Studienarbeit

Einsatz des Generators DEViL zur Animation von Logikbausteinen

Manuel Wickert <wickert@upb.de>

30. Juni 2005

Betreuer: Dipl.-Inform. Carsten Schmidt

Gutachter: Prof. Dr. Uwe Kastens

Inhaltsverzeichnis

Erklärung und Dank

Hiermit versichere ich, dass ich diese Studienarbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel verwendet, sowie Zitate kenntlich gemacht habe.

An dieser Stelle möchte ich Carsten Schmidt für die hervorragende Betreuung meiner Studienarbeit danken.

Paderborn, den 30. Juni 2005

1 Einleitung

Ziel dieser Arbeit war die Entwicklung eines Editors, mit dessen Hilfe digitale Schaltungen erstellt werden können und der die Animation und Simulation dieser aus Logikbausteinen bestehenden Schaltungen ermöglicht. Die Erstellung dieses Editors basierte auf dem Werkzeugsystem DEViL. DEViL ist ein System, das Entwicklungsumgebungen aus einer Spezifikation einer visuellen Sprache generiert.

Bisher war die Animation von visuellen Sprachen in DEViL nicht vorgesehen. Somit war es ebenfalls Teil dieser Arbeit, DEViL auf die Möglichkeit der Generierung von Animationsumgebungen zu untersuchen. Bei dieser Untersuchung ging es darum, Aussagen darüber zu treffen, inwieweit DEViL die Generierung von Animationsumgebungen erlaubt und wo es noch Änderungen an diesem Werkzeugsystem geben muss.

Die Entwicklung des Programms erfolgte in mehreren Schritten. Zuerst wurde ein Editor entworfen, der die Zeichnung einer digitalen Schaltung ermöglicht. Im Anschluss daran wurde die Simulation und die Animation auf Basis des Editors entwickelt. Nach Fertigstellung des Programms konnten dann Schlüsse auf die Fähigkeit DEViLs bzgl. der Animationen von visuellen Sprachen gezogen werden.

Die Arbeit umfasst vier Kapitel. Nach dieser Einleitung folgt das Grundlagenkapitel, in dem eine Einführung in die verschiedenen Themenbereiche der Arbeit gegeben wird. Im Kapitel 3 wird die Umsetzung der Konzepte und die Realisierung des beschriebenen Programms vorgestellt. Kapitel 4 schließlich fasst die Ergebnisse der Arbeit zusammen.

2 Grundlagen

In diesem Kapitel soll eine Einführung in die verschiedenen Themenbereiche dieser Arbeit gegeben werden. Zu Anfang sollen die Begriffe Simulation und Animation beschrieben werden. Anschließend werden die Grundlagen der beiden Themenkomplexe der Arbeit erläutert. Diese sind die Logikbausteine und deren Simulation auf der einen Seite und visuelle Sprachen und DEViL auf der anderen.

2.1 Animation und Simulation

Diese Arbeit beschäftigt sich mit der Animation und Simulation von Logikbausteinen. Diese beiden Begriffe werden im Folgenden näher erklärt. Dazu werden die Gemeinsamkeiten der Themen herausgearbeitet und es wird eine Abgrenzung der Begriffe vorgenommen.

Animation wird meistens mit einer Bewegung in Zusammenhang gebracht. Um eine Bewegung auf dem Bildschirm eines Computers zu erzeugen, wird nacheinander eine Reihe von Bildern angezeigt. Wenn nur ein kurzes Zeitintervall zwischen dem Anzeigen der Bilder liegt, wird eine flüssige Bewegung wahrgenommen. In [4] wird die Animation ähnlich definiert:

> „Unter Computer-Animation versteht man die Technik der Erzeugung von bewegten oder belebten Bildern durch den Computer.“ [4]

Animation muss allerdings nicht notwendigerweise einen flüssigen Bewegungsablauf vorweisen. Sie kann als die Darstellung zeitabhägiger Daten auf dem Bildschirm charakterisiert werden. Die Darstellungsform kann dabei variieren. Eine Änderung der Daten über die Zeit sollte allerdings erkennbar sein. Die reine Darstellung statischer Daten kann man nicht als Animation bezeichnen.

Ein Anwendungsgebiet für die Animation ist die Darstellung von Simulationsdaten. In [4] wird dies sogar als ein Einsatzschwerpunkt der Animation gesehen. Dort heißt es:

> „Ein Einsatzschwerpunkt der Animation ist die Visualisierung von Simulationsvorgängen ,... “ [4]

In dieser Arbeit wird die Animation ausschließlich für die Visualisierung dieser Vorgänge verwendet. Die Animation stellt dabei die sich im Zeitverlauf ändernden Daten der Simulation dar. In [4] wird in diesem Zusammenhang die Animation wie folgt definiert.

> „Technik zur Wiedergabe der zeitabhängigen Veränderungen bei der Simulation von Prozessen.“ [4]

Diese Definition passt sehr gut in den Kontext dieser Arbeit. Im weiteren Verlauf dieser Arbeit wird diese Definition verwendet.

Unter Simulation wird die Nachahmung bestimmter Prozesse in der Realität verstanden. Die Realität wird dabei durch ein Modell vereinfacht. Die Nachahmung erfolgt unter den Annahmen, die im Modell getroffen wurden. Meist wird bei der Simulation das Zeitverhalten von Prozessen betrachtet. Ähnlich wird auch in [4] die Simulation beschrieben.

> „Simulation eines Systems ist die Arbeit mit einem Modell, das das wirkliche System unter anderem auch in Bezug auf sein Zeitverhalten abbildet.“ [4]

Sowohl der Simulation als auch der Animation wird ein Zeitverhalten zugrunde gelegt. Diese Eigenschaft haben beide gemeinsam. Die Simulation ahmt bestimmte Prozesse über den Zeitverlauf nach, die Animation stellt bestimmte Daten im Zeitverlauf dar. Die Ziele unterscheiden sich jedoch deutlich. Während die Simulation die Gewinnung von Informationen über dynamische Prozesse als Ziel hat, ist das Ziel der Animation die Veranschaulichung von dynamischen Abläufen.

2.2 Logikbausteine

Bei der Betrachtung von Logikbausteinen oder Logikgattern muss man zwei Sichtweisen unterscheiden. Die erste ist eine rein technische, die sich mit Logikbausteinen beschäftigt, die in der Realität zum Bau von digitalen Schaltungen verwendet werden. Mit dieser Sicht beschäftigt sich der Abschnitt Logikgatter.

Im zweiten Abschnitt wird die Simulation von digitalen Schaltungen betrachtet. Hierzu werden Modelle für Logikbausteine verwendet, die von bestimmten physikalischen Eigenschaften abstrahieren.

