Erstellung eines visuellen Editors zur automatisierten Erzeugung von Skripten

im SPLICE Projekt

D i p l o m a r b e i t

an der Hochschule für angewandte Wissenschaften Hof
Fakultät Informatik/Technik
Studiengang Technische Informatik

Vorgelegt bei

Prof. Dr. (USA) R. Lano
Alfons-Goppel-Platz 1
95028 Hof

Vorgelegt von

Dominik Reichelt
Vogtlandstraße 11
95152 Selbitz

Hof, den 20. Juni 2007

Das Software-Projekt *Lauts* ist ein System zur generischen automatisierten Durchführung von Testabläufen. Diese können relativ statisch über eine graphische Benutzerschnittstelle konzipiert und erstellt werden. Zusätzlich besteht die Möglichkeit, selbst geschriebene Skripte zu verwenden um komplexere Testabläufe zu realisieren. Da die Benutzer des Systems jedoch nicht zwangsläufig über die notwendigen Kenntnisse zur Erstellung solcher Skripte verfügen ist es notwendig, auch eine benutzerfreundliche Eingabemethode zur Verfügung zu stellen.

Hier setzt die Diplomarbeit *„Erstellung eines visuellen Editors zur automatisierten Erzeugung von Skripten“* an. Das Hauptziel besteht darin, eben diese benutzerfreundliche Eingabemethode für Skripte zur Verfügung zu stellen. Sie werden aus einzelnen graphischen Komponenten zusammengestellt, die dann per Knopfdruck in ein ausführbares Skript umgewandelt werden. Neben der Unterstützung für allgemeine Konstrukte prozeduraler Skriptsprachen (Schleifen, Verzweigungen etc.) werden weiterhin besondere anwendungsspezifische *funktionale Konstrukte* unterstützt. Diese dienen unter anderem dazu, einzelne *Funktionen* der *Lauts* - Software (wie beispielsweise das Absetzen eines Befehls an das *SUT*) auszuführen. Außerdem muss es möglich sein, selbst funktionale Skriptbausteine definieren zu können um trotz der graphischen Darstellung eine Übersichtlichkeit des Skriptablaufs zu wahren.

Die eigentliche Aufgabenstellung der Softwarekomponente ist zweigeteilt. Zum einen liegt das Augenmerk auf einer anwenderfreundlichen graphischen Darstellung und Bearbeitung der Skripte mit Hilfe des Editors, zum anderen jedoch auch auf der Erzeugung des eigentlichen ausführbaren Quelltexts des so erstellten Skriptes. Zusätzlich muss es möglich sein, die aufgebauten Graphen, welche die Semantik des Skriptes repräsentieren, sinnvoll (und damit auch reproduzierbar) in der Datenbank der Anwendung hinterlegen zu können.

Zur Realisierung dieser Anforderungen wird bei der Implementierung *Java (J2SE)* verwendet. Hauptgrund hierfür ist die Integration in das Gesamtsystem, welches auch in Java entwickelt wurde. Weitere verwendete Bibliotheken sind an den entsprechenden Stellen der Diplomarbeit näher erläutert.

Inhaltsverzeichnis

Darstellungsverzeichnis

Abkürzungsverzeichnis

CVS ..... Concurrent Versions System

DOM ..... Document Object Model

DTD ..... Document Type Definition

HMI ..... Human Machine Interface – Mensch-Maschine-Schnittstelle

J2SE ..... Java 2 Platform, Standard Edition

Lauts ..... Loewe Automated Testing System

MSR-Technik ..... Mess-, Steuerungs- und Regelungstechnik

PAP ..... Programm-Ablauf-Plan

SPLICE ..... Software Platform for Intelligent Consumer Electronics

SUT ..... System under Test – Das Objekt, welches von *Lauts* getestet wird

SVN ..... Subversion

UML ..... Unified Modelling Language

XML ..... Extensible Markup Language

XSLT ..... Extensible Stylesheet Language Transformations

1 Einleitung

1.1 Das Softwaresystem Lauts

Lauts - das *Loewe Automated Testing System* - dient prinzipiell zum automatisierten Durchführen von Gerätetests. Hierzu werden in einem Editor bestimmte Testfälle konstruiert, die dann im Anschluss mit dem *SUT* - dem zu testenden Gerät - durchlaufen werden. Die konstruierten Testabläufe werden in einer Datenbank hinterlegt und stehen jederzeit zur weiteren Bearbeitung und Ausführung zur Verfügung. Die Software unterstützt zwei unterschiedliche Betriebsmodi - zusätzlich zum normalen Editieren und Ausführen besteht die Möglichkeit, den *Direct Mode* zu benutzen. Dieser dient in erster Linie zum Testen der aufgebauten Tests selbst und wird prinzipiell nicht massenhaft ausgeführt. Im Gegensatz zu dieser Betriebsart ist es im Normalbetrieb nur möglich, komplette Teststrukturen - sogenannte *TestSuites* auszuführen anstelle kleinerer Untereinheiten.

Merkmale der Software sind unter anderem:

- **Verteilte Systemarchitektur:** Das Softwaresystem *Lauts* besteht nicht nur aus einer einzelnen Anwendung zur Bearbeitung und Ausführung der Testfälle sondern ist als verteiltes System mehrerer größerer Einheiten konzipiert. Die einzelnen Bestandteile (die teilweise auch auf ein und demselben Rechner laufen könnten) sind in Abbildung 1.1 dargestelllt.

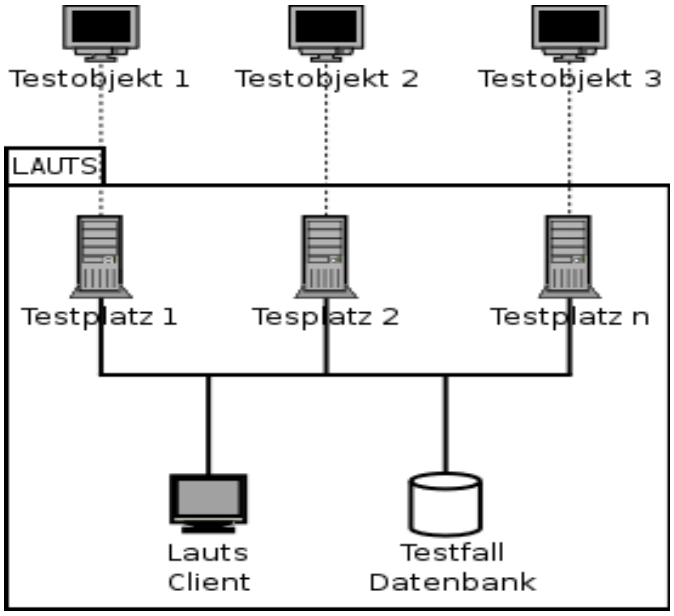


Abbildung 1.1: Grundaufbau des *Lauts*-Systems

- **Graphische Konstruktion der Testabläufe:** Der Editor für die *TestSuites* bietet in erster Linie die Möglichkeit, ein Diagramm des Ablaufs zu bearbeiten. Dieses wird ansprechend und übersichtlich hinsichtlich der Positionierung der einzelnen Objekte dargestellt. Es bildet die Grundlage für den Aufbau der Tests und dient bei einer überwachten Ausführung derselben der Verlaufskontrolle durch den Benutzer. So ist es für den Ersteller eines Tests leicht möglich, den entsprechenden Ablauf am zu testenden Gerät (*SUT*) nachzuvollziehen und entsprechend auftretenden Fehlern des Geräts oder auch des konzipierten Testablaufs entgegenzuwirken.

- **Automatisierte Ausführung der Testabläufe:** Sind die Tests erst einmal konzipiert und in dem graphischen Editor aufgebaut, werden sie in der Datenbank hinterlegt. Vorliegende Tests können dann massenhaft an den Testplätzen ausgeführt werden. Einzig der Austausch der Testgeräte ist dann noch erforderlich. Das Fehlschlagen eines Tests (d.h. das zu testende Gerät scheint defekt zu sein) wird entsprechend protokolliert und der Tester benachrichtigt. Daraufhin kann das Gerät dann nachgebessert oder ausgesondert werden.

- **Flexibilität und Modularität:** Eines der Hauptkriterien bei der Erstellung der Software war von Anfang an die Unterstützung verschiedener möglicher Kommunikationsprotokolle zum *SUT*. Diese Anforderung schlug sich in der Implementierung

einer entsprechenden Zwischenschicht nieder. Dieser Grundgedanke zog sich jedoch auch weiterhin durch die Implementierung. So ist mittlerweile nicht nur die Kommunikationsschicht zum *SUT* austauschbar, sondern auch die Diagrammdarstellung der Teststruktur oder die für komplexere Vorgänge verwendete Skriptsprache.

Wie schon erwähnt ist es für die Darstellung komplexer Testvorgänge notwendig und auch möglich, ein entsprechendes Skript zu erstellen, das die geforderte Funktionalität bietet. Dieses Vorgehen fordert allerdings vom Benutzer ein tiefgreifenderes Verständnis der Programmierung in der jeweiligen Skriptsprache zusätzlich zum grundlegenden Verständnis des zu konstruierenden Testablaufs. Um diese Anforderungen an den Benutzer zu entkoppeln ist es nun sinnvoll, einen graphischen, sprachunabhängigen Editor für Skripte zu projektieren und in das gesamte Softwaresystem zu integrieren.

1.2 Detaillierte Zielbeschreibung

1.2.1 Aufgaben von Skripten

An dieser Stelle soll zunächs ein Einblick in die eigentliche Notwendigkeit für die Verwendung von Skripten in der *Lauts* - Software gegeben werden. Basisgrund für ihre Einführung ist die relative Starrheit der normalen Teststruktur in Bezug auf komplexere Vorgänge. So lassen sich zum Beispiel recht einfache Testfälle sehr schlecht oder gar nicht in der Baumartigen Struktur der *Lauts*-Tests abbilden. Hierzu gehören unter anderem:

- **Zufälliges Setzen von Kommandowerten:** Beim Testen der Geräte kann es nützlich sein, diese mit zufälllig gewählten Werten eines bestimmten Werteintervalls zu versorgen. Somit wäre es beispielsweise möglich, einen Fernseher zufällig 50 mal den Programmplatz wechseln zu lassen und die hierfür benötigte Zeit auszuwerten.

- **Schleifen mit Abbruchbedingungen:** Des weiteren kann es sinnvoll sein, ein Gerät einen bestimmten Test ausführen zu lassen bis eine bestimmte Endbedingung eintritt. So wäre es beispielsweise möglich, einen Fernseher alle Helligkeitswerte von

0 bis 100 zu durchlaufen ohne für jeden Wert ein eigenes Element in der Teststruktur definieren zu müssen.

- **Dynamische Verzweigungen der Testausführung:** Außerdem kann es sinnvoll sein, dynamisch innerhalb der Teststruktur zu verzweigen. Dies kann bedeuten, dass ein nachfolgender Test nicht mehr sinnvoll ist, sobald ein vorhergehender Test fehlgeschlagen ist. So wäre es zum Beispiel nicht angebracht, bei einem Gerät dass kein Ausgabebild liefert, die Übernahme von Helligkeitseinstellungen zu testen.

- **Allgemeine dynamische Ablaufsteuerung anhand vorhergehender Tests:** Ähnlich wie beim vorhergehenden Punkt kann es notwendig sein, anhand der Ergebnisse vorheriger Tests dynamisch zu verzweigen. Das generelle Vorgehen ist ähnlich, jedoch dient als Basis für die Auswertung nicht das Ergebnis des gerade ausgeführten Tests, sondern ein weiter zurück liegendes.

Zusätzlich zu den hier aufgeführten Beispielen existieren sicherlich noch weitere Problemstellungen, die sich mit Hilfe von Skripten im Gegensatz zur regulären Teststruktur (besser) lösen lassen. Somit stellt die Möglichkeit der Verwendung von Skripten einen großen Schritt in Richtung Erweiterbarkeit und Dynamik der *Lauts*-Software dar.

1.2.2 Benutzerfreundliche Handhabung

Da die normale Eingabe von Skripten ein zumindest grundlegendes Verständnis allgemeiner Programmierung erfordert, ist es sachdienlich, unbedarfteren Benutzern eine einfachere und übersichtlichere Möglichkeit zur Verfügung zu stellen auch komplexere Abläufe zu visualisieren und zu erstellen. Es muss möglich sein, alle Funktionen der *Lauts* - Software aus der graphischen Benutzerschnittstelle anzusteuern und hinsichtlich ihres Ablaufs leicht zu erkennen.

Es existiert keine allgemeingültige „beste“ Darstellung von Programmabläufen. Deshalb ist es sinnvoll, zwischen mehreren verschiedenen Darstellungsvarianten wechseln zu können.

Dies ermöglicht es, für das jeweilige Problem die entsprechend besser geeignete Diagrammart zu wählen, die den selben Sachverhalt ausdrückt. Weitere Informationen zu unterschiedlichen Darstellungsarten finden sich in Abschnitt 1.4. Die Anordnung der einzelnen Elemente des Skriptes sollte möglichst automatisch vonstatten gehen können, um dem Benutzer feste Orientierungspunkte hinsichtlich des Ablaufs geben zu können.

Im Allgemeinen jedoch sollten Darstellung und Handhabung so intuitiv wie möglich konzipiert sein, um einen leichten Einstieg in die Benutzung des Editors zu gewährleisten. Ein weiterer Aspekt, der Berücksichtigung verdient sind Standardaktionen wie beispielsweise *Ausschneide-*, *Einfüge-* und *Lösch-* Operationen, die ähnlich wie in anderen Editoren funktionieren müssen.

1.2.3 Generierung sicheren ausführbaren Quellcodes

Der erzeugte Quellcode muss immer ausführbar sein, sowie einige weitere Richtlinien erfüllen:

- **Vermeidung von Endlosschleifen:** Um dieses Ziel zu erreichen ist es notwendig, direkte Sprünge innerhalb des Skriptes zu verbieten. Dies führt zu einer besseren Strukturiertheit des Skriptes und dient der Lesbarkeit des erstellten Diagramms.

- **Unterstützung gängiger Konstrukte:** Um ein effektives Skript erstellen zu können ist es notwendig, gängige Sprachkonstrukte höherer Programmiersprachen zu unterstützen. Hierzu gehören Schleifen, Verzweigungen oder auch für die Auswertung vorheriger Ergebnisse benötigte logische Operatoren wie *UND* und *ODER*. Mit ihrer Hilfe ist es möglich, auch komplexere Bedingungszusammenhänge darzustellen.

- **Unterstützung der *Lauts*-Spezifischen Konstrukte:** Hierzu gehören Möglichkeiten zur Ansteuerung von Variablen des *Lauts*-Systems, zum Ausführen von Elementen der *Lauts*-Teststruktur sowie zur Ausgabe von Texten (beispielsweise zur Fehlererkennung) zum Client hin beziehungsweise in das Testprotokoll.

Um diese Bedingungen einhalten zu können, muss es möglich sein, das erstellte Skript hinsichtlich einzelner Punkte automatisch zu validieren. So wäre es beispielsweise möglich, zyklische Verbindungen der Elemente (wie zum Beispiel in Programmablaufplänen) zu unterbinden.

1.3 Betrachtung der Skriptsemantik

Die Aufgaben der Skripte im Software-System *Lauts* sind nun klar umrissen. Hieraus ergeben sich Konsequenzen für das ihnen zugrunde zu legende Programmierparadigma. Auf ein objektorientiertes Design der Skripte wird bewusst verzichtet, um das Hauptaugenmerk des Anwenders auf den eigentlichen Ablauf des Skriptes zu lenken. Dennoch ist es notwendig, dass die verwendeten Skriptsprachen mit Objekten umgehen können um die Integration in das *Lauts*-System gewährleisten zu können. Somit müssen die Skripte ein *objektbasiertes* Programmmierparadigma nach (HV07, Seite 14) erfüllen. Die verwendeten Skriptsprachen müssen nicht in der Lage sein, Konzepte der objektorientierten Programmierung wie Klassen und Vererbung anzuwenden, sondern nur den Ersteller befähigen, derartige Konzepte im vorhandenen System zu nutzen.

1.3.1 Allgemeine Sprachkonstrukte

Diese Sprachkonstrukte dienen zur allgemeinen Kontrollflusssteuerung des Skriptes. Sie dienen der Abbildung des konzipierten Ablaufs in eine ausführbare Form.

Kommentare: Kommentare dienen der Strukturierung und Dokumentation des Quelltextes. Sie sind ein unabdingbares Kriterium von Programmiersprachen und müssen in irgend einer Form unterstützt werden.

Elementare Anweisungen: Anweisungen sind die Grundbausteine der Skripte. Sie beschreiben, *was* ausgeführt werden soll. Das einfachste Skripte besteht aus einer einzelnen Anweisung. Hierbei ist zu beachten, dass solche Anweisung prinzipiell

von der verwendeten Skriptsprache abhängig sind und nicht abstrakt repräsentiert werden können.

Verzweigungen: Verzweigungen dienen der Steuerung des Kontrollflusses. Sie basieren auf der Auswertung einer bestimmten Bedingung (z.B. `x == 5`). Die Evaluation des Ausdrucks ergibt dann einen binären Wert (*wahr/falsch*), der festlegt, welcher der nachfolgenden Pfade weiterverarbeitet wird. Auch Mehrfachverzweigungen, bei denen mehrere Ausgangspfade existieren, sind möglich, jedoch nicht unbedingt notwendig und abhängig von den verwendeten Datentypen.

Schleifen: Schleifen dienen zur Iteration / Wiederholung eines bestimmten (inneren) Ablaufpfades bis eine bestimmte Bedingung eintritt. Diese Bedingungsprüfung kann entweder vor einer möglichen ersten Ausführung (*while*) oder danach (*do-while*) erfolgen. Zählerschleifen (*for*) stellen nur eine Sonderform von *while*-Schleifen dar, können jedoch durch ihre modifizierte Darstellung der Übersichtlichkeit des Skriptes zuträglich sein.

Variablenzuweisungen: Variablenzuweisungen stellen einen Sonderfall von Elementaren Anweisungen dar. In vielen heutigen Programmiersprachen müssen Variablen nicht mehr am Anfang des Quelltextes definiert werden (wie z.B. in C). Dennoch ist es wichtig, ihre Definition und ihre Zuweisung getrennt zu betrachten, da die Deklaration Aufschluss über den Gültigkeitsbereich der Variablen gibt.

Funktionsaufrufe: Funktionsaufrufe sind in allen Sprachen möglich. Sie dienen zur Unterstrukturierung des Ablaufs. Somit ist es möglich, komplexere Systeme aus einzelnen „Bausteinen“ aufzubauen.

Vergleichsoperationen: Vergleichsoperationen dienen prinzipiell zur „Umwandlung“ von konkreten Zahlenwerten in bool'sche Ausdrücke beziehungsweise Bedingungen. Somit lassen sich Eingangswerte zur Steuerung des weiteren Programmablaufs nutzen. Auch solche Vergleiche sind in allen gängigen Skriptsprachen vorhanden und müssen unterstützt werden.

Des weiteren ist die Benutzung von Variablen essentiell für einen kontrollierten Programmablauf. Es werden folgende Variablenarten berücksichtigt:

- Bool'sche Variablen: Sie bestehen nur aus den Elementarwerten wahr/falsch und dienen hauptsächlich der Flusskontrolle des Programmablaufs.

- Ganzzahlige Variablen: Dieser Variablentyp dient zur Flusssteuerung bei iterativen Schleifen.

- Fließkommazahlen: Sie können zum Beispiel die Ausgabe eines Zufallsgenerators oder etwas Ähnlichem darstellen und durch „Streckung" in einen zufälligen ganzzahligen Wert eines bestimmten Bereichs umgewandelt werden.

- Zeichenketten: Strings dienen primär der strukturierten Ausgabe und Protokollierung des Skriptverlaufs.

1.3.2 Lauts-spezifische Sprachkonstrukte

Knoten der Teststruktur ausführen: Dies bewirkt eine Ausführung des übergebenen Testknotens durch die TestEngine. Das hierbei erzeugte *Test-Resultat* wird als Rückgabewert an das Skript zurückgegeben und kann im Anschluss an die Ausführung evaluiert werden.

Ausgabemeldungen: Ausgabemeldungen stellen sich innerhalb des *Lauts*-Systems komplizierter dar als bei einer generellen Ausgabe auf die Kommandozeile. Dies liegt in der verteilten Architektur des Systems begründet. Ausgabemeldungen, die von der TestEngine generiert werden, werden sowohl an eventuell verbundene Clients weitergegeben, als auch im Ausführungsprotokoll abgelegt. Diese Komplexität rechtfertigt ihre Repräsentation als einzelne Skriptbausteine.

Variablenübernahme aus der Teststruktur: Es muss möglich sein, bereits in der TestEngine bestehende Variablen (wie z.B. vorhergehende Testergebnisse) in das

zu erstellende Skript zu übernehmen. Dies ermöglicht es, eine skriptgestützte Auswertung der Variablen vorzunehmen und auch anhand ihrer Werte den folgenden Testablauf zu beeinflussen.

Variablendeklaration und Wertzuweisungen: Des weiteren muss es möglich sein, innerhalb des Skripts neue (nur innerhalb des Skriptes gültige) Variablen zu deklarieren und ihre Werte zu modifizieren. Dies kann zum Beispiel der Iteration und Verzweigung des Kontrollflusses dienen. Sollen Variablen ihren Wert über das Skript hinaus behalten (wenn sie z.B. in einem anderen Skript weiter ausgewertet werden sollen) ist es notwendig, die Variablen in einem übergeordneten Knoten der Teststruktur zu hinterlegen. Die dort vorliegenden Variablen können dann ausgewertet und verändert werden (s.o.).

Weitere benutzbare Funktionsbausteine: Hierzu gehören beispielsweise Zufallsgeneratoren, die Werte aus einem bestimmten Wertebereich zurückliefern oder auch andere Bausteine, die *Lauts*-Testelemente manipulieren.

Weiterhin ist es notwendig, *Lauts*- interne Variablentypen zu benutzen. Dies sind im Prinzip Objekte und sind die eigentliche Begründung, warum die zu verwendende Skriptsprache *objektbasiert* sein muss. Hierzu gehören:

- **Testresultate:** Diese Objekte stellen das eigentliche Ergebnis eines Tests des Gerätes dar. Sie enthalten unter anderem Informationen darüber, ob ein Test erfolgreich ausgerührt wurde und dienen der nachfolgenden Auswertung durch das Skript.

- **Kommandoobjekte:** Kommandoobjekte stellen die eigentlichen, an das *SUT* geschickten Kommandos dar. Sie müssen durch Skripte verändert werden können, um einen dynamischen Testaufbau zu ermöglichen. Zum Beispiel wäre es so möglich, das zu testende Gerät auf ein zufällig gewähltes Programm schalten zu lassen. Besondere Beachtung hierbei erfordert die Tatsache, dass Kommandoobjekte – genauer gesagt die Protokolle zur Übertragung von Kommandos – austauschbar sind. Dies hat zur Konsequenz, dass sie nicht direkt verwendet werden sollten. Eine Benutzung der seit dem Prototyp neu eingeführten *Funktionselemente* beziehungsweise eine Modifikati-

on ihrer Parameter ist an dieser Stelle angebracht um die Wiederverwendbarkeit der Skripte auch bei einem Wechsel des Kommunikationsprotokolls zu gewährleisten.

- **Elemente der Teststruktur:** Natürlich müssen auch alle anderen Elemente der *Lauts*-Teststruktur in Skripte eingebaut werden. Einfachstes Beispiel hierfür ist die Einbindung eines hierarchisch weiter unten liegenden Testobjekts in das Skript um zu definieren, dass an dieser Stelle genau der ausgewählte Test ausgeführt werden soll.

- **Schnittstellenelemente der TestEngine:** Diese Elemente stellen bestimmte Mechanismen der Testengine zur Verfügung. Beispielhaft hierfür wäre die Ausgabe eines Skriptes. Da dieses innerhalb der Testengine ausgeführt wird, muss es ihr direkt sagen, dass an dieser Stelle eine Ausgabe einer Meldung erfolgen soll. Die eigentliche Ausgabe der Meldung an den Client und in das Testprotokoll übernimmt im Anschluss daran dann die Testengine.

1.3.3 Metasprachen und Übersetzung

Da bei der Verwendung des *Lauts*-Systems keine Skriptsprache die eindeutige Präferenz trägt ist es notwendig, bei der Erzeugung von Skripten eine sprachunabhängige Darstellungsform zu benutzen. Dies lässt sich durch die Verwendung einer Metasprache wie z.B. XML realisieren. Dieses Vorgehen birgt deutliche Vorteile. So ist es beispielsweise möglich, durch generische Verfahren zum einen eine korrekte Abfolge von Elementen zu gewährleisten – dies geschieht entweder mittels *XML Schema* oder eine *DTD* . Des weiteren besteht auf diese Art das Potential, den erzeugten Meta-Quelltext des Skriptes direkt mittels *XSLT* in den korrespondierenden Skriptquelltext umzuwandeln, ohne einen weiteren Zwischenschritt über ein semantisches Modell zu gehen. Dies hätte den Vorteil einer relativ leichten Einführung einer neuen Skriptsprache. Hierbei wäre es lediglich notwendig, ein neues *Template* – eine neue Vorlage – zu erstellen und mit den Daten der semantischen Struktur zu befüllen. Auch viele andere Darstellungsarten für die Skripte wären durch die Verwendung von XML als Zwischensprache denkbar wie z.B. eine baumartige Darstellung.

Es stellt eine Herausforderung dieser Arbeit dar, eine passende Darstellungsweise in einer Metasprache zu finden, die den Anforderungen der Skripte gerecht wird.

Ein Beispiel für die Verwendung von XML zur Definition von Programmabläufen findet sich in MetaL[1], einem erweiterbaren Code-Generator für Java, PHP und Perl. Dieser traversiert das zugrundeliegende XML-Dokument und wandelt dieses in den entsprechenden Quellcode der Zielsprache um.

Dieses allgemeine Konzept der automatischen Codeerzeugung ermöglicht es auch, das Abstraktionsniveau der Programmierung stark anzuheben. So ist es beispielsweise einfach möglich, einen Code-Generator zu entwickeln, der wiederum die Eingabe für einen weiteren Generator erzeugt. Eine Fallstudie hierfür ist in (Her03, S. 3-15) zu finden, welches eindrucksvoll die Mächtigkeit solcher *Meta-N-Programmierung* schildert.

1.4 Mögliche Darstellungsarten

Da die einzusetzenden Skripte nicht objektorientiert sondern prozedural verarbeitet werden, jedoch trotzdem Objekte ansteuern (z.B. Instanzen der *Java* - Klassen des *Lauts*-Systems) erfüllen sie ein *objektbasiertes* Programmierparadigma (HV07, vgl. Seite 14). Diese Aussage bezieht sich weniger auf die eigentliche Mächtigkeit der verwendeten Skriptsprachen als vielmehr auf ihre konkrete Anwendung innerhalb des Systems. Ihr sequentieller / prozeduraler Charakter soll durch die Darstellung betont werden um dem Zielbenutzer ein möglichst einfaches Verständnis des Skriptablaufs zu vermitteln. Für die Visualisierung der Abläufe existieren bereits Ansätze die im Folgenden näher erläutert werden sollen.

1.4.1 Nassi-Shneiderman Diagramme

Nassi-Shneiderman Diagramme stellen eine Art von Struktogrammen dar. Ihr Hauptaugenmerk liegt auf einer programmiersprachenunabhängigen Darstellung sequentieller

[1] http://www.meta-language.net

Algorithmen und Abläufe. Genormt ist diese Darstellungsart in der DIN-Norm 66261 wie in (DIN07) dargestellt. Die Darstellung erfolgt durch eine Aneinanderreihung bzw. Schachtelung der einzelnen Konstrukte. Es fehlt eine graphische Darstellung in Form von Knoten und Kanten, wie diese in (Kön07) behandelt wird.

Sprachkonstrukte

Sequenz: Dies stellt einen regulären sequentiellen Ablauf von Anweisungen dar und bildet somit die Grundlagen z.B. für Variablenzuweisungen oder auch Funktionsausführung (z.B. dem Starten einer *Funktion* des *Laut*-Systems).

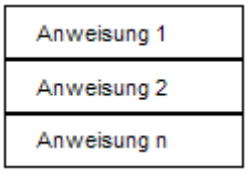


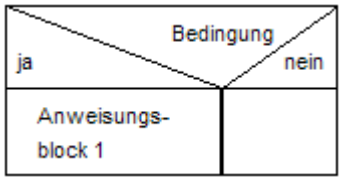
Abbildung 1.2: Sequenzen im Nassi-Shneiderman-Diagram

Verzweigung: Die Verzweigung dient zur Unterscheidung unterschiedlicher nachfolgender Kontrollstrukturen / Anweisungen anhand der Auswertung von Variablen. Die „inneren“ Anweisungsfolgen werden innerhalb der Verzweigung im entsprechenden Zweigfeld dargestellt. Es werden sowohl Einfach- als auch Mehrfachunterscheidungen unterstützt.

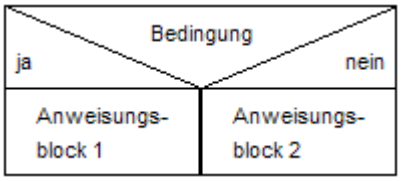
Schleifen: Auch Iterationskonstrukte können verwendet werden. Alle gängigen Formen (*for*-, *while*- und *do-while*) werden als eigenständige Elemente behandelt und können im Diagramm angezeigt werden.

Aussprung: Dieses Konstrukt dient zum Verlassen einer umliegenden Schleife und entspricht einer *break*-Anweisung.

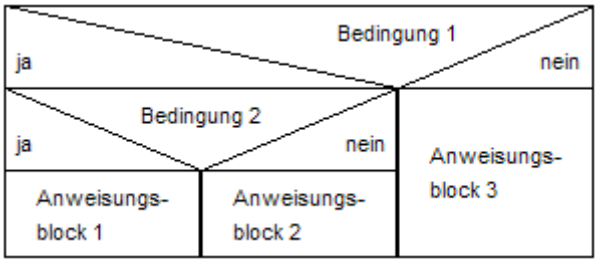
Funktionsaufruf: Um eine übersichtliche Darstellung auch bei komplexeren Abläufen zu ermöglichen besteht die Möglichkeit, Teildiagramme zu Funktionen zusammen-



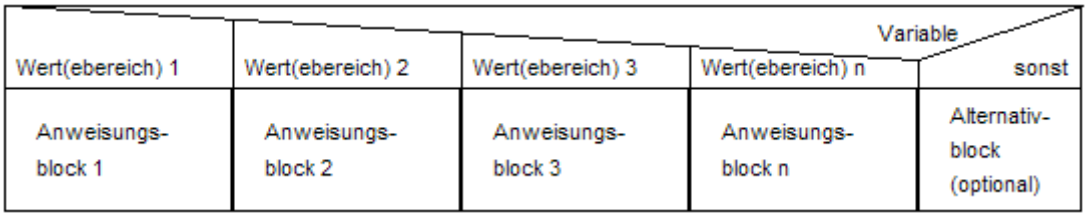
(a) Einfachauswahl



(b) Zweifachauswahl

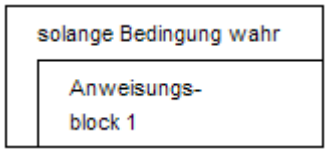


(c) Verschachtelte Zweifachauswahl

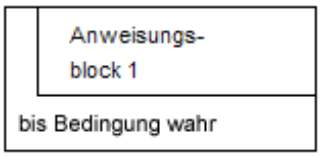


(d) Fallauswahl

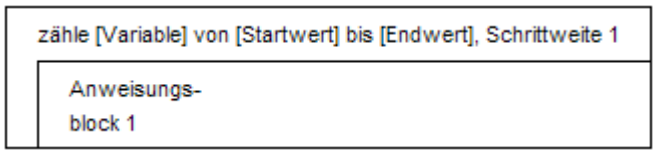
Abbildung 1.3: Verzweigungen im Nassi-Shneiderman-Diagramm



(a) Kopfgesteuert



(b) Fußgesteuert



(c) Zählschleife

(d) Endlosschleife

Abbildung 1.4: Schleifen im Nassi-Shneiderman-Diagramm

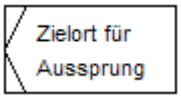


Abbildung 1.5: Aussprung im Nassi-Shneiderman-Diagramm

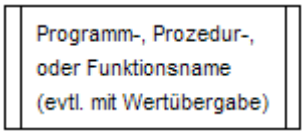


Abbildung 1.6: Funktionsaufrufe im Nassi-Shneiderman-Diagramm

zufassen und in der darüber liegenden Ablaufebene als einzelnes Konstrukt darzustellen.

Diese Strukturkonstrukte reichen aus um Quellcode in beliebigen Sprachen generieren zu können, solange diese einen prozedural orientierten Ansatz vertreten.

Beurteilung

Vorteile:

- Die verwendeten Elemente sind standardisiert, was eine einfache Einarbeitung in das Diagramm ermöglicht.
- Besonders geschachtelte Konstrukten (wie Schleifen) werden recht einfach und deutlich als solche erkkenbar.

Nachteile:

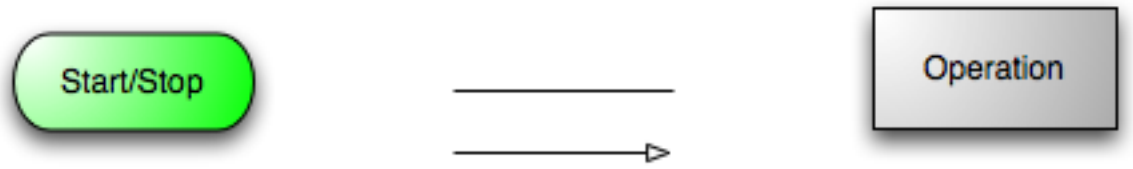
- Dieser Diagrammtyp ist durch die recht starren Vorgaben nur schwer um neue Elemente erweiterbar.
- Obwohl geschachtelte Abläufe recht gut darstellbar sind ist diese Darstellungsart nicht sehr intuitiv. Dies ist besonders durch das Fehlen von Pfeilen bedingt, die noch einmal die Flussrichtung veranschaulichen.
- Es besteht keine Möglichkeit, einen Datenflussablauf anschaulich zu modellieren.

1.4.2 Programmablaufpläne

Programmablaufpläne (*PAP*s) sind in ihrer Darstellungsform in der DIN-Norm 66001 festgelegt. Sie dienen wie auch Nassi-Shneiderman-Diagramme zur Abbildung eines prozeduralen Programmflusses. Es bestehen jedoch fundamentale Unterschiede hinsichtlich der Darstellung, denn ein Programmablaufplan stellt einen Graph dar, der sowohl Elemente als auch gerichtete Kanten enthält. Gerade diese gerichteten Kanten (Pfeile) ermöglichen eine intuitivere Lesart des Programmablaufs. Weiterführende Informationen zu Programmablaufplänen finden sich in (Wes71).

Sprachkonstrukte

Sequenz: Sequenzen werden als Abfolge von Operationen dargestellt. Diese sind durch Linien / Pfeile verbunden um den Kontrollfluss darzustellen. Eine gesonderte Stellung nimmt hierbei der Start-/Stop-Knoten ein, der anders als reguläre Operationen dargestellt wird.



(a) Start/Stop Knoten (b) Verbindungskanten (c) Operationen

Abbildung 1.7: Sequenzen im Programmablaufplan

Verzweigung: Verzweigungen werden als Rauten dargestellt. Die Bedingung, anhand derer verzweigt wird ist in ihrer Fläche dargestellt, die entsprechenden ausgehenden Kanten (*ja/nein*, *wahr/falsch*) sind passend beschriftet.

Funktionsaufruf: Wie auch im Nassi-Shneiderman-Diagramm existiert auch im Programmablaufplan die Möglichkeit, komplexere Abläufe durch Einführung von Unterdiagrammen / -programmen aufzuteilen.

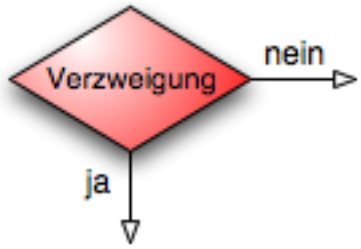


Abbildung 1.8: Verzweigungen im Programmablaufplan

Abbildung 1.9: Unterprogramme im Programmablaufplan

Schleifen: In Programmablaufplänen existiert keine direkte Darstellungsform von Schleifen. Im Allgemeinen werden Schleifen jedoch durch eine rückkoppelnde Kantenverbindung nach einer Verzweigung dargestellt. Dies entspricht in etwa einem bedingten Sprungbefehl und steht in krassem Gegensatz zu Edsger W. Dijkstra's Paradigma der strukturierten Programmierung (Dij68). Es besteht jedoch die Möglichkeit, den Programmablaufplan als Grundlage zu benutzen und ein weiteres (nicht genormtes) Konstrukt einzuführen, welches Schleifen darstellen kann.

Ein-/Ausgabeelemente: Im Unterschied zu Nassi-Shneiderman Diagrammen bieten Programmablaufpläne die Möglichkeit, Ein- und Ausgabeoperationen gesondert hervorzuheben. Dies geschieht unter Verwendung einer Symbolik, die allerdings nicht zur DIN-Norm 66001 von 1983 gehört.

