Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Jan Strothmann

VHDL Codegenerierung aus Nassi-Shneiderman-Diagrammen

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Jan Strothmann

VDHL Codegenerierung aus Nassi-Shneiderman-Diagrammen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. rer. nat. Wolfgang Fohl
Zweitgutachter : Prof. Dr.-Ing. Bernd Schwarz

Abgegeben am 5.11.2013

Autor: Jan Strothmann
Thema der Arbeit: VHDL Codegenerierung aus Nassi-Shneiderman-Diagrammen
Stichworte: VHDL, Nassi-Shneiderman-Diagramm, Struktogramm, Java, RMI, Swing, C, Codegenerierung, Verteilte Systeme, Mehrzyklus-Datenpfad, Entwurfsmuster

Kurzzusammenfassung: Es wird eine bisher nicht beachtete Modellierungsmethode für die VHDL Codegenerierung untersucht. Im Gegensatz zu den häufig verwendeten Methoden, das Systemverhalten über mathematische Funktionen, Zustandsautomaten oder Blockschaltbilder zu beschreiben, werden Struktogramme verwendet. Es wird ein System von Ersetzungsregeln erstellt, welches ein Nassi-Shneiderman-Diagramm in einen verwendbaren Mehrzyklus-Datenpfad umwandelt. Um das entwickelte Konzept in der Praxis zu erproben, wird die Implementierungssprache Java verwendet, um sowohl einen grafischen Editor als auch einen Codegenerator zu erstellen. Die Priorität bei der Entwicklung des Editors liegt dabei auf einer einfachen Benutzbarkeit und der Fähigkeit, simultan über das Netzwerk an einem Diagramm arbeiten zu können. Dies wird durch die Verwendung von Java Swing und RMI erreicht. Die auftretenden Besonderheiten und Probleme des verteilten Systementwurfs werden analysiert und durch den Einsatz von Entwurfsmustern gelöst. Die Fähigkeit zur VHDL Codegenerierung wird durch zahlreiche Testfälle verifiziert und an verschiedenen Anwendungsfällen theoretisch als auch praktisch untersucht.

Author: Jan Strothmann
Title of the paper: VHDL code generation from Nassi-Shneiderman-Diagrams
Keywords: VHDL, Nassi-Shneiderman-Diagram, structured flowcharts, Java, RMI, Swing, C, code generation, distrubuted system, multicycle datapath

Abstract: This thesis is about VHDL code generation. In contrary to the common modeling types, like mathematical functions, state machines and block diagrams, it will use Nassi-Shneiderman-Diagramms.
A graphical editor is implemented via JAVA RMI to ensure network capability. The main feature is the ability to work on the shared diagrams simultaneously.
The theoretical part includes the analysis of the system architecture as a distributed system. The capability of generating code is revised via theoretical and practical analysis of the generator output. This output will be verified via several tests and examples.

Inhaltsverzeichnis

1. Einleitung

1.1 Modellierung von Software-Systemen

Die Benutzung von grafischen Elementen in der Darstellung von Problemen hat eine lange Tradition. Seit langer Zeit versuchen Menschen, komplizierte und textuell schlecht darstellbare Sachverhalte in verständliche Modelle zu überführen. Grafische Notationen eignen sich dazu sehr gut und ermöglichen es, mit anderen Menschen über abstrakte Vorgänge zu sprechen. Auf diese Weise können Gedankengänge verdeutlicht und eigene Fehler leichter entdeckt werden. Die grafische Form der Wissensvermittlung bietet den weiteren Vorteil, dass Algorithmen unabhängig von der konkreten Realisierung verdeutlicht werden können. Dies ermöglicht es einem weiten Personenkreis, sich über Problematiken auszutauschen. Die so geprüften Algorithmen können dann in einer Zielsprache realisiert werden. Dazu muss die Person, die den Algorithmus erstellt, keine Kenntnisse in der konkreten Implementierungssprache haben. Dies führt durch die Entkopplung von Entwurf und Realisierung zu einer besseren Arbeitsteilung.

Um eine breite Akzeptanz und weitreichende Verwendung von Darstellungsformen zu erreichen, wurden diese häufig systematisiert und standardisiert. Diese Vereinheitlichung gestattet es, mit Grafiken zu arbeiten, ohne die Bedeutung der einzelnen Elemente klären zu müssen. Dabei wurden die Darstellungen immer strukturierter, umfassender und an die neuen Programmiermodelle und Programmiertechniken angepasst. Von Programmablaufplänen, über die Nassi-Shneiderman-Diagramme bis zur modernen UML-Notation gab es eine fortlaufende Entwicklung. Dies ermöglichte es, den Fortschritt von einfachen sequenziellen Problemen zu immer komplexeren, objektorientierten und abstrakten Programmen grafisch zu begleiten. Viele dieser Diagrammtypen eignen sich hauptsächlich für die Anwendungsprogrammierung. Für die Modellierung von Hardware finden andere Formen Verwendung. Neben der Darstellung als Blockschaltbild werden für stark mathematische Probleme Modelle mit MATLAB erstellt. Zustandsautomaten und ASM-Charts werden verwendet, um das Verhalten von Systemen zu beschreiben.

1.2 Motivation zur Themenauswahl

Ausgangssituation

Heutzutage ist eine Situation erreicht, in der die Verwendung von Diagrammen allgegenwärtig ist. Für jedes technisches Problem existieren Diagramme oder spezielle Notationen, um die Lösungswege greifbar zu machen. In den verschiedenen Fachbereichen werden diese Darstellungen flächendeckend eingesetzt und dementsprechend auch von allen Beteiligten verstanden. Die einzelnen Diagrammtypen haben sich dabei durch ihren Einsatz in der Praxis bewährt und wurden im Laufe der Zeit an die Vorgaben angepasst oder durch bessere Modellierungen ersetzt. Manche allgemein verwendbare Darstellungen haben sich dabei fächerübergreifend durchgesetzt, andere blieben einzelnen Fachbereichen exklusiv vorbehalten. Dies gilt sowohl für das Verständnis als auch deren Einsatz. Dadurch existieren Barrieren bei der Zugänglichkeit. Es stellt sich die Frage, ob diese gewachsene Beschränkung nötig ist und ob allgemein gehaltene Diagramme nicht auch spezielle Anwendungsfälle lösen können.

Entwicklung eines freien Editors

Die Nassi-Shneiderman-Diagramme, auch Struktogramme genannt, sind ein gutes Beispiel für etablierte und fächerübergreifend verwendete Diagramme. Durch sie können Algorithmen beschreiben werden und sie wurden lange für die Entwicklung von Software verwendet. Durch ihren allgemeingültigen Charakter ist auch eine Verwendung für das Erstellen von funktionalen Hardwareeinheiten denkbar. Dies würde es einer breiten Menge Menschen die Möglichkeit eröffnen, mit bekannten Beschreibungen neue Lösungswege zu beschreiten.

Dieses Konzept wird in dieser Bachelorarbeit untersucht. Hierzu wird ein Konzept entwickelt, welches die Strukturen des Nassi-Shneiderman-Diagramms in Hardware überführt. Um die Praxistauglichkeit des Konzeptes unter Beweis zu stellen und es weiter zu erproben, wird ein entsprechendes Tool entwickelt. Dieses ermöglicht die grafische Erzeugung eines Diagramms und die anschließende Erzeugung von dazu passendem VHDL-Code. Der VHDL-Code kann im nächsten Schritt mit anderen Tools simuliert und schlussendlich implementiert werden. Um das Verhalten des beschriebenen Algorithmus´ schon vorher testen zu können, ist die Erstellung von C-Code integriert.

Da einfacher Zugang und Kooperation von verschiedenen Menschen bei der Entwicklung des Tools im Vordergrund steht, wird Netzwerkfähigkeit direkt in den Entwurf integriert und über Java Remote Method Invocation realisiert.

1.3 Kapitelübersicht

Grundlagen
Im Anschluss an eine kurze Chronologie der strukturierten Programmierung werden die Nassi-Shneiderman-Diagramme eingeführt. Hierzu werden deren zentrale Fähigkeiten sowie die wichtigsten Blöcke grafisch vorgestellt.

Stand der Technik
Der Stand der Technik wird zweigeteilt dargestellt: Zum einen werden die Möglichkeiten für die Erzeugung von Hardwarebeschreibungen betrachtet, zum anderen werden einige grafische Editoren, die Nassi-Shneiderman-Diagramme verwenden, kurz beschrieben.

Entwurf des verteilten Editors
Der Entwurf wird nach den Phasen des V-Modells dargestellt und begründet. Hier findet sich die Auflistung des Funktionsumfangs und der ausgelassenen Diagrammaspekte. Die Aufteilung des Gesamtsystems in Komponenten sowie der generelle Aufbau der Diagrammdatenstruktur werden ebenso beschrieben.

Implementierung der Basiskomponenten
Beginnend mit der Datenhaltung über den allgemeinen Aufbau des Codegenerators und den Editor wird hier die Implementierung aller Programmteile beschrieben. Bei der Datenhaltung liegt der Fokus auf den Zugriffsmöglichkeiten und die Darstellung der inneren Struktur. Für die Codeerzeugungskomponente wird gezeigt, wie diese und die Ersetzungsregeln aufgebaut sind. Anschließend wird die Realisierung sowie die Verwendung des Editors kurz dargestellt.

RMI als Bindeglied
Dieses Kapitel beschreibt die Zusammenarbeit der Komponenten über RMI und die Probleme, die durch den Entwurf als verteiltes System aufgetreten sind. Auch werden die Maßnahmen zum Zugriffsschutz erläutert.

Diagrammbasierte VHDL-Codeerzeugung
Dieses Kapitel beschäftigt sich erst mit der grundlegenden Abbildungsregel, nach der die Konstrukte des Diagramms in VHDL umgesetzt werden, und geht anschließend auf die mehrstufige Implementierung der Umsetzung ein.

Verifikation des erzeugten VHDL-Codes

Der Testplan, die Teststrategie, sowie die Modul-, Integrations- und Systemtests werden hier erläutert. Es wird weiterhin beschrieben, auf welche Weise die Regressionstests realisiert sind. Mit einem konkreten Beispiel wird das Testvorgehen detailliert erläutert.

Komplexe Beispiele der Codegenerierung

Anhand von drei komplexeren Beispielen werden die Fähigkeiten des Codegenerators präsentiert. Die Beispiele lösen verschiedene Aufgaben. Es handelt sich dabei um einen Wurzelalgorithmus, einen Sinusgenerator und eine Fahrstuhlsteuerung.

Ausblick

Im Ausblick finden sich die Ideen und Funktionen wieder, die es auf Grund des Zeitrahmens nicht in das fertige Programm geschafft haben. Es werden dabei Hinweise gegeben, wie diese implementiert werden könnten, oder an welchen Punkten noch Feinarbeit notwendig ist, um die entsprechende Funktionalität in Betrieb nehmen zu können.

Fazit

Es wird ein Rückblick auf die theoretische und praktische Arbeit durchgeführt. Die Fragestellung, ob sich Nassi-Shneiderman-Diagramme zur Erzeugung von VHDL-Code eignen, steht dabei im Mittelpunkt. Dabei wird überprüft, wie weit die Ziele erfüllt wurden und welcher Erkenntnisgewinn erreicht wurde.

2. Grundlagen

2.1 Das Nassi-Shneiderman-Diagramm

2.1.1 Entstehung und grundlegende Eigenschaften

Entstehungsgeschichte

Die ersten Jahre der noch jungen Informatik waren durch Programme bestimmt, die von unbedingten Sprüngen durchzogen waren. Dadurch kam es zu zahlreichen Fehlern und die Wartung von Software war sehr aufwändig. Auch die Verwendung von Flussdiagrammen für den Entwurf änderte daran wenig. Um diesen Problemen entgegenzuwirken, wurden Wege gesucht, saubere und leichter zu wartende Programme zu erzeugen. Dijkstra bewies 1968, dass auf die Verwendung von unbedingten Sprüngen verzichtet werden kann und lediglich drei Kontrollstrukturen benötigt werden[1].

Damit begann die Zeit der strukturierten Programmierung. 1972 wurde von Niklaus Wirth die Sprache Pascal eingeführt[2]. Diese war streng typisiert und zwang so zu sauberer Programmierung. In den folgenden Jahren wurde die ursprünglich für didaktische Zwecke entwickelte Sprache zu Turbopascal weiterentwickelt und produktiv eingesetzt. Später war es durch Borland Delphi sogar möglich, Oberflächen zu erzeugen.

Um die strukturierte Programmierung grafisch zu unterstützen, wurden 1973 von Isaak Nassi und Ben Shneiderman auf der SIGPLAN die Struktogramme eingeführt[3]. Diese Diagramme ermöglichten einen hierarchischen Entwurf und die damit zusammenhängende Top-Down-Programmierung. Auch die Struktogramme wurden weiterverwendet und wurden als DIN 66261 standardisiert[4]. Heutzutage liegt der Anwendungsschwerpunkt bei der Ausbildung von Informatikern.

Fähigkeiten

Die Nassi-Shneiderman-Diagramme ermöglichen es, jeden beliebigen Algorithmus abzubilden. Der Aufbau der komplexen Strukturen wird dabei durch die Verschachtelung und Verkettung der Grundblöcke erreicht. So erwächst aus den primitiven Bausteinen die Fähigkeit, auch schwierige Sachverhalte darzustellen.

Obwohl sie nicht für die automatische Codeerzeugung entworfen wurden, eignen sie sich durch das Vorhandensein einer direkten Abbildung der Diagrammkonstrukte zur Erzeugung von Quelltext für prozedurale Sprachen. Dabei müssen für bestimmte Verhaltensweisen Konventionen entwickelt werden. So ist die Deklaration von Variablen nicht vorgesehen, deshalb werden diese im Editor direkt in der Anweisung erzeugt.

2.1.2 Basiselemente des Struktogramms

Erklärung zu den Abbildungen

Nachfolgend werden die wichtigsten Konstrukte der Nassi-Shneiderman-Diagramme eingeführt. Zu diesem Zweck werden die jeweiligen in der DIN 66261 im Abschnitt 5 festgelegten Sinnbilder und Namen verwendet[4]. Die Grafiken wurden dabei von der entwickelten Software erzeugt. Für jeden Block wird dessen Sinn und eine Entsprechung in C-Code angeben.

Die Verarbeitung

Der Verarbeitungsblock (engl. imperative) enthält eine einfache Anweisung und gibt an, wie sich der Wert der Variablen in diesem Schritt ändert (Abbildung 1).

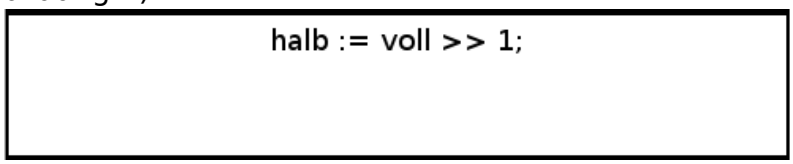


Abbildung 1: Ein Verarbeitungsblock

Er ist der Grundbaustein und der einzig sinnvolle Block auf der untersten Ebene, da er nicht für den Kontrollfluss zuständig ist, sondern eine auszuführende Aktion angibt.

Diesem Block würde der C-Code „*halb=voll >> 1;*“ entsprechen.

Die Folge

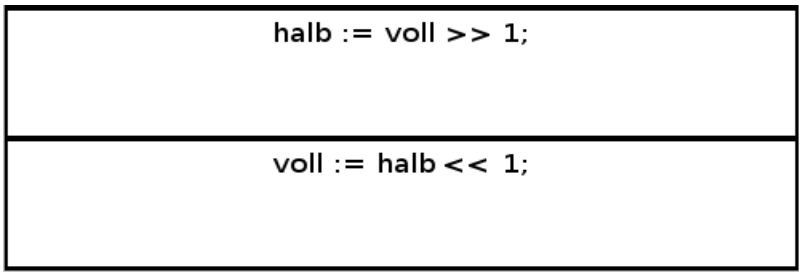


Abbildung 2: Eine Folge

Die Verkettung von mehreren Verarbeitungsblöcken (Abbildung 2) wird als Folge (engl. serial) bezeichnet und entspricht der unbedingten Abfolge von Verarbeitungen. Die Folge hat die Aufgabe, die Reihenfolge der verschiedenen Verarbeitungsblöcke festzulegen.

Diesem Block entspricht der C-Code „*halb=voll >> 1; voll= halb << 1;*“.

Die Alternative

Die Alternative (engl. selective choice) dient einer Aufspaltung des Kontrollflusses in Abhängigkeit von einer oder mehrerer Bedingungen. Hier ist die Ausführung nicht zum Zeitpunkt des Entwurfs bekannt, sondern ergibt sich aus dem Ablauf des Algorithmus.

Es gibt dabei drei verschiedene Arten der Alternative, die verschiedene Einsatzzwecke und Möglichkeiten bedienen.

Bedingte Verarbeitung (engl. monadic selective)

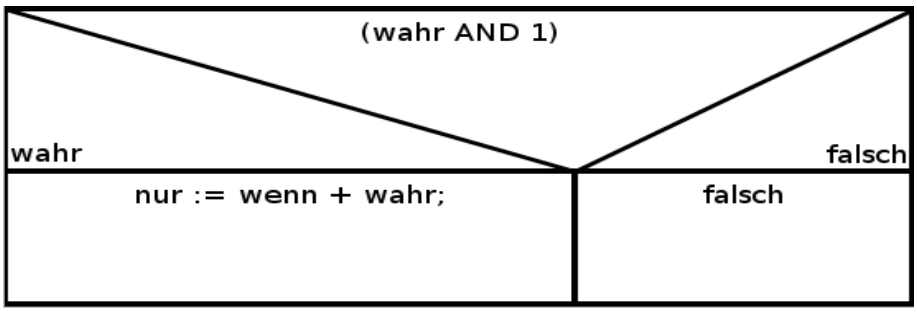


Abbildung 3: Eine bedingte Verarbeitung

Es wird der eingefasste Block nur unter der angegebenen Bedingung (Abbildung 3) ausgeführt. Ansonsten wird zum darauf folgenden Block gesprungen und das „Innenleben“ ignoriert.

Der C-Code lautet hier: *if(wahr & 1){nur=wenn + wahr;}*.

Einfache Alternative (engl. dyadic selective)

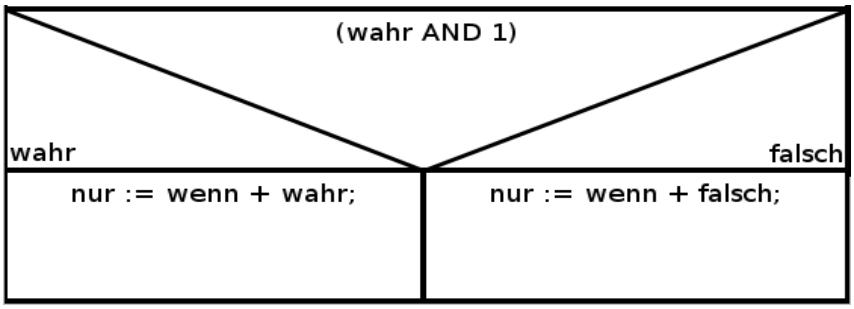


Abbildung 4: Eine einfache Alternative

Wird bei der bedingten Verarbeitung noch ein Zweig angegeben, der ausgeführt wird, wenn die Bedingung nicht zutrifft, erhält man eine einfache Alternative (Abbildung 4).

Der C-Code dafür lautet dann: *if(wahr & 1){nur=wenn + wahr;}else{nur=wenn + falsch;}*

Mehrfache Alternative (multiple exclusive selective)

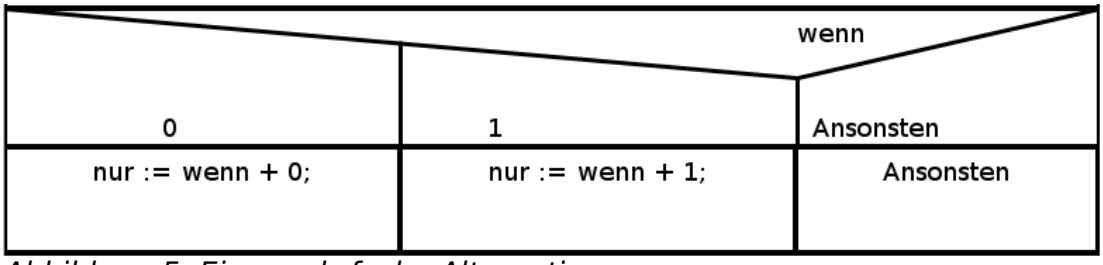


Abbildung 5: Eine mehrfache Alternative

Hier (Abbildung 5) besteht die Möglichkeit nicht nur eine, sondern beliebig viele Bedingungen anzugeben, die nacheinander geprüft werden. Es gibt auch die Möglichkeit einen Fall anzugeben, der ausgeführt wird, falls keine der Bedingungen zutrifft.

Der vergleichbare C-Code lautet *switch(wenn){case 0: nur=wenn+0; break; case 1: nur=wenn+1; break; default:}*.

Die Wiederholung

Wiederholungen (engl. Iterative) gibt es in drei verschiedenen Ausprägungen, von denen allerdings nur zwei essenziell sind. Die Wiederholung ohne Bedingung wird hier nicht betrachtet. Ihnen gemein ist, dass sie jeweils einen eingeschlossenen Schleifenrumpf besitzen, der je nach Typ und Variablenbelegung kein-, ein- oder mehrmals durchlaufen wird. Sie unterscheiden sich im Zeitpunkt der Prüfung der Bedingung zum Schleifendurchlauf. Grafisch drückt sich dies in der Position des Anwendungsblocks und der Bedingung in Schriftform aus. Allgemein sind sie sich jedoch ähnlich.

Wiederholung mit vorausgehender Bedingungsprüfung

Bei dieser Wiederholung (Abbildung 6) wird vor dem Betreten des inneren Blocks geprüft, ob dieser ausgeführt werden soll.

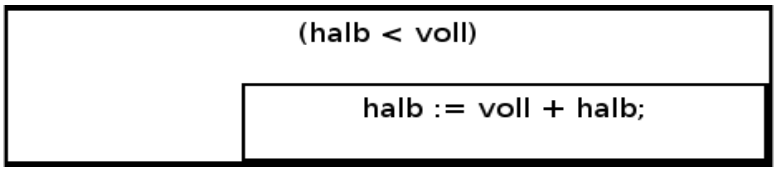


Abbildung 6: Wiederholung mit vorausgehender Bedingungsprüfung

Diese Art der Wiederholung kann auch benutzt werden, um eine Schleife eine vorher bekannte Anzahl durchlaufen zu lassen.

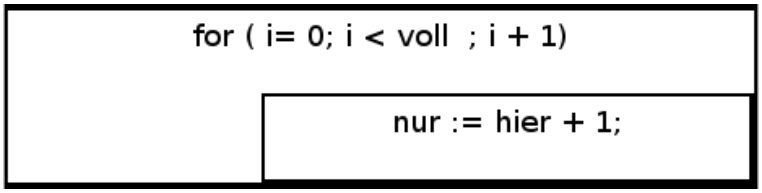


Abbildung 7: Wiederholung mit fester Anzahl an Durchläufen

Für die erste Variation (Abbildung 7) lautet der C-Code: *while(halb<voll) {halb=voll+halb};*

Für die zweite Variation: *for(i=0;i<voll;i + 1){nur=hier+1;}*

Wiederholung mit vorausgehender Bedingungsprüfung

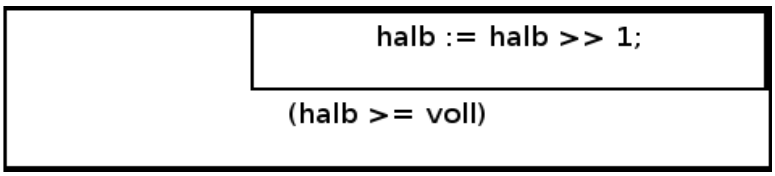


Abbildung 8: Wiederholung mit vorausgehender Bedingungsprüfung

Hier wird erst nach dem ersten Durchlauf geprüft, ob die Schleifenbedingung erfüllt ist (Abbildung 8). Ansonsten verhält es sich wie bei den anderen Wiederholungen.

C-Code: *do{halb=halb>>1;}while(halb>=voll);*

Wiederholung ohne Bedingungsprüfung

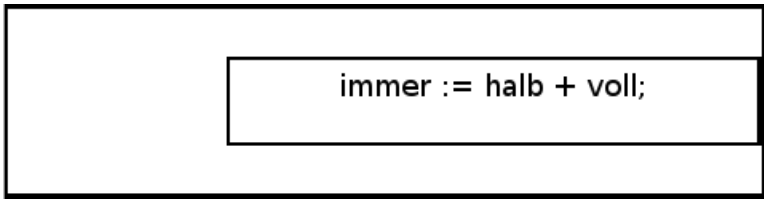


Abbildung 9: Wiederholung ohne Bedingungsprüfung

Dieses Konstrukt (Abbildung 9) führt die enthaltenen Blöcke bedingungslos und dauerhaft aus. Es ist deshalb gut dafür geeignet, reaktives Verhalten zu erzeugen.

C-Code: *while(true){halb=halb+voll);}*

Aufruf

Der Aufrufblock wird dazu benutzt, einen Unterprogramm- oder einen Systemaufruf darzustellen. Er kann genauso wie die Verarbeitung auf unterster Ebene eingesetzt werden und stellt keine Kontrollstruktur dar. Der Aufruf ist nicht in der DIN 66261 enthalten, aber dennoch wird dieser gerne eingesetzt, um den besonderen Charakter der dargestellten Funktionalität zu betonen.

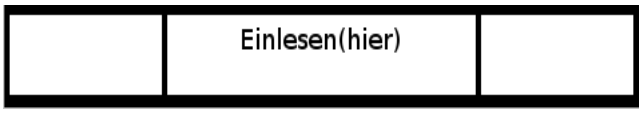


Abbildung 10: Darstellung eines Aufrufs

Häufig wird der Aufrufblock (Abbildung 10) dazu eingesetzt, Lese-, Schreib- oder andere Systemfunktionen abzubilden.

C-Code: *scanf("%d", &hier);*

3. Stand der Technik

3.1 Methoden der VHDL-Codegenerierung

3.1.1 Modellierungstechniken

Zur Erzeugung von VHDL-Quelltext werden verschiedene Modellierungsmethoden eingesetzt. Dabei wird je nach Anwendungsfall entschieden, auf welche Weise und ob überhaupt modelliert wird. Die händische Erstellung von Quelltext ist zwar aufwändig, aber auch sehr gut an die Problemstellung anzupassen. Eine weitere Methode, die direkt die einzelnen Bausteine der Schaltung verwendet, ist die grafische Erstellung von Blockschaltbildern. Andere Methoden orientieren sich an dem Verhalten, welches erzeugt werden soll. Dazu werden mathematische Modelle und Zustands- oder Flussdiagramme erstellt und durch hochkomplexe Algorithmen umgewandelt. Ein weiterer Ansatz ist die Verwendung von Programmiersprachen, die ursprünglich für andere Anwendungsfälle entwickelt wurden. Dabei können bereits erzeugte Quelltexte umgewandelt werden oder die Sprachen um Konstrukte für die Hardwareerstellung erweitert werden.

Bei den vorgestellten Lösungen handelt es sich um professionelle Software, die für die Verwendung durch Fachexperten vorgesehen ist. Diesen bieten sie einen enormen Funktionsumfang und hervorragende Möglichkeiten, die Codegenerierung zu beeinflussen. Damit kann und soll die Software nicht konkurrieren. Die Ziele der Entwicklung liegen in der Einsteigerfreundlichkeit und dem Beweis der Theorie, dass aus Nassi-Shneiderman-Diagrammen VHDL-Code mit definiertem Zeitverhalten erzeugt werden kann. Durch die Modellierung über die bekannten Struktogramme setzt die Bedienung der Software kein tiefgreifendes Fachwissen voraus. Da keine der aufgeführten Methoden Struktogramme verwendet, ist ein Vergleich des generierten Codes durch die unterschiedlichen Ansätze nicht möglich. Der Vollständigkeit halber werden die vorhandenen Produkte kurz vorgestellt. Dabei wird die Verhaltensmodellierung von Hardware nicht auf VHDL begrenzt.

3.1.2 Konkrete Produkte der VHDL-Codegenerierung

The MathWorks – MATLAB mit HDL Coder

Es werden mathematische Modelle erstellt, um daraus unter anderem synthesefähigen VHDL-Code zu erzeugen[5]. Dafür ist es notwendig, die gewünschten Funktionen in MATLAB Syntax einzugeben. Es ist notwendig, die Problemstellung mathematisch zu durchdringen und für die Eingabe umzuformen. Da dies vom Entwickler persönlich durchgeführt werden muss, setzt diese Methode hohe mathematische Fähigkeiten voraus.

The MathWorks - Simulink

Diese Software verwendet für die Modellierung von Verhalten Blockdiagramme[6]. Es werden verschiedene vorhandene Komponenten zu einem großen Ganzen kombiniert, um das gewünschte Verhalten zu beschreiben. Es ist möglich, mit MATLAB erzeugte Algorithmen in den Blöcken zu verwenden.

The MathWorks - Stateflow

Mit Hilfe von Zustands- und Flussdiagrammen werden Vorgänge beschrieben, anschließend wird daraus VHDL-Code erzeugt[7]. Die Verwendung von Flussdiagrammen lässt dabei auch Konstrukte zu, die die Nassi-Shneiderman-Diagramme verbieten.

SystemC

SystemC ist an sich keine eigene Sprache, sondern nur eine Erweiterung von C++. Durch Makros und eine neue Klassenbibliothek ist es möglich, Schaltungen zu modellieren und diese zu Netzlisten umzuformen. Sie wird hauptsächlich für die Beschleunigung von Simulationen eingesetzt[8].

