: <https://javacc.dev.java.net/> (2006-05-19)
- [JFLAP]** *Java Formal Languages and Automata Package*  
URL: <http://www.jflap.org/> (2006-05-19)
- [Kara]** *Programmieren lernen mit Kara – URL:*  
<http://www.swisseduc.ch/informatik/karatojava> (2006-05-19)
- [LP Hessen]** HESSISCHES KULTUSMINISTERIUM: *Lehrplan Informatik – Gymnasialer Bildungsgang Jahrgangsstufe 11 bis 13 – URL:*  
<http://lernarchiv.bildung.hessen.de/archiv/lehrplaene/gymnasium/informatik/LPGymInformatik.pdf> (2006-05-18)

- [LP Sachsen]** SÄCHSISCHES STAATSMINISTERIUM FÜR KULTUS:  
*Lehrplan Gymnasium Informatik Klassenstufen 7 und 8  
Jahrgangsstufen 11 und 12* - URL: [http://dil.inf.tu-dresden.de/sf2/hs\\_dida\\_gym/materialien/lp\\_gy\\_informatik\\_veroeff.pdf](http://dil.inf.tu-dresden.de/sf2/hs_dida_gym/materialien/lp_gy_informatik_veroeff.pdf)  
(2006-05-18)
- [Sander 1992]** SANDER, STUCKY, HERSCHEL: *Automaten, Sprachen, Berechenbarkeit*. Stuttgart: B. G. Teubner 1992
- [Wagenknecht 2001]** WAGENKNECHT CHRISTIAN: *Theoretische Informatik mit Scheme – Teil 2: Sprachübersetzer*. URL: <http://www.inf.hs-zigr.de/~wagenkn/TI/Berechenbarkeit/Skript/Sprachuebersetzer.pdf> (2006-05-19)
- [Wagenknecht 2006]** WAGENKNECHT CHRISTIAN: *Theorie der formalen Sprachen und Automaten*. URL: <http://www.inf.hs-zigr.de/~wagenkn/TI/Automaten> (2006-05-19)
- [Wirth 1996]** WIRTH NIKLAUS: *Compiler Construction*. Addison Wesley 1996
- [XML 2006]** LAURENT SIMON SAINT: *XML kurz und gut*. O'Reilly 2006

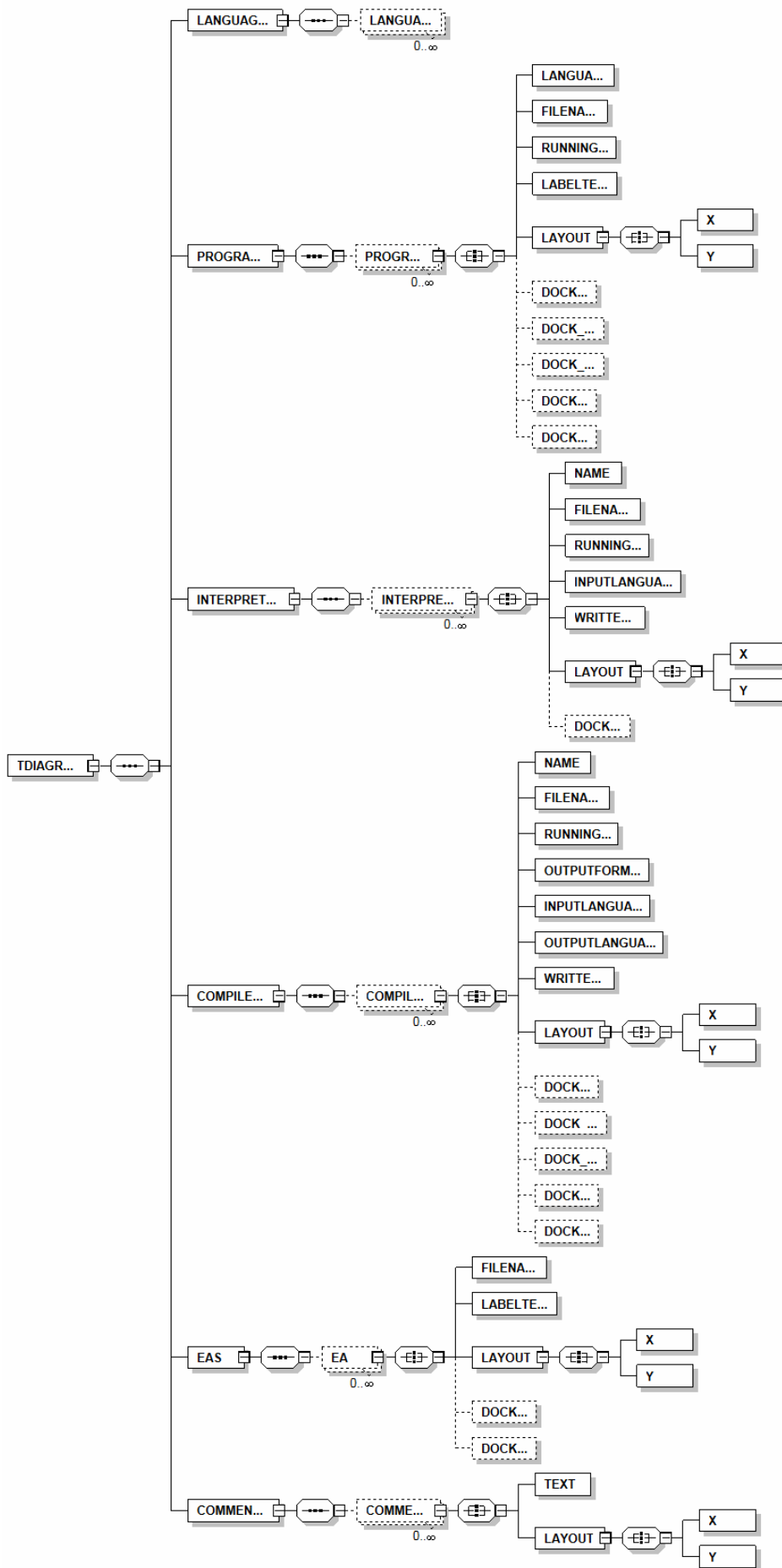
## Anhang A: XML Schemata der Komponenten

In diesem Abschnitt sind die XML Schemata der einzelnen Werkzeuge aufgelistet und dienen als Referenz. Die stets aktuellste Version findet man in den Ordnern der Programme mit der Dateierweiterung .xsd. Die hier verwendeten Diagramme zeigen ausschließlich die Elemente der XML Datei, jedoch nicht deren Attribute. Deshalb ist ein solches Diagramm nicht mit dem vollständigen XML Schema gleichzusetzen, sondern dient lediglich zur Orientierung. Beispiele für konkrete XML Dateien sind auszugsweise in den Kapiteln über die Werkzeuge gezeigt.

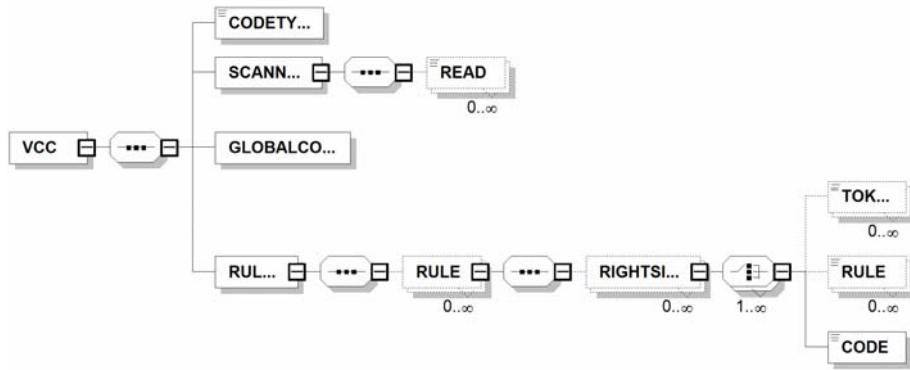
### AutoEdit / AutoEdit Workbook XML Schema



# TDiag XML Schema



## VCC XML Schema



## Anhang B: Anwendungsbeispiele für AtoCC

In diesem Abschnitt soll der jeweilige Verwendungszweck der einzelnen AtoCC Komponenten anhand von Anwendungsbeispielen aufgezeigt werden. Es handelt sich dabei in der Regel um Auszüge umfangreicherer Projekte. Die nachfolgenden Abbildungen sind stets neu nummeriert und nicht im Abbildungsverzeichnis der Arbeit aufgelistet.

### AutoEdit: Vom Automat zum LaTeX Dokument

In diesem Beispiel sollen die markantesten Schritte zur Erstellung eines Automaten mit AutoEdit und der anschließenden Einbindung des Transitionsdiagramms in ein LaTeX Dokument gezeigt werden.

AutoEdit präsentiert sich nach dem Start wie in Abbildung 1 gezeigt. Man beginnt mit einem Klick auf „Neuer Automat“.

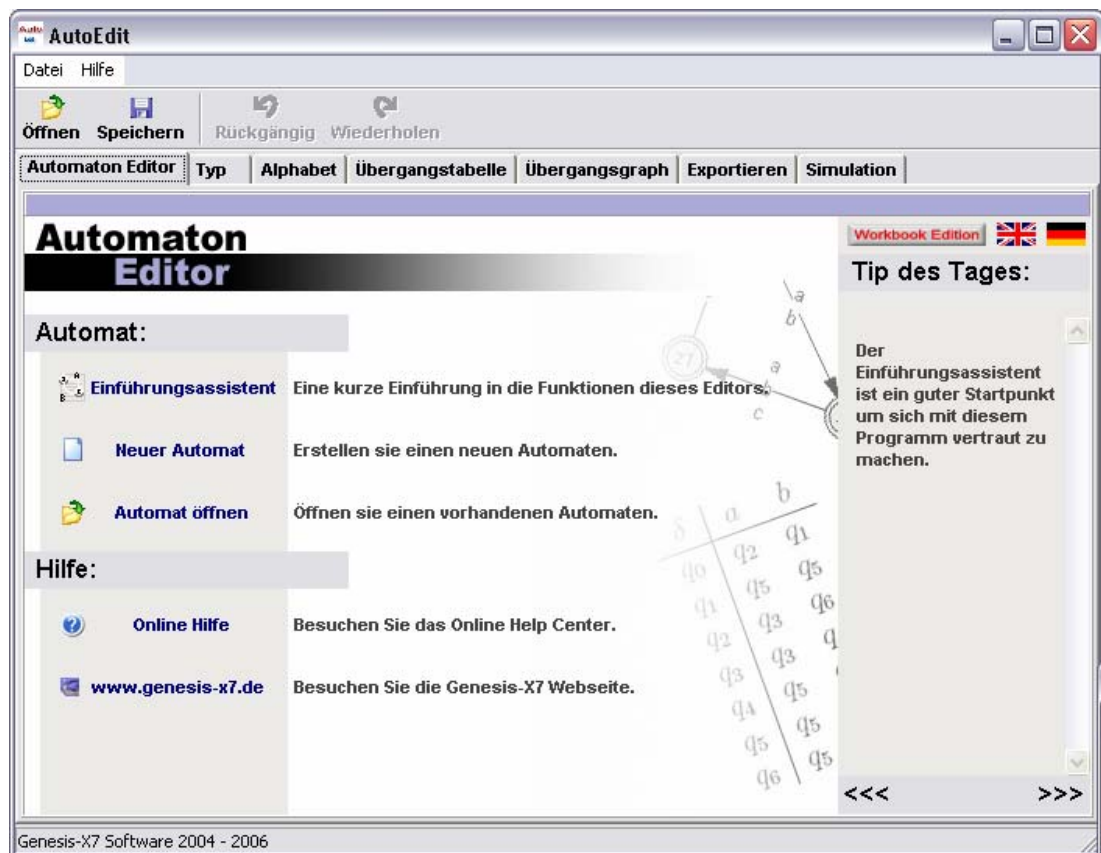


Abbildung 1: AutoEdit Startbildschirm

AutoEdit wechselt zum nächsten Tab, zur Auswahl des Automatentyps. Zu jedem Typ wird die entsprechende Definition angezeigt (Abbildung 2). Transformationen von einem Automatentyp in einen anderen sind in dieser Ansicht auch möglich.