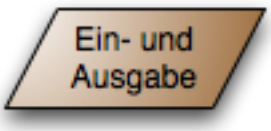


Abbildung 1.10: Ein-/Ausgabeoperationen im Programmablaufplan

Beurteilung

Vorteile:

- Auch bei diesem Diagrammtyp erleichtert die Standardisierung der Elemente den Einstieg für neue Benutzer.
- Die Darstellung des Diagramms wird durch die Verwendung von gerichteten Kanten und Knoten sehr ansprechend durchgeführt. Besonders die Pfeile erlauben eine intuitive Lesart des Ablaufs.
- Das Diagramm kann bei einer Abweichung von der eigentlichen Normdarstellung – insbesondere durch das Hinzufügen eigener, neuer Elemente – gut erweitert werden.
- Dieser Diagrammtyp besitzt weiterhin eine direkte Unterstützung für Kommentare, die einzelnen Elementen zugeordnet werden können. Dies erlaubt eine elementare Dokumentation des Ablaufs.
- Durch die Gruppierung einzelner Ablaufstrukturen zu „Untergraphen“ kann die Übersichtlichkeit des Diagrammes zunehmen.
- Es besteht ein hoher Freiheitsgrad bei der Positionierung der Elemente.

Nachteile:

- Von Haus aus hat eine graphische Darstellung durch das Entstehen von Zwischenräumen zwischen den Elementen einen hohen Platzbedarf.
- Entstehen viele Kanten im Graph, kann das die Übersichtlichkeit des Diagrammes trüben.
- Es besteht keine direkte Darstellungsmöglichkeit für Schleifen.
- Durch eine unvorteilhaft gewählte Positionierung der Elemente kann die Übersichtlichkeit möglicherweise stark beeinträchtigt werden.

1.4.3 Abgrenzung zu Petri-Netzen

Petri-Netze stellen eine Möglichkeit dar, komplexere Vorgänge graphisch darzustellen. Sie sind im Geschäftsumfeld oder auch bei technischen Anlagen verbreitet und besitzen eine tiefgreifende mathematische Grundlage (vgl. (Bal01, S. 346 – 370)) aus dem Bereich der Automatentheorie. Durch die Verwendung gerichteter Kanten ergibt sich eine „fließende “ Lesart des dargestellten Prozesses. Durch entsprechende Erweiterungen ist es sogar möglich, mit Hilfe von Petri-Netzen eine *Touring-vollständige* Darstellung zu erhalten. Sie sind jedoch für die Darstellung der Abläufe in einem Skript nicht oder nur bedingt geeignet, da ihr Hauptaugenmerk anders gesetzt ist. Im einzelnen bezieht sich dies auf:

Gleichzeitigkeit: Petri-Netze legen einen starken Wert auf die Darstellung parallel verlaufender Ereignisketten. Da die mit dem Editor zu bearbeitenden Skripte jedoch ausschließlich prozeduraler Natur sind entfällt dieser Punkt komplett aus der zu wählenden Darstellungsform.

Darstellung der „Stellen“ als eigene Elemente: In Petri-Netzen werden Zustände, die nach dem Ablauf einer Aktion erreicht werden, als sogenannte „Stellen“ dargestellt. Da diese Zustände jedoch den Benutzer von der eigentlichen Planung des *Ablaufs* ablenken würden entfällt ihre Darstellung.

Tokens: Tokens sind in Petri-Netzen die Darstellung für Datenobjekte. Diese werden von Zustand/Stelle zum nachfolgenden Zustand transportiert. Es existieren zwar auch in den zu bearbeitenden Skripten Datenobjekte, jedoch unterscheidet sich ihre Darstellung deutlich.

2 Aktueller Stand der Forschung

Dieser Abschnitt diskutiert den gerade aktuellen Stand der Forschung im Umfeld visueller Programmierung. Es werden Software-Lösungen vorgestellt, die eine teilweise ähnliche Handhabung wie der Skriptgenerator besitzen. Im Anschluss daran wird auf die Unterschiede der Softwareprodukte zu der hier vorliegenden Arbeit eingegangen. Da Visuelle Programmierung allgemein eine Vielzahl möglicher Systeme und „Sprachen“ umfasst, soll an dieser Stelle nur auf die am weitesten verbreiteten Systeme eingegangen werden. (Wik07) gibt einen allgemeinen Überblick über visuelle Programmiersysteme.

2.1 LabVIEW

LabVIEW[1] ist eine weit verbreitete visuelle Programmiersprache von National Instruments. (Kod07) gibt einen knappen Überblick über die Programmierphilosophie der Software. Das Basiskonzept hierbei ist die Verarbeitung von Daten. Diese wird als *Datenfluss* modelliert. Datenquellen und Ziele werden durch Linien miteinander verbunden. Dies resultiert in einer graphischen Darstellung mittels Knoten und Kanten. Die Behandlung der Daten durch „Virtuelle Instrumente“ erfolgt auf ähnliche Art wie dies das Paradigma der Petri-Netze vorschreibt. Die Objekte werden erst dann ausgeführt, wenn alle Dateneingänge mit Eingangsdaten versorgt sind. Auch nebenläufige Vorgänge lassen sich wie in Petri-Netzen umsetzen. Da viele visuelle Programmiersysteme eine ähnliche Aufgabenstellung (insbesondere im Bereich der MSR-Technik) vertreten wie LabVIEW, dieses Produkt jedoch am weitesten fortgeschritten scheint, soll an dieser Stelle besonders auf LabVIEW eingegangen werden.

[1] http://www.ni.com/labview/

Die visuelle Sprache von *LabVIEW* nennt sich *G*. Elemente des *G*-Programms können Datenquellen mehrfach nutzen. Dies veranschaulicht weiterhin das Datenfluss-Paradigma, bringt mit sich aber den Nachteil eines möglichen Verlust der Übersichtlichkeit. Durch die Mehrfachverbindungen entstehen mehr Kanten – was an sich kein Problem wäre, würde dies nicht das Risiko für „Kreuzungen“ der Linien erhöhen. Zusätzlich jedoch kann es sein, dass die Datenquelle teilweise ziemlich weit von ihren Zielen entfernt positioniert ist. Dadurch entstehen sehr lange Kanten, was diese Gefahr noch weiter erhöht.

LabVIEW besitzt mehrere verschiedene Ansichten. Zur Programmieransicht, die einem Flussdiagramm ähnelt, kommt eine weitere Darstellung für eine Benutzerschnittstelle hinzu. Dieses „Frontpanel“ dient der einfachen Erstellung von graphischen Benutzeroberflächen zu dem erstellten Programm. Das eigentliche Programm-Diagramm stellt somit quasi das „Innenleben“ der Anwendung dar, das Frontpanel hingegen die Anzeige- und Steuerelemente, die für den Benutzer sichtbar sein sollen.

Ein weiteres Charakteristikum von LabVIEW ist die Anlehnung der Elemente an Objekte der realen Welt. So existieren beispielsweise Oszilloskop-ähnliche Elemente zur Darstellung von Verlaufsfunktionen und Ähnlichem. Dies erlaubt den Benutzern einen schnellen Einstieg, da die verwendeten Elemente allein schon durch ihre Darstellung gut verständlich sind.

Der hauptsächliche Verwendungszweck von *LabVIEW* sind Anwendungsfälle aus dem Bereich der MSR-Technik. Dies kommt dem Datenflussmodell zur Programmerstellung sehr entgegen. Auch für automatisierte Testumgebungen und Messeinrichtungen kann LabVIEW gut eingesetzt werden. Weitergehende Informationen zu LabVIEW finden sich in (AH04).

Ausgewählte Features sind:

- **Debugging-Werkzeuge:** LabVIEW beinhaltet gängige Werkzeuge – *Step Into*, *Step Over* wie auch das Auskommentieren einzelner Blöcke – zur Fehlererkennung und -Analyse. Dies ist im Prinzip eine Grundvorraussetzung für die Erstellung komplexer Programme.

- **Erzeugung eigenständiger Anwendungen:** Mit Hilfe eines LabVIEW Moduls (dem *Application Builder*) ist es möglich, das erzeugte Programm in eine eigenständige Anwendung umzuwandeln. Dies macht vor allem für den Einsatz des Programmes auf vielen Zielsystemen Sinn.

- **Direkte Ansteuerung von physikalischen Datenquellen:** Viele externe Eingabegeräte zur Datenerfassung werden von LabVIEW direkt unterstützt. Insbesondere Produkte der Firma National Instruments sind natürlich problemlos verwendbar.

- **Module zur Datenanalyse:** Zur Analyse der gesammelten Daten existieren viele verschiedene Module in LabVIEW. Zusätzlich ist eine Anbindung an Software von Drittherstellern wie beispielsweise *MathLab* oder *Simulink* prinzipiell möglich.

- **Unterstützte Plattformen:** Als Betriebssysteme für die entwickelten Programme kommen sowohl Linux, als auch MacOS und Windows in Frage. Auch Embedded Devices – beispielsweise unter Windows CE oder allgemein auf 32-Bit-Mikroprozessoren – können als Zielplattformen für die Anwendungsentwicklung dienen.

- **Reporting:** Auch Funktionaltäten zum Erstellen von Berichten der Durchläufe und der gesammelten Daten sind vorgesehen. So ist es beispielsweise möglich, Übersichtsdiagramme oder Berichte in verschiedenen Ausgabeformaten (beispielsweise HTML) erstellen zu lassen. Auch eine Schnittstelle zu Office-Produkten (*Word* und *Excel*) existiert.

- **Eigene Module:** Auch die Entwicklung eigener Module für LabVIEW ist möglich. Diese können entweder in LabVIEW selbst aus elementareren Bausteinen aufgebaut sein oder durch die Verwendung externer, selbst entwickelter Bibliotheken erzeugt werden.

- **Externe Programmierschnittstellen:** In LabVIEW können externe Bibliotheken eingebunden werden, um Programmbausteine zu erhalten. Zusätzlich ist es möglich, einzelne Programmbausteine aus LabVIEW in eine externe Bibliothek umzuwandeln, die dann in anderen Anwendungen verwendet werden kann.

- **Echtzeitfähigkeiten:** Prinzipiell ist LabVIEW für die Entwicklung von Echtzeitprogrammen geeignet. Auch hierbei sind verschiedene Zielplattformen für die fertige Anwendung möglich. Selbst ein bereits vorhandener handelsüblicher PC kann mittels LabVIEW auf Echtzeitfähigkeiten überprüft und eventuell als Echtzeitsystem für die Anwendung genutzt werden.

Zudem findet sich zu LabVIEW eine große Entwicklergemeinde, die auch untereinander Programmmodule zu unterschiedlichsten Zwecken austauschen.

2.2 Weitere Visuelle Programmiersysteme

2.2.1 Agilent VEE

VEE[2] ist eine visuelle Programmiersprache der Firma Agilent. Sie wurde ursprünglich von HP entwickelt und findet auch in der Hochschule für Angewandte Wissenschaften Hof Verwendung. Sie besitzt eine sehr starke konzeptionelle Ähnlichkeit zu *LabVIEW*, was wohl in den gleichen Anwendungsgebieten begründet liegt. Auch hier existiert eine Programmflussdarstellung als Diagramm sowie eine Benutzeransicht, die die vorher vom Benutzer / Programmierer ausgewählten Elemente beinhaltet.

Auch hier basiert die Programmierung auf dem *Datenflussmodell*. Einzelne Elemente mit mehreren Dateneingängen „schalten“ auch hier erst dann, wenn alle Eingänge mit Daten versorgt sind.

Abbildung 2.1 zeigt ein typisches kleines Programm in der Programmflussdarstellung von *Agilent VEE*. Deutlich zu erkennen hierbei ist die Tatsache, dass das Erstellen des Programmes auch komplett ohne die Verwendung von Kontrollflussverbindungen (sie sind auch in *VEE* vertikale Verbindungslinien) allein durch die Modellierung des Datenflusses möglich ist. Auch lässt sich eine Nebenläufigkeit erkennen: Im oberen rechten Teil des Diagramms finden sich drei Elemente, die mit dem restlichen Graphen nicht verbunden sind. Diese werden jedoch trotzdem parallel zu diesem ausgeführt.

[2] http://www.agilent.com/find/vee

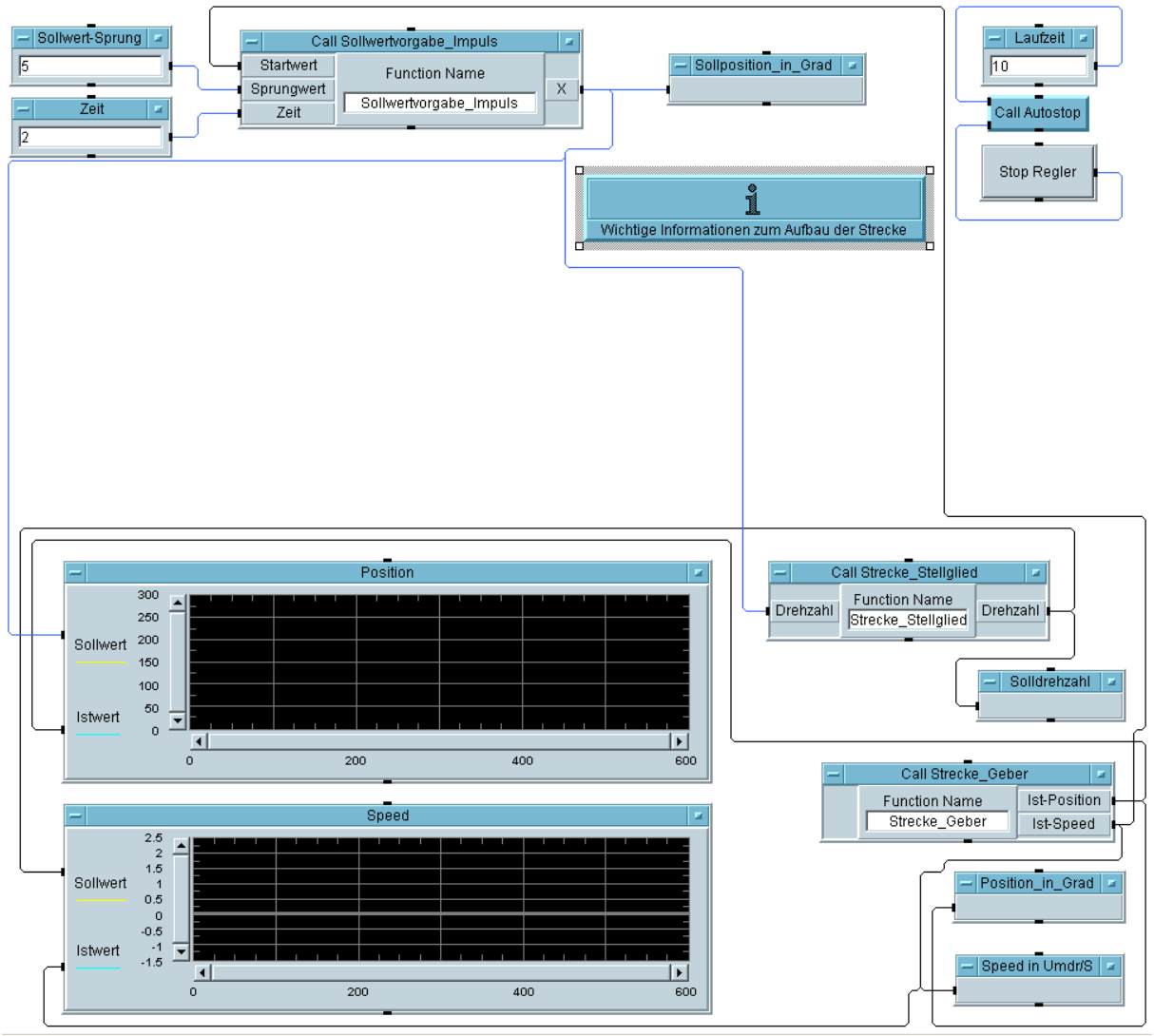


Abbildung 2.1: Ein kurzes Beispielprogramm in *Agilent VEE*

2.2.2 tresos GUIDE

Das Hauptanwendungsgebiet von *tresos GUIDE* liegt in der Spezifikation und Gestaltung von *HMI*-Systemen speziell im Automotive Bereich. Die Software wurde von der Elektrobit Group Plc. im Bereich Automotive (vormals 3soft GmbH) entwickelt und befindet sich in einem ständigen Verbesserungsprozess. Das Basiskonzept hierbei bietet ein *Stateflow*-Editor, der den Ablauf des Programms darstellt. Auch eine Benutzeransicht ist wie bei den anderen Systemen ist hier vorhanden. Zusätzlich existiert noch ein Simulator um das erstellte Programm testen zu können. Es existieren eine Reihe von Zusatzmodulen wie beispielsweise *tresos GUIDE+Speech*, welches dazu dient, die später erzeugte Software per Sprachbefehl steuern zu können.

Im Gegensatz zu den vorher beschriebenen Produkten erzeugt *tresos GUIDE* ein Modell, das dann im Anschluss mit Hilfe eines Codegenerators in eine von mehreren möglichen Zielsprachen übersetzt wird. Somit entfällt die Bereitstellung einer eigenen Laufzeitumgebung wie beispielsweise bei *LabView* und die erstellte Software ist auf die Zielplattform zugeschnitten.

Da *tresos GUIDE* eine stark domänenspezifische Anwendung (Spezifikation von *HMI* im Automotive Bereich) ist finden sich leider relativ wenig Informationen zu diesem System außer direkt von Elektrobit.

2.3 Abgrenzung zu vorhergehenden Produkten

Im Gegensatz zu den oben vorgestellten Systemen verzichtet der visuelle *Lauts*-Skripteditor auf eine Modellierung gleichzeitig ablaufender Prozesse. Dies hängt damit zusammen, dass nur prozedurale Skripte erstellt werden, die in eine Skriptsprache übersetzbar sein müssen. Somit ist der visuelle Skripteditor wirklich ein Codegenerator für Skriptsprachen. Die vorgestellten kommerziellen Produkte hingegen erzeugen direkt Bytecode, der entweder in einer eigenen Laufzeitumgebung ausgeführt wird oder ein natives Pogramm der Zielplattform darstellt. Dies resultiert in deutlichen Laufzeitvorteilen gegenüber dem *Lauts*-Skripteditor, was jedoch kein starkes Manko darstellt, da dieser nicht zeitkritisch einge-

setzt wird sondern nur eine Vor- und Nachbehandlung für die eigentliche Testausführung durchführt.

Der *Lauts*-Skripteditor setzt prinzipiell auf ein mehrstufig verteiltes System. Das Skript wird am Client erstellt und übersetzt, in der Datenbank hinterlegt und erst bei der Ausführung durch die TestEngine interpretiert.

2.3.1 Keine Erstellung von graphischen Benutzeroberflächen

Der *Lauts*-Skripteditor legt bei der Erstellung der Skripte eindeutigen Wert auf die Logik des Skriptes. Dieses wird auf der TestEngine ausgeführt und benötigt kein weiteres Eingreifen des Benutzers. Somit ist es nicht notwendig, eine Fassade für das laufende Skript zu erstellen. Notwendige Ausgaben (beispielsweise zur Fehlersuche) werden an den Client gesendet, der dann die Textausgabe vornimmt.

2.3.2 Unterstützung unterschiedlicher Diagrammarten

Zur Bearbeitung der Skripte unterstützt der Skripteditor verschiedene Ansichten. Diese ermöglichen es dem Benutzer, bestimmte Aspekte des Skriptablaufs (Datenfluss $\leftrightarrow$ Ablaufstruktur) deutlicher darzustellen. Da die meisten visuellen Programmiersysteme hingegen auch eine graphische Benuzeroberfläche zur Laufzeit des Programms anbieten besitzen diese zwar auch mehrere Ansichten, jedoch beschränken sich diese auf eine Darstellung des Programms an sich (meist als Flussdiagramm) und eine Darstellung der erzeugten Benutzeroberfläche. Diese Benutzeroberflächendarstellung beschreibt ausschließlich das Aussehen des laufenden Programms, nicht jedoch die Logik dahinter. Es kann somit als Vorteil gesehen werden, mehrere verschiedene Ansichten auf den eigentlichen Programmablauf zu haben wie dies im *Lauts*-Skripteditor der Fall ist.

2.3.3 Unterstützung verschiedener Zielsprachen

Die meisten verbreiteten Systeme zur visuellen Programmierung bringen eine Laufzeitumgebung mit sich, in der dann das erstellte Programm ausgeführt wird. Dies hat auf der einen Seite den Vorteil einer kontrollierten Umgebung für das laufende Programm und bringt Laufzeitvorteile, da direkt Bytecode interpretiert wird. Auf der anderen Seite jedoch macht es die Ausführung unbedingt von dem Vorhandensein einer solchen Laufzeitumgebung abhängig. Dieser Faktor stellt damit auch das einzige Kriterium für die Portabilität des erstellten Programms dar. Die Migration auf eine neuere Version des Programmiersystems könnte hier eine kritische Stellung einnehmen, da unklar ist inwiefern der ältere Bytecode eines vorhandenen Programms von einer neueren Laufzeitumgebung interpretiert werden kann.

Der *Lauts*-Skripteditor hingegen erzeugt Quelltext einer gewählten Zielsprache. Diese ist austauschbar und die Auswahl verfügbarer Sprachen kann erweitert werden. Die Erzeugung von Quelltext hat den Vorteil, dass auch nachträglich noch manuelle Änderungen im auszuführenden Programm möglich sind, unabhängig von den Fähigkeiten des visuellen Skripteditors. Da die Zielsprache prinzipiell wählbar ist, ist es möglich mit einer Skriptsprache zu arbeiten, die dem jeweiligen Ersteller des Skriptes am besten liegt. Davon, und weil der erzeugte Quelltext noch manuell erweitert und verändert werden kann, profitiert dann im Endeffekt die Qualität des erstellten Skriptes.

2.3.4 Straffere Richtlinien für Darstellung und Verbindungen

Durch die Einführung spezieller Richtlinien für die Positionierung einzelner Elemente im Flussdiagramm wird eine verbesserte Übersichtlichkeit erreicht. Abbildung 2.1 zeigt, dass selbst kleinere Diagramme schnell undurchsichtig werden können, wenn es möglich ist, die Elemente komplett frei zu positionieren. So ist beispielsweise nicht sofort ersichtlich, wo der Endpunkt für die Verarbeitung der Daten (im Beispiel die beiden Diagramme) erreicht ist. Auch wird dadurch das Auftreten von sich überschneidenden Kanten, was auf den Anwender verwirrend wirkt weitgehend vermieden.

2.3.5 Fazit

Zusammenfassend lässt sich sagen, dass der *Lauts*-Skripteditor eine Synthese verschiedener unabhängig voneinander existierenden Konzepten darstellt. Dies umfasst sowohl Datenflusskonzepte zur Modellierung der Skripte, wie sie ähnlich in anderen visuellen Programmiersystemen wie *LabView* vorkommen, als auch Konzepte der Metaprogrammierung. Die Verwendung von Metasprachenkonzepten erhöht die Flexibilität und Erweiterbarkeit des Skripteditors, und Techniken aus dem Bereich der Codegenerierung erlauben es, stabilen Quelltext in der entsprechenden Zielsprache zu erzeugen.

Einen besonders auf datenflussorientierte Programmierung gerichteten Einblick in den aktuellen Stand der Forschung bietet (WMJM04).

3 Prototyping

3.1 Allgemeine Erkenntnisse

Die Implementierung des visuellen Skripteditors erfolgte prinzipiell in einer Prototyping-Phase. Die hierbei gewonnenen Erkenntnisse sollen einem strukturierten Engineering des Software-Moduls zugute kommen. An dieser Stelle sollen deshalb besonders kritische Aspekte, die bei der prototypischen Entwicklung zu Tage getreten sind, erläutert werden.

3.1.1 Besondere Betrachtung von Variablen

Besonders in einer Nassi-Shneiderman-ähnlichen Darstellungsform nehmen Variablen eine zentrale Stellung ein. Sie stellen hier die einzige Möglichkeit dar, Datenfluss zu modellieren. Bei einer PAP-ähnlichen Darstellung hingegen können sie dazu dienen, die Übersichtlichkeit durch Entkopplung einzelner Elemente entscheidend zu verbessern.

In Zusammenhang mit Variablen müssen verschiedene Aktionen unterschieden werden:

1. **Deklaration einer Variable:** Skriptübergreifende Variablen müssen außerhalb des Skriptes in der Teststruktur deklariert werden. Innerhalb eines Skriptes deklarierte Variablen hingegen besitzen nur innerhalb des Skriptes (genauer: innerhalb des Anweisungsblocks) Gültigkeit. An dieser Stelle wird der Variablen auch ein Variablentyp zugeordnet. Dieser kann entsprechend Abschnitt 1.3.1 definiert sein.

2. **Setzen des Wertes einer Variable:** Beim Setzen eines Variablenwertes muss auf die Korrektheit des Variablentyps geachtet werden. So darf es nur möglich sein, einer Variablen eines bestimmten Typs einen Wert zuzuweisen, der dem Variablentyp entspricht. Konvertierungen werden nicht (oder nur durch entsprechende Skriptbausteine) durchgeführt.

3. **Auslesen des Wertes einer Variable:** Das Auslesen einer Variable erfolgt relativ einfach, es ist jedoch eine Unterscheidung zu treffen, ob es sich um eine skriptinterne Variable handelt, oder ob Variablen der TestEngine / der Teststruktur gelesen werden.

Zur Einbettung des Skripteditors in das Gesamtsystem *Lauts* ist es weiterhin notwendig, Systemvariablen im Skript benutzen zu können, da auf diese Art eine Kommunikation unterschiedlicher, über den Testbaum verteilter Skripte ermöglicht wird. Auch der Typ der Variablen ist hierbei von Bedeutung, da nicht jedes Element jeden beliebigen Typ von Variable verwenden kann. Selektionen und Schleifen zum Beispiel können nur mit boolschen Eingangswerten umgehen.

3.1.2 Besondere Betrachtung der Datenobjekte

Alle Datenobjekte, die in einem Skript verwendet werden können entsprechen einem der folgenden Typen, wie sie schon in den Abschnitten 1.3.1 und 1.3.2 angesprochen wurden:

Boolean: Dieser Datentyp dient unter anderem zur Auswertung von Verzweigungen oder auch Schleifen. Ein Objekt diesen Typs kann zum Beispiel durch die Auswertung eines durchgeführten Tests am SUT gewonnen werden.

Long: Ganzzahlige Datenobjekte dienen unter anderem als Zählervariable für Schleifen. Zusätzlich können sie als Eingabeparameter für Funktionsbausteine wie Zufallsgeneratoren oder *Funktionen* des Lauts-Systems verwendet werden.

Double: Fließkommawerte dienen Hauptsächlich als Ein- und Rückgabeparameter einzelner Funktionsbausteine, können jedoch genau wie ganzzahlige Werte zur dynamischen Verzweigung innerhalb des Skriptes anhand von Vergleichsoperationen dienen.

String: Zeichenketten werden im fertigen System hauptsächlich zur (formatierten) Ausgabe von Meldungen und Protokollierungseinträgen verwendet. Auch sie können jedoch verglichen werden um z.B. eventuell aufgetretene Fehlermeldungen zu analysieren. Dieses Vorgehen ist allerdings nicht unproblematisch, da die Form einer solchen Meldung eventuell nicht verlässlich gleich bleibt. Dies ist beispielsweise bei der Lokalisierung der Meldungen der Fall.

Result: Dieser Datentyp beschreibt ein konkretes Testergebnis. Es wird durch die Ausführung eines *Lauts*-Testelementes erzeugt und kann im weiteren Verlauf weiter ausgewertet werden.

Object: Dieser Datentyp umfasst die komplexen, allgemeinen Datentypen, die in der *Lauts*–Software vorkommen können. Bei ihrer Verwendung ist Vorsicht walten zu lassen, da sie jeden beliebigen anderen Objekttyp umfassen. Somit ist immer darauf zu achten, dass der verwendete *wirkliche* Datentyp feststeht um anschließend mit ihm arbeiten zu können (durch entsprechende *Casts*). Somit ist anzuraten, solche Datenobjekte nur in speziellen, *Lauts*-spezifischen Skriptelementen gekapselt zu verwenden.

Variablen sind in einer Klasse gekapselt, die Informationen über das enthaltene Datenobjekt enthält. Hierzu gehört der Typ des Objekts, welcher unter anderem dazu dient, unmögliche Verbindungen zwischen Skriptelementen auszuschließen. Diese Standardisierung dient dazu, eine passende Schnittstelle der Datenobjekte zu anderen Elementen bereitzustellen.

Zusätzlich müssen Datenobjekte zu denen, die im regulären Skripteditor und im restlichen *Lauts*-System verwendet werden, kompatibel sein. Weitere Eigenschaften sind in Abschnitt 3.2.3 der Diplomarbeit von Sebastian Kliesch (Kli07) erläutert.

Die Darstellung im Blockdiagramm erfolgt als Subkomponente des Elements. Im Flussdiagramm hingegen werden Datenobjekten eigene Elemente zugeordnet. Bei beiden Diagrammarten werden sie farblich als Datenobjekte blau gekennzeichnet.

3.1.3 Einfügen von Elementen in das Diagramm

Das Einfügen neuer Elemente in das Diagramm (und damit auch in das semantische Modell des Skriptes) erfolgt mittels *Drag&Drop*. Die Elemente werden aus einer Leiste aufgenommen und in das Diagramm gezogen. Daraufhin wird ein neues Element im Diagram (an der entsprechenden Stelle) erzeugt. Dieser Ansatz ist für beide Darstellungsarten praktikabel und ein recht intuitiver Weg der Interaktion. Das Einfügen beschränkt sich hierbei nicht nur auf das Einfügen allgemeiner Konstrukte wie Verzweigungen, Schleifen usw. sondern erstreckt sich auch auf die Verwendung von Elementen der eigentlichen Teststruktur wie z.B. „globalen“ Variablen und Funktionsaufrufen die Aktionen am *SUT* auslösen.

Dies hat zur Folge, dass der Inhalt der Leiste verfügbarer Elemente nicht ausschließlich statisch ist sondern auch vom jeweiligen Knoten abhängt, der mit dem Skript assoziiert ist.

Auch dieser Punkt ist bereits in Sebastian Klieschs Diplomarbeit ((Kli07) - Abschnitt 3.2.1) diskutiert. Die Darstellung möglicher Unterknoten der Teststruktur erfolgt als Baumdiagramm (*JTree*) in einem abgetrennten Bereich auf der rechten Seite des Applikationsfensters.

Des weiteren muss es an dieser Stelle möglich sein, *Variablen* der Teststruktur in das Diagramm – und somit auch in das Skript – zu übernehmen. Ihre Darstellung kann ähnlich erfolgen, so dass beliebig viele Lese- oder Schreiboperationen einer Variablen in das Diagramm eingefügt werden können.

3.2 Darstellungsarten

Bei der Erstellung des Prototyps wurden verschiedene Darstellungsarten getestet. Keine der beiden ursprünglichen Darstellungsarten (Nassi-Shneiderman / Programmablaufplan) erwies sich jedoch als vollkommen benutzerfreundlich und übersichtlich. Daraus ergab sich die Notwendigkeit, beide Darstellungsarten um einzelne Funktionen zu erweitern.

Als eine gemeinsame Basis zur Bearbeitung von Elementen dienen Dialogfenster, die beim Doppelklicken auf Elementen geöffnet werden. Diese sind diagrammunabhängig und verändern nur das Element an sich. Im Anschluss daran wird die Anzeige des Elements aktualisiert.

3.2.1 Nassi-Shneiderman-ähnliche Darstellung – Blockdiagramm

Die Nassi-Shneiderman-ähnliche Darstellung erfolgt unter Verwendung von *Swing*-Komponenten. Dies hat den Vorteil, dass die Positionierung und Anordnung der einzelnen Elemente vom jeweiligen *LayoutManager* übernommen werden kann. Somit ist auch bei Veränderung des umgebenden Fensters eine Korrekte Anordnung der einzelnen Elemente gewährleistet.

Verschachtelungen (wie beispielsweise die inneren Elemente von Schleifen) werden auch visuell als solche dargestellt. Ein Schleifen-Element enthält die inneren Elemente im Diagramm als Subkomponenten.

Das Einfügen und Neuanordnen einzelner Elemente erfolgt mittels *Drag&Drop*. Dies ermöglicht einen relativ intuitiven Umgang mit den einzelnen Elementen. Hierfür wurden spezielle „Ankerbereiche“ innerhalb der Komponenten definiert, an denen jedes Element mit der Maus unabhängig von seinem weiteren Inhalt genommen und verschoben werden kann.

Wie schon erwähnt werden Datenquellen (z.B. Auswertungen von Variablen oder Bedingungen für Verzweigungen) als Subkomponenten der sie enthaltenden Elemente dargestellt. Zusätzlich werden sie der Erkenntlichkeit halber eingefärbt. Sie können somit

nicht in mehreren anderen Elementen gleichzeitig enthalten sein. Sollten z.B. zwei Verzweigungen auf die selbe Bedingung zurückgreifen müssen, so ist es notwendig, mehrmals diese Bedingung darzustellen oder ihr eine Variable zuzuweisen und diese dann mehrmals auszuwerten.

Um die Übersichtlichkeit der Darstellung weiter zu erhöhen werden Umrandungen für die einzelnen Elemente verwendet. Hierzu gehört auch der Ankerbereich jedes einzelnen Elementes, der sowohl als Angriffsfläche zum Verschieben von Elementen als auch zum Öffnen des Bearbeitungsdialoges für das Element dient. Die *Borders* kennzeichnen somit klar die Grenzen der einzelnen Elemente, sowohl bei eingebetteten Elementen (Datenobjekte innerhalb der sie verwendenden Elemente) als auch bei Abfolgen von Elementen.

Um die Darstellung noch intuitiver zu gestalten besteht weiterhin die Möglichkeit, die verwendeten Komponenten um selbstgestaltete Komponenten zu erweitern. Dies erlaubt es, deutlich näher an die zugrundeliegende Darstellungsform eines Nassi-Shneiderman-Diagramms heranzureichen. Sie können dazu dienen, strukturelle Abläufe besser zu kennzeichnen.

Trotz all dieser Verbesserungen und Features ist das Blockdiagramm nicht unbedingt das Diagramm der Wahl zur Darstellung von Skriptabläufen. Beide Diagrammtypen haben ihre Vorteile bei der Darstellung von Skripten und sind für unterschiedliche Betrachtungsarten optimiert. Dennoch erscheint eine Darstellung als Flussdiagramm – wie sie im Folgenden näher betrachtet wird – einleuchtender, da sie eine bessere Perspektive auf die Veränderung und Auswertung der Datenobjekte liefert.

3.2.2 Programmablaufplan-ähnliche Darstellung – Flussdiagramm

Die beträchtlichste Erweiterung des Programmablaufplans besteht in der Einführung einer Datenflussmodellierung zusätzlich zu der ursprünglichen reinen Modellierung des Kontrollflusses. Hierbei werden auch die ursprünglichen Elemente des Ablaufplans um weitere Strukturen ergänzt:

Datenflussverbindungen: Die Anschaulichkeit des Diagramms wird durch die Einführung von Elementen, welche Datenquellen und -ziele repräsentieren verbessert. Diese Modellierung des Datenflusses findet durch neue Schnittstellen der einzelnen Elemente sowie durch eine Unterscheidung der Linientypen statt. Zusätzlich zu den ursprünglichen Linien, die den Kontrollfluss (Ablauf) des Programms charakterisieren wird ein neuer Typ eingeführt, welcher den Datenfluss (Verarbeitungsrichtung von Informationen) repräsentiert.

Variablen: Variablen dienen primär zur Speicherung und Abfrage bestimmter Ergebnisse einzelner Bausteine des Programms. Mit ihrer Hilfe ist es möglich, im späteren Verlauf eines Skripts noch einmal auf vorhergehende Ergebnisse zurückzugreifen um sowohl die Übersichtlichkeit des Diagramms zu erhöhen (durch den Wegfall störender Verbindungslinien) als auch an mehreren Stellen auf vorhergehende Ergebnisse zurückzugreifen.

Bedingungen: Bedingungen stellen eine Form von impliziten Variablen dar. Sie drücken einen binären Sachverhalt (wahr/falsch) aus und können somit von Schleifen oder auch Selektionen ausgewertet werden um den weiteren Kontrollfluss zu beeinflussen.

Aufruf- und Rückgabeparameter: Beim Aufruf einzelner Programmbausteine kann es notwendig sein, diesen Parameter mitzugeben (z.B. der Wertebereich bei einem Zufallszahlengenerator). Auch sie sind als Datenflusselemente anzusehen. Dies gilt auch für eventuell vom Unterprogramm erzeugte Rückgabewerte.

Erlaubte Verbindungen

Die Erstellung von Verbindungslinien erfolgt intuitiv anhand der zu verbindenden Elemente. Wird ein Element, welches nur den Kontrollfluss unterstützt (z.B. der Aufruf einer Methode ohne Rückgabe- und Aufrufparameter) mit einem anderen Element verbunden, so wird die einzig mögliche Verbindungsart verwendet. Um dem Editor einen weiteren Anhaltspunkt zu liefern wird jeweils immer die Quelle mit dem Ziel verbunden. Dies ermöglicht es, eine sinnvolle Prüfung darüber anzustellen, ob eine Verbindung erlaubt

ist oder im Fall von mehreren möglichen Verbindungen den Benutzer zu fragen, welche Art von Verbindung er wünscht. Unerlaubte Verbindungen sind:

Kontrollflussverbindungen:

Selbstverbindungen: Eine Verbindung eines Kontrollflusselements zu sich selbst würde semantisch einer sehr einfach konstruierten Endlosschleife gleichkommen. Dieses Verhalten kann vom Ersteller eines Skriptes nicht beabsichtigt sein und ist somit ungültig.