2.2.1 Logikgatter

> „Logische Gatter sind die elementaren Grundbausteine digitaler Schaltungen und Systeme.“ [12]

Logikgatter bilden eine von mehreren Abstraktionsebenen beim Entwurf von digitalen Schaltungen. Dabei bestehen Logikgatter technisch gesehen aus Widerständen, Dioden, Transistoren und anderen elektrischen Bauteilen. Wie diese elektronischen Bauteile zu dem logischen Gatter zusammengebaut werden, ist sehr unterschiedlich. Es existieren viele verschiedene Schaltungsfamilien wie TTL, DTL, CMOS, usw., die vorschreiben wie z.B. ein UND-Gatter in dieser Familie intern aufgebaut ist. Die

logische Funktion eines Gatters ist dabei vollkommen unabhängig von der Schaltungsfamilie. D.h. ein TTL UND-Gatter verhält sich logisch gesehen genau wie ein DTL UND-Gatter. Die Abstraktion erleichtert den Entwurf großer Schaltungen, da bestimmte Aspekte wie z.B. Spannungspegel nicht mehr betrachtet werden müssen.

Unter digitalen Schaltungen wird in dieser Arbeit eine Menge von Logikgattern verstanden, die über Leitungen miteinander und mit den Ein- und Ausgängen der Schaltung verbunden sind. Ein Logikgatter besteht dabei aus einer Menge von Eingängen und einem Ausgang. Signale, die an den Ein- und Ausgängen solcher Logikgatter anliegen, werden mit „L“ für Low und „H“ für High, beschrieben. Die Ein- und Ausgänge der Logikgatter können nur diese beiden Signale annehmen, die meist als binäre Zahlen 0 für „L“ und 1 für „H“ dargestellt werden. Auch die Bezeichnung mit false und true ist üblich. Die logische Funktion von Gattern kann mit der booleschen Algebra beschrieben werden.

In [12] werden als elementare Grundgatter das UND-, das ODER-, das NICHT-, das NAND- und das NOR- Gatter betrachtet, wobei das NAND- und das NOR-Gatter als eine Kombination eines UND- bzw. ODER- mit einem NICHT-Gatter gesehen werden kann. Aus diesen 5 Gattern lassen sich alle booleschen Funktionen aufbauen. Im Folgenden sollen die Funktionen des UND, ODER und NICHT-Gatters, das auch als Negator bezeichnet wird, aufgeführt werden:

> „Bei einem UND-Gatter tritt nur dann ein 1-Signal am Ausgang auf, wenn alle Eingangssignale 1 annehmen. Für alle übrigen Eingangssignalkombinationen erscheint 0 am Ausgang. “ [12]

> „Der Ausgang eines ODER-Gatters nimmt 1-Signal an, wenn an mindestens einem seiner Eingänge 1-Signal liegt. Nur wenn alle Eingänge 0-Signal führen, erscheint 0-Signal am Ausgang.“ [12]

> „Der Negator (Inverter) hat nur einen Eingang und einen Ausgang. Am Ausgang tritt stets das komplementäre Signal zum Eingang auf: ..“ [12]

Für die grafische Darstellung von Logikbausteinen gibt es zwei Alternativen. Die erste Alternative war lange Zeit im amerikanischen Raum verbreitet und wurde auch in Deutschland von einigen Wissenschaftlern benutzt. Die zweite Alternative ist im deutschen Raum verbreiteter und ist seit längerem in einer DIN und seit einiger Zeit sogar in einer europäischen Norm festgeschrieben (DIN EN 60617-12). Die grafische Notation nach DIN EN 60617-12 (siehe Abb. 2.1) wird im weiteren Verlauf dieser Arbeit verwendet.

2.2.2 Simulation von digitalen Schaltungen

Die Simulation von digitalen Schaltungen ist heutzutage essentiell für die Entwicklung solcher Schaltungen. In der Industrie wird seit einiger Zeit jede neue Schaltung auf einem Computer simuliert, bevor diese gebaut wird.

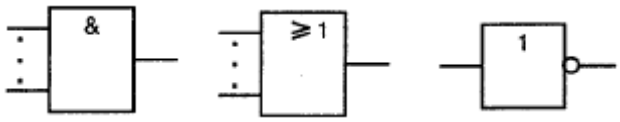

Abbildung 2.1: Darstellungen des UND-, ODER und NICHT-Gatters nach DIN EN 60617-12 Quelle: [1]

Bei der Simulation von digitalen Schaltungen gibt es verschiedene Ebenen der Abstraktion. Eine dieser Ebenen ist die Registertransfer-Ebene, eine andere die Simulation auf Ebene der Logikbausteine. In dieser Arbeit wird nur die Simulation auf Ebene der Logikbausteine betrachtet. Die Wahl dieser Ebene hat verschiedene Auswirkungen auf die Genauigkeit und Geschwindigkeit, mit der die Simulation durchgeführt werden kann.

Bei der Simulation von Logikbausteinen wird das Laufzeitverhalten einer Schaltung betrachtet. Es werden also nicht nur einmalig Werte berechnet, sondern es wird eine Schaltung während der Laufzeit beobachtet. Diese Betrachtungsweise kann helfen Fehler aufzudecken, die im Endergebnis nicht sichtbar sind, allerdings Störungen während der Laufzeit verursachen könnten.

Im Themenkomplex der Simulation von Logikbausteinen ist es schwer, Grundlagenliteratur zu finden. Zwar gibt es einige Arbeiten, die sich mit bestimmten Teilbereichen der Simulation sehr genau befasst haben, eine ausführliche Arbeit über den Themenkomplex im Allgemeinen ist aber schwer zu finden. Ein kleiner Überblick soll an dieser Stelle gegeben werden.

Bei der Simulation von Logikbausteinen müssen das Modell, das einer Simulation zugrunde liegt, und der Simulationslauf an sich auseinandergehalten werden.

2.2.2.1 Modell

> „Modelle geben immer nur Ausschnitte der Wirklichkeit wieder.“ [3]

Das Modell, das einer Simulation von Logikbausteinen zugrunde liegt, sollte vor allem zwei Aspekte betrachten. Erstens die Verzögerung von Logikbausteinen und zweitens die Signalzustände, die die Ein- oder Ausgänge der Logikbausteine annehmen können.

Verzögerung Durch die Betrachtung von Verzögerungen von Logikbausteinen kann das Verhalten dieser Bausteine berücksichtigt werden, das ohne die Verzögerung nicht unbedingt ersichtlich ist. So kann z.B. eine schwingende Schaltung ohne die Betrachtung von Verzögerungen nicht analysiert werden. Auch Hazards können nicht ohne die Betrachtung von Verzögerungen von Gattern bei der Simulation aufgedeckt werden.