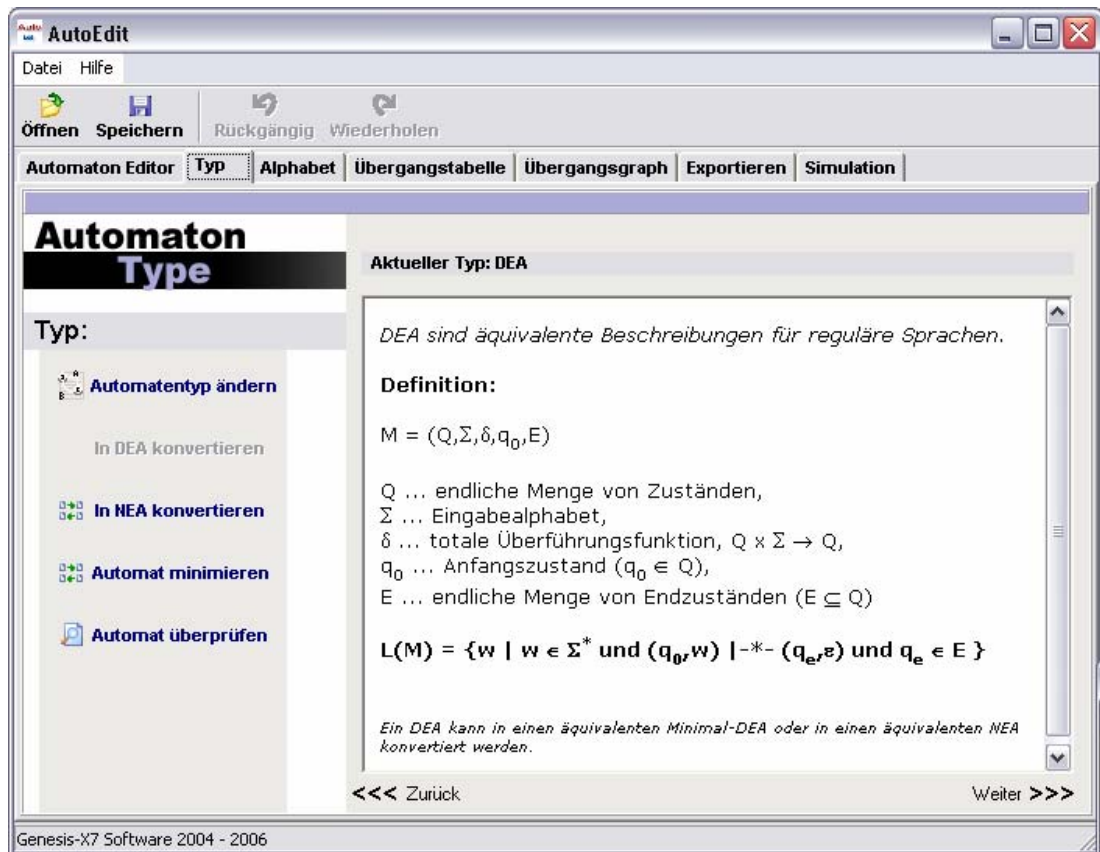
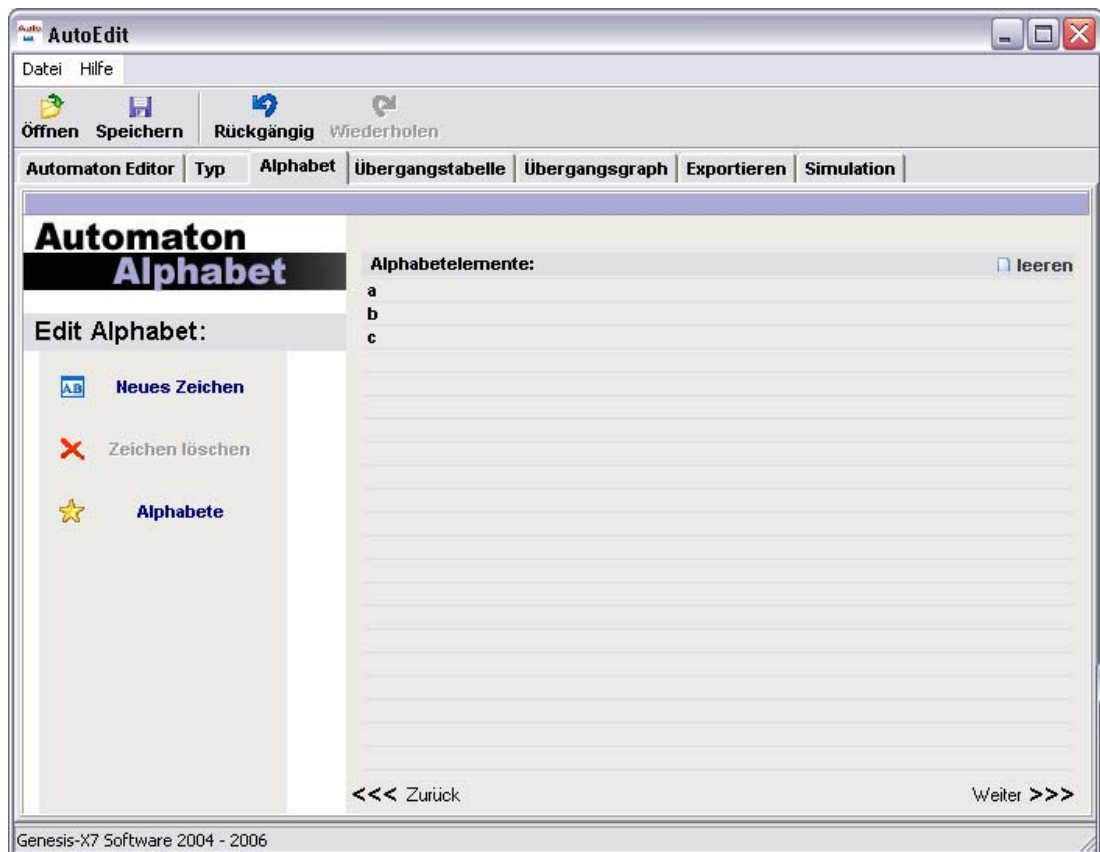


Abbildung 2: AutoEdit Typauswahl

Für dieses Beispiel soll der Typ „NEA“ (nichtdeterministische Endliche Automat) gewählt werden, was durch einen Klick auf „Automatentyp ändern“ bewerkstelligt wird. Die Auswahl wird durch einen Klick auf „Weiter“ im unteren rechten Eck bestätigt.

Das Alphabet des Automaten bildet die Grundlage für die spätere Diagrammerstellung und wird im nun folgenden Tab festgelegt (nachträgliche Änderungen sind natürlich auch zu einem späteren Zeitpunkt möglich). Ein Alphabet kann in AutoEdit manuell, Zeichen für Zeichen erstellt werden oder aber auch aus einer Auswahl vorgegebener Alphabete gewählt werden.

In diesem Beispiel soll das Alphabet  $\Sigma = \{a,b,c\}$  verwendet werden (Abbildung 3).



**Abbildung 3:** AutoEdit Alphabet

Bestätigt wird dies wieder durch einen Klick auf „Weiter“. Dem Anwender werden nun zwei unterschiedliche Möglichkeiten geboten ein Transitionsdiagramm zu erstellen. Der tabellarische Entwurf verzichtet vollständig auf eine graphische Darstellung des Diagramms (ein Beispiel für diese tabellarische Darstellung wird in Abbildung 4 gezeigt). Dem entgegengesetzt bietet der graphische Entwurf die Möglichkeit das Transitionsdiagramm direkt durch Manipulation der Zustände und Übergänge zu entwickeln (Abbildung 5). Der Nutzer kann zu jedem Zeitpunkt zwischen diesen beiden Entwurfsarten wechseln. Zum direkten Vergleich sind in Abbildung 4 und 5 jeweils die identischen Diagrammstellen eingefärbt.

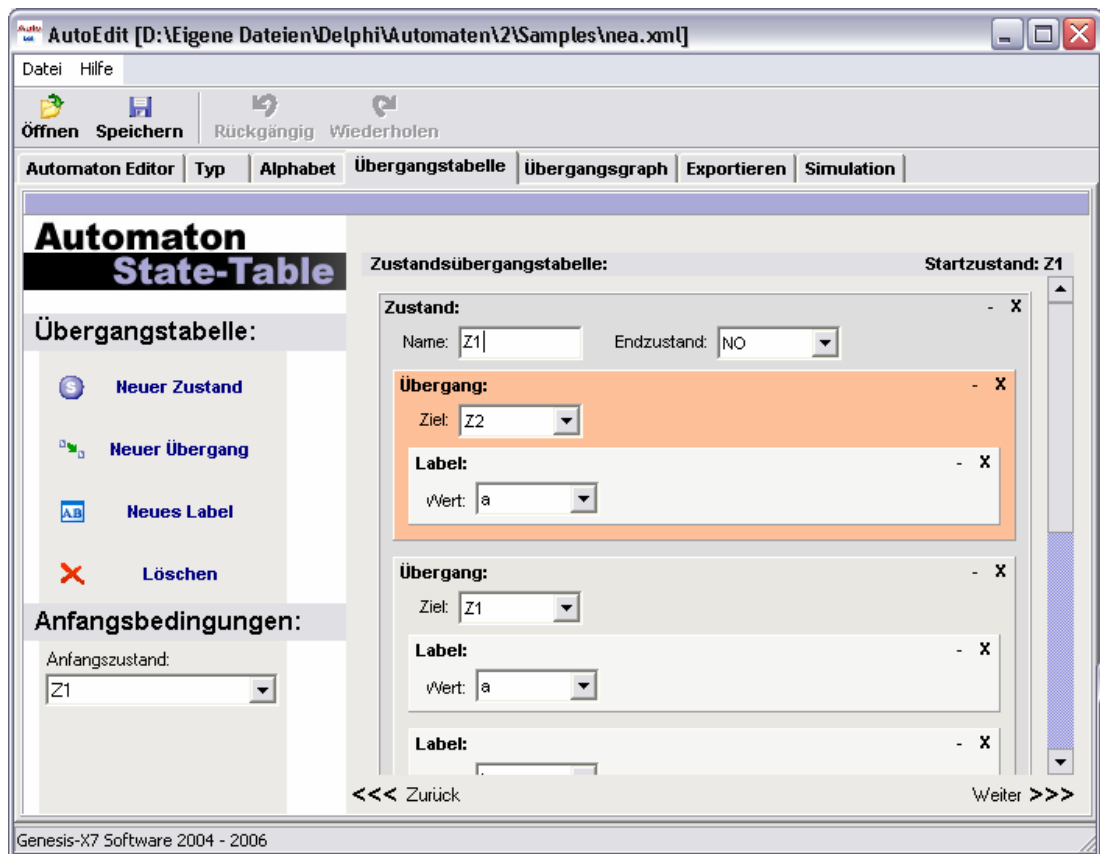


Abbildung 4: AutoEdit tabellarischer Entwurf

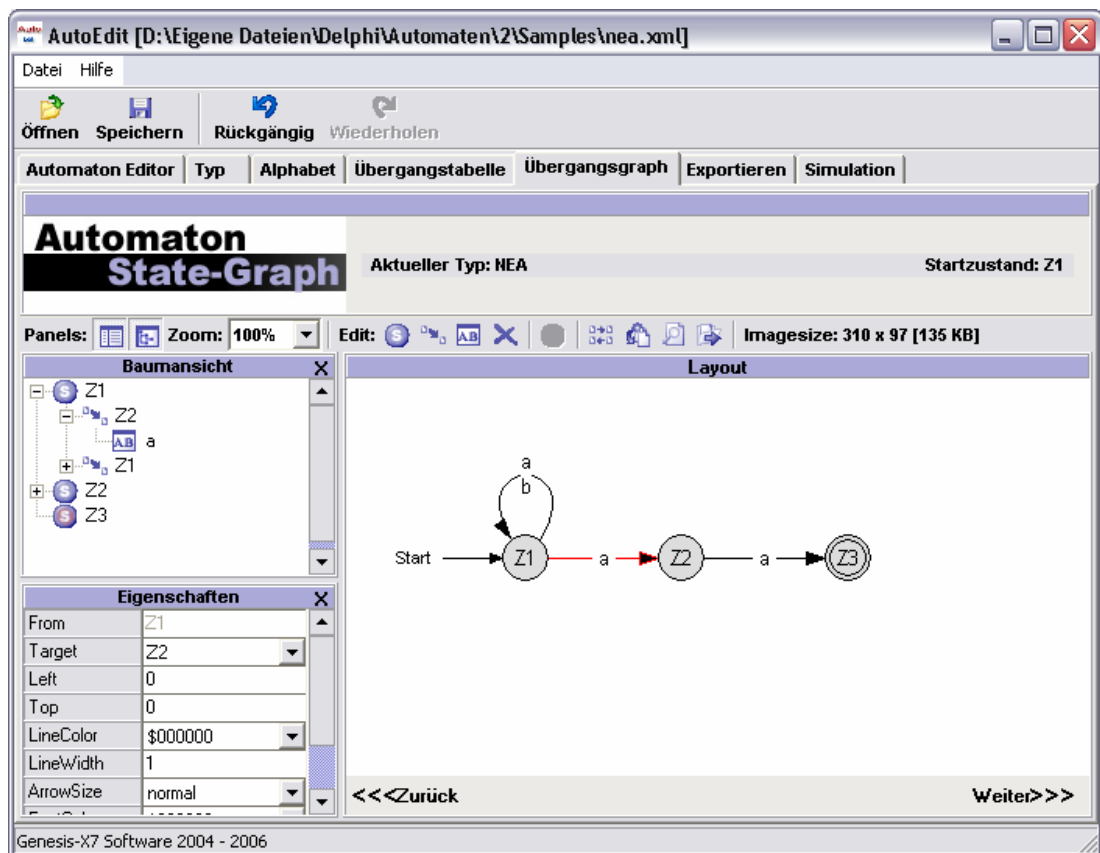
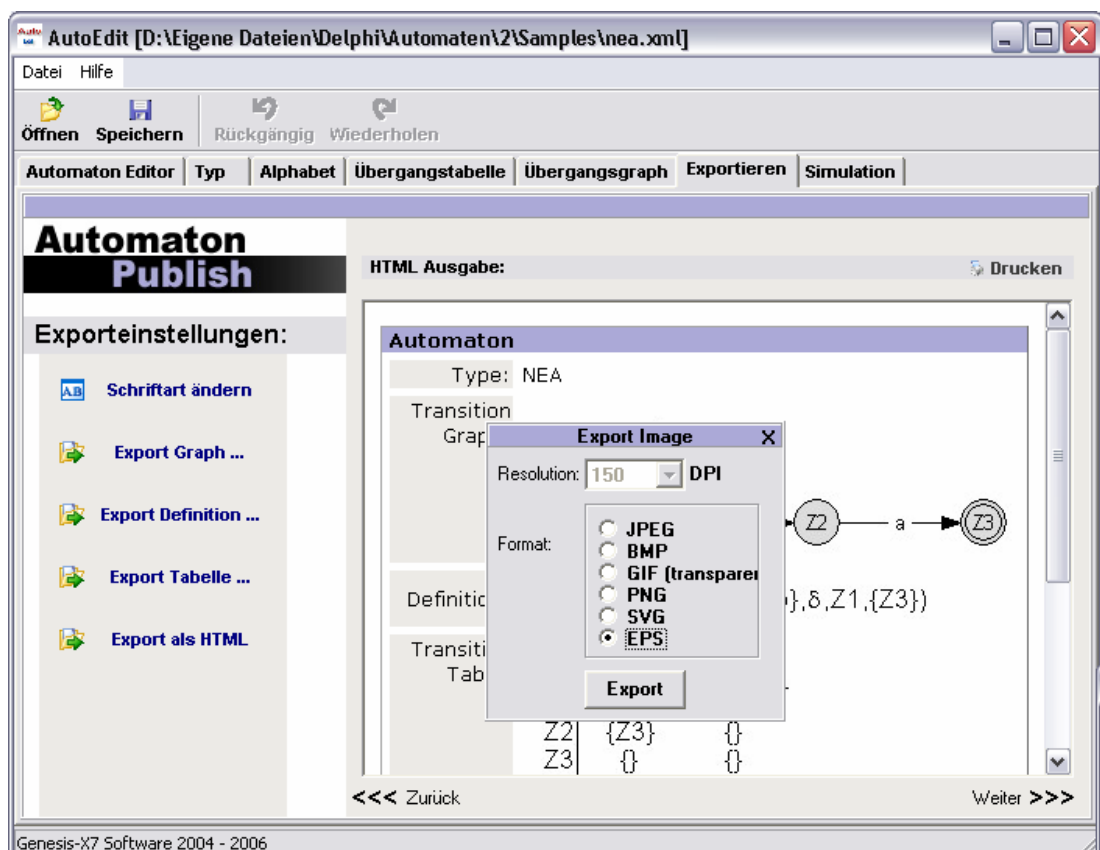


Abbildung 5: AutoEdit graphischer Entwurf

Die Erstellung des in Abbildung 4 und 5 gezeigten endlichen Automaten erfolgt über das Hinzufügen mehrerer Zustände und deren anschließender Verbindung durch Übergänge. Im tabellarischen Entwurf erfolgt dies durch Verwendung der entsprechenden Schaltflächen auf der linken Seite. Im graphischen Entwurf werden die Zeichenwerkzeuge im oberen Bereich verwendet, um anschließend auf der virtuellen Zeichenfläche die Diagrammelemente zu platzieren. Mit Hilfe der Maus kann das Diagramm solange manipuliert werden, bis es den Vorstellungen des Nutzers entspricht.