Handel-C

Ähnlich wie SystemC dient eine Programmiersprache als Grundlage. In diesem Fall wird ein um die Darstellung von Kanälen und Parallelität erweiterter C-Syntax verwendet. Der erzeugte Quelltext wird anschließend in eine Hardwarebeschreibungssprache umgewandelt[9].

OpenCL for Altera FPGAs

Die Firma Altera ermöglicht es, das Verhalten von FPGAs über die Programmierschnittstelle OpenCL zu beschreiben. OpenCL dient der Verwendung von Grafikprozessoren für allgemeine Berechnungen. Die Eigenschaft, sowohl FPGAs als auch allgemein verfügbare parallele Recheneinheiten nutzen zu können, ist die Stärke dieses Ansatzes[10].

3.2 Existente Nassi-Shneiderman-Editoren

3.2.1 Auswahlkriterien für die Diagramm-Editoren

Auch heute noch werden Struktogramme benutzt, um den Ablauf von Programmen zu beschreiben. Deshalb gibt es zahlreiche kommerzielle Software, Freeware, Shareware und Open-Source-Software, die die Nassi-Shneiderman-Diagramme verwenden. Teilweise wird aus diesen Programmen direkt Code erzeugt. Ein Großteil dieser Produkte beschränkt sich jedoch auf das Erstellen, die Verarbeitung und das Drucken der Diagramme. Die meisten der Programme mit integriertem Codegenerator können dabei nur Code für Hochsprachen wie C und Pascal ausgeben. Von den untersuchten Programmen war nur ein einziges in der Lage, VHDL-Code zu erzeugen. Dies verdeutlicht, wie wenig dieser Bereich bis jetzt beachtet wurde.

3.2.2 Auflistung verbreiteter Editoren

Easycode SPX

Die Firma Easycode bietet mit dem Produkt Easycode SPX die einzige Software an, die direkt aus Nassi-Shneiderman-Diagrammen VHDL-Code erzeugen kann. Die Software hat zudem die Fähigkeit, Quellcode in einigen Programmiersprachen einzulesen und in Struktogramme umzuwandeln. Auch eignet es sich für Versionskontrolle, da für deren Integration Schnittstellen vorhanden sind. Die Ersetzungsregeln werden über Textdateien eingebunden, wodurch das Vorgehen zur Erzeugung von VHDL-Code bestimmt werden kann. Es wird mit Hilfe von Loop-Statements ein einziger kombinatorischer Prozess erzeugt. Zeitverhalten und die Möglichkeit ein reaktives System zu erstellen ist damit nicht gegeben[11].

Structorizer

Es handelt sich um ein nicht-kommerzielles Projekt, welches unter der GPL-Lizenz steht. Durch die Verwendung von JAVA ist es unter vielen Betriebssystemen lauffähig. Es unterstützt Codegenerierung und kann Pascal-Code einlesen. Diese Software wurde betrachtet, da durch die freie Lizenz eine Weiter- beziehungsweise Mitentwicklung möglich wäre. Nach einer Analyse der Software wurden dabei zwei Tatsachen deutlich: Zum einen bietet sie schlicht zu freie Eingabemasken, um daraus VHDL-Code zu erzeugen, zum anderen ist die Datenhaltung nicht sauber von der grafischen Oberfläche getrennt. Auf diese Weise ist eine Integration der gewünschten Netzwerkfähigkeit nur schwer möglich und es wurde eine von Grund auf neue Software entworfen[12].

4. Entwurf des verteilten Editors

4.1 Anforderungsdefinition

Ziel ist die Implementierung eines einsteigerfreundlichen Editors mit der Fähigkeit zur Erzeugung von VHDL- und C-Code. Es soll den Verwendern möglich sein, ihre Algorithmen zu entwerfen, ohne über Kenntnisse in den Eigenschaften und speziellen Problemen der Zielsprachen zu verfügen. Für die Analyse der Benutzeranforderungen wird ein Anwendungsfalldiagramm (Abbildung 11) verwendet. Dieses ist auf die Interaktion zwischen den Akteuren und dem System beschränkt[13] und dabei sehr allgemein gehalten.

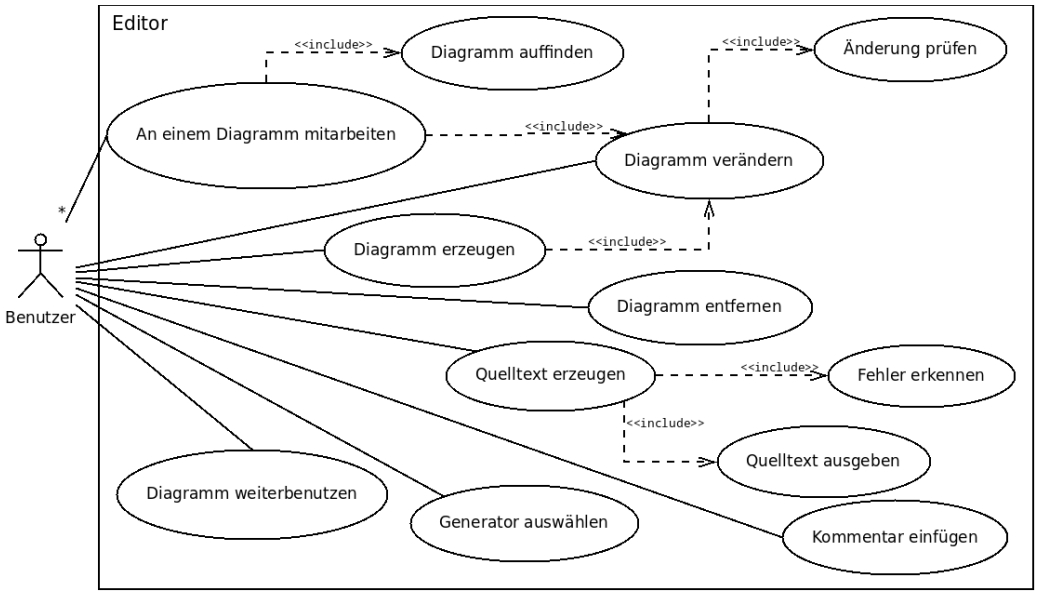


Abbildung 11: Anwendungsfalldiagramm für die Benutzerinteraktion

4.2 Funktionale Spezifikation des Gesamtsystems

4.2.1 Funktionsumfang

Die in der Anforderungsdefinition beschriebenen Anwendungsfälle müssen nun in eine konkrete Beschreibung des Funktionsumfangs umgewandelt werden.

Das Softwaresystem wird in der Programmiersprache Java implementiert. Um den Anwendungsfall der Mitarbeit an einem Diagramm zu entsprechen, wird durch die Verwendung von RMI Netzwerkfähigkeit erreicht.

Die Verwendung der Software beginnt mit dem Aufruf des Programmstarters. Dieser ermöglicht die Auswahl der Betriebsart. Es stehen dabei der Betrieb als Editor, Datenhaltung und Editor, Generator, Datenhaltung und „Alles zugleich" zur Verfügung. In Abhängigkeit von der Auswahl des Benutzers wird die entsprechende Menge an Teilprogrammen gestartet. In einem Systemzusammenhang sind dabei beliebig viele Editoren und beliebig viele, auch verschiedene Codegeneratoren möglich. Für alle Teilsysteme existiert eine grafische Umsetzung in Swing, um benutzerfreundlich zu sein.

Das für den Benutzer wichtigste Teilprogramm ist der grafische Editor. Die Struktogramme werden der DIN 66261 entsprechend dargestellt. Mit Hilfe eines Kontextmenüs ist es möglich, das Diagramm zu ändern, es abzuspeichern, die Codegenerierung anzustoßen, die Betriebsart der Codegeneratoren zu wählen und den selektierten Block zu verändern. Um dem Benutzer die Auswahl des richtigen Diagrammblocks zu ermöglichen, wird diese durch einen Hover-Effekt unterstützt. Das Diagramm kann durch das Hinzufügen neuer Blöcke vor und hinter dem aktuellen Knoten und das Löschen des selbigen verändert werden. Die Änderungsmöglichkeiten an dem Diagrammblock selbst richten sich nach dessen Typ. So kann entweder die enthaltene Bedingung oder Aktion verändert werden. Zusätzlich ist die Möglichkeit gegeben, einen Kommentar einzufügen. Die verfügbaren Variablen werden im unteren Teil angezeigt, neue Variablen werden durch die erstmalige Verwendung hinzugefügt. Die Variablen verfügen dabei über die drei Typen Integer, Character und Boolean. Die Eingaben der Benutzer werden auf ihre Richtigkeit überprüft und es werden Hinweise gegeben, wo der Fehler liegt.

Die Datenhaltung ermöglicht es dem Benutzer, diese und alle angeschlossenen Editoren und Codegeneratoren grafisch zu beenden und die für die Kommunikation verwendete IP-Adresse abzulesen.

Für den Codegenerator existiert ebenfalls eine Oberfläche. Diese stellt den durch den letzten Aufruf des Generators erzeugten Quelltext formatiert dar. Der Vorgang der Codeerzeugung kann dabei direkt aus dem Generator Fenster, über den Editor, als auch automatisch bei Veränderungen erfolgen. Über das Kontextmenü lässt sich dieser Vorgang auslösen, der Quelltext abspeichern, der Inhalt der Zwischenablage abrufen und der verwendete Codegenerator auswählen. Zur Auswahl stehen Generatoren für die Erzeugung von VHDL- und C-Code.

Der durch den Generator erstellte Code setzt dabei das in dem Diagramm beschriebene Verhalten in Code um. Die Namen der im VHDL-Code erzeugten Register und die verwendeten Signale richten sich dabei nach den Vorgaben des Diagramms. Da als Umwandlungsziel ein Mehrzyklus-Datenpfad verwendet wird, wird eine Zustandsmaschine erzeugt. Die Zustände und Zustandsübergänge setzen dabei den Programmfluss um. Um eine leichte Übersicht zu ermöglichen, sind die Zustände mit den Kommentaren aus dem Diagramm versehen. Falls keine Kommentare angegeben wurden, wird das Verhalten des Blocks in Pascal-Notation beschrieben. Bei der Erzeugung des C-Codes wird in Abhängigkeit von dem Diagrammnamen entweder eine main -Funktion oder eine Unterfunktion erzeugt. Bei dem Speichervorgang wird eine passende Header-Datei mitgespeichert.

Die Fehlermeldungen und für die Anmeldung über das Netzwerk nötigen Fenster werden von allen Teilprogrammen verwendet. Es existiert eine Eingabemaske für die IP-Adresse und ein Auswahlbildschirm für die Diagramme der Datenhaltung. Der Speichervorgang schlägt automatisch die richtigen Endungen vor, diese lautet für die Diagramme „*.neted“.

4.2.2 Einschränkungen in der Modellierung

Ausgelassene Blöcke

Einige der in den Nassi-Shneiderman-Diagrammen verwendeten Blöcke sind nicht zwingend erforderlich und wurden deshalb weggelassen.

- **Abbruchanweisung:** Die Abbruchanweisung entspricht das in vielen Programmiersprachen vorhandene „break“. Durch sie findet ein Sprung aus der aktuell laufenden Wiederholung statt. Dies dient nur der Beschleunigung und ist nicht erforderlich.
- **Parallelverarbeitung:** Es gibt in Nassi-Shneiderman-Diagrammen die Möglichkeit, explizite Parallelität anzugeben. Da das Hauptaugenmerk auf der Realisierung von sequenziellen Abläufen liegt und es bessere Möglichkeiten zur Modellierung von parallelen Abläufen gibt, wird auf diesen Block verzichtet.
- **Block:** Durch einen Block ist es möglich, Unterdiagramme zu referenzieren. Für die Modellierung einer Anwendung ist dies nötig, für die in VHDL-Entitäten umgesetzten Algorithmen nicht.

Einschränkung von Aktionen und Bedingungen

Um die Abarbeitung von Aktionen und Bedingen direkt und in einem Takt umsetzen zu können, werden diese eingeschränkt.

Dabei lautet die verwendete Schemata für die Aktionen:

“*Zielfeld := Operand1 Operation Operand2*“

“*Zielfeld := Operand1 Operation*“

“*Zielfeld := Operation Operand2*“

Das Schema für die Bedingungen:

“*(Operand1 Operation Operand2)*“

Zulässig für die in den Aktionen und Bedingungen verwendeten Operationen sind dabei folgende Grundoperationen:

*+, -, *, >>, <<, ==, !=, >, <, <=, >=, AND, OR, XOR, NOT*

Die Mehrfachauswahl

In der Mehrfachauswahl kann der folgende Pfad nur nach verschiedenen Werten und nicht nach Bedingungen ausgewählt werden. Da der erste Fall weitaus häufiger verwendet wird, stellt dies keine große Einschränkung dar.

4.3 Technischer Systementwurf

4.3.1 Einsatz des Model-View-Controller-Entwurfsmusters

Die Software wird in drei Komponenten zerlegt, den Diagrammeditor, die Datenhaltung und den Codegenerator (Abbildung 12). Die Aufteilung richtet sich nach dem Model-View-Controller-Entwurfsmuster[14]. Dieses spaltet ein System in ein Modell, die Steuerung des selbigen und eine beobachtende Sicht auf. Das Modell enthält alle Daten des Systems und die Steuerung führt als einziges Veränderungen an diesem durch. Es können dabei verschiedene Sichten gleichzeitig aktiv sein und den Zustand des Modells wiedergeben.

Im Entwurf des verteilten Editors wird die Datenhaltung als Modell und der Codegenerator als Sicht aufgefasst. Bei dem Editor handelt es sich um eine Kombination aus Sicht und Steuerung, da er das Diagramm manipulieren kann.

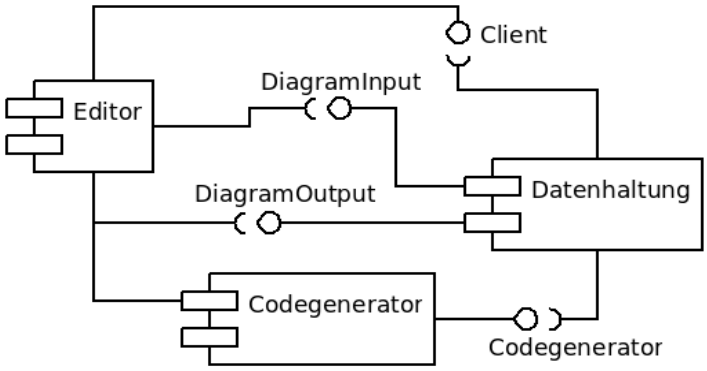


Abbildung 12: Komponentendiagramm des Gesamtsystems

Die verschiedenen Rollen wirken sich dabei in der Art, wie auf die Datenhaltung zugegriffen wird, aus. Die Codegeneratoren verfügen nur über einen lesenden Zugriff, die Editoren zusätzlich über Schreibrechte. Das Interface für die Manipulation heißt *DiagramInput*, gelesen wird über das Interface *DiagramOutput*.

Die Zusammenarbeit der Komponenten wird über JAVA RMI realisiert. Dies ermöglicht es, zur Laufzeit Komponenten hinzuzufügen und zu entfernen. Da sowohl Editoren als auch Codegeneratoren auf die Datenhaltung zugreifen und diese die essentiellen Informationen enthält, steht diese im Zentrum des Systems. Das Stattfinden von Änderungen am Diagramm und der Anstoß zum Start des Codegenerators werden über die Schnittstellen *Client* und *Codegenerator* übertragen.

4.3.2 Die Datenstruktur als ungerichteter Baum

Nachdem die Software in einzelne Komponenten zerlegt wurde, stellt sich nun die Frage nach der internen Darstellung der Diagrammdaten. Diese ist für den verteilten Editor und die Erzeugung von Code essentiell, deshalb erfordert der Entwurf dieser Struktur eine präzise Planung.

Für den internen Aufbau existieren grob zwei Möglichkeiten. Zum einen die Darstellung als Kontrollflussgraph, zum anderen die Übernahme der Diagrammstruktur als ungerichteter Baum. In der Darstellung (Abbildung 13) befindet sich auf der linken Seite der Kontrollfluss in schematisierten Diagrammblöcken, rechts enthalten diese die Baumstruktur.

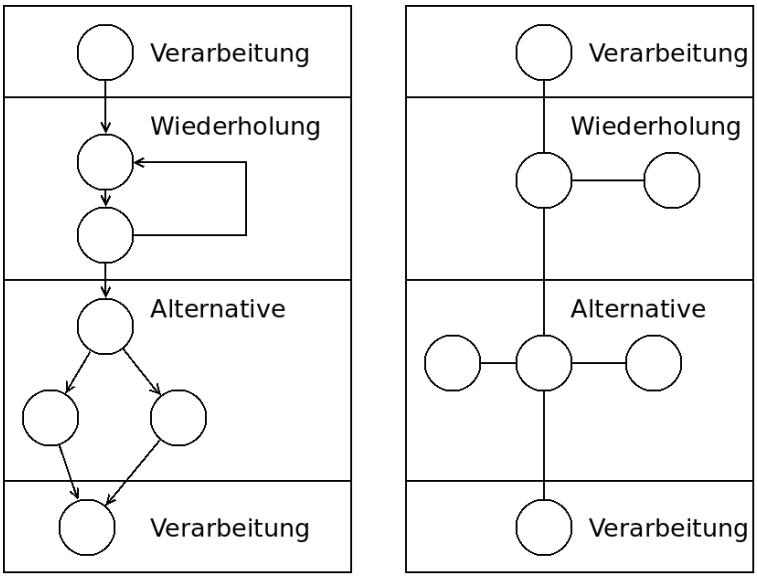


Abbildung 13: Vergleich der möglichen Datenstrukturen

Für eine an einen Kontrollflussgraphen angelehnte Darstellung spricht, dass sehr wenige und ähnliche Typen von Knoten verwendet werden können. Gravierender Nachteil dieser Struktur ist, dass die Konstrukte der Struktogramme hier aus mehren Knoten zusammengesetzt sind. Dies sorgt dafür, dass bei jedem Zugriff mehrere Knoten manipuliert werden müssen und diese nicht eindeutig einem Diagrammblock zugeordnet werden können.

Aus diesen Grund wird ein ungerichteter Baum verwendet und die Struktur der Diagramme direkt übernommen. Es müssen dadurch zwar deutlich mehr und auch komplexere Knotentypen erzeugt werden, durch die klare Zuordnung sind Veränderungen an den Daten aber auf einfachere Weise möglich. Auch ist keine zentrale Instanz notwendig, da die Struktur der Daten ihrer Repräsentation entspricht. Bei der Verwendung eines Kontrollflussgraphen ist diese für die Veränderung der Daten und der Aufrechterhaltung der Konsistenz erforderlich.

4.3.3 Zugriff auf den Diagrammbaum

Die zweite wichtige Fragestellung bei der Struktur der Datenhaltung ist, ob der Zugriff auf die einzelnen Knoten des Diagramms direkt oder indirekt stattfindet. Da RMI für die Realisierung verwendet wird, existiert immer ein Proxy. Dieser wird dann aber als Referenz auf die ausgewählte Zugriffsmöglichkeit verwendet. Nachfolgend werden die zwei Ansätze an symbolischen Klassendiagrammen verdeutlicht.

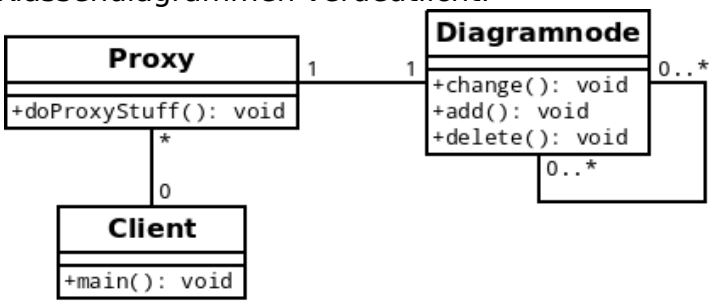


Abbildung 14: Symbolisches Klassendiagramm direkter Zugriff

Der direkte Ansatz (Abbildung 14) bietet durch parallelen Zugriff Vorteile bei der Performanz. Bei Änderungen ist nur der aktuelle Knoten betroffen und es existiert kein Flaschenhals, den alle Benutzeranfragen passieren müssen. Betreffen die Aktionen jedoch alle Benutzer oder das gesamte Diagramm, wird hieraus ein Nachteil.

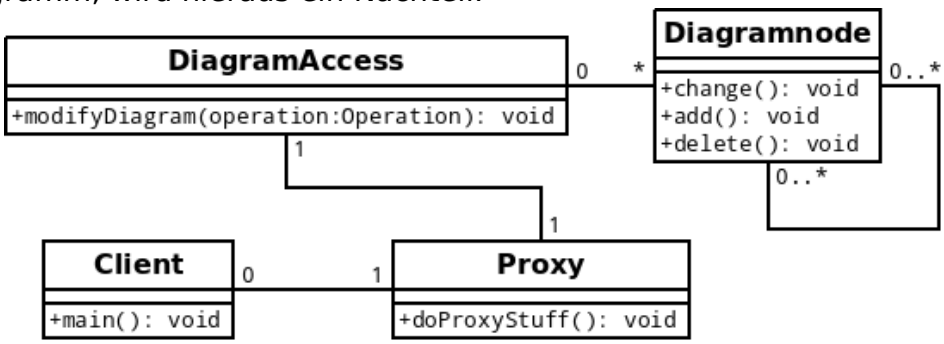


Abbildung 15: Symbolisches Klassendiagramm indirekter Zugriff

Dies ist der große Vorteil des indirekten Ansatzes (Abbildung 15). Durch einen zentralen Punkt des Anwenderzugriffs ist die Konsistenz leicht aufrechtzuerhalten. Damit schöpft dieser Entwurf die über RMI mögliche Parallelität nicht aus. Um einen interessanten Entwurf und eine auch im Falle vieler Benutzer gute Performanz zu erhalten, findet der Zugriff auf die einzelnen Knoten direkt statt.

5. Implementierung der Basiskomponenten

5.1 Die Diagrammdatenhaltung im Detail

Die Datenhaltung spielt eine zentrale Rolle für den verteilten Diagrammeditor. Sie enthält nicht nur die Diagrammdaten, sondern koordiniert auch den Zugriff auf diese. Durch die verschiedenen administrativen Aufgaben, die dieses Teilsystem realisiert, ist eine interne Dreiteilung notwendig. Zum einen werden so die Verwaltungsaufgaben von den eigentlichen Diagrammdaten, zum anderen der Zugriff auf die Diagrammliste von der Administration eines einzelnen Diagramms getrennt.

5.1.1 Verwaltung der Diagrammdaten

Die Diagrammlistenverwaltung

Die Diagrammlistenverwaltung administriert die Liste der Diagrammzugriffe und ermöglicht so das Hinzufügen neuer sowie das Löschen und die Auflistung der vorhandenen Diagramme. Da sie die zentrale Instanz der Verwaltung darstellt, wird mit Hilfe von RMI ein Singleton Entwurfsmuster realisiert[14]. Die Implementierung dieser Verwaltungsstruktur befindet sich im Paket *DataSystem.controller*. Die Klasse *DataControllerImpl* implementiert die Schnittstellendefinition *DataController.*

Das Diagrammzugriffssystem

Aufgabe des Diagrammzugriffssystems ist es, Veränderungen an alle angemeldeten Editoren und Codegeneratoren zu propagieren und den Zugriff auf die Diagrammdatenstruktur zu koordinieren. Da sie für die Informationsweiterleitung und Zugriffskontrolle verwendet wird, enthält sie eine Liste dieser Komponenten. Dadurch wird eine Vermischung von

Benutzerdaten und anderen für das Diagramm unnützen Informationen verhindert. Durch ihren lokalen Zugriff auf das Diagramm ermöglicht sie das Beziehen einer serialisierten Kopie und so die Speicherfunktion. Eine weitere Aufgabe ist es, bei der Abschaltung der Datenhaltung die angeschlossenen Editoren und Codegeneratoren über diesen Vorgang zu informieren. Das Interface *DiagrammHandler* sowie die Implementierung *DiagrammHandlerImpl* befinden sich im Paket *DataSystem.handler.*

5.1.2 Implementierung der Diagrammdatenstruktur

Realisierung der Baumstruktur

Bei der Diagrammdatenstruktur handelt es sich um die interne Datendarstellung des Diagramms. Dem Entwurf aus Kapitel 25 entsprechend, repräsentieren die Knotentypen direkt die einzelnen Diagrammblöcke und sind als ungerichteter Baum angeordnet. Die Reihenfolge der Diagrammblöcke wird durch das Auszeichnen eines Knotens als Wurzel und die lineare Verbindung der Folgeknoten über Kanten realisiert. Unterstrukturen werden als zusätzliche Knoten angefügt. Anders als in einem gerichteten Baum, werden die Knoten durch ungerichtete Kanten verbunden. Dies ermöglicht es, sowohl die Kindknoten als auch den Elternknoten zu erreichen, ohne von dem Wurzelknoten abwärts suchen zu müssen. Die Baumstruktur ist wichtig, da der Codegenerator so durch Traversierung alle Knoten erreichen kann. Die Schnittstelle für die Diagrammdatenstruktur befindet sich im Paket *diagrammdata ,*die Implementierung im Unterverzeichnis *node.*

Einsatz des Zustand-Entwurfsmusters

Die Knotentypen enthalten die für sie benötigten Daten und stellen Operationen für deren Manipulation bereit. Manche dieser Informationen und Methoden liegen in allen, manche nur in wenigen Knotentypen vor. Um eine einheitliche Schnittstellen für die verschiedenen Bestandteile des Diagramms zu erhalten, müssen die Unterschiede zwischen den Knotentypen verborgen werden.

Das Vorgehen ist dabei an das GOF-Zustand-Entwurfsmuster angelehnt[14]. Dieses dient dem Aufbau einer Zustandsmaschine, lässt sich durch die ähnliche Problematik aber gut auf die Diagrammstruktur übertragen. In dem Entwurfsmuster wird ein Interface erstellt, welches alle Zustandsübergänge als Methodenköpfe enthält. Diese Schnittstelle wird von einer abstrakten Klasse implementiert, die für alle Methoden eine funktionslosen Rumpf bereitstellt. Die konkreten Klassen der Zustände erweitern dann diese und es müssen nur die tatsächlich verwendeten Methoden geschrieben werden.

Für die Diagrammdatenstruktur heißt dies, dass alle Knotentypen die gemeinsame, abstrakte Oberklasse *NodeImpl* erweitern und nur die benötigten Methoden implementieren. Da das Verhalten aller Knotentypen von nur einem Interface abhängt, wird ähnliches Verhalten zusammengefasst. Die Schnittstelle wäre sonst zu umfangreich. Eine Maßnahme, um die Anzahl der Methoden zu begrenzen ist, den Zugriff auf die Schleifenrümpfe aller Wiederholungen und die Pfade der Alternativen über die Methode *getSubDiagram()* zu realisieren. Das Auslesen und die Veränderung der in diesen Diagrammblöcken verwendeten Bedingungen wurden über die Methoden *getCondition()* und *setCondition()* ebenfalls zusammengefasst. Neben den Methoden für die Manipulation der Knoten enthält das Interface auch die Navigation durch den Graphen. Diese ist über die Methoden *getRoot()*, *getParent()* und *getChild()* umgesetzt. Das Anhängen neuer Kindknoten ist so gestaltet, dass der Methode ein Knoten vom gewünschten Typ übergeben wird. Auf diese Weise muss nicht für jeden neuen Knotentyp das Interface erweitert werden und die Anzahl der Methoden kann weiter reduziert werden. Um für die Verwendung des Model-View-Controller-Entwurfsmuster Lese- und Schreibzugriffe zu bieten, setzt sich das Interface *Diagramm* aus den Schnittstellen *DiagramInput* und *DiagramOutput* zusammen (Abbildung 16).

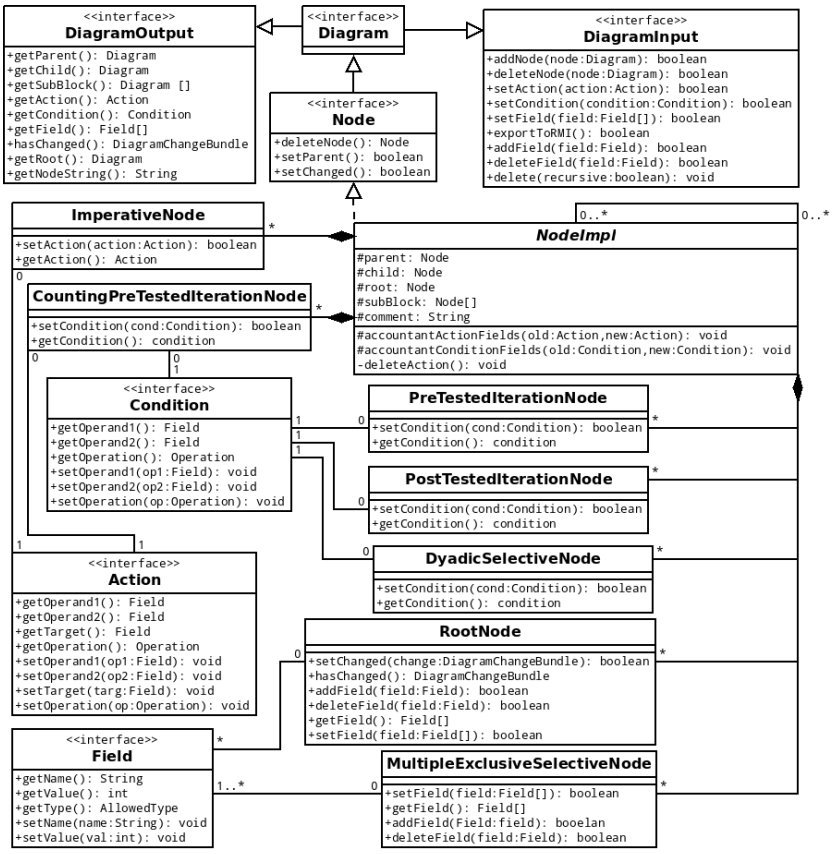


Abbildung 16:Klassendiagramm der Datenhaltung

Interner Aufbau der Knoten

Bis jetzt wurde nur die Kooperation und der Zugriff auf die Knoten betrachtet. Die über den Editor eingegebenen Daten müssen jedoch auch in den Knoten gespeichert werden (Tabelle 1). Zu diesem Zweck wurden die Möglichkeiten der Nassi-Shneiderman-Diagramme betrachtet und die notwendigen Datenstrukturen abstrahiert. Für die Steuerung des Kontrollflusses ist es nötig, unterschiedliche Pfade auswählen und den Abbruch von Schleifen bestimmen zu können. Die dazu verwendeten Bedingungen werden durch das Interface *Condition* beschrieben. Die Schnittstelle *Action* und ihre Implementierung ermöglicht die Erhöhung der Laufvariable in der Zählschleife und den Rechenschritt in einem Verarbeitungsblock.