> „Hazards sind kurze Impulse bzw. Impulsfolgen, die gewollt oder ungewollt in Abhängigkeit von Gattersignalverzögerungszeiten der Schaltung

generiert werden oder entstehen und die Funktion der Schaltung beeinflussen (verfälschen). “ [7]

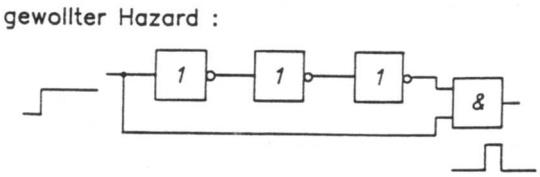

Abbildung 2.2: gewollter Hazard aus [7]

Auf der linken Seite des Bildes ist die Veränderung des Eingangssignales über die Zeit zu sehen, unter dem Gatter die Veränderung des Ausgangssignales. Es tritt am Ausgang ein kleiner Impuls auf, der allein durch die Gatterverzögerungen der NICHT-Gatter entsteht. Dieser Impuls wird Hazard genannt. In diesem Beispiel ist der Hazard gewünscht. Es ist allerdings durchaus möglich, dass ein solcher Impuls unerwünscht auftritt.

Die Betrachtung von Gatterverzögerungen scheint also durchaus sinnvoll. Dabei gibt es viele Möglichkeiten, die Verzögerung von Gattern zu modellieren. So ist es z.B. möglich, dass ein UND-Gatter eine andere Verzögerung als ein ODER-Gatter hat.

In [7] werden verschiedene Möglichkeiten betrachtet, wie im Allgemeinen eine Gatterverzögerung modelliert werden kann. Die Modelle unterscheiden dabei nicht zwischen den verschiedenen Gattertypen (UND, ODER, NICHT), sondern sind für alle Gattertypen anwendbar. Im ersten sog. Unit-Delay-Modell wird eine Verzögerung am Ausgang des Gatters vorgenommen. Es gibt in diesem Modell also nur eine Verzögerungszeit für ein Gatter.

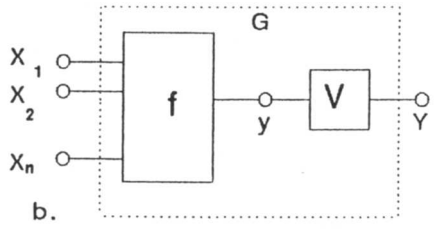

Abbildung 2.3: Unit Delay Modellierung Quelle: [7]

Bei der eingangsbezogenen Delay-Modellierung wird jedem Eingang des Gatters eine Verzögerung zugeordnet. Die Verzögerungen der Eingänge müssen dabei nicht zwangsläufig gleich sein. Es können daher einige Eingänge eine andere Verzögerung haben als andere. Dies hat den Vorteil gegenüber dem Unit-Delay-Modell, dass Gatter mit großer Funktionalität besser beschrieben werden können. Wenn ein Gatter z.B. einen Clock-Eingang hat, so ist es möglich, dass eine Signaländerung an diesem

Eingang sich durchaus schneller auf den Ausgang auswirkt als eine Signaländerung an anderen Eingängen. Erweitern kann man diese Modellierung, indem man verschiedene Verzögerungszeiten für den 0-1 bzw. den 1-0 Übergang verwendet. Eine solche Modellierung bewegt sich näher an der Realität, da die Schaltzeiten eines Gatters vom Übergang abhängig sind.

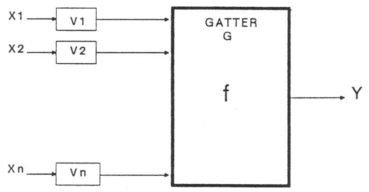

Abbildung 2.4: Eingangsbezogene Delay Modellierung aus [7]

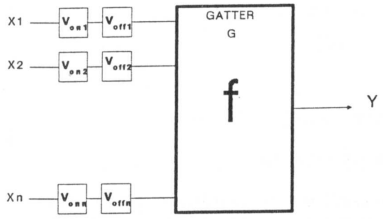

Abbildung 2.5: Eingangsbezogene Delay Modellierung mit verschiedenen Verzögerungszeiten für die 0-1 bzw. 1-0 Übergänge aus [7]

Eine weitere Möglichkeit ergibt sich daraus, sowohl mit einer eingangs- wie auch einer ausgangsseitigen Verzögerung zu modellieren. Mit einer solchen ein- und ausgangseitigen Delay-Modellierung läßt sich sowohl eine allgemeine Verzögerung einstellen (die am Ausgang), als auch eine Betrachtung der einzelnen Eingangsverzögerungen erreichen.

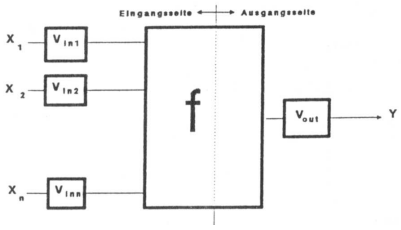

Abbildung 2.6: Ein- und Ausgangsbezogene Delay Modellierung aus [7]

In dieser Arbeit wird das Unit-Delay-Modell verwendet. Warum dieses Modell gewählt wurde, wird in 3.3.1.1 genauer beschrieben.

Signalzustände Ein- und Ausgänge von Logik-Gattern können prinzipiell zwei Werte annehmen: 1 oder 0. Aus dieser Tatsache könnte man schließen, dass auch für die Simulation von Logikgattern diese beiden Zustände ausreichen.

In [7] wird ein einfaches Modell der Signalzustände vorgestellt, bei dem Logikbausteine auch nur diese Zustände annehmen können. Ein Problem ergibt sich allerdings in diesem Modell. Vor dem ersten Simulationslauf muss eine Schaltung initialisiert werden. D.h. es müssen Werte angenommen werden, die ein Logikgatter am Ausgang hat, bevor die ersten Signale an den Eingängen anliegen. In der Realität hängt es von der Schaltungsfamilie der Logikbausteine ab, welche Werte bzw. Spannungen an den Ausgängen dieser anliegen, bevor die ersten Signale an die Logikbausteine angelegt werden. Bei der Simulation von Logikbausteinen sollte allerdings von Schaltungsfamilien abstrahiert werden, damit eine simulierte Schaltung unabhängig von der Schaltungsfamilie, in der diese später gebaut wird, einwandfrei funktioniert. Es ist also nicht klar, ob eine 1 oder eine 0 am Ausgang eines Gatters anliegt, bevor dieses zum ersten Mal geschaltet hat.

Diese Tatsache ist nicht problematisch, wenn alle Ausgänge der Gatter in einer Schaltung von den Eingängen der Schaltung abhängen. Dies ist allerdings nur dann der Fall, wenn innerhalb der Schaltung keine Werte gespeichert werden. Wird eine Schaltung z.B. unter Verwendung von Flip Flops konstruiert, so entstehen Situationen, in denen der Eingang eines Gatters direkt oder indirekt von seinem Ausgang abhängt. In einer solchen Situation kann also eine Schaltungsberechnung nur dann durchgeführt werden, wenn der Ausgang eines Gatters schon vor dem ersten Schalten dieses Gatters einen Wert hat. Aus den eben genannten Gründen muss demnach mindestens ein weiterer Zustand eingeführt werden, der benutzt wird, um die Ausgänge der Gatter zu initialisieren. In [7] wird ein Zustand „X“ bzw. „U“ genannt der einen undefinierten Zustand repräsentiert. Dieser könnte für einen solchen Zweck benutzt werden.