Ist das gewünschte Diagramm entworfen, kann über einen erneuten Klick auf „Weiter“ der Graph in eine separate Datei exportiert werden (auch Zustandsübergangstabellen und Automatendefinitionen können auf gleiche Weise exportiert werden). Nach einem Klick auf „Export Graph ...“ verlangt AutoEdit die Wahl eines Ausgabeformats und gegebenenfalls dessen Auflösung in DPI. Für die Verwendung in LaTeX eignet sich besonders das EPS Format (embedded Post Script). Durch die Bestätigung der Auswahl mit einem Klick auf „Export“ (Abbildung 6) erscheint der Windows Standarddialog, um die generierte Datei zu speichern.



**Abbildung 6:** AutoEdit Exportfunktion

Die anschließende Einbindung der EPS Datei in ein LaTeX Dokument kann mit folgendem Codefragment geschehen:

```
\begin{figure}
  \centering
  \includegraphics{Transitionsdiagramm}
  \label{fig:Transitionsdiagramm}
\end{figure}
```

Dies gilt unter der Annahme, dass die EPS-Datei als „Transitionsdiagramm.eps“ gespeichert wurde. Bei vorhandener GhostScript Installation erstellt AutoEdit mit Hilfe des Kommandozeilentools eps2pdf automatisch die passende PDF Datei des Diagramms, welche sich für pdfLaTeX eignet.

## AutoEdit Workbook: Eine Aufgabe lösen

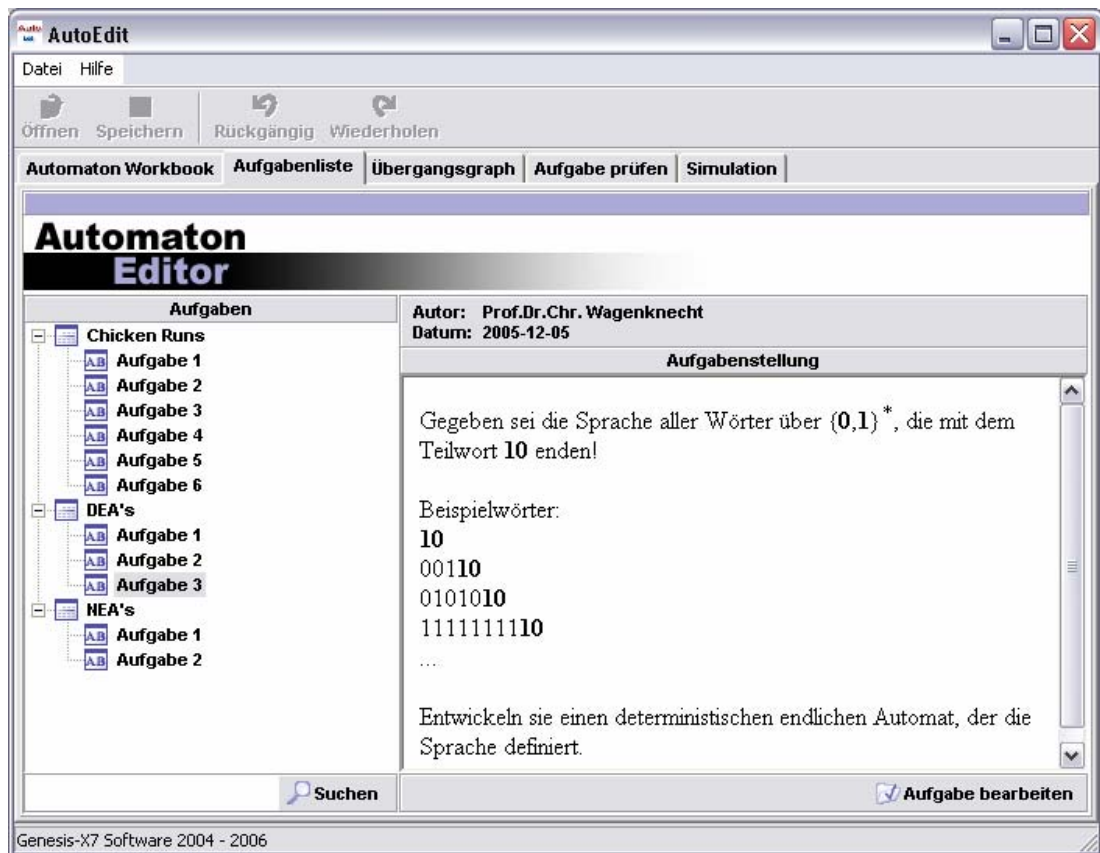
In diesem Beispiel soll eine Übungsaufgabe von AutoEdit Workbook gelöst werden. Die Konstruktion des Lösungsautomaten erfolgt über den graphischen Entwurf und ist analog zum Vorgehen im vorangehenden Abschnitt zu AutoEdit.

AutoEdit Workbook präsentiert sich nach dem Start wie in Abbildung 1 gezeigt.



**Abbildung 1:** AutoEdit Workbook

Nach einem Klick auf „Start neue Aufgabe“ wird automatisch die Liste aller verfügbaren Aufgaben vom Webserver geladen (Abbildung 2).



**Abbildung 2:** AutoEdit Workbook Aufgabenliste

Über die Suchoption ist es zusätzlich möglich den Aufgabenbaum auf bestimmte enthaltene Suchbegriffe zu beschränken. Wird eine Aufgabe aus der Baumdarstellung ausgewählt, wird die eigentliche Aufgabenstellung vom Webserver geladen und im rechten Fenster angezeigt. Die Wahl der Aufgabe wird durch einen Klick auf „Aufgabe bearbeiten“ bestätigt.

AutoEdit Workbook verwendet nun das Alphabet und den Automatentyp der Musterlösung und wechselt in die graphische Entwurfsansicht (Abbildung 3). Der Schüler kann nun eine Lösung für die gewählte Aufgabe erstellen. Ist dies abgeschlossen, kann durch einen Klick auf „Weiter“ im nächsten Tab die Lösung überprüft werden (Abbildung 4).

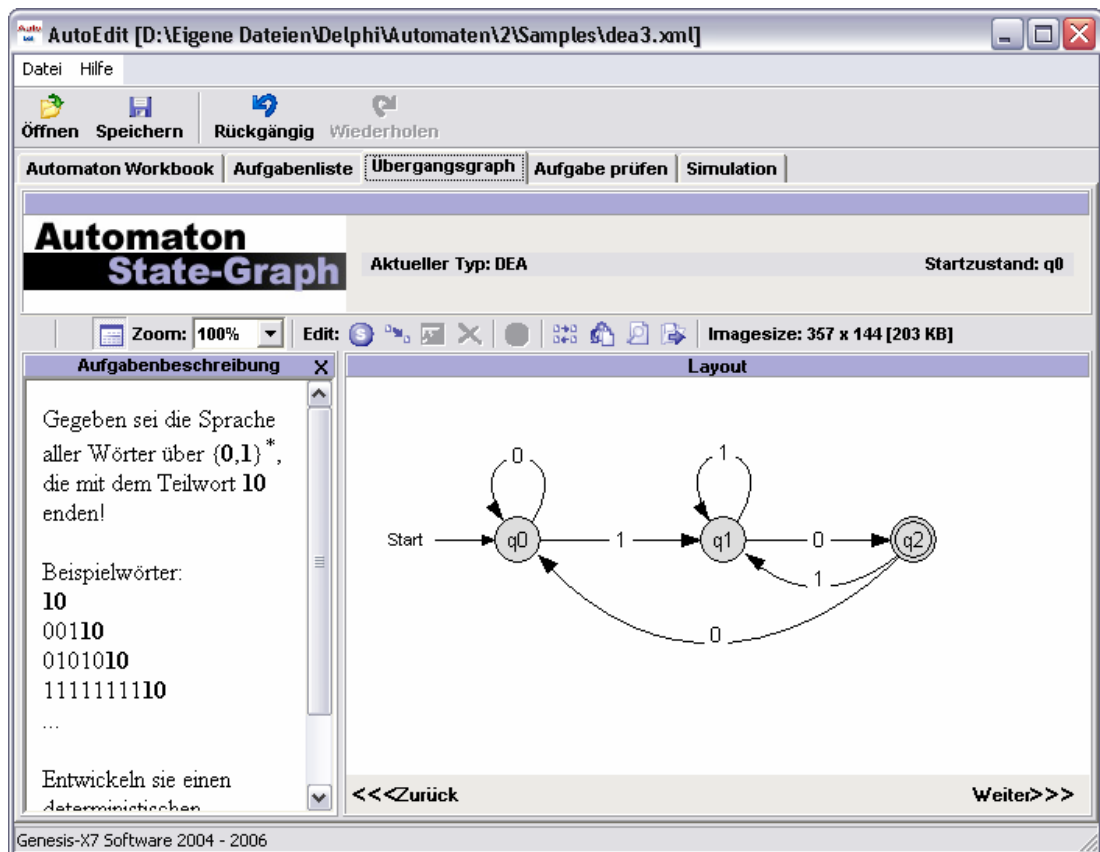


Abbildung 3: AutoEdit Workbook Entwurfsansicht

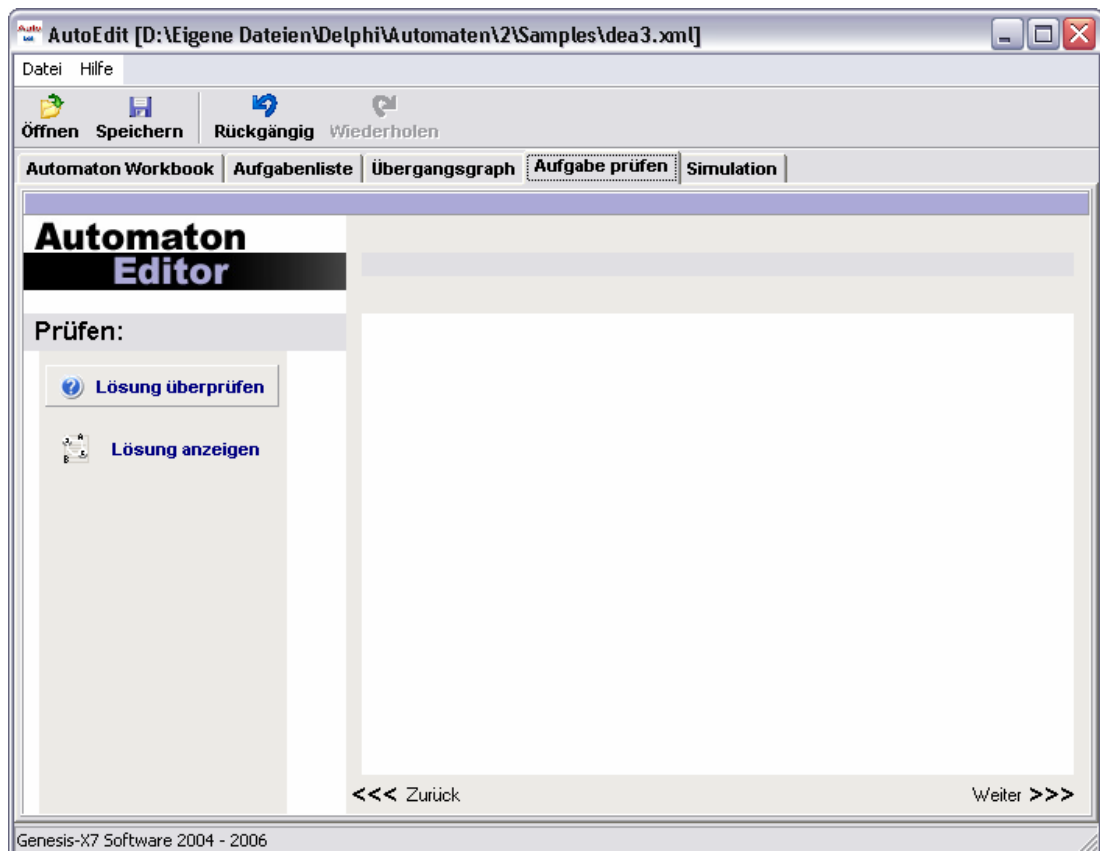
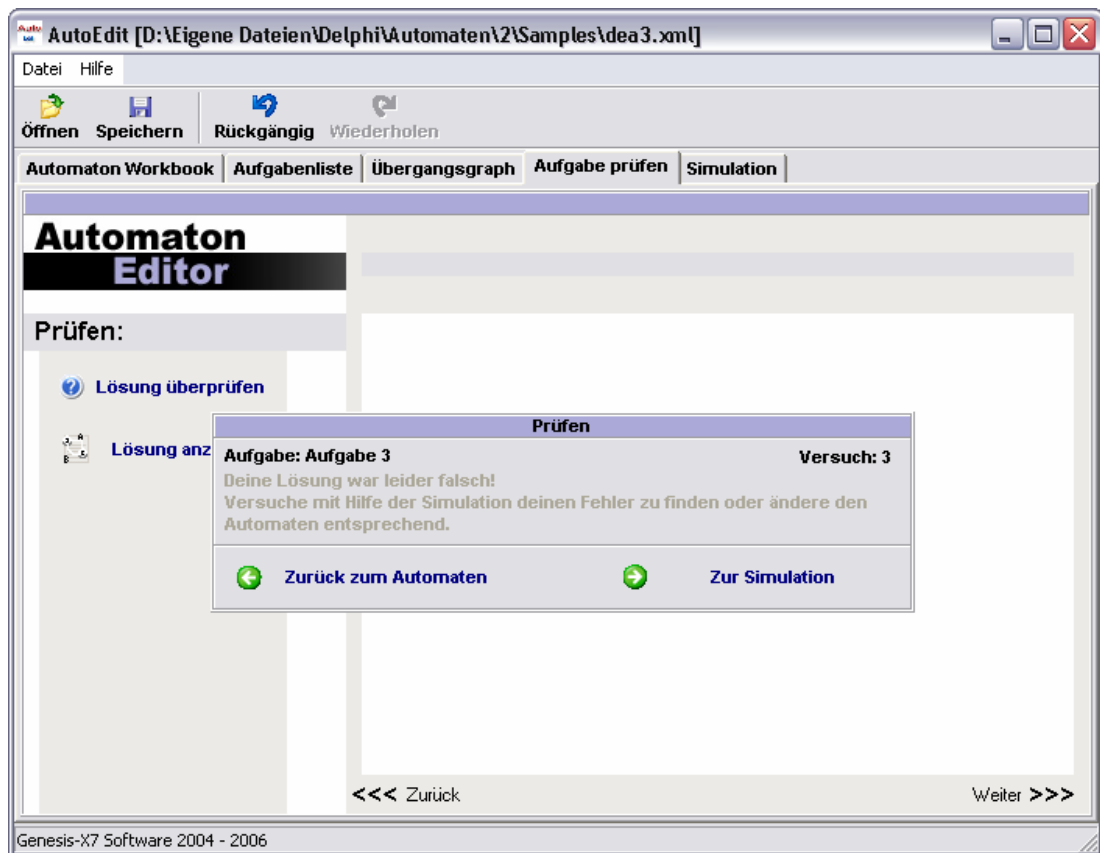