Mehrfachverbindungen: Es darf jedem Kontrollflusseingang nur genau ein Vorgänger zugeordnet werden. Andererseits ist es möglich, dass ein einzelnes Element zwei verschiedene Nachfolger hat. Dies ist zum Beispiel bei Schleifen (ein innerer, wiederholter Teil und ein äußerer, nachfolgender Teil) oder Verzweigungen sinnvoll. Verbindungen mit bereits belegten Ein-/Ausgängen werden jedoch abgelehnt. In diesem Fall besteht für den Benutzer die Möglichkeit, das Folgeelement an die Abarbeitungskette anzuhängen, direkt nach dem gewählten Vorgängerelement einzufügen oder die Aktion abzubrechen.

Endlosschleifen: Die Erstellung von Endlosschleifen im Editor soll möglichst unterbunden werden. Somit wäre es notwendig, neu erstellte Verbindungen auf eine zyklische Eigenschaft hin zu überprüfen. Da jedoch jedes Element zwangsweise nur ein Vorgängerelement haben kann ist es nicht möglich, einen neuen Ablaufstrang wieder zu einem Vorgängerelement zurückzuführen.

Datenflussverbindungen:

Selbstverbindungen: Die Verbindung eines Datenelements zu sich selbst ist wie auch beim Kontrollfluss nicht möglich. Dieses Vorgehen wäre sowohl semantisch nicht als sinnvoll einzustufen als auch bei der Codegenerierung sehr problematisch.

Mehrfachverbindungen: Datenflussobjekte besitzen jeweils nur maximal einen Datenflussausgang. Logische Operatoren (`UND` / `ODER` oder auch Vergleichsoperationen und

Funktionsaufrufe) besitzen im Gegensatz zu den meisten Elementen mehrere Datenflusseingänge. Mehrfachverbindungen sind bei Datenflussobjekten im Gegensatz zu Kontrollflussverbindungen in Ausnahmefälllen möglich.

Kommentare hingegen können mit jedem beliebigen Element verbunden werden. Die Zuordnungsrichtung ist hierbei unerheblich.

Einschränkungen

Die PAP-ähnliche Darstellung ist von Haus aus sehr frei, was die Positionierung der einzelnen Elemente betrifft. Dies kann jedoch die Übersichtlichkeit des erstellten Diagrammes stark gefährden, da so nicht unbedingt leicht erkennbar ist, welche Untergruppen z.B. zu einer Schleife gehören. Auch ist es sinnvoll, Konventionen für die Positionierung von Datenflusselementen einzuführen. Basiseinschränkungen sind durch die Flussrichtung gegeben:

- **Datenflussrichtung:** Datenquellen werden links von ihrem Ziel positioniert. Somit lässt sich der Datenfluss immer von links nach rechts lesen.

- **Kontrollflussichtung:** Da das erzeugte Skript immer sequentiell ausgeführt wird ist es möglich, diesen sequentiellen Ablauf als Flussrichtung von oben nach unten darzustellen. Dies entspricht der Lesart eines Nassi-Shneiderman-Diagramms und auch des erzeugten Quelltexts. Zu beachten ist hierbei die Behandlung von Elementen mit mehreren möglichen Kontrollflussausgängen. Bei Schleifen wird der innere Ausführungsblock nach rechts eingerückt, während der übergeordnete Ablaufpfad auf gleicher Ebene nach unten führt. Verzweigungen besitzen eine weitere Besonderheit, da sie zwei „Unterpfade“ für nachfolgende Elemente eröffnen. Diese Ausführungsstränge werden durch ein zusätzliches virtuelles Element, das auch zur Verzweigung gehört, abgeschlossen und wieder zusammengeführt.

Sonderfälle der Darstellung

- **Selektion:** Selektionen besitzen zwar prinzipiell nur zwei Ausgänge - einen für den *WAHR*- und einen für den *FALSCH*-Pfad, dennoch muss auch der Ablauf nach der Beendigung der Verzweigung darstellbar sein. Dies könnte über eine Zusammenführung der beiden erzeugten Ablaufpfade – ähnlich der Selektion im UML-Aktivitätsdiagramm – geschehen. Eine weitere Möglichkeit wäre die Einführung eines dritten Kontrollflussausgangs am Element. Beim Prototyp fiel die Entscheidung zugunsten der zweielementigen Darstellung der Verzweigung (ein Eingang, ein Ausgang, zwei Ablaufpfade dazwischen). Dieser Ansatz bringt weiterhin die Möglichkeit einer Positionierungseinschränkung der untergeordneten Elemente je nach Pfadzugehörigkeit. So ist es sinnvoll, Unterelemente nicht oberhalb des Einsprungs und nicht unterhalb des Aussprungs der Verzweigung positionieren zu lassen. Zusätzlich sollte ihre horizontale Positionierung eingeschränkt werden, um die Verlaufspfade klar zu kennzeichnen.

- **Schleifen:** Schleifen besitzen immer auch einen inneren Ablaufteil. Bei diesem muss klar erkenntlich sein, dass er zu der ihn umgebenden Schleife gehört, wie dies bei einer Nassi-Shneiderman-ähnlichen Darstellung selbstverständlich der Fall ist. Als Lösungsansatz empfiehlt es sich, die Positionierung untergeordneter Skriptelemente einzuschränken, so dass sie nur rechts und unterhalb der sie umgebenden Schleife positioniert werden können. Zusätzlich wäre es denkbar, der graphischen Darstellung der Schleife eine Umrandung der untergeordneten Elemente hinzuzufügen um die Zugehörigkeit noch weiter zu verdeutlichen.

3.3 Repräsentation der Semantik

Um das graphisch erstellte Skript in lauffähigen Quellcode umzuwandeln ist es notwendig, dieses in eine interne strukturierte Darstellung zu überführen. Diese dient neben der Repräsentation der einzelnen Elemente vorrangig zur Wiedergabe von Abhängigkeiten der einzelnen Elemente (Parameterübergabe, Datenfluss, Kontrollfluss). Zusätzlich exis-

tiert eine Zwischendarstellung in XML, die es beispielsweise ermöglichen würde, einen externen Übersetzer für den Quelltext zu verwenden.

3.3.1 Generelle Struktur

Die generelle Struktur von Skripten ist baumähnlich aufgebaut. Jedes einzelne Element in einem Skript besitzt Nachbarelemente und kann nicht losgelöst von diesen existieren. Es gibt grundsätzlich zwei Arten der Verwandtschaft zwischen Elementen: Datenfluss-orientiert und Kontrollfluss-orientiert. Die Abbildung 3.1 stellt ein generisches Element mit allen möglichen Ausgängen dar. Blaue Ankerpunkte sowie Pfeile geben die Datenflussrichtung an, rote Pfeile und Ankerpunkte hingegen den Kontrollflussverlauf.

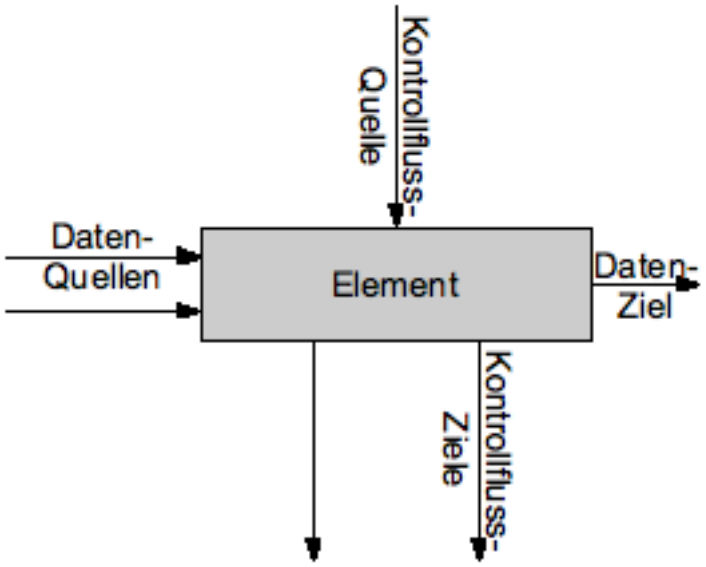


Abbildung 3.1: Ein einzelnes Element in der Modelldarstellung

Beachtenswert hierbei ist, dass die Elemente jeweils nur eine Nachfolgerichtung haben können – entweder es sind Kontrollflusselemente, die nur ausschließlich eine gewisse Funktionalität ausführen, oder sie sind Datenquellen, die ein bestimmtes Datenobjekt (Rückgabewert) erzeugen. Dies bedeutet, dass der Rückgabewert z.B. von Funktionen weiterverarbeitet werden *muss*, wenn einer existiert. Sollte die Funktion keinen Rückgabewert erzeugen (sie ist vom Typ `void`), so besitzt ihr Element lediglich einen Kontrollfluss- und

keinen Datenflussausgang. Da Elemente nur maximal einen solchen Datenflussausgang besitzen können erfolgt im Laufe des Skriptes immer eine Reduzierung des Datenvolumens, wie es auch normalerweise in Programmen der Fall ist.

Ein Skript darf nur maximal ein Anfangselement besitzen. Dies ist eine Gemeinsamkeit mit einer Baumstruktur. Jedoch tritt eine weitere Einschränkung noch zusätzlich für die Skripte in Kraft: Sie dürfen nur genau *ein* Endelement besitzen. Diese Einschränkung wurde getroffen, um den Skripten eine festere Struktur zu geben als dies normalerweise der Fall wäre.

Die Abbildung 3.2 stellt noch einmal mögliche Beziehungen zwischen den einzelnen Elementen im semantischen Modell des Skriptes dar.

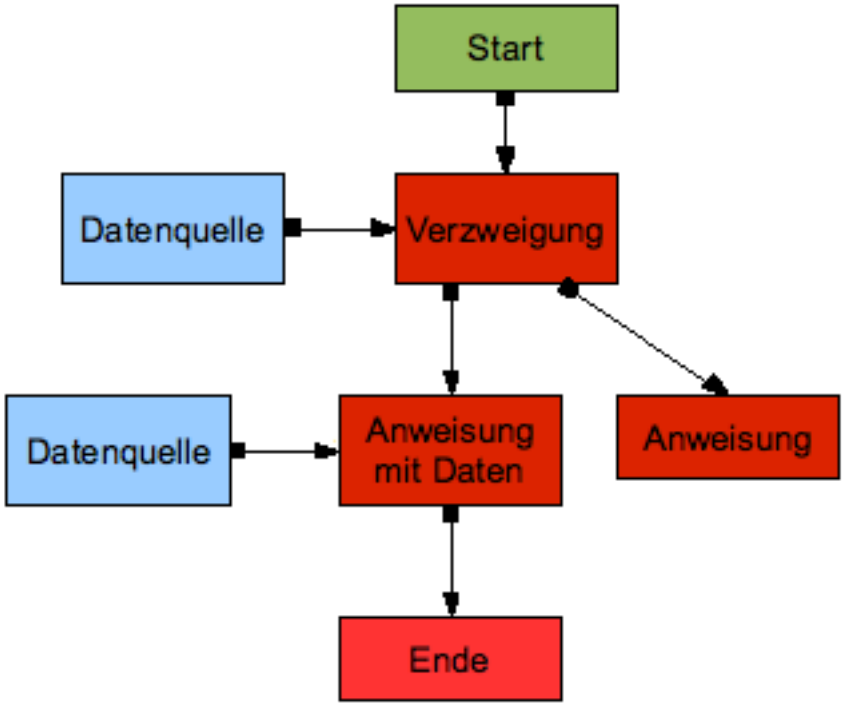


Abbildung 3.2: Ein beispielhaftes Netz aus Skriptelementen in der Modelldarstellung

3.3.2 Trennung von Semantik und Darstellungsform

Im Rahmen des Prototyping wurde klar, dass eine Trennung der Semantik des Skriptes von der Darstellung der einzelnen Elemente unabdingbar ist. Dies ist eine Vorrausset-

zung für die Verwendung unterschiedlicher Darstellungsarten und stellt die eigentliche Abstraktion des Skriptes dar. Weitere Informationen wie z.B. Positionsangaben beim Programmablaufplan-ähnlichen Diagramm werden bewusst von der Semantik getrennt. Dies hat zur Folge, dass das Diagramm in der Lage sein muss, eine automatische Anordnung der einzelnen Elemente vorzunehmen. Im Falle der Nassi-Shneiderman-ähnlichen Darstellung lässt sich das leicht verwirklichen, da sie nicht wirklich graphisch erfolgt. Bei einer Programmablaufplan-ähnlichen Darstellung hingegen ist es notwendig, graphentheoretische Grundlagen mit einfließen zu lassen. Tiefergehende Informationen zum Thema Graphenlayout finden sich in (Kön07).

Observer-Pattern

Zur Anpassung der Darstellung an die semantischen Gegebenheiten wird ein `Observer`-Pattern verwendet. Dies dient dazu, die Darstellungskomponente darüber zu informieren, wenn der Zustand eines Skriptelements verändert wird. Hierfür wird einem Skriptelement bei der Erzeugung einer graphischen Repräsentation desselben eine Referenz auf dieses Darstellungsobjekt mitgegeben. Wird dann das Skriptelement verändert (zum Beispiel mit anderen Elementen verbunden), so werden alle registrierten Darstellungen benachrichtigt. Dies führt zu einer Aktualisierung der Darstellung des Elements im Diagramm und verhindert Abweichungen zwischen der internen Darstellung und der Sicht des Benutzers auf das Skript. Zusätzlich ermöglicht dieses Vorgehen das gleichzeitige Darstellen des selben Elements in verschiedenen Diagrammen – und auch der Aktualisierung der Darstellungen beim Anwenden von Veränderungen in einem der beiden Diagramme. An dieser Stelle ist zwar noch unklar, ob es Sinn macht, dasselbe Element mehrfach darzustellen, doch die Möglichkeit sollte in Betracht gezogen werden.

Bei der Registrierung der beobachtenden Objekte unterscheiden sich momentan die Implementierungen der Diagramme. Beim Blockdiagramm wird nur jeweils die darstellende Komponente benachrichtigt, wohingegen beim Flussdiagramm das gesamte Diagramm über eine Veränderung des entsprechenden Elementes informiert wird.

3.3.3 XML-Darstellung

Die Darstellung in *XML* besitzt einige Vorteile gegenüber z.B. einer internen Bytecodedarstellung bei der Speicherung. So lässt sich mit Hilfe von *DTD*s oder *XML Schemata* eine flexible Definition erstellen, wie Skripte aufgebaut sein dürfen. Infolgedessen wird die Struktur des Skriptes automatisch überprüfbar. Weitergehende Prüfungen auf die Sinnhaftigkeit des Skriptes werden dadurch zwar nicht ersetzt, es ist jedoch eine grundlegende strukturelle Prüfung möglich.

Neben der Validierung der aufgebauten Skriptstruktur hat eine Darstellung in XML den Vorteil eines möglichst generischen Ansatzes. Sie ermöglicht es, externe Software-Tools zur Darstellung der erzeugten Struktur zu verwenden und im Fehlerfall die Fehlersuche zu vereinfachen.

Auch wäre es möglich, direkt aus dem erzeugten XML-Code einen ausführbaren Quelltext in einer Programmiersprache zu erzeugen ohne den Umweg über den Editor nehmen zu müssen. Diese Übersetzung könnte von einzelnen Modulen übernommen werden, welche austauschbar gestaltet sein können. Einen ähnlichen Ansatz verwendet auch das Meta-Language Projekt (*MetaL* - vgl. Abschnitt 1.3.3).

Da auch die Unterstützung für Skriptsprachen im *Lauts*-System bereits austauschbar gestaltet ist bietet es sich an, diese um die Funktionalität der Codegenerierung auf Basis der XML-Daten zu erweitern. Somit wäre es auch möglich, z.B. das Syntax-Highlighting des textbasierten Skripteditors zu erweitern oder auch diesen XML-Code (für bekannte Konstrukte) generieren zu lassen. Eine weitere Konsequenz wäre die Möglichkeit einer Konvertierung bestehender Skripte in die Metadarstellung in XML. Diese könnte dann wiederum dazu dienen, eine graphische Darstellung des Skriptes zu erreichen.

3.4 Code-Generierung

Die eigentliche Erzeugung des Quelltextes des Skripts stellt den wesentlichen und wichtigsten Punkt des Editors dar. Schließlich sollen die erstellten Abläufe auch direkt im

Lauts-System Verwendung finden und ausgeführt werden.

Die eigentliche Erzeugung des Quellcodes findet vom Benutzer verdeckt statt und muss somit verlässlich vonstatten gehen. Dies bedeutet für den User, dass das erstellte Skript während bzw. vor der Umwandlung geprüft werden muss, um ihm im Fehlerfall entsprechende Informationen zur Verbesserung des Skriptes zur Verfügung zu stellen. Sollte kein Fehler auftreten bleibt das gesamte Übersetzungsverhalten jedoch unsichtbar um den Benutzer nicht zu verunsichern.

Das gesamte Übersetzungssystem muss des weiteren modular aufgebaut sein, da prinzipiell verschiedene Skriptsprachen unterstützt werden sollen. Dies erhöht die Komplexität des Verfahrens enorm, da sowohl eine Zwischenschicht, die die Austauschbarkeit gewährleistet, mit eingebunden werden muss als auch in allen Sprachen entsprechende Konstrukte für alle möglichen Skriptelemente zur Verfügung stehen müssen. Dies ist ein weiterer Punkt der der Überprüfung vor der Erzeugung eines Skriptes bedarf.

3.4.1 Validierung des zu erzeugenden Skriptes

Vor dem Starten der eigentlichen Übersetzung des erstellten Skriptablaufs muss dieser auf verschiedene Kriterien hin überprüft werden. Dies dient zum einen dazu, grobe Fehler im Ablauf selbst – wie beispielsweise Endlosschleifen – festzustellen, zum anderen muss jedoch auch geprüft werden, ob die gewählte Zielsprache alle notwendigen Konstrukte des Ablaufs unterstützt. Im Idealfall sollte dies gewährleistet sein, jedoch ist es durch die Austauschbarkeit und Erweiterbarkeit der Skriptsprachen durchaus möglich, dass eine spätere Skriptsprache das entsprechende Konstrukt nicht „versteht“. Diese Prüfungen beschränken sich nicht allein auf den kontrollflussseitigen Ablauf, sondern erfolgen auch unter Berücksichtigung des Datenflusses. So darf es beispielsweise nicht möglich sein, dass ein erzeugtes Datenobjekt (wie es beispielsweise beim Ausführen einer Funktion entsteht) nicht als seine eigene Datenquelle – auch nicht über Zwischenschritte – dienen darf. Dies wäre zwar in einem Graphen – wie bei der Darstellung im Flussdiagramm – möglich, jedoch nicht im Blockdiagramm, wo es – dadurch, dass Datenquellen als in ihrem Ziel enthalten dargestellt werden – nicht darstellbar ist. Wichtigster Punkt bei der

Unterbindung solcher Konstrukte ist jedoch nicht die fehlende Darstellungsmöglichkeit, sondern die Unmöglichkeit, solche Konstrukte effektiv in Quellcode umzuwandeln. *Somit ist es unmöglich, in dem Skripteditor rekursive Programmabläufe ohne die Verwendung von Funktionsaufrufen zu erstellen.*

Diese Überprüfungen finden teilweise schon beim Erstellen des Skriptes – genauer gesagt beim Verbinden von Elementen – statt. Ist eine mögliche Verbindung als ungültig klassifiziert worden wird sie nicht hergestellt und der Benutzer nach Möglichkeit über den Grund hierfür informiert.

Ein weiteres Kriterium, das geprüft werden muss ist das Vorhandensein *genau eines* Endknotens des Skriptes. Dieser Endknoten muss in jedem Fall ein Ziel des Kontrollflusses sein. Besitzt der Endknoten eine zugeordnete Datenquelle, so entspricht diese dem Rückgabewert des Skriptes. Andernfalls besitzt das gestaltete Skript keinen Rückgabewert. Die Verwendung des Skriptes als Datenquelle – und somit als effektiv benutzbarem Skriptbaustein – bedingt eine genauere Untersuchung als dies im Prototyp geschehen ist.

Auch die weiteren im Skript vorkommenden übertragenen Datenobjekte müssen vor der eigentlichen Übersetzung überprüft werden. Es darf beispielsweise nicht sein, dass eine Verzweigung, die einen bool'schen Eingangswert erwartet, einfach direkt einen anderen Wert erhält. Dies wäre im erstellten Skript nicht möglich und darf auch im dazugehörigen semantischen Modell nicht gültig sein. Genauso ist es notwendig, zu überprüfen ob die verwendeten Datentypen überhaupt vom Skriptinterpreter verwendet werden können. Dies bedingt bei der Codegenerierung auch die automatische Generierung von `import`-Anweisungen im Quelltext.

3.4.2 Schwierigkeiten bei der Unterstützung mehrerer Sprachen

Aufgrund der Notwendigkeit für den Skripteditor, mehrere verschiedene Zielsprachen unterstützen zu müssen ergeben sich eine Reihe von Hindernissen, die überkommen werden müssen. Hierzu gehören:

- **Unterschiedliche Syntaxen:** Unterschiedliche Skriptsprachen benutzen natürlicherweise auch unterschiedliche Syntaxen. Diese unterschiedlichen Formulierungen für die selben semantischen Gegebenheiten müssen selbstverständlich vom Übersetzungsmodul des Skriptgenerators unterstützt werden. Somit ist es unabdingbar, jedes mögliche Element in jeder möglichen Skriptsprache abzubilden.

- **Leichte Erweiterbarkeit:** Die tatsächlich in der *Lauts*-Software eingesetzte Skriptsprache ist nicht fest definiert. Die Erweiterbarkeit der Skriptsprachen war von Beginn an ein fundamentaler Punkt bei der Integration des Scriptingsystems in *Lauts*. Mit der Einführung des visuellen Skripteditors erstreckt sich diese Anforderung nun auch auf dessen Übersetzungsmodul. Somit ist es notwendig, eine klare Schnittstelle für das Hinzufügen neuer Skriptsprachen zur Verfügung zu stellen.

- **Umgang mit LAUTS-spezifischen Elementen:** Es ist sehr wahrscheinlich, dass nicht nur die Syntax der einzelnen allgemeinen Skriptelemente (Schleifen, Verzweigungen etc.) von Sprache zu Sprache unterschiedlich ist, sondern auch die Ansteuerung von *Lauts*-internen Konstrukten wie beispielsweise dem Ausführen eines Testfalles. Als weitere Schwierigkeit kommt hierzu die Unterstützung der internen Datentypen (wie eines *Testresultats*) durch die Skriptsprache. Hierzu kann es notwendig sein, entsprechende Klassen des Systems in das Skript zu importieren um diese Datenobjekte dann sinnvoll ansteuern zu können.

- **Kopf und Fuß der Skripte:** Die somit notwendigen `import`-Anweisungen verdeutlichen ein weiteres Problem – die Generierung der für das Skript notwendigen Kopf- und Fußzeilen. Diese können nicht statisch immer für die selbe Skriptsprache geich sein, da es auszuwerten gilt, welche internen Datentypen verwendet werden. Zusätzlich besteht für Skripte – und Skriptbausteine – die Möglichkeit eines Rückgabewertes. Dieser muss dann selbstverständlich auch beim fertig generierten Skript mir berücksichtigt werden. Die Ausgabe muss so erfolgen dass dieser Parameter von anderen Skripten reibungslos benutzt werden kann.

- **Generische Elemente:** Unter die Rubrik „Generische Elemente“ fallen unter anderem solche Elemente wie Zufallsgeneratoren und ähnliches. Ihre Funktionsweise ist zwar nicht direkt von den Gegebenheiten innerhalb des *Lauts*-Systems abhängig, sie

können jedoch eine drastisch unterschiedliche Übersetzung erfordern. Im Extremfall kann dies so ablaufen, dass ein Hilfsobjekt bzw. eine Hilfsklasse erst in der Schicht der Skriptsprache selbst manuell implementiert werden muss. Somit stellen solche generischen Elemente eine besondere Herausforderung für die Anpassung der untersten Übersetzungsebene dar. Zusätzlich kann es vonnöten sein, diese dynamische Schicht auf ihre Fähigkeit, diese Elemente zu unterstützen, hin abzufragen. Abhilfe für diese Problematik würde die Verwendung von einzelnen textuellen Skriptbausteinen schaffen.

3.4.3 Direkte Umsetzung der Semantik

Der naheliegendste Ansatz für die Umsetzung des semantischen Modells in ausführbaren Quellcode liegt in der direkten Umwandlung der einzelnen Elemente in Codefragmente. Hierzu wird einer Übersetzungsklasse das Modell übergeben um im Anschluss daran – sofern keine Fehler auftreten – den fertigen Quellcode zu erhalten.

Vorteile hiervon sind:

1. **Geschwindigkeit:** Die direkte Umsetzung sorgt für gewisse Geschwindigkeitsvorteile, da das gesamte Modell bereits im Hauptspeicher vorliegt. Es ist somit nicht nötig, vor der Übersetzung des Modelles in Quelltext irgend etwas einzuparsen.

2. **Schnelle Implementierung:** Für das Prototyping erlaubt dieser Ansatz eine schnelle Implementierung. Es wird einfach jedem einzelnen Element eine Methode zugeordnet, die dessen Quelltext – unter Berücksichtigung benachbarter Elemente – erzeugt. Dies ermöglicht es recht schnell zu einem ersten Ergebnis zu kommen.

Dem stehen jedoch auch Nachteile gegenüber:

1. **Unterstützung mehrerer Sprachen:** Da der Editor Quelltext in mehreren verschiedenen Sprachen erzeugen können soll ist dieser Ansatz kritisch. Er würde es erfordern, wieder ein Plugin-System einzuführen, das die Austauschbarkeit der Skript-

sprachen gewährleistet. Diese Schnittstelle wäre jedoch relativ komplex, da die Unterstützung jedes einzelnen Elementes des Skriptes abgeprüft werden müsste.

2. **Unübersichtlichkeit:** Die Verwendung von Codegeneratoren erfolgt zumeist unter Zuhilfename von Vorlagen bzw. *templates*. Bei diesem Ansatz fehlen direkt verwendete und leicht änderbare Vorlagen, da der zu erzeugende Quelltext fest in die zur Übersetzung verwendeten Klassen eingebunden ist. Dies macht eine Erweiterung oder Veränderung der Übersetzung schwierig bis unmöglich.

In der Konsequenz lässt sich klar sagen, dass die festgestellten Nachteile stark überwiegen. Vor allem die mangelnde Kontrolle im Nachhinein über den erzeugten Quellcode und die Komplexität der Schnittstelle für das Plugin-System machen ein solches Vorgehen unpraktikabel.

3.4.4 XML als Zwischenschritt

Eine Möglichkeit, die Probleme des vorhergehenden Ansatzes zu mindern ist der Einsatz einer weiteren Zwischenschicht. Konkret bedeutet dies den Einsatz von *XML* als Zwischenschritt vor der eigentlichen Übersetzung in die Zielsprache. Diese Repräsentation der Semantik in einer Metasprache ist für die weitere Übersetzung durchaus sinnvoll, da ja mehrere Zielsprachen unterstützt werden sollen. Der Abschnitt 1.3.3 bietet eine weiterführende Diskussion zu diesem Thema.

Vorteile bei der Verwendung vom *XML* sind:

1. **Weitere Möglichkeit zur Validierung:** Die Einführung eines weiteren Zwischenschrittes bietet die Möglichkeit einer weiteren Überprüfung des erstellten Skriptes auf semantischer Ebene. Da *XML* ein weit verbreiteter Standard ist bietet dies sowohl die Möglichkeit Drittsoftware zur Analyse und Validierung des Dokuments zu nutzen als auch Standardfunktionalitäten der Sprache – wie beispielsweise eine *DTD*.

2. **Menschenlesbarkeit:** *XML* hat den weiteren Vorteil, dass es sowohl von Maschinen als auch von Menschen aufgrund seiner strukturierten Syntax und der Benutzung von *Unicode*-Text zur Darstellung anstelle von Binärdaten gut lesbar ist. Dies ermöglicht es, auch manuell die Struktur des erzeugten Dokuments zu analysieren.

3. **Passende Darstellungsart:** *XML* ähnelt durch seinen inhärenten Aufbau sehr stark dem semantischen Modell, das hinter den Skripten steht. Sie besitzt prinzipiell eine streng hierarchische Struktur, wie dies auch bei Elementen der Fall ist, die andere Elemente enthalten – als Beispiel seien hier Schleifen genannt, die einen inneren Block besitzen. Des weiteren besitzt es eine prinzipiell sequentielle Struktur für „Nachbarknoten“, was einem sequentiellen Ablauf im Skript entspricht.

4. **Generischer Ansatz:** Die Verwendung von *XML* hat zur Folge, dass die daraus erzeugten Dokumente gut und einfach weiterverarbeitet werden können. Zusätzlich hat sie den Vorteil, dass nicht direkt der Quelltext für die nachfolgende Übersetzung in die Zielsprache modifiziert werden muss. Das erzeugte Zwischendokument ist für alle Zielsprachen gleich, die eigentliche Übersetzung erfolgt dann in einem nachfolgenden Schritt. Dies erleichtert die Konstruktion verschiedener Übersetzungsmodule für verschiedene Zielsprachen.

Demgegenüber stehen gewisse Nachteile bei der Verwendung dieses weiteren Zwischenschrittes:

1. **Höhere Komplexität:** Die Einführung einer weiteren Zwischenschicht in das Procedere beim Erzeugen des Quellcodes bringt eine weitere Erhöhung der Komplexität der Software mit sich. An sich ist dies nicht unbedingt ein negativer Aspekt, jedoch erhöht dies auch die Anzahl möglicher Fehlerquellen im Gesamtsystem.

2. **Zeit:** Zusätzlich benötigt dieser zusätzliche Aufwand eine gewisse Zeitspanne, die die Dauer des gesamten Übersetzungsvorganges erhöht. Da die Übersetzung jedoch nicht ständig sondern nur zu bestimmten Zeiten manuell vorgenommen wird ist dieser Nachteil im Vergleich zu dem erbrachten Nutzen zu vernachlässigen.

3. **Bibliotheken:** Der erzeugte *XML*-Code muss im Anschluss an seine Erzeugung wieder durch das nachfolgende Modul im Übersetzungsprozess eingeparst werden. Hierzu kann es nötig sein, weitere umfangreichere Biblioheken – wie beispielsweise einen anderen XML-Parser – einzubinden. Auch dieser Punkt erhöht die Komplexität des gesamten Übersetzungsvorgangs.

Die Folge dieser Abwägung ist, dass trotz der bestehenden Nachteile – besonders in Hinblick auf die Performance des Übersetzungsvorgangs – die Verwendung von XML als Zwischenschicht zur Repräsentation der Semantik durchaus Sinn macht. Die hieraus gewonnene bessere Validierung des Skriptes und die immense Steigerung der Portabilität und Flexibilität rechtfertigen den Einsatz.

3.4.5 XSLT als Übersetzungsmethode

XSLT ist eine Transformationssprache zur Umwandlung von generischen *XML*-Datenstrukturen in beliebige andere Dokumentvarianten. Obwohl die Zielsprache meist auch *XML*-basiert ist sind viele andere Ausgabearten möglich. Unter anderem können hiermit auch generische Textdokumente erstellt werden. Dies ist genau der Aspekt von *XSLT* welcher zur Code-Generierung genutzt werden kann. Die generelle Vorgehensweise hierbei ist das manuelle Erstellen einer XSLT-Vorlage, welche alle Elemente des möglichen Skriptes abdeckt. Diese wird dann zusammen mit der XML-Darstellung des Skriptes einem XSLT-Prozessor (beispielsweise *Xalan*[1]) übergeben. Dieser wendet dann die Regeln aus der Vorlage, die angeben, in welcher Form die Umwandlung erfolgen soll, auf die XML-Struktur an. Durch die Verwendung von unterschiedlichen Vorlagen für unterschiedliche Zielsprachen, die auf denselben XML-Eingangsdaten beruhen lässt sich eine starke Flexibilität hinsichtlich der Zielsprache erreichen. Es wird unnötig, den Quelltext des Codegenerators selbst zu verändern.

Die Vorteile bei der Verwendung von *XSLT* seien hier noch einmal zusammengefasst:

[1] http://xml.apache.org/xalan-j/

1. **Verwendung von Vorlagen:** Augenscheinlichster und wohl auch wichtigster Vorteil bei der Verwendung von XSLT ist die praktische Verwendung von Vorlagen. Vorlagen sind insofern sehr sinnvoll als sie einen guten Überblick über das Format des Zieldokumentes erlauben. Beim Erstellen dieser *Templates* kann sehr strukturiert vorgegangen werden. Das starre Konstrukt, das ein Element in der Zielsprache ausmacht wird einfach eingegeben, im Anschluss daran werden die einzelnen dynamischen Teile des Quelltexts durch Meta-Statements ausgedrückt, welche dann im „Ernstfall“ durch den XSLT-Prozessor mit den eigentlichen Daten bzw. Parametern des eingefügten Elements aufgefüllt werden.

2. **Flexibel:** Dieser generische Ansatz zur Transformation von XML-Daten in beliebige andere Darstellungsformen den XSLT bietet ließe sich noch auf viele andere Arten nutzen. So wäre es beispielsweise denkbar, nebenher noch aus den Kommentaren, die den einzelnen Elementen zugeordnet sind, eine kleine technische Dokumentation zum Skript automatisch erstellen zu lassen.

3. **Leicht Erweiterbar:** Die Tatsache, dass in XSLT beliebig viele Vorlagen zu den selben Knoten der XML-Datenstruktur zugeordnet werden können findet die praktischste Anwendung in der Anforderung, mehrere verschiedene Zielsprachen unterstützen zu müssen. Käme eine neue Zielsprache hinzu wäre es leicht möglich, für diese eine neue Vorlage zu erstellen. Diese müsste dann einfach dem in die Software integrierten XSLT-Prozessor als Vorlage für – beispielsweise schon bestehende in XML ausgedrückte Skripte – dienen um eine neue Ausgabe in einer neuen Skriptsprache zu ermöglichen. Es müsste keine Zeile *Java*-Code der Software verändert werden.

4. **Ersetzung der Schnittstelle Übersetzer ⇔ Zielsprache:** Das vorher geschilderte *Plugin-System* zur Unterstützung verschiedener Zielsprachen kann durch die Verwendung mehrerer XSLT-Templates vollständig ersetzt werden. Dies ist ein großer Vorteil, da es eine Erweiterung des Skriptgenerators ohne die weitere Erstellung von *Java*-Klassen erlaubt. Auch die relativ komplexe Schnittstelle zur Erzeugung einzelner Elemente in der Zielsprache wird dadurch hinfällig.

In der Konsequenz bedeutet der Einsatz von *XSLT* eine starke Erhöhung der Flexibi-

lität, weiterhin zu Lasten der für die Übersetzung benötigten Zeit. Da diese jedoch – wie bereits erwähnt – keine kritische Rolle spielt ist der Einsatz von XSLT für die Übersetzung durchaus angebracht. Sie sollte zumindest als Ergänzung zu der vorher geschilderten Übersetzungsmethoden mittels XML in Betracht gezogen werden, da dieses Vorgehen eine vollständige Entkopplung der Zielsprache von dem übersetzenden System – dem Skriptgenerator – erlaubt. Weitere Informationen zu *XSLT* finden sich in (Man05) und (Bur01).

4 Spezifikation

4.1 Allgemeiner Ablauf und Integration in Lauts

In diesem Abschnitt soll der allgemeine Ablauf bei der Verwendung des visuellen Skripteditors aus Benutzersicht geschildert werden. Insbesondere die Handhabung des Editors und seine Stellung innerhalb des *Lauts*-Systems wird hier näher erläutert.

4.1.1 Handhabung des Editors

Die allgemeinen Kontrollelemente für bestimmte Aktionen des Benutzers orientieren sich stark an denen des textuellen Skripteditors. Dies dient der Schaffung eines einheitlichen Bedienumfeldes für den Benutzer. So bestehen Möglichkeiten zum Ausschneiden, Kopieren, Löschen einzelner Elemente. Diese können sowohl über das Menü *Bearbeiten* erreicht werden als auch durch einen direkten Klick auf die entsprechende Schaltfläche in der Werkzeugleiste, welche unterhalb des Menüs angeordnet ist.

Der allgemeine Aufbau der Benutzeroberfläche ist in Abbildung 4.1 dargestellt.

Im oberen Bereich des Editors befindet sich die Menüleiste. Diese beinhaltet sowohl Befehle zum Bearbeiten einzelner Elemente als auch übergeordnete Funktionalitäten. Hierunter fallen beispielsweise Aktionen zum Laden und Speichern des erstellten Skriptes (siehe Abschnitt 4.6) oder auch zum Umschalten der Diagrammansicht. Auch diagrammspezifische Aktionen – so zum Beispiel die automatische Anordnung der Elemente im Flussdiagramm – werden in das Menü eingetragen.