Die beiden Strukturen bestehen ihrerseits aus einer Operation und mindestens zwei Feldern. Bei einer Implementierung der Schnittstelle *Operation* handelt es sich entweder um einen Vergleichsoperator oder eine Rechenoperation. Die drei Klassen *VariableField*, *ConstantField* und *DummyField* implementieren das Interface *Field.* Der Datentyp eines Feldes wird über das Enum *AllowedType* festgelegt. Außer in den Rechnungen und Bedingungen werden die Felder für die Identifikation der Ausführungspfade in der Mehrfachauswahl und die Auswahlvariable verwendet. Auch die Aufrufknoten verwenden sie, um ein- oder ausgegebene Daten zu spezifizieren. Die in einem Diagramm verwendeten Felder werden im Wurzelknoten verwaltet. Eine Referenzzählung überprüft, wie häufig ein Feld benutzt wird und löscht es, wenn es nicht mehr benötigt wird.

Knotentyp	**Bedingung**	**Aktion**	**Felder**
Diagrammwurzel	-	-	X
Verarbeitung	-	X	-
Bedingte Verarbeitung	X	-	-
Einfache Alternative	X	-	-
Mehrfach Alternative	-	-	X
Wiederholung mit vorausgehender Prüfung	X	-	-
Wiederholung mit nachfolgende Prüfung	X	-	-
Wiederholung Zählschleife	X	X	-
Aufruf	-	-	X

Tabelle 1: In den Knoten gespeicherte Informationen

5.2 Die Codeerzeugungskomponente

5.2.1 Interner Aufbau des Codegenerators

Der Codegenerator dient der Erzeugung von Quelltexten für die verschiedenen Programmiersprachen. Dem Entwurfsmuster des Gesamtsystems folgend, greift er als Sicht lesend auf die Datenhaltung zu und wird von dieser über die Änderungen informiert. Der interne Aufbau wird von der Lösung der gegebenen Teilaufgaben bestimmt. Bei diesen handelt es sich um die Kommunikation mit der Datenhaltung, die Erzeugung des Quelltextes und der Ausgabe der Ergebnisse. Dabei ist die Möglichkeit gegeben, sowohl die Erzeugungs-, als auch die Ausgabekomponente auszutauschen, um das Verhalten anzupassen.

In der Implementierung äußert sich dies, indem die Hauptklasse *CodgeneratorImpl* die Schnittstelle *Codegenerator* implementiert und die Änderungsinformationen von der Datenhaltung entgegennimmt. Diese startet daraufhin die hinter dem Interface *ReplacingRule* verborgene Ersetzungsregel. Bei dieser handelt es sich entweder um eine *CReplacingRule* oder *VHDLReplacingRule*. Die Ausgabe wird über die Schnittstelle *CodegeneratorOutput* definiert und von einer grafischen und einer grundlegenden Konsolenausgabe implementiert. Die Ausgabeklasse wird zum Start des Codegenerators ausgewählt, die Ersetzungsregeln können zur Laufzeit ausgewechselt werden. Alle Klassen und Interfaces des Codegenerators befinden sich in dem Paket C*odegenerator* und dessen Unterverzeichnissen.

5.2.2 Implementierung der Ersetzungsregel

Allgemeiner Aufbau und Vorgehensweise

Die Ersetzungsregel dient der Überführung der Diagrammdaten in Quelltext. Zu diesem Zweck traversiert sie von dem Wurzelknoten ausgehend durch den Diagrammbaum. Da dieses Verhalten für alle Ersetzungsregeln gleich ist, wird es in der abstrakten Klasse *AbstractReplacingRule* realisiert. Die Traversierung durch den Baum ist dabei rekursiv implementiert. Aus diesem Grund müssen für alle Knotentypen die Ersetzungsregeln als Platzhalter in der abstrakten Klasse vorliegen. Die Kommunikation mit dem Codegenerator findet über verschiedene Wege statt. So wird mit dem Aufruf von *replace(dia,sb,sem)* die Rekursion gestartet. Treten Fehler während der Umwandlung auf, führt dies durch eine *ErrorCodegeneratorException* zum sofortigen Abbruch des Vorgangs. Wenn während der Rekursion der mit der *CodegeneratorImpl* geteilte Semaphor anzeigt, dass Änderungen stattgefunden haben, beendet sich die Rekursion mit einer *AbortCodegeneratorException*.

Implementierung der C-Codegenerierung

Um die grundsätzliche Verhaltensweise der Ersetzungsregel an einer konkreten Implementierung zu demonstrieren, wird die Erzeugung des C-Codes betrachtet. Diese ist deutlich weniger aufwändig als die des VHDL-Codes und eignet sich somit besser für einen Überblick. Im ersten Schritt wird die *CReplacingRule* von der *CodegeneratorImpl* gestartet. Dazu erhält diese die Referenzen auf den Wurzelknoten, den zu befüllenden *StringBuilder* und den Steuerungs-Semaphor des Editors. Nun beginnt der von der *AbstractReplacingRule* geerbte rekursive Abstieg (Abbildung 17). Bei diesem wird jeder Knoten exakt einmal besucht, was dem Verhalten eines Einphasen-Compilers entspricht[15].

```
if (dia != null) {
    if (dia instanceof Imperative) {
        replaceImperative(dia);
    } else if (dia instanceof DyadicSelective) {
        replaceDyadicSelective(dia);
    } else if (dia instanceof PostTestedIteration) {
        replacePostTestedIteration(dia);
    } else if (dia instanceof ContinousIteration) {
        replaceContinousIteration(dia);
    } else if (dia instanceof PreTestedIteration) {
        replacePreTestedIterationWhile(dia);
    } else if (dia instanceof Root) {
        replaceRoot(dia);
    }
    if (this.checkChanged()) {
        throw new AbortCodegeneratorException();
    }
    this.recursion(dia.getChild());
}
```

Abbildung 17: Auswahl der Ersetzungsregel

Im ersten Schritt wird die Klasse des aktuellen Diagrammobjektes über den instanceof-Operator bestimmt, um dessen spezifische Ersetzungsregel auszuwählen. Der markierte Knoten wird dann in der entsprechenden Methode verarbeitet. Handelt es sich bei dem Knoten um eine Alternative oder Wiederholung, wird in dieser die Rekursion für den Rumpf aufgerufen. Nach der Verarbeitung des aktuellen Knotens wird der Steuerungs-Semaphor auf Veränderungen überprüft. Sind diese aufgetreten, wird der Vorgang abgebrochen. Hat sich das Diagramm nicht geändert, wird der Verkettung entsprechend der Kindknoten umgesetzt. Wenn es keinen Kindknoten mehr gibt, wird die Rekursion beendet. Während dieses Vorgehen für alle Generatoren identisch ist, werden die Ersetzungsregeln für die Diagrammblöcke in der konkreten Klasse implementiert.

Die Ersetzung der Knoteninformationen durch C-Code entspricht dabei dem in Kapitel 2.1.2 angegebenen Schema. Die nicht beschriebenen Diagrammblöcke sind der Wurzel-, der Schreib- und der Parameterknoten. Der Schreibknoten wird durch *printf()* ersetzt. Der Wurzelknoten bestimmt durch den Namen, ob es sich um die main- oder eine gewöhnliche Funktion handelt. Das Vorhandensein eines Parameterblocks mit Feldern führt dazu, dass die Variablen in dem Funktionskopf deklariert oder im Falle der *main()* an der aktuellen Position eingelesen werden.

5.3 Realisierung des Grafiksubsystems

5.3.1 Grundlegender Aufbau des grafischen Editors

Zur Darstellung des Diagramms wird die in Java enthaltene Standardoberfläche Swing eingesetzt. Diese ist für die Erzeugung einer relativ geordneten Oberfläche mit einem kleinem interaktivem Anteil ausreichend. Die einzelnen grafischen Elemente, die für den Aufbau einer Oberfläche verwendet werden können (Tabelle 2), stellt Swing auf einer hohen Abstraktionsebene zur Verfügung. Auf diese Weise ist es möglich, relativ schnell die benötigten Strukturen zu erzeugen und anzupassen. Swing baut dabei auf der grundlegenden API AWT auf[16]. Folglich bietet diese eine höhere Performanz und mehr Möglichkeiten bei der Darstellung von Bildschirminhalten. Beides ist jedoch nicht nötig und der Vorteil der einfachen Handhabung von Swing überwiegt. Zur Realisierung der Oberfläche wäre auch die Verwendung eines GUI-Builders denkbar gewesen. Dies hätte allerdings zu Problemen mit den dynamisch erzeugten Diagrammstrukturen geführt.

Der Editor selbst ist nach dem Model-View-Controller-Entwurfsmusters sowohl Sicht als auch Steuerung. Dessen Bestandteile befinden sich im Paket *clientsystem*. Die Hauptklasse *GuiClient* befindet sich im Unterpaket *gui* und erweitert die abstrakte Klasse *ClientImpl.* Diese enthält den grundlegenden Ablauf und die Basisoperationen. Es implementiert die externe Schnittstelle *Client.* Die Veränderungen des Diagramms und die Abschaltung der Datenhaltung werden dem Editor darüber mitgeteilt.

Name	**Verwendung**
JFrame	Abstrahiert die Fähigkeiten eines Fensters als Klasse.
JPanel	Zeichenfläche, die mit anderen Komponenten gefüllt wird.
LayoutManager	Ordnet die enthaltenen Elemente eines *JPanels*.
JLabel	Einzeiliges und nicht veränderbares Textfeld.
JTextField	Textfeld, welches vom Benutzer verändert werden kann.
JButton	Darstellung eines Knopfes, benötigt einen *ActionListener.*
ActionListener	Kapselt die Kombination von Ereignis und Aktion.
MouseListener	Reagiert auf die Bewegungen und Aktionen der Maus.

Tabelle 2: Verwendete Elemente von Java Swing

5.3.2 Musterdurchlauf durch den Editor

Der Programmstarter

Die Benutzung des Programms beginnt mit dem parameterlosen Ausführen der Klasse *NetStructEDIT*. Daraufhin öffnet sich der Auswahlbildschirm für die Betriebsart (Abbildung 18).

Abbildung 18: Der Programmstarter

Das erscheinende Fenster erweitert *JFrame* und enthält eine Anzahl an *JButtons.* In diesen sind *ActionListener* enthalten, die bei einem Linksklick die ausgewählte Betriebsart starten. Zum Einstieg in das Programm empfiehlt es sich, zuerst eine Datenhaltung auf dem eigenen Rechner anzulegen. Dazu muss der Menüpunkt „Datenhaltung“ ausgewählt werden.

Anschließend erscheint das Fenster der Datenhaltung (Abbildung 19). Dieses enthält die Netzwerkinformationen, die über *java.net.InetAddress.getLocalHost()* zugreifbar sind in einem *Jlabel.* Zusätzlich lässt sich die Software über einen JButton beenden.

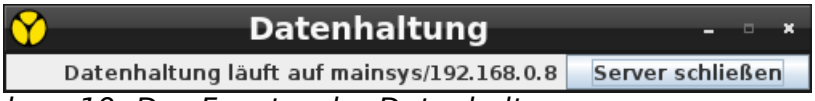


Abbildung 19: Das Fenster der Datenhaltung

Da die Funktion dieses Fensters darin besteht, die Datenhaltung und alle verbundenen Komponenten zu beenden, wird dieses nicht weiter beachtet. Durch einen weiteren Start von *launcher.NetStructEDIT* wird wieder der Auswahldialog angezeigt.

Durch die Auswahl der Option „Editor“ wird die Eingabemaske (Abbildung 20) für die IP-Adresse der Datenhaltung angezeigt.

Abbildung 20: Das Fenster für den Verbindungsaufbau

Dieses enthält einen Knopf für den Aufbau einer Verbindung und ein initialisiertes *JTextField*. Es wurde die für den lokalen Betrieb sinnvolle Loopback-Adresse als Wert eingetragen.

Durch einen Klick auf „Verbinden“ wird ein Verbindungsversuch zur Datenhaltung unternommen.

Das Diagrammauswahlfenster

Nun erscheint das Fenster der Diagrammauswahl (Abbildung 21). In diesem werden *notify()* und *wait()* eingesetzt, um den Ablauf zu ordnen.

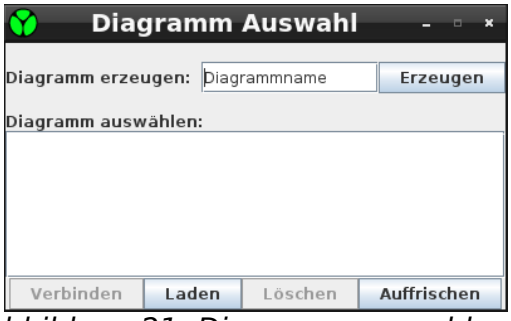


Abbildung 21: Diagrammauswahl

Dem Benutzer wird nun eine *JList* präsentiert, in deren *ListModel* sich die Namen der vorhanden Diagramme befinden. Durch den Aufruf eines *JFileChoosers* ist das Laden eines vorher serialisierten Diagramms möglich, dieses wird der Liste dann hinzugefügt. Da keine Diagramme mitgeladen werden, muss über den Knopf „Erzeugen“ ein Neues angelegt werden.

Dieses wird sofort angezeigt und kann in der Liste ausgewählt werden. Durch das Anklicken von „Erzeugen“ wird die Auswahl bestätigt.

Die Zeichenfläche

Nun gelangt man zur Zeichenfläche (Abbildung 22), dem *ManipulateFrame*. Er verwendet ein *BorderLayout,* um mit darin enthaltenen *JPanels* eine grau umrandete Zeichenfläche zu erzeugen. Bei dem auf weißem Grund sichtbaren Diagrammnamen handelt es sich um die grafische Repräsentation des Wurzelknotens. Die dazu verwendete Klasse repräsentiert auch die anderen Knoten und trägt den Namen DiagramBlockRepresentation.

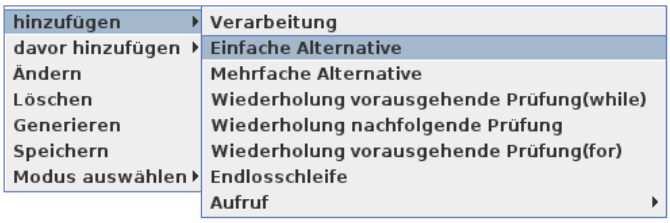
Abbildung 22: Die Zeichenfläche

Dabei handelt es sich um eine von *JPanel* abgeleitete Klasse, welche mit einem *MouseListener* und einer *Border* als Rand ausgestattet ist.

Die Repräsentationen werden von der Fabrikmethode *getDiagramRepresentation()* erzeugt[14]. Der Text wird wieder über ein *JLabel* realisiert. Die geometrischen Formen werden durch die Verwendung von *BoxLayouts* in X- und Y-Ausrichtung erzeugt. Die komplizierten Diagrammstrukturen bauen sich primär auf der Verkettung und Verschachtelung der *DiagramBlockRepresentation* auf. Dieses Prinzip wurde direkt aus den Struktogrammen übernommen. Um dem *LayoutManager* maximalen Spielraum zu geben, gibt es lediglich für die Texte eine Minimalgröße. Der Platz wird dadurch optimal ausgenutzt. Falls dieser nicht ausreicht, befindet sich das *JPanel* des Zeichenfensters in einer *JScrollPane*, die bei Bedarf virtuell mehr Platz schafft.

Interaktion mit den Diagrammblöcken

Mit einem Rechtsklick auf die Zeichenfläche wird der *MouseListener* aktiv und öffnet das allgemeine Kontextmenü (Abbildung 23). Dieses ermöglicht diagrammweite Aktionen, wie das Abspeichern, sowie das Erzeugen, Löschen und die Manipulation von Diagrammblöcken. Der *MouseListener* wird für einen Hover-Effekt verwendet. Durch die Methode *mouseEntered(event)* wird der fokussierte Block eingefärbt.



Das Kontextmenü ist aus einem *JPopupMenu*, *JMenu* und *JMenuItem* aufgebaut. Durch den Menüpunkt hinzufügen, wird ein neuer Block hinter dem aktuellen einzufügen.

Abbildung 23: Das Kontextmenü

Zu Demonstrationszwecken wird eine „einfache Alternative“ ausgewählt. Um die DIN-Norm einzuhalten, wurde in der *DiagramBlockRepresentation* die *paint(g)* Methode des *JPanels* überschrieben. Bei jedem Aufruf wird dadurch mit *g.drawPolyLine()* eine verbundene Linie gezeichnet. Um Antialiasing für diese zu erhalten, wird die Klasse *Graphics2D* verwendet.

Das Veränderungsfenster

Um die Bedingung der Alternative zu ändern wird im Kontextmenü der Menüpunkt „Ändern“ aufgerufen (Abbildung 24).

Das Veränderungsfenster passt sich dabei an die verschiedenen Knotentypen an, sodass diese sinnvoll manipuliert werden können. In diesem Fall ist im unteren Bereich eine *JList* mit den bereits vorhanden Variablen zu sehen und darüber die Möglichkeit, durch *JTextField* die verschiedenen Felder und die Operation der Bedingung zu ändern. Im oberen Abschnitt kann ein Kommentar eingetragen und über „Ändern“ gespeichert werden.

Abbildung 24: Das Änderungsfenster

Mit dem Veränderungsfenster wurden alle wichtigen grafischen Elemente vorgestellt. Der Beispieldurchlauf durch den Editor endet damit.

6. RMI als Bindeglied

6.1 Realisierung der Netzwerkkommunikation

Die bisherigen Betrachtungen zur Implementierung und Realisierung des Editors ließen einen wichtigen Aspekt außer Acht: Die Zusammenarbeit und Kommunikation der verschiedenen Komponenten und der in ihnen enthaltenen Objekte. Diese Zusammenarbeit ist aufgrund der gewünschten Netzwerkfähigkeit besonders relevant und durch zusätzliche Fehler- und Problemquellen auch nicht trivial zu integrieren. Die erste grundlegende Frage, die diesbezüglich erörtert werden muss, ist die Frage nach der Abstraktionsebene, auf der die Netzwerkkommunikation realisiert wird. In der gewählten Programmiersprache stehen sich zwei unterschiedliche Ansätze gegenüber.

Mit Sockets bietet Java die Möglichkeit einer direkten Kommunikation über das Netzwerk. Diese Kommunikation arbeitet nahe an der Netzwerkprotokollebene und es besteht eine sehr klare Trennung von Server und Client[17]. Bei den Sockets wird der Netzwerkverkehr als Datenstrom, ähnlich dem Lesen und Schreiben von Dateien, behandelt. Dies ermöglicht eine starke Optimierung der Kommunikation, macht es aber auch notwendig, jede Aktion im Detail zu betrachten. Auch gehen durch die Nähe zu der Protokollebene alle objektorientierten Zugriffe verloren, da eine explizite Netzwerkschicht existiert. In dieser müssen alle Übertragungen empfangen, ausgewertet und schließlich an die als Ziel bestimmten Objekte weitergereicht werden. Anschließend muss der Kommunikationspartner ebenso explizit über den Erfolg seiner Änderungswünsche an den Objekten informiert werden. Bei der Benutzung von Sockets fällt also eine große Menge zu wartenden und testenden Codes in die Entwicklerverantwortung, gibt diesem aber die Möglichkeit, die Performanz zu steigern.

Die Alternative stellt die Verwendung einer Vermittlungssoftware (engl. Middleware) dar. Sie abstrahiert die Netzwerkoperationen und ermöglicht es auf die Objekte, die auf einem anderen Rechner liegen, zuzugreifen,

ohne die Kommunikation als eingehende und ausgehende Datenströme zu betrachten.[18]. Dieser Ansatz ist also weiter von der Netzwerkprotokollebene entfernt und dementsprechend schlechter für die Feinoptimierungen geeignet. Die Verwendung von Vermittlungssoftware ermöglicht im Gegenzug eine Beibehaltung des objektorientierten Entwurfs und den Rückgriff auf ein bereits vorhandenes und getestetes Kommunikationssystem. Da für den Editor das Zeitverhalten nur im Rahmen der menschlichen Wahrnehmung beachtet werden muss, ist die fehlende Möglichkeit zur Optimierung zu vernachlässigen. Die Vorteile überwiegen.

Nachdem nun also die große Frage nach der Abstraktionsebene, auf der die Netzwerkkommunikation stattfindet, mit der Verwendung einer Vermittlungssoftware beantwortet wurde, stellt sich die Frage nach dem konkret verwendeten Produkt aus dieser Kategorie. Für das Java Softwaresystem stehen dabei drei Produkte zur Auswahl. Es muss zwischen den sprachunabhängigen Systemen CORBA, definiert durch die OMG[19], Webservices, definiert durch das W3C[20] und Oracles Java Remote Method Invocation[17] gewählt werden. CORBA und Webservices bieten den Vorteil, dass durch die Sprachunabhängigkeit auch Komponenten angebunden werden können, die nicht in Java geschrieben wurden. RMI hingegen bietet den Vorteil, dass durch die Beschränkung auf nur eine Programmiersprache deren Möglichkeiten besser ausgenutzt und entsprechend die Verwendung von javaspezifischen Konstrukten weniger einschränkt wird. Auch bietet RMI den Vorteil, die Interfaces der verwendeten Komponenten nicht vorher in einer Metasprache definieren zu müssen, um diese dann mit einem externen Compiler in sprachspezifische Konstrukte umzuwandeln. Es reicht, das Interface *Remote* zu implementieren[21], ansonsten ändert sich an der Vorgehensweise bei der Implementierung wenig. Vor Java 1.5 war noch die Benutzung eines externen Compilers notwendig. Um die sprachspezifischen Konstrukte von Java ausnutzen zu können und den Overhead durch die Verwendung von CORBA und Webservices zu sparen, wird RMI verwendet.

6.2 Die Fähigkeiten von RMI im Überblick

Als Ergebnis der Diskussion im vorherigen Abschnitt wird Java RMI für den Aufbau eines verteilten Systems benutzt. RMI übernimmt dabei die Funktion einer Vermittlungssoftware und stellt so die auf einer Maschine vorhandenen Objekte über das Netzwerk anderen Computern zur Verfügung.

Der grundlegende Aufbau und die Fähigkeiten von Java RMI entsprechen dabei dem einer jeden Vermittlungssoftware. Durch die Beschränkung auf eine Implementierungssprache sind bestimmte Aspekte für dieses verteilte System jedoch unwichtig. Der Aufbau ist aber ansonsten mit anderen Produkten aus dieser Kategorie vergleichbar[22]. Das eigentliche System besteht dabei aus drei funktionalen Teilen, jedoch ist auch die Betrachtung der Kommunikation über Maschinen- und Sprachgrenzen hinweg wichtig.

6.2.1 Namensdienst

Der Namensdienst ähnelt in seinem Verhalten dem eines DNS- Servers. Im Gegensatz zu diesem können jedoch nicht Tupel aus identifizierendem Namen und IP-Adresse(n), sondern Tupel aus Namen und Objektreferenzen abgerufen werden[18]. Da durch den Namensdienst garantiert wird, dass kein Name doppelt vorkommt, kann auf diese Weise leicht ein Singleton Entwurfsmuster[14] erzeugt werden. Der Namensdienst wird RMI-Registry genannt und ist, wie fast alles in Java, eine Klasse. Der Dienst kann entweder als separates Programm oder direkt mit dem als Server dienenden Objekt in dessen JVM gestartet werden. Der erste Ansatz hat dann Vorteile, wenn der Server instabil läuft oder der Namensdienst schon vor oder auch noch nach der Verwendung des Servers vorhanden sein muss. Die Registry mit dem Server zu starten hat den Vorteil, dass es benutzerfreundlicher ist, da nur ein Programm ausgeführt werden muss.

Die Implementierung der Datenhaltung ist so gestaltet, dass diese entweder einen per Parameter definierten Namensdienst benutzt oder diesen ansonsten selber startet. Die Registry wird dabei mit dem Befehl *rmiregistry* extern gestartet, intern wird diese mit *LocateRegistry.getRegistry()* referenziert. Das Erzeugen findet mit der Anweisung *LocateRegistry.createRegistry(PORT)* in der Datenhaltung statt[16]. Nachdem die Registry gestartet wurde, können exportierte Objekte angemeldet werden, um diese über einen Namen zugreifbar zu machen. Eine Anfrage nach einem entfernten Objekt wird mit der Anweisung *LocateRegistry.getRegistry().lookup("Name")* durchgeführt. Als Rückgabewert gibt der Aufruf die Objektreferenz auf das entfernte Objekt zurück.

Im folgenden Sequenzdiagramm[13] (Abbildung 25) ist die Initialisierung der Datenhaltung und das Verbinden eines nachträglich gestarteten Editors abgebildet. Das Exportieren des eigentlichen Diagramms wird dabei vereinfacht dargestellt, ansonsten werden die Klassennamen aus dem Code und die realen Methodennamen verwendet. Die Vorgänge des Exportierens wurden dabei in die im ganzen System benutzte Klasse *sharedHelper.RMIHelper* ausgelagert. Diese führt den eigentlichen Aufruf *UnicastRemoteObject.exportObject(obj, 0)* aus und meldet Objekte mit *LocateRegistry.getRegistry().rebind("name", obj)* bei dem Namensdienst an, falls dies gewünscht ist. Besonderes Augenmerk liegt in der zentralen Rolle des Namensdienstes zum Auffinden der Datenhaltung und der unterschiedlichen Behandlung von Objekten mit und ohne Namenseintrag.

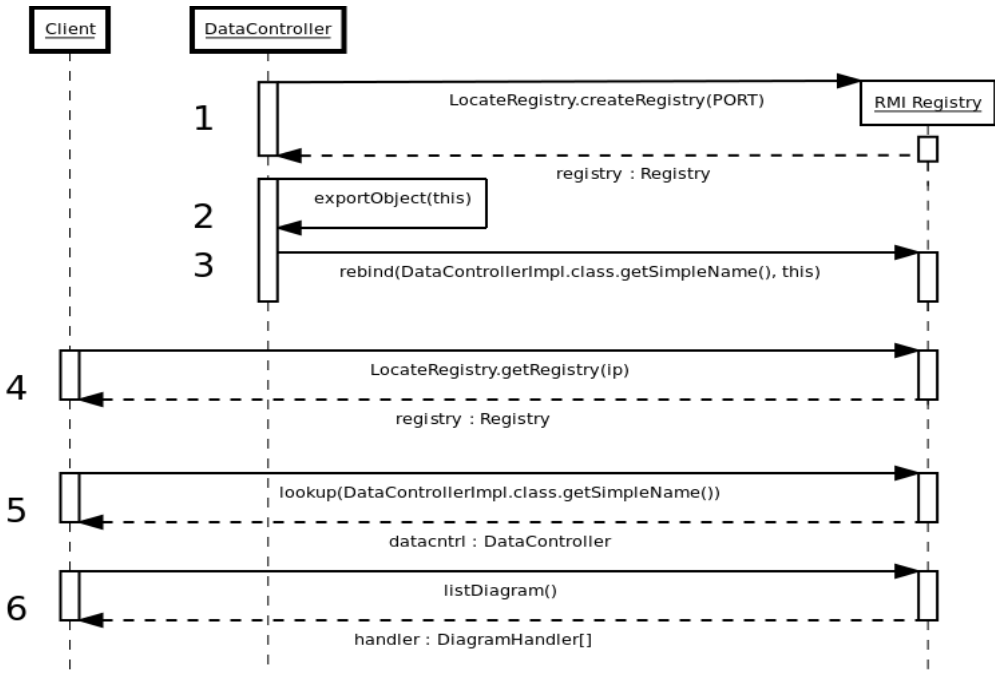


Abbildung 25: Start von Datenhaltung und Editor

1. Der Namensdienst wird aus der Datenhaltung heraus gestartet.
2. Die Datenhaltung exportiert sich selbst.
3. Das nun zugreifbare entfernte Objekt wird unter dem eigenen Klassennamen bei dem Namensdienst registriert .
4. Durch die Angabe der IP erhält der Editor Zugriff auf die RMI-Registry.
5. Über den bekannten Klassennamen der Datenhaltung erhält der Editor eine Referenz auf diese.
6. Die noch keine Einträge umfassende Diagrammliste wird abgefragt.

6.2.2 Bereitstellung von entfernten Objekten

Entferntes Objekt nennt man solche Objekte, die auf einem Rechner liegend von anderen Maschinen aus zugreifbar sind[18]. Diese müssen, bevor sie über das Netzwerk erreichbar sind, exportiert werden. Der Vorgang des Exportierens ist jedoch nicht mit einem Zwang zur Anmeldung bei dem Namensdienst verbunden.