Abbildung 4: AutoEdit Workbook Lösung prüfen

Nach einem Klick auf „Lösung Prüfen“ wird von AutoEdit Workbook in Abhängigkeit zur Lösung ein Informationsfenster gezeigt (Abbildung 5 zeigt die Ausgabe bei einer falschen Lösung).



**Abbildung 5:** AutoEdit Workbook Lösung prüfen Ausgabe: Falsch

Mit Hilfe der Automaten simulation soll es dem Schüler leichter fallen einen Fehler zu finden, indem Eingabewörter auf den entwickelten Automaten angewendet werden können.

## **AutoEdit Workbook: Eine Chicken Run Aufgabe**

Einen spielerischen Ansatz bilden die so genannten Chicken Runs. Diese können zwar nicht direkt von AutoEdit auf Richtigkeit geprüft werden, aber durch graphische Simulation kann der Schüler seine Lösung mit der Aufgabenstellung vergleichen.

Die bei Chicken Runs zu entwickelnden Automaten sind meist sehr komplex (Beispiel in Abbildung 1)

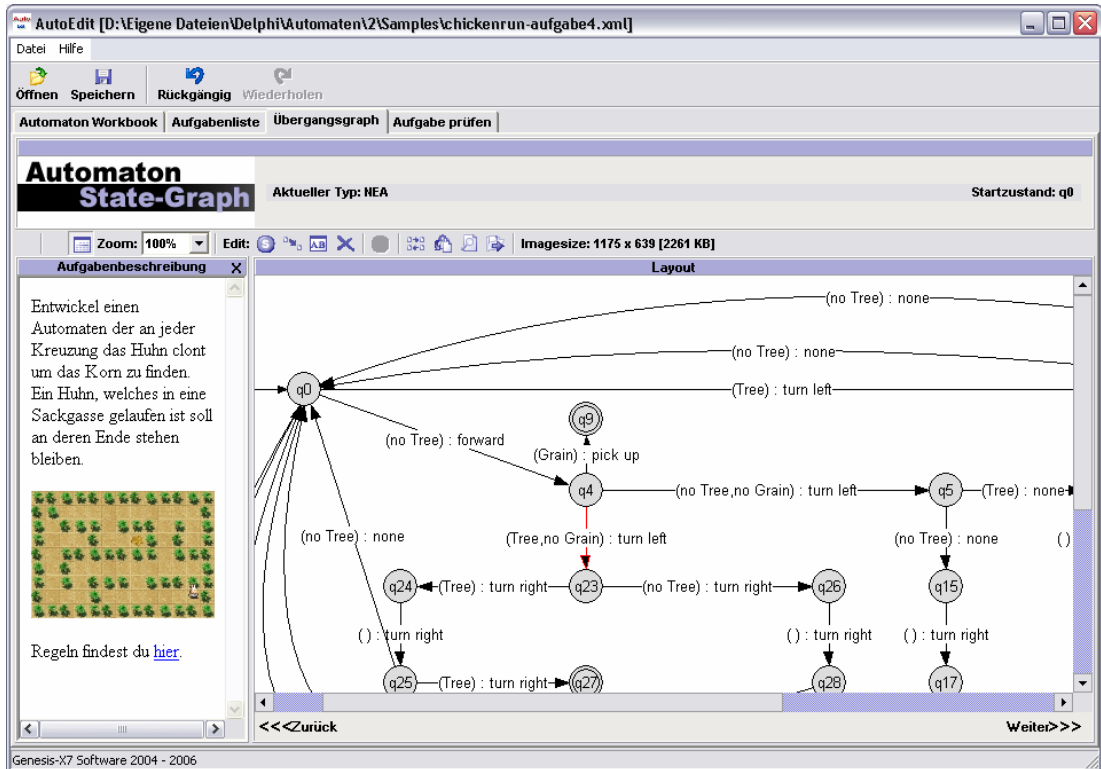


Abbildung 1: AutoEdit Workbook Chicken Run Beispiel

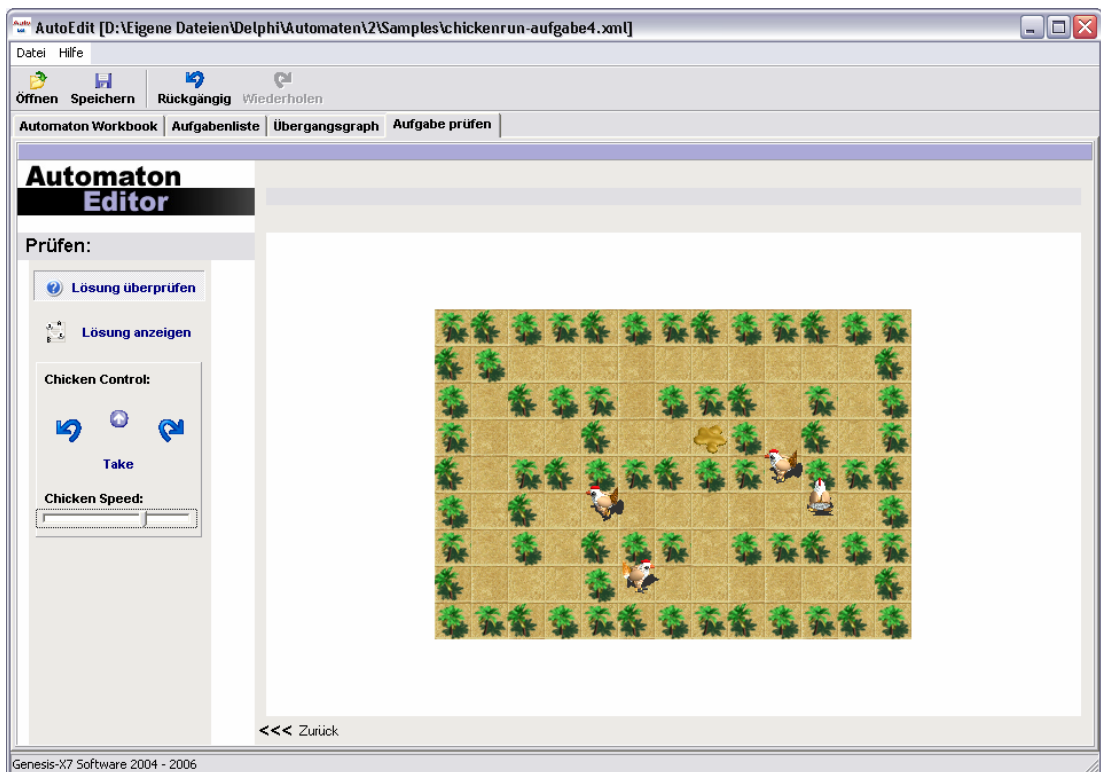


Abbildung 2: AutoEdit Workbook Chicken Run Simulation

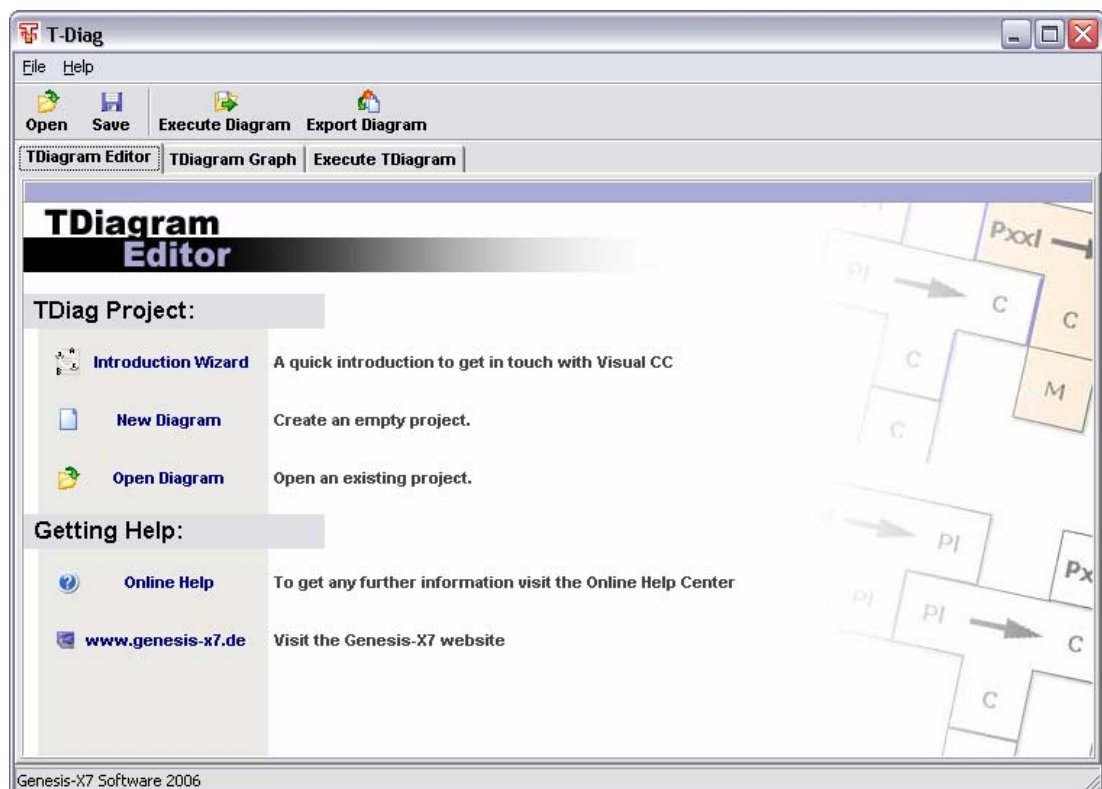
In Abbildung 2 wird die Simulation von Chicken Runs gezeigt. Es handelt sich dabei um eine animierte Kartendarstellung, auf der ein Huhn jeweils einen laufenden Automaten symbolisiert. Bei nichtdeterministischem Verhalten clonen sich die Hühner und die Abarbeitung läuft parallel für jeden Automaten weiter.

Chicken Runs laden zum Experimentieren ein und sind zweifellos durch ihre Komplexität eine Herausforderung für Schüler, welche die Aufgaben mit DEA's und NEA's bereits erfolgreich gelöst haben.

## **TDiag: Entwicklung und Ausführung eines T-Diagramms**

In diesem Beispiel soll mit Hilfe von TDiag ein T-Diagramm erstellt und anschließend abgearbeitet werden.

TDiag präsentiert sich nach dem Start wie in Abbildung 1 gezeigt.



**Abbildung 1: TDiag Startseite**

Durch einen Klick auf „New Diagram“ wechselt TDiag in die Entwurfsansicht, wie in Abbildung 2 gezeigt.