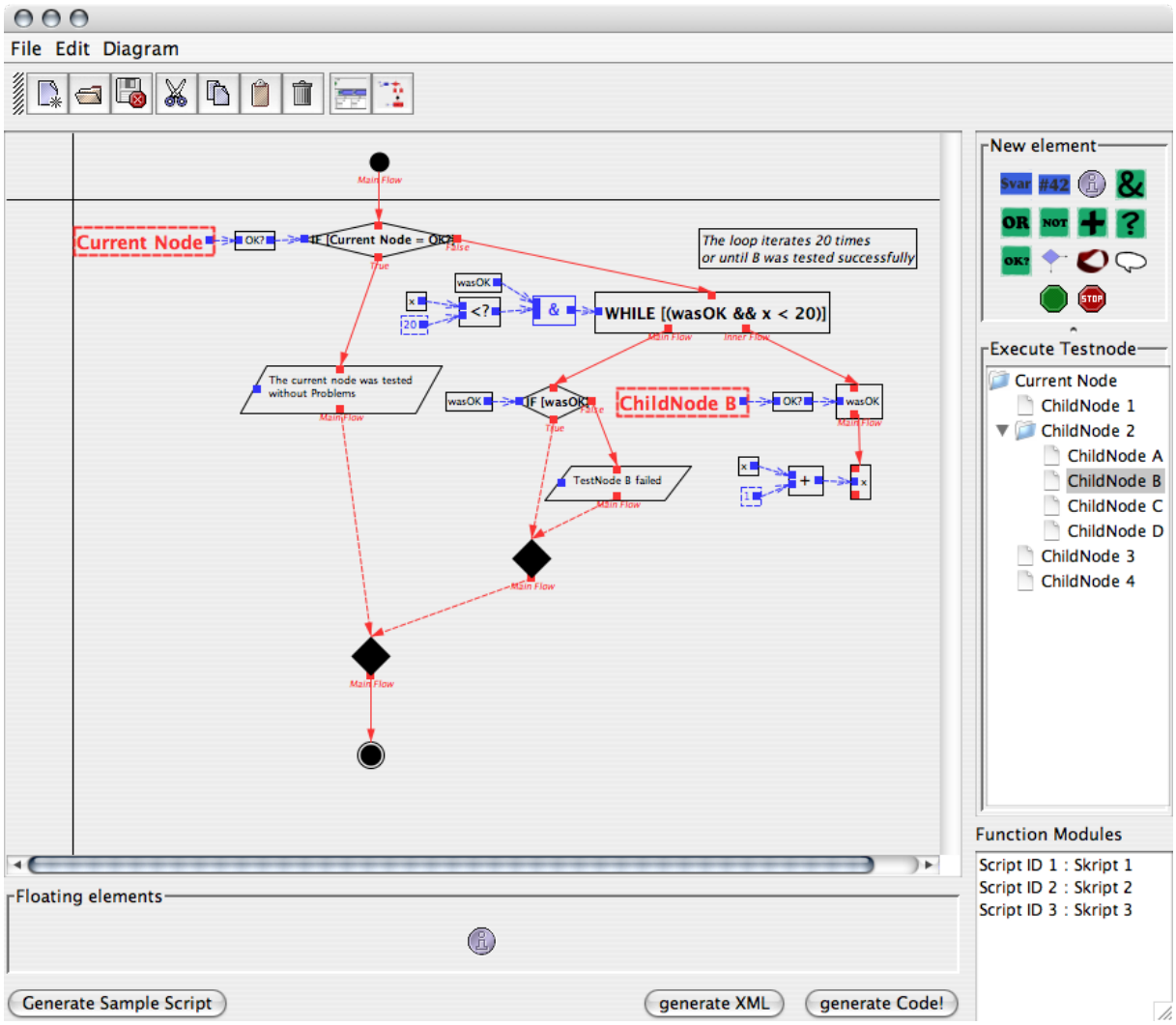


Abbildung 4.1: Benutzeroberfläche des visuellen Skripteditors

Im Anschluss daran folgt unterhalb die Werkzeugleiste. Diese beinhaltet, wie auch das Menü, Standardaktionen zur Bearbeitung des Skriptes (*Ausschneiden* etc.) und orientiert sich von den verfügbaren Aktionen her an der Werkzeugleiste des textuellen Editors. Auch Standardschaltflächen für Dateioperationen wie das Öffnen, Speichern, Importieren und Exportieren sind vorhanden. Des weiteren finden sich hier Buttons zum Rückgängigmachen der letzten durchgeführten Veränderung des Skriptes sowie zum Wiederholen derselben, die jedoch im Prototyp nicht vorhanden sind. Zusätzlich bestehen Symbolschaltflächen zum schnellen Wechseln der Diagrammansicht.

Auf der rechten Seite des Fensters befinden sich GUI-Elemente, die dem Einfügen einzelner Skriptelemente in das Diagramm dienen. Im oberen Bereich dieses Abschnitts befinden sich allgemeine Elemente wie Verzweigungen, Schleifen, Ausgaben etc. Diese werden mittels *Drag&Drop* in das Diagramm eingefügt und können im Anschluss daran mit anderen, schon vorhandenen Elementen verbunden werden. Im Falle des Blockdiagramms wird das neu eingefügte Element auf einem bereits Bestehenden abgelegt. Eine skriptseitige Verbindung wird (wenn möglich) automatisch hergestellt.

Im unteren Abschnitt dieses Einfügebereichs befindet sich eine dynamische Komponente – beispielsweise eine Baumansicht – die dem Einfügen von Elementen der *Lauts*-Teststruktur dient. Diese Elemente bewirken eine Ausführung des entsprechenden Tests. Der Inhalt dieser Komponente hängt von dem *Lauts*-Element ab, dem das gerade bearbeitete Skript zugeordnet ist. Es sind nur das aktuelle Element sowie dessen hierarchische Nachfolgeelemente enthalten. Ähnlich wie beim Einfügen von Standardskriptelementen erfolgt auch hier die Einfügeoperation in das Diagramm per *Drag&Drop*.

Im unteren Bereich des Editors befindet sich eine Komponente, die eine knappe Übersicht über nicht verbundene, „schwebende“ Skriptelemente bietet. Dieser Bereich dient vor allem im Blockdiagramm dazu, solche Elemente kenntlich zu machen, da in dieser Diagrammart nur solche Elemente direkt im Diagramm dargestellt werden, die direkt Bestandteil des Skriptes sind. Um diese losgelösten Elemente, wie sie beispielsweise durch Löschoperationen im Diagramm entstehen können, wieder in das Skript zu integrieren genügt es, die entsprechende Kurzdarstellung in ein Element zu ziehen, das mit einem solchen Skriptelement verbunden werden kann. Beim Flussdiagramm bietet dieser Be-

reich die Möglichkeit der Lokalisierung des schwebenden Elementes im Diagramm durch einen Doppelklick auf die Kurzdarstellung.

In der rechten unteren Ecke des Editors , im Grenzbereich des Einfügebereichs mit dem Bereich für schwebende Elemente, wird eine Komponente angezeigt, die das Einfügen selbst erstellter Skriptbausteine enthält. Diese Skriptbausteine werden von der Einfügemethodik her genau wie Standardelemente auch behandelt, ihre Verfügbarkeit jedoch ist dynamisch geregelt. Unter Zuhilfenahme dieser Komponente können komplexe Skriptabläufe bis hin zu rekursiven Programmen gestaltet werden.

Handhabung des Flussdiagramms

Das Flussdiagramm wird graphisch gestaltet, die einzelnen Elemente sind als Knoten des Graphen dargestellt. Das Einfügen eines neuen Elementes in das Diagramm erfolgt durch Ziehen des entsprechenden Icons in das Diagramm.

Will man ein Element bearbeiten, so führt man einfach einen Doppelklick auf das Element durch. Daraufhin erscheint der diagrammunabhängige Bearbeitungsdialog, in dem Veränderungen durchgeführt werden können.

Zum Verbinden der einzelnen Elemente wird unter Verwendung der rechten Maustaste eine Verbindungslinie vom Ausgangselement zum Zielelement gezogen. Ist die Verbindung erlaubt, wird der Pfeil im Diagramm dargestellt – die Verbindung ist hergestellt.

Handhabung des Blockdiagramms

Bei der Handhabung des Blockdiagramms sind die *Anker* der Elemente von entscheidender Bedeutung. Sie dienen als Angriffspunkt zum Verschieben und Bearbeiten der einzelnen Elemente.

Das Verschieben der Elemente erfolgt mittels *Drag&Drop*. Die Elemente werden bei gedrückter Maustaste gezogen und über einem anderen Element wieder losgelassen. Die Semantik des Skriptes wird automatisch an diese Umordnung der Elemente angepasst.

Die Bearbeitung eines Elements erfolgt wie auch im Flussdiagramm durch Doppelklick auf das Element.

4.1.2 Aufbau der Skripte

Prinzipiell sind Skripte prozedural aufgebaut. Sie beschreiben einen sequentiellen Ablauf einzelner Programmteile.

Ein Skript an sich besitzt immer genau ein Startelement und genau ein Endelement. Dies dient dazu, den Ablauf des Skriptes – der ja prozedural vonstatten geht – genau eingrenzen zu können. Zu beachten hierbei ist die Behandlung des Endknotens. Dieser kann einfach das Ende der Verarbeitung darstellen, jedoch auch als Ziel eines Datenobjekts dienen. In diesem Fall stellt der Endknoten die Rückgabe des betreffenden Wertes an ein übergeordnetes Skript dar. Dies findet hauptsächlich bei der Definition von Funktionsbausteinen Vewendung.

Ein Skript ist prinzipiell immer als Abfolge bzw. *Netz* einzelner Elemente zu charakterisieren. Die Darstellungsart derselben unterscheidet sich zwar von Diagrammart zu Diagrammmart, jedoch das grundlegende Prinzip bleibt gleich. Ausgangspunkt ist der Startknoten. Alle nachfolgenden Elemente des Skriptes – bis hin zum Endknoten – sind mittelbar mit diesem verbunden. Die Art dieser Verbindung kann kontrollflussorientiert (ein Anweisungsblock folgt sequentiell auf einen anderen) oder auch datenflussorientiert (ein Anweisungsblock nutzt ein anderes Element als Datenquelle zur Verarbeitung) sein.

Um komplexere Abläufe zu modellieren existieren spezielle Elemente, die mehrere Nachfolgeelemente besitzen können. Dies bezieht sich ausschließlich auf kontrollflussorientierte Verbindungen. Zu dieser Klasse von Elementen gehören beispielsweise Verzweigungen – sie besitzen jeweils ein Nachfolgeelement falls die Bedingung *wahr* bzw *falsch* ist sowie ein Nachfolgeelement, das im Anschluss in jedem Fall ausgeführt wird. Auch Schleifen fallen unter diese Kategorie, da sie sowohl ein „inneres“ Nachfolgeelement (das wiederrum Nachfolgeelemente haben kann) besitzen, als auch ein äußeres Nachfolgeelement, welches nach Einsetzen der Abbruchbedingung ausgeführt wird.

4.1.3 Integration in Lauts

Der visuelle Skripteditor wird genauso in das *Lauts*-System integriert wie auch der textuelle Skripteditor. Beim Doppelklick auf ein entsprechendes Element der Teststruktur öffnet sich ein Dialog, der dem Benutzer eine Auswahl des Editors ermöglicht. Sollte ein Testknoten noch kein Skript zugeordnet haben, so wird dem Benutzer die Möglichkeit gegeben, ein bereits in der Datenbank vorhandenes Skript zu öffnen um dieses dem Element zuzuordnen oder ein neues Skript zu erstellen. Nach der Bearbeitung dieses Skriptes kann der Nutzer dieses dann in die Datenbank speichern lassen. Ein Export bzw. Import eines vorhandenen Skriptes ist im Editor ebenfalls möglich.

Die Ausführung seitens der Testengine erfolgt genau wie beim normalen Vorgehen mittels traditionellem Skripteditor. Der vom visuellen Skripteditor generierte Quellcode wird im Zusammenhang mit der Ausführung des Testknotens erkannt und dessen Ausführung angestoßen.

Variablen und Parameter

Das zu erstellende Skript muss im Rahmen der Ausführung auch Zugriff auf Variablen und Parameter des *Lauts*-Systems besitzen. Dies ist notwendig für eine strukturierte Integration in das Gesamtsystem.

Sowohl beim Erstellen der Skripte im Editor, als auch später beim Ausführen auf der Testengine muss gewährleistet sein, dass diese Variablen vorhanden sind. Für den Zugriff auf Variablen gelten die Einschränkungen des Testknotens, dem das entsprechende Skript zugeordnet ist. Variablen behalten ihre Gültigkeit nur in den jeweiligen „Kindern“ des Knotens, in dem sie definiert sind.

4.2 Allgemeine Darstellung von Skriptelementen

Das Erstellen und Modifizieren von Skripten wird nach einem Baukastensystem vorgenommen. Hierzu werden einzelne Elemente des Skriptes in das Diagramm eingefügt und im Anschluss miteinander verbunden. Zusätzlich besteht die Möglichkeit, eigene Bausteine durch die weitere Erstellung von Skripten zu definieren. Diese können dann in anderen Skripten wiederum als Elemente des Skriptablaufs verwendet werden.

Hierzu ist es selbstverständlich notwendig, die einzelnen Elemente im Diagramm darzustellen um den eigentlichen Ablauf des Skriptes zu modellieren. Die Darstellung der Elemente unterscheidet sich abhängig von der verwendeten Diagrammart teilweise beträchtlich. Dies ist durch die unterschiedlichen Darstellungsparadigmen der beiden verwendeten Diagrammarten bedingt.

4.2.1 Flussdiagramm

Das Flussdiagramm ist in seiner Gestaltung stark am Programmablaufplan (wie in Abschnitt 1.4.2 beschrieben) orientiert. Jedoch fließen weiterhin Erkenntnisse aus dem Prototyping (siehe Abschnitt 3.2.2) ein. Einzelne Elemente des Skriptes werden als Knoten eines Graphen interpretiert. Die Verbindungen zwischen ihnen werden durch gerichtete Kanten (Pfeile) dargestellt. Diese stellen die Flussrichtung dar.

Es bestehen zwei Arten von Beziehungen zwischen Elementen:

1. **Datenfluss:** Der Datenfluss charakterisiert die Transformation und Verarbeitung von Datenelementen wie beispielsweise Variablen. Ihr Verlauf wird einheitlich von links nach rechts dargestellt. Verbindungspfeile und reine Datenobjekte werden blau eingefärbt dargestellt. Dies dient zur Hervorhebung der Datenobjekte und ihrer Verarbeitung.

2. **Kontrollfluss:** Der Kontrollfluss gibt den eigentlichen Ablauf des Programmes an. Dieser wird von oben nach unten senkrecht ausgerichtet. Verbindungspfeile und Kon-

trollflussobjekte werden rot gekennzeichnet. Somit lässt sich der Programmablauf leicht ersehen.

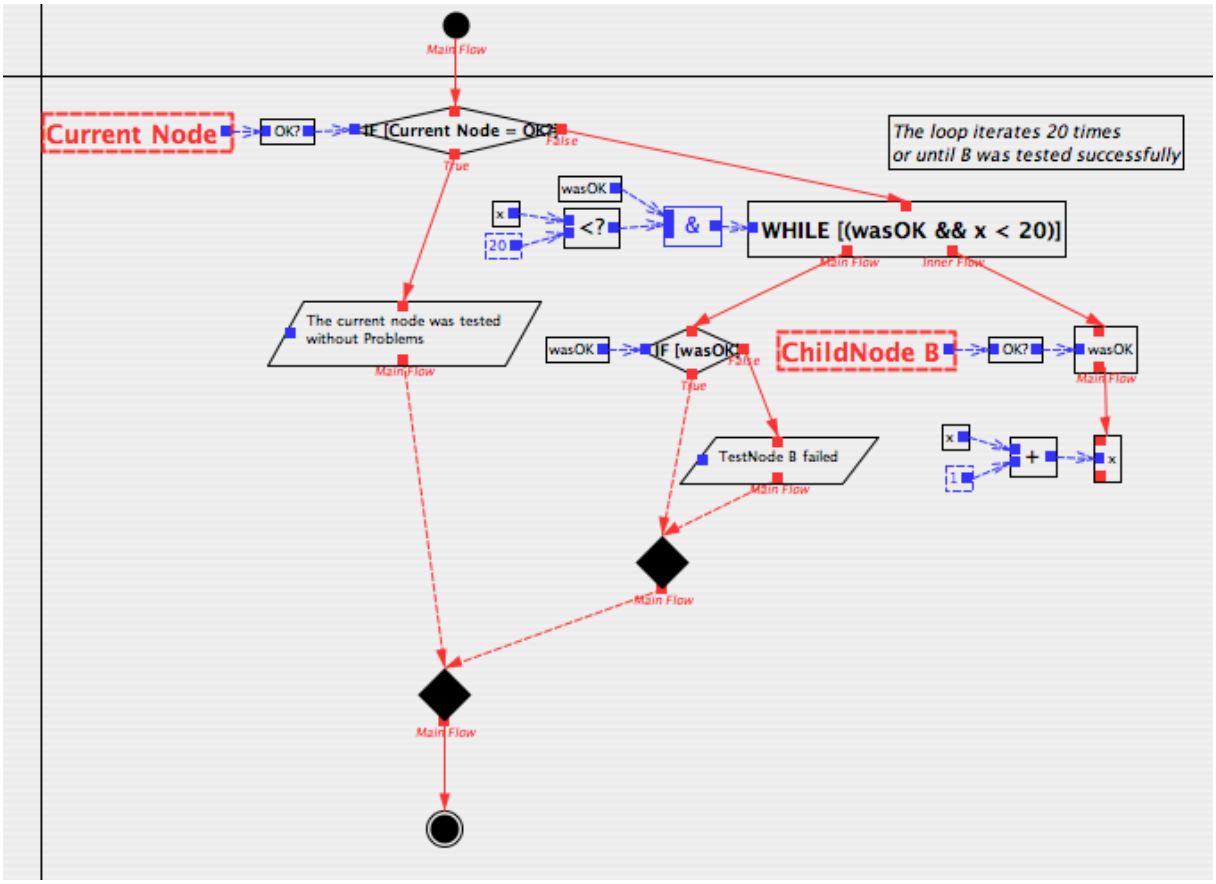


Abbildung 4.2: Detailliertere Darstellung des Flussdiagramms

Die Abbildung 4.2 gibt einen konkreteren Eindruck der Arbeitsweise des Flussdiagramms. Der Schwerpunkt dieser Darstellungsmethode liegt auf der Visualisierung von Flussrichtungen. Besonders die Zusammenhänge von Datenobjekten und ihren Zielelementen werden klar dargestellt.

Einschränkungen

Da die einzelnen Elemente prinzipiell frei positionierbar sind kann es vorkommen, dass die Darstellung unübersichtlich wird. So kann bei ungünstiger Verteilung mehrerer Elemente nicht mehr klar ersichtlich sein, in welcher Beziehung diese zueinander stehen.

Insbesonders sich kreuzende Verbindungslinien und sich überlappende Elemente stellen ein Problem für die Visualisierung dar.

Deshalb ist es angeraten, die Bewegung einzelner Elemente zueinander einigen Regeln zu unterwerfen. Die Spezifizierung dieser Regeln ermöglicht außerdem in beschränktem Maße eine automatische Positionierung der Elemente auf Knopfdruck.

Neben diesen Positionierungseinschränkungen ist es weiterhin sinnvoll, Richtlinien für die Verbindung der einzelnen Elemente aufzustellen. Diese gewährleisten, dass nur erlaubte Verbindungen im Diagramm vorgenommen werden. Somit können beispielsweise Kontrollflusselemente nur genau einen Vorgänger besitzen. Die Prüfung dieser Verbindungsrichtlinien erfolgt direkt durch Analyse der hinter dem Diagramm stehenden Semantik.

- **Positionierungseinschränkungen:** Bei den Positionierungseinschränkungen gilt es nicht nur direkt verbundene Elemente zu berücksichtigen. Obwohl diese vorher schon den Vorgängeransprüchen genügen mussten können auf nachfolgende Elemente noch weitere Einschränkungen hinzukommen. Grund hierfür ist die Vererbung der Restriktionen. Darf beispielsweise ein Nachfolgeelement einer WHILE-Schleife im inneren Ablaufteil nicht weit nach links verschoben werden um diese Beziehung hervorzuheben muss diese linke Begrenzungslinie auch alle ihre Nachfolgeelemente betreffen. Definiert man nur die gewöhnliche Einschränkung dass Kontrollflussnachfolger nicht oberhalb ihres Vorgängers positioniert werden dürfen würde die Einschränkung in der horizontalen Achse ohne Vererbung wegfallen.

- **Verbindungseinschränkungen:** Die Verbindungseinschränkungen werden jedes mal beim Ziehen einer neuen Verbindungslinie abgeprüft. Mit ihrer Hilfe werden unerlaubte Verbindungen erkannt und verweigert. Dies bezieht sich sowohl auf die Kontrollflusskriterien als auch auf die Datenflusskriterien.

  Kontrollfluss: Ein Element kann nur jeweils ein Vorgängerelement besitzen. Dies dient der genauen Strukturierung des Skriptablaufs und beugt der Erzeugung von Endlosschleifen vor. Zusätzlich wird beim Erstellen einer Verbindung geprüft, ob durch diese Verbindung eine zyklische Beziehung im Kontrollfluss

entstehen würde. Auch diese Maßnahme zielt auf die Vermeidung von Endlosschleifen hin.

Datenfluss: Auch beim Datenfluss wird vor dem eigentlichen Ziehen der Verbindung geprüft, ob dadurch eine zyklische Beziehung entstehen würde. Somit sind Rückkopplungen ausgeschlossen. Weiterhin wird geprüft, ob das Zielelement den Datentyp der Datenquelle auch akzeptiert. Dies dient als semantischer Prüfungsmechanismus um Fehler zur Laufzeit zu verhindern.

An dieser Stelle sei noch einmal auf Abschnitt 3.2.2 des Prototypings verwiesen, in dem das Flussdiagramm näher betrachtet wird.

4.2.2 Blockdiagramm

Das Blockdiagramm ist von der Gestaltung her stark an das Nassi-Shneiderman-Diagramm (siehe Abschnitt 1.4.1) angelehnt. Erkenntnisse aus dem Prototyping wie in Abschnitt 3.2.1 geschildert fließen natürlich mit in die Umsetzung ein.

Prinzipiell werden im Blockdiagramm zwei Arten von Beziehungen deutlich dargestellt:

1. **Sequentielle Abfolge:** Diese Beziehung stellt den einfachsten Regelfall dar. In einer Anweisungskette werden einfach die aufeinander folgenden Elemente der Reihe nach abgearbeitet. Die Darstellung dieses Ablaufpfades erfolgt als vertikale Aneinanderreihung der einzelnen Anweisungsblöcke.

2. **Hierarchische Struktur:** Diese Beziehung dient zum einen zur Darstellung von Datenquellen und ihrem zugeordneten Element, zum anderen jedoch auch der Visualisierung von strukturellen Eigenschaften des Programmablaufs. So enthält beispielsweise eine Schleife als Element alle Anweisungsblöcke, die innerhalb der Schleife (also mehrfach) ausgeführt werden. Dies dient zur schnellen optischen Abgrenzung solcher hierarchischer Strukturen im Programmablauf. Auch bei Datenquellen lässt sich durch diese Darstellungstechnik schnell feststellen, wofür sie benutzt werden, da sie in dem betreffenden Element enthalten sind. Dieser Ansatz schließt jedoch

die mehrfache Verwendung von Datenquellen im selben Programm – ohne die Verwendung von Variablen – aus.

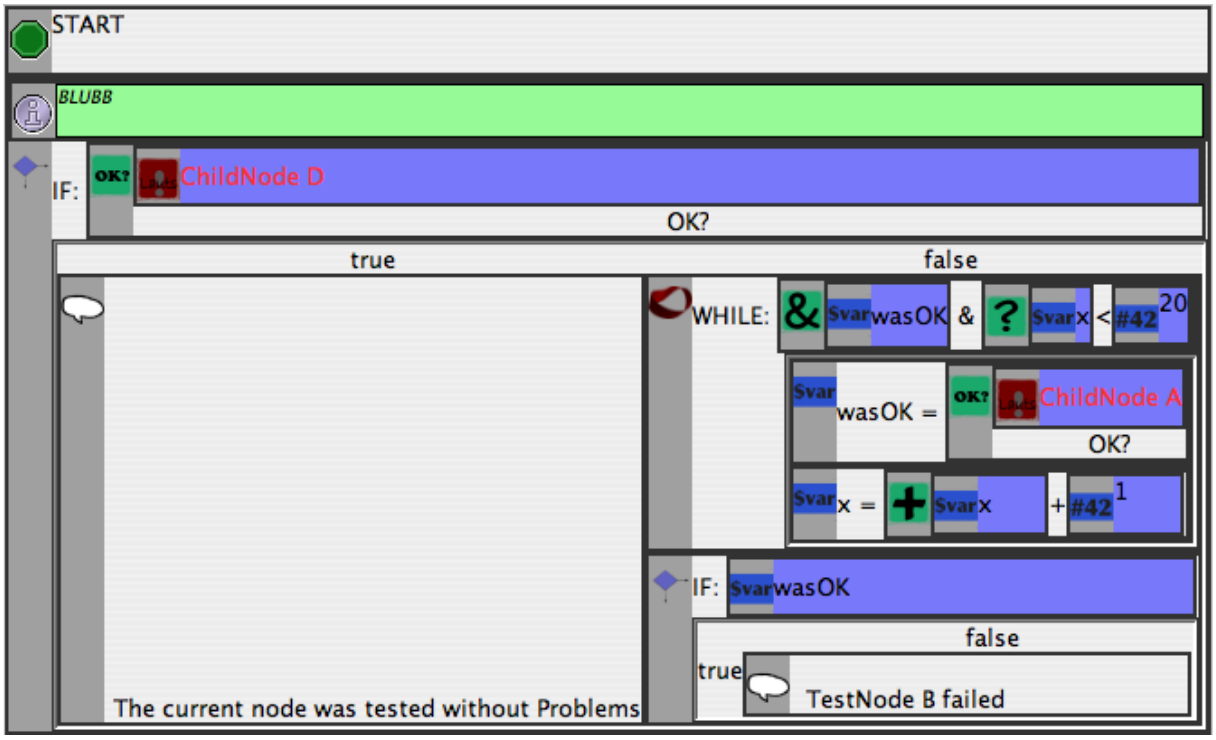


Abbildung 4.3: Detailliertere Darstellung des Blockdiagramms

Abbildung 4.3 verdeutlicht nochmals den grundlegenden Aufbau des Blockdiagramms. Hierarchische Verbindungen zwischen einzelnen Elementen sind gut erkennbar, jedoch birgt die kompakte Darstellung einzelner Elemente noch Verbesserungspotential für den Prototyp. Vor allem der Einsatz selbst definierter Komponenten ist angebracht um eine stärkere Annäherung an Nassi-Shneiderman-Diagramme zu erreichen.

Die Verbindung der einzelnen Elemente erfolgt mittels *Drag&Drop*-Verfahren. Die Elemente besitzen einen „Anker“ an ihrem linken Rand, an dem der Benutzer das entsprechende Element greifen kann um es dann über einem anderen Element abzusetzen. Dies ist sowohl für Elemente, die Bestandteil des Skriptes sind, als auch für „schwebende“ Elemente möglich. Weiterhin dient der Anker zur genaueren Identifikation des Elements, da er eine Darstellung des Piktogrammes ist, welches auch für das Einfügen neuer Elemente verwendet wird. Auch das Einfügen neuer Elemente in das Skript erfolgt auf diese Art. Beim Absetzen des Elementes wird jedoch zuerst geprüft, ob die Zielposition für dieses

Element erlaubt ist. Dies entspricht den Verbindungseinschränkungen wie sie auch im Flussdiagramm gültig sind.

Um Datenquellen zu kennzeichnen werden diese stets als in ihrem Ziel enthalten dargestellt. Die Verwendung von festen Positionen für diese Elemente hilft bei der Orientierung in dem Skript. Auch die farbliche Kennzeichnung der Datenflusselemente dient diesem Zweck. Wie auch im Flussdiagramm werden sie blau gefärbt dargestellt. Somit lassen sich kritische Datenobjekte für einzelne Anweisungsblöcke leicht identifizieren.

Der Schwerpunkt der Darstellung im Blockdiagramm liegt ganz klar auf einer übersichtlichen Repräsentation von Elementhierarchien. Einzelne Anweisungsblöcke sind klar umrissen dargestellt, Elemente die wiederum andere Elemente enthalten – wie beispielsweise Schleifen oder Verzweigungskonstrukte – besitzen eine klare Trennung ihrer Unterelemente von den benachbarten Nachfolger- und Vorgängerelementen.

4.3 Darstellung der einzelnen Sprachkonstrukte

An dieser Stelle soll ein Einblick in die Darstellung der einzelnen Elemente der Skripte gegeben werden. Die Darstellungsvarianten der Skriptelemente, die vom jeweilig ausgewählten Diagrammtyp abhängen, werden vergleichend gegenübergestellt. Dies ermöglicht den direkten Vergleich der Stärken und Schwächen des jeweiligen Darstellungskonzepts.

Zunächst soll jedoch an dieser Stelle eine allgemeine Darstellung der einzelnen Skriptelemente als Icons erfolgen. Die Icons finden beim Einfügen neuer Elemente in das Diagramm Verwendung. Auch Elemente des Blockdiagramms werden durch ihre Einbindung gekennzeichnet. Dies ermöglicht eine schnellere Identifizierung des dargestellten Elements als dies ohne die Verwendung der Symbole möglich wäre. Abbildungen 4.4 und 4.5 geben einen Überblick über die verwendeten Symbole.

(a)
Logisches *UND*

(b)
Logisches *ODER*

(c)
Logisches *NICHT*

(d)
Vergleichsoperation

(e)
Mathematische Operation

(f)
Konstanten

(g)
Variablen

(h)
Lauts-Elementauswertung

(i)
Lauts-Elementausführung

Abbildung 4.4: Icons der datenflussorientierten Skriptelemente

(a)
Ausgabeelement

(b)
Aufruf eines anderen Skriptes

(c)
Verzweigung

(d)
Kommentar

(e)
Schleife

(f)
Startknoten

(g)
Ende-Knoten

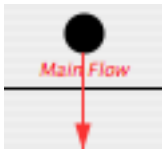
Abbildung 4.5: Icons der kontrollflussorientierten Skriptelemente sowie der Kommentare

4.3.1 Skriptanfang und Skriptende

Diese beiden Elemente existieren in jedem Skript genau einmal. Sie definieren den genauen Startpunkt des Skriptes sowie dessen Ende.

Skriptanfang

Der Skriptanfang stellt den eigentlichen Einsprungspunkt des Skriptes dar. Von hier aus wird die Abarbeitung aller nachfolgender Kontrollflusselemente begonnen.



(a)
Darstellung im Flussdiagramm

(b) Darstellung im Blockdiagramm

Abbildung 4.6: Darstellung von Startelementen

Skriptende

(a)
Darstellung im Flussdiagramm

(b) Darstellung im Blockdiagramm

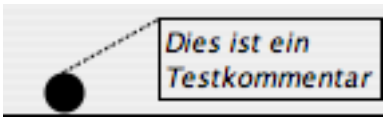
Abbildung 4.7: Darstellung von *Ende*-Elementen

Das Skriptende stellt das absolute Ende dieses Skriptes / Skriptbausteines dar. Dieses kann in zwei Verianten auftreten – entweder das Skript besitzt keine Rückgabeparameter

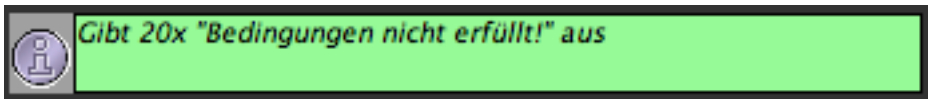
(dann besitzt es keine Datenquelle) oder es wird genau ein Wert zurückgegeben (dann wird der *ENDE*-Knoten als Ziel einer Datenflusskette verwendet). Auch in letzterem Fall ist es notwendig, eine Kontrollflussverbindung zum Skriptende zu verwenden.

4.3.2 Kommentare und Ausgaben

Kommentare



(a)
Darstellung im Flussdiagramm



(b) Darstellung im Blockdiagramm

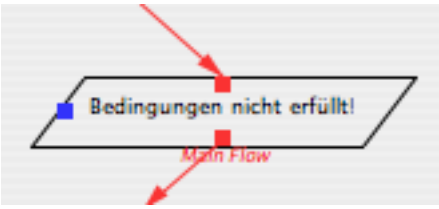
Abbildung 4.8: Darstellung von Kommentaren

Kommentare können, wie bereits erwähnt, jedem beliebigen Skriptelement (mit Ausnahme von Kommentaren) zugeordnet werden. Diese Zuordnung stellt ihre einzige Verbindung zum restlichen Skript dar, da sie programmtechnisch nicht relevant sind. Deswegen besitzen sie, wie Abbildung 4.8 illustriert, im Flussdiagramm einen anderen Kantentyp als andere Elemente um die Verbindung auszudrücken. Im Blockdiagramm hingegen werden Kommentare als *in einem Element enthalten* dargestellt. Dies betont die eineindeutige Zuordnung dieser beiden Elemente zueinander und dient der klaren Beschreibung einzelner Elemente.

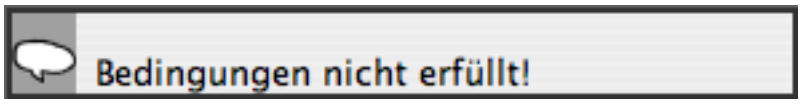
Ausgaben

Ausgaben werden als Kontrollflussziele charakterisiert, da sie einer konkreten Anweisung des Skriptes entsprechen. Zusätzlich ist es möglich, ihnen verschiedene Datenquellen zuzuordnen. Dies dient dazu, nicht nur statischen Text ausgeben zu können sondern auch

laufzeittechnisch relevante Objekte, was für die Fehlersuche günstig ist. Die Formatierung des auszugebenden Texts erfolgt durch eine Eigenschaft der Ausgabeelemente – einer Textmaske – die als Attribut des Elements vermerkt wird.



(a) Darstellung im Flussdiagramm

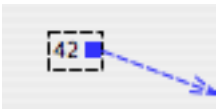


(b) Darstellung im Blockdiagramm

Abbildung 4.9: Darstellung von Ausgaben

4.3.3 Konstanten

Konstanten sind wichtige Elemente beim Erstellen von Skripten. Sie dienen ausschließlich als Datenquellen für andere Elemente, werden jedoch häufig gebraucht. Wegen dieser häufigen Verwendung wurde bewusst eine möglichst kompakte Darstellung in allen Diagrammtypen gewählt. Die Eigenschaften von Konstanten (ihr Wert und ihr Datentyp) können beim Bearbeiten der Elemente verändert werden, im Diagramm hingegen wird nur ihr Wert angezeigt.



(a) Darstellung im Flussdiagramm

(b) Darstellung im Blockdiagramm

Abbildung 4.10: Darstellung von Konstanten

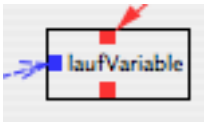
4.3.4 Variablenoperationen

Ein weiteres häufig verwendetes Element sind Variablen. Diese können prinzipiell auf zwei unterschiedliche Arten verwendet werden:

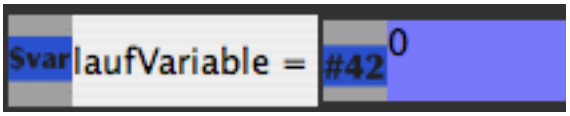
1. **Lesend:** Diese Verwendungsart stellt das Auslesen eines Variablenwertes innerhalb des Skriptes dar. Die Variable fungiert somit als Datenquelle für nachfolgende Elemente.

2. **Schreibend:** Hierbei wird einer Variablen ein bestimmter Wert zugewiesen. Dieser stammt aus einer Datenquelle und muss vom gleichen Typ wie die Variable sein. Die Darstellung im Diagramm erfolgt als Kontrollflussziel, da eine Zuweisung eine Anweisung für das Skript darstellt.

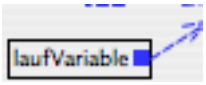
Die Darstellung des Elementes an sich bleibt jedoch möglichst unabhängig von der Verwendungsart um den Anwender nicht zu verwirren. Der einzige Darstellungsunterschied besteht im Flussdiagramm in der Führung der Kanten die an das Element grenzen. Im Blockdiagramm jedoch ist dies, bedingt durch die allgemeine Darstellungsmethodik, Datenquellen als in ihrem Ziel enthalten darzustellen nicht haltbar. Hier unterscheiden sich beide Verwendungsarten wie Abbildung 4.11 zeigt.



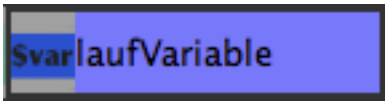
(a)
Variablenzuweisung im Flussdiagramm



(b) Variablenzuweisung im Blockdiagramm



(c)
Lesen einer Variablen im Flussdiagramm



(d)
Lesen einer Variablen im Blockdiagramm

Abbildung 4.11: Darstellung von Variablen

4.3.5 Weitere Datenelemente

Diese Elemente können sowohl zur Typumwandlung von Datenquellen, als auch zur Veränderung ihrer Werte dienen. Diese Gruppe von Elementen besitzt Datenquellen (typischerweise zwei) und genau einen Ausgangswert. Der Typ dieses Ausgangswertes hängt, wie auch der Datentyp der Eingangswerte, vom jeweiligen Datenflusselement ab.

Logische Operationen

Logische Operationen besitzen bool'sche Eingangswerte. Diese werden dann in einen der bool'schen Algebra entsprechenden Ausgangswert umgewandelt. Zu diesen Elementen zählen insbesondere logische *UND* (Abbildung 4.12) und *ODER*- Verknüpfungen (Abbildung 4.13), die zur Auswertung mehrerer Bedingungen für Schleifen und Verzweigungen dienen. Auch ein *NICHT*-Operator (Abbildung 4.14) existiert zur Abbildung komplexerer logischer Strukturen.