Ein weiterer Zustand wäre nach [7] nötig, wenn man mit sog. „Tristate-Buffern“ ein Bus-System modellieren möchte. Dieser sog. hochohmige Zustand wird benötigt, wenn mehrere Schaltungen gleichzeitig auf einen Bus schreiben. Da in dieser Arbeit allerdings nicht mit Tristate-Buffern gearbeitet wird, soll dies nur der Vollständigkeit halber erwähnt sein.

Professionelle Simulationsysteme arbeiten heutzutage mit weit mehr als diesen Zuständen. Die Sprache VHDL, die auch für die Simulation von digitalen Schaltungen verwendet wird, unterscheidet beispielsweise neun verschiedene Signalzustände. Weitere Informationen zu VHDL sind in [9] zu finden.

2.2.2.2 Simulationslauf

Im Simulationslauf werden aus den Eingabedaten (die Eingänge der Schaltung) die Ausgänge der Schaltung berechnet.

Vor dem Simulationslauf muss eine Schaltung ggf. initialisiert werden und die Eingänge der Schaltung müssen auf die Eingabedaten gesetzt werden. Bestimmte Simulationseinstellungen können ggf. zu dieser Zeit geladen werden.

Um das Zeitverhalten einer Schaltung zu simulieren, wird meist eine Simulationszeit eingeführt. Dabei wird die kürzeste zeitliche Einheit definiert, in der simuliert werden kann. Kürzer als diese Einheit kann kein Signal sein. Alle anderen Zeiten, z.B. die Verzögerungszeit eines Gatters, sind Vielfache dieser Zeit.

Für die eigentliche Simulation werden in [7] zwei Methoden genannt, erstens die compilerorientierte und zweitens die interpretierende. In [7] wird nicht näher auf die Verfahren eingegangen. Krodel [8] schreibt, dass beim compilerorientierten Verfahren ein ausführbares Programm erzeugt wird, das die Schaltung repräsentiert und welches durch das Ausführen die Simulation durchführt. Dieses Verfahren wird sowohl in [7] als auch in [8] als sehr schnell bezeichnet. Krodel bemerkt allerdings, dass dieses Verfahren ungeeignet für die Modellierung von Gatterverzögerungen ist. Mit interpretierend wird ein Verfahren bezeichnet, das die Schaltung in einer internen Datenstruktur hält und darauf einen Algorithmus anwendet, der die Zustände der Gatter berechnet.

Eine Verfeinerung beider Verfahren stellt das ereignisorientierte Simulieren dar. Bei dem ereignisorientierten Simulieren werden jeweils nur die Gatter simuliert, bei denen sich die Werte geändert haben. Dieses Verfahren ist eine Art und Weise, die Simulation zu beschleunigen.

2.3 Visuelle Sprachen

Visuelle Sprachen gewinnen in letzter Zeit in der Informatik immer mehr an Bedeutung. Einige dieser Sprachen wie z.B. Labview oder UML haben sich längst auf dem Markt etabliert und sind allgemein anerkannt. Dieser Abschnitt soll eine Einführung in visuelle Sprachen geben und den Generator DEViL erklären. Außerdem soll an dieser Stelle der Zusammenhang zwischen visuellen Sprachen und dieser Arbeit herausgearbeitet werden.

2.3.1 Was ist eine visuelle Sprache?

Leider haben sich auf dem Gebiet der visuellen Sprachen und dem damit verwandten Gebiet der visuellen Programmiersprachen keine Definitionen durchgesetzt, die allgemein anerkannt sind. Verschiedene Autoren haben dargelegt, was sie unter den Begriffen verstehen. Die Begriffe sind leider nicht immer klar formuliert oder sehr weit gefasst.

Ich verstehe unter einer visuellen Sprache eine Sprache, die visuell oder grafisch dargestellt wird. Eine grafische oder visuelle Darstellung ist eine Präsentation der Daten, die nicht ausschließlich auf Text, sondern auf grafischen Symbolen oder Objekten basiert. Ein Beispiel für eine visuelle Sprache ist die UML (siehe Abb. 2.7).

Stefan Schiffer hat in seinem Buch „Visuelle Programmierung“ den Begriff visuell noch etwas genauer gefasst.

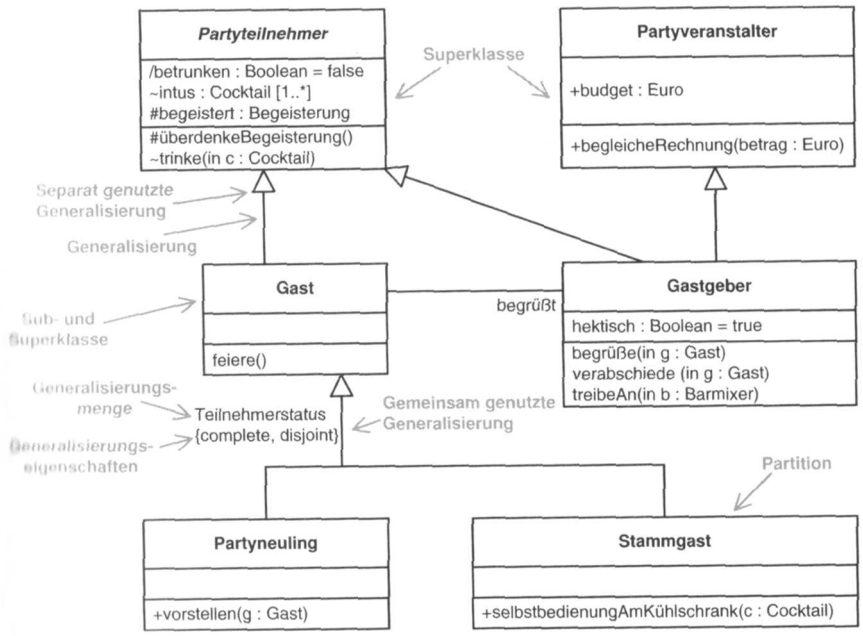

Abbildung 2.7: UML Klassendiagramm [5]

> „Der Begriff »Visuell«
> Visuell ist die Bezeichnung für jene Eigenschaft eines Objektes, durch die mindestens eine Information über das Objekt, die für das Erreichen eines Handlungsziels unverzichtbar ist, nur durch das visuelle Wahrnehmungssystem des Menschen gewonnen werden kann. “ [10]

Eine visuelle Sprache muss also eine visuelle Darstellung haben. Allerdings reicht dieses Kriterium nicht aus, um eine visuelle Sprache zu klassifizieren. Ansonsten müsste jede Zeichnung eine Ausprägung einer visuellen Sprache sein. Eine visuelle Sprache benötigt außer der visuellen Darstellung auch noch eine Syntax und eine Semantik. Eine Syntax, die z.B. beschreibt, in welcher Art und Weise die grafischen Symbole miteinander verknüpft oder wie diese im Raum platziert werden können. Und eine Semantik, die den Symbolen und den Beziehungen der Symbole Bedeutungen zuweist. Schiffer umschreibt visuelle Sprachen ähnlich.