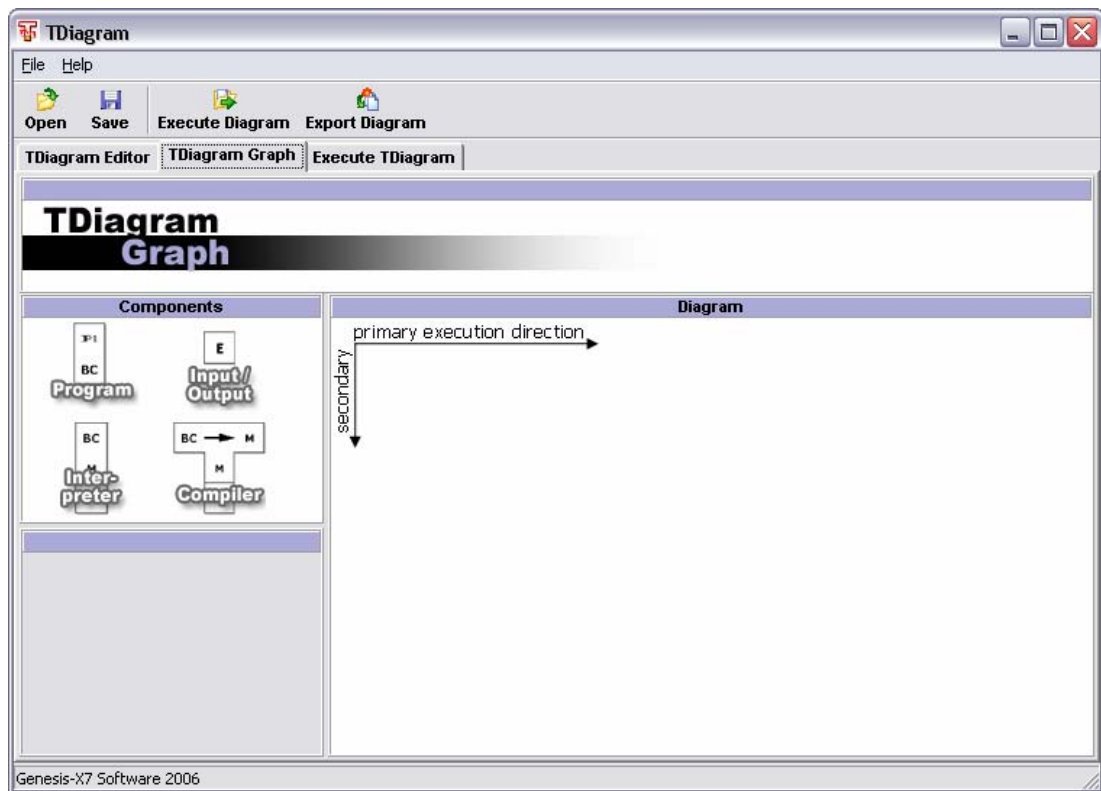


Abbildung 2: TDiag Entwurfsansicht

Auf der linken Seite sind die Diagrammkomponenten symbolisch dargestellt. Durch Drag & Drop können diese auf die virtuelle Zeichenfläche in der Mitte platziert werden.

Für dieses Beispiel soll zunächst ein Java Compiler auf die Zeichenfläche gezogen werden. Durch Drag & Drop der „Compiler“ Komponente auf die Zeichenfläche erscheint ein Dialog, in welchem ein fertiges Compilersetup für den Java Compiler gewählt werden kann (Abbildung 3). Die benötigten Einstellungen werden von TDiag automatisch vorgenommen und können auch nachträglich in der „Property“ Liste (unten links) abgeändert werden.

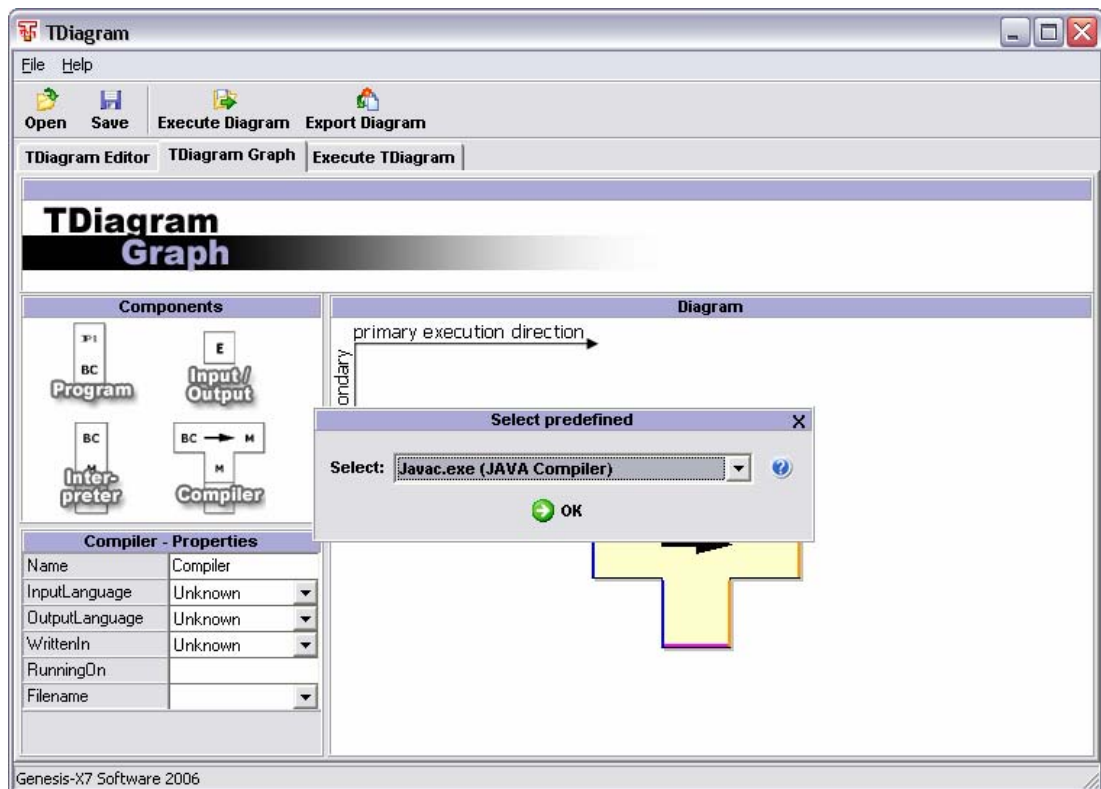


Abbildung 3: TDiag Compilerdialog

Durch Hinzufügen von zwei Programmbausteinen entsteht das in Abbildung 4 gezeigte Diagramm. Alle Komponenten sind noch rot eingefärbt, um anzuzeigen, dass diese noch nicht aneinander passen.

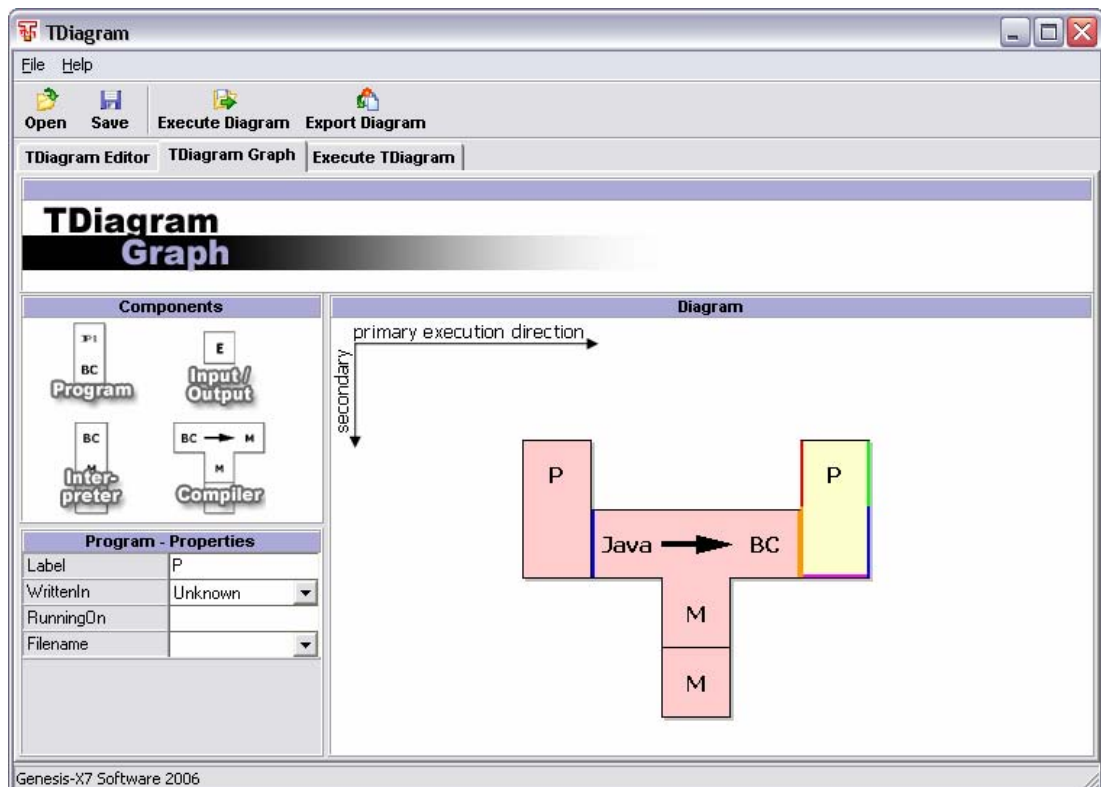


Abbildung 4: TDiag Diagramm

Die Programmbausteine benötigen noch die fehlende Einstellung „WrittenIn“ um festzulegen, in welcher Sprache die Programme vorliegen. Passend zum Compiler muss am linken Baustein die Sprache Java und am rechten Baustein die Sprache Java Bytecode gewählt werden. Durch die Wahl des Dateinamens beider Programmbausteine ist dieses T-Diagramm bereits lauffähig (Abbildung 5).

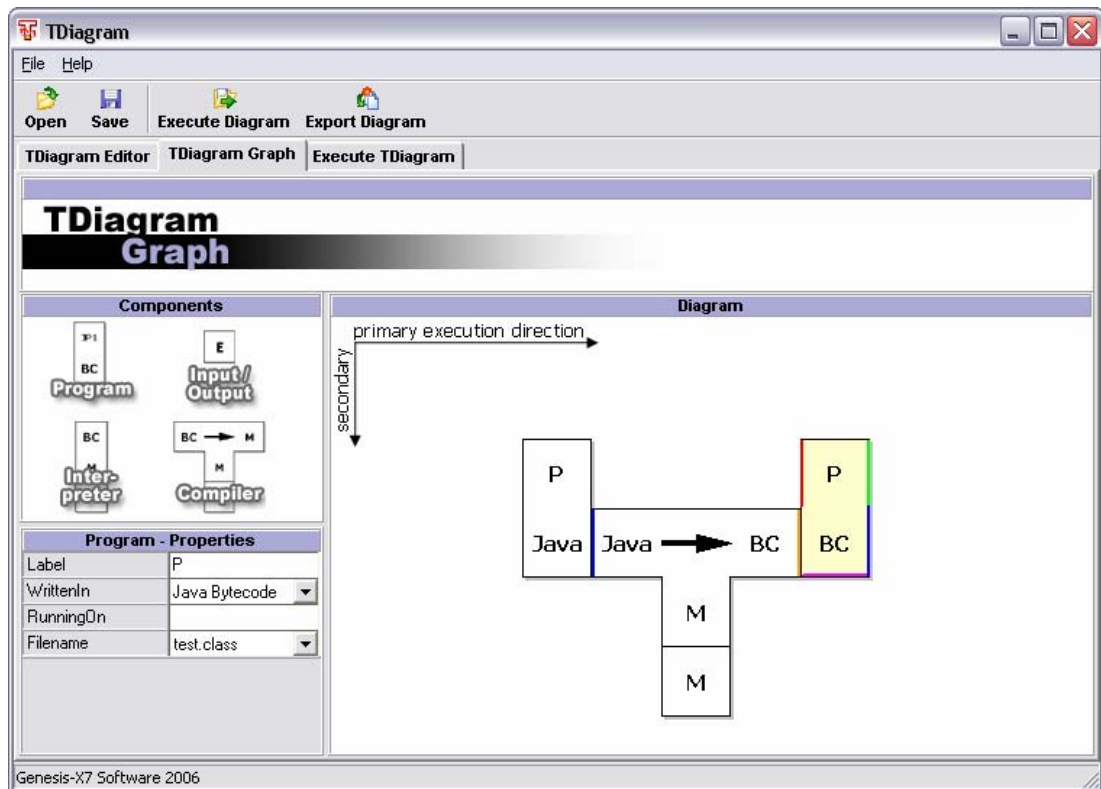


Abbildung 5: TDiag lauffähiges Diagramm

Über den Button „Execute Diagram“ (oder über die Tabs) wechselt TDiag in die Ausführungsansicht. In Abbildung 6 ist die Ausgabe des obigen Diagramms gezeigt. Die einzelnen Verarbeitungsschritte werden durch „ECHO“ Anweisungen angezeigt. Komplexere Diagramme liefern in dieser Ansicht entsprechend umfangreichere Ausgaben.