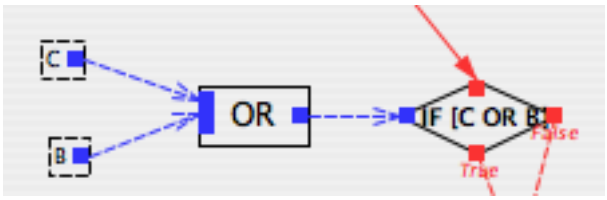
(a)
Darstellung im Flussdiagramm

(b) Darstellung im Blockdiagramm

Abbildung 4.12: Darstellung von logischen *UND*-Elementen

Auswertungsoperationen

Auswertungsoperationen dienen dazu, Datenquellen eines anderen Datentyps in einen bool'schen Wert umzusetzen. Dies ist vor allem für Verzweigungen und Schleifen unbedingt notwendig um „komplexere" Bedingungen abzuprüfen. Auch können Auswertungen dazu dienen, ein erzeugtes *Lauts*-Testergebnis auf einen Testerfolg hin zu überprüfen.



(a) Darstellung im Flussdiagramm

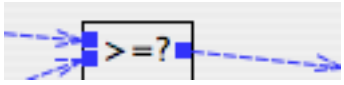
(b) Darstellung im Blockdiagramm

Abbildung 4.13: Darstellung von logischen *ODER*-Elementen

(a)
Darstellung im Flussdiagramm

(b) Darstellung im Blockdiagramm

Abbildung 4.14: Darstellung von logischen *Nicht*-Elementen



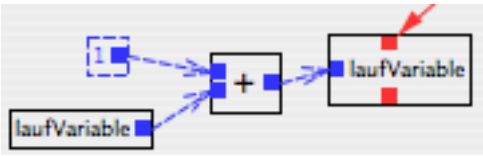
(a)
Darstellung im Flussdiagramm

(b) Darstellung im Blockdiagramm

Abbildung 4.15: Darstellung von Vergleichsoperationen

Mathematische / Wertverändernde Operationen

Mathematische Operationen dienen dazu, Werte von Variablen zu verändern. Dies kann beispielsweise die Addition einer Konstanten zu einem Variablenwert (wie in einer Zählschleife), oder aber auch das Zusammensetzen einer Zeichenkette bedeuten. Die Werttypen der Datenquellen müssen auch hier den Operationen entsprechen, sonst kann keine sinnvolle Verbindung hergestellt werden.



(a) Darstellung im Flussdiagramm

(b) Darstellung im Blockdiagramm

Abbildung 4.16: Darstellung von mathematischen / wertverändernden Operationen

4.3.6 Anweisungsblöcke

Anweisungsblöcke dienen zum Ausführen der *Lauts*-eigenen Testobjekte. Sie dienen als Datenquellen, die *Testergebnisse* zurückliefern. Somit werden sie in den Datenfluss eingebunden und erzeugen immer ein Objekt, das das Testresultat repräsentiert. Somit dienen Anweisungsblöcke als Datenquellen, deren Ergebnis dann entweder direkt in Variablen gesichert werden kann, oder durch spezielle Elemente ausgewertet wird. Sie werden im Gegensatz zu normalen Skriptelementen erzeugt, indem Elemente der *Lauts*-Unterknotenübersicht in das Diagramm gezogen werden.

4.3.7 Verzweigungen und Schleifen

Verzweigungen und Schleifen stellen einen Sonderfall dar, da sie im Gegensatz zu allen anderen Anweisungselementen jeweils mehrere Kontrollflussausgänge haben. Dies liegt in

(a)
Darstellung im Flussdiagramm

(b)
Darstellung im Blockdiagramm

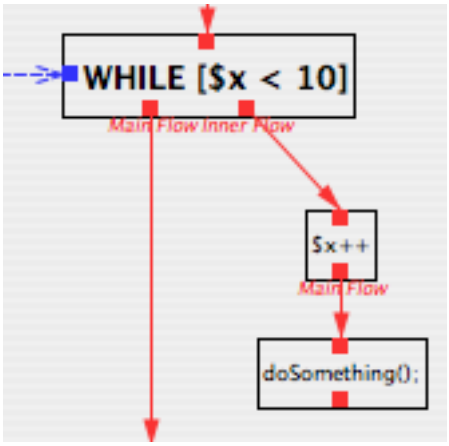
Abbildung 4.17: Darstellung von auszuführenden *Lauts*-Elementen

der Dynamik der nachfolgenden Elemente begründet. Als Gemeinsamkeit besitzen beide Elemente einen Eingang für Datenquellen. Dieser benötigt eine *bool'sche* Datenquelle als Bedingung für die Ausführungssteuerung der Nachfolgeelemente.

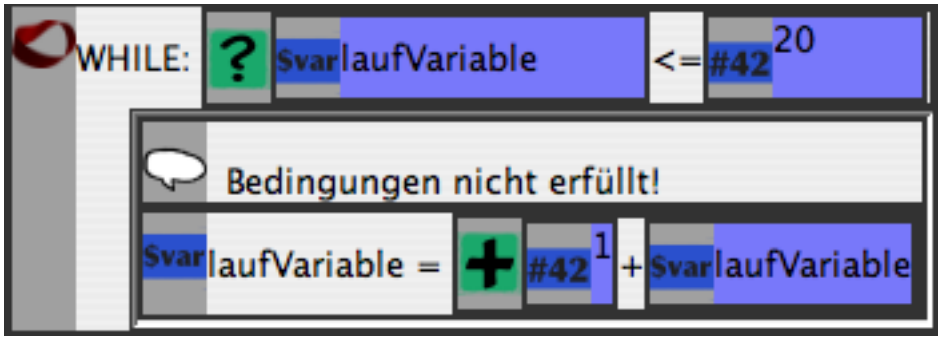
Abbildung 4.18 verdeutlicht die allgemeine Darstellung von Schleifen in beiden Diagrammtypen. Die beiden nachfolgenden Kontrollflusspfade (innerer, wiederholter Teil der Schleife und äußerer, endgültig nachfolgender Teil) sind besonders im Flussdiagramm klar zu erkennen. Das Blockdiagramm hebt besonders die Schachtelung des Ablaufs hervor und erlaubt eine wesentlich kompaktere und übersichtlichere Visualisierung der Schleife in dem Gesamtskript. Es wird klar deutlich, welche Elemente wiederholt ausgeführt werden.

Verzweigungen besitzen noch einen weiteren Kontrollflussausgang, da ihr „innerer“ Teil zwei getrennt zu betrachtende Ablaufpfade enthält. Diese werden je nach Auswertung der Bedingung unabhängig voneinander durchlaufen. Der dritte Kontrollflussausgang verbindet das Konstrukt mit dem – in jedem Fall unabhängig von der gewählten Bedingung auszuführenden – Nachfolgeelement.

Auch hier findet sich eine kompaktere Darstellung im Blockdiagramm. Das Verzweigungselement ist klar von umgebenden Elementen abgegrenzt. Im Flussdiagramm jedoch findet sich eine anschaulichere Darstellung der beiden internen Ablaufpfade. Diese können optisch einfach entlang ihrer Kanten unabhängig voneinander verfolgt werden bis sie am Ende der Verzweigung wieder zusammengeführt werden.

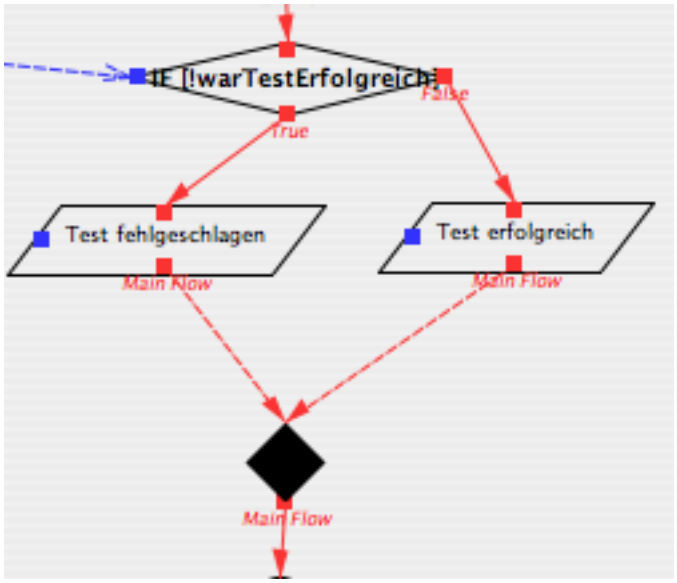


(a) Darstellung im Flussdiagramm



(b) Darstellung im Blockdiagramm

Abbildung 4.18: Darstellung von Schleifen



(a) Darstellung im Flussdiagramm

(b) Darstellung im Blockdiagramm

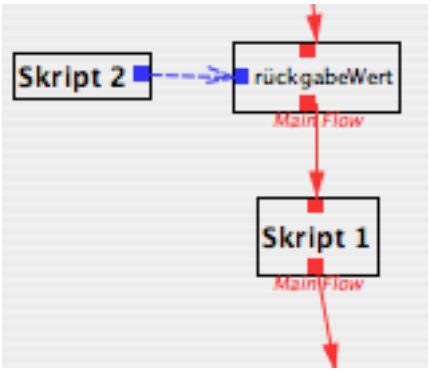
Abbildung 4.19: Darstellung von Verzweigungen

4.3.8 Funktionsbausteine und generische Elemente

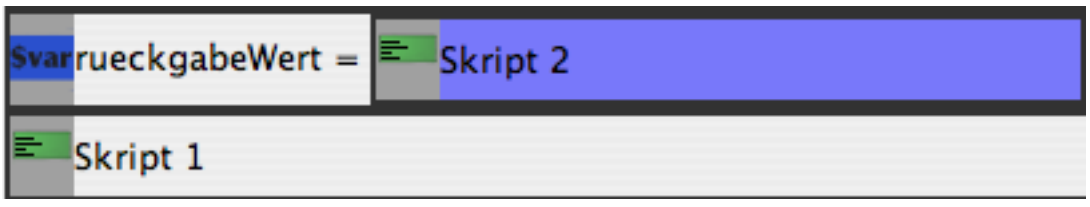
Funktionsbausteine

Funktionsbausteine stellen weitere Aufrufe eines anderen Skriptes innerhalb eines Skriptes dar. Sie können entweder einfach als Anweisungsblock verwendet werden oder eine Datenquelle darstellen. Bei der Verwendung als Anweisungsblock werden sie in den Kontrollfluss eingebunden. Eventuell vorhandene Datenquellen werden wie bei anderen Elementen als solche gekennzeichnet und dienen bei der Ausführung des Funktionsaufrufs als Übergabeparameter.

Wird ein Funktionsbaustein hingegen als Datenquelle benutzt wird sein Rückgabewert in einem anderen Element weiterverarbeitet. Im einfachsten Fall erfolgt seine Speicherung in einer Variablen, die dann später ausgewertet werden kann.



(a) Darstellung im Flussdiagramm



(b) Darstellung im Blockdiagramm

Abbildung 4.20: Darstellung von Funktionsaufrufen

Generische Elemente

Generische Elemente stellen Anweisungsblöcke dar, die beim Einparsen eines Skriptes nicht eindeutig in eine semantische Repräsentation überführt werden konnten. In ihnen werden die nichtübersetzten Codefragmente zusammen mit ihrer jeweiligen Skriptsprache hinterlegt. Solche generischen Elemente sollten bei der Erstellung eigener Skripte nicht verwendet werden, da sie das Skript von einer bestimmten Skriptsprache abhängig machen. In Ausnahmefällen kann es jedoch notwendig sein, solche Elemente zu verwenden um Abläufe darzustellen, die mit den Standardelementen nicht abbildbar wären. In diesem Fall wäre die Verwendung eines textuelle geschriebenen Funktionsbausteins sinnvoller.

4.3.9 Schwebende Elemente

Elemente ohne Verbindung zu anderen Elementen stellen sogenannte „Schwebende Elemente“ dar. Ihre Darstellung ist von der verwendeten Diagrammart abhängig. Im Flussdiagramm sind solche schwebenden Elemente einfach als normales Elemente ohne jegliche Verbindungskanten erkennbar. Im Blockdiagramm jedoch sind sie prinzipiell unsichtbar, da in diesem Diagramm nur Elemente dargestellt werden, die mittelbar mit dem Startknoten verbunden sind. Wie schon in Abschnitt 4.1.1 erläutert werden diese Elemente immer in einem gesonderten Bereich dargestellt. Dies ermöglicht dem Nutzer eine Schnellübersicht über die Elemente, die nicht direkt im Skript verwendet sind. Das Verhalten, das durch die Auswahl eines solchen Elements angestoßen wird, ist in beiden Diagrammarten unterschiedlich.

Wird gerade das Blockdiagramm verwendet, kann das gelöste Element per *Drag&Drop* mit anderen Elementen, die Bestandteil des Skriptes sind, verbunden werden.

Im Flussdiagramm hingegen erfolgt als Reaktion auf das Anklicken eines solchen Elementes eine automatische Positionierung des Diagrammausschnitts und eine automatische Auswahl des entsprechenden Elementes im Diagramm.

4.4 Interne Repräsentation der Sprachkonstrukte

Das Diagramm zur Darstellung der Skripte basiert auf dem Model/View/Controller-Design (MVC-Design). Dies bedeutet im konkreten Fall, dass die Repräsentation der eigentlichen Semantik des Skriptes unabhängig von seiner Darstellung im Diagramm erfolgt. Dies hat sowohl eine Verbesserung der allgemeinen Struktur des Editors als auch die Austauschbarkeit der Diagrammansicht zur Folge. Zusätzlich ermöglicht es eine getrennte Bewertung der Semantik des Skriptes von seiner graphischen Repräsentation. Somit wird die Flexibilität des gesamten Editors erhöht, da austauschbare Zwischenschichten entstehen.

4.4.1 Eigenschaften der Konstrukte

Allgemein

Die Skriptelemente besitzen allgemein gewisse Eigenschaften, die ihnen zugeordnet werden können. Diese dienen primär dazu, ein Netz aus ihnen aufzubauen, das dann bei der Übersetzung in der entsprechenden Zielsprache abgebildet wird. Zu den allgemeinen Eigenschaften der Konstrukte zählen:

- **Kommentare:** Jedem einzelnen Element (außer Kommentaren) kann ein Kommentar zugeordnet werden, der das Element beschreibt. Dies dient zur Strukturierung des erstellten Skriptes.

- **Datenflussquellen:** Falls ein Element Datenquellen benutzt (beispielsweise eine Schleife zur Überprüfung des Abbruchkriteriums) wird eine Referenz auf diese im Objekt vermerkt.

- **Kontrollflussziele:** Auch die jeweils nachfolgenden Elemente werden im Ausgangselement referenziert. Ihre Reihenfolge ist nicht zufällig gewählt und entspricht dem *Pfad* der Ausführung.

- **Die Kontrollflussquelle:** Diese Referenz charakterisiert das Vorgängerelement im Kontrollflussablauf. Diese Referenz wird für das Lösen des Elementes aus der Skriptstruktur (beispielsweise beim Löschen des Elementes) sowie für semantische Prüfungen verwendet.

- **Der zugeordnete Pfad:** Dieser Wert dient dazu, der Darstellungsschicht Informationen über die detailliertere strukturelle Anordnung der Elemente mitzuteilen. Somit lassen sich beispielsweise Positionseinschränkungen in Abhängigkeit der Beziehung der Elemente untereinander realisieren.

- **Maximale Anzahl an Kontrollflusszielen und Datenquellen:** Diese Werte dienen zur Überprüfung, ob eine neue Verbindung zu einem anderen Element gestattet werden kann. Die Maximalwerte werden bei der Erstellung eines neuen Elements in Abhängigkeit von seinem Typ gesetzt.

- **Die zu benachrichtigende Objekte:** Diese Referenzen dienen dazu, die Darstellungsschicht automatisch über Änderungen in der internen Repräsentation zu benachrichtigen. Dies kann beispielsweise beim Erstellen einer neuen Verbindung oder auch beim Verändern der individuellen Werte des Elements ausgelöst werden.

Einzelne Elemente

Einzelne Elemente besitzen abhängig von ihrem Typ noch weitere Eigenschaften, die sie näher charakterisieren und auch im semantischen Modell hinterlegt werden müssen.

- **Vergleichsoperationen:** Vergleichsoperationen dienen dazu, zwei Eingangswerte (Variablen oder Konstanten) miteinander zu vergleichen um einen bool'schen Wert zu weiteren Behandlung (z.B. für Verzweigungen) zu erhalten. Hierzu ist es notwendig, den Typ des Vergleichs zu hinterlegen. Es wird beispielsweise geprüft, ob der erste Wert dem zweiten entspricht, er größer oder kleiner ist und inwieweit die Grenzbedingungen zutreffen. Mögliche Werte sind somit $=$, $>$, $\geq$, $<$ und $\leq$.

- **Konstanten:** Konstanten besitzen einen fest voreingestellten Wert. Sie dienen beispielsweise zur Überprüfung von Bedingungen oder zur Festlegung von Schleifendurchläufen. Zusätzlich ist ihnen, wie allen Datenquellen, eine Eigenschaft zugeordnet, die ihren *Typ* eindeutig charakterisiert. Anhand dieser kann entschieden werden, ob diese Datenquelle zu ihrem entsprechenden potentiellen Ziel kompatibel ist.

- **Variablen:** Variablen sind prinzipiell den Konstanten recht ähnlich. Auch sie haben einen bestimmten *Typ*, der hinsichtlich ihrer Verwendung maßgeblich ist. Zusätzlich jedoch müssen sie einen Namen zugeordnet haben, der sie eindeutig identifiziert. Unter Verwendung des Namens kann dann im Anschluss weiterhin auf ihren Wert zugegriffen werden. Dieser wird selbst nicht im Skripteditor hinterlegt, da er eben variabel ist. Die Wertzuweisung erfolgt durch die Verwendung von Konstanten oder anderen Datenquellen, wobei auch hier wieder der *Typ* der Datenquelle und der Variablen entscheidend sind. Zusätzlich besitzen sie eine Eigenschaft, der ihre Verwendung charakterisiert. Sie gibt an, ob von der Variablen gelesen, oder ihr ein Wert zugewiesen wird.

- **Ausgaben:** Ausgaben besitzen als Eigenschaft eine Textmaske, die eine entsprechende Formatierung der Ausgabe erlaubt. Der Text von eventuell vorhandenen Datenquellen wird an die entsprechenden Stellen dieser Maske eingefügt. Dies ermöglicht es, das Ausgabeformat anzupassen um auftretende Meldungen eingängiger zu gestalten.

- **Kommentare:** Kommentare besitzen als einzige Eigenschaft ihren Kommentartext. Sie können weiterhin keine hierarchisch nachfolgenden Elemente besitzen.

- **Generische Elemente:** Solche Anweisungsblöcke dienen zur direkten Abbildung von Quelltextstücken. Dies kann dazu dienen, einzelne Funktionalitäten einer bestimmten Skriptsprache in einzelne Elemente zu kapseln. Ihre Eigenschaften erstrecken sich auf den eigentlichen Quelltext sowie die verwendete Skriptsprache, da solche textuellen Elemente sprachenspezifisch sind. Generische Elemente könnten beim Einparsen eines Quelltexts in den Editor entstehen, wenn unerkannte Textblöcke in den visuellen Editor gebracht werden sollen. Es existiert auch eine generische Repräsentation für unbekannte Textblöcke, die als Datenquellen erkannt wurden.

- **Anweisungen zum Ausführen von *Lauts*-Elementen:** Die Anweisung zum Ausführen eines *Lauts*-Element enthält alle Eigenschaften, die das auszuführende Element charakterisieren. Hierzu gehört ausdrücklich die eineindeutige ID des auszuführenden Testknotens (benötigt für die Ausführung auf der Testengine) und der dazugehörige Name zur Anzeige im Skripteditor.

- **Funktionsbausteine:** Funktionsbausteine dienen zum Aufrufen extern definierter Skripte. Somit benötigen sie als Eigenschaft in erster Linie den Namen des Skriptes und seine eineindeutige ID innerhalb der Datenbank. Auch der Rückgabetyp des Skriptes wird als Attribut hinterlegt, um die Einbindung in das aktuelle Skript korrekt vornehmen zu können. Sie werden in Abschnitt 4.5 näher betrachtet.

4.4.2 Darstellung in XML

Für die Repräsentation der Semantik wird eine XML-basierte Metasprache verwendet. Diese findet sowohl beim Exportieren eines Skriptes als auch als Vorstufe beim Erzeugen des Quelltextes in der Zielsprache Verwendung. Grund hierfür ist zum Einen die starke Erhöhung der Flexibilität bei der Erweiterung der Sprachkonstrukte, zum Andern bringt dieses Vorgehen auch Vorteile bei der Übersetzung in die jeweilige Zielsprache. Die Verwendung von XML ist eine Schlussfolgerung aus den Erkenntnissen beim Prototyping wie in Abschnitt 3.3.3 beschrieben.

XML bietet sich deshalb an, weil die generelle Dokumentenstruktur dem Aufbau des semantischen Modells recht nahekommt. Es lassen sich sowohl sequentielle Abfolgen von Elementen als auch hierarchische Strukturen gut in XML abbilden.

Auch den individuellen Eigenschaften der Elemente wird in XML Rechnung getragen. Diese werden als Attribute in der XML-Dokumentenstruktur bei dem jeweiligen Elemente vermerkt.

4.5 Funktionale Kapselung von Elementgruppen

Ein weiterer wichtiger Aspekt bei der Verwendung von Skripten im visuellen Skripteditor ist das Zusammenfassen einzelner Abfolgen von Skriptelementen zu übergeordneten funktionalen Gruppen. Die Notwendigkeit hierfür liegt in mehreren Aspekten begründet:

- **Übersichtlichkeit des Diagrammes:** Durch das Zusammenfassen vieler einzelner Elemente zu einem übergeordneten Funktionsbaustein lässt sich die Anzahl der Elemente in dem übergeordneten Diagramm drastisch verringern. Dies führt auch zu einer Reduktion an dargestellten Kanten (im Flussdiagramm), was auch der Übersichtlichkeit zugute kommt. Die einzelnen Funktionsbausteine können dann in einem anderen Diagramm bearbeitet werden.

- **Wiederverwendbarkeit von Programmteilen:** Durch die Definition von Funktionsbausteinen kann auch Arbeitszeit bei der Erstellung von Skripten eingespart werden. Für gleich ablaufende Bestandteile von Skripten können die gleichen Funktionsbausteine verwendet werden – vor allem in unterschiedlichen Skripten.

- **Austauschbarkeit von Programmteilen:** Vor allem zu Testzwecken kann es notwendig sein, nur einzelne Teilbereiche eines Skriptes auszutauschen. Auch bei der Umstellung auf andere Testvorgänge kann dies helfen, da detailliertere Bereiche eines Skriptes ausgetauscht werden können ohne die gesamte Restlogik verändern zu müssen.

Prinzipiell können Funktionsbausteine auf zwei verschiedene Arten eingesetzt werden:

1. **Als Ausführungsblock:** Der erstellte Funktionsbaustein stellt eine (prinzipiell) sequentielle Abfolge von Anweisungen dar. Er dient nicht als Datenquelle für übergeordnete Skripte und besitzt einen definierten Start- und einen Endpunkt. Ausführungsblöcke besitzen zwar möglicherweise Aufrufparameter beziehungsweise Datenquellen, erzeugen jedoch keine Rückgabewerte und sind somit keine Datenquelle.

2. **Zur Datengewinnung:** Hier dient der erstellte Funktionsbaustein in erster Linie als Lieferant für Daten. Beispiel hierfür wäre ein Funktionsbaustein, der einen

Testknoten ausführt und im Anschluss daran eine Auswertung vornimmt ob eine bestimmte Fehlerart vorliegt. Die somit erzeugte Aussage (`WAHR/FALSCH`) kann von übergeordneten Skripten weiter zur dynamischen Ablaufsteuerung genutzt werden.

Diese Problematik betrifft in erster Linie den `ENDE`-Knoten des jeweiligen Skriptes. Ihm kann damit entweder eine Datenquelle zugeordnet werden – dann dient der gesamte Funktionsbaustein als Datenquelle und kann nicht auf oberster Skriptebene eingesetzt werden – oder aber der daraus resultierende Funktionsbaustein besitzt keinen Datenausgang.

Eine Austauschbarkeit der Funktionsbausteine könnte Probleme aufwerfen falls ein Unterknoten der *Lauts*-Teststruktur ausgeführt werden soll. Diese Unterknoten sind nicht für jedes *Lauts*-Element, dem der Baustein zugeordnet werden kann gleich. Somit muss validiert werden, ob der gewählte Unterknoten zur Verfügung steht. Es ist jedoch höchst unwahrscheinlich, dass zwei *Lauts*-Elemente, denen der Skriptbaustein zugeordnet ist die gleiche Unterstruktur besitzen. Somit müssen in Funktionsbausteinen *Lauts*-Elemente die verwendet werden sollen und nicht genau dem Element entsprechen, dem das Skript zugeordnet ist, unbedingt als Aufrufparameter übergeben werden. Der Aufruf der Bausteine erfolgt dann individuell in einem übergeordneten Skript pro Testknoten, das das gesuchte Unterelement aus der Baumstruktur ausliest und als Parameter übergibt.

4.6 Persistenz und Reproduzierbarkeit

4.6.1 Datenbank

Der vom Editor generierte Quellcode des Skriptes wird in der Datenbank abgelegt. Dies erfolgt unter Verwendung von *Hibernate*, einer objekt-relationalen Brücke. Diese findet auch bei den übrigen Datenstrukturen aus *Lauts* Anwendung und ermöglicht es, direkt einzelne Datenobjekte in der Datenbank abzulegen und wieder abzufragen. Detailliertere Informationen zu Hibernate finden sich in (Bau05).

Die zugrundeliegende Datenbank selbst wird mit *MySQL*[1] realisiert.

[1] http://www.mysql.org/

In ihr wird sowohl der Quelltext des erstellten Skriptes abgelegt, als auch seine graphische Repräsentation. Auch alle anderen Informationen, die zur Benutzung des Skriptes notwendig sind (Name, Beschreibung, Übergabeparameter, ID usw.) sind enthalten. Somit ist eine konsistente Schnittstelle für die Skripte – textuell wie visuell – gewährleistet.

Skripte die auch ausgeführt werden sollen sind mit entsprechenden Knoten der *Laus*-Teststruktur assoziiert. Trifft die Testengine bei der Ausführung auf einen solchen Knoten, wird das vermerkte Skript automatisch angestoßen. Die Ausführung seitens der Testengine bezieht lediglich den eigentlichen Quellcode des Skriptes aus der Datenbank. Für sie sind zusätzliche Informationen, wie sie beispielsweise zur reproduzierbaren Positionierung der einzelnen Elemente im visuellen Skripteditor notwendig sind nicht relevant. Diese Informationen werden getrennt von dem eigentlichen Quellcode in der Datenbank abgelegt, so dass beim Laden eines Skriptes im Editor aus der Datenbank die gleiche Darstellung wie beim letzten Speichern gewährleistet ist. Sollten keine erweiterten Informationen für die gewählte Diagrammart zur Verfügung stehen – weil beispielsweise ein anderes Diagramm zur Erstellung des Skriptes benutzt wurde – wird eine automatische Darstellung des Skriptes aktiv. Dies bedeutet im Falle des Flussdiagrammes eine automatische Positionierung der Elemente basierend auf den Einschränkungen zur Positionierung (vgl. Abschnitt 4.2.1).

4.6.2 Export

Neben der Speicherung des Skriptes in die Datenbank besteht noch die Möglichkeit, dieses als Datei zu exportieren. Hierbei bestehen grundsätzlich verschiedene Möglichkeiten.

Gesamtes Skriptmodell

Bei der Speicherung des gesamten Skriptmodells wird eine Datei erzeugt, die alle Informationen zum gerade bearbeiteten Skript enthält. Sie stellt ein Bytecode-Abbild des spezifischen, diagrammabhägigen Modells dar. Dies bedeutet, dass beispielsweise alle Positionierungsinformationen erhalten bleiben. In der Folge hat das jedoch zur Konsequenz,

dass das Modell mit dem selben Diagrammtyp zur Betrachtung oder weiteren Bearbeitung geöffnet werden muss. Eine spätere Umschaltung zum jeweils anderen Diagrammtyp ist jedoch möglich.

Diese Exportierungsart hat allerdings den Vorteil, dass die Ansicht des Skriptes nach dem Importieren genau der Darstellung beim Exportieren entspricht. Somit wird eine Desorientierung des Benutzers durch Neuanordnung der einzelnen Elemente vermieden.

XML

Die Speicherung des Skriptes im XML-Format stellt die nächsthöhere Abstraktionsschicht bei der Exportierung dar. Sie erlaubt es, auch mit externen Mitteln eine Veränderung am Skript durchzuführen. Erweiterte Informationen zur Darstellung der einzelnen Elemente gehen hierbei jedoch verloren, da nur das abstrakte Modell des eigentlichen Skriptablaufs gespeichert wird.

Diese Export- und Importfunktion ist komplett unabhängig von der gerade gewählten Diagrammart. Durch die Verwendung von XML ist es auch beispielsweise möglich, die erzeugten Skripte einer externen Versionskontrolle (wie *CVS* oder *SVN*) zu unterwerfen, die unabhängig von *Lauts*-internen Mechanismen arbeitet.

Quelltext

Der Export eines Skriptes als Quelltext ermöglicht es, dieses nachträglich auf Fehler hin zu untersuchen. Dies kann manuell oder auch durch entsprechende Interpreterbenutzung geschehen. Hierbei entdeckte Fehler müssen jedoch als Schwachstellen des Übersetzungsprozesses gewertet werden. Somit dient der Export eines Skriptes in der Zielsprache in erster Linie zur Verbesserung der verwendeten Übersetzungsmechanismen. Durch die Verwendung von XSLT in der Übersetzungsschicht lassen sich Fehler in einzelnen Übersetzungsvorlagen ohne Veränderung des eigentlichen Skripteditors relativ leicht beheben.

Bei dem Export des Skriptes in Quelltext ist zudem die gewünschte Zielsprache auszuwählen. Mit dieser sollte der Anwender bei der Fehlersuche vertraut sein.

Da dieses Vorgehen jedoch direkt in den Übersetzungsprozess des Skripteditors eingreift sollte diese Funktionalität nur erfahrenen Benutzern vorbehalten sein.

4.7 Code-Generierung

Die eigentliche Generierung des Quelltexts in der gewünschten Zielsprache stellt die zentrale Aufgabe des visuellen Skripteditors dar. Deswegen ist bei der Implementierung dieser Funktionalität mit besonderer Vorsicht zu walten, da sich Fehler an dieser Stelle die Produktivität des gesamten Scripting-Systems stark beeinträchtigen.

4.7.1 Verarbeitungsschritte beim Erstellen eines Skriptes

Um den zentralen Stellenwert der Übersetzung hervorzuheben soll an dieser Stelle noch einmal ein kurzer Überblick über die Schritte gegeben werden, die bei der Erstellung und Benutzung eines visuellen Skriptes durchlaufen werden:

1. **Visuelle Erstellung/Modifikation:** Dieser erste Schritt beschreibt die eigentliche Modellierung des Skriptes mit Hilfe des visuellen Skripteditors. Hier übt der Anwender seinen Einfluss auf den Ablauf des Skriptes voll aus.

2. **Validierung:** Wurde ein entsprechendes Modell vom Benutzer entworfen erfolgt im Anschluss daran eine Überprüfung des erstellten Skriptes. Diese findet sowohl während der Bearbeitung des Skriptes statt (unter anderem beim Verbinden unterschiedlicher Skriptelemente) als auch in einem finalen Schritt bei der Speicherung des Skriptes. Dies ermöglicht es, Schwachstellen und Unklarheiten in der Semantik des Skriptes auszumachen. An diesem Punkt ist es immer noch möglich, dass Rückfragen an den Benutzer gestellt werden und Warnungen bzw. Fehlermeldungen generiert werden die der Fehlerdiagnose im Skript dienen.

3. **Generierung des Metacodes:** Ist das Skript semantisch überprüft worden kann mit der eigentlichen Übersetzung begonnen werden. Der erste Schritt hierbei ist die vollständige Loslösung der Semantik von der Darstellung im Diagramm – es wird ein Metacode generiert. Dieser basiert auf XML und ist noch vollkommen unabhängig (mit Ausnahme von generischen Anweisungsblöcken) von der gewählten Zielsprache.

4. **Übersetzung des Metacodes in die Zielsprache:** Dieser Schritt stellt die eigentliche Übersetzung dar. Unter Verwendung von Vorlagen für einzelne Elemente wird die abstrakte Repräsentation der Skriptsemantik in den konkreten Quelltext der gewählten Zielsprache abgebildet. Dieser Schritt ist kritisch für den späteren Erfolg der Ausführung und muss fehlerfrei ablaufen.

5. **Speichern des Zielcodes (und des Modells) in der Datenbank:** Nach erfolgter Übersetzung des Skriptes in die Zielsprache ist es notwendig, den generierten Quelltext zur späteren Ausführung in der Datenbank zu hinterlegen. Dies ist darin begründet, dass *Lauts* prinzipiell ein verteiltes System darstellt. Die Konzeption und Modellierung der Skripte erfolgt auf Seiten des *Clients*, die Ausführung hingegen findet auf der *Testengine* statt. Die zentrale Schnittstelle für den Austausch der Daten zwischen diesen beiden Komponenten bildet die Datenbank. Zudem ermöglicht das Ablegen des Skriptes in der Datenbank die spätere Weiterbearbeitung und Verbesserung.

6. **Ausführen des Zielcodes auf der TestEngine:** Dieser Schritt ist das Endresultat aller vorausgegangenen Schritte. Die Testengine führt das generierte Skript wie ein reguläres, mittels textuellem Editor geschriebenes Skript aus. Ausgaben und Fehlermeldungen werden protokolliert und zurück an den Client gesendet, falls dieser verbunden ist. Nach der Abarbeitung des Skriptes, das einem bestimmten Knoten in der *Lauts*-Teststruktur zugeordnet ist, wird nach den weiteren Kriterien der Teststruktur fortgefahren.

4.7.2 Validierung eines Skriptes

Die Überprüfung der Skripte erfolgt an mehreren verschiedenen Stellen:

1. **Bei der Bearbeitung des Skriptes:** Schon bei der Erstellung und Bearbeitung des Skriptes werden Prüfungen durchgeführt. Diese Prüfungen erstrecken sich sowohl auf semantische Probleme – beispielsweise die Erkennung zyklischer Abhängigkeiten – als auch auf Kompatibilitäten einzelner Elemente bei der Verbindung. So werden beim Verbinden von Elementen die jeweiligen Datentypen miteinander verglichen. Dies ermöglicht es, sicherzustellen dass die Zielelemente den Datentyp der Quellelemente als Eingabe akzeptieren.

2. **Bei der Übersetzung des Metacodes in die Zielsprache:** Eine weitere Überprüfung der Skripte erfolgt beim Übersetzen des abstrakten semantischen Modells in die eigentliche Zielsprache des Skriptes. Insbesondere Elemente, die keine Darstellung in der Zielsprache besitzen – beispielsweise durch fehlende Vorlagen oder weil die Skriptsprache ein bestimmtes Konstrukt einfach nicht darstellen kann – werden hier entdeckt.

4.7.3 Speichern eines Skriptes

Die Speicherung der Skripte erfolgt wie in Abschnitt 4.6 beschrieben. Zur späteren Ausführung wird nur die Speicherung in die Datenbank berücksichtigt, da dies der einzige Punkt ist, über den der Client und die Testengine Daten austauschen.

4.7.4 Laden eines Skriptes

Das Laden eines existierenden Skriptes im visuellen Skripteditor kann prinzipiell auf drei verschiedene Arten erfolgen, unabhängig von der Datenquelle (einer Datei oder der Datenbank):

1. **Import eines existierenden Skriptes in Quelltextform:** Das Importieren eines bestehenden Skriptes in textueller Form ist problematisch. Es können beispielsweise Sprachkonstrukte auftreten, die vom visuellen Skripteditor nicht unterstützt werden, was dann Sprachenabhängigkeit einzelner Skriptfragmente bewirkt. Solche Fragmente würden nur als generische Codeabschnitte in den Editor übernommen werden können. Zugunsten des weiteren Ausbaus der Basisfunktionalität (Erstellen eines ausführbaren Skriptes) des visuellen Editors wird diese Funktionalität jedoch nicht bereitgestellt. An dieser Stelle sei auf den Abschnitt 6.1 des Ausblicks verwiesen, in dem diese Problematik noch einmal aufgegriffen wird.

2. **Laden von Metacode:** Der nächsteinfachere Fall ist das Importieren einer abstrakten Repräsentation eines Skriptes in den Skripteditor. Diese besitzt jedoch auch keinerlei erweiterte Informationen wie beispielsweise über die Positionierung einzelner Elemente im Diagramm. Es ist jedoch gewährleistet, dass alle dort vorkommenden Konstrukte auch vom visuellen Skripteditor unterstützt werden, so dass diese Importierung relativ einfach vorgenommen werden kann. Einzig eine automatische Positionierung beziehungsweise die automatische Erstellung von diagrammspezifischen Zusatzinformationen muss noch vorgenommen werden um eine übersichtliche Darstellung zu gewährleisten.

3. **Laden eines Modells:** Beim Laden eines kompletten Skriptmodells sind Zusatzinformationen – wie beispielsweise Angaben zur Positionierung der einzelnen Elemente – bereits enthalten. Die entsprechende Datenquelle kann entweder die Datenbank sein (unter Verwendung von Hibernate) oder eine Datei, die das gesamte Modell in Bytecodedarstellung enthält.

4.7.5 Transformation eines Skriptes in die Zielsprache

Wie die Erkenntnisse aus dem Prototyping (vgl. Abschnitt 3.4) zeigen, ist es sinnvoll zur vollständigen Umsetzung eines Skriptes in die Zielsprache einen Zwischenschritt über eine XML-basierte Metasprache zu machen. Diese Skriptschablone wird dann mit Hilfe von

XSLT in das fertige Skript umgewandelt. Hierzu sind für die einzelnen Elemente Formatierungskonstrukte bzw. Vorlagen hinterlegt, die dann mit den Attributen der einzelnen konkreten Elemente befüllt werden.