> „Der Begriff »Visuelle Sprache«
> Eine visuelle Sprache ist eine formale Sprache mit visueller Syntax oder visueller Semantik und dynamischer oder statischer Zeichengebung.“ [10]

Eine klare Abgrenzung muss zu einer visuellen Programmiersprache gemacht werden. Eine visuelle Programmiersprache ist eine visuelle Sprache, mit der man Programme spezifizieren kann. Aus einer Ausprägung einer visuellen Programmiersprache sollte ein Programm erstellbar sein. Schiffer nennt härtere Kriterien für visuelle Programmiersprachen:

„Der Begriff »Visuelle Programmiersprache«
Eine visuelle Programmiersprache ist eine visuelle Sprache zur vollständigen Beschreibung der Eigenschaften von Software. Sie ist entweder eine Universalprogrammiersprache oder eine Spezialprogrammiersprache.“ [10]

2.3.2 Zusammenhang zu dieser Arbeit

An dieser Stelle soll nun der Zusammenhang zwischen visuellen Sprachen und dieser Arbeit hergestellt und der Begriff visuelle Programmiersprache davon abgegrenzt werden.

Um nun einen genauen Zusammenhang herzustellen, müssen die zwei Teile der Arbeit getrennt behandelt werden. Der eine Teil meiner Arbeit beschäftigt sich mit der Erstellung eines Programms zur Simulation und Animation von Logikbausteinen. Im zweiten Teil soll anhand dieses Programms untersucht werden, in wieweit sich DEViL für die Generierung von Animationsumgebungen eignet.

Das Programm zur Simulation und Animation von Logikbausteinen, das in dieser Arbeit erstellt wurde, enthält einen Editor für eine visuelle Sprache. Die Darstellung der Logikbausteine erfolgt visuell und unterliegt bestimmten syntaktischen Regeln. Aus diesem Grund besitzt diese Sprache eine visuelle Syntax. Diese visuelle Sprache ist allerdings keine visuelle Programmiersprache. Aus der Schaltungsbeschreibung wird keine Software generiert, sondern es erfolgt lediglich eine Simulation und Animation der Schaltung.

Im zweiten Teil der Arbeit soll das System DEViL, das für die Generierung von Editoren visueller Sprachen entwickelt wurde, auf Aspekte der Animation untersucht und ggf. erweitert werden. Dabei stellt sich natürlich die Frage, warum für ein System wie DEViL Animation überhaupt eine Rolle spielt. Animation und visuelle Sprachen haben die Gemeinsamkeit, dass sie beide grafische bzw. visuelle Darstellungen verwenden, um Informationen zu vermitteln. Visuelle Sprachen sind allerdings eher statischer Natur, die Animation ist dagegen dynamisch. Eine visuelle Sprache zu animieren, ist deshalb der Schritt von der statischen in die dynamische Darstellung. Meist kann die statische Darstellung der visuellen Sprache beibehalten werden. Die Animation erweitert diese Darstellung, um Programmzustände darzustellen. Am Beispiel der Petrinetze soll dies verdeutlicht werden. Petrinetze als visuelle Sprache definieren den Zustandsraum eines Systems. Die Folgezustände des Systems könnten durch eine Animation dargestellt werden (siehe Abb. 2.8). Da visuelle Sprachen und Animation so verwandt sind, ist es also sinnvoll, mit DEViL Editoren zu generieren, die die Animation einer visuellen Sprache erlauben.

2.3.3 DEViL

DEViL steht für „Development Environment for Visual Languages“ und wurde an der Universität Paderborn in der Arbeitsgruppe „Programmiersprachen und Über-

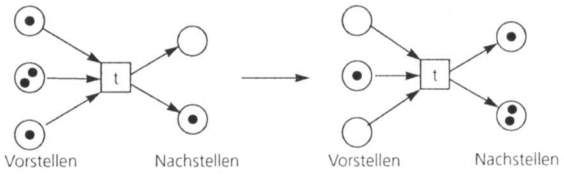

Abbildung 2.8: Petrinetze

setzer“ entwickelt. Es entstand aus einer Erweiterung des Eli Systems, das ein Werkzeugsystem zum Generieren von Übersetzern textueller Programmiersprachen ist.

DEViL verwendet attributierte Grammatiken, um visuelle Programmiersprachen zu spezifizieren. Unterstützt wird dieses Konzept durch Bibliotheken, die u.a. Visuelle Muster (engl. Patterns) zur Verfügung stellen. Diese Muster repräsentieren einfache grafische Strukturen, die in vielen visuellen Sprachen benutzt werden.

2.3.3.1 Aufbau von DEViL

In DEViL wird eine visuelle Sprache immer über eine abstrakte Struktur und Sichten auf diese Struktur spezifiziert. Die abstrakte Struktur ist eine Datenstruktur, die alle Daten visueller Ausdrücke und Beziehungen zwischen diesen Daten beschreibt. Sie wird auch oft als abstrakter Strukturbaum bezeichnet, weil eine Baumstruktur zugrunde liegt. Die Daten der abstrakten Struktur sind persistent, d.h. sie sind über die Laufzeit hinweg vorhanden. DEViL stattet zu diesem Zweck jede visuelle Programmierumgebung mit Funktionen zum Speichern und Laden dieser Daten aus.

Die grafische Repräsentation der Daten wird durch Sichten realisiert. Sichten visualisieren Teile der abstrakten Struktur. Zu einer abstrakten Struktur können beliebig viele Sichten definiert werden. An den Sichten vorgenommene Änderungen wirken sich direkt auf die abstrakte Struktur aus. Umgekehrt werden Änderungen an der abstrakten Struktur in jeder davon betroffenen Sicht sofort visualisiert. Dieses Konzept der Sichten ermöglicht also eine komplette Trennung zwischen Daten und Aussehen einer visuellen Sprache.

Die Generierung von Code wird auch direkt auf dem abstrakten Strukturbaum durchgeführt. Da der abstrakte Strukturbaum dem einer textuellen Sprache entspricht, kann die Codegenerierung analog zur Codegenerierung einer textuellen Sprache durchgeführt werden. Die nötigen Werkzeuge, die Eli bereits zur Codegenerierung von textuellen Sprachen zur Verfügung stellt, können hier im vollen Umfang genutzt werden.