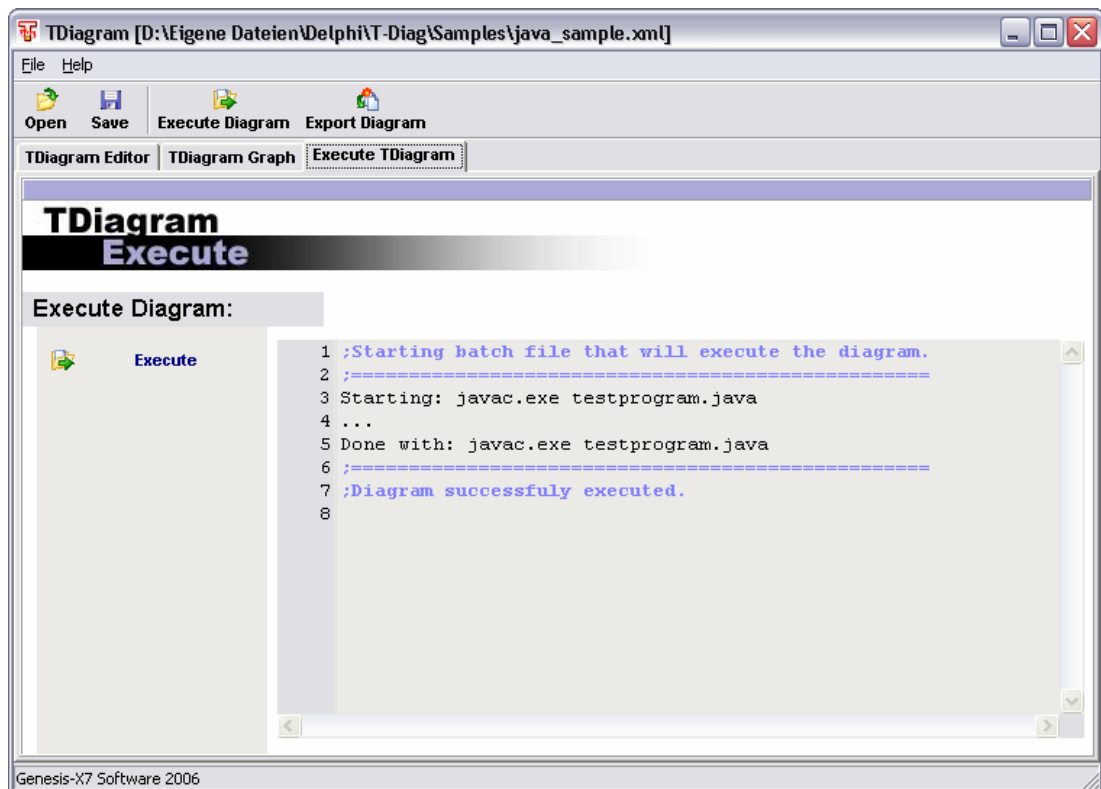


Abbildung 6: TDiag Ausführungsansicht

## VCC: Entwicklung eines einfachen Compilers

In diesem Beispiel soll ein einfacher Compiler entwickelt werden, der primitive mathematische Ausdrücke als Eingabedatei lesen und berechnen kann und anschließend das Rechenergebnis in die Ausgabedatei schreibt.

VCC präsentiert sich nach dem Start wie in Abbildung 1 gezeigt. Zunächst sollte auf einem Blatt Papier eine Grammatik aufgestellt werden, die anschließend in VCC eingegeben werden kann. Für das Taschenrechnerbeispiel sollen die vier Grundrechenarten und Klammerausdrücke verwendet werden. Um dieses Beispiel nicht unnötig zu komplizieren, werden ausschließlich ganze Zahlen verwendet.

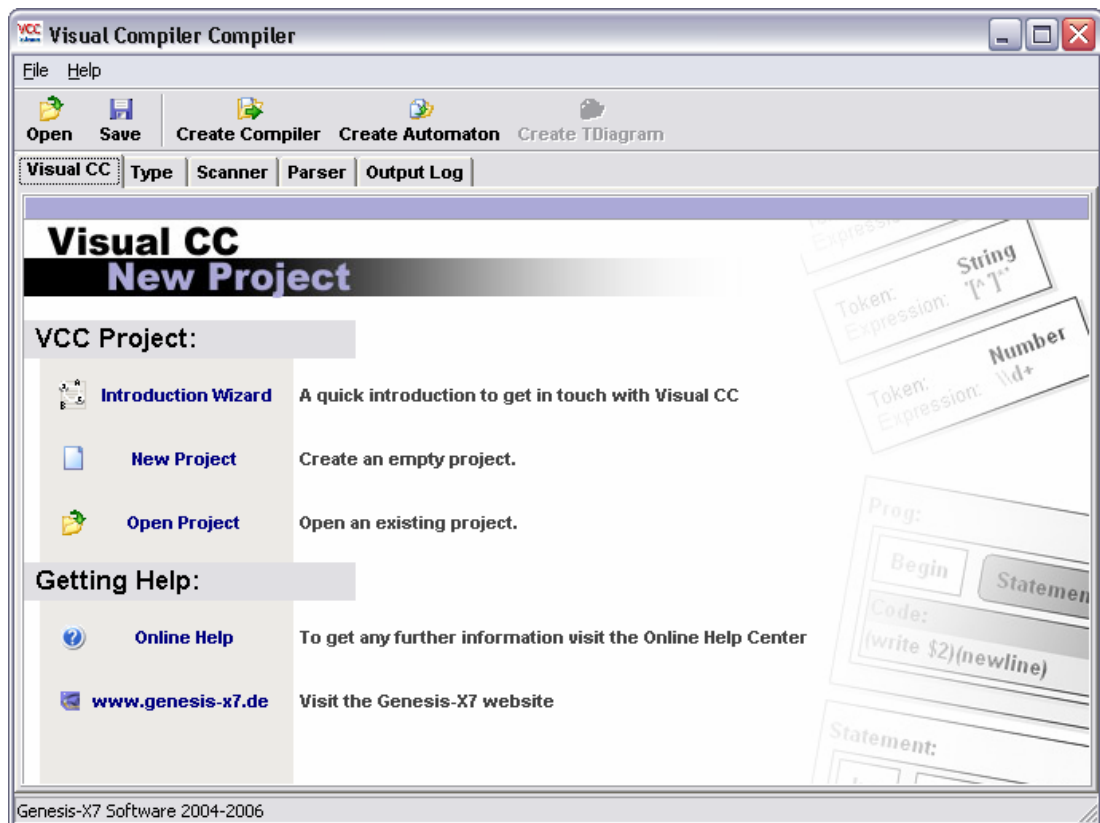
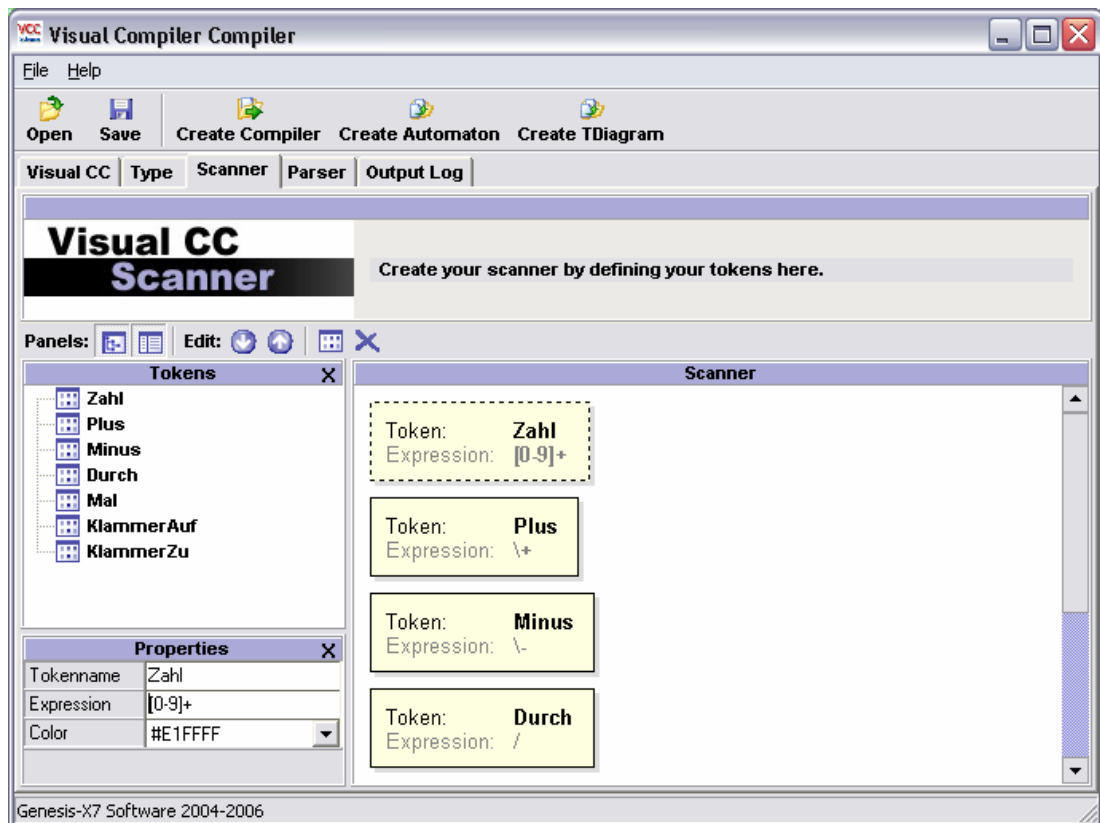


Abbildung 1: VCC Startbildschirm

Die im Weiteren verwendete Grammatik:

Rechnung	→	Ausdruck Rechnung   ε
Ausdruck	→	Ausdruck * Ausdruck   Ausdruck / Ausdruck   Ausdruck + Ausdruck   Ausdruck - Ausdruck   ( Ausdruck )   Zahl
Zahl	→	Ziffer Zahl   ε
Ziffer	→	1   2   3   4   5   6   7   8   9   0

Der Scanner für diese Grammatik beinhaltet nur 7 Token. In Abbildung 2 ist auszugsweise ein Scanner für obige Grammatik gezeigt. Wichtig ist bei den Expressions (reguläre Ausdrücke), entsprechende Sonderzeichen mit einem vorangestellten „\“ zu ergänzen. Das Nichtterminal Zahl wurde bereits im Scanner durch die Verwendung von „[0-9]+“ aufgelöst. Dies ist nicht zwingend nötig, spart aber Aufwand.



**Abbildung 2:** VCC Scanner

Anschließend kann über die Tabs auf die Parseransicht gewechselt werden. Zunächst werden die Nichtterminale als Regeln (Rechnung, Ausdruck und Zahl) hinzugefügt. Wie in Abbildung 3 gezeigt, werden diese 3 Regeln in VCC dargestellt. Nun kann Schritt für Schritt die gesamte Grammatik in diese Darstellung übertragen werden. Dazu dienen die Schaltflächen im oberen Bereich, oder es kann mit dem Popup Menü (durch Rechtsklick) gearbeitet werden. Unter Token bzw. Regeln werden ausschließlich die bereits vorhandenen in Scanner bzw. Parser vorgegeben.

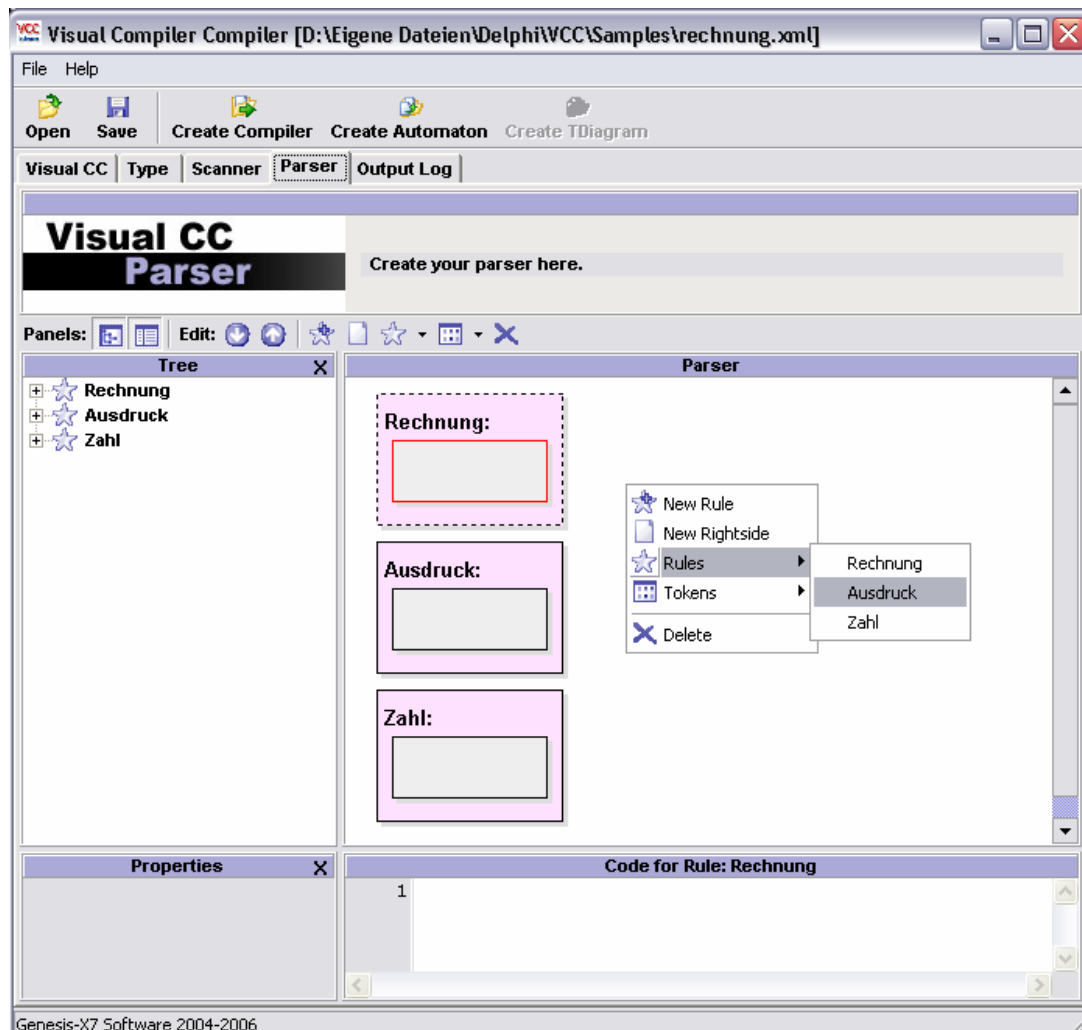


Abbildung 3: VCC Parser 1

Wie die einzelnen Regeln in VCC übertragen werden, soll am Beispiel des Startsymbols „Rechnung“ gezeigt werden.