Somit wird der erzeugte Metacode des Skriptes zur Quellcodegenerierung benutzt. Eine direkte Umsetzung der internen, modellhaften Darstellung des gerade bearbeiteten Skriptes erfolgt nicht. Dies bringt zwar weitere Zeitkosten für die Übersetzung mit sich, stellt jedoch einen weitaus generischeren Ansatz dar. Die Erweiterbarkeit der Übersetzung - hinsichtlich der verwendbaren Sprachen wie auch der verwendbaren Skriptelemente profitiert hiervon.

4.7.6 Ausführung eines Skriptes

Die Ausführung eines mit Hilfe des visuellen Skripteditors erstellten Skriptes für das *Lauts*-System erfolgt genau wie die Ausführung eines regulären textuellen Skriptes. Hierzu wird neben dem Modell des Skriptes – das zur Bearbeitung im Editor verwendet wird – auch der davon erzeugte Quelltext in der Datenbank abgelegt. Die TestEngine erkennt das Vorhandensein eines Skriptes für ein bestimmtes Element der Teststruktur und startet automatisch die Ausführung wie in (Kli07) beschrieben mit Hilfe des entsprechenden Interpreters für die gewählte Skriptsprache.

Zur Fehlererkennung bei der Erstelllung und Bearbeitung eines Skriptes wäre es sinnvoll, einen Simulator zur Verfügung zu stellen, der sich ähnlich verhält wie das Scripting-Modul der Testengine. Da dies jedoch den Rahmen dieser Arbeit sprengen würde muss an dieser Stelle auf Abschnitt 6.2 des Ausblicks verwiesen werden.

5 Konstruktion

5.1 Struktur

5.1.1 Packages

Der visuelle Skripteditor ist in verschiedene *Packages* unterteilt, die den einzelne Funktionen der Software entsprechen. Dies dient der Strukturierung des Gesamtprojekts und der Wartbarkeit im Falle von Erweiterungen. Abbildung 5.1 stellt diese Struktur in *UML* dar.

Die einzelnen Packages erfüllen im einzelnen folgende Zwecke:

- *persistence:* Dieses Package enthält alle Klassen, die zur Abspeicherung des Skriptes notwendig sind. Hierzu gehört sowohl die Speicherung des Skriptes in die Datenbank unter Verwendung von *Hibernate*[1] als auch das Exportieren eines Skriptes in der abstrakten XML-Darstellung.

- *actions:* Dieses Package dient der graphischen Benutzeroberfläche zur Ansteuerung bestimmter Funktionen von anderen Komponenten. Hier sind Klassen hinterlegt, die einzelne Aktionen der graphischen Benutzeroberfläche (*Laden*, *Speichern*, *Rückgängig* usw.) auslösen. Zusätzlich enthält dieses Package alle Klassen, die zur Abwicklung der *Drag&Drop*-Funktionalitäten notwendig sind. Hierzu gehören die verwendeten *TransferHandler* sowie ein Wrapper-Objekt, das ein Skriptelement repräsentiert.

[1] http://www.hibernate.org

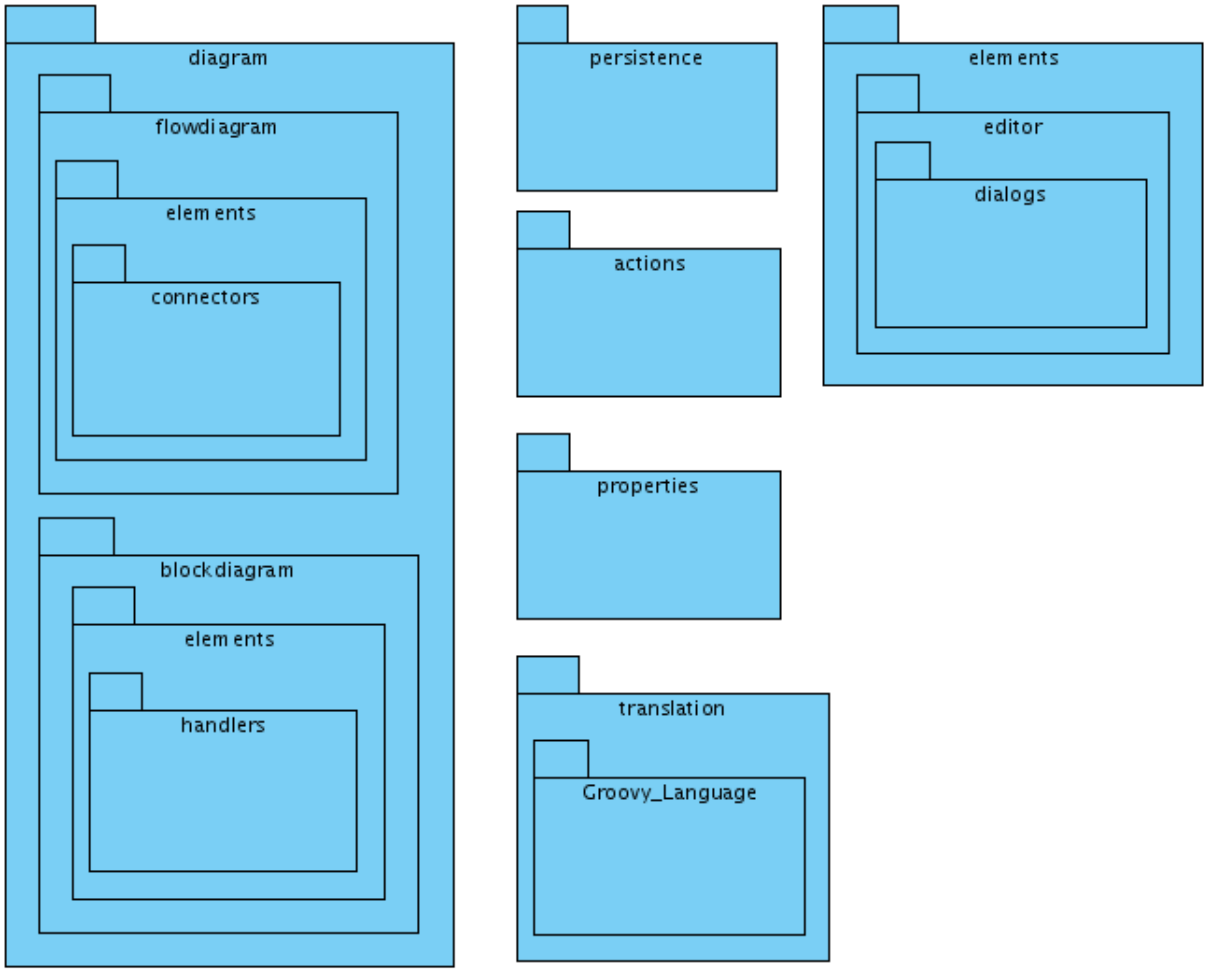


Abbildung 5.1: Package-Struktur des visuellen Skripteditors

- *properties:* Dieses Package ist für die Einstellungen des visuellen Skripteditors zuständig. Hierzu gehört die Konfiguration von vordefinierten Funktionalitäten (z.B. automatische Positionierung beim Flussdiagramm), Farbeinstellungen oder auch die Lokalisierung von Beschreibungen, Meldungen und Beschriftungen.

- *translation:* Der Hauptzweck dieses Packages ist die eigentliche Übersetzung eines erzeugten Skriptes in die gewählte Zielsprache. Hierzu enthält das Paket alle benötigten Klassen und fungiert als Schnittstelle zu möglicherweise vorhandenen Bibliotheken. Ein Beispiel hierfür wäre die Verwendung von *Xalan-Java*[2] zur Umsetzung der Elementvorlagen mittels *XSLT*.

- *elements:* In diesem Package sind alle möglichen Elemente als Klassen hinterlegt, die in einem Skript vorkommen können. Sie beziehen sich ausschließlich auf die semantische Repräsentation. Zur Darstellung der einzelnen Elemente befinden sich weitere Klassen in dem entsprechenden Unterpackage des jeweiligen Diagramms.

- *diagram:* Dieses Paket enthält die beiden verfügbaren Diagrammtypen und alle ihre benötigten Bestandteile. Hierzu gehören jeweils immer visuelle Darstellungen der einzelnen Skriptelemente sowie diagrammspezifische Besonderheiten. Dies können beispielsweise im Falle des Flussdiagramms die Repräsentationen der Verbindungslinien sein.

5.1.2 Ablauf

1. **Starten des Editors:** Prinzipiell bestehen beim Start des Editors zwei Möglichkeiten: Entweder es soll ein bereits bestehendes Skript geöffnet werden (das mit dem aktuellen Knoten der *Lauts*-Teststruktur verknüpft ist), oder es wird ein neues Skript erstellt. Im ersten Fall wird dieses Skript dann von der Datenbank geladen und dargestellt, im zweiten Fall erfolgt die Anzeige eines leeren Diagramms, das nur den Startknoten enthält. Weitere Elemente werden dann vom Benutzer hinzugefügt.

[2] http://xml.apache.org/xalan-j/

2. **Laden/Importieren eines Skriptes:** Ist ein Skript mit dem entsprechenden Testknoten assoziiert, wird dieses von der Datenbank geladen. Alternativ kann auch ein in Dateiform bestehendes Skript importiert werden. Dies kann sowohl ein in abstrakter Form als XML-Datei bestehendes Skript sein als auch eine Bytecodedarstellung eines Skriptes, das zusätzlich Positionierungsinformationen zu den einzelnen Elementen für das Flussdiagramm enthält. Für das Blockdiagramm sind solche Zusatzinformationen nicht notwendig, da die Anordnung der einzelnen Elemente immer automatisch erfolgt. Als Datenquellen für das Laden kommt sowohl die Datenbank in Frage als auch einzelne Dateien.

3. **Bearbeiten des Skriptes:** Die Bearbeitung des Skriptes erfolgt mit den Werkzeugen, die die Benutzeroberfläche zur Verfügung stellt. Hinzu kommen Spezialfunktionen des jeweils gewählten Diagrammtyps. Hierzu gehört beispielsweise eine automatische Positionierung der Elemente, was beim Importieren eines Skriptes aus der semantischen Darstellung in das Flussdiagramm besonders hilfreich ist.

4. **Export des Skriptes:** In manchen Fällen kann es sinnvoll sein, das erstellte Skript nicht nur in der Datenbank abzulegen sondern es im Dateisystem abzulegen. Für diese Exportfunktion bieten sich wieder mehrere Möglichkeiten an – es kann eine abstrakte Repräsentation des Skriptes in einem *XML*-Dokument oder das Diagramm mit allen Zusatzinformationen in einer Bytecodedarstellung gespeichert werden. Auch ein Export des erzeugten Quelltexts in der Zielsprache ist möglich.

5. **Übersetzung des Skriptes:** Um eine Ausführung des Skriptes auf der Testengine überhaupt erst ermöglichen zu können ist es notwendig, dieses in den eigentlichen Quelltext der Zielsprache umzuwandeln. Dieser wichtigste Schritt des gesamten Systems wird in Abschnitt 5.7 näher erleutert. Der erzeugte Code wird zusammen mit der abstrakteren Darstellung des Diagramms in der Datenbank abgelegt und bei jeder Veränderung des Skriptes im visuellen Editor beim Speichern neu erstellt.

6. **Speichern des Skriptes in der Datenbank:** Wie bereits erwähnt wird im Anschluss das Skript sowohl als Quelltext für die Testengine als auch als Diagrammdarstellung für eine spätere Weiterbearbeitung abgelegt. Somit *kann* ein Skript das einem *Lauts*-Testknoten zugeordnet ist eine visuelle Repräsentation besitzen (dann

wurde es als Diagramm entworfen), muss eine solche aber nicht aufweisen (dann wurde es mit dem textuellen Skripteditor erstellt).

7. **Ausführung des Skriptes seitens der Testengine:** Der abschließende Schritt im Lebenszyklus eines Skriptes ist die Ausführung auf der Testengine. Dort wird beim Durchlaufen der regulären Teststruktur die Präsenz eines Skriptes festgestellt und dessen Verarbeitung angestoßen. Notwendig hierfür ist das Vorhandensein eines Moduls für die entsprechende Skriptsprache. Innerhalb des Skriptes wiederum können interne Mechanismen der Testengine angestoßen werden wie beispielsweise die protokollierten Ausgabemeldungen an den Client oder auch das Ausführen eines Funktionsbausteins (eines weiteren Skriptes).

5.2 Darstellung der Semantik

Neben der eigentlichen Erzeugung des Quelltextes ist es die Hauptaufgabe des visuellen Skripteditors, die aktuell bearbeiteten Skripte korrekt und vollständig darzustellen. Hierzu ist es notwendig, neben einer internen Darstellung für Skripte allgemein auch zwei visuelle Darstellungsmethoden für die einzelnen Elemente sowie deren Beziehungen untereinander zu finden.

5.2.1 Interne Darstellung

Hinsichtlich der internen Darstellung muss zunächst eine Unterscheidung zwischen dem Skriptmodell – repräsentiert durch die Klasse `CVisualScript` – und dem einzelnen Skriptelement – Klasse `AScriptElement` – getroffen werden.

Das Skript an sich

Das Modell gibt in erster Linie Aufschluss über die Gesamtheit der Skriptelemente. Diese sind in einem `Vector` hinterlegt. Dieser dient in erster Linie dazu, ein Skriptmodell mit

einem anderen (beispielsweise beim Diagrammwechsel) zu synchronisieren.

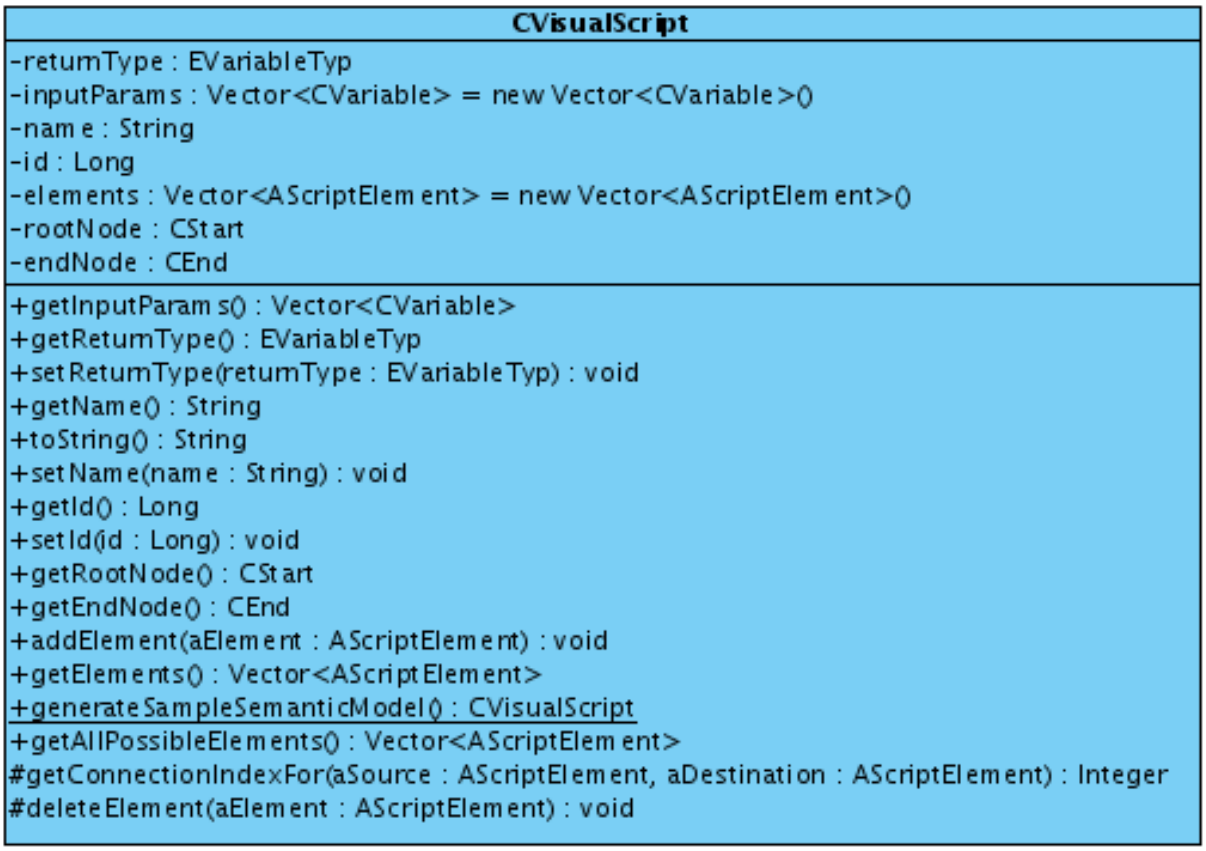


Abbildung 5.2: Die Klasse *CVisualScript*

Weiterhin enthält es alle Informationen, die benötigt werden um das Skript in die Anwendung zu integrieren. Dies beinhaltet den Namen des Skriptes sowie eine eineindeutige Identifikationsnummer zur Verwendung mit Hibernate. Auch externe Abhängigkeiten wie die Übergabeparameter, die vom Skript erwartet werden und der Rückgabewert des Skriptes werden hier verwaltet. Für die interne Darstellung sind weiterhin Referenzen auf den Wurzelknoten des Skriptes und sein Ende hinterlegt.

Zusätzlich muss bei der Betrachtung eines Skriptes den verwendeten Variablen besondere Aufmerksamkeit gewidmet werden. Deshalb existiert in der Klasse `CVisualScript` eine Liste der verwendeten Variablen – interner wie externer. Diese ermöglicht bei der Verwendung des Editors eine schnellere Auswahl, was durch die Zeitersparnis der Benutzerfreundlichkeit zugute kommt. Auch die Deklaration interner Variablen kann dadurch automatisch bei der Codegenerierung erfolgen.

Das einzelne Skriptelement

Die interne Repräsentation eines einzelnen Skriptelements hingegen ist für die Vernetzung der einzelnen Skripte zuständig. Hierfür enthält sie Referenzen auf alle Nachbarelemente. Kontrollflusstechnisch bedeutet das eine Referenz für den Vorgänger, mehrere für die möglichen Nachfolger, datenflusstechnisch hingegen werden mehrere mögliche Referenzen für Datenquellen und eine Referenz für das Datenziel hinterlegt. Auch eine Referenz auf einen möglichen Kommentar, der jedem anderen Skriptelement zugeordnet werden kann, ist enthalten. Für die Verbindungsarten, für die mehrere Nachbarobjekte denkbar sind (mehrere Datenquellen oder mehrere mögliche Kontrollflussziele) sind zusätzliche Maximalwerte, sowie die Anzahl der tatsächlich belegten Verbindungsmöglichkeiten hinterlegt, die zur Überprüfung dienen ob eine zusätzliche Verbindung zu diesem Element möglich ist.

Für die Verwaltung von Datenflussverbindungen ist weiterhin eine Eigenschaft hinterlegt, die den Datentyp des einzelnen Elementes beschreibt. Ist ein Element keine Datenquelle, so ist dieser Typ `VOID`, andernfalls ist er einer der folgenden Werte: `DOUBLE`, `LONG`, `STRING`, `BOOLEAN`, `RESULT` oder `OBJECT`. Diese Datentypen entsprechen allgemein den Datentypen von Variablen, auch in der *Lauts*-Teststruktur. Es sollte eine absolute Ausnahme sein, ein `OBJECT` im Datenfluss zu verwenden, da sich dieser Datentyp nicht weiter verarbeiten lässt ohne zu wissen, was genau für ein Objekttyp dahintersteht.

Zusätzlich zu diesen allgemeinen Verbindungseigenschaften besteht in der Klasse eine Liste von `Observer`-Objekten, die bei einer Veränderung des individuellen Skriptelementes benachrichtigt werden. Dies kann beim Bilden einer neuen Verbindung mit dem Element oder aber auch nach dem Ändern der individuellen Eigenschaften des Elements geschehen. Getreu dem `Observer`-Pattern wird dann allen registrierten Objekten (in erster Linie visuelle Repräsentationen des Elements) mitgeteilt, dass sie die Darstellung des Elements aktualisieren müssen.

In der Klasse `AScriptElement` sind zusätzlich noch Informationen hinterlegt, die angeben, auf welche Art das Element verwendet werden kann. Diese sind als bool'sche Flags dargestellt und sagen im einzelnen aus, ob das Element möglicherweise Datenquellen besitzt,

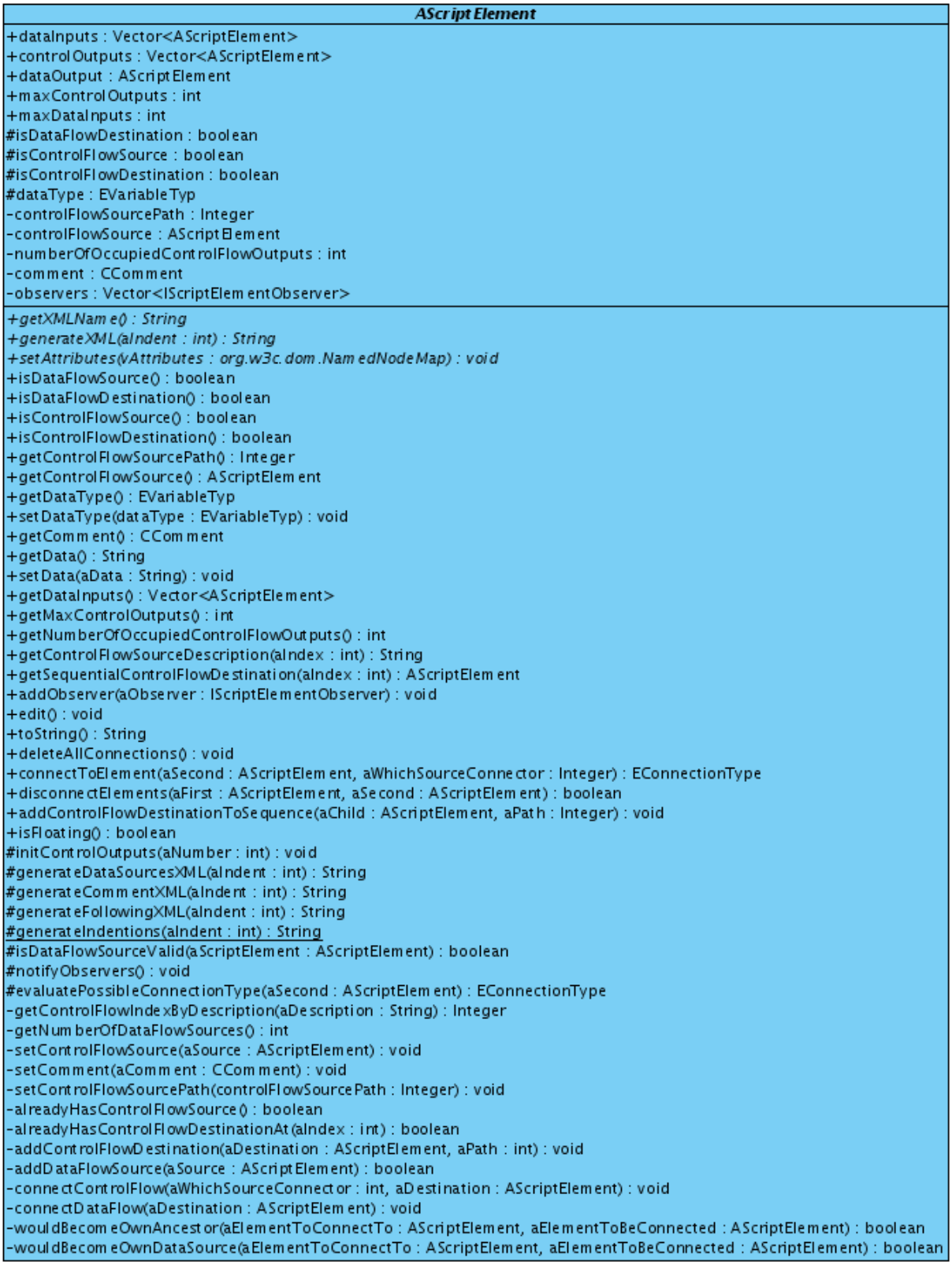


Abbildung 5.3: Die Klasse *AScriptElement*

als Ziel des Kontrollflusses verwendet werden kann und/oder ob es Kontrollflussnachfolger haben kann. Die Frage, ob ein Element als Datenquelle dient wird durch den Datentyp geregelt – ist dieser `VOID`, so werden keinerlei Daten an andere Elemente geliefert.

Weiterhin bietet diese Klasse wie in Abbildung 5.3 dargestellt eine Reihe von Methoden an, die von anderen Programmteilen verwendet werden können. Diese sind im Einzelnen:

- **Methoden zur Flusssteuerung:** Methoden wie `isDataFlowSource()` oder `isControlFlowDestination()` dienen dazu, die Flusseigenschaften des jeweiligen konkreten Elements zu bestimmen. Die hierfür benötigten Flags bzw. der Rückgabetyp werden im Konstruktor der jeweiligen Elementklasse festgelegt. Diese Methoden werden insbesondere beim Verbinden zweier Elemente aufgerufen und können von konkreten Elementklassen überladen werden. Dies dient dazu, um eine gewisse Dynamik in die Flusskontrolle einzubringen. So ist es beispielsweise möglich, dass Variablen je nach Verwendung entweder als Datenquelle dienen (lesender Zugriff), oder aber in den Kontrollfluss mit einer Datenquelle eingebunden werden (schreibender Zugriff).

- **Getter und Setter:** Die Getter und Setter der Klasse dienen dazu, fundamentale Eigenschaften des jeweiligen Skriptelements zu beeinflussen bzw. auszuwerten. Zu diesen Eigenschaften zählen insbesondere der Datentyp des Elements, der ihm zugeordnete Kommentar, sowie die zugeordneten Datenquellen. Eine Besonderheit hierbei ist die Methode `getData()`, die bei Datenquellen eine Zeichenkette zurückliefert, welche die Daten der Quelle repräsentiert. Dies wird zur anschaulicheren Darstellung von Elementen verwendet, die Datenquellen benutzen. Auch zur Steuerung der ausgehenden Kontrollflussverbindungen existieren besondere Methoden, die dem Benutzer zur Entscheidung dienen, wie zwei Elemente verbunden werden sollen wenn mehrere Verbindungen möglich sind. So wird der Benutzer beim Verbinden eines Elements mit einer Schleife gefragt, ob das nachfolgende Element innerhalb oder außerhalb der Schleife nachfolgt.

- **Observer-Steuerung:** Zur Synchronisierung der Diagrammdarstellung mit den Gegebenheiten des jeweiligen Elements existieren weiterhin Methoden zur Regis-

trierung eines Beobachters beziehungsweise dessen Entfernen. Die registrierten Beobachter werden dann bei Veränderung des Elements benachrichtigt um ihre Darstellung entsprechend anpassen zu können.

- **Edit-Methode:** Für jedes Skriptelement existiert weiterhin eine `edit()`-Methode. Diese bringt mit Hilfe einer statischen Methode der Klasse `CElementEditorFactory` ein Dialogfeld hervor, in welchem die individuellen Eigenschaften des Skriptelements bearbeitet werden können. Im Konkreten können somit beispielsweise der Wert und der Datentyp einer Konstanten verändert werden.

- **Verbindungssteuerung:** Für alle Elemente müssen weiterhin Methoden zur Verfügung stehen, die Verbindungen zu anderen Elementen verwalten. Hierzu gehören Methoden um zwei Elemente miteinander zu verbinden wie `connectToElement()`, welche eine Verbindung zu einem anderen Element herstellt oder eine Methode zum Anhängen eines Elementes an einen bestehenden Kontrollflusspfad – `addControlFlowDestinationToSequence()`. Dementsprechend existieren natürlich auch Methoden zum Trennen der Verbindung zweier Elemente (`disconnectElements()`) beziehungsweise zum kompletten Auslösen eines Elementes aus der Skriptstruktur (`deleteAllConnections()`).

- **Hilfsmethoden:** Um die Darstellung nichtverbundener Elemente übersichtlicher zu machen ist weiterhin eine Methode vorhanden, die zurückliefert, ob das Element „schwebt“.

Bei der Erstellung eines neuen, konkreten Skriptelementes gibt es einige Punkte, die besondere Beachtung verdienen. Hierbei ist besonders der Ablauf innerhalb des Konstruktors zu erwähnen, der bei der Instanziierung einer solchen Unterklassen von `AScriptElement` aufgerufen wird. So ist ein neu erstelltes Skriptelement immer eine Unterklasse von `AScriptElement`. Infolgedessen muss es einige Methoden implementieren, die von dieser abstrakten Klasse gefordert werden. Diese hängen in erster Linie mit der Metadarstellung des Elements in XML zusammen und sind wiefolgt:

- `getXMLName()`: Diese Methode liefert als Rückgabewert den Bezeichner des Elementes in der XML-Darstellung zurück. Dies dient zum einen dem dynamischen

Generieren des Metacodes, zum anderen hilft es dem Parser beim Lesen des entsprechenden Tags ein neues Objekt der jeweiligen Klasse zu instanziieren.

- `generateXML()`: Diese Methode ist dafür zuständig, eine korrekte Abbildung des Elements und aller seiner Eigenschaften in dem XML-Dokument zu erreichen. Dies schließt insbesonders auch verknüpfte Elemente ein, für die auch jeweils der XML-Text erstellt wird. Aus Performancegründen sollte an dieser Stelle ein dynamisches `StringBuilder`-Objekt bearbeitet werden statt mit `String`s an sich zu arbeiten, die ja bei jeder Veränderung neu erzeugt werden müssen.

- `setAttributes(NamedNodeMap aAttributes)`: Diese Methode dient dazu, einer neu erstellten Elementinstanz alle hinterlegten Eigenschaften zuzuweisen. Hierfür wird der Methode eine `NamedNodeMap` des *DOM*-Parsers übergeben, die dieser für das individuelle Element aus der XML-Datei gelesen hat. Diese Eigenschaften werden dann nach bestimmten Kriterien durchsucht, und falls sie vorhanden sind werden diese Eigenschaften des Elementes gesetzt.

An dieser Stelle wäre es möglicherweise sinnvoller *Betwixt*[3]– eine Java-Bibliothek zum Mapping von Java-Beans auf XML-Dokumente – zu verwenden. Im Prototyp erfolgte dies jedoch nicht und es wurde der oben erwähnte Ansatz zur Generierung des XML-Dokumentes gewählt.

Der nächste Schritt bei der Erstellung eines Skriptelementes ist dann im Anschluss die Anpassung des Konstruktors. Dies ist notwendig um die Eigenschaften des Elements klar zu definieren, welche dann bei seiner Verwendung im Skripteditor unter anderem für die Validierung der Verbindungen benutzt werden. Die notwendigen Schritte sind wie folgt:

1. **Konfiguration der Kontrollflussziele:** Es muss für jede einzelne Skriptelementklasse definiert werden, wie viele Kontrollflussausgänge sie besitzt. Im Falle von Schleifen sind das zwei (ein innerer und ein äußerer Ablaufpfad), bei regulären Anweisungen einer und bei Verzweigungen sind dies drei (`WAHR`, `FALSCH` und der allgemeine Fluss nach der Verzweigung). Dies wird durch Aufruf der privaten Methode

[3]http://jakarta.apache.org/commons/betwixt/

`initControlOutputs()` erledigt. Erfolgt dieser Aufruf nicht tritt das Standardverhalten in Kraft, das das Element einfach als Datenquelle ohne Kontrollflussausgang ansieht.

2. **Konfiguration der Datenquellen:** Soll ein Elementtyp Datenquellen benutzen, so muss dies auch im Konstruktor ermöglicht werden. Dies erfolgt durch Setzen der Eigenschaft `maxDataInputs` auf den entsprechenden Wert. Logische `UND`-Operatoren beispielsweise können beliebig viele Datenquellen miteinander verknüpfen, wohingegen Schleifen beispielsweise nur genau eine Bedingung als Datenquelle auswerten.

3. **Setzen der Flusseigenschaften des Elements:** Im Anschluss müssen für das Element die Flusseigenshaften gesetzt werden. Diese Konfiguration erfolgt in erster Linie über das Setzen der Flags die diese beschreiben – `isControlFlowDestination`, `isControlFlowSource` und `isDataFlowDestination`.

4. **Konfiguration des Datentyps des Elements:** Abschließend muss bei Datenquellen noch der Datentyp des Elements bestimmt werden. Dies erfolgt durch Setzen der Eigenschaft `dataType`. Von Haus aus werden Elemente so konfiguriert, dass sie keine Datenquellen darstellen, infolgedessen ist ihr Datentyp `VOID`.

Weiterhin erwähnenswert zur Instanziierung von Skriptelementen ist die Notwendigkeit eines als `public` deklarierten Konstruktors ohne Übergabeparameter. Dieser muss vorhanden sein, da beim Einfügen eines neuen Elementes in das Diagramm das von dem Label repräsentierte Element „geklont“ wird. Dieses Vorgehen lässt sich dank der dynamischen Instanzenverwaltung von Java recht einfach verwirklichen und bringt eine starke Dynamik in die Übersicht der einfügbaren Elemente. Diese Elemente werden durch graphische *Label*-Komponenten dargestellt. Bei diesen Komponenten wiederum wird ein `TransferHandler` registriert, der für die Bereitstellung der *Drag&Drop*-Funktionalität zuständig ist. Genau an dieser Stelle – in der Klasse `CScriptElementInsertionHandler` – wird das zu transportierende Objekt erstellt. Beim Klonen eines Elements – wie das beim Einfügen eines neuen Elementes der Fall ist – wird dann das oben erwähnte Verfahren angewendet. Das nachfolgende Listing gibt dafür beispielhaft den Quelltext an.

Listing 5.1: Klonen von Klasseninstanzen

```
1 if(createNewInstanceOfElements)
2   try {
3     // this always creates new instances of the same type
4     // as the element associated with the label
5   return new CScriptElementTransferable(vLabl.getElement().getClass()
        .newInstance());
6   } catch (InstantiationException e) {
7     e.printStackTrace();
8   } catch (IllegalAccessException e) {
9     e.printStackTrace();
10  }
11 else return new CScriptElementTransferable(vLabl.getElement());
```

5.2.2 Flussdiagramm

Ein Element des Flussdiagramms wird durch eine Unterklasse von `AFlowElement` repräsentiert. Diese Klasse enthält allgemeine Informationen, die für jeden Elementtyp gültig sind. Dies betrifft in erster Linie Informationen zur Positionierung des Elements im Diagramm. So sind dies Werte für die X- und Y-Koordinaten des Elements wie auch die allgemeinen Abmessungen *Breite* und *Höhe*. Zusätzlich besitzt ein Element des Flussdiagramms natürlich auch eine Referenz auf das von ihm dargestellte Skriptelement um weitere Informationen für die Darstellung abrufen zu können.

Für die Einschränkung der Positionierung einzelner Elemente sind weiterhin Referenzen auf die Elemente hinterlegt, die die Positionierung beeinflussen. Konkret bedeutet dies, dass für ein Element innerhalb einer IF-Anweisung als obere Grenze der Start der Verzweigung hinterlegt wird, als untere Grenze hingegen das Ende der Verzweigung. In horizontaler Richtung ist die Einschränkung abhängig vom gewählten Pfad, in dem sich das Element befindet. Elemente im `WAHR`-Pfad müssen links angeordnet werden, während Elemente im `FALSCH`-Pfad rechts angeordnet werden müssen.

Die Klasse `AFlowElement` stellt an seine Unterklassen als Anforderung lediglich die Implementierung einer einzigen Methode: `void drawShape(Graphics2D aG)`. Diese dient dazu, das Element zu zeichnen und muss für jedes Element des Flussdiagramms implementiert werden.