2.3.3.2 Zugrundeliegende Konzepte

Die Spezifikation der abstrakten Struktur ist das Einzige, das in DEViL zwingend erforderlich ist. Weder eine Sichtdefinition noch die Codegenerierung muss vorhanden sein. Das liegt u.a. daran, dass alle anderen Spezifikationen auf der abstrakten Struktur aufbauen. Um die abstrakte Struktur zu beschreiben, verwendet DEViL

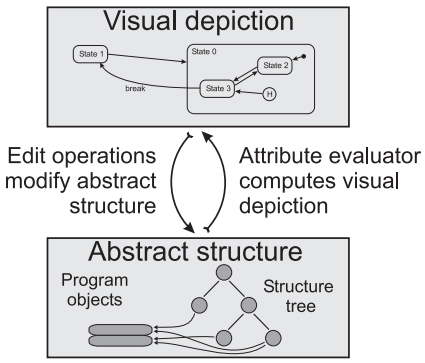

Abbildung 2.9: Zusammenwirken von Abstrakter Struktur und Sichten
Quelle: [13]

eine spezielle Spezifikationssprache. Diese Sprache ähnelt stark der Klassenbeschreibung einer objektorientierten Sprache. Ein Beispiel aus dem Handbuch von DEViL [11] für die Spezifikation der abstrakten Struktur ist hier zu sehen:

```
CLASS Transition {
  name: VAL VLString;
  from: REF State;
  to: REF State;
  actions: SUB Action*;
}
```

Dabei stellen die Klassen und die Attribute jeweils Baumknoten dar. Die VAL-Attribute speichern Werte in ihren Knoten, die SUB-Attribute Unterbäume und die REF-Attribute Querreferenzen im Baum. Querreferenzen sind vorhanden, um Abhängigkeiten zwischen den Teilbäumen darzustellen. In vielen visuellen Sprachen kann man die Struktur nicht direkt auf einen Baum abbilden. Für diesen Zweck kann man Querreferenzen im Baum benutzen, um eine solche Struktur richtig abzubilden. Für die VAL-Attribute gibt es in DEViL vordefinierte Datentypen, die benutzt werden können, um z.B. Zeichenketten in den abstrakten Baum einzufügen.

Die Sichten werden durch attributierte Grammatiken spezifiziert. Um einen genaues Verständnis der Sichtdefinitionen zu bekommen, werden im Folgenden attributierte Grammatiken in DEViL beschrieben. Darauf aufbauend kann dann die Beschreibung der Sichtdefinition folgen.

Attributierte Grammatiken sind eine Standardtechnik, die im Übersetzerbau schon seit langem z.B. für Typenprüfung, Namenanalyse und Zwischencode-Erzeugung verwendet wird.

> „Eine attributierte Grammatik erweitert eine kontextfreie Grammatik um kontextabhängige Spezifikationen in Form von Regeln zur Berechnung von Attributen.“ [6]

Mit anderen Worten, bei attributierten Grammatiken werden die Produktionen einer kontextfreien Grammatik um Regeln erweitert. Diese Regeln beschreiben die Berechnung von Attributen. In DEViL wird die Sprache Lido benutzt, um attributierte Grammatiken auszudrücken. Diese Sprache kommt aus dem Werkzeugsystem Eli, wo sie u.a. für die oben genannten Zwecke bei der Generierung textueller Übersetzer benutzt wird. In DEViL wird dabei eine spezielle Form der Attributierung benutzt. Hierbei werden nicht Produktionen sondern sog. Symbole mit bestimmten Regeln zur Attributierung versehen. Diese Art der Attributierung wird Symbolattributierung genannt.

Knoten des abstrakten Strukturbaums sind in diesem Kontext Instanzen der Symbole. Die Attributauswertung, d.h. die Berechnung der Attribute aufgrund der Symbolattributierung, wird auf dem abstrakten Strukturbaum durchgeführt. Die dabei zu berechnenden Attribute werden dem Baum hinzugefügt. Es werden nicht die Knoten des abstrakten Strukturbaumes geändert. Aus den Lido Dateien werden von dem Eli System Attributauswerter generiert.

Mit attributierten Grammatiken können verschiedene Sichten auf einen abstrakten Strukturbaum spezifiziert werden. Diese Sichten sind voneinander unabhängig, d.h. weder die Attribute noch andere Teile einer Sichtdefinition auf andere Sichtdefinition Einfluss haben. Die einzige Gemeinsamkeit aller Sichtdefinitionen ist, dass Sie auf der gleichen Datenstruktur arbeiten, der abstrakten Struktur. Die vom Attributauswerter berechneten Attribute bleiben nur so lange am Leben, wie der Attributauswerter läuft. Nach einem Lauf werden diese Attribute verworfen.

Bei der Beschreibung der Sichten werden attributierte Grammatiken im Zusammenhang mit visuellen Mustern benutzt. Die berechneten Attribute sind z.B. die Größe und Position eines visuellen Objekts auf dem Bildschirm. Die visuellen Muster in Form einer Bibliothek bieten bestimmte „Standard“-Attributberechnungen, die auf die Symbole übertragen werden können. Indem bestimmte Symbole der Grammatik ein oder mehrere dieser Muster erben (das Erben dieser Muster ist nur für eine Sicht gültig), werden Standard-Funktionalitäten des Musters auf die Visualisierung des Knotens übertragen. Eine solche Funktionalität eines Musters ist z.B. die Bewegbarkeit eines Objektes frei in einem bestimmten Raum des Mengen-Musters. Die genaue Beschreibung der Muster können Sie der Diplomarbeit Schmidt/Schindler [2] entnehmen.

Die Definition der Codegenerierung funktioniert in DEViL analog zu der Definition von Sichten. Es wird auch hierfür eine attributierte Grammatik entworfen, diesmal ohne die Benutzung von Mustern, aus der ein Attributauswerter generiert werden kann, der aus dem abstrakten Strukturbaum Code generieren kann. Unterstützt wird lido in diesem Fall von einigen Werkzeugen, die auch während der Spezifikation von Sichten benutzt werden können. Eines dieser Werkzeuge ist PTG (Pattern based Text Generator), der die Ausgabe von strukturiertem Text unterstützt.

2.3.3.3 Technische Aspekte

DEViL generiert aus den verschiedenen Spezifikationsdateien C und TCL Code. Hierzu durchläuft die Spezifikation schrittweise verschiedene Übersetzer. Als letzter Schritt wird aus dem C-Quelltext ausführbarer Code erstellt. Der C-Code enthält dabei einen Interpreter, der den TCL-Teil der Anwendung zur Laufzeit interpretieren kann. Alles, was man mit DEViL und den zur Verfügung stehenden Werkzeugen nicht lösen kann, kann aus diesem Grund in TCL und C programmiert werden. DEViL stellt dabei geeignete Schnittstellen zur Verfügung, um selbst programmierten Code direkt in die spezifizierte Software einzubinden. Aus Lido beispielsweise können C-Funktionen direkt aufgerufen werden. Außerdem gibt es eine Menge Bibliotheksfunktionen, die von C- sowie von TCL-Code aufgerufen werden können.

3 Umsetzung

In dieser Arbeit wurde ein Programm entwickelt, das die Simulation und Animation von Logikbausteinen ermöglicht. Das Programm gliedert sich in drei Teile, den Editor, die Simulation und die Animation, die im Folgenden beschrieben werden. Doch zuvor soll kurz die Zielsetzung verdeutlicht werden, unter der dieses Programm entwickelt wurde.