Rechnung  $\rightarrow$  Ausdruck Rechnung |  $\epsilon$

Der linke Teil „Rechnung“ wird durch die Regel selbst repräsentiert. Für diese Regel gibt es zwei mögliche Auflösungen. Einmal ein „Ausdruck“ gefolgt von einer neuen „Rechnung“, oder Epsilon. Die typische Darstellung „|“ wird in VCC durch weiße Rechtecke innerhalb der Regeln untereinander dargestellt. Über „New Rightside“ können weitere Rechtecke hinzugefügt werden. In das erste bereits vorhandene Rechteck (Selektion durch einfachen Mausklick, erkennbar durch rote Umrandung) werden nun zunächst die Regel „Ausdruck“ und anschließend die Regel „Rechnung“ aufgenommen. Durch einen Klick auf „New Rightside“ wird die zweite, in diesem Fall leere rechte Regelseite hinzugefügt. Das Ergebnis wird in Abbildung 4 gezeigt.

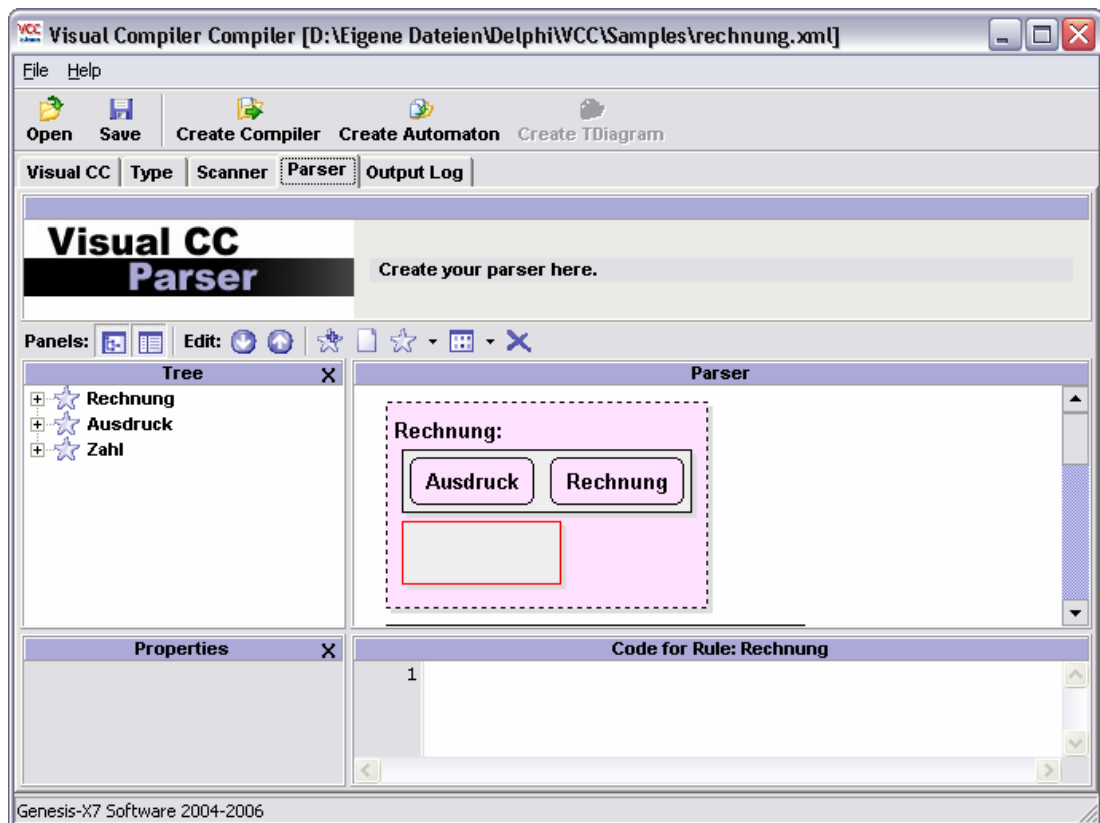


Abbildung 4: VCC Parser 2

Analog kann die Regel „Ausdruck“ übertragen werden, wobei hier auch Token (für die Rechenzeichen) verwendet werden (Abbildung 5).

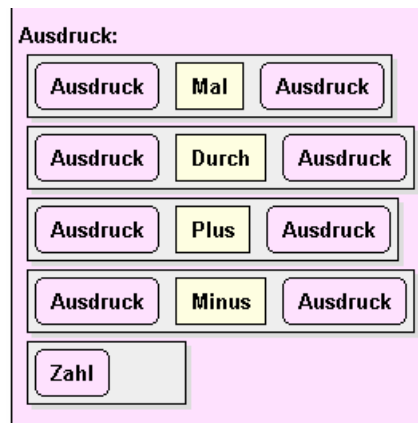


Abbildung 5: VCC Parser 3

Wurde die gesamte Grammatik auf diese Weise in VCC übertragen, kann die Codegeneration des Compilers hinzugefügt werden. Dabei müssen äquivalent zu YACC für die einzelnen Regeln Codefragmente hinzugefügt werden, die bei der Abarbeitung der Regel ausgeführt werden sollen. Wichtig ist hierbei das Verständnis der Abarbeitungsreihenfolge und die Kenntnis der Sondervariablen \$\$ und \$n.

Hierzu kann zum einen das mitgelieferte VCC Beispielprojekt mit Hinweisen für diese Problematik herangezogen werden, oder aber auch Literatur zu YACC verwendet werden.

Eine typische Codezeile für das Taschenrechnerbeispiel (hier in C#) würde wie folgt aussehen:

```
$$ = (int.Parse($1) + int.Parse($3)).ToString();
```

Da VCC prinzipiell immer mit Zeichenketten in den \$\$, \$n Variablen arbeitet, müssen diese zunächst in Ganzzahlen umgewandelt, anschließend addiert und danach wieder in einen String zurück transformiert werden.

In Abbildung 6 ist dieses Quellcodefragment bei der Additionsregel in VCC eingetragen (unterer Bereich). Analog werden die anderen Regeln mit Quellcode ergänzt (In diesem Falle muss lediglich das Rechenzeichen abgeändert werden.).

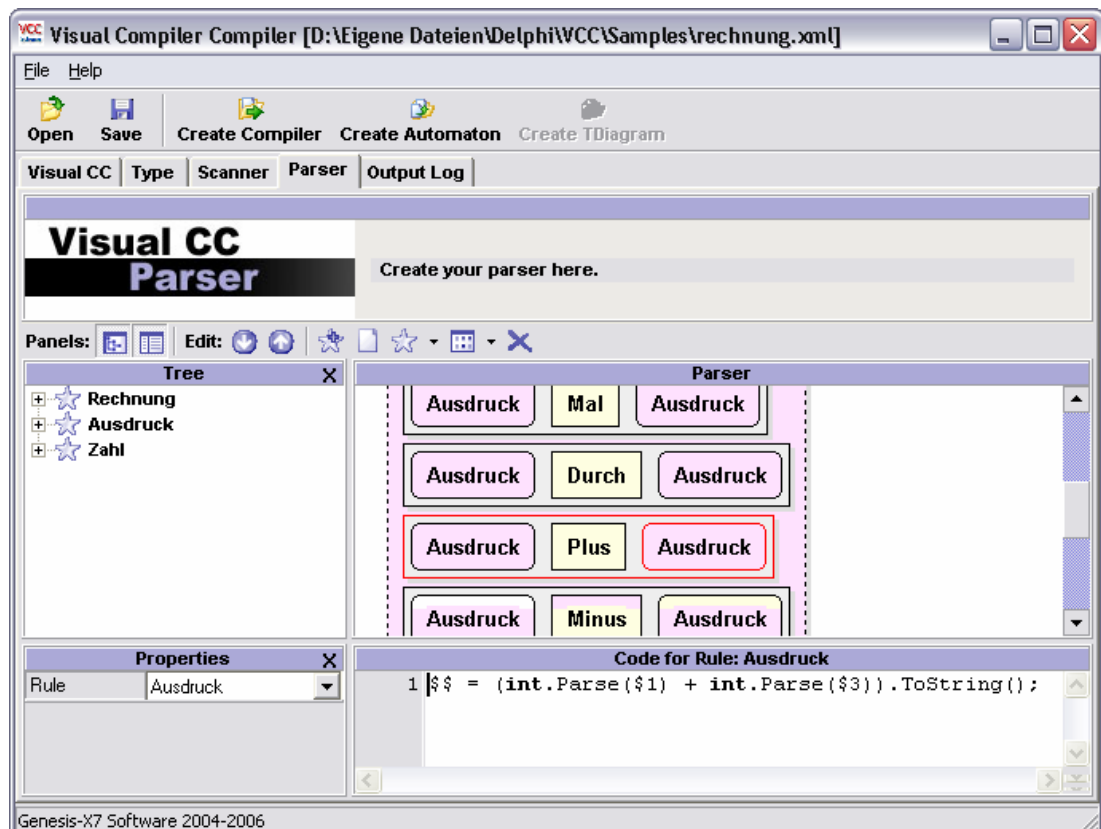



Abbildung 6: VCC Parser Code

Über „Create Compiler“ kann ein fertiger Compiler erzeugt werden. VCC öffnet automatisch eine Konsole im betreffenden Verzeichnis und gibt Hinweise über die Startparameter:



```
C:\WINDOWS\system32\cmd.exe
You can run your compiler here on command line.
Type: calc.exe [-t] input.txt [output.txt]
Parameter: -t show tokens
D:\Eigene Dateien\Delphi\UCC\Samples>
```

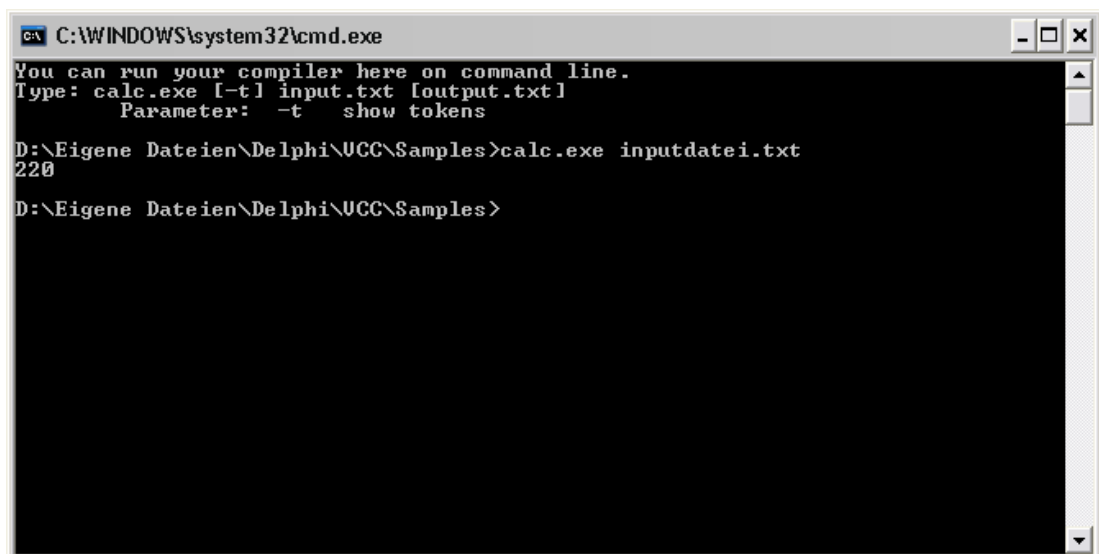
Abbildung 7: VCC Konsole

Eine Eingabedatei für den entwickelten Taschenrechner-Compiler könnte nun wie folgt aussehen:



```
((32*12)-(82/2))-123
```

Über den Aufruf `calc.exe inputdatei.txt` wird dieser Ausdruck berechnet und das Ergebnis: 220 zurückgegeben.



```
C:\WINDOWS\system32\cmd.exe
You can run your compiler here on command line.
Type: calc.exe [-t] input.txt [output.txt]
Parameter: -t show tokens
D:\Eigene Dateien\Delphi\UCC\Samples>calc.exe inputdatei.txt
220
D:\Eigene Dateien\Delphi\UCC\Samples>
```

Abbildung 8: VCC Konsole Ausgabe

## Anhang C: Ausgewählte Quellcodebeispiele

In diesem Abschnitt sollen einige ausgesuchte Codebeispiele gezeigt werden, denn der Gesamtumfang des AtoCC Quellcodes mit allen Komponenten beläuft sich auf über 30'000 Zeilen (entspricht mehr als 400 DIN A4 Seiten in Schriftgröße 8). Für die Entwicklung der Werkzeuge wurde Borland Delphi 5 verwendet.

### Generieren von Scheme Code für einen DEA

AutoEdit bietet die Möglichkeit einen Automaten in einen Quellcode für die Programmiersprache Scheme zu transformieren. Umgesetzt wurde dies mit einer fest programmierten Schablone, die mit den entsprechenden Informationen des aktuellen Automaten aufgefüllt wird. Je nach Automatentyp sind diese Schablonen mehr oder weniger komplex. Die Prozedur für den DEA Code ist die am wenigsten Aufwendige.

```
procedure TAutomatonSchemeCode.GenerateDEACode;
var i,z,t : Integer;
begin
Lines.Clear;
with Lines do begin
Add(';;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;');
Add(';;; DEA');
Add(';;; example: (DEA "("+Automaton.RandomAlphabetItem+' '+
Automaton.RandomAlphabetItem+' '+
Automaton.RandomAlphabetItem+' '+
Automaton.RandomAlphabetItem+' '+
Automaton.RandomAlphabetItem+')')');
Add(';;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;');
Add(' (define DEA');
Add(' (lambda (input) ');
Add(' (letrec');
with Automaton.Automaton do
for i := 0 to high(Statements) do begin
if i = 0 then
Add(' (('+Statements[i].Name+' (lambda (tape) ') else
Add(' ('+Statements[i].Name+' (lambda (tape) ');
```

```
Add(' (printf "~a: ~a~n" ""+Statements[i].Name+' (tape "(show))) ');
Add(' (case (tape "(read)) ');
for z := 0 to high(Statements[i].Transitions) do begin
  for t := 0 to high(Statements[i].Transitions[z].Conditions) do begin
    Add(' (('+Statements[i].Transitions[z].Conditions[t].Value+
      ') (tape "(right!)) ('+Statements[i].Transitions[z].Statement+
      ' tape)))');
  end;
end;
if Statements[i].EndStatement then
  Add(' (($) #t) ');
  Add(' (else #f))) ');
end;
Add(' (DFATape (lambda ())) ');
Add(' (let ((left-part "$')) ');
Add(' (right-part "$)) ');
Add(' (lambda (message) ');
Add(' (case (car message) ');
Add(' ((init!));
Add(' (set! left-part (reverse (caadr message))) ');
Add(' (set! right-part (cadadr message))) ');
Add(' ((right!));
Add(' (set! left-part (cons (car right-part) left-part)) ');
Add(' (if (not (equal? right-part "$)) ');
Add(' (set! right-part (cdr right-part))) ');
Add(' ((show));
Add(' (list (reverse left-part) right-part)) ');
Add(' ((read));
Add(' (car right-part));
Add(' (else (error "tape "~a" (car message)))))))); ');
Add(' (let ((t (DFATape))) ');
Add(' (t (list "init! (list (list "$) (append input (list "$)))))); ');
Add(' ('+Automaton.Automaton.StartStatement+' t))) ');
Add(' ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; ');
```

Der fett hervorgehobene Teil ist die eigentliche Transformation. Die restlichen Zeilen bilden das Grundgerüst, das DFATape, welches das virtuelle Band verkörpert.