Um ein solches Objekt unter RMI zu erhalten, muss dessen Interface die Schnittstelle *Remote* erweitern und alle Methoden die Angabe enthalten, dass eine *RemoteException* auftreten kann[16]. Zur Laufzeit können so aufgebaute Objekte dann exportiert werden, wodurch ein Skeleton erzeugt wird. Dieser nimmt die über Remote Procedure Call gesendeten Aufrufe vom Netzwerk entgegen und leitet diese an die lokal vorhandenen Objekte weiter. Die Skeletons sind dabei durch die Klasse RemoteServer implementiert. Sie bietet Hilfsfunktionen für die Serverobjekte an, z.B. kann überprüft werden, wer zur Zeit ein Objekt verwendet. Dies wird für den *Client* und den *Codegenerator* eingesetzt, um den Aufruf der *disconnect()* Methode entweder der Datenhaltung oder dem *KeepAliveThread* zuzuordnen. Die Zugriffe auf die exportierten Objekte sind nicht gegeneinander abgeschirmt, dementsprechend muss dafür gesorgt werden, dass die Daten konsistent bleiben.

Um die Netzwerkfähigkeit des Systems bereitzustellen, sind alle Teile der Datenhaltung über das Netzwerk zugreifbar. Es sind also die Diagrammknoten, welche die Klasse *NodeImpl* erweitern, die Diagrammlistenverwaltung *DataController* und die Diagrammverwaltung *DiagramHandler* auf diese Weise realisiert. Es besteht nicht die Beschränkung, dass nur auf einem Rechner liegende Objekte zugreifbar sind. Diese Eigenschaft ermöglicht die Implementierung eines Rückkanals. So sind auch der Editor und der Codegenerator ebenfalls entfernte Objekte und können so über RPC von der Datenhaltung über Änderungen oder die Beendigung des Gesamtsystems informiert werden.

6.2.3 Verwendung von Objektreferenzen

Die Objektreferenz dient dem Zugriff auf ein entferntes Objekt. Es handelt sich um eine Referenz auf ein Objekt, welches auf einem anderen Rechner liegt. Diese Referenzen können entweder direkt von dem Namensdienst oder einem bereits referenzierten Objekt erhalten werden. Es handelt sich bei der Objektreferenz um einen Proxy, der für die Implementierung den Anschein erweckt, dass das Objekt lokal zugreifbar ist. Das ermöglicht es, das entfernte Objekt genauso wie ein lokales Objekt zu behandeln. Dieses Verhalten nennt man Zugriffstransparenz[22].

Unter RMI werden die Proxys durch die Klasse *RemoteStub* realisiert. Diese Klasse bietet auch Hilfsfunktionen an. So kann für ein einmal exportiertes Objekt jederzeit die Referenz erhalten werden. Da es sich bei dem *RemoteStub* um einen Proxy für ein Interface handelt, ist es möglich, zur Laufzeit einer Komponente Änderungen am Code eines darauf zugreifenden Systems durchzuführen und mit den vorhandenen Daten wieder weiterzuarbeiten. Diese Eigenschaft ist für die Entwicklung des Codegenerators und des Editors sehr hilfreich, da so immer nur eine Komponente für Änderungen bearbeitet werden muss. Bei der Verwendung des *RemoteStubs* ist zu beachten, dass die über RPC beauftragte Aktion nicht abgeschlossen sein muss, bevor die nächste in Auftrag gegeben werden kann. Für alle im Abschnitt entfernte Objekte angegebenen Systemkomponenten werden für den Zugriff die eben eingeführten Proxys verwendet.Um die Verwendung von Proxy und entferntem Objekt zu verdeutlichen wird wieder auf ein Sequenzdiagramm zurückgegriffen. Hierbei wird auch dargestellt, wie der Rückkanal für das Senden von Informationen an den Editor hergestellt wird.

Der Aufbau des Rückkanals wird in dem folgenden Diagramm dargestellt (Abbildung 26).

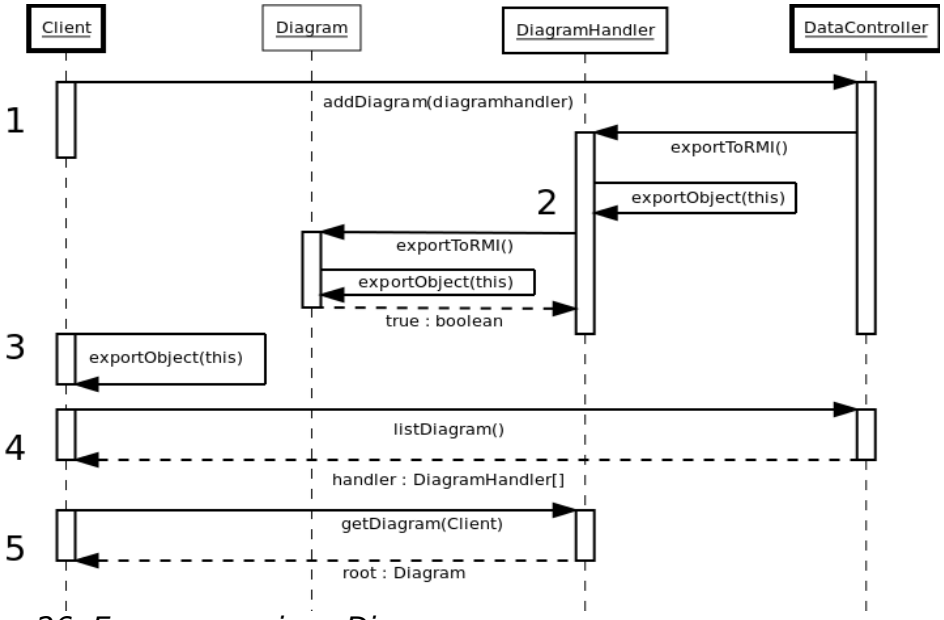


Abbildung 26: Erzeugung eines Diagramms

1. Der Editor erzeugt lokal eine Diagrammverwaltung. Deren Konstruktor fordert für die Erzeugung ein Diagramm, dessen Name als Identifikator verwendet wird. Bei dem übergebenen Diagramm kann es sich auch um ein vorher serialisiertes handeln. Auf diese Weise ist das Laden eines gespeicherten Diagramms und das folgende Hochladen zur Datenhaltung realisiert.
2. Die Diagrammlistenverwaltung nimmt nach Überprüfung des Namens auf Eindeutigkeit die neue Diagrammverwaltung in seine Liste auf und exportiert diese. Die Diagrammverwaltung stellt zu diesem Zweck erst sich selbst und durch den Aufruf einer rekursiven Methode, die das gesamte Diagramm exportiert, das Diagramm zur Verfügung. Dieser Schritt findet wie alles synchronisiert statt, um nicht Gefahr zu laufen, Zugriff auf ein nur zum Teil exportiertes Diagramm zu geben.
3. Anschließend exportiert sich der Editor selbst und ermöglicht so die Verwendung eines Rückkanals.
4. Im nächsten Schritt wird die nun einen Eintrag umfassende Liste von der Listenverwaltung heruntergeladen.
5. Um Zugriff auf das in der Diagrammverwaltung liegende Diagramm zu erhalten, muss der Editor die Referenz auf sich selbst übergeben. So ist sichergestellt, dass die Informationen über Änderungen bei diesem ankommen.

6.2.4 Übermitteln von Objekten und primitiven Typen

Die Übertragung eines Parameters, gleich ob primitiver Typ oder komplexes Objekt, muss bei den meisten Realisierungen von Vermittlungssoftware in ein Austauschformat überführt werden[23]. Diese Umwandlung wird Marshalling genannt und ist nötig, da zum einen unterschiedliche Programmiersprachen unterstützt werden und zum anderen diese auch auf verschiedenen Architekturen zusammenarbeiten können. Ein Problem bei der Verwendung unterschiedlicher Sprachen wäre z.B., dass der Datentyp String unterschiedlich definiert ist. Auch bei den primitiven Typen kann es ohne Marshalling bei der Verwendung unterschiedlicher Prozessorarchitekturen zu Problemen kommen. So kann der eine Teil des Systems auf einem Prozessor mit Little-Endian und der andere Teil des Systems auf einem Rechner mit Big-Endian Architektur laufen. Keine Zahl würde korrekt übertragen werden können.

Bei RMI gibt es nur die Sprache Java, die Objekte sind also einheitlich und konsistent definiert. Auch das Problem mit Little- und Big-Endian tritt nicht auf, da in der JVM die Bytereihenfolge für die primitiven Typen vorgegeben ist[24]. Einzige Bedingung für den Austausch von Parametern und Rückgabewerten ist, dass diese serialisierbar sind. Dies ist einfach durch das Implementieren der Schnittstelle Serializable getan. Basistypen und Arrays können ohne spezielle Vorbereitungen über das Netzwerk verschickt werden.

Im System werden hauptsächlich Aktionen, Bedingungen und Felder verschickt. Diese werden durch die Interfaces *Action*, *Condition* und *Field*, repräsentiert. Um das Laden und Speichern von Diagrammen zu ermöglichen, müssen diese genauso wie die mit Ihnen zusammenhängende Verwaltungsstruktur *DiagramHandler Serializable* implementieren.

Bei der Übertragung von Parametern wird zwischen der Übertragung als Referenz und als serialisiertes Objekt unterschieden. Wenn beide Eigenschaften erfüllt werden, wird die Objektreferenz weitergegeben.

6.3 Entwurfsrelevante Eigenschaften von RMI

6.3.1 Komplexitätssteigerung in der Fehlerbehandlung

Die größte Auswirkung hat die Verwendung von RMI bei der Fehlerbehandlung, denn bei jedem Methodenaufruf über den Proxy kann es zu Netzwerkfehlern kommen. So kann es zu einer Unterbrechung der Verbindung, einem Absturz des Servers oder zu einem Zugriff auf ein bereits wieder abgemeldetes Objekt kommen.

Auf diese Probleme reagiert Java mit dem Werfen einer *RemoteException*. Die *RemoteException* ist von der *IOException* abgeleitet und wird ihrerseits von der *NoSuchObjectException* erweitert[25]. Die Erweiterung der IOException ist insofern sinnvoll, als dass der Netzwerkzugriff als Ein-/Ausgabe betrachtet werden kann. Die *NoSuchObjectException* wird für den Fall verwendet, dass das Objekt nicht (mehr) über Netzwerk zugreifbar ist. Wird das Programm ordnungsgemäß beendet, werden auch die darauf zugreifenden *RemoteStubs* informiert. Wird allerdings die JVM beendet oder die Verbindung unterbrochen, gibt es keine Automatik, die dies erkennt. So würde es bis zur nächsten Benutzerinteraktion dauern, bis der Fehler festgestellt wird.

Um dies trotzdem erkennen und auf diese Fehlersituation reagieren zu können, wurde ein sogenannter Keep-Alive Thread erzeugt. Dieser überprüft, ob der Wurzelknoten noch vorhanden ist und beendet ansonsten den angeschlossenen Editor oder Codegenerator. Die dazu verwendete Klasse heißt *sharedHelper.KeepAliveThread* und wird im grafischen Editor und dem Codegenerator verwendet.

Tritt im Programm, z.B. beim Durchlauf des Diagrammbaums zum Zwecke der Codeerzeugung, eine *RemoteException* auf, wird zuerst überprüft, ob durch die lose Kopplung die Änderungsinformation über das Entfernen eines Knotens nur noch nicht durchgedrungen ist oder tatsächlich das gesamte Diagramm nicht mehr zu erreichen ist.

6.3.2 Auswirkungen auf das Debugging der Anwendung

Eine erste, noch recht unkritische Besonderheit bei dem Zugriff auf entfernte Objekte über den *Remotestub* tritt bei der Verwendung der *toString()* Methode auf. Jedes Objekt in Java verfügt darüber und in den Programmierkonventionen ist festgelegt, dass ihr Rückgabewert den Zustand des Objektes als Text repräsentieren soll.

Die in dem entfernten Objekt sinnvoll implementierte *toString()* Methode wird jedoch nicht übertragen. Stattdessen erhält der Benutzer eine Zustandsbeschreibung des *RemoteStubs*. Im Debugger wird dadurch weder die innere Struktur des entfernten Objektes dargestellt noch eine nützliche Kurzzusammenfassung gegeben.

Dies führt dazu, dass sowohl auf der Seite des Proxys als auch der des entfernten Objektes Breakpoints gesetzt und beide Programmteile im Debugger gestartet werden müssen.

6.3.3 Objektsynchronisation im verteilten System

In der Java Programmierung ist es üblich, Objekte mit den von der Klasse *Object* ererbten Schlüsselwörtern *notify()* und *wait()* (Abbildung 27) zu synchronisieren[16]. Ein Thread blockiert auf den Aufruf der Methode *wait()*, bis von dem Synchronisationsobjekt die *notify()* Methode aufgerufen wurde. Würde die Software nur lokal laufen, könnte auf diese Weise leicht die Verteilung der Information über Änderungen im System stattfinden. Die Anzeige würde auf eine Änderungen an einer Stelle im Diagramm warten und dann reagieren indem z.B. der Text des dargestellten Knotens aktualisiert wird.

```
synchronized(obj){
    obj.wait();
}
```

Abbildung 27: Verwendung von wait()

Die Verwendung dieses praktischen und seit Java Version 1.0[25] vorhandenen Mechanismus ist über RMI allerdings nicht möglich. RMI verhält sich in diesem Aspekt nicht zugriffstransparent, denn die Objekte blockieren nicht. Eine Synchronisation ist auf diese Weise schlicht nicht möglich. Die Ursache ist in der Funktionsweise von *notify()* und *wait()* begründet. Diese Methoden dürfen nur innerhalb eines *synchronized* Blocks verwendet werden, denn zur Synchronisation wird der Objektmutex verwendet. Dieser Mutex ist bei einem Zugriff über das Netzwerk nicht verfügbar, kann also nicht gelockt werden[17].

Das Verteilen von Informationen im System muss auf anderem Wege realisiert werden. Eine einzelne Anzeige kann leicht über Änderungen informiert werden, indem von dem betrachtetem Objekt eine Methode

aufgerufen wird, die erst nach Änderungen entblockiert. Lokal sind die Mutexe selbstverständlich verwendbar und dementsprechend können diese für diese Aufgabe benutzt werden. Mit Hilfe eines Semaphors kann die Anzahl der Änderungen bekanntgegeben werden.

Mit dieser Technik wird der *DiagramHandler* über Veränderungen des Diagramms informiert. Er blockiert auf der *hasChanged()* Methode des Wurzelknotens und wird bei jeder Änderung wieder freigegeben und kann darauf reagieren. Der Wurzelknoten wird, wie auch schon bei der zentralen Verwaltung der Felder, hierfür verwendet, da er als einziger Teil des Diagramms immer existiert. Dementsprechend müssen alle Kind- und Kindeskind-Knoten den Wurzelknoten informieren, wenn es bei ihnen zu Veränderungen kommt. Dies stört selbstverständlich die Möglichkeit, parallel Änderungen am Diagramm durchzuführen. Glücklicherweise tritt diese Einschränkung nur bei verändernden und nicht bei lesenden Operationen auf.

Auf diese Weise kommt die Information zuverlässig und zeitnah bei dem *DiagramHandler* an. Dieser muss die Informationen dann an die Endverbraucher, also die Editoren oder Codegeneratoren, weitergeben. Diese müssen jedoch eine andere Methode verwenden, da es sich um potenziell sehr viele zu informierende Objekte handelt. Indem sowohl die Editoren als auch die Codegeneratoren per RMI exportierte Objekte sind, die über RMI angesprochen werden können, sowie dem *DiagramHandler* bekannt sind, werden diese durch einen Methodenaufruf über die Änderungen informiert.

Um die Bandbreite zu schonen, wird ein Informationstuple aus der dem Typ und Ort der Veränderung übergeben. Diese Informationen können dann vom Editor verwendet werden, um die Änderung präzise durchzuführen.

Um das Verhalten des Systems bei einer Veränderung zu demonstrieren, wird wieder auf ein Sequenzdiagramm[13] (Abbildung 28) zurückgegriffen. In diesem wird in ein bestehendes Diagramm ein weiterer Knoten eingefügt. Zum Zeitpunkt des Einfügens sind sowohl ein Codegenerator als auch ein Editor angemeldet. Vor der Einfügen Operation besteht das Diagramm aus der Verkettung von Wurzelknoten, Einleseoperation und Schreiboperation.

Das Hinzufügen eines Knotens wird nachfolgend dargestellt (Abbildung 28).

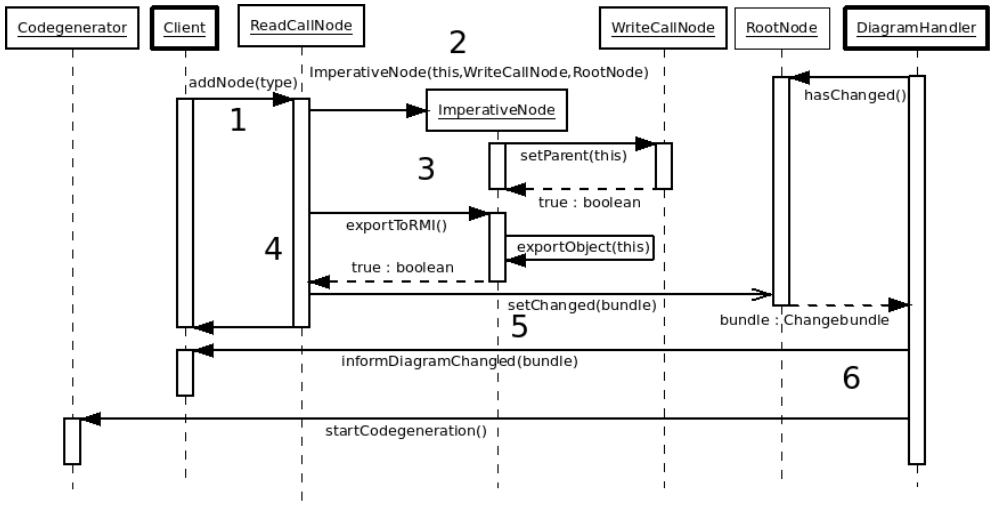


Abbildung 28: Hinzufügen eines Knotens

1. Der Editor gibt das Hinzufügen eines neuen Knotens hinter der Leseoperation in Auftrag. Um dies zu erreichen, sendet er über RMI einen uninitialisierten Block vom gewünschten Typ an den Knoten, hinter dem dieser eingefügt werden soll.
2. Der angesprochene Knoten erzeugt lokal einen neuen Verarbeitungsknoten mit den konkreten Parametern. Es wird eine Referenz auf sich selbst als Elternknoten, eine Referenz auf das vormalige Kind als Kindknoten und eine Referenz auf den Wurzelknoten übergeben.
3. Da ein Kindknoten angegeben wurde, wird dieser über den neuen Elternknoten informiert.
4. Der Knoten, der die Einfügeoperation durchführt, gibt anschließend den Befehl an den neuen Knoten, sich über RMI zur Verfügung zu stellen.
5. Nachdem der Knoten sich selbst im Netzwerk verfügbar gemacht hat, informiert der Leseoperationsknoten den Wurzelknoten über die Veränderung.
6. Die Diagrammverwaltung hat auf eine Anfrage nach Veränderungen blockiert und kann nun die Informationen an alle angemeldeten Editoren und Codegeneratoren übermitteln. Die Übermittlung findet synchronisiert statt, um den Ausfall eines zu informierenden Editors oder Codegenerators zu bemerken. Intern wird dafür gesorgt, dass die Diagrammverwaltung nicht alle Codegenerierungs- und Darstellungsvorgänge abwarten muss, bevor die nächste Operation verbreitet werden kann.

6.3.4 Identifikation der Knotentypen zur Laufzeit

Der *instanceof* Operator ermöglicht es, den Typ eines Objektes zur Laufzeit zu erfragen, von dem nur die Oberklasse oder das Interface bekannt sind[25]. Gerade für die auf Polymorphie aufbauende Datenhaltung ist diese Operation existenziell, da mit ihrer Hilfe überprüft wird, um welchen konkreten Knotentyp es sich bei dem Diagrammknoten handelt. Durch diesen Vorgang muss nicht parallel eine Enum-Klasse mitgeführt werden, die diese Information auch bereitstellen könnte.

Leider gehen mit RMI die Informationen über den konkreten Typ einer Unterklasse verloren. Es lässt sich lediglich prüfen, welche *Remote* erweiternden Interfaces verwendet werden. Im Package *example.abstractRMI* ist ein ausführbares Beispiel für dieses Verhalten. Um dennoch *instanceof* verwenden zu können, wird der Umweg über Markerinterfaces gegangen[16]. Diese markieren ein Objekt einer Klasse, sodass die Überprüfung nun wieder möglich ist. Folglich muss für alle konkreten Diagrammknoten, welche die abstrakte Klasse *NodeImpl* erweitern, ein eigenes Interface vorhanden sein, um ihren Typ bestimmen zu können.

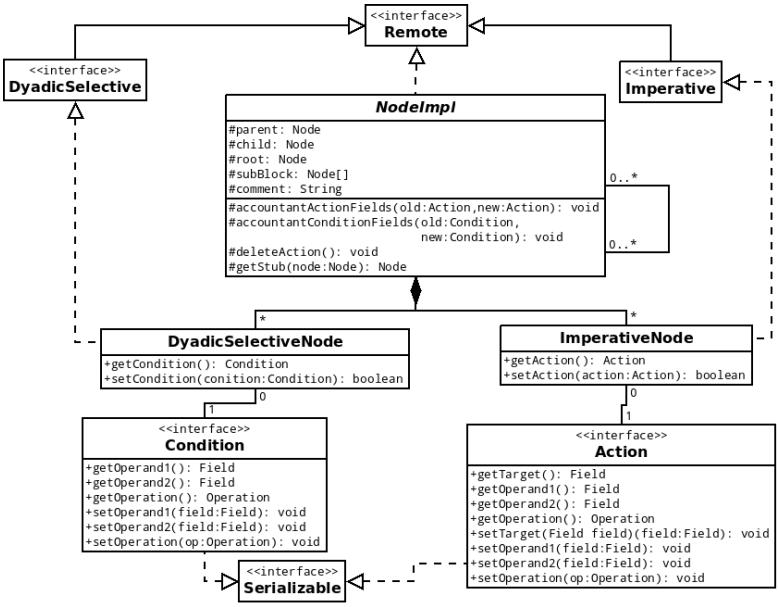


Abbildung 29: Verwendung von Markerinterfaces

Der Ausschnitt aus dem Klassendiagramm (Abbildung 29) zeigt die Verwendung von Markerinterfaces anhand der Klassen *ImperativeNode* und *DyadicSelectiveNode*. Nicht im Diagramm befinden sich die weiteren von *NodeImpl* abgeleiteten Klassen sowie die von diesen implementierten Interfaces. Die Schnittstellen *DyadicSelective* und *Imperative* treten hier in der Rolle von Markerinterfaces auf.

6.3.5 Objektserialisierung über das Netzwerk

Auch bei der Objektserialisierung tritt das Fehlen vollständiger Zugriffstransparenz wieder auf. Es ist zwar problemlos möglich, einen *RemoteStub* zu serialisieren, allerdings werden dadurch nicht die Informationen des referenzierten Objektes gespeichert, sondern der Zustand des *RemoteStubs*. Dieses Vorgehen speichert statt der gewünschten Diagrammdaten lediglich die Information, auf welchem Computer sich das Diagramm zur Serialisierungszeit befand.

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();
ObjectOutputStream oos;
ObjectInputStream ois;
ByteArrayInputStream byin;
Diagram copy = null;