3.1 Zielsetzung

In erster Linie diente das zu entwickelnde Programm zur Animation und Simulation von Logikbausteinen dem Zweck, DEViL auf die Tauglichkeit zur Generierung von Animationsumgebungen zu untersuchen. Interessant könnte dieses Programm allerdings auch für Lehrzwecke sein. Einfache Schaltungen sollten damit verständlich dargestellt werden können. In der Industrie gibt es mit großer Wahrscheinlichkeit keinen Einsatzzweck für dieses Programm. In einem solchen Umfeld werden Programme benötigt, die ihren Fokus sehr auf die elektrotechnischen Eigenschaften legen und im Rahmen dieser Arbeit nicht einmal ansatzweise konzipiert werden könnten. Aus diesem Grund musste die Konzeption dieses Programms eher auf Lehrzwecke ausgerichtet sein, wobei hier an Lehreinrichtungen im schulischen Bereich gedacht wird. Für universitäre Lehrzwecke wird eher mit Programmen gearbeitet, die auch in der Industrie zum Einsatz kommen.

Auf Grundlage dieser Zielsetzung kann nun die Umsetzung beschrieben werden. Die Entscheidungen in der Konzeption dieses Programms hängen dabei stark von dem vorgestellten Szenario ab.

3.2 Der Editor

Der sichtbare Teil des Programms besteht hauptsächlich aus dem Editor, der es ermöglicht, eine digitale Schaltung zu zeichnen. Die Simulation und die Animation bauen auf diesem auf. Es wird nun zunächst ein Überblick über die Funktion des Editors gegeben. Darauf folgend wird die Umsetzung des Editors genauer beschrieben. Der Editor wurde vollständig mit dem Werkzeugsystem DEViL entwickelt. Dafür wurde ein Modell erstellt und zwei Sichten auf dieses Modell spezifiziert.

3.2.1 Überblick

Der Editor enthält Standardfunktionalitäten wie das Erstellen, Öffnen, Speichern und Schließen von Dateien. In einer Datei können mehrere Schaltungen gezeichnet werden. Beim Erstellen oder Öffnen einer Datei wird eine Übersichtsdarstellung geöffnet, in der beliebig viele Schaltungen eingefügt werden können.

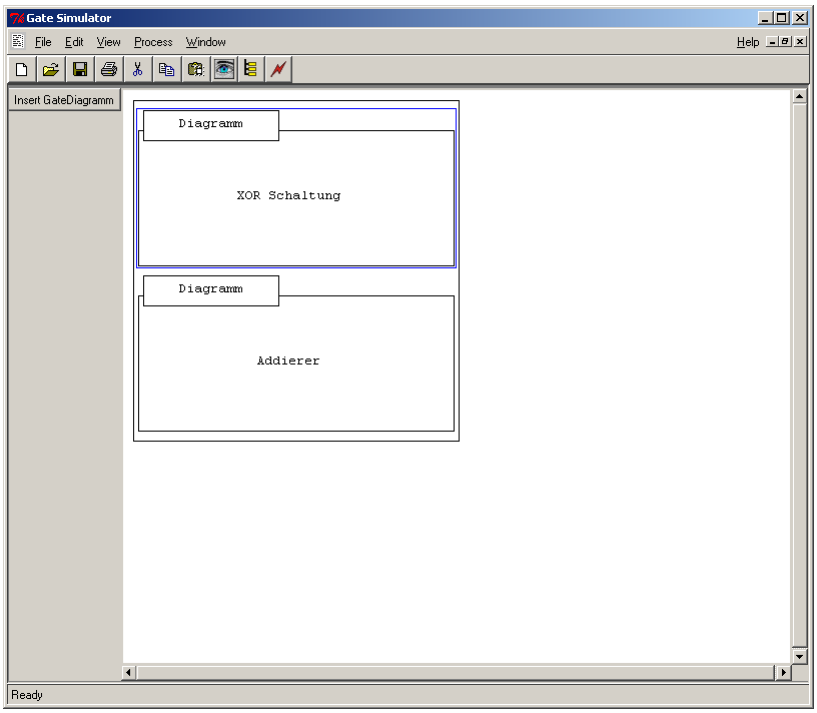

Abbildung 3.1: Übersichtsdarstellung

Durch das Selektieren einer dieser Schaltungen gelangt man zu der eigentlichen Zeichenfläche, in der die Schaltung gezeichnet werden kann. In Abb. 3.2 wurde die XOR-Schaltung geöffnet und ist nun auf der Zeichenfläche sichtbar.

Neben der Zeichenfläche befindet sich eine Reihe von Buttons. Mit Hilfe dieser Buttons können die verschiedenen Objekte auf der Zeichenfläche platziert werden. Einzufügende Objekte sind die Gatter, die Ein- und Ausgänge der Schaltung und die Leitungen, die die Schaltungselemente miteinander verbinden. Die Schaltungserstellung beschränkt sich auf wenige Grundelemente. Die eingesetzten Logikgatter sind das UND-, das ODER- und das NICHT-Gatter. Dazu kommt, dass jeder Ein- und Ausgang dieser Gatter negiert werden kann. Mit einem Doppelklick auf einen Ein- oder Ausgang öffnet sich ein Fenster, in dem die Negationseigenschaft geändert werden kann (siehe Abb. 3.3).

Negierte Ein- und Ausgänge werden mit einem Kreis dargestellt. Mit dieser Eigenschaft können somit auch NAND- und NOR-Gatter erstellt werden, ohne diese aus zwei Gattern zusammensetzen zu müssen. Ansonsten werden für die Schaltungserstellung nur noch die Ein- und die Ausgänge der Schaltung sowie die Leitungen