## Generieren von MS-DOS Batch Dateien

TDiag bietet die Möglichkeit ein T-Diagramm in eine ausführbare Batchdatei zu transformieren. Durch die Verwendung von ECHO Ausgaben kann der Anwender die Abarbeitung verfolgen und wird gegebenenfalls auf Fehler hingewiesen. Transformationen sind entweder für das gesamte Diagramm oder auch nur für selektierte Bausteingruppen möglich.

```
procedure TDiagramExecuter.CreateBatchFile(filename : String; ...);
var outf : TStringList;
    DoneList : TList;
    s,paras : String;
    i : Integer; ea : PEA; p : PProgram; int : PInterpreter; c : PCompiler;
    strin,strout : string;

procedure AddElement (s : String); // Hilfsfunktion für ECHO Ausgaben
begin
    s := Replace(s,'\\','\');
    outf.Add('ECHO Starting: ' + s);
    outf.Add('ECHO ... ');
    outf.Add(s);
    outf.Add('IF ERRORLEVEL 1 GOTO ERROR ');

    outf.Add('ECHO Done with: ' + s);
    outf.Add('ECHO ;=====');
    outf.Add('');
end;

begin
    outf := TStringList.Create;
    DoneList := TList.Create;
// Anfang
    outf.Add('@ECHO OFF');
    outf.Add('REM This file was automaticly created by T-Diag');
    outf.Add('ECHO ;Starting batch file that will execute the diagram. ');
    outf.Add('ECHO ;=====');
```

```
// Elemente bearbeiten
for i := 0 to Diagram.SortedList.Count - 1 do begin
  if SelectedOnly and not PProgram(Diagram.SortedList[i]).Layout.Selected
    then continue;
// Ausführbare Programme suchen
  if PProgram(Diagram.SortedList[i]).Layout.ParentType = 'P' then begin
    p := PProgram(Diagram.SortedList[i]);
    strin := "";
    strout := "";
    if p.DOCK_L <> nil then strin := p.DOCK_L.Filename;
    if p.DOCK_R <> nil then strout := p.DOCK_R.Filename;
    if p.DOCK_U <> nil then begin // With Interpreter
      s := p.Filename;      // Program CMD
      ReplaceInOut(strin,strout,s);
      paras := GetParameter(s);
      strin := s; strout := "";
      s := p.DOCK_U.ExeFile; // Interpreter CMD
      ReplaceInOut(strin,strout,s);
      s := s + ' '+ Paras;
      AddElement(s);
    end;
    if p.RunningOn <> "" then begin // Without Interpreter
      s := p.Filename;
      ReplaceInOut(strin,strout,s);
      AddElement(s);
    end;
  end;
// Ausführbare Compiler suchen
  if PProgram(Diagram.SortedList[i]).Layout.ParentType = 'C' then begin
    c := PCompiler(Diagram.SortedList[i]);
    strin := "";
    strout := "";
    if c.DOCK_L <> nil then begin
      strin := FindCompilerInputFileName(c);
    end;
    if c.DOCK_R <> nil then begin
      if C.DOCK_R.Layout.ParentType = 'P' then begin
        strout := c.DOCK_R.Filename;
      end;
    end;
  end;
end;
```

```
if C.DOCK_R.Layout.ParentType = 'C' then begin
  if PCompiler(PCompiler(C.DOCK_R).DOCK_L) = C then begin
    s := C.OutputForm;
    ReplaceInOut(strin,'temp.tmp',s);
    strout := s;
  end else begin
    strout := PCompiler(c.DOCK_R).Filename;
  end;
end;
end;
end;
if (c.DOCK_L <> nil)and(((pos('%outfile',LowerCase(c.OutputForm))<>0)and
  (c.DOCK_R <> nil))or
  (pos('%outfile',LowerCase(c.OutputForm)) = 0))then begin
  if c.DOCK_U <> nil then begin // With Interpreter
    s := c.Filename; // Compiler CMD
    ReplaceInOut(strin,strout,s);
    paras := GetParameter(s);
    strin := s; strout := '';
    s := c.DOCK_U.ExeFile; // Interpreter CMD
    ReplaceInOut(strin,strout,s);
    s := s + ' ' + Paras;
    AddElement(s);
  end;
  if c.RunningOn <> '' then begin // Without Interpreter
    s := c.Filename;
    ReplaceInOut(strin,strout,s);
    AddElement(s);
  end;
end;
end;
end;

// Immer am Ende
outf.Add('ECHO ;Diagram successfully executed. ');
outf.Add('GOTO END ');
outf.Add(' ');
outf.Add(': ERROR ');
outf.Add('ECHO .');
```

```
outf.Add('ECHO ;=====');
outf.Add('ECHO ;There was an error while executing the diagram');
outf.Add(':END');
outf.SaveToFile(filename);
outf.Free;
DoneList.Free;
end;
```

Kurz gefasst kann der obige Quellcode als ein Aufsuchen von ausführbaren Bausteinen (Programme und Compiler) angesehen werden. Die Ein- und Ausgabeparameter müssen gesucht und eingesetzt werden. Der Aufruf wird mit der Hilfsfunktion AddElement in ECHO Anweisungen verpackt. Einige Interpreter (wie Chez Scheme) verstehen Pfadangaben mit nur einem Backslash nicht. AddElement ersetzt aus diesem Grund einfache Backslashes durch doppelte (diese Schreibweise wurde von allen getesteten Interpretern und Compilern akzeptiert).

## **TDiag Compiler / Interpreter Presets**

Die in TDiag verwendeten Compiler- und Interpreterbausteine benötigen verschiedene Einstellungen, um anschließend eine ausführbare Datei erzeugen zu können. Um den Aufwand für den Anwender zu senken, bietet TDiag einige Voreinstellungen für bekannte Interpreter. An diesem Quellcodeauszug, der diese Presets verwaltet, kann man zudem gut die Parameter und Platzhalter erkennen (fett dargestellt).

```
// Compiler Presets
if PProgram(CompilerSelectWin.Tag).Layout.ParentType = 'C' then begin
  c := PCompiler(CompilerSelectWin.Tag);
  if CompilerSelect.ItemIndex = 0 then begin // Ausführbarer MyCompiler
    C.InputLanguage := Diag.FindLanguage('Unknown');
    C.OutputLanguage:= Diag.FindLanguage('Unknown');
    C.WrittenIn     := Diag.FindLanguage('Binary Executables');
    C.RunningOn    := 'M';
    C.FileName     := 'MyCompiler.EXE %InFile %OutFile';
    C.OutputForm   := '%OutFile';
    C.Name         := 'MyCompiler';
  end;
```

```
if CompilerSelect.ItemIndex = 1 then begin // nur Quellcode MyCompiler
  C.InputLanguage := Diag.FindLanguage('Unknown');
  C.OutputLanguage:= Diag.FindLanguage('Unknown');
  C.WrittenIn     := Diag.FindLanguage('Unknown');
  C.RunningOn    := "";
  C.FileName      := "";
  C.OutputForm    := "";
  C.Name          := 'MyCompiler';
end;

if CompilerSelect.ItemIndex = 2 then begin // Java Compiler
  C.InputLanguage := Diag.FindLanguage('Java');
  C.OutputLanguage:= Diag.FindLanguage('Java Bytecode');
  C.WrittenIn     := Diag.FindLanguage('Binary Executables');
  C.RunningOn    := 'M';
  C.FileName      := 'javac.exe %InFile';
  C.OutputForm    := '%InFileNoExt.class';
  C.Name          := 'Java Compiler';
end;

if CompilerSelect.ItemIndex = 3 then begin // C# .NET Compiler
  C.InputLanguage := Diag.FindLanguage('C#');
  C.OutputLanguage:= Diag.FindLanguage('Binary Executables');
  C.WrittenIn     := Diag.FindLanguage('Binary Executables');
  C.RunningOn    := 'M';
  C.FileName      := 'csc.exe /out:%OutFile %InFile';
  C.OutputForm    := '%outfile';
  C.Name          := 'C# Compiler';
end;

if CompilerSelect.ItemIndex = 4 then begin // eigener Scheme Compiler
  C.InputLanguage := Diag.FindLanguage('Unknown');
  C.OutputLanguage:= Diag.FindLanguage('Unknown');
  C.WrittenIn     := Diag.FindLanguage('Scheme');
  C.RunningOn    := "";
  C.FileName      := 'compiler.ss %InFile %OutFile';
  C.OutputForm    := '%outfile';
  C.Name          := 'Scheme Compiler';
end;
end;
```

```
// Interpreter Presets
if PProgram(CompilerSelectWin.Tag).Layout.ParentType = 'I' then begin
  i := PInterpreter(CompilerSelectWin.Tag);
  if CompilerSelect.ItemIndex = 0 then begin
    i.InputLanguage := Diag.FindLanguage('Unknown'); // eigener Interpreter
    i.WrittenIn := Diag.FindLanguage('Binary Executables');
    i.RunningOn := 'M';
    i.ExeFile := 'MyInterpreter.EXE %InFile';
    i.Name := 'MyInterpreter';
  end;
  if CompilerSelect.ItemIndex = 1 then begin // Java Interpreter
    i.InputLanguage := Diag.FindLanguage('Java Bytecode');
    i.WrittenIn := Diag.FindLanguage('Binary Executables');
    i.RunningOn := 'M';
    i.ExeFile := 'java.EXE -cp %InFilePath %InFileNoExtNoPath';
    i.Name := 'Java Interpreter';
  end;
  if CompilerSelect.ItemIndex = 2 then begin // Perl Interpreter
    i.InputLanguage := Diag.FindLanguage('Perl');
    i.WrittenIn := Diag.FindLanguage('Binary Executables');
    i.RunningOn := 'M';
    i.ExeFile := 'perl.EXE %InFile';
    i.Name := 'Perl Interpreter';
  end;
  if CompilerSelect.ItemIndex = 3 then begin // Python Interpreter
    i.InputLanguage := Diag.FindLanguage('Python');
    i.WrittenIn := Diag.FindLanguage('Binary Executables');
    i.RunningOn := 'M';
    i.ExeFile := 'python.EXE %InFile';
    i.Name := 'Pyhton Interpreter';
  end;
  if CompilerSelect.ItemIndex = 4 then begin // Scheme Interpreter
    i.InputLanguage := Diag.FindLanguage('Scheme');
    i.WrittenIn := Diag.FindLanguage('Binary Executables');
    i.RunningOn := 'M';
    i.ExeFile := 'petite --script %InFile';
    i.Name := 'Scheme Interpreter';
  end; end;
```

Die enthaltenen MyCompiler und MyInterpreter sind dabei Vorbereitungen für selbst erstellte Programme. Diese bilden somit die Grundeinstellungen und der Anwender muss anschließend Anpassungen vornehmen. Über den Index einer ComboBox (CompilerSelect) wird der entsprechende Abschnitt im obigen Quellcode abgearbeitet. Da die Programmiersprachen in einem Baustein nur als Pointer festgelegt werden, wird die Funktion FindLanguage verwendet. Voraussetzung für die Funktion dieser Presets ist jedoch, dass die entsprechenden Compiler / Interpreter im System über die Path Variable gefunden werden können. Sollten diese Eintragungen fehlen, muss vom Anwender manuell der absolute Pfad hinzugefügt werden.

## Anhang D: Datenträgerbeschreibung

Auf der beiliegenden CD befinden sich die Quelldateien für das gesamte AtoCC Projekt. Welche Dateien dies im Einzelnen sind, soll das nachfolgende Verzeichnis aufzeigen.

<i>Diplomarbeit in elektronischer Form</i>	: \Diplomarbeit.doc : \Diplomarbeit.pdf
<i>Inno Setup Installation (Freeware)</i>	: \Inno Setup-5.1.6.exe
<i>AtoCC Setup Datei (Inno Setup 5.1.6)</i>	: \AtoCC.iss
<i>AtoCC Installationsdateien</i>	: \Output
<i>AutoEdit Quellcodedateien (Delphi 5)</i>	: \AutoEdit
<i>AutoEdit Installationsdateien</i>	: \AutoEdit\Output
<i>TDiag Quellcodedateien (Delphi 5)</i>	: \ T-Diag
<i>TDiag Installationsdateien</i>	: \ T-Diag \Output
<i>VCC Quellcodedateien (Delphi 5)</i>	: \ VCC
<i>VCC Installationsdateien</i>	: \ VCC \Output
<i>SchemeEdit Quellcodedateien (Delphi 5)</i>	: \ SchemeEdit
<i>SchemeEdit Installationsdateien</i>	: \ SchemeEdit \Output

In den Unterverzeichnissen der Werkzeuge befinden sich noch verwendete Grafiken und Beispieldateien. In der Summe sind dies über 600 Dateien, die hier nicht individuell aufgelistet werden können.

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Görlitz, 07.07.2006

---

Michael Hielscher