try {
        oos = new ObjectOutputStream(baos);
        oos.writeObject(this.diagram);
        oos.close();
        byin = new ByteArrayInputStream(baos.toByteArray());
        ois = new ObjectInputStream(byin);
        copy = (NSDNode) ois.readObject();
} catch (ClassNotFoundException e) {
        e.printStackTrace();
} catch (IOException e) {
        e.printStackTrace();
}
```

Abbildung 30: Workaround für die entfernte Objektserialisierung

Um dennoch eine Kopie des Diagramms anfertigen zu können, muss direkter Zugriff auf dieses vorhanden sein. Dann ist mit Hilfe der tiefen Objektserialisierung (Abbildung 30) das Erstellen einer verwendbaren Kopie möglich[16].

Das Vorgehen besteht darin, das Objekt abzuspeichern und dann wieder zu laden, um nicht nur eine Objektreferenz, sondern den aktuellen Zustand verschicken zu können. Der *DiagrammHandler* übernimmt aufgrund seines direkten Zugriffs auf das Diagramm diese Aufgabe.

6.3.6 Einschränkungen von RMI durch das TCP/IP-Protokoll

Um die Kommunikation zwischen den Objekten über RMI herzustellen, wird auf das TCP Protokoll zurückgegriffen. Es wird dabei zum Zeitpunkt des Exportierens die Netzwerkadresse des Rechners und damit auch des Objektes bestimmt. Leider ist dieses Verfahren in zweierlei Hinsicht nicht optimal.

Unter unixartigen Betriebssystemen wird immer die erste unter dem Rechnernamen angegebene IP-Adresse in der Datei */etc/hosts* verwendet. Häufig ist hier die Loopback-Adresse eingetragen, sodass ein *RemoteStub* auf einem anderen Rechner das entfernte Objekt auf der eigenen Maschine sucht, nicht findet und deshalb einen Fehler wirft. Durch Eintragen der verwendeten Netzwerkadresse kann dieses Problem behoben werden.

Zum anderen setzt RMI darauf, dass jeder Computer weltweit eine eindeutige Adresse besitzt. In dem heutigen IPv4 Netz ist dies nicht der Fall, dort hat jeder Hausanschluss eine einzige IP-Adresse, die von allen Rechnern im lokalen Netzwerk zur externen Kommunikation verwendet wird. Dieser Vorgang wird Network Address Translation genannt. Läuft das Programm nicht auf dem Router, enthalten alle Objektreferenzen Adressen aus dem jeweiligen lokalen Netzwerk. Auf diese Weise wird nicht über die WAN-Schnittstelle, sondern im eigenen Netzwerk nach dem entfernten Objekt gesucht und dieses dementsprechend nicht gefunden.

Durch die Einführung von IPv6 wird es möglich, jeden Rechner global eindeutig zu identifizieren, da jeder Hausanschluss einen IP-Bereich statt einer einzelnen IP-Adresse zugewiesen bekommt[26].

6.3.7 Datenkonsistenz in der Diagrammstruktur

Wie bereits im Absatz 6.2.2 erwähnt, liegt es bei der Verwendung von RMI in der Entwicklerverantwortung, sicherzustellen, dass die Daten der entfernten Objekte konsistent bleiben und sich gleichzeitige Zugriffsversuche nicht gegenseitig stören. Die Datenhaltung ist dafür verantwortlich, die Konsistenz aufrecht zu erhalten.

Die Zugriffsproblematik lässt sich dabei in zwei Teilprobleme zerlegen. Zum einen gibt es Änderungen, die nur einen Knoten betreffen, zum anderen Vorgänge mit diagrammweiten Auswirkungen.

Änderungen, die nur einen Knoten betreffen, können mit dem Objektmutex und der Verwendung von *synchronized* als Schlüsselwort für die verwendeten Methoden abgesichert werden. Damit kann jeweils nur eine Änderung zur Zeit stattfinden und es wird eine Happend-Before-Relation erzeugt[25].

Die zweite Problematik wird gelöst, indem verändernde Operationen im Baum nur in Abwärtsrichtung möglich sind. Bei der Einfügeoperation ist neben dem aufrufenden Knoten nur ein weiterer Knoten beteiligt, hier können keine Probleme auftreten.

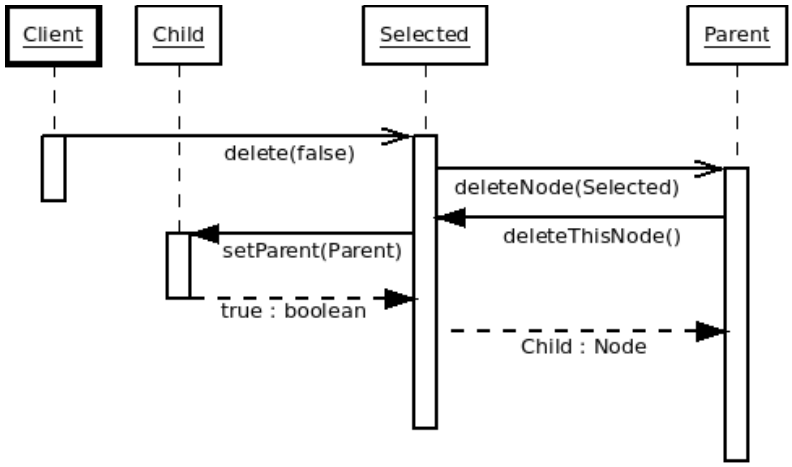


Abbildung 31: Löschen in der Diagrammdatenstruktur

Die Löschen Operation bezieht allerdings drei Knoten mit ein. Sie wird aufgrund der Nebenläufigkeit dreistufig durchgeführt (Abbildung 31).

Im ersten Schritt wird dem Knoten im Baum über dem zu löschenden Knoten der Befehl gegeben den übergebenen Kindknoten zu löschen. Dadurch wird sichergestellt, dass kein frisch eingefügter statt des gewünschten Knotens gelöscht wird. Der Elternknoten ruft die eigentliche Löschen-Operation bei dem ausgewählten Kindknoten aus, sodass sich ab diesem Zeitpunkt zwei Knoten im kritischen Abschnitt befinden. Im letzten Schritt wird der vormalige Kindknoten des zu löschenden Knotens über seinen neuen Elternknoten informiert. Nach dem Erreichen des Maximums an drei gesperrten Knoten werden nun die Locks gelöst und der entfernte Knoten entexportiert sich.

7. Diagrammbasierte VHDL-Codeerzeugung

7.1 Konzept der VHDL-Codeerzeugung

7.1.1 Definition des Umwandlungsziels

Die komplizierte Erzeugung von VHDL-Quelltext ist die Hauptaufgabe des Diagrammsystems. Im Gegensatz zur C-Codegenerierung bestehen mehrere Möglichkeiten, die Struktogramme in VHDL-Quelltext abzubilden. Als Verfahren existieren dabei die Erstellung eines einzelnen kombinatorischen Prozesses, der Aufbau einer Pipeline-Struktur und die Erzeugung eines Mehrzyklus-Datenpfad. Jedes Verfahren bieten seine eignen Vor- und Nachteile und es bestehen deutliche Unterschiede zwischen den verschiedenen Herangehensweisen.

Kombinatorischer Prozess

Bei diesem Lösungsansatz werden alle Operationen des gesamten Diagramms von einem einzelnen kombinatorischen Prozess ausgeführt. Es können dabei nur solche Konstrukte benutzt werden, die innerhalb eines solchen Prozesses verwendet werden dürfen. Diese Lösung ist in der Implementierung am einfachsten, da die hier verwendeten Statements denen einer prozeduralen Sprache gleichen. Auf diese Weise könnte eine Abbildung von einem Diagrammblock direkt auf einer Zeile Code gefunden werden. Realisiert würden die Schleifen in der Form von Loop-Statements mit verschiedenen Abbruchbedingungen, die Alternativen in Abhängigkeit von der Anzahl der Auswahlfälle als IF- oder Case-Konstrukte. Variablen könnten einfach als solche im Prozess deklariert werden. Die Ausgangssignale müssten dann in der umschließenden Architektur erzeugt werden. Die so erzeugte Komponente wäre schnell und die Implementierung der Codegenerierung einfach. Allerdings kann ohne Taktung kein Zeitverhalten erzielt werden und der Ressourcenverbrauch wäre relativ hoch.

Pipeline Struktur

In einer Pipeline Struktur werden die einzelnen Verarbeitungsschritte zeitlich aufgeteilt und die Eingangswerte von einer Stufe zur nächsten weitergereicht. So befindet sich der neueste Wert am Anfang der Verarbeitungskette, während die älteren Werte schon weiter vorgedrungen sind[27]. Dies ist sehr gut für das Abarbeiten schnell herein strömender Messwerte geeignet, die nach einem immer gleichen Muster verarbeitet werden müssen. Für die Nachbildung eines beliebigen Algorithmus ist diese Struktur allerdings ungeeignet. Das Konzept eignet sich sehr schlecht für die Realisierung von Verzweigungen und Schleifen. Realisiert werden würde dies über mehrere getaktete Prozesse, die die Werte des vorherigen als Eingangssignale verwenden.

Mehrzyklus-Datenpfad

In einem Mehrzyklus-Datenpfad wird die Abfolge der Rechenschritte von einem endlichen Zustandsautomaten gesteuert und die Rechnungen in einem Datenpfad ausgeführt. Die Arithmetik-Ressourcen werden dabei geteilt. Aus diesem Grund verfügen sie über Eingangsmultiplexer, die in Abhängigkeit von den anliegenden Selektions-Signalen die benötigten Registerausgänge an die Eingänge der Arithmetik-Einheiten legen. Diese Register werden ebenfalls von dem Zustandsautomaten gesteuert. Sie erhalten das Enable-Signal. Auch ihnen ist ein Multiplexer vorgeschaltet, falls sie Rechenergebnisse von verschiedenen Arithmetik-Einheiten aufnehmen müssen. Die Zustandsübergänge finden taktgesteuert statt und können von dem Ergebnis von Komparatoren abhängig sein[27].

Durch dieses Verhalten bei den Zustandswechseln kann eine dem Kontrollflussgraphen ähnliche Struktur erzeugt werden, die jedes Programm abbilden kann. Im Gegensatz zur Erzeugung eines kombinatorischen Prozesses werden hier weniger Ressourcen verbraucht und durch die getakteten Zustandsübergänge kann zeitabhängiges Verhalten erreicht werden. Der Nachteil liegt in der geringeren Geschwindigkeit und dem höheren Verdrahtungsaufwand.

Auswahl der Umwandlungsmethode

Nachdem die drei Verfahren mit ihren jeweiligen Vor- und Nachteilen vorgestellt wurden, muss nun abgewogen werden, welche Methode realisiert wird. Durch Spezialisierung auf sich schnell ändernde Eingangssignale und die großen Probleme, Verzweigungen zu realisieren, fällt die Pipeline-Lösung weg. Für die Erstellung von beliebigen Programmstrukturen sind die verbliebenen Lösungen kongruent. Mit dem einfacher zu implementierenden kombinatorischen Prozess ist es nicht möglich, Zeitverhalten in synthetisierbarer Form zu erzeugen. Aus diesem Grund und dem geringeren Ressourcenverbrauch wird aus dem Struktogramm ein Mehrzyklus-Datenpfad erzeugt.

7.1.2 Generischer Mehrzyklus-Datenpfad

Automatischer Aufbau der Strukturen

Um den automatischen Aufbau eines Mehrzyklus-Datenpfades zu ermöglichen, muss von diesem eine möglichst universell einsetzbare Form entwickelt und verwendet werden. Dies gilt auch für alle beteiligten Funktionseinheiten und die Zusammenarbeit zwischen diesen. Auf besondere Strukturen, die in Sonderfällen für eine höhere Ausführungsgeschwindigkeit oder Ressourcenersparnis sorgen, wird zu Gunsten eines klar strukturierten Verhaltens verzichtet. Betrachtet werden nachfolgend die dazu ergriffenen Maßnahmen bei der Zustandsmaschine, den Registern, dem Datenpfad und dessen einzelnen Elementen.

Zustandsmaschine

Die zwei Stellschrauben, an denen die Zustandsmaschine vereinfacht werden kann, sind die Zustandsübergänge und die Anzahl sowie die Organisation ihrer Ein- und Ausgänge. Die Zustandsübergänge werden dahingehend vereinfacht, dass es sich entweder um einen bedingungslosen Zustandsübergang handelt oder zwischen zwei Folgezuständen ausgewählt wird. Für diese Auswahl wird ein einziges Eingangssignal ausgewertet. Auf dieses Bedingungs-Signal werden die Ergebnisse aller Komparatoren zugewiesen, um diese aus der Zustandsmaschine fernzuhalten und möglichst wenig Eingänge zu erhalten. Als Ausgänge liegen für alle Register Enable-Signale vor. Falls diese als Operanden verwendet werden, existieren für diese Selektions-Signale. Sollte von den Eingangsports gelesen werden, existiert ein Lese-Signal. Des weiteren werden die Signale, die bestimmen, welche Recheneinheiten oder Komparatoren zur Zeit aktiv sind, an den Datenpfad gesendet. In den Zuständen sind nur die Folgezustände und die Informationen über aktive Signale gespeichert, sodass vor der Auswertung alle anderen Signale auf Null gesetzt werden müssen.

Register

Um intern keinerlei Umrechnungen zwischen den verschiedenen Bitbreiten zu haben, wird immer mit 32 Bit gerechnet. Beim Einlesen externer Signale werden diese deshalb vorzeichenrichtig erweitert, um diese Informationen nicht zu vernichten. Dies soll die Verwendung des Q-Formats unterstützen. Den einzelnen Registern ist ein Multiplexer vorgeschaltet, der zwischen dem internen Ergebnissignal und dem Einspeichern externer Signale umschaltet. Dieses Ergebnissignal enthält das Rechenergebnis der zur Zeit ausgewählten Arithmetik-Komponente.

Datenpfad

Im Datenpfad befinden sich einerseits die Arithmetik-Einheiten als auch die Komparatoren. Auf diese Weise können beide Baugruppen wiederverwendet werden. Sowohl die Ausgangssignale der rechnenden als auch der vergleichenden Komponenten müssen im Datenpfad auf jeweils ein einzelnes Signal zugewiesen werden. Zu diesem Zweck gibt es den Bedingungs- und den Arithmetik-Multiplexer. Beide werden von der Zustandsmaschine über das Signal der zur Zeit aktiven Komponente gesteuert. Von diesen wird das Ausgangssignal der aktiven Komponente gewählt und dem Ergebnis- oder Bedingungssignal zugewiesen.

Komparatoren und Arithmetik-Ressourcen

Die einzelnen Komponenten sind dabei ähnlich aufgebaut. Sie verfügen über einen Eingangsmultiplexer, der durch Selektions-Signale gesteuert die verwendeten Register auswählt. Diese weisen sie auf interne Variablen zu, verrechnen diese und geben sie dann auf einem nach ihnen benannten Signal aus. Im Falle der Komparatoren befindet sich statt der Rechnung ein Komparator in der Komponente. Diesem kann eine Berechnung, z.B. ein UND-Gatter, vorgeschaltet sein.

Verbindung von Datenpfad und Zustandsmaschine

Die Verbindungen im Mehrzyklus-Datenpfad müssen vereinfacht werden, um diesen automatisch erzeugen zu können. Aus diesem Grund werden hinter den Arithmetik-Ressourcen und Komparatoren jeweils Multiplexer erzeugt. Diese fassen die Ausgänge der einzelnen Komponenten zu einem Ergebnis- und einem Komparator-Signal zusammen. Durch Selektion des gewünschten Ergebnissignals der Arithmetik-Komponenten kann dieses in den Registern verwendet werden. Das Komparator-Signal wird zusammengefasst und anschließend in der FSM für den Zustandsübergang verwendet.

7.2 Entwurf der Ersetzungsregel

7.2.1 Das algorithmische Zustandsdiagramm

Fähigkeiten des algorithmischen Zustandsdiagramms

Um das Verhalten einer Zustandsmaschine zu beschreiben, werden die sogenannten algorithmischen Zustandsdiagramme verwendet. Diese enthalten die Informationen, welche Berechnungen in dem jeweiligen Zustand ausgeführt werden, welche Signale erzeugt werden und welches der dann erreichte Folgezustand ist[28]. Im Englischen werden diese Diagramme als ASM-Charts bezeichnet.

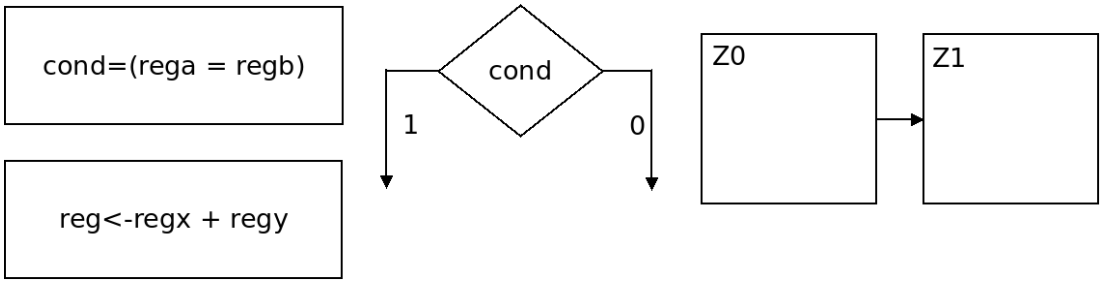


Abbildung 32: Grundelemente der ASM-Charts

Die Zustände werden dabei als Rechtecke mit dem jeweiligen Namen in der rechten, oberen Ecke dargestellt. In diesem enthalten sind die Vorgänge, die innerhalb des Taktes stattfinden. Es kann sich dabei entweder um eine Entscheidungsraute, Registerzuweisungen oder Mealy- und Moore-Ausgangssignale handeln (Abbildung 32).

Beschreibung der Ersetzungsregel

Um aus dem Nassi-Shneiderman-Diagramm VHDL-Quelltext zu erzeugen, muss für jeden Diagrammblock eine entsprechende ASM-Chart gefunden werden. Die verschiedenen ASM-Charts können dann verkettet und verschachtelt werden, um so die Funktionalität des beschriebenen Programms abzubilden.

Das Entwerfen dieser Blöcke stellt dabei die Entwicklung der Ersetzungsregel dar[29]. Dies ist ein sehr wichtiger, aber nicht der einzige Schritt, da für die Generierung aus den ASM-Charts anschließend noch der Quelltext erzeugt werden muss.

7.2.2 Ersetzungsregeln für die Diagrammstrukturen

Die Folge

Die Darstellung einer Folge in der ASM-Notation ist relativ leicht möglich. Es wird in dem jeweiligen Zustand das Rechenergebnis der Arithmetik-Komponente in das Zielregister getaktet. Anschließend findet ein unbedingter Zustandswechsel in den vorher bekannten Folgezustand statt. Zwischen, vor und nach den Zuständen können beliebig viele andere Konstrukte liegen (Abbildung 33).

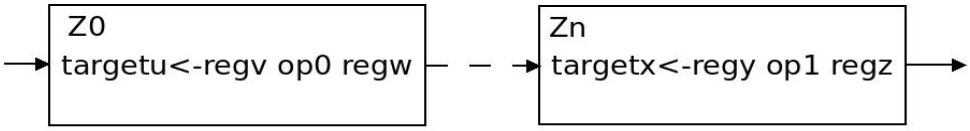


Abbildung 33: ASM-Chart der Folge

Die einfache Alternative

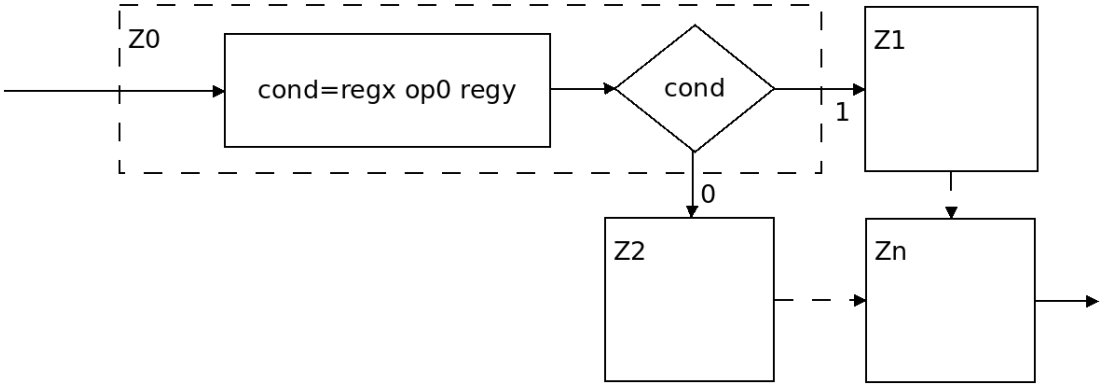


Abbildung 34: ASM-Chart der einfachen Alternative

Der Aufbau der Alternative ist wesentlich komplexer (Abbildung 34). So wird zuerst das Bedingungssignal erzeugt und anschließend mit einer Entscheidungsraute der Zustandswechsel festgelegt. Hierbei wird je nach Ergebnis des Komparators entweder der Folgezustand *Z1* oder *Z2* ausgewählt. Diese stehen als Platzhalter für alle in den Pfaden vorhandenen Strukturen. Anschließend werden diese zwei Ausführungspfade wieder im Zustand *Zn* zusammengeführt, um mit dem nächsten Schritt verkettet werden zu können.

Die mehrfache Alternative

Die mehrfache Alternative ermöglicht es, aus sehr vielen verschiedenen Pfaden auszuwählen. Da die Zustandsmaschine nur über einen einzigen Eingang verfügt, müssen die Bedingungen zur Auswahl des passenden Pfades sukzessive auf ihre Gültigkeit überprüft werden (Abbildung 35). Für jede der Prüfungen ist dabei ein Rechenschritt mit Komparator-Benutzung notwendig. Die Auswertung über den VHDL-Code „*case when=>*“ in der Zustandsmaschine wäre hier deutlich schneller. Allerdings würde nur durch diesen einen Knoten die Anzahl der benötigten Eingänge drastisch steigen, deshalb wird das Verhalten durch die Verkettung von einfachen Alternativen erreicht.

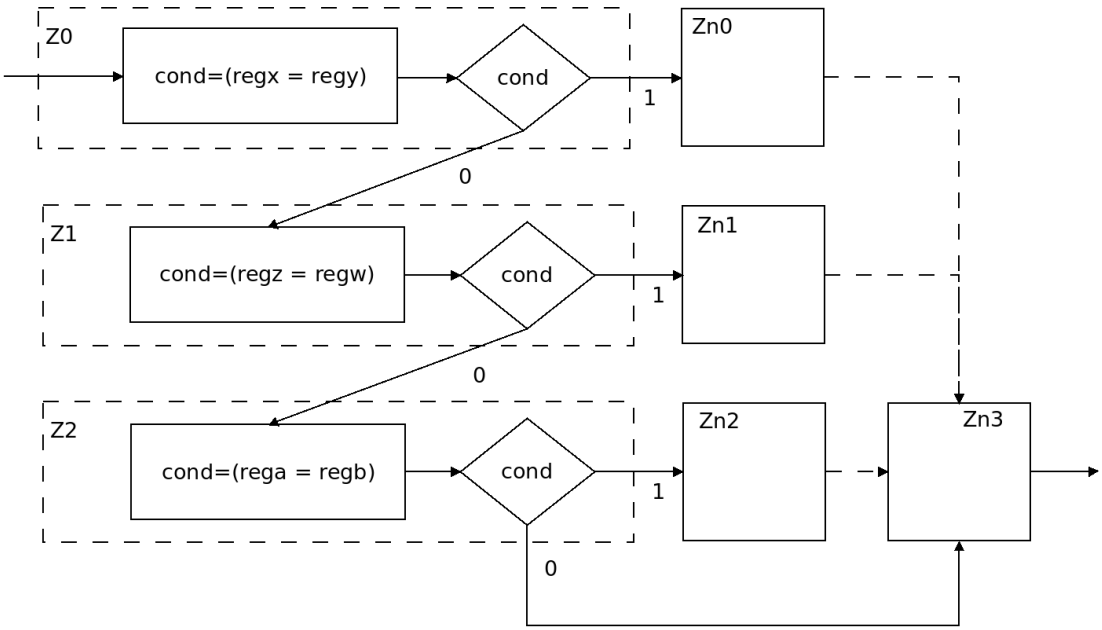


Abbildung 35: ASM-Chart der Mehrfachauswahl

Wiederholung mit vorausgehender Bedingungsprüfung

Die Wiederholung mit vorausgehender Prüfung liegt in zwei Variationen vor, einmal als Konstrukt, welches einer while-Schleife entspricht (Abbildung 36), und als Zählschleife. Sie sind in ihrem groben Aufbau sehr ähnlich, bei beiden wird, wie auch bei der Alternative, zuerst ein Komparator ausgewertet. Auf Grundlage von dessen Ergebnis wird der Folgezustand gewählt. Wenn die Bedingung nicht zutrifft, wird zu Zustand Zn+1 an das Ende der Schleife gesprungen, hinter diesem ist wieder Verkettung möglich. Trifft die Bedingung zu, wird der Zustand Z1 ausgewählt. Dies ist der erste Knoten des Schleifenrumpfs. Der Platz für Folgezustände ist durch die gestrichelten Linien angedeutet. Der Zustand Zn ist der letzte dieser Zustände, aus ihm wird in jedem Fall zurück zum Schleifenkopf gesprungen.

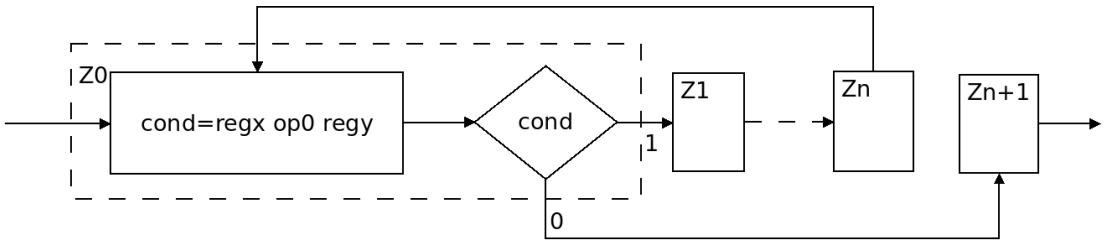


Abbildung 36: ASM-Chart der Wiederholung mit vorausgehender Prüfung

Die Wiederholung mit vorausgehender Prüfung, die als Zählschleife verwendet wird (Abbildung 37), verhält sich bis auf ein kleines Detail identisch zu der vorherigen. Es gibt die zusätzlichen Zustände Z-1 und Z1. Diese dienen der Initialisierung und der Erhöhung des Schleifenzählers.

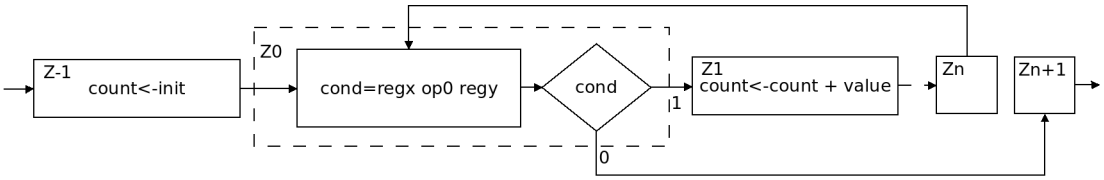


Abbildung 37: ASM-Chart der Zählschleife

Wiederholung mit nachfolgender Bedingungsprüfung

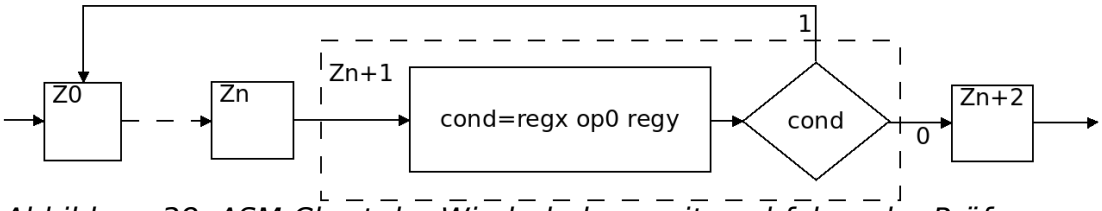


Abbildung 38: ASM-Chart der Wiederholung mit nachfolgender Prüfung

Die Wiederholung mit nachfolgender Bedingung (Abbildung 38) ist so aufgebaut, dass hinter dem Zustand Z0 der Schleifenrumpf beginnt und bis Zn reicht. In Zn+1 findet dann die Entscheidung statt, ob mit einem Sprung nach Z0 die Schleife noch einmal durchlaufen wird oder mit dem Wechsel zu Zustand Zn+2 die Schleife verlassen wird. Hinter diesem können dann wieder andere stehen.

Wiederholung ohne Bedingungsprüfung

Die Wiederholung ohne Bedingungsprüfung ist ähnlich aufgebaut wie die vorausgehenden Schleifen, allerdings ist sie durch den fehlenden Auswertungsschritt wesentlich weniger komplex (Abbildung 39).

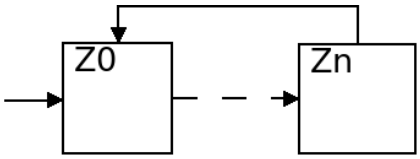


Abbildung 39: ASM-Chart der Wiederholung ohne Bedingungsprüfung

Übrige Blöcke

Die übrigen Blöcke sind trivial zu implementieren oder ihr Verhalten wird durch die bereits beschriebenen ASM-Charts erzeugt, letzteres ist bei dem Warten-Aufruf der Fall. Das Warten einer bestimmten Anzahl von Takten wird durch eine im Diagramm instanziierte Zählschleife erreicht. Dies setzt voraus, dass nicht exportierte Diagrammknoten während des Generierungsprozesses hinzugefügt werden können, die Datenstruktur ermöglicht dies.