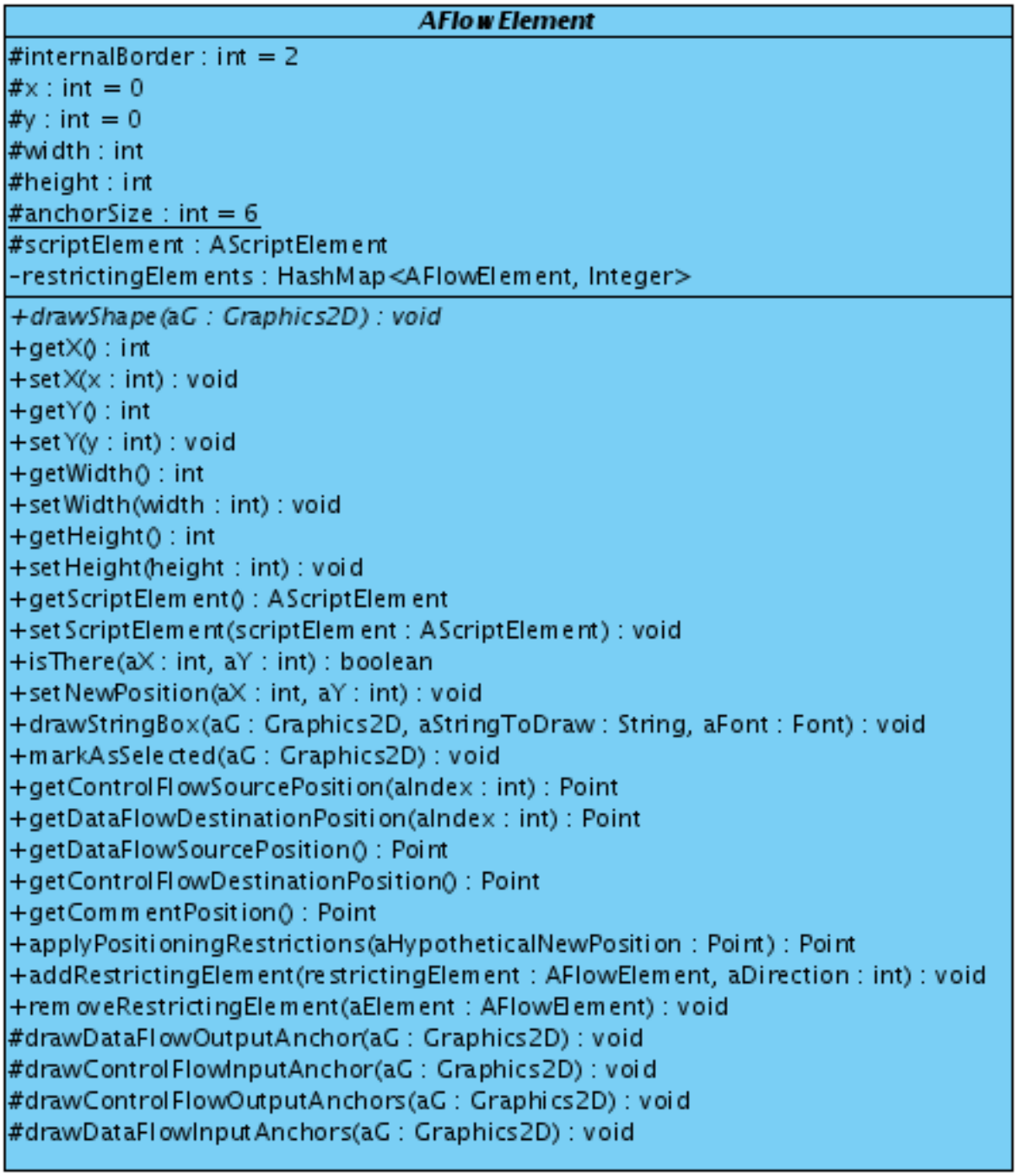


Abbildung 5.4: Die Klasse *AFlowElement*

Als Hilfestellung stellt die Klasse `AFlowElement` einige Methoden bereit, die es den Unterklassen recht schnell ermöglichen, einfache Repräsentationen von Elementen zu generieren und andere allgemeine Funktionen auszuführen:

- `drawStringBox()`: Diese Methode dient dazu, einfach eine Repräsentation des Elementes als Rechteck zu erhalten. Hierbei wird der Methode einfach eine String-Repräsentation des Elements übergeben. Zeilenumbrüche und Höhen-/Breitenberechnungen werden automatisch behandelt. Das Ergebnis ist dann ein Rechteck, in dem die übergebene Zeichenkette (in der übergebenen Schriftart) dargestellt wird, und das über alle Verbindungsanker verfügt, die im zugeordneten Skriptelement möglich sind.

- `applyPositioningRestrictions()`: Diese Methode wird bei entsprechender Konfiguration beim Verschieben eines Elementes aufgerufen. Sie überprüft anhand der beschränkenden Elemente, inwieweit die übergebene hypothetische Position übernommen werden kann. Treten Begrenzungen in Kraft, so liefert die Methode den grenzwertigen Punkt zurück, der die äußerste Näherung an die gewünschte Position darstellt.

- **Methoden zum Zeichnen der Verbindungsanker:** Diese Methoden – wie `drawControlFlowOutputAnchors()` – können von Unterklassen aufgerufen werden um an den entsprechend definierten Stellen die Verbindungsanker zu zeichnen. Bei komplexeren Elementen – wie der Selektion, die aus zwei Elementen aufgebaut ist – ist es möglich, die Methode `getControlFlowSourcePosition(int aIndex)` zu überladen um eine spezifischere Positionierung der Anschlussstellen zu erreichen.

Die Gesamtheit der Elemente des Flussdiagramms wird in der Klasse `CFlowModel` hinterlegt. Sie verwaltet zudem noch die Verbindungslinien zwischen den einzelnen Elementen von denen es drei verschiedene Typen gibt. Diese sind momentan als recht einfache direkte Pfeile zwischen zwei Punkten realisiert, es ist jedoch denkbar sie mit mehr Intelligenz zur Vermeidung von Kreuzungen im Graphen auszustatten. Dies würde die Lesbarkeit des erstellten Diagrammes erheblich verbessern. Gerade in diesem Bereich sind graphentheoretische Grundlagen (vgl. (Kön07)) anwendbar.

Die Anbindung des Diagramms an die Benutzerschnittstelle erfolgt über die Methode `getComponent()` der Klasse `CFlowDiagram`. Zusätzlich wird für das Einfügen neuer Elemente ein `TransferHandler` registriert, der das Ablegen von Skriptelementen mittels *Drag&Drop* ermöglicht. Die üblichen Aktionen der Benutzerschnittstelle wie das Ausschneiden, Kopieren und Einfügen von bestehenden Elementen ist mit Hilfe der entsprechenden *Actions* ohne weiteres möglich. Die Elemente des Diagramms zu markieren und verschieben sowie Verbindungen zu ziehen ist durch die Verwendung eines `MouseListeners` möglich. Details hierzu finden sich in Abschnitt 5.3.

5.2.3 Blockdiagramm

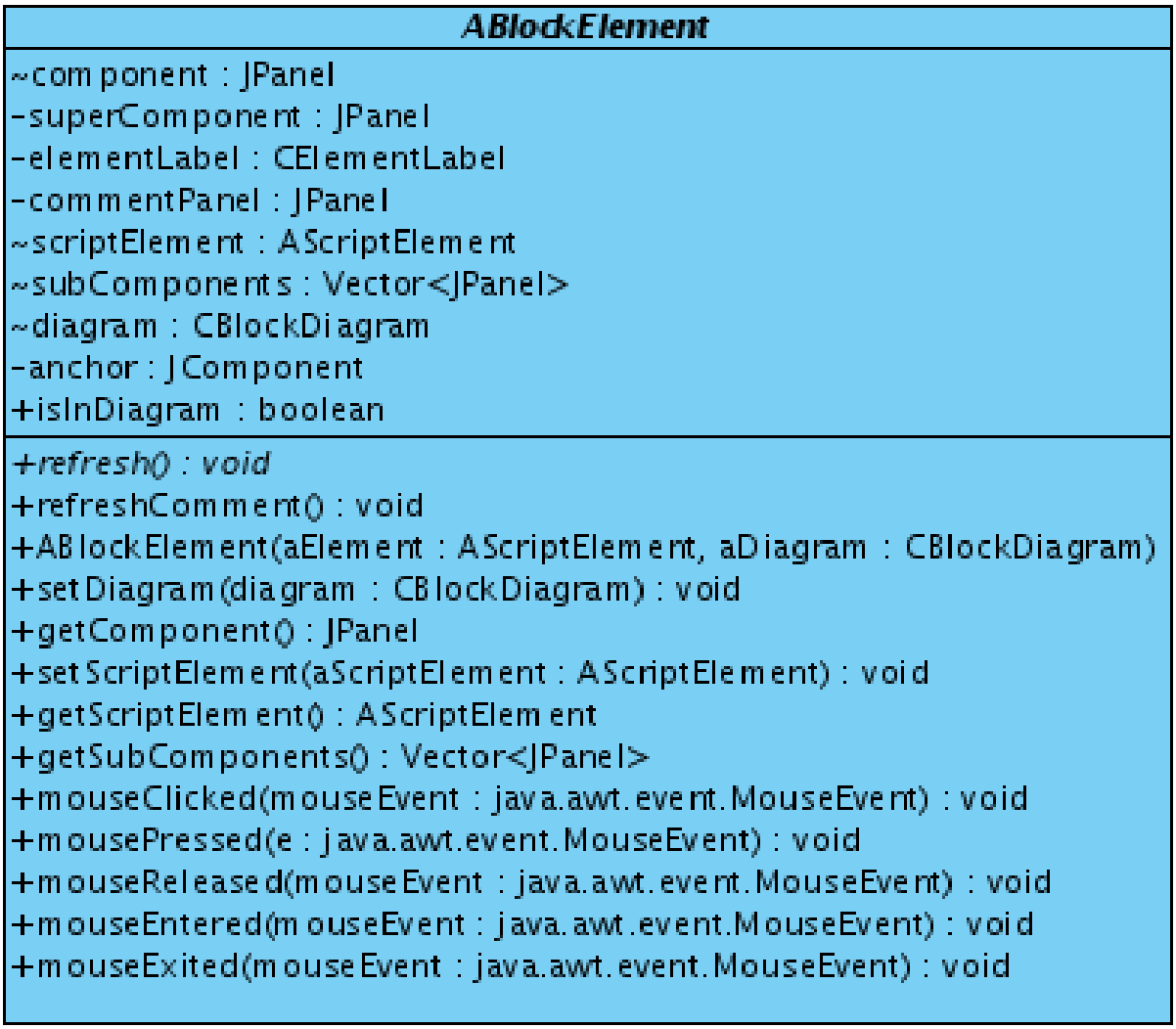


Abbildung 5.5: Die Klasse *ABlockElement*

Die Klasse `ABlockElement` stellt ein einzelnes Element des Skriptes im Blockdiagramm dar. Seine Darstellung erfolgt streng hierarchisch orientiert und basiert auf regulären *Java* Komponenten. Der allgemeine Aufbau eines Elements im Blockdiagramm ist ein `JPanel`, das alle anderen Komponenten enthält. Hierzu gehört auf der linken Seite eines Elements insbesondere ein *Ankerbereich*, der zur Manipulation des jeweiligen Elementes – dem Verschieben an andere Stelle und somit dem Lösen und Setzen von Verbindungen – dient. Dieser Bereich enthält auch ein symbolisches *Label*, das die Art des jeweiligen Elementes kennzeichnet. Des weiteren besteht ein gesonderter Abschnitt für Kommentare, die dem Element zugeordnet sein können, im oberen Bereich des Blockelementes.

Die Methoden, die ein Element im Blockdiagramm zur Verfügung stellt, dienen in erster Linie – neben Gettern und Settern – der Behandlung von Mausereignissen.

Bei der Implementierung eines konkreten Elements im Blockdiagramm wird im Prinzip nur die Methode `refresh()` gefordert. Diese Methode dient dazu, die Darstellung eines einzelnen Elementes an die neuen Gegebenheiten der internen Repräsentation anzupassen. Hierzu werden im Prinzip die aktuellen Komponenten des Elements durch neue Komponenten, die den aktuellen Gegebenheiten entsprechen ersetzt. Der eigentlichen Aufbau der „inneren“ Darstellung der Elements steht bei der Implementierung frei, lediglich der äußere Rahmen mit Icon, Anker und Bereich für Kommentare ist vorgegeben.

Beim Aufbau des Diagramms werden die Komponenten sequentiell verknüpfter Kontrollflusselemente in das Diagramm eingefügt. Innere Elemente – wie Datenquellen oder auch innere Elemente einer Schleife – werden beim Einfügen der übergeordneten Elemente mit übernommen, da sie Bestandteil der Komponente dieser Elemente sind.

5.2.4 Tests

Das Testen der semantischen Repräsentation der Skripte steht in engem Zusammenhang mit dem Testen der beiden verwendeten Diagrammarten. Aus diesem Grund sollen an dieser Stelle auch diese Tests mit behandelt werden. Die hauptsächliche Herausforderung

hierbei ist die Detektion von Abweichungen zwischen der internen, semantischen Darstellung des Skriptes und seiner Darstellung im Diagramm.

Die Basis für die Tests bildet die Spezifikation aller möglichen Skriptelemente. Es wird dann versucht, diese in vollständiger Abdeckung jeweils miteinander zu verbinden. Für jeden dieser Fälle wird anhand des semantischen Modells geprüft, ob die Verbindung hergestellt wurde. Dieses Ergebnis wird mit vordefinierten Ergebnissen verglichen, die angeben, ob die Verbindung stattfinden durfte. Da das Testen von graphischen Benutzerschnittstelllen den Rahmen dieser Diplomarbeit sprengen würde sei an dieser Stelle lediglich gesagt, dass zum einen die Klasse `java.awt.Robot` benutzt werden kann um diese Tests auf Diagrammebene durchzuführen, zum anderen dass dieser automatisierte Mechanismus kein vollwertiger Ersatz für das manuelle Testen der Diagramme ist.

Das Testen erfolgt im Flussdiagramm durch Einfügen der Elemente und Ziehen möglicher Verbindungslinien. Im Blockdiagramm hingegen werden per *Drag&Drop* die neuen Elemente direkt auf ihre Zielelemente gezogen. Im Falle des Flussdiagramms kommt das Löschen und Neuerstellen von Verbindungen weiterhin zum Test hinzu, beim Blockdiagramm vor allem das Löschen von Elementen und ihr Verschieben – was einem Neuverbinden in der semantischen Darstellung entspricht.

Auch der Wechsel der beiden unterschiedlichen Diagrammarten sollte in diesem Zusammenhang getestet werden. Insbesondere ist es wichtig festzustellen, dass die Synchronisierung des Diagramms an das – im anderen Diagramm veränderte – semantische Modell korrekt ausgeführt wird. Zu diesem Zweck bietet es sich an, immer wieder die Diagrammansicht zu wechseln während die Verbindungstests ausgeführt werden.

Ein weiterer Bereich, der beim Testen abgedeckt werden sollte ist das Layouting der Elemente. Im Blockdiagramm sollte das relativ leicht vonstatten gehen, da hierbei die LayoutManager der Komponenten eine große Rolle spielen. Die Darstellung im Flussdiagramm hingegen ist manuell implementiert und als solche auch ausgiebig zu testen. Konkret betrifft dies die automatische Positionierung von Elementen sowie die Bewegungseinschränkungen beim Verschieben der Elemente.

Weiterhin ist es sinnvoll, Zeitmessungen beim Diagrammaufbau durchzuführen. Diese

könnten Aufschluss über die Reaktionszeit des Editors auf Benutzereingaben geben. Eine niedrige Reaktionszeit ist fundamental für eine positive Erfahrung des Benutzers beim Bedienen der Software. Effizientes Zeichnen von Diagramm ist jedoch nicht Thema dieser Diplomarbeit, so dass die Toleranzzeiten für den Aufbau der Diagramme sehr hoch anzusiedeln sind.

5.3 Details zum Flussdiagramm

5.3.1 Automatische Positionierung

Bei der Verwendung des Flussdiagramms ist es sinnvoll, eine Funktion zur automatischen Positionierung der Elemente bereitzustellen. Diese tritt zum einen beim Importieren eines Skriptes aus der abstrakten XML-Darstellung in Kraft, zum anderen ist sie beim Diagrammwechsel sinnvoll anzuwenden. Sind neue Elemente während der Verwendung des Blockdiagramms eingefügt (und verbunden) worden, so wäre es sinnvoll, diese gleich den Richtlinien für die Positionierung entsprechend im Diagramm anzuordnen. Die Richtlinien sind wiefolgt:

1. Die Anordnung von Kontrollflusselementen erfolgt in vertikaler Richtung von oben nach unten. Das Nachfolgeelement darf somit nicht über das Vorgängerelement bewegt werden und wird bei der automatischen Positionierung darunter angeordnet.

2. Die Anordnung von Datenflusselementen erfolgt in horizontaler Richtung von links nach rechts. Datenquellen müssen somit immer links von ihrem Datenziel positioniert werden.

3. Bei Sequenzen müssen die Kontrollflussnachfolger innerhalb der beiden Elemente (der Gabelung des Ablaufs und dessen Zusammenführung) liegen. Dies ist der einzige Fall, in dem Elemente nach unten hin abgegrenzt werden. Zudem besteht eine horizontale Einschränkung der Elemente je nach dem Ablaufpfad, der ihnen zugeordnet ist.

4. Bei Schleifen gibt es immer einen äußeren und einen inneren Teil, dessen Ablauf wiederholt wird. Dieser innere Teil muss als Kontrollflussnachfolger unterhalb der Schleife positioniert werden. Zusätzlich jedoch tritt eine horizontale Positionierungseinschränkung in Kraft – die Elemente müssen rechts vom Schleifenelement angeordnet werden um ihre hierarchische Abhängigkeit vom Schleifenelement deutlich zu machen.

Die Umsetzung dieser Positionierung erfolgt unter Zuhilfenahme eines Wertes aus den Einstellungen, der einen Mindestabstand der Elemente untereinander definiert.

Problematisch gesehen werden muss in diesem Zusammenhang dass eine Umordnung benachbarter Elemente notwendig werden kann. Dies ist beispielsweise der Fall wenn die Breite einer Datenflussquelle deren Anordnung über den linken Rand hinaus erfordern würde.

5.3.2 Bewegungseinschränkungen

Es existieren in einem Flussdagrammelement vier Referenzen auf andere Elemente, die die Positionierung des Elements begrenzen. Diese sind in einer `HashMap` hinterlegt, zusammen mit der Art der Positionierungseinschränkung (ob das aktuelle Element beispielsweise links von dem Referenzierten sein muss).

Beim Verbinden zweier Elemente müssen diese Referenzen entsprechend der Verbindungsart gesetzt werden. Zu beachten ist hierbei die Vererbung der Positionierungseinschränkungen. Wird ein Element mit einem anderen Element verbunden, so ist zu prüfen, ob dessen Positionierung schon Einschränkungen unterliegt. Ist dies der Fall, werden diese für das neu verbundene Element übernommen und um die zusätzlichen Einschränkungen durch die aktuelle Verbindung ergänzt.

Die beim Verbinden der Elemente so verzeichneten einschränkenden Elemente werden dann beim Verschieben des Elementes abgeprüft. Dies geschieht in der Methode `applyPositioningRestrictions()`. Dieser wird eine hypothetische neue Position übergeben, auf die dann alle Einschränkungen aus den begrenzenden Elementen angewandt

werden. Die so erhaltene neue Positionierungsangabe wird dann benutzt um die eigentliche Position des Elements beim Verschieben zu setzen. Die Einschränkung der Positionierung kann in der Konfiguration des visuellen Skripteditors ausgeschaltet werden um dem Benutzer mehr Freiheit zu lassen. Die automatische Positionierung bleibt davon unberührt. Mögliche Einschränkungen in der Bewegung sind:

Allgemeine Einschränkungen: Allgemein müssen alle Elemente innerhalb eines vordefinierten Rahmens positioniert werden. Dieser grenzt den oberen und linken Teil des Diagrammes ab, um ihn bestimmten Elementen – wie beispielsweise externen Variablen – vorzubehalten.

Kontrollfluss: Wie schon erwähnt müssen Kontrollflussnachfolger unterhalb ihres Vorgängers positioniert werden.

Datenfluss allgemein: Datenquellen werden allgemein links von ihren Zielen positioniert.

Schleifen: In Schleifen kommt zur vertikalen Einschränkung der Nachfolgerelemente noch eine horizontale Beschränkung: Elemente des inneren Ablaufpfades müssen rechts von der Schleife positioniert werden. Diese Einschränkung ist für alle Nachfolger des inneren Elements vererblich.

Verzweigungen: Ähnlich den Schleifen tritt bei den Verzweigungen auch eine horizontale Einschränkung in Kraft. Diese hängt vom jeweiligen Ablaufpfad des Elementes ab – im `FALSCH`-Pfad nachfolgende Elemente müssen rechts vom Verzweigungselement angeordnet sein, befinden sie sich im `WAHR`-Pfad, so müssen sie sich auf der linken Seite befinden. Da die Verzweigung als Besonderheit aus zwei Elementen besteht tritt hier eine weitere Einschränkung in vertikaler Richtung in Kraft. Die inneren Nachfolgeelemente müssen oberhalb des Verzweigungsendes positioniert werden.

Allgemein gesagt sind die Einschränkungen der Positionierung den Richtlinien der automatischen Positionierung somit sehr ähnlich.

5.4 Details zum Blockdiagramm

An dieser Stelle soll das Blockdiagramm eingehender betrachtet werden. Da die prototypische Entwicklung des visuellen Skripteditors jedoch hauptsächlich auf Basis des Flussdiagramms durchgeführt wurde ist dieser Bereich nur relativ schwach abgedeckt. Es sei jedoch auf Abschnitt 5.2.3 verwiesen, in dem einige Kriterien zum Aufbau der einzelnen Elemente des Blockdiagramms aufgeführt sind.

Die Darstellung der einzelnen Elemente im Blockdiagramm basiert auf Java-Komponenten. Ein einzelnes Element wird als `JPanel` aufgefasst, das wiederum andere Elemente beziehungsweise deren Komponenten enthalten kann.

Das Verbinden von einzelnen Elementen erfolgt mittels *Drag&Drop*. Hierfür werden gesonderte Handler benötigt, die diese Ereignisse verarbeiten und bei den Komponenten des Blockdiagramms registriert werden.

5.5 Austauschbarkeit

Dieser Abschnitt beschäftigt sich eingehender mit austauschbaren Komponenten des visuelllen Skripteditors. Dies betrifft zum einen die unterschiedlichen Diagrammtypen die zur Darstellung der Skripte verwendet werden können, zum anderen jedoch auch die Möglichkeit, verschiedene Skriptsprachen als Zielsprache zu verwenden. Diese Austauschmöglichkeiten bringen verschiedene Aspekte mit sich, die näher erläutert werden sollen.

5.5.1 Diagramme

Um das Wechseln der beiden Diagrammarten zu ermöglichen werden beide Diagramme in der Hauptklasse `CVisualEditorFrame` hinterlegt. Bearbeitet wird immer das aktive Diagramm, erst beim Wechseln wird das *alternative* Diagramm an die neuen Gegebenheiten des Skriptes angepasst. Hierzu wird die Methode `syncWithSemanticModel()` aufgerufen, die neu hinzugekommene Elemente in das Diagramm einfügt, Elemente die inzwischen

gelöscht wurden aus dem Diagramm entfernt sowie die Verbindungen zwischen den einzelnen Elementen neu aufbaut.

In diesem Zusammenhang ist das verwendete `Observer`-Pattern kritisch zu sehen, da beim Einfügen eines Elementes in das Diagramm dieses als Beobachter registriert wird. Das alternative Diagramm jedoch muss auch registriert werden, damit dieses nach einem Wechsel immer noch auf Änderungen des Elementes korrekt reagiert. Dazu muss jedoch das Element in das Diagramm eingefügt sein.

Somit ist es notwendig, ein neues Element immer in beide Diagramme einzufügen und immer beide Diagramme als Beobachter bei diesem Element zu registrieren.

5.5.2 Übersetzung und Skriptsprachen

Die Austauschbarkeit der Skriptsprachen wird durch die Verwendung von *XSLT* gewährleistet. Die entsprechende Zielsprache jedoch muss vom Benutzer ausgewählt werden um die entsprechende Vorlage auszuwählen. Diese sind für die einzelnen Sprachen einzigartig und werden beim Übersetzungsvorgang mit den Daten aus dem abstrakten XML-Dokument, das das Skript repräsentiert, versorgt.

5.5.3 Tests

Die Austauschbarkeit ist ein kritischer Punkt in der Anwendung, da sie viel Dynamik enthält. Deshalb ist es notwendig, diesen Bereich ausgiebig zu testen um die Stabilität des Systems zu gewährleisten. Nachfolgend sind einige Tests aufgeführt, die zur Verifikation der austauschbaren Module zu geeignet sind:

1. **Ausführung von Standardaktionen mit einem Skript:** Zunächst muss gewährleistet werden, dass die elementaren Funktionen des Skripteditors zur Verfügung stehen. Hierzu bietet es sich an, ein vorgefertigtes Skript zu laden. Danach erfolgen vordefinierte, reproduzierbare Modifikationen im Skriptablauf, gefolgt vom Export

des so entstandenen Skriptes. Das erzeugte XML-Dokument muss einer vorgefertigten Datei gleichen, ansonsten ist die Reproduzierbarkeit des erzeugten Metacodes nicht gewährleistet, was den *Worst Case* bei der Verwendung des Skriptgenerators darstellt.

2. **Modifikationen:** Die durchgeführten Modifikationen des Skriptes müssen den vollen Funktionsumfang des visuellen Skripteditors darstellen um eine entsprechend vollständige Codeabdeckung zu erreichen. Hier werden Fehler bei der Durchführung einzelner Aktionen erkannt.

3. **Export als Metacode:** Der Metacode stellt das Herzstück bei der Generierung eines Skriptes dar. Dementsprechend ist er auch immer wieder – nach dem Durchführen einzelner Modifikationen – zu erzeugen. Im Anschluss erfolg jeweils ein Vergleich mit vorgefertigten Dokumenten, die das Skript in den einzelnen Entwicklungsstadien repräsentieren.

4. **Generierung des Quellcodes:** Um Fehler in der eigentlichen Erzeugung des Quellcodes aufzudecken muss auch dieser jeweils nach einer erfolgreich durchgeführten Aktion mit unterschiedlichen Vorlagen verglichen werden. Diese sind skriptsprachenabhängig und eignen sich somit immer nur zum Test einer konkreten Zielsprache. Ist einer der vorhergehenden Schritte fehlgeschlagen ist es sinnlos, diesen Test durchzuführen, da aus unterschiedlichem Metacode nicht der selbe Quelltext in der Zielsprache entstehen kann.

5. **Diagrammwechsel:** Um nun die eigentliche Austauschbarkeit des Diagrammes bzw. dessen Wechsel während der Bearbeitung eines Skriptes zu gewährleisten ist es notwendig, in diesem kontrollierten Testablauf auch immer wieder zwischen den Diagrammarten umzuschalten. Dies dient insbesondere dazu, die Konsistenz der Daten beider Diagrammtypen zu gewährleisten.

5.6 Persistenz

Bei einem Editor ist es selbstverständlich, dass auch der Persistenz der erzeugten Daten Aufmerksameit gewidmet wird. Dieser Abschnitt bietet einen Überblick, wie die Skripte gespeichert werden können.

5.6.1 Datenbank

Der wichtigste Aspekt bei der Speicherung der Skripte ist das Ablegen des jeweiligen Skriptes in der Datenbank. Dies geschieht in zwei verschiedenen Formen. Zum einen ist es notwendig, den erzeugten Quellcode abzulegen um eine Ausführung seitens der Testengine zu gewährleisten. Zum anderen jedoch ist es auch notwendig, eine reproduzierbare Darstellung des Skriptes in der Datenbank zu speichern. Diese dient dazu, ein bestehendes Skript im visuellen Skripteditor weiter zu verändern. Alle enthaltenen Informationen – wie Positionierungsdaten bleiben hier bei enthalten.

Beide Speichermethoden werden mittels *Hibernate* realisiert.

- **Speicherung des visuellen Skriptmodells:** Das visuelle Skriptmodell wird in der Datenbank ähnlich abgespeichert wie beim Export als Datei. Es wird eine binäre Repräsentation (eine serialisierte Version) der Klasse `CVisualScript` in der Datenbank abgelegt. Auch in ihr enthaltene Objekte – wie die einzelnen Skriptelemente – werden mit serialisiert und in diese eine Bytecodedarstellung mit überführt. In der Datenbank wird das Objekt dann als *Binary Object* abgelegt.

- **Speicherung des ausführbaren Skriptes:** Der Quellcode des Skriptes an sich wird als Zeichenkette in der Datenbank hinterlegt. Die Zuordnung erfolgt in der Mapping-Datei zur Klasse `CScript`.

Die Klasse `CScript` dient als Container für Skripte – egal ob diese nur in textueller Form oder auch als Diagramm vorliegen. Auch der Rückgabetyp des Skriptes (falls vorhanden) und notwendige Übergabeparameter werden in dieser Klasse verwaltet. Eine genauere

Beschreibung dieser Klasse findet sich in Abschnitt 5.8.2. Im Prinzip werden Objekte dieser Klasse in der Datenbank abgelegt, die dann die einzelnen Repräsentationen des Skriptes enthalten.

5.6.2 Export

Der Export des aktiven Diagrammes in eine Datei kann auf zwei verschiedene Arten erfolgen – entweder es wird das gesamte Skript als Binärdatei einschließlich aller Zusatzinformationen wie Positionsdaten gespeichert, oder aber es wird das gerade bearbeitete Skript in XML exportiert. Hierbei gehen alle Zusatzinformationen verloren, nur die für das eigentliche Skript relevanten Daten werden gespeichert. Auch der fertig generierte Quelltext des Skriptes kann exportiert werden, beispielsweise um Fehler in der Quelltextgenerierung aufzudecken.

5.6.3 Tests

Auch die Tests für die Persistenzschicht des visuellen Skripteditors sind sehr stark an die Tests der semantischen Repräsentation der Skripte angelehnt. Bei dem regulären Testvorgehen (vgl. Abschnitt 5.2.4) werden lediglich zusätzlich an definierten Punkten Speicher- und Lade- beziehungsweise Im- und Exportoperationen ausgeführt. Die aktuelle Darstellung im gerade bearbeiteten semantischen Modell wird dann immer wieder mit den neu eingelesenen Daten verglichen. Dieses Vorgehen sollte ausreichen um eine Speichersicherheit zu gewährleisten. Hierbei ist jedoch noch die Unterscheidung zu treffen, ob eine Exportierung stattfand (in diesem Fall können nur semantische Gesichtspunkte überprüft werden) oder ob die gesamte Darstellung einschließlich Postionierungsdaten gesichert wurde (in diesem Fall werden die Positionsdaten aller Elemente mit in die Überprüfung einbezogen).

5.7 Code-Generierung

5.7.1 Interner Ablauf

Die eigentliche Codegenerierung läuft in verschiedenen Stufen ab, die aufeinander aufbauen.

1. **Erzeugung des Metacodes:** Zunächst wird aus der Struktur der Elemente eine abstrakte semantische Darstellung in XML gewonnen. Diese dient als Basis für den Übersetzungsvorgang.

2. **Validierung:** An dieser Stelle tritt die Überprüfung des Metacodes hinsichtlich verschiedener Gesichtspunkte in Kraft. Sie wird mittels *XML Schema* durchgeführt und dient der Sicherstellung einer kohärenten Basis für die Übersetzung.

3. **Transformation mittels XSLT:** Abschließend erfolgt die eigentliche Übersetzung in die gewählte Zielsprache. Abhängig von der Benutzerauswahl wird die entsprechende *XSL*-Vorlage verwendet um aus dem bestehenden XML-Dokument den korrespondierenden Quelltext in der Zielsprache zu erzeugen.

Als Abschluss dieses Vorgehens bietet es sich an, den erzeugten Quelltext noch einmal auf Fehler zu überprüfen. Dass diese endgültige Überprüfung jedoch nicht trivial ist wird in Abschnitt 5.7.5 näher erläutert.

Anhang A stellt die unterschiedlichen Verarbeitungsstufen bei der Erstellung eines Skriptes in den verschiedenen Stadien anschaulich dar.

5.7.2 Erzeugung des Metacodes

Die Erzeugung des Metacodes erfolgt Elementweise rekursiv. Angefangen beim Wurzelement (dem `START`-Knoten des Skriptes) wird ein `StringBuilder` immer um die XML-Darstellung der nachfolgenden Elemente erweitert. Als nachfolgende Elemente werden sowohl innere Elemente (beispielsweise in Schleifen oder Datenquellen) als auch sequentiell

folgende Elemente bezeichnet. Das generelle Vorgehen innerhalb eines einzelnen Elementes ist hierbei:

1. **Beginn des XML-Tags für das Element:** Die Bezeichnung des Tags wird aus der Methode `getXMLName()` erhalten. Hier werden auch alle relevanten Eigenschaften des Elements (wie beispielsweise der *Typ* einer Schleife) mit in das Dokument eingetragen.

2. **Einfügen des Kommentars:** Falls vorhanden wird der dem Element zugeordnete Kommentar als Unterknoten in das Dokument eingefügt. Dies geschieht unter Verwendung der Methode `generateCommentXML()` der Klasse `AScriptElement`.

3. **Einfügen der Datenquellen:** Bei Elementen, die Datenquellen verwenden wird deren XML-Repräsentation an dieser Stelle in das Dokument eingefügt. Verwendet wird hier die Hilfsmethode `generateDataSourcesXML()` der Klasse `AScriptElement`.

4. **Einfügen innerer Elemente:** Im Anschluss daran werden innere Kontrollflusselemente (beispielsweise die Elemente im `WAHR`- bzw. `FALSCH`-Pfad einer Verzweigung) in das Dokument eingefügt. Diese werden von Tags umschlossen, die den Pfad der komplexeren Struktur definieren. Diese werden benötigt um beim Rückparsen der XML-Struktur eine eineindeutige Zuordnung des Kontrollflusses der Elemente zu ermöglichen.

5. **Fortfahren mit nachfolgenden Elementen:** Als letzter Schritt wird dann jeweils die XML-Repräsentation des nachfolgenden Skriptelements (falls vorhanden) an das Dokument angehängt und somit die Verarbeitung bis zum Ende-Knoten fortgesetzt.

Für einige wenige Elemente werden weiterhin Daten innerhalb des Elements in der XML-Struktur abgelegt. Dies sind insbesondere Daten, die sich nicht unbedingt für eine Darstellung als Attribut des Knotens eignen, da sie beispielsweise Sonderzeichen enthalten, die in XML-Attributen nicht vorkommen dürfen oder schlicht zu umfangreich sind als dass sie strukturell als Attribut gehandhabt werden könnten.

Ein weiterer Aspekt bei der Repräsentation des Skriptes in XML ist die Behandlung von Sonderzeichen. Hier bestehen sowohl in der Datenspeicherung einer Eigenschaft als Attribut, als auch als `PCDATA` des XML-Knotens Einschränkungen. Deshalb ist es notwendig, für einzelne Sonderzeichen eine alternative Darstellungsform zu finden. Beispiel hierfür wäre die Speicherung eines mehrzeiligen Kommentars – hier würden Zeilenumbrüche bei der Speicherung des Textes als Attribut beim Rückparsen einfach überlesen. Umlaute hingegen können schon durch Kodierungsprobleme des Dokuments zu gravierenden Fehlern führen.

Zur Darstellung der einzelnen Elemente in XML sei an dieser Stelle in erster Linie auf Abschnitt 4.4.1 verwiesen, in dem im Detail auf die individuellen Eigenschaften der Skriptelemente eingegangen wird. Diese werden wie oben beschrieben innerhalb des XML-Dokuments mit abgelegt.

Eine weitere Möglichkeit, die einzelnen Elemente in eine XML-Darstellung zu überführen wäre die Verwendung der *Betwixt*-Bibliothek der *Jakarta Commons*, wie dies in Abschnitt 5.2.1 angesprochen wurde. Dies wurde jedoch beim Prototyping nicht getestet.

Um eine Rückumwandlung von Skripten zu unterstützen, die nicht mit dem visuellen Skripteditor generiert wurden existieren zwei weitere Elementtypen, die vorher keine Erwähnung fanden. Diese *generischen Elemente* dienen dazu, einzelne unidentifizierte Anweisungsblöcke in einer bestimmten Skriptsprache auch im visuellen Skripteditor darstellen zu können. Dies sind die Elementklassen `Condition` – welche eine unbekannte Datenquelle repräsentiert – und `COperation` — welche einen unbekannten Anweisungsblock repräsentiert.

5.7.3 Übersetzung des Metacodes in die Zielsprache

Der erzeugte Metacode, der in XML vorliegt, wird mittels *XSLT* in die eigentliche Zielsprache übersetzt. Dies hat vor allem den Vorteil, dass der Sprachschatz des Codegenerators erweitert werden kann, ohne dass ein weiterer Eingriff in seinen Quelltext nötig ist.

Es muss lediglich eine weitere Vorlage erstellt werden und in dem Verzeichnis *Templates* hinterlegt werden. Beim Start des Übersetzungsmoduls wird dann dieses Verzeichnis durchsucht, woraufhin der Benutzer die Zielsprache auswählt und der Übersetzungsvorgang beginnt.

Prinzipiell läuft die Übersetzung von domänenspezifischen Aufrufen – wie das Ausführen eines *Lauts*-Testknotens – genauso ab wie die Übersetzung von Standard-Sprachkonstrukten. Der einzige Unterschied liegt im Aufruf einer Methode der Skriptengine. Dies war auch der primäre Grund, warum Zielsprachen *objektbasiert* sein müssen statt rein prozedural.

Das nachfolgende Listing stellt die XSLT-Vorlage für eine Verzweigung im Skript dar. Da in XSLT Variablen im weiteren Verlauf nicht veränderbar sind muss für die Einrückung des Quelltexts ein Parameter verwendet werden, der rekursiv an die einzelnen Templates weitergegeben wird.

Listing 5.2: Die XSLT-Vorlage für eine Verzweigung in *Groovy*

```
1 <!-- This Template is used to create the source code for a selection
    -->
2 <xsl:template match="selection">
3   <xsl:param name="indent"/>
4   <!-- create the comment text (if there is a comment assigned)-->
5   <xsl:apply-templates select="comment">
6     <xsl:with-param name="indent" select="$indent"/>
7   </xsl:apply-templates>
8   <!-- print "if(" (indented)-->
9   <xsl:call-template name="indent">
10    <xsl:with-param name="indent" select="$indent"/>
11  </xsl:call-template>
12  <xsl:text>if(</xsl:text>
13  <!-- insert the code for the condition and finish the condition
      line-->
14  <xsl:apply-templates select="dataSources"/>
15  <xsl:text>){&#10;</xsl:text>
16  <!-- insert the elements from the "true" path (indention increased)
      -->
17  <xsl:for-each select="true/*">
18    <xsl:apply-templates select=".">
19      <xsl:with-param name="indent" select="$indent+1"/>
20    </xsl:apply-templates>
21  </xsl:for-each>
22  <!-- print "}else{"-->
23  <xsl:call-template name="indent">
24    <xsl:with-param name="indent" select="$indent"/>
25  </xsl:call-template>
26  <xsl:text>} else{&#10;</xsl:text>
```

```
27   <!-- as for the "true" path, so for the "false" path-->
28   <xsl:for-each select="false/*">
29     <xsl:apply-templates select=".">
30       <xsl:with-param name="indent" select="$indent+1"/>
31     </xsl:apply-templates>
32   </xsl:for-each>
33   <!-- finish the if-statement-->
34   <xsl:call-template name="indent">
35     <xsl:with-param name="indent" select="$indent"/>
36   </xsl:call-template>
37   <xsl:text>}&#10;</xsl:text>
38   <!-- continue generating code for the following element-->
39   <xsl:apply-templates select="controlDestination">
40     <xsl:with-param name="indent" select="$indent"/>
41   </xsl:apply-templates>
42 </xsl:template>
```

Die Verwendung der XSLT-Vorlage zur eigentlichen Codegenerierung erfolgt in Java mit Hilfe des Packages `javax.xml.transform`. Das Vorgehen hierbei erfolgt in drei Schritten – Erstellung einer `Factory` für den Übersetzer, Erzeugen des `Transformer`s für das gewählte XSLT-Dokument und Aufrufen der Methode `transform()`.