Der Einlesen-Aufruf wird durch das Verbinden des entsprechenden Registereingangs mit dem Eingangsport und dem Setzen des zuständigen Clock-Enable-Signals realisiert. In der VHDL-Codeerzeugung hat der Parameterblock das gleiche Verhalten. Der Schreiben-Aufruf wurde so umgesetzt, dass der Ausgang des benannten Registers ständig auf den Ausgangsport gelegt wird.

7.2.3 Zeitverhalten der synthetisierten Konstrukte

Der Editor ist auf einfache, atomare Instruktionen beschränkt und erfordert dadurch kleinschrittiges Denken. Dies macht den Entwurf etwas aufwändiger, ermöglicht es jedoch, schon vor der Umwandlung zum synthesefähigen VHDL-Code, das Zeitverhalten des Algorithmus zu bestimmen. Durch die Kenntnis der Ersetzungsregeln, die das Nassi-Shneiderman-Diagramm auf eine ASM-Chart abbildet, lässt sich dieses präzise berechnen. In der nachfolgenden Tabelle (Tabelle 3) sind die dazu nötigen Formeln abgebildet, um das Zeitverhalten taktgenau bestimmen zu können. Anschließend kann durch das Einfügen von parametrisierten Warte-Aufrufen das Zeitverhalten an die Vorgaben oder eigenen Wünsche angepasst werden.

Knotentyp	**Formel**
Anweisung	*Aufwand=1 Takt*
Auswertung	*Aufwand=1 Takt*
Einfache Alternative	*Aufwand=Auswertung + Rumpf + 1 Takt*
Mehrfach Alternative	*Aufwand= Auswertung * Anzahl + Rumpf + 1 Takt*
Wiederholung vorausgehende Prüfung	*Aufwand=Auswertung+Anzahl*(Auswertung+Rumpf)+1 Takt*
Zählschleife	*Aufwand=Auswertung+Anzahl*(Auswertung+1Takt+Rumpf)+2Takte*
Wiederholung nachfolgende Prüfung	*Aufwand=(Anzahl+1 Takt)*(1 Takt+Rumpf+Auswertung)+1 Takt*
Endlosschleife	*Aufwand=Unendlich*(1 Takt + Rumpf)*
Warten-Aufruf	*Aufwand=Anweisung+(Parameter)*
Lesen-Aufruf	*Aufwand=1 Takt*
Schreiben-Aufruf	*Aufwand=1 Takt*

Tabelle 3: Arbeitstakte der Diagrammblöcke

Das Schlüsselwort Rumpf steht in der Tabelle entweder für die Takte, die der Schleifenkörper oder einer der alternativen Ausführungspfade verbraucht. Das Wort Anzahl gibt an, wie häufig eine Schleife durchlaufen wird. Bei den zusätzlichen Takten handelt es sich um Zustände, die den Programmfluss wieder zusammenführen.

7.3 Implementierungsdetails des VHDL-Codegenerators

7.3.1 Aufbau des Ersetzungssystems

Anschluss an die Datenhaltung

Die Erzeugung des VHDL-Codes ist analog zu der des C-Codes realisiert. Wie bei dem C-Codegenerator wird die Ersetzungsregel als *ReplacingRule* realisiert und die Klasse *AbstractReplacingRule* erweitert, um die grundsätzlichen Funktionen und Abläufe zu erhalten. Auch bei der Erzeugung von VHDL-Code wird die Diagrammdatenstruktur durchlaufen und die nötigen Informationen über das Netzwerk abgerufen. Die Umwandlung ist dabei wesentlich komplexer.

Im Gegensatz zu der Generierung von C-Code gibt es keine direkte Ersetzungsregel, die einen Block des Diagramms auf eine Zeile oder einen sequenziellen Abschnitt im erzeugten Quelltext abbilden. Für die VHDL-Codeerzeugung ist die Verwendung von Zwischenstrukturen nötig, die bei der Iteration und den rekursiven Abstiegen durch das Diagramm befüllt werden und anschließend die gespeicherten Informationen zu synthesefähigen VHDL-Code umwandeln. Das Befüllen und Erzeugen dieser Strukturen findet dabei in der Klasse *VHDLReplacingRule* statt. Diese führt auch alle Netzwerkoperationen auf dem Diagramm aus, sodass in den darunter liegenden Schichten die entsprechenden Problematiken nicht beachtet werden müssen.

Mehrzyklus-Datenpfad-Komponenten als Klassen

Die für die Erzeugung von VHDL-Code nötigen Klassen und Interfaces befinden sich im Paket *codegenerator.implementation.replacing.vhdl* und bilden ein geschlossenes Teilsystem (Abbildung 40). Die Strukturen des Mehrzyklus-Datenpfades werden dabei objektorientiert abgebildet. Die Schnittstellen *Entity*, *FSM* und *DataPath* stellen dabei die Schnittstellen zur Verfügung, die für die Erzeugung von Datenpfad, Zustandsmaschine und des Gesamtkonstrukts nötig sind.

Die Klasse *VHDLEntity* realisiert das Interface *Entity* und wird über Aufrufe durch die *VHDLReplacingRule* mit den Daten zur Codeerzeugung versorgt. Einige dieser Aufrufe werden in der Entität direkt verarbeitet z.B. bei der Erstellung von Ein- und Ausgabeports. Andere, wie die Deklaration von Registern, werden an die entsprechende Struktur weitergeleitet.

Die den Ablauf steuernden Zustände und Zustandsübergänge werden von ihr in der *FSM* erzeugt und mit den nötigen Informationen über die Steuergrößen versorgt. An den *DataPath* werden ebenso die Steuergrößen übergeben, von diesem werden die für die Ausführung nötigen Arithmetik Ressourcen erzeugt.

Die Klassen *VHDLDataPath* und *VHDLFSM* realisieren diese beiden Interfaces und verfügen ihrerseits über enthaltene Strukturen, die Zustände und die Arithmetik Komponenten. *VHDLDataPath* beinhaltet die Erzeugung der Multiplexer für die Arithmetik-Ressourcen und die Komparatoren. In dieser Klasse liegen die Ressourcen als Liste von *Arithmetic* vor, die durch *VHDLArithmetic* realisiert werden. Die

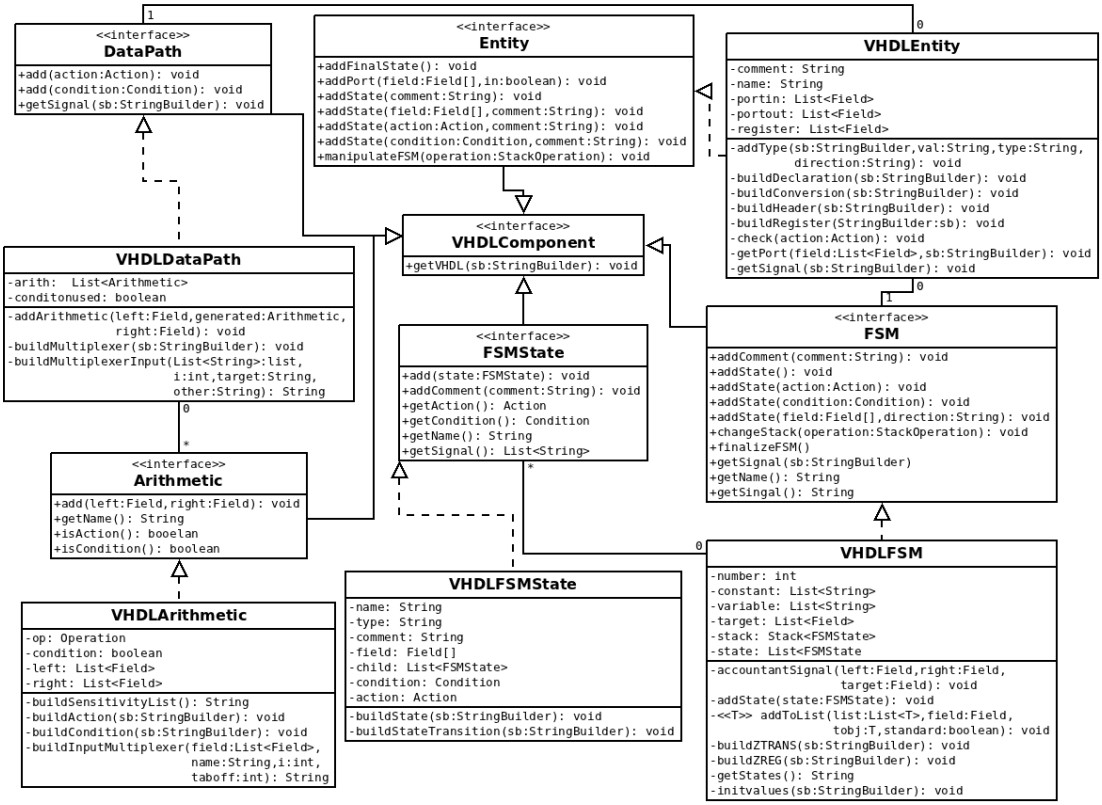


Abbildung 40: Klassendiagramm des VHDL Subsystems

Zustandsmaschine wird durch die Klasse *VHDLFSM* realisiert. Diese verknüpft und verwaltet die *VHDLFSMState,* die Implementierung des Interfaces *FSMState.* Diese stellt die eigentlichen Zustände dar. Da alle genannten Strukturen der Erzeugung von VHDL-Quelltext dienen, implementieren sie das Interface *VHDLComponent*, welches für diese Eigenschaft eine Methode einfordert.

7.3.2 Befüllen der Hilfskonstrukte

Erster Schritt der Codeerzeugung

Das Befüllen der Hilfskonstrukte ist der erste und anspruchsvollere Schritt bei der Codeerzeugung. Er ist auch wesentlich schwerer zu realisieren als die Erzeugung des Quelltextes aus den erstellten Konstrukten. Dabei besteht das Hauptproblem nicht in der Verkettung der Zustände, sondern in deren Verschachtlung. Bei dem Durchlauf das Diagramms müssen jedoch bei beiden Kombinationsmethoden Probleme umschifft werden.

Verkettung von Aktionen

Bei der Verkettung von Ausführungsblöcken kann es dazu kommen, dass der gleiche Operand in verschiedenen Zuständen an verschiedenen Positionen verwendet wird, ein Register also sowohl als linker und rechter Operand in Erscheinung tritt. Wenn für dieses nur ein Selektions-Signal verwendet werden würde, wäre die Reihenfolge im Multiplexer bestimmend dafür, dass der richtige Operand ausgewählt wird. Um dies zu verhindern, gibt es getrennte Selektions-Signale für die beiden Positionen des Operators.

Verschachtelung der Diagrammblöcke

Die Schwierigkeit bei der Verschachtelung von Diagrammstrukturen liegt darin, dass diese beliebig tief reichen kann. Hier muss sichergestellt sein, dass die Wechsel der Zustände den in den ASM-Charts angegebenen Vorgaben entsprechen. Der Folgezustand darf dabei nicht im Laufe des Vorgangs durch den einer niedrigeren Ebene überschrieben werden.

Die Verschachtlung wird über die FSM realisiert und mit Stack-Operationen erzielt. Zu diesem Zweck gibt es einen Puffer, einen Marker und den Stack. Der Marker zeigt jeweils auf den zuletzt angefügten Zustand. Der Puffer dient dem temporären Speichern von Stack-Elementen, um diese an den letzten Zustand anzufügen. Der Stack stapelt die Folgezustände, die im Laufe einer Rekursion noch nicht erreicht wurden. Das Vorgehen wird an dem folgenden Beispiel verdeutlicht (Abbildung 41).

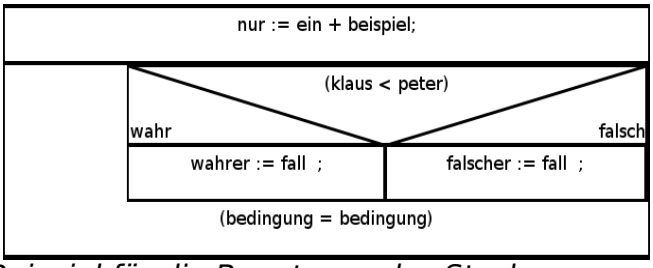


Abbildung 41: Beispiel für die Benutzung des Stacks

Der Zustandsbaum wird sukzessive aufgebaut (Abbildung 42).

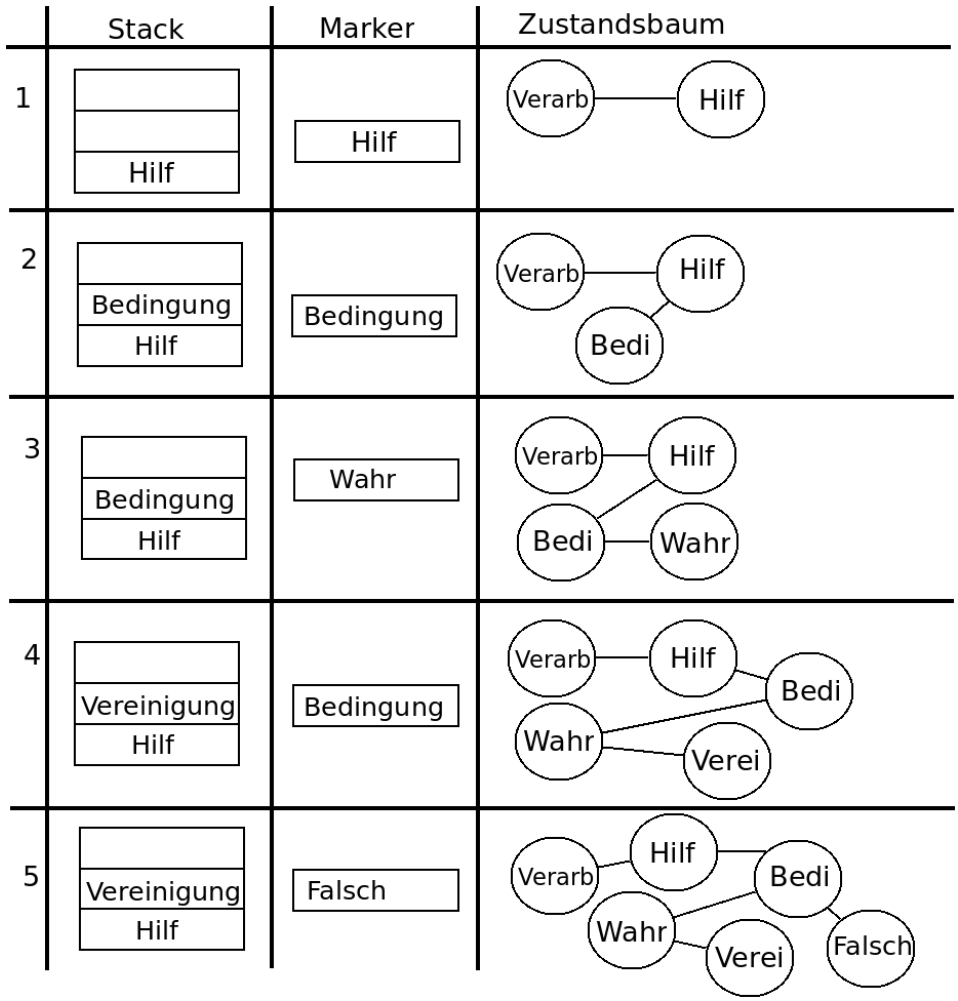


Abbildung 42: Aufbau des Zustandsbaums(1)

1. An den Verarbeitungs-Knoten wird der Hilfs-Knoten angefügt und zusätzlich auf den Stack gelegt.
2. Jetzt wird die Bedingung hinzugefügt und für den „Falschen“-Fall auf den Stack gelegt.
3. Es wird das Innere des Pfades hinzugefügt.
4. Es wird ein Block für die Vereinigung der Pfade hinzugefügt. Mit *Stack.pop()* wird das oberste Element geladen und der neue Block anschließend auf den Stack gelegt.
5. Jetzt wird der Pfad für den Negativ-Fall an die Bedingung angefügt.

Nach erreichen eines Maximums nimmt die Belegung des Stacks wieder ab (Abbildung 43).

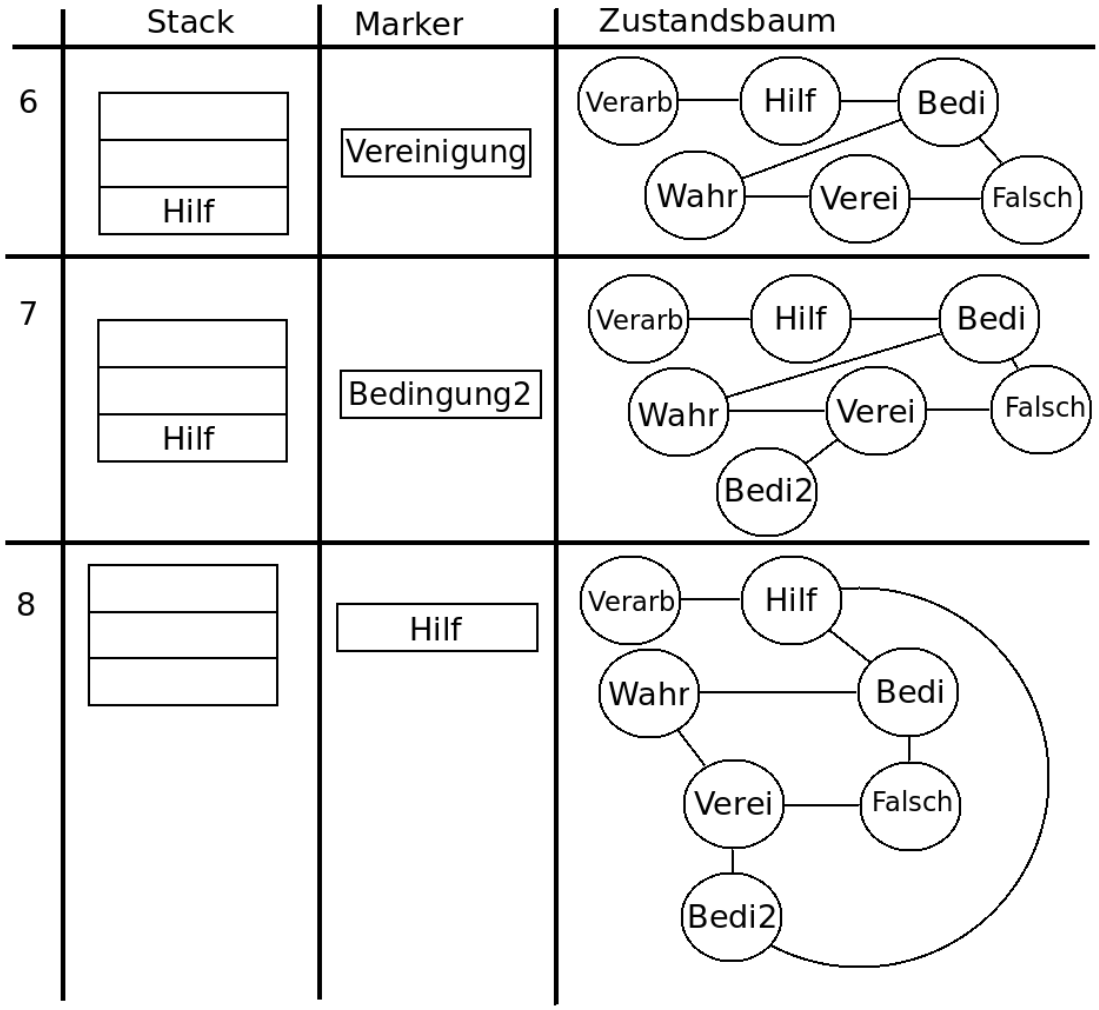


Abbildung 43: Aufbau des Zustandsbaums(2)

6. Vom Stack wird der Block für die Vereinigung der Bedingung geladen und an den Negativ-Pfad angefügt.
7. Jetzt wird die Bedingung der umschließenden Wiederholung angefügt.
8. An diese wird vom Stack die Vereinigung geholt und an den negativen Fall angefügt.

7.3.3 Verwendung der Hilfsstrukturen

Zweiter Schritt der Codeerzeugung

Nachdem durch den Durchlauf durch das Diagramm die nötigen Informationen durch die *RemoteStubs* eingeholt wurden, sind die Hilfsstrukturen bereit, Quelltext zu erzeugen. Dies geschieht durch einen von oben nach unten durchlaufenden Aufruf der *getVHDL()* Methode.

Ablauf der Codeerzeugung

1. Mit den Informationen der *VHDLEntity* wird die Portmap der Entität erstellt, dabei werden die Typen der Felder als Bitbreiten beachtet. Auch wird der Datentyp für die Zustände definiert.
2. Die Signale der Komponenten werden gesammelt und eingefügt.
3. In der *VHDLEntity* werden die Umwandlungssignale von innen nach außen und in die gegensätzliche Richtung erzeugt. Dabei werden die Bitbreiten und das Vorzeichen beachtet.
4. Nun wir die *VHDLFSM* umgewandelt. Der Prozess für den Zustandsübergang liegt hart kodiert vor. Die Übergänge werden über eine Iteration durch die Liste der VHDLFSMStates erzeugt.
5. Jetzt werden die Register angelegt und bei Bedarf deren Eingangs-Multiplexer hinzugefügt.
6. Es wird der *VHDLDataPath* hinzugefügt. Wie bei der Zustandsmaschine wird dabei die Liste der inneren Bestandteile iteriert. Die Eingangs-Multiplexer der Arithmetik-Komponenten werden durch rekursive Abstiege durch die Listen der linken und rechten Operanden erzeugt.
7. Im letzten Schritt werden die Multiplexer für die Arithmetik- und Komparator-Signale hinzugefügt.. Die Namen richten sich nach der Operation, bzw. dem Vergleich und der Komponentenart. Der Bedingungs- Multiplexer ist dabei optional.

8. Verifikation des erzeugten VHDL-Codes

8.1 Entwurf der Testfälle

8.1.1 Aufteilen der Testaktivitäten nach dem V-Modell

Tests an das V-Modell angelehnt

Die Überprüfung der Generierungs-Ergebnisse auf Vollständigkeit und Richtigkeit ist bei der Erstellung eines Codegenerators von besonderer Wichtigkeit. Nur durch aussagekräftige und zahlreiche Tests kann gezeigt werden, dass die Umwandlung von der Diagrammdatenstruktur hin zu VHDL-Quelltext richtig implementiert wurde.

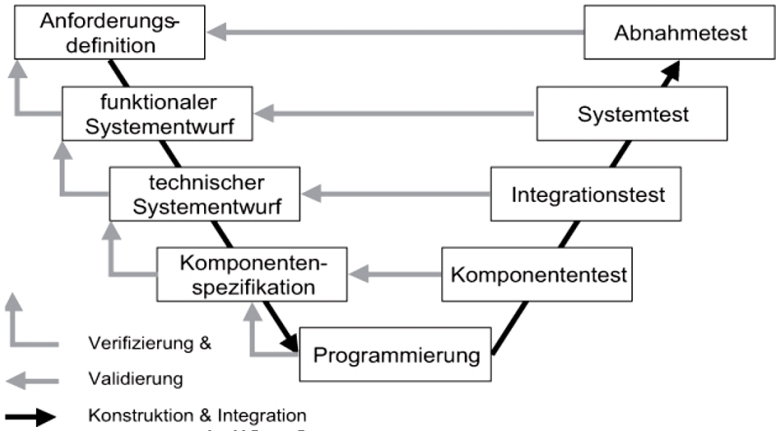


Abbildung 44: Das V-Modell[30]

Der Entwurf der Tests richtet sich dabei nach den Phasen des V-Modells[31]. Hierbei steht jedem Schritt in der Entwicklung eine Testphase gegenüber (Abbildung 44). Den Grundbausteinen der Software stehen die Modultests, dem technischen Systementwurf die Integrationstests und dem funktionalen Systementwurf der Systemtest gegenüber.

Getestet wird dabei in allen Phasen immer gegen die Spezifikationen. Diese bestimmen vor der Testdurchführung das Soll-Verhalten, sodass dieses nach dem Durchlauf mit dem Ist-Verhalten vergleichen werden kann. Besteht zwischen den Verhalt ein Delta, so wird der Testfall als fehlgeschlagen angesehen. Die Quelle des Soll-Verhaltens unterscheidet sich dabei natürlich von Phase zu Phase. Es werden die entsprechenden Entwurfsdokumente genutzt, dies ist der entscheidende Unterschied zwischen den verschiedenen Teststufen.

Aufteilung in Teststufen

Im konkreten Fall der Code-Generierung aus Modellen, wird dabei folgendes Schema benutzt. Die einzelnen Grundkonstrukte des Diagramms, also die Anweisung, die Verzweigung oder die Schleife, werden als Module aufgefasst. Für diese müssen also Modultests geschrieben werden. Die Verschachtelung und Verkettung der Basiskonstrukte wird als technischer Systementwurf aufgefasst. Hierfür müssen Integrationstests geschrieben werden, die das Verhalten bei der Verknüpfung überprüfen. Als letzte Teststufe wird im Systemtest überprüft, ob das Gesamtkonzept funktioniert, indem sinnvolle Komponenten erstellt werden. Es geht hier primär darum zu zeigen, dass das Konzept leistungsfähig genug ist, um auch größere Probleme zu lösen.

8.1.2 Realisierung der Teststufen

Modultests zur Verifikation der Diagrammblöcke

Im Modultest wird jede grundlegende Softwareeinheit für sich getestet. Auf diese Weise können gefundene Fehler sehr gut lokalisiert und frühzeitig entfernt werden. Wichtig bei dem Vorgehen ist es, dass die Komponente isoliert getestet wird. Abhängigkeiten zu anderen Bestandteilen oder das Testen mehrerer Grundbausteine muss, so weit wie möglich, vermieden werden. Es soll einzig und allein ein Grundbauteil getestet werden. Auf diese Weise können Einwirkungen anderer Bausteine ausgeschlossen werden.

Im konkreten Fall heißt das, dass die einzelnen Blöcke des Diagramms auf ihre Richtigkeit überprüft werden. Auf diese Weise wird festgestellt, ob für jeden dieser Bausteine die richtige Methode aufgerufen wird, und diese richtig implementiert wurde. Die dann überschaubare Codebasis des getesteten Bereichs ermöglicht es, den problematischen Abschnitt schnell zu finden.

Für die Erstellung von Modultests werden die Testfälle entweder auf Basis der bekannten inneren Struktur oder nur auf Basis der Spezifikation getestet. Tests zur Basis der Spezifikation nennt man Blackbox-Tests;

Tests, die die Innere Struktur berücksichtigen, heißen Whitebox-Tests. Der Name steht für die bekannte Implementierung[30]. Da in diesem Fall Entwickler und Tester zusammenfallen, wird der Vorteil genutzt, die innere Struktur zu berücksichtigen zu können. Die Testfälle werden dabei mit Hilfe der Grenzwertanalyse ausgewählt. Es wird dabei eine möglichst hohe Pfadabdeckung angestrebt.

In den Testfällen (Tabelle 4) äußert sich das wie folgt: Es werden jeweils die Stellen im Generator, an denen entschieden wird, welches Konstrukt gebaut wird, getestet. In den Blöcken werden die verschiedenen Verhaltensweisen abgeprüft. So wird bei Schleifen das einmalige und mehrmalige Durchlaufen, bei der einfachen Alternative die Ausführungspfade, bei der Mehrfachalternative kein und mehrere Fälle getestet. Die Testfälle an sich sind dabei relativ einfach gehalten, aber ziemlich nützlich. Die Erstellung der richtigen Arithmetik-Ressourcen wird durch die Verwendung aus jeder Kategorie erreicht. Dieses Prinzip wird auch bei den Komparatoren angewendet. Die Kategorien sind bei den Arithmetik-Einheiten Rechenoperationen und Schiebeoperationen, bei den Komparatoren Zahlenvergleiche und Vergleiche mit vorgeschalteten Gattern.

Die folgenden Testfälle wurden realisiert, die dazugehörigen Simulationsergebnisse befinden sich im Anhang.

Nr.	**Testfall**
1	Ausgabe eines Festwerts per Schreiben-Aufruf.
2	Einlesen und Ausgeben eines Wertes über die Aufrufe.
3	Verarbeitung mit zwei festen Werten, Typ Addition.
4	Verarbeitung mit gelesenem und festem Wert, Typ Minus.
5	Verarbeitung mit gelesenem Wert und festem Wert, Typ UND.
6	Verarbeitung mit zwei gelesenen Werten, Typ Schieben.
7	Einfache Verzweigung Auswertung auf „Wahr“.
8	Einfache Verzweigung Auswertung auf „Falsch“.
9	Wiederholungen mit vorausgehender Bedingungsprüfung, keinmal.
10	Mehrfache Wiederholungen mit vorausgehender Bedingungsprüfung.
11	Mehrfachauswahl, nur der „Ansonsten“-Fall.
12	Mehrfachauswahl, zwei Fälle.
13	Wiederholung mit nachfolgender Bedingungsprüfung, einmal.
14	Wiederholung mit nachfolgender Bedingungsprüfung, mehrmals.
15	Zählschleifen Wiederholungen, mehrmals durchlaufen.
16	Warte-Aufruf
17	Verarbeitung, Typ Zuweisung.
18	Verarbeitung, Typ Negation.
19	Unbedingte Wiederholung.
20	Einlesen, Typ Character.
21	Einlesen, Typ Boolean.

Tabelle 4: Durchgeführte Modultests

Integrationstest

Der Integrationstest testet die Zusammenarbeit der Komponenten und prüft so, ob diese Problemlos kombiniert werden können (Tabelle 5).

Nr.	**Testfall**
22	Registerinhalt als Operand auf beiden Seiten.
23	Register zweimal zum Speichern verwendet.
24	Verkettung aller Blöcke.
25	Verschachteln aller Blöcke.

Tabelle 5: Durchgeführte Integrationstests

Systemtest

Dieser Test prüft das Verhalten des Systems als Ganzes, der erfolgreiche Abschluss der vorherigen Teststufen ist Voraussetzung für die Durchführung. Ziel ist es zu überprüfen, ob das System benutzbar ist und die tatsächlichen Anforderungen der Benutzung meistert. Dafür werden komplexere Beispiele ausgewählt. Basis dieser Tests ist der funktionale Systementwurf. Die Tests befinden sich in Kapitel 9.

Entwurf des Regressionstests

Jede Änderung an dem Codegenerator sorgt dafür, dass sich neue Fehler eingeschlichen haben könnten. Um diesem Problem zu begegnen, müssten theoretisch alle Teststufen noch einmal durchlaufen werden. Durch die Notwendigkeit, Simulationen durchzuführen, ergibt sich daraus ein extrem großer Umfang.

Durch den Regressionstest wird eine Auswahl an immer einsatzbereiten Tests, die automatisch durchgeführt werden können, realisiert. Die Testfälle werden dabei mit lokal erzeugten Diagrammen erzeugt. Die Erstellung über das Interface bietet den Vorteil, dass die Implementierung der Datenstruktur angepasst werden kann. Das Laden einmal erstellter Diagramme über die Java Funktionalität *Serializable* würde dadurch nicht mehr möglich sein. Die Überprüfung, ob eine Generator-Ausgabe als valide angesehen wird, wird anhand von Referenzdateien durchgeführt. Für alle oben beschriebenen Testfälle und die Beispiele für die Systemtests wurde diese Funktionalität integriert.

Implementierung des Regressionstests

Realisiert wird die Erstellung und Ausführung der Testfälle in der Klasse *codegenerator.verification.TestGenerator*. Die Referenzwerte befinden sich im Unterpaket *reference*, die für die Simulation notwendigen *.do-Datein im Unterpaket *dofile*. Um auch aus einer *.jar-Datei Testfälle erstellen zu können, wird wie bei den Icons der GUI über *NetStructEDIT.class.getResource(<path>)* auf die Dateien zugegriffen. Es wird dabei nicht nur VHDL-, sondern auch der passende C-Quelltext generiert. Für die Systemtests werden zusätzlich Diagramme zum Laden gespeichert.

Testdokumentation im Anhang

Um die Bachelorarbeit nicht mit den zahlreichen Tests und deren Simulation zu überladen, befinden sich diese im Anhang. Da die einzelnen Simulationen sehr viele Signale enthalten, ist es jedoch ratsam, die automatische Generierung des Regressionstests zu benutzen, um bei Interesse die Fälle selber zu simulieren.

8.2 Betrachtung eines konkreten Testfalls

8.2.1 Auswahl eines beispielhaften Testfalls

Auswahlkriterien

Die vorherigen Auflistungen verdeutlichen den Umfang der Testtätigkeit. Es können einfach nicht alle Tests dargestellt werden. Um dennoch einen Einblick in die Vorgehensweise und Schwierigkeiten zu erhalten, wird das Vorgehen exemplarisch an einem Beispiel dargestellt.

Ausgewählt wurde der Test des Positiv-Falls einfachen Alternative. Dieser ist von überschaubarer Größe und zudem sieht man die Funktionsweise sowohl der Komparatoren als auch der Arithmetik-Komponente.

Spezifikation des Soll-Verhaltens

Der Test dient der Überprüfung der Kontrollflusskomponente „Einfache Verzweigung“. Es wird dabei das Verhalten bei einem Positiv-Fall überprüft (Abbildung 45). Das Verhalten wird anhand des generierten C-Codes gezeigt.

```c
int condtruewrite(void){
int ein1 = 12;
int ein2 = 8;
int name = 0;
//Einlesen(ein1)
scanf("%d", &ein1);
//(ein1 < ein2)
if (ein1 < ein2){
    name = ein1 + ein2 ;
}
//Schreiben(name)
printf("%d",name);

}
```

Abbildung 45: Verhalten des Testfalls als C-Code

Zur Laufzeit wird der Wert 7 eingelesen, um an einem Grenzwert zu testen. Die Variable „ein1“ wird vorher mit der Zahl Zwölf initialisiert, um sicherzustellen, dass der eingelesene Wert für die Zahl verantwortlich ist.

8.2.2 Testdurchführung

RTL-Schaltbild

Um die Ergebnisse des Generators und die Umsetzung des Testfalls deutlich zu machen, wird ein Schaltbild des Datenpfades verwendet (Abbildung 46). Die Register werden mit den Signalen „*ein1LEFT_SELECT_SI*“ und „*ein2RIGHT_SELECT_SI*“ in den Multiplexern selektiert. Das Signal „*LESS_COMP_COMPONENT_SELECT_SI*“ wählt den richtigen Komparator, „*ADDER_ARITH_COMPONENT_SELECT_SI*“ die zu benutzende Addition. Das Vorhandensein dieser Multiplexer verdeutlicht den generischen Charakter des erzeugten Quelltexters. Der Ausgang des „*CONDMUX*“ steuert die Zustandsübergänge der FSM über das Signal „*COND_SI*“, das Ausgangssignal des „*ARITHMUX*“ ist das aktuell erzeugte Ergebnissignal „*RESULT_SI*“. Dieses kann in den drei Registern verwendet werden. In Register „*ein1*“ besteht zusätzlich die Möglichkeit, das Eingangssignal „*ein1_IN_SI*“ einzulesen. Das Register „*name*“ verfügt über ein Ausgangssignal.

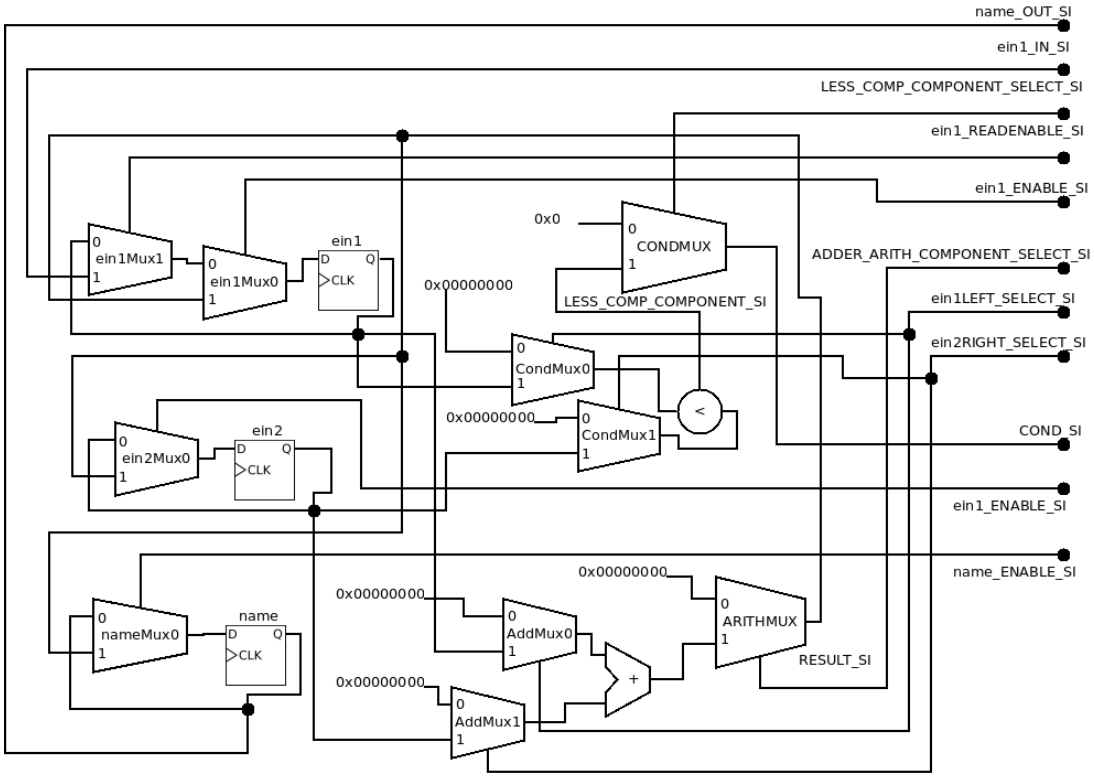


Abbildung 46: Datenpfad des Testfalls

Nicht im Bild befinden sich die Takt-Signale der Register. Da alle Register 32 Bit fassen, wird auf die Kennzeichnung der Leitungsbreiten verzichtet.

Simulation der Schaltung

Mit der Simulation (Abbildung 47) kann das Ergebnis überprüft und das Verhalten der erzeugten Struktur verdeutlicht werden.

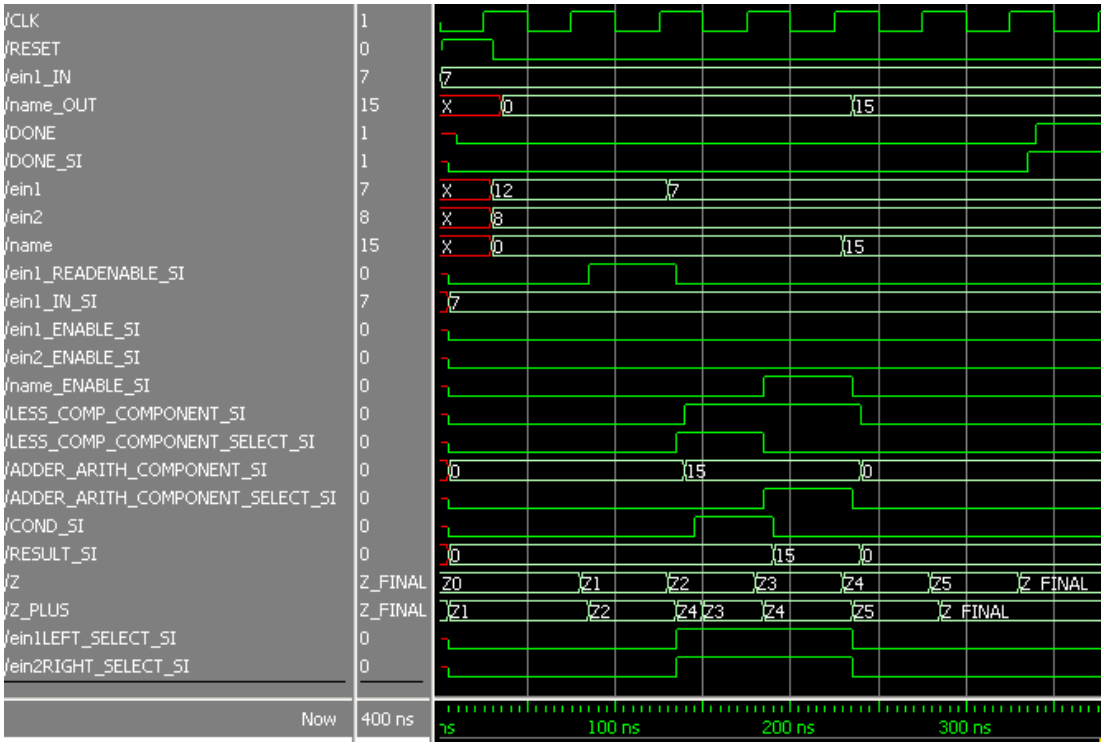


Abbildung 47: Simulation des Testfalls

Mit dem Reset-Signal werden die Register initialisiert und die Komponente in den Startzustand *Z0* gesetzt. Im Zustand *Z1* liegt das Read-Enable-Signal für das Register *„ein1“* an, sodass bei der nächsten Taktflanke das anliegende Signal *„ein1_IN“* gespeichert wird. Anschließend wird die Verzweigung ausgeführt. Dazu werden die Operanden mit den beiden Selektions-Signalen *„ein1LEFT_SELECT_SI“* und *„ein2RIGHT_SELECT_SI“* ausgewählt. Der Ausgang des „Kleiner“-Komparators *„LESS_COMP_COMPONENT_SI“* wird über das Selektions-Signal *„LESS_COMP_COMPONENT_SELECT_SI“* ausgewählt und dem Bedingungs-Signal *„COND_SI“* zugewiesen. Durch dieses wird der Folgezustand *Z_Plus* auf den Folgezustand *Z3* gesetzt. In diesem wird die Addition durchgeführt. Analog zum Verhalten des Komparators wird der Ausgang des Addierers *„ADDER_ARITH_COMPONENT_SI“* auf das Ergebnissignal *„RESULT_SI“* zugewiesen. Die Selektions-Signale der beiden Eingänge liegen weiterhin an, da die Operanden an den selben Positionen wiederverwendet werden. Durch das Signal *„name_ENABLE_SI“* wird das Ergebnis in das Register *„name“* gespeichert. Damit ist der Test erfolgreich abgeschlossen.

9. Komplexe Beispiele der Codegenerierung

9.1 Quadratwurzelalgorithmus

9.1.1 Erzeugung des Diagramms im Editor

Als ersten Praxistest des Codegenerators wird ein Algorithmus zur Ermittlung der Quadratwurzel herangezogen. Dieser bietet sich durch die überschaubare Größe und die Beschränkung auf Grundrechenarten für die Modellierung als Struktogramm (Abbildung 48) an. Das Modell wurde konsequent auf die Benutzung als einzelne Komponente optimiert. Zu diesem Zweck sind die Bitbreiten der Eingangs- und Ausgangssignale möglichst schmal gewählt und das Diagramm auf alleinstehende Verwendung optimiert.

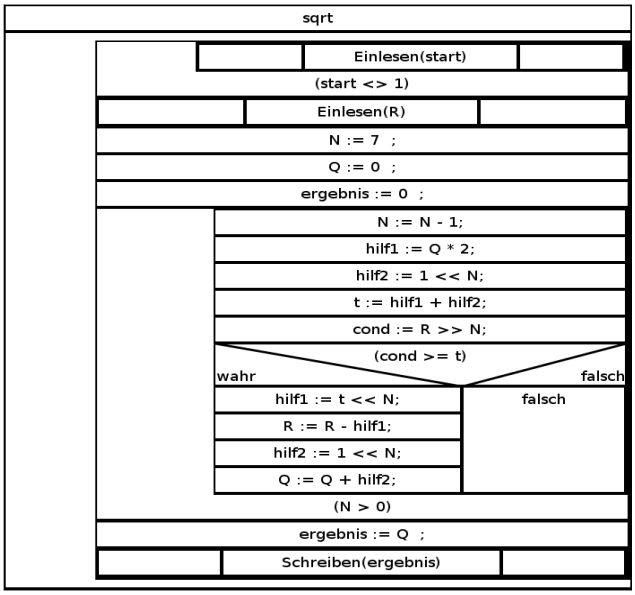


Abbildung 48: Wurzelalgorithmus als Struktogramm

Im ersten Schritt wartet der Algorithmus auf das Startsignal. Wechselt der Wert von „*start*“ auf eins, beginnt die Initialisierung. In ihr wird der Wert, für den die Wurzel bestimmt werden soll, eingelesen und die Variablen initialisiert. Anschließend werden in einer Schleife Stelle für Stelle die Ergebnisbits berechnet. Im letzten Schritt wird das Rechenergebnis umgespeichert und ausgegeben. Um ein reaktives System zu erzeugen, wird der beschriebene Ablauf von einer Endlosschleife umschlossen. Nach einem Durchlauf wird wieder auf das Startsignal gewartet, das Ergebnis bleibt bis dahin am Ausgang anliegen.

9.1.2 Betrachtung der Generator-Ausgabe

Nachdem der Algorithmus mit einem Struktogramm modelliert wurde, wird im nächsten Schritt das Verhalten des Modells überprüft. Um Fehler im Algorithmus oder dessen Modellierung aufzudecken, wird der Konsolen-C- Codegenerator verwendet. Dies ermöglicht es, eine grobe Funktionsprüfung durchführen zu können, ohne die für eine Simulation notwendigen Stimuli entwerfen zu müssen. Auch verbraucht dieses Vorgehen deutlich weniger Zeit.

Nachdem mit durch Ausführen und Compilieren des C-Code korrektes Verhalten festgestellt werden konnte, folgt im nächsten Schritt die Simulation des generierten VHDL-Codes. Dabei ist wichtig zu überprüfen ob die Rechenergebnisse den Erwartungen entsprechen und das reaktive Verhalten richtig umgesetzt ist. Zu diesem Zweck wird das Anschalten der Maschine, einmalige Durchlaufen, eine kurze Wartezeit, und ein erneuter Start mit anderen Werten simuliert (Abbildung 49).

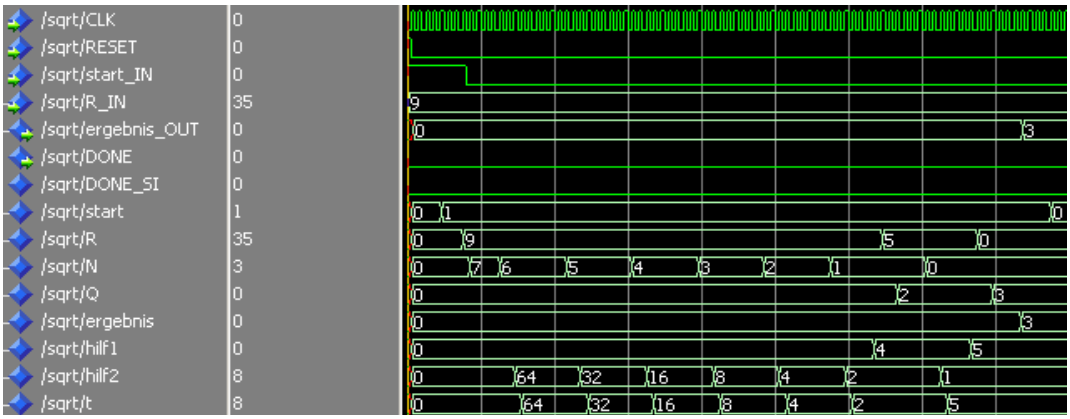


Abbildung 49: Erster Start des Wurzelalgorithmus

Die Testwerte wurden dabei nach der Grenzwertanalyse ausgewählt. Im konkreten Fall heißt das, dass die Werte an den Grenzen von der durch Integer-Rechnung vorhandenen Abrundung gewählt werden. Die untere Grenze wird durch den Eingangswert neun, die obere Grenze wird durch den Eingangswert 35 angetestet. Der Bildausschnitt ist so gewählt, dass

das Verhalten der einzelnen Register überprüft werden kann. Um eine leichte Lesbarkeit zu gewährleisten, wurden die Dezimalwerte der Signale angegeben.

An dem Signal *„ergebnis_out“* lässt sich ablesen, dass das System richtig arbeitet. Da der Wert der Variable *„N“* im Laufe des Algorithmus beständig schrumpft, ist an dem entsprechenden Register sehr gut die Position im Algorithmus ersichtlich. Es fällt dabei auf, dass die Anzahl der Takte, die der gleiche Wert in dem Register gespeichert ist, variiert. Dieses Verhalten erklärt sich durch die Verzweigung, deren Ausführung im positiven Fall zusätzliche Zeit kostet.

Die untere Grenze wird also richtig berechnet. Der nächste Test prüft gleichzeitig die obere Grenze der Berechnung und den erneuten Start eines Systems ab.

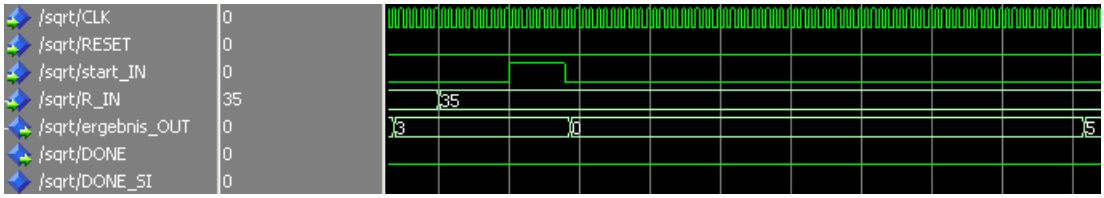


Abbildung 50: Zweiter Start des Wurzelalgorithmus

In der Abbildung (Abbildung 50) sind zwei wichtige Dinge zu erkennen. Einmal sieht man, wie der Wert der letzten Berechnung bis zum erneuten Startsignal ausgeben wird. Auch ist zu erkennen, wie der neue Wert erst verwendet wird, wenn das Signal dazu gegeben wird. Die Integer Wurzel von 35 ist fünf, somit funktioniert das Gerät auch an der Oberen Grenze.

Da die Erzeugung nach einem festen Schema erfolgt, kann aus dem Diagramm abgelesen werden wie viele Ressourcen verwendet werden. Die Anzahl der Register entspricht dabei den der verwendeten Variablen und für jede im Algorithmus verwendete Rechen- und Vergleichsoperation muss eine Arithmetik-Ressourcen bzw. Komparator instanziiert werden. Da das Zeitverhalten von konkreten Eingangswerten abhängig ist, lässt sich dieses nur von der der Simulation ablesen. Mit einer Kombination der Informationen aus den beiden Quellen lassen sich so die wichtigsten Kennwerte der generierten Komponente ablesen.

Ressourcentyp	**Verwendet**
Register	8 Register mit je 32 Bit
Arithmetik	Minus, Plus, Mal, Zuweisung und zwei Schieberegister
Komparator	Ungleich, Größer Gleich, Größer
Arbeitstakte	65 Takte

9.2 Sinusgenerator

9.2.1 Mathematische Grundlagen

Differenzengleichung für den Sinusoszillator

Das zweite Beispiel welches erzeugt werden soll, ist ein Sinusgenerator. Dieser wird mit Hilfe einer Differenzengleichung realisiert.

Die dafür verwendete Formel lautet :

$$A = Amplitude$$
$$f_0 = Gewünschte\ Frequenz$$
$$f_A = Abtastrate$$
$$\omega' = \frac{2 * \pi * f_0}{f_A}$$

$$K = 2 * \cos(\omega')$$
$$y(n-1) = 0$$
$$y(n-2) = -A * \sin(\omega')$$
$$y(n) = K * y(n-1) - y(n-2)$$

Der aktuelle Wert der Gleichung ist nicht von Eingangsgrößen abhängig, sondern nur von den beiden vorherigen Ergebnissen und der Konstante K. Die Gleichung wird durch wohldefinierte Startwerte und eine passend gewähltes K zum Schwingen gebracht. Parametrisiert wird die Schwingung über die gewünschte Amplitude und Frequenz, die Abtastrate ist vorgegeben.

Anpassung an den diskreten Zahlenraum

In der Formel ist die Verwendung von reellen Zahlen vorgesehen, die am ehesten den Fließkommazahlen entsprechen. Diese beherrscht der Editor zur Zeit ebenso wenig wie die für die Berechnung der Konstanten und des Startwertes von y(n-2) nötigen mathematischen Funktionen.

Die Problematik mit den Konstanten wird pragmatisch gelöst, in dem diese extern berechnet und dem System nur übergeben werden. Da die Gleichung zwischen -1 und 1 schwingt, müssen die vorhandenen Integerwerte für Festkommaarithmetik benutzt werden. Intern haben die Register 32 Bit. Durch die nötige Multiplikation würde die Verwendung der vollen Bitbreite für die Rechengenauigkeit zu Überläufen führen. Da es sich um vorzeichenbehaftete Werte handelt, muss das erste Bit als Vorzeichenbit verwendet werden, so dass statt 16 maximal 15 Bit für den Zahlwert übrig bleiben. Da K maximal den Wert zwei annehmen kann, fällt ein weiteres Bit weg. Für die Implementierung werden die Startwerte und K als Fließkommazahlen berechnet, mit 2^{14} multipliziert und dann zu Integerwerten umgewandelt. Die Experimente zu der Umwandlung des Algorithmus finden sich in *example.sinus.Sinus*.

9.2.2 Implementierung über den Diagramm-Editor

Diagramm

Der Sinusgenerator (Abbildung 51) an sich braucht keine Eingänge und nur einen Ausgang für den Zahlenwert. Dieser wird mit dem größten möglichen Datentyp realisiert. Um eine Anpassung des Generators zu ermöglichen, werden die extern berechneten Konstanten als Parameter übergeben.

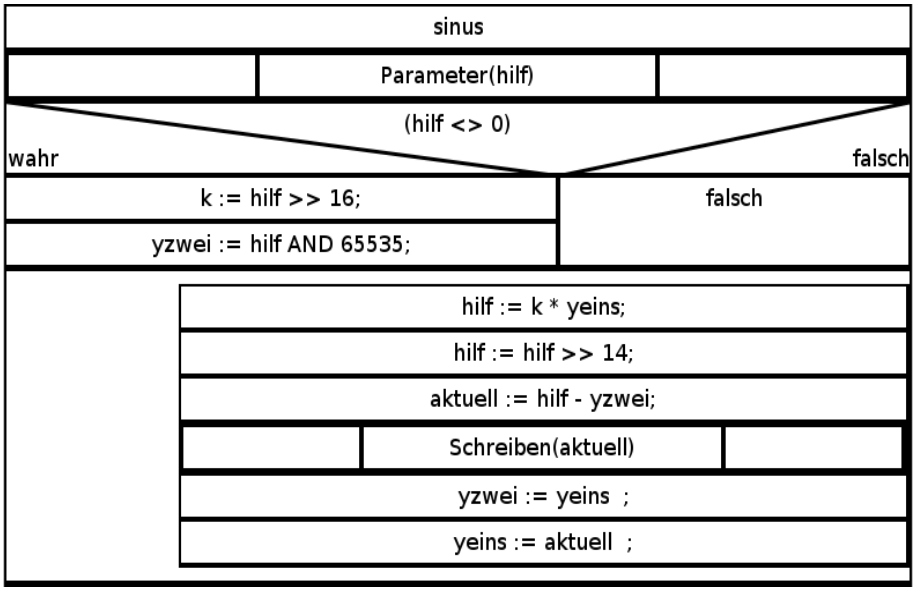


Abbildung 51: Struktogramm des Sinusgenerators

Um die Bitbreite des durch den Parameterblock erzeugten Eingangs, gering zu halten, werden die beiden konstanten Werte zusammen eingetaktet. Die vorderen 16 Bit werden für die Konstante k, die hinteren 16 Bit für den Startwert von y(n-2) verwendet.

Nachdem diese Werte eingelesen wurden, wird wieder eine Endlosschleife verwendet. Die eigentliche Differenzengleichung wurde in die Einzelschritte zerlegt und um eine Größenkorrektur erweitert. Diese ist nötig, da durch die Multiplikation zweier Werte, die größer als 1 sind, der Arbeitsbereich verlassen würde. Die Anpassung wird durch einen Rightshift um 14 Stellen erreicht.

Aus dem Diagramm müssen nun die benötigten Arbeitstakte abgelesen werden, dies ist nötig, um die mögliche Abtastrate zu berechnen:

$$Arbeitstakte = 4 * Anweisung + 2 * Zuweisung + Schreiben + Endlosschleife = 8\,Takte$$

Parametrisierung des Sinusgenerators

Die maximal mögliche Abtastrate richtet sich nach der Frequenz des verwendeten FPGAs und den benötigten Arbeitstakten der Schaltung. Konkret wurden die folgenden Parameter für ein Beispiel ausgewählt:

$$A = 1 \qquad \omega' = \frac{2 * \pi * f_0}{f_A} = 0{,}55292$$

$$n_{Sinus} = 8 \qquad K = 2 * \cos(\omega') = 1{,}702$$

$$f_0 = 440\text{Hz} \qquad y(n-2) = -A * \sin(\omega') = -0{,}52517$$

$$f_{FPGA} = 40\text{KHz} \qquad f_A = \frac{f_{FPGA}}{n_{Sinus}} = \frac{1}{8} * 40\text{KHz} = 5\text{KHz}$$

Die Frequenz des FPGAs wurde mit Absicht unrealistisch niedrig gewählt, um das Beispiel anschaulich zu halten. Durch die Abtastfrequenz von 5KHz können sind die Nulldurchgänge und nach zehn Iterationen ein vollständige Schwingung zu sehen. Die Abtastfrequenz erfüllt die Forderungen des Nyquist-Shannon-Abtasttheorems, indem sie mehr als doppelt so hoch ist wie die Zielfrequenz[32].

Verifikation über Testwerte

Zur Verifikation wird wieder ein per C-Codegenerator erzeugtes Programm verwendet. Mit diesem wurden die ersten 100 Werte erzeugt und mit der Software GNU Octave geplottet. Dazu wird der Befehl *`plot(A); xlabel("Abtastvorgang");ylabel("Ausgabewert"); print("sinus.png")`* verwendet, A ist dabei der Vektor, der die Zahlenwerte enthält.

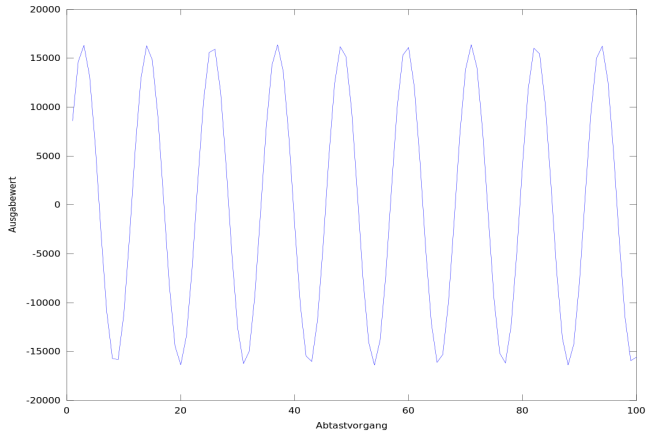


Abbildung 52: Octave-Plot der Sinusschwingung

Wie erwartet sind zehn Schwingungen (Abbildung 52) zu sehen. Die Abtastposition wandert durch die Sinusschwingung, da die Abtastfrequenz kein Vielfaches der Zielfrequenz ist.

Simulation des Sinusgenerators

Nach der Berechnung der Parameter für die Differenzengleichung und der Verifikation des verwendete Algorithmus durch den C-Quelltext, wird nun die Simulation (Abbildung 53) in Modelsim durchgeführt. Da beide Implementierungen die gleichen Datentypen und Operationen verwenden, wird die Richtigkeit des VHDL-Quelltextes durch die identischen Ausgabewerte bestätigt (vgl. Abbildung 52).

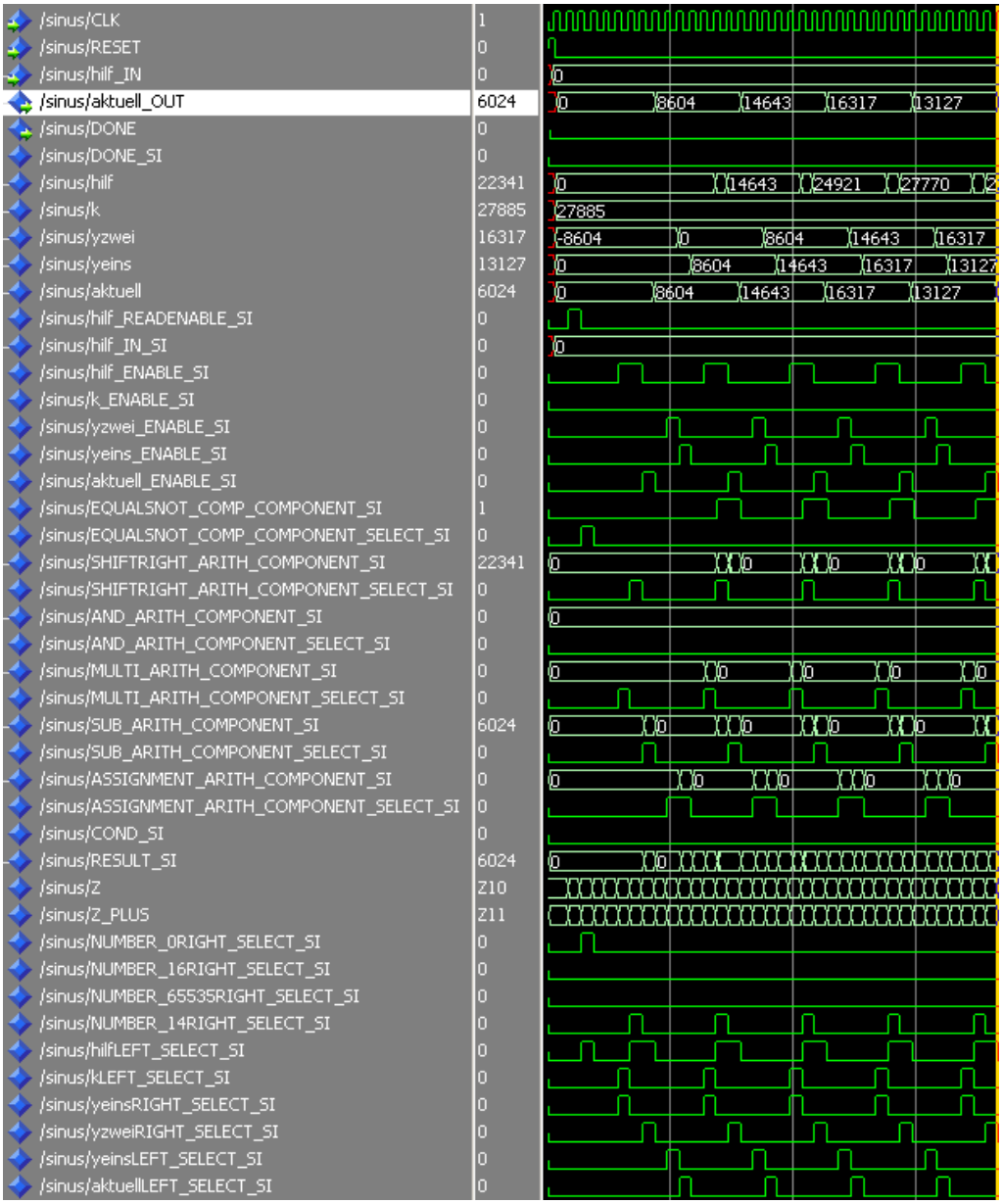


Abbildung 53: Sinusgenerator mit allen Signalen

9.3 Fahrstuhlsteuerung

9.3.1 Entwurf der Fahrstuhlsteuerung

Besondere Eigenschaften der Fahrstuhlsteuerung

Die Fahrstuhlsteuerung ist wie der Wurzelalgorithmus eine Praktikumsaufgabe im Computer Engineering Praktikum. Es wird ein Fahrstuhl mit Tür, drei Stockwerken und eben so vielen Zielknöpfen simuliert. Die Aufgabe der Steuerung ist es, eine optimale Route für die Haltewünsche zu finden und den Fahrstuhl richtig zu bewegen. Natürlich sollte die Tür während der Fahrt nicht geöffnet sein.

Diese an sich einfache Aufgabe enthält zwei verschiedenartige Probleme. Zum einen ist die Berechnung des optimalen Weges alles andere als trivial, zum anderen sind parallele Prozesse nötig. Durch die Reduzierung der Wegfindung auf eine gute Lösung nimmt die Komplexität der Berechnung stark ab, das Struktogramm bleibt trotzdem komplex. Die fehlende Parallelität tritt bei einer Fahrt mit zwei Zielen negativ auf. So muss der Fahrstuhl warten und es können Knöpfe gedrückt werden, hier muss eine Lösung gefunden werden. Jedoch lässt sich das Entprellen der Knöpfe und das Zeitverhalten nicht sauber trennen.

Aufteilung des Fahrstuhlproblems in Teilprobleme

Da keine parallele Lösung möglich ist, muss das Verhalten der Steuerung in verschiedene Phasen unterteilt werden. Eine geschickte Anordnung der Arbeitsabläufe ermöglicht dann die Realisierung der Funktionalität.

Dabei wird das Gesamtproblem in die Aktivitäten Bewegen, Abfragen der Haltewünsche, Vermerken der Haltewünsche und Berechnen der Route zerlegt. Das Bewegen ist der einfachste Teilschritt, es wird die Position in Abhängigkeit vom aktuellen Ziel verändert. Wird ein Ziel erreicht, gibt es einen extra Warteschritt, um das Aussteigen zu ermöglichen. In diesem werden die Haltewünsche abgefragt. Da dies sonst auch nach der Berechnung des Ziels passiert, musste dies von dem Vermerken der Wünsche entkoppelt werden. Die Wünsche werden dabei über eine Bitmaske zu den bereits bestehenden hinzugefügt. Über eine Mehrfachauswahl wird dann auf Basis dieser Wünsche das Zielstockwerk eingestellt und eine Maske zum Löschen des erreichen Stockwerks erstellt. Die Wünsche bleiben so lange aktiv, bis diese Löschenmaske verwendet wurde.

9.3.2 Realisierung der Fahrstuhlsteuerung

Die direkte Gegenüberstellung zeigt, dass das Ist-Verhalten (Abbildung 55) und die Belegung der Register mit dem Soll-Verhalten übereinstimmen (Abbildung 54).

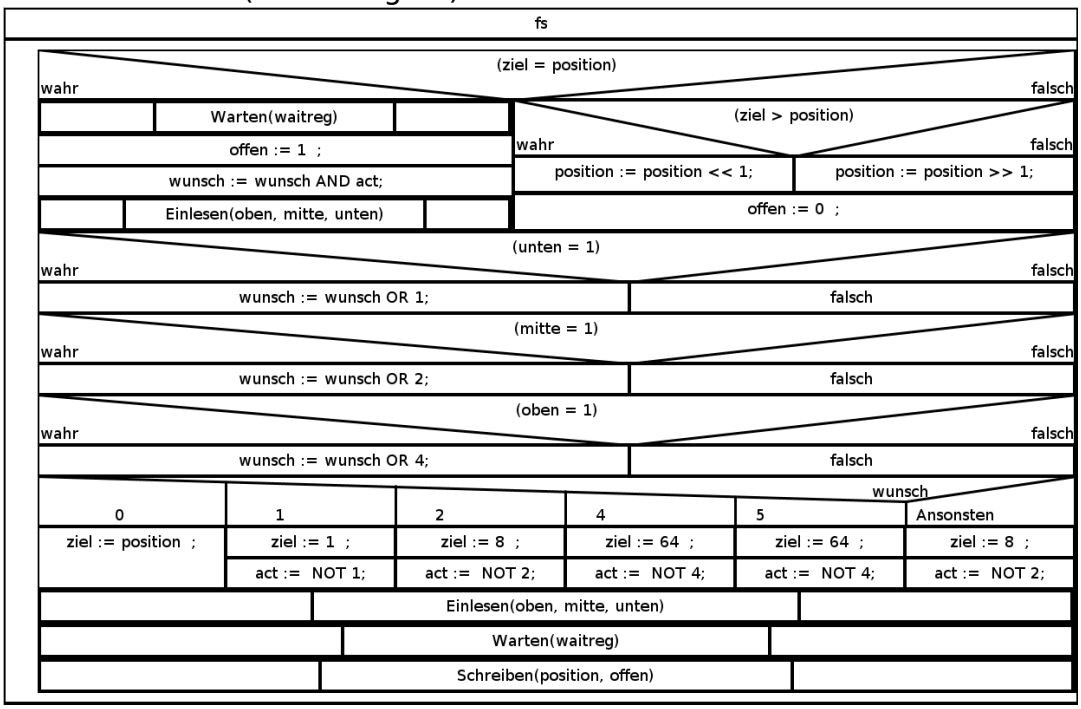


Abbildung 54: Struktogramm der Fahrstuhlsteuerung

Zu Beginn ist kein Stockwerk ausgewählt, nach einer Wartezeit wird das mittlere Stockwerk als Ziel markiert. Durch Schiebeoperationen erreicht der Fahrstuhl wie geplant die Position drei und damit das Zielstockwerk.

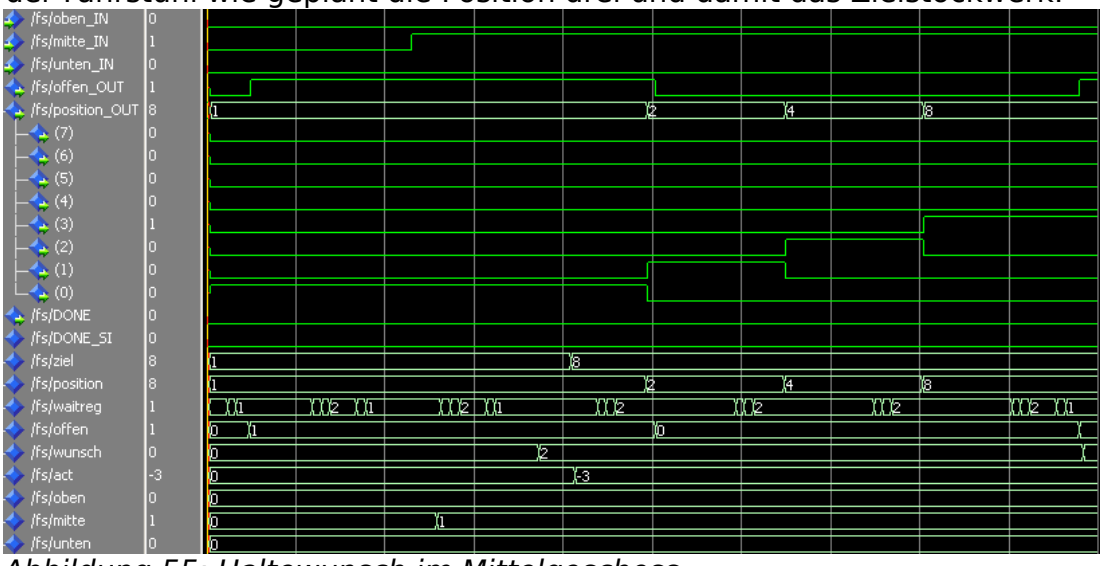


Abbildung 55: Haltewunsch im Mittelgeschoss

10. Ausblick

10.1 Implementierung der Kernfunktionalität

Neben den von vornherein geplanten, aber nicht notwendigen Funktionen wurden andere, sinnvolle Erweiterungen erst im Laufe der Arbeiten oder bei der Verwendung des fertigen Systems aufgedeckt. Diese konnten dann nicht mehr integriert werden, da sie bei der Zeitplanung nicht berücksichtigt wurden. Auch die Tatsache, dass nur eine einzige Person an diesem Projekt arbeitete, führte dazu, dass an vielen Stellen zwangsweise Einschränkungen vorhanden sein mussten.

Obwohl bis jetzt die gesamte Entwurfsarbeit, als auch die Implementierung von einer einzelnen Person durchgeführt wurden, ist das Projekt so angelegt, dass sich andere Menschen leicht einarbeiten können. Dies wird durch eine saubere Codebasis und die vorhandene Dokumentation erreicht. Die Benennung der Programmiereinheiten als auch deren innere Strukturen richten sich nach den Java-Codekonventionen. Dies ermöglicht es im Zusammenhang mit der starken Kapselung durch die Ableitungshierarchien der Klassen und ihrer Interfaces zu navigieren und die Stellen aufzufinden, die bearbeitet werden müssen. Selbstverständlich wird der Code ohne Warnungen kompiliert. Um es möglichst vielen Menschen zu ermöglichen, an dem Projekt weiter zu arbeiten, wurde die gesamte Implementierung in englischer Sprache durchgeführt. Für alle Variablen-, Attribut-, Methoden- und Klassennamen wurden englische Bezeichner verwendet. Die Namen sind auf diese Weise nicht nur international verständlich, sondern zumeist auch deutlich kürzer. Natürlich ist auch die Javadoc auf Englisch verfasst, um dort keine Sprachbarriere aufzubauen.

10.2 Mögliche Erweiterungen des Editors

Unterprogramme und Bibliotheksfunktionen

Die größte Einschränkung bei der Erstellung von Diagrammen mit Hilfe des Editors ist das Fehlen der Möglichkeit, Unterdiagramme zu erstellen oder zu referenzieren. In den Nassi-Shneiderman-Diagrammen ist vorgesehen, dass der Verarbeitungsblock nicht nur für einfache Zuweisungen und Rechnungen benutzt wird, sondern auch ganze Rechenschritte repräsentieren kann.

Dies ermöglicht es, häufig benutzte Anweisungsabfolgen an verschiedenen Stellen wieder zu benutzen und komplexe Funktionen in nur einer grafischen Anweisung abzuhandeln. Diese Verwendung von Unterdiagrammen ist ein Merkmal des hierarchischen Entwurfs und der damit zusammenhängenden Top Down Programmierung. Deren Einführung und die Unterstützung dieser Programmierdoktrin im Entwurf war eines der Entwicklungsziele der Struktogramme. Auch der Aufruf von Bibliotheksfunktionen, also z.B. komplexere mathematischen Anweisungen, könnte auf diesem Wege elegant in den Editor integriert werden.