5.7.4 Validierung

Da der visuelle Skripteditor von Nichtprogrammierern verwendet werden soll ist es notwendig, gewisse Mechanismen zur Prüfung der erstellten Skripte zu verwenden. Diese Prüfungen werden sowohl während der Erstellung des Skriptes, als auch bei der Übersetzung des Skriptes ausgeführt.

Beim Erstellen des Skriptes

Während der Erstellung bzw. Bearbeitung eines Skriptes findet eine Laufzeitprüfung statt. Diese wird beim Verbinden einzelner Elemente angestoßen und dient dazu, Fehler in der Semantik der Elementstruktur aufzudecken. Wird ein solcher festgestellt, wird die Verbindung nicht erlaubt und nicht durchgeführt. Die Logik hierfür ist in den einzelnen

Elementen hinterlegt, wie es dem objektorientierten Ansatz entspricht. Die jeweilige Elementklasse an sich entscheidet, ob sie eine bestimmte Verbindung akzeptiert oder nicht. Es werden verschiedene Aspekte berücksichtigt:

- **Prüfen belegter Ein-/Ausgänge:** Zunächst wird geprüft, ob die angeforderte Verbindung nicht hergestellt werden kann, weil an der entsprechenden Stelle bereits ein Element verbunden ist. Bereits bestehende Verbindungen werden nicht ersetzt, dies ist Aufgabe des Benutzers.

- **Prüfen des benutzten Datentyps:** Speziell bei Datenflussverbindungen muss geprüft werden, ob das aktuelle Element das zu verbindende Element als Datenquelle akzeptiert. Dies geschieht durch Aufruf der Methode `isDataFlowSourceValid()` des Zielelements. Parameter ist das aktuelle Element, `this`, das mit dem Element verbunden werden soll.

- **Prüfen zyklischer Verbindungen:** Spricht an dieser Stelle noch nichts gegen die Verbindung der beiden Elemente, so wird eine weitere Prüfung ausgelöst. Diese dient dazu, zyklische Verbindungen von Elementen auszuschließen. Solch zyklische Verbindungen würden bei der Generierung des Metacodes starke Komplikationen auslösen (möglicherweise endlose Rekursion). Je nach ermitteltem Verbindungstyp (Daten- oder Kontrollfluss) werden die Verbindungen des Zielelementes abgelaufen. Wäre das aktuelle Element in dieser Verbindungskette zu finden wäre dies eine zyklische Verbindung. In einem solchen Fall wird die Verbindung abgelehnt.

Bei einzelnen Elementen ist es durchaus sinnvoll, diese Logik um weitere Aspekte zu erweitern. So ist beispielsweise der Charakter einer Verbindung zu einer Variablen abhängig von ihrer Verwendung. Bei lesendem Zugriff dient sie als Datenquelle, bei schreibendem Zugriff hingegen dient sie als Ziel für eine Datenquelle und ist in den Kontrollfluss eingebunden. Anhand dieser Eigenschaft muss somit die Entscheidung getroffen werden, welche Art von Verbindungen erlaubt sind. Es macht durchaus Sinn, sie bei der ersten Verbindung der Variable zu einem anderen Element dynamisch zu setzen und alle anschließenden Verbindungsentscheidungen darauf zu basieren. Dies würde die Verwendung der Variablen beim Erstellen des Skriptes intuitiver und produktiver gestalten.

Beim Erzeugen des Quellcodes

Zwischen der Generierung des Metacodes und der eigentlichen Übersetzung in die Zielsprache ist es sinnvoll, eine weitere Überprüfung des Skriptes durchzuführen. Diese wird direkt auf das erzeugte XML-Dokument angewendet und kann somit mit Standardmitteln zur Verifikation von XML-Strukturen – wie *XML Schema* oder *DTD*s – durchgeführt werden.

Weiterhin ist es sinnvoll, beim eigentlichen Übersetzungsvorgang mittels XSLT Knoten, die keine Entsprechung in der Vorlage besitzen zu reklamieren und die Verarbeitung abzubrechen. Solche Knoten ohne Vorlage wären ein Zeichen dafür, dass die gewählte Vorlage nicht alle möglichen Elemente eines Skriptes unterstützt und somit eine Überarbeitung benötigt.

Zur weiteren Überprüfung – insbesondere der Übersetzungsvorlage – ist es sinnvoll, den fertig erzeugten Quelltext in einem Interpreter beziehungsweise einen Simulator der Skriptengine laufen zu lassen. Dieses Vorgehen würde ganz klar stellen, dass der Quelltext auch später im Ernstfall auf der Testengine ausgeführt werden kann.

5.7.5 Tests

Prinzipiell sind die Tests für die Codegenerierung denen für die semantische Repräsentation der Skripte (vgl. Abschnitt 5.2.4) sehr ähnlich. Es werden vordefinierte Metaskripte importiert oder aber im Rahmen des Unit-Tests „manuell“ erzeugt. Diese werden dann in kontrollierten Zuständen in den Metacode umgesetzt und im Anschluss mit vorgefertigten Schablonen verglichen. Dieses Vorgehen – für möglichst alle Skriptelemente ausgeführt – sollte Ausreichen um Unstimmigkeiten in der Erzeugung des Metacodes aufzudecken.

Der nächste Schritt beim Test der Codegenerierung ist die Übersetzung dieser validierten Metacode-Repräsentationen von Skripten in eine (zu testende) Zielsprache. Auch hier werden die Ergebnisse wieder mit vorliegenden Dateien verglichen. Das Ergebnis darf

nicht abweichen. Dies gewährleistet auch, dass im Fall einer Änderung der *XSLT*-Vorlage das entsprechende Referenzskript mit angepasst werden muss.

Das vorher schon angesprochene Ausführen des Zielcodes in einem Interpreter zur Aufdeckung von Fehlern in den Vorlagen ist auch und vor allem bei deren Veränderung anzuraten. Jedoch lässt sich dies nur schlecht automatisieren, da eine prompte Reaktion auf gefundene Fehler – eine Anpassung der *XSLT*-Vorlage sowie des Referenzskriptes – erfolgen sollte.

Ein zusätzliches Problem bei der Durchführung dieses Testschrittes ist die Abhängigkeit der Skripte von ihrer vordefinierten Umgebung – der Skriptengine. Da diese für alle *Lauts*-relevanten Aufrufe innerhalb eines Skriptes zuständig ist ist es an dieser Stelle sinnvoll, einen Simulator zu erstellen, der zu testenden Skripten bzw. Vorlagen diese konsistente Umgebung zur Verfügung stellt. Der Abschnitt 6.2 des Ausblicks greift diese Problematik auf.

5.8 Integration in Lauts

Die Integration des visuellen Skripteditors in das *Lauts*-System ist hinsichtlich verschiedener Aspekte kritisch. Es muss clientseitig möglich sein, bestehende Skripte zu bearbeiten, neue Skripte zu erstellen und die Änderungen in die Datenbank abzulegen. Auf der anderen Seite muss eine korrekte Verwendung der Skripte auf der Testengine bei der Ausführung sichergestellt werden.

Da jedoch bei der prototypischen Implementierung des visuellen Skripteditors diese Integration vorerst nicht stattfand soll an dieser Stelle nur ein konzeptioneller Überblick über das Vorgehen gegeben werden.

5.8.1 Einstellungen des Editors

Die Einstellungen des Editors (beispielsweise Schriftarten, Farbwahl und ähnliches) werden in eines Klasse verwaltet, die als Singleton implementiert wird. Sie ist eine Unterklasse der Java `Properties`-Klasse und wertet dementsprechend eine dazu existierende Konfigurationsdatei bei der ersten Instanziierung aus.

Da der visuelle Skripteditor stark in den *Lauts*-Client integriert wird ist es an dieser Stelle sinnvoll, dieselbe Klasse für die Einstellungen beider Module zu verwenden.

Ein ähnliches Vorgehen ist bei der Lokalisierung des visuellen Skripteditors angeraten – auch hier lässt sich das selbe Ressource Bundle wie für den Client für den visuellen Skripteditor nutzen.

5.8.2 Repräsentation eines Skriptes in Lauts

Die Repräsentation eines einzelnen Skriptes lehnt sich stark an (Kli07) an. Hinzu kommt jedoch, dass ein Skript aus zwei Teilen bestehen kann – dem ausführbaren Skript an sich und der visuellen Repräsentation als Diagramm.

Deshalb existiert eine Klasse – `CScript` – die alle Eigenschaften des Skriptes repräsentiert. Neben den geforderten Übergabeparametern und dem Rückgabetyp des Skriptes enthält sie Informationen, die zur eineindeutigen Charakterisierung des Skriptes dienen. Hierzu gehört insbesondere die ID des Skriptes, anhand derer es in der Datenbank identifiziert wird.

Sowohl der Quelltext des Skriptes, als auch die visuelle Repräsentation sind in dieser Klasse hinterlegt – der Quelltext als Zeichenkette und die visuelle Repräsentation direkt als Objekt der Klasse `CVisualScript`.

5.8.3 Aufruf des Editors

Der Aufruf des visuellen Skripteditors erfolgt, wie beim textuellen Skripteditor, durch das Doppelklicken auf einen Testknoten. Ist diesem Knoten noch kein Skript zugeordnet, wird der Benutzer jedoch aufgefordert auszuwählen, ob das Skript textuell oder visuell erstellt werden soll. Bei einer bestehenden Zuordnung eines Skriptes entfällt diese Nachfrage – es wird der Editor gestartet, mit dem dieses Skript erstellt wurde.

Da der visuelle Skripteditor nicht in der Lage ist, bestehenden Quelltext in eine abstrakte Darstellung zu überführen, ist die einzig mögliche Behandlungsweise für textuelle Skripte das Öffnen im textuellen Editor. Für fortgeschrittene Benutzer besteht die Möglichkeit, den erzeugten Quelltext eines visuellen Skriptes im textuellen Editor zu öffnen um dieses um weitere Funktionalitäten zu erweitern. Dieses Skript verliert jedoch seine Darstellbarkeit als abstraktes Modell und seine Skriptsprachenunabhängigkeit. Es wird somit effektiv in ein textuelles Skript konvertiert.

5.8.4 Parameter- und Variablenübernahme

Die an das Skript übergebenen Parameter sowie der Rückgabewert des einzelnen Skriptes werden in der Klasse `CScript` verwaltet. Diese ist sowohl für textuelle als auch visuelle Skripte zuständig. Dies ermöglicht es, auch textuelle Skriptbausteine im visuellen Skripteditor zu benutzen. Übergabeparameter werden hierbei als *HashMap* geführt – dem Namen einer Variablen wird ihr Datentyp zugeordnet. Der Rückgabewert des Skriptes selbst ist als Attribut `dataType` hinterlegt, auf das mit entsprechenden Settern und Gettern zugegriffen werden kann.

Die für das Skript verfügbaren Variablen der *Lauts*-Teststruktur abzufragen ist jedoch nicht einfach innerhalb des Scripting-Systems möglich. Sie hängen primär von der Stelle ab, in der das Skript in die umgebende Teststruktur eingebunden ist. Zudem müssen sie bei der Bearbeitung des Skriptes im Editor genauso zugänglich sein wie beim Ausführen des Skriptes auf der Testengine. Um dieses Problem zu lösen gibt es nur eine Möglichkeit – es muss im *Lauts*-Testknoten eine Methode existieren, mit der die aktuell verwendbaren

Variablen abgefragt werden können. Diese Methode, `getAvailableVariables()` liefert eine HashMap zurück, die genauso aufgebaut ist wie die Repräsentation der Übergabeparameter. Diese dient dann im Editor zur Einbindung von Umgebungsvariablen in das aktuell bearbeitete Skript.

Bei der Ausführung des Skriptes hingegen wird dann unter Verwendung zweier Methoden – `getVariableValue()` und `setVariableValue()` – auf diese Umgebungsvariablen zugegriffen. Bei der Generierung des Quellcodes ist es somit notwendig, eine Unterscheidung zwischen internen Skriptvariablen und externen *Lauts*-Variablen zu treffen, da ihr Zugriff zur Laufzeit des Skriptes gänzlich unterschiedlich erfolgen muss.

5.8.5 Ausführung der Skripte

Allgemeine Ausführung

Die eigentliche Ausführung der Skripte erfolgt wie in (Kli07) beschrieben. Die Testengine durchläuft die *Lauts*-Teststruktur bis sie auf einen Testknoten trifft, dem ein Skript zugeordnet ist. Dieses wird dann ohne weiteres mit Hilfe der *Scriptengine* ausgeführt. Diese stellt alle nötigen Umgebungseigenschaften zur Verfügung.

Hierzu gehört auch ein Interpreter für die Sprache des jeweiligen Skriptes, der im Rahmen eines *Plugin*-Systems zur Verfügung steht. Dieser muss alle im Skript vorkommenden Ausdrücke und Konstrukte beherrschen. Das Skript wird somit innerhalb dieser kontrollierten Umgebung ausgeführt, bis ein domänenspezifischer Aufruf erfolgt.

Ausführung und Ansteuerung domänenspezifischer Methoden

Domänenspezifische Aufrufe sind Aufrufe, die nicht direkt vom Interpreter der Skriptsprache ohne weiteres behandelt werden. Hierzu gehören alle Aufrufe, die *Lauts*-interne Objekte verwenden. Ein Beispiel hierfür wäre das Ausführen eines *Lauts*-Testknotens aus einem Skript heraus. Auch eine simple Ausgabeoperation zählt hierzu, da sie über die Testengine protokolliert und an den verbundenen Client weitergeleitet wird.

Solche Aufrufe werden durch Ansteuerung von Methoden realisiert, die die Scriptengine dem Interpreter zur Verfügung stellt. Genauere Details hierzu finden sich in (Kli07).

6 Ausblick

6.1 Darstellung vorhandener Skripte

Theoretisch ist es möglich, bestehende textuelle Skripte in eine visuelle Darstellung zu überführen. Dies ist in der begrenzten Anzahl an sprachlichen Konstrukten der Sprachen begründet. Erforderlich hierfür ist jedoch ein massiver Aufwand für die Unterstützung möglichst vieler Sprachelemente pro Sprache.

Eine weitere Einschränkung bei der visuellen Darstellung vorhandener Skrippte ist das gewählte Paradigma der *objektbasierten Programmierung*. Soll ein vorhandenes Skript dargestellt werden, muss es dieses erfüllen, sonst treten Probleme auf. Der visuelle Skripteditor ist nicht darauf ausgelegt, *objektorientierte* Strukturen wie Klassen und ähnliches darzustellen. Alle solche unbekannten Konstrukte würden als „generisches Element“ in das Diagramm einfließen. Dies würde zu einer Häufung solcher Elemente im Diagramm führen die zum einen die Übersichtlichkeit stark bedrohen würden, zum anderen jedoch auch die Unabhängigkeit des Skriptes von einer bestimmten Sprache nichtig machen würden.

Somit ist es nicht allgemein sinnvoll, bestehende Skripte in eine visuelle Darstellung zu überführen. Jedoch wäre dies möglich, und beim Erfüllen bestimmter Vorraussetzungen seitens des Skriptes auch angebracht. Da jedoch der visuellle Skripteditor stark domänenspezifisch aufgebaut ist (hauptsächliche Unterstützung für Elemente der *Lauts*-Teststruktur) würde eine Erweiterung des Skripteditors auf eine allgemeinere Basis sehr viel Aufwand bedeuten.

Die konsequente Trennung von Semantik und Darstellungsform hingegen könnte sich in

diesem Zusammenhang als Vorteil erweisen, da sich beispielsweise Module zur Anzeige neuer Elemente (auf abstrakter Ebene) und einzelne Übersetzungsmodule (auf Skriptsprachenebene) getrennt entwickeln lassen.

6.2 Simulation von Skriptabläufen

Wie schon bei der Beschreibung des Testvorgangs erzeugter Skripte (vgl. Abschnitt 5.7.5) beschrieben ist es sinnvoll, einen Simulator zu erstellen, in dem dann die erstellten Skripte getestet werden können. Dieser muss im Prinzip die komplette Umgebung wie bei der Ausführung auf der Testengine zur Verfügung stellen. Auf diese Art könnten dann sowohl die Vorlagen für die Übersetzung der Skripte getestet werden, als auch der Ablauf der Skripte selbst, ohne dass ein konkretes Testsystem belegt würde.

Ein weiterer Vorteil eines solchen Simulators wäre die Möglichkeit, den eigentlichen Ablauf des Skriptes besser visuell darzustellen. Es könnte beispielsweise gekennzeichnet werden, welches Element des Skriptes gerade in Bearbeitung ist oder wie die einzelnen Datenflusselemente miteinander in Beziehung stehen.

Auch wäre es sinnvoll, in diesem Zusammenhang Funktionalitäten zur Fehlererkennung bzw. zum Debugging der Skripte zur Verfügung zu stellen. Es wäre beispielsweise möglich, die Ausführung des Skriptes zu einem bestimmten Zeitpunkt anzuhalten und die existierenden Datenobjekte auf deren Inhalt hin zu kontrollieren.

Leider muss an dieser Stelle gesagt werden, dass ein solcher Simulator recht aufwendig in der Implementierung wäre. Besonders die Austauschbarkeit der Zielsprachen könnte große Probleme nach sich ziehen. Notwendig wäre vor allem eine API für die jeweilige Skriptsprache die in etwa den Funktionsumfang der *Java Reflection* API besitzt. Eine Lösung hierfür könnte die Verwendung der internen Darstellung für die einzelnen Skriptelemente für die Ablaufsimulation sein. Die verwendeten Java-Klasseninstanzen könnten dann mittels *Reflection* analysiert werden. Dieser Ansatz würde jedoch nur eine Scheinsicherheit bieten, da nicht das eigentliche, auf der Testengine ausgeführte, Skript getestet würde.

6.3 Weitere denkbare Anwendungsszenarien

Sowohl der Ansatz der visuellen Programmierung als auch die Prinzipien der Quellcode-Generierung sind vielseitig einsetzbar. Im Folgenden sollen einige Anwendungsszenarien dieser beiden Konzepte kurz skizziert werden.

6.3.1 Ausbildung

Da der visuelle Ansatz zur Programmierung relativ intuitiv ist, bietet es sich an so etwas wie einen visuellen Skripteditor im Informatikunterricht zu verwenden. Er könnte eine Möglichkeit bieten, den Zusammenhang zwischen den Abläufen des Programms und dem verwendeten bzw. erzeugten Quelltext für Programmiereinsteiger klarer darzustellen.

So ließe sich von kleinen Beispielprogrammen bis hin zu größeren Zusammenhängen verschiedener Elemente eines Skriptes die Komplexität von Übungsaufgaben schrittweise steigern. Frustrationserfahrungen wie Abschreib- oder Syntaxfehler bei der Erstellung des Quelltexts könnten gerade bei Anfängern vermieden werden. Dies könnte die Motivation für das Unterrichtsfach erhöhen.

6.3.2 Steuerungssysteme

Visuelle Programmiersysteme werden bei Steuerungssystemen – wie beispielsweise *Siemens S7* schon seit längerer Zeit verwendet. Die intuitive Darstellung der Abläufe – insbesondere auch die klare Trennung von Kontroll- und Datenfluss hat durchaus einen gewissen Anteil an der Popularität solcher Systeme.

In den meisten Fällen solcher Steuerungssysteme wird zwar nicht Quelltext erzeugt sondern meist gleich eine kompilierte Binärdatei die in einer kontrollierten Umgebung ausgeführt wird, es gibt jedoch auch Gegenbeispiele wie *tresos GUIDE* (vgl. Abschnitt 2.2.2), die Codegenerierung betreiben.

6.3.3 Aufwandsreduzierung bei größeren IT-Systemen

Praktiken der Codegenerierung sind vor allem auch bei der Entwicklung größerer Systeme von Vorteil. So gibt es mittlerweile in sehr vielen Entwicklungsumgebungen für objektorientierte Sprachen einzelne Generatoren die beispielsweise Getter und Setter für einzelne Klassenattribute generieren.

Dieses Vorgehen lässt sich durch die Verwendung eigener Codegeneratoren noch deutlich weiter treiben. Gerade im Bereich von *J2EE*-Anwendungen und ähnlich skalierten Projekten ist es manchmal durchaus notwendig, dass viele Klassen annähernd den gleichen Aufbau und ähnliche Eigenschaften beziehungsweise Methoden aufweisen. An dieser Stelle kann es deutlich einfacher sein anstelle 50 Klassen manuell zu erstellen einen Codegenerator zu entwerfen, der die Klassen aus einer Vorlage und individuellen Eingangsdaten automatisch erstellt. Dies erhöht auch die Wartbarkeit des Codes, falls eine Änderung – wie im Falle eines nötigen Bugfixes – alle Klassen betreffen würde. Hier könnte man einfach die Vorlage ändern und die Klassen neu erzeugen lassen

A Beispielskript

Dieser Abschnitt stellt noch einmal die Entwicklungsstadien eines konkreten Skriptes, von der Erstellung im Editor – mit beiden Diagrammansichten – über die Metadarstellung in XML bis hin zum fertigen Quelltext, dar.

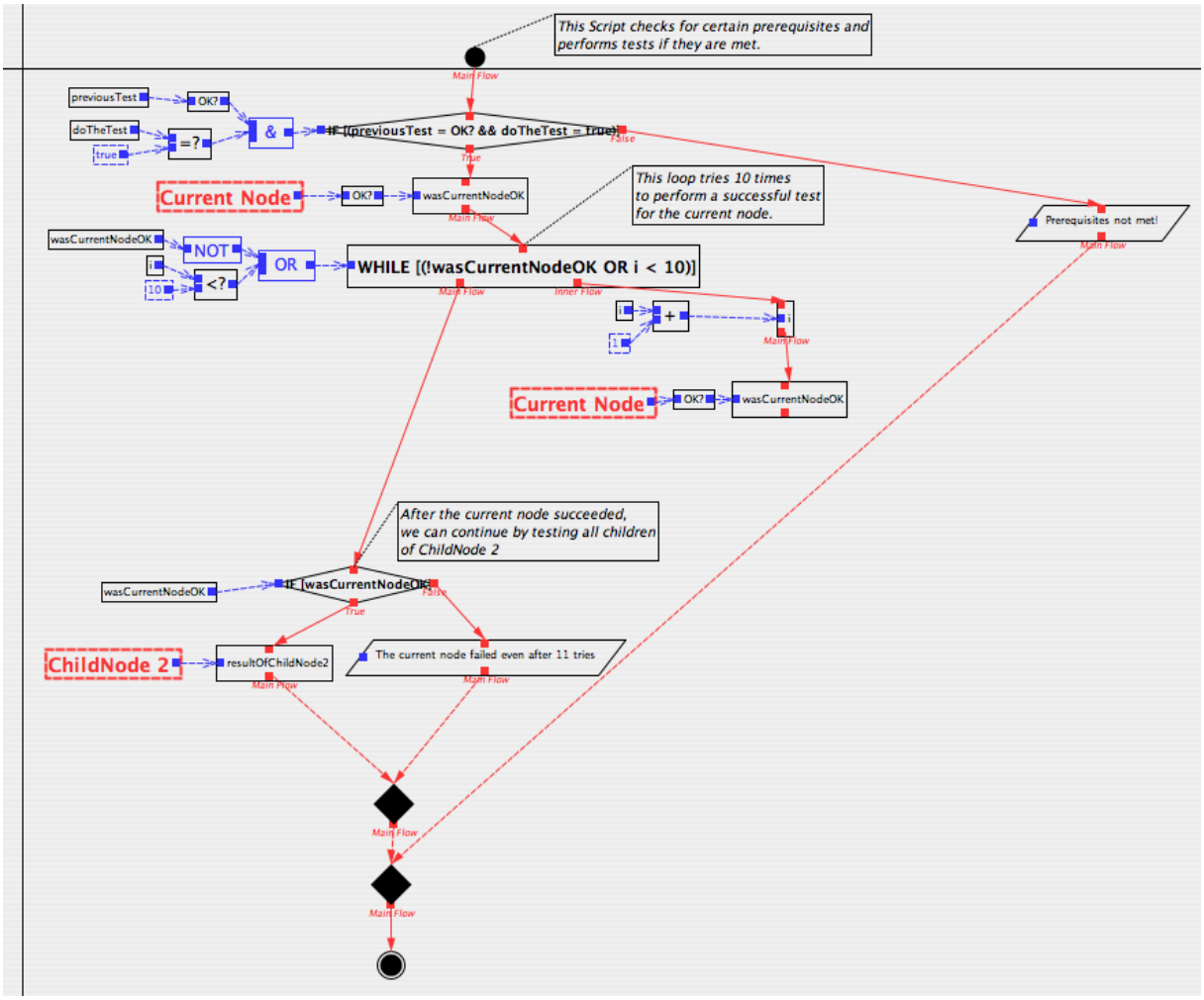


Abbildung A.1: Das Beispielskript im Flussdiagramm

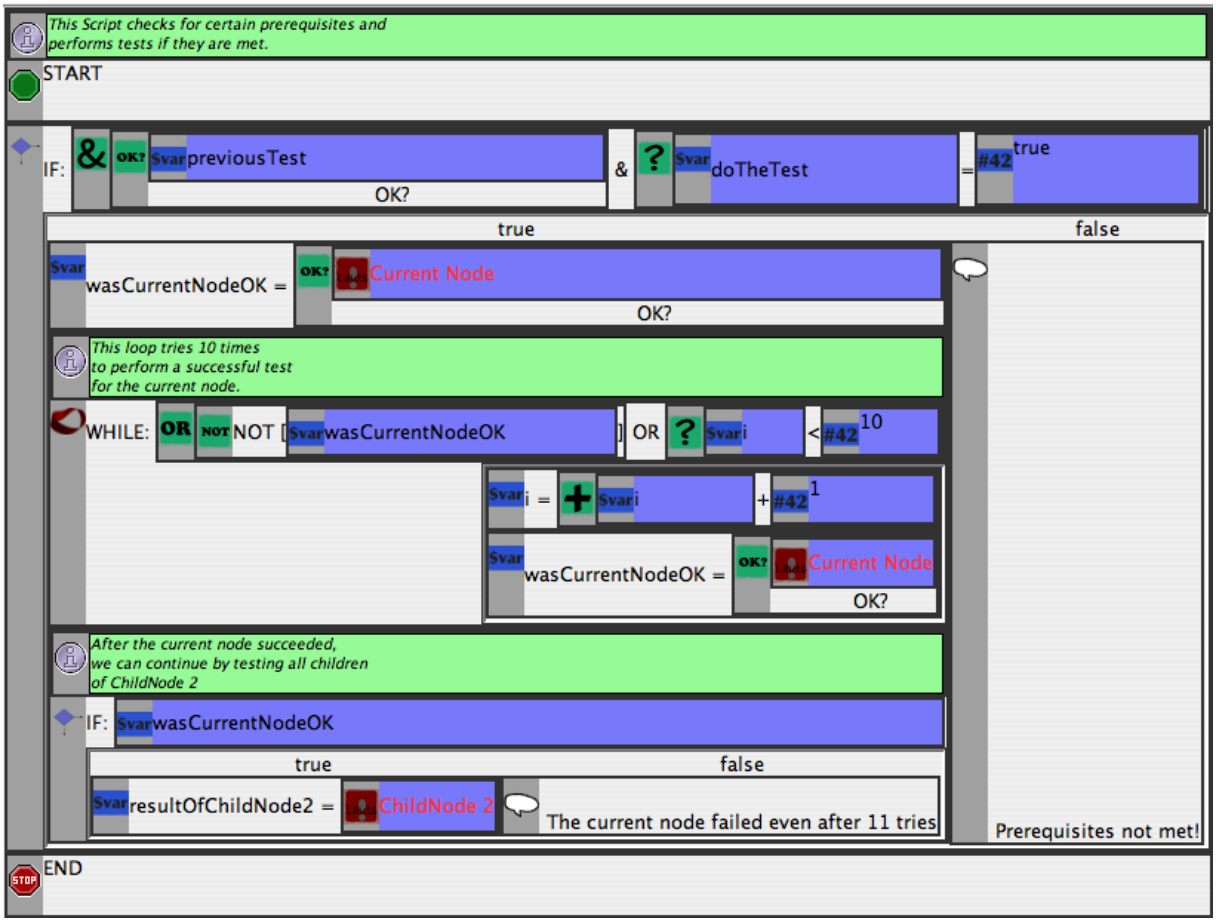


Abbildung A.2: Das Beispielskript im Blockdiagramm

Listing A.1: Der Metacode des Beispielskriptes

```
<startNode>
  <comment>This Script checks for certain prerequisites and
performs tests if they are met.</comment>
  <controlDestination>
    <selection>
      <dataSources>
        <AND>
          <dataSources>
            <resultevaluation method="WAS_OK">
              <dataSources>
                <variable name="previousTest" type="RESULT" mode="
                    READ">
                </variable>
              </dataSources>
            </resultevaluation>
            <comparison type="Equal">
              <dataSources>
                <variable name="doTheTest" type="BOOLEAN" mode="READ"
                    >
                </variable>
                <constant type="BOOLEAN" value="true">
                </constant>
              </dataSources>
            </comparison>
          </dataSources>
        </AND>
      </dataSources>
      <true>
        <variable name="wasCurrentNodeOK" type="BOOLEAN" mode="WRITE"
            >
          <dataSources>
            <resultevaluation method="WAS_OK">
              <dataSources>
                <execution name="Current_Node" id="84">
                </execution>
              </dataSources>
            </resultevaluation>
          </dataSources>
          <controlDestination>
            <loop type="WHILE">
              <comment>This loop tries 10 times
to perform a successful test
for the current node.</comment>
              <dataSources>
                <OR>
                  <dataSources>
                    <not>
                      <dataSources>
                        <variable name="wasCurrentNodeOK" type="
                            BOOLEAN" mode="READ">
```

```
47                        </variable>
48                      </dataSources>
49                    </not>
50                    <comparison type="Less">
51                      <dataSources>
52                        <variable name="i" type="LONG" mode="READ">
53                        </variable>
54                        <constant type="LONG" value="10">
55                        </constant>
56                      </dataSources>
57                    </comparison>
58                  </dataSources>
59                </OR>
60              </dataSources>
61              <inner>
62                <variable name="i" type="LONG" mode="WRITE">
63                  <dataSources>
64                    <mathop operationtype="Add" datatype="LONG">
65                      <dataSources>
66                        <variable name="i" type="LONG" mode="READ">
67                        </variable>
68                        <constant type="LONG" value="1">
69                        </constant>
70                      </dataSources>
71                    </mathop>
72                  </dataSources>
73                  <controlDestination>
74                    <variable name="wasCurrentNodeOK" type="BOOLEAN"
                        mode="WRITE">
75                      <dataSources>
76                        <resultevaluation method="WAS_OK">
77                          <dataSources>
78                            <execution name="Current_Node" id="84">
79                            </execution>
80                          </dataSources>
81                        </resultevaluation>
82                      </dataSources>
83                    </variable>
84                  </controlDestination>
85                </variable>
86              </inner>
87            </loop>
88            <selection>
89              <comment>After the current node succeeded,
90 we can continue by testing all children
91 of ChildNode 2</comment>
92              <dataSources>
93                <variable name="wasCurrentNodeOK" type="BOOLEAN" mode
                    ="READ">
94                </variable>
95              </dataSources>
```

```
96                <true>
97                  <variable name="resultOfChildNode2" type="RESULT"
                        mode="WRITE">
98                    <dataSources>
99                      <execution name="ChildNode 2" id="637">
100                     </execution>
101                   </dataSources>
102                 </variable>
103               </true>
104               <false>
105                 <output mask="The current node failed
106 even after 11 tries">
107                 </output>
108               </false>
109             </selection>
110           </controlDestination>
111         </variable>
112       </true>
113       <false>
114         <output mask="Prerequisites not met!">
115         </output>
116       </false>
117       <controlDestination>
118         <endNode>
119         </endNode>
120       </controlDestination>
121     </selection>
122   </controlDestination>
123 </startNode>
```

Listing A.2: Der erzeugte Quelltext des Beispielskriptes

```
1  if((previousTest.wasOk() && doTheTest == true)){
2      wasCurrentNodeOK = SERVITOR.startTestNode(84).wasOk();
3      /******************
4      This loop tries 10 times to perform a successful test for the
           current node.
5      ******************/
6      while((!wasCurrentNodeOK || i < 10)){
7          i = i + 1;
8          wasCurrentNodeOK = SERVITOR.startTestNode(84).wasOk();
9      }
10     /******************
11     After the current node succeeded, we can continue by testing
           all children of ChildNode 2
12     ******************/
13     if(wasCurrentNodeOK){
14         resultOfChildNode2 = SERVITOR.startTestNode(637);
15     } else{
16         SERVITOR.writeOutput("The current node failed even after 11
               tries");
17     }
18 } else{
19     SERVITOR.writeOutput("Prerequisites not met!");
20 }
```

B CD-ROM

Auf der beiliegenden CDROM befindet sich der Quellcode des visuellen Skripteditors sowie die dazugehörige Dokumentation. Desweiteren enthält die CDROM die verwendeten Internetquellen sowie eine pdf-Datei der vorliegenden Arbeit.

Dokumente	Enthält die Dateien der Diplomarbeit
Quellcode	Code des visuellen Skripteditors
Dokumentation	Dokumentation des Codes in *JavaDoc*
Quellen	Internetquellen

Literaturverzeichnis

[AH04] Andre Hagestedt, Rahman J.: *LabVIEW - Das Grundlagenbuch*. Boston : Addison-Wesley, 2004. – ISBN 3–8237–2051–8

[Bal01] Balzert, Helmut: *Lehrbuch der Software-Technik*. Heidelberg : Spektrum, Akad. Verl., 2001. – ISBN 3–8274–0480–0

[Bau05] Bauer, Christian: *Hibernate In Action*. Greenwich : Manning Publications Co., 2005. – ISBN 1–932394–15–X

[Bur01] Burke, Eric M.: *Java and XSLT*. Sebastopol : O'Reilly & Associates, Inc., 2001. – ISBN 0–596–00143–6

[Dij68] Dijkstra, Edsger W.: Go To Statement Considered Harmful. In: *Communications of the ACM* 11 (1968), März, Nr. 3, S. 147–148

[DIN07] *DIN 66 261: Sinnbilder für Struktogramme nach Nassi-Shneiderman*. http://wwwlrh.fh-bielefeld.de/IN_Prak/inprak6.htm. Version: 2007. – [Online; Stand 26. Januar 2007]

[Her03] Herrington, Jack: *Code Generation In Action*. Greenwich : Manning Publications Co., 2003. – ISBN 1–930110–97–9

[HV07] Holger Vogelsang, Peter A. H. (Hrsg.): *Handbuch Programmiersprachen: Softwareentwicklung zum Lernen und Nachschlagen*. München : Carl Hanser Verlag, 2007. – ISBN 3–446–40558–5

[Kli07] Kliesch, Sebastian: *Ermittlung einer geeigneten Skriptsprache und Entwicklung des LAUTS Scriptingsystems im SPLICE Projekt*. Hof, Hochschule für Angewandte Wissenschaften Hof, Diplomarbeit, 2007

[Kod07] Kodosky, Jeff: Is LabVIEW a general purpose programming language? In: *The VIEW* (Stand: 2007). http://zone.ni.com/devzone/cda/tut/p/id/5313

[Kön07] König, Markus: *Automatic Layout of Graphs for LOEWE Automated Testing System*. Hof, 2007. – Master's Thesis

[Man05] Mangano, Sal: *XSLT Cookbook, 2nd Edition*. Sebastopol : O'Reilly & Associates, Inc., 2005. – ISBN 0–596–00974–7

[MW06] Metsker, Steven J. ; Wake, William C.: *Design Patterns in Java*. Boston : Addison-Wesley, 2006. – ISBN 0–321–33302–0

[Wes71] Westermayer, Hans: *Programmierlogik – Programmablaufpläne*. München : Oldenbourg, 1971. – ISBN 3486388819

[Wik07] Wikipedia: *Visual programming language — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Visual_programming_language&oldid=122528374. Version: 2007. – [Online; Stand 20. April 2007]

[WMJM04] Wesley M. Johnston, J. R. Paul H. ; Millar, Richard J.: Advances in Dataflow Programming Languages. In: *ACM Computing Surveys* 36 (2004), Nr. 1. http://www.ittc.ku.edu/~rsass/rcreading/johnston04.pdf

Eidesstattliche Erklärung

gemäß §31 Abs. 7 RaPO

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.
Die Arbeit wurde nach meiner besten Kenntnis bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Hof, den 20. Juni 2007