Um die besonderen Verarbeitungsblöcke von den allgemeinen zu unterscheiden, ist es üblich, ein nicht in der DIN 66261 standardisiertes Symbol zu verwenden. Es wird schon von den unter „Aufruf“ zusammengefassten Blöcken, also den Lese-, Schreib-, Parameter- und Warteblöcken, zur Darstellung im Editor verwendet. Für die Anzeige dieses neuen Blocktyps sind also keine Arbeiten an der GUI mehr nötig, die vorhandene Blockgrafik müssten dem neuen Knotentyp nur bei der Darstellung zugewiesen werden. Die Veränderungen in der Datenstruktur sind etwas größer, in diese müsste ein allgemeiner Funktionsaufruf integriert werden. Der neue Knotentyp könnte ähnlich wie die bisherigen Aufrufe realisiert werden, bei denen die Parameter als Felder übergeben werden und die sich ansonsten wie die übrigen Knoten verhalten. Um das Unterdiagramm zu bestimmen, wäre es denkbar, dass dieses über die Oberfläche aus den auf der gleichen Datenhaltung liegenden Diagrammen ausgewählt wird. Die Referenz auf den Wurzelknoten des Unterdiagramms könnte dann wie ein Schleifenrumpf an den Codegenerator weitergegeben werden.

Im Codegenerator selber würde sich das Verhalten bei den verschiedenen Sprachen stark unterscheiden. Während bei der Erzeugung von C-Code nicht mehr zu tun ist als das Einbinden von Header-Dateien und die Ausgabe des Funktionsnamens samt Aufruf-Parametern, gestaltet sich die Einbindung der neuen Funktionalität in den VHDL-Codegenerator als deutlich schwieriger.

Sinnvoll wäre es, bei diesem die Bibliotheksfunktionen und Unterdiagramme unterschiedlich zu behandeln. Für die Bibliotheksfunktion könnte eine VHDL-Komponente instanziiert werden, deren Ein- und Ausgangsports dann mit den Registern verbunden werden, die über die Felder als Parameter definiert wurden. Die Bearbeitung von Unterdiagrammen im Codegenerator könnte ähnlich wie die bereits implementierte Wartefunktionalität realisiert werden. Das Unterdiagramm würde während des Generierungsprozesses in den Baum des Oberdiagramms eingeklinkt und mit diesem traversiert werden. Im Ergebnis würde eine Zustandsmaschine mit erweitertem Zustandsraum und ein um die benötigte Funktionalität erweiterter Datenpfad erzeugt werden. Problematisch sind hier die initialen Zuweisungen der Registerwerte und generell Probleme mit den Namensräumen der verbundenen Diagramme. Hierfür müsste eine Methode entwickelt werden, die für beliebige Verschachtelungstiefen konsistente Namensräume erstellt.

Höherer Komfort im Editor

Diagramme, die zur Visualisierung eines Softwaresystems verwendet werden, sind häufig in deren Dokumentation zu finden. Um die im Diagrammeditor erstellten Grafiken zu exportieren, ist zur Zeit noch der Einsatz externer Software nötig. So müssen mit dem Snippingtool oder GIMP Bildschirmfotos von dem Fenster des Editors erstellt werden. Hier wäre es gut, wenn die Software über die Möglichkeit verfügen würde, die angezeigte Grafik ohne Fensterdekoration in einem Schritt als Grafikdatei zu exportieren. Dies könnte über die Klasse *Robot* und der Kenntnis der Position des *Jpanels*, in dem das Diagramm angezeigt wird, geschehen.

Um generell die Arbeit im Diagramm einfacher und effizienter zu gestalten, wäre es hilfreich, Diagrammteile verschieben, kopieren und einfügen zu können. Ein Anwendungsfall ist hier das Verschieben des Schleifenrumpfs einer Endlosschleife in den wahren Fall einer Einfachauswahl. Zur Realisierung müsste der *DiagramHandler* um eine Liste von gepufferten Teildiagrammen erweitert werden. Auf diese Weise müssten diese nicht wieder exportiert werden. Das Interface zum Einfügen von Knoten ist mit der Methode *addNode(Diagram)* so gestaltet, dass hier theoretisch auch Teilbäume hinzugefügt werden könnten. Es würde dann der erste Knoten des Teildiagramms übergeben und dessen letztes Element würde der neue Elternknoten für den alten Kindknoten.

Auch das Erstellen von komplizierten Gleichungen in einer Verarbeitung wäre denkbar. Hier müsste entweder die Aktion erweitert werden oder automatisch Hilfsvariablen erstellt werden.

Das Einlesen eines speziell formatierten C-Quelltextes wäre hilfreich. Hierfür müsste dieser geparsed werden, der Zugriff wäre mit dem eines Editors identisch.

Fehlende Blöcke

Die zweite Einschränkung, die direkt Fähigkeiten der Struktogramme betrifft, ist diejenige, dass zwei Blocktypen nicht zur Verfügung stehen. Bei dem einen handelt es sich um die Darstellung von parallelem Verhalten, bei dem anderen um die Aussprunganweisung.

In der GUI sind beide Blöcke noch nicht vorhanden. Die grafische Darstellung der Parallelität ähnelt jedoch der Mehrfachauswahl und die Aussprunganweisung dem Anweisungsblock. Beide verfügen zur Unterscheidung von diesen Blöcken über charakteristische Linien. Diese müssten, wie bei der Einfach- und Mehrfachauswahl, mit *Graphics.getLine()* gezeichnet werden.

Um den Aussprung-Knoten zu realisieren, ist in der C-Codegenerierung nichts weiter nötig, als das entsprechende Schlüsselwort einzufügen und anschließend weiter durch den Diagrammbaum zu traversieren. Hierbei sollte die aktuelle Rekursion abgebrochen werden, da die folgenden Codezeilen nicht mehr erreicht werden können. Für die VHDL-Codegenerierung ist auch ein Abbruch der aktuellen Rekursion nötig. Das Sprungziel, also der von dem Aussprung-Knoten aus erreichte Zustand, befindet sich als oberstes Element auf dem Stack und darf nur gelesen werden.

Für die Realisierung des Parallelitäts-Blocks würde sich für die C-Codeerzeugung die Verwendung von Threads der *Pthread* Bibliothek anbieten. Für jeden parallelen Strang könnten generisch benannte Methoden erzeugt werden, um diese dann nacheinander als eigene Threads zu starten. Der Programmfluss könnte danach durch Aufruf von *join()* wieder zusammengefasst werden. So würde das Verhalten des Blocks sauber umgesetzt.

Während die Erstellung von Parallelität im C-Code also noch vorstellbar ist, wird dies im Mehrzyklus-Datenpfad sehr schwierig. Hier müssten entweder mehrere Zustände gleichzeitig aktiv sein oder in einem Zustand deutlich mehr Eingangssignale verarbeitet und auch mehr Ausgangssignale erzeugt werden. Die Erstellung und die Komplexität der Zustandsmaschine würde dadurch stark zunehmen. Auch wenn dies gelänge, gäbe es ein nachfolgendes Problem: Die Arithmetik-Ressourcen und Register könnten gleichzeitig im Ablauf angesprochen werden. Es müsste dann eine Logik zum Zugriffsschutz integriert werden oder es müssten die entsprechenden Elemente mehrfach vorhanden sein. Dies würde dem Gedanken eines Mehrzyklus-Datenpfades widersprechen. Eine pragmatische Lösung wäre es, die Parallelität nicht umzusetzen und sequenziell die eigentlich parallelen Abläufe abzuarbeiten. Für die Reihenfolge in dem parallelen Block bestehen keine Zusicherungen, sodass dort keine Ausgaben erfolgen sollten.

Benutzermanagement

Zur Zeit ist das Benutzermanagement nur rudimentär implementiert. Es wird gespeichert, welche Editoren gerade auf das Diagramm zugreifen und es ist diesen Editoren möglich, Knoten zur Bearbeitung zu sperren. Allerdings sind die angemeldeten Editoren alle mit den gleichen Rechten ausgestattet und es kann sich auch jeder, der Kenntnis über die IP-Adresse der RMI-Registry besitzt, mit dem Diagramm verbinden.

Für größere Projekte und öffentlich zugreifbare Rechner müssten verschieden privilegierte Benutzerrollen eingeführt werden und ein Anmeldevorgang implementiert werden. Es müsste dann bei den angemeldeten Editoren zwischen denen mit administrativen Rechten und jene mit normalen Privilegien unterschieden werden. Die Administratoren dürften dann die Benutzer freischalten, löschen und deren initiale Passwörter festlegen. Diese Passwörter und Benutzernamen könnten dann bei der Anmeldung übertragen, überprüft und anschließend bei den Benutzeraktionen überprüft werden.

Als Ort für das erweiterte Benutzermanagement würde sich der *DiagramHandler* anbieten. Dieser verfügt schon zu Teilen über die Kenntnis der Editoren, sodass hier die zusätzlichen Attribute untergebracht werden könnten. Bei diesen Attributen handelt es sich um die Rolle, einen Benutzernamen und das hinterlegte Passwort. Diese Informationen könnten dann ausgewertet werden. Um die Sicherheit des Systems zu garantieren, müsste zumindest die Anmeldung und die administrativen Tätigkeiten über eine SSL Verbindungen stattfinden. Dies könnte über SSL-Sockets realisiert und über die Manipulation der bei RMI verwendeten *SocketFactory* integriert werden.

11. Fazit

Die These, dass es möglich ist, aus Nassi-Shneiderman-Diagrammen synthetisierbaren VHDL-Code zu erzeugen, wird durch diese Arbeit bestätigt. Es ist hiermit nachgewiesen, dass sich alle Grundkonstrukte umsetzen lassen und diese beliebig kombiniert werden können. Auch die direkte Übernahme der Testmethodiken aus dem Software-Engineering erweist sich als erfolgreich. Durch den speziellen hierarchischen Aufbau der Nassi-Shneiderman-Diagramme ist eine konkrete Zuordnung zu den Teststufen des V-Modells möglich.

Besonders interessant ist dabei zu sehen, wie das über die Diagramme beschriebene Verhalten in der Simulation der untersuchten Beispiele wiederzufinden ist. Durch die Verwendung der gleichen Bezeichner und einer übersichtlichen Struktur ist es möglich, die Umsetzung des eigenen Algorithmus ohne großen Aufwand zu überprüfen. Damit wird auch dem zweiten Ziel der Forschung entsprochen, einen einfachen Entwurf komplexer Schaltungen zu ermöglichen.

Interessant ist die Fragestellung, wie die vorgeschlagenen Erweiterungen in der Zukunft umgesetzt werden. Durch eine konsequente Weiterentwicklung kann ein gleichzeitig mächtiges und benutzerfreundliches Tool geschaffen werden. Durch das unverbrauchte Forschungsgebiet gibt es gleichzeitig zahlreiche Möglichkeiten, sich in dessen Erschließung einzubringen.

Die Arbeit zeigt, dass auch die Kombination erprobter Techniken, im konkreten Falle Nassi-Shneiderman-Diagramme und die Hardwarestruktur Mehrzyklus-Datenpfad, einen Erkenntnisgewinn erzielt.

Auch in der heutigen Zeit ist es von Vorteil, auf erprobte und weit verbreitete Methoden zu setzten. Nur so ist es möglich, die Technik der Zukunft auf eine solide Basis zu stellen.

12. Literatur

1: Edsger Wybe Dijkstra, C. A. R. Hoare, Ole-Johan Dahl. (Februar 1972) Structured Programming. 1st Edition. Academic Press

2: Kathleen Jensen, Niklaus Wirth. (Oktober 2007) PASCAL User Manual and Report. 18. Ausgabe. Springer

3: I. Nassi, B. Shneiderman. (August 1973) Flowchart Techniques for Structured Programming. -. SIGPLAN Notices

4: DIN 66261, NABD, November 1985

5: The MathWorks, HDL Coder, o.J., http://www.mathworks.de/products/hdl-coder/, Stand: 01.11.2013

6: The MathWorks, Simulink, o.J., http://www.mathworks.de/products/simulink/, Stand: 01.11.2013

7: The MathWorks, Stateflow, o.J., http://www.mathworks.de/products/stateflow/, Stand: 01.11.2013

8: Hannes Muhr. (November 2000) Einsatz von SystemC im Hardware/Software-Codesign. -. Technischen Universität Wien

9: Mentor Graphics, Handel-C Synthesis Methodology, o.J., http://www.mentor.com/products/fpga/handel-c/, Stand: 01.11.2013

10: Altera, OpenCL for Altera FPGAs, o.J., http://www.altera.com/products/software/opencl/opencl-index.html, Stand: 01.11.2013

11: EasyCODE, EasyCODE 9 SPX, o.J., http://www.easycode.de/produkte/easycode-spx.html, Stand: 01.11.2013

12: Structorizer, Structorizer, o.J., http://structorizer.fisch.lu/, Stand: 01.11.2013

13: Christoph Kecher. (März 2011) UML 2: Das umfassende Handbuch. 4. Ausgabe. Galileo Computing

14: John Vlissides, Richard Helm, Ralph Johnson, Erich Gamma. (Oktober 1994) Design Patterns. Elements of Reusable Object-Oriented Software. 1st ed.. Addison-Wesley Longman

15: Niklaus Wirth. (Mai 2011) Grundlagen und Techniken des Compilerbaus. 3., bearbeitete Auflage. Oldenbourg Wissenschaftsverlag

16: Christian Ullenboom. (Oktober 2011) Java ist auch eine Insel: Das umfassende Handbuch. 10. Ausgabe. Galileo Computing

17: William Grosso. (Oktober 2001) Java RMI. First Edition. O'Reilly Media

18: Andrew S. Tanenbaum, Maarten van Steen. (November 2007) Verteilte Systeme: Prinzipien und Paradigmen. 2. Ausgabe. Addison-Wesley Verlag

19: Common Object Request Broker Architecture (CORBA) Specification, OMG, August 2011

20: Web Services Architecture, David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, David Orchard, Februar 2001

21: Sharon Biocca Zakhour, Sowmya Kannan, Raymond Gallardo. (März 2013) The Java Tutorial: A Short Course on the Basics. 5th Edition. Addison-Wesley Professional

22: Oliver Haase. (Juni 2008) Kommunikation in verteilten Anwendungen: Einführung in Sockets, Java RMI, CORBA und Jini. Überarbeitete und erweiterte Auflage. Oldenbourg Wissenschaftsverlag

23: George Coulouris, Jean Dollimore, Tim Kindberg. (März 2002) Verteilte Systeme Konzepte und Design. 3. Ausgabe. Pearson Studium

24: Robert F. Stärk, Joachim Schmid, Egon Börger. (Juni 2001) Java and the Java Virtual Machine: Definition, Verification, Validation. 1. Ausgabe. Springer

25: Kathy Sierra, Bert Bates, Lars Schulten. (Mai 2006) Java von Kopf bis Fuß. 1. Ausgabe. O'Reilly

26: Silvia Hagen. (Mai 2006) IPv6 Essentials. 2nd ed.. O'Reilly Media

27: Jürgen Reichardt, Bernd Schwarz. (Dezember 2012) VHDL-Synthese: Entwurf digitaler Schaltungen und Systeme. 6. Ausgabe. Oldenbourg Wissenschaftsverlag

28: Zvonko Vranesic, Stephen Brown. (April 2008) Fundamentals of Digital Logic with VHDL Design. 3rd Edition.

29: Bernhard Rumpe. (Oktober 2004) Agile Modellierung mit UML. Codegenerierung, Testfälle, Refactoring. Auflage 2005. Springer

30: Andreas Spillner, Tilo Linz. (September 2012) Basiswissen Softwaretest: Aus- und Weiterbildung zum Certified Tester - Foundation Level nach ISTQB-Standard. 3. Ausgabe. Dpunkt Verlag

31: Thomas Grechenig, Mario Bernhart, Roland Breiteneder, Karin Kappel. (Oktober 2009) Softwaretechnik: Mit Fallbeispielen aus realen Entwicklungsprojekten. 1. Ausgabe. Pearson Studium

32: Alan V. Oppenheim, Ronald W. Schafer. (Dezember 1998) Zeitdiskrete Signalverarbeitung. 2. Ausgabe. Oldenbourg Wissenschaftsverlag

13. Glossar

API: Programmierschnittstelle
ASM: Algoithmic State Machine
AWT: Abstract Window Toolkit
Big-Endian & Little-Endian: Interne Zahlendarstellungsformate in Prozessoren
CORBA: Common Object Request Broker Architecture, eine Middleware
Diagrammblock: Grafischer Teil des Diagramms mit Funktion
FSM: Finite State-machine, endlicher Zustandsautomat
GOF: Gang of Four(Erich Gamma,Richard Helm,Ralph Johnson,John Vlissides)
GPL: Gnu Public License
IPv4: Aktueller Netzwerkstandard, 32 Bit für die Adressierung
IPv6: Neuer Netzwerkstandard, 128 Bit für die Adressierung
JVM: Java Virtual Maschine, führt den Bytecode der Javaklassen aus
Loopback: Virtuelle Netzwerkschnittstelle zur internen Kommunikation
Middleware: Vermittlungssoftware, kapselt Objekttransporte über Netzwerk
NAT: Network Address Translation
NSD: Nassi-Shneiderman-Diagramm
OMG: Object Managment Group, standardisiert CORBA
OpenCL: Open Computing Language
RemoteStub: Zugriffsmöglichkeit auf ein entferntes Objekt
RMI: Remote Method invocation, Javas native Middleware
RMI-Registry: Namensregister für RMI-Objekte
RPC: Remote Procedure Call
Singleton Entwurfsmuster: Stellt sicher, dass ein Objekt nur einmal existiert
Socket: Kapselung des Netzwerkzugriffs auf niedriger Ebene
SWING: Grafikbibliothek für Java
TCP: Transmission Control Protocol
Verteiltes System: System auf mehren Maschinen, das eine Aufgabe löst
VHDL: Very High Speed Integrated Circuit Hardware Description
Webservices: Middleware des W3C
Zugriffstransparenz: Ortsunabhängiges Verhalten bei Methodenaufrufen

14. Abbildungsverzeichnis

15. Tabellenverzeichnis

16. Anhang

16.1 Benutzte Hilfsmittel

OpenJDK 7
Die neuste Version des Java Development Kits in der Open Source Variante. Es werden keine Neuerungen verwendet, es ist dennoch nützlich gleich auf Basis der neusten Version zu entwickeln.

Link: http://openjdk.java.net/

Subversion
Für die Entwicklung wird eine Versionskontrolle benutzt. Aufgrund positiver Erfahrungen und eines bereits eingerichteten Servers wird SVN (Version 1.6.17) verwendet. Die Netzwerkkommunikation findet dabei über einen Apache Webserver statt.

Link: http://subversion.tigris.org/

Eclipse
Eclipse ist das Standardentwicklungswerkzeug für Java. Es ist für viele Betriebssysteme verfügbar und sehr gut durch Plugins erweiterbar. (Version 3.7.2)

Link: http://www.eclipse.org/

GNU Octave
Mathematisches Konsolen-Werkzeug, welches die MATLAB Syntax verwendet. (Version 3.2)

Link: http://www.gnu.org/software/octave/

Subclipse
Bei Subclipse handelt es ich um ein SVN-Plugin für Eclipse. Es vereinfacht das Refactoring gegenüber dem SVN-Client auf der Konsole oder Tortoise enorm. (Version 1.07)

Link: http://subclipse.tigris.org/

DIA
Ein freier Diagrammeditor. Er wurde für die Erzeugung aller Diagramme verwendet. (Version 0.97.2)

Link: http://dia-installer.de/index.html.de

LibreOffice
Textverarbeitung der ehemaligen OpenOffice Mitarbeiter. Unterstützt größere Arbeiten durch Verzeichnisse. (Version 3.5)

Link: http://www.libreoffice.org/

Ubuntu 12.04
Das System für die Entwicklung und für Tests der Lauffähigkeit unter Linux. Durch seine hohe Verbreitung ist es ein guter Repräsentant.

Link: http://www.ubuntu.com/desktop

GIMP
Ein Zeichenprogramm mit der Fähigkeit verschiedene Ebenen zu verwenden und Bildschirmfotos zu erstellen. (Version 2.6.12)

Link: http://www.gimp.org/

ModelSim
Die Verhaltenssimulationen wurden mit der Akademischen Version von Modelsim durchgeführt. (Version PE Student Edition 10.2.c)

Link: http://www.mentor.com/products/fpga/model

16.2 Inhalt der CD

Die Software

Neben dem Quelltext-Paket „bachelor.zip“ liegt auch die startfähige Datei „*NetStructEDIT.jar*“ vor.

Die Software lässt sich bei installierter JAVA VM unter Windows und Linux durch einen Doppelklick auf die **.jar* Datei starten. Falls dies scheitert, kann die Software mit dem folgenden Befehl ausgeführt werden:

„java -jar NetStructEDIT.jar“

Um die Testfälle und die zugehörigen Trigger zu generieren, muss das Programm mit folgenden Parametern im Testmodus gestartet werden: *„java -jar NetStructEDIT.jar test <Ordner>“*

Dokumentation der Testfälle

Die Komponenten und Integrationstests, sowie die Beispiele, befinden sich im Ordner *Verifikation*.

Diagramme

Um die Diagramme besser lesen zu können, liegen diese im Ordner *Diagramme* vor.

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, den _____ _____