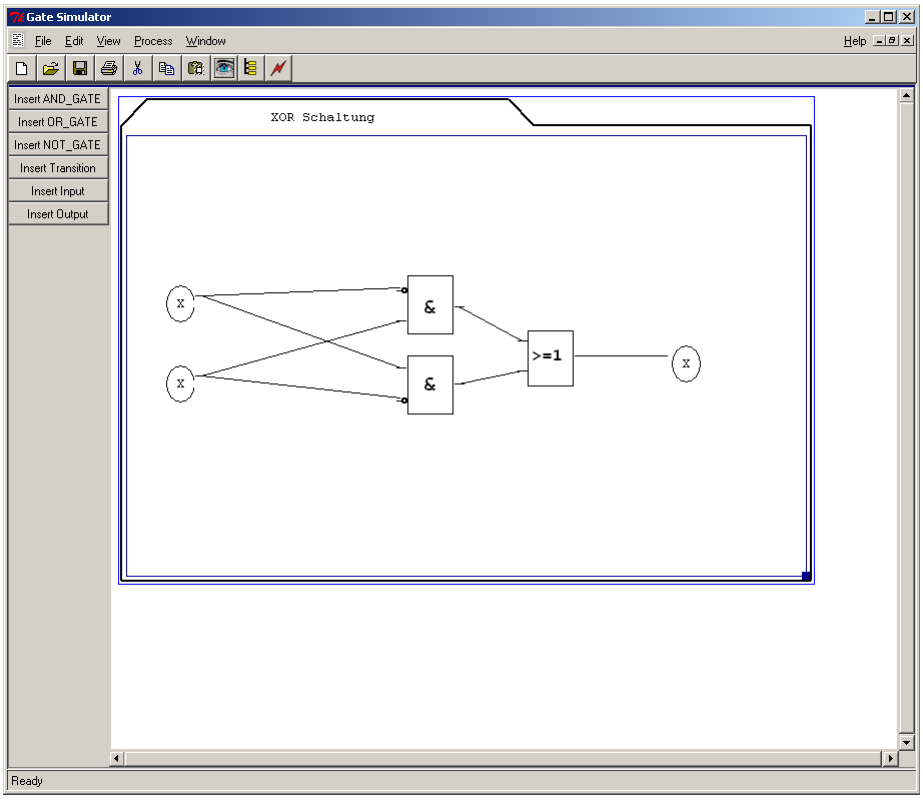

Abbildung 3.2: XOR-Schaltung

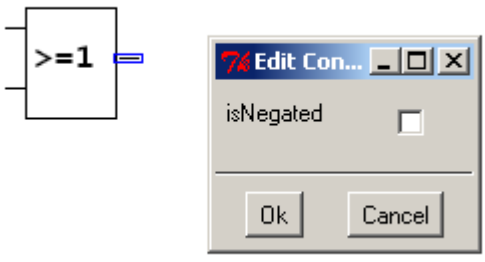

Abbildung 3.3: Negieren eines Ausganges des ODER-Gatters

zwischen diesen benötigt. Für die Eingänge der Schaltung kann über das Kontextmenü ein Fenster geöffnet werden, in dem sich die Eingabewerte der Schaltung ändern lassen (siehe Abb. 3.4).

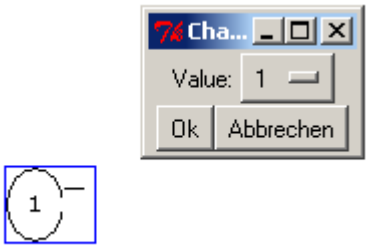

Abbildung 3.4: Eintragung der Eingabewerte

Die Leitungen lassen sich per Drag and Drop mit den Ein- und Ausgängen der Schaltungselemente verknüpfen. Diese Grundelemente der Schaltung reichen aus, um einfache Schaltungen zu erstellen.

3.2.2 Das abstrakte Struktur-Modell

Instanzen des abstrakten Struktur-Modells sind Bäume. In Abb. 3.5 ist eine Ausprägung zu sehen. Die Wurzel des Baumes bildet der Knoten Root. Diese Wurzel hat als einzigen Unterbaum eine Menge von digitalen Schaltungen, auch GateDiagramm genannt. Ein GateDiagramm besitzt wiederum eine Menge von Gattern (Gates) und eine Menge von Leitungen (Wires).

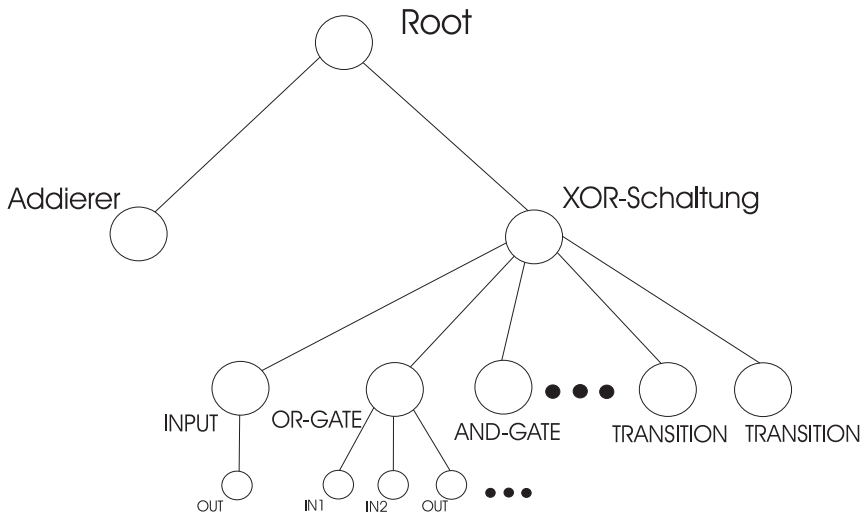

Abbildung 3.5: Abstrakte Struktur (eine Ausprägung des Modelles)

Diese Ausprägung ist eine starke Vereinfachung der wirklichen abstrakten Struktur, die eine Instanz des Modells ist. Die wesentlichen Knoten sind allerdings enthal-

ten. Wie ebenfalls in Abb. 3.5 zu sehen werden die Ein- und Ausgänge der Schaltung wie Gatter betrachtet und sind aus diesem Grund im Modell Nachbarknoten des Gatters.

Jedes Gatter bzw. die Ein- und Ausgänge der Schaltungen haben Ein- und Ausgänge. Diese Ein- und Ausgänge werden im Modell als eigene Knoten behandelt, die als SUB-Attribute noch unter den Gattern bzw. den Ein- und Ausgängen der Schaltungen liegen. Diese Ein- und Ausgänge der Gatter haben eine Eigenschaft, die angibt, ob dieser Ein- bzw. Ausgang negiert ist.

Ein NICHT-Gatter hat einen negierten Ausgang. Um dies auch im Modell abzubilden, muss der Ausgang des Gatters bei der Initialisierung negiert werden. Dies übernimmt eine selbstgeschriebene Initialisierungsmethode, die eine kontextsensitive Initialisierung durchführt.

Die Entwicklung der abstrakten Struktur als Baum ist in DEViL vorgeschrieben. Eine digitale Schaltung dagegen weist eher die Form eines Graphen auf. Um diesen Graphen auf den Baum abzubilden, werden Querreferenzen zwischen den Baumknoten benutzt. Jede Leitung besitzt zwei Querreferenzen auf die Ein- bzw. Ausgänge der Gatter, mit denen sie verbunden sind.

3.2.3 Grafische Darstellung

Für die grafische Darstellung wurden zwei Sichten entworfen. Eine Übersichtsdarstellung (rootView), in der man digitale Schaltungen einfügen und auswählen kann (siehe Abb. 3.1) und die eigentliche Zeichenfläche (gateView), in der man die Schaltungen zeichnen kann (Abb. 3.2). Beide Sichten wurden, wie in DEViL üblich, mit attributierten Grammatiken spezifiziert. Die Übersichtsdarstellung basiert auf einem einfachen Muster und einer generischen Zeichnung. Das SimpleList Muster wurde verwendet, um die verschiedenen digitalen Schaltungen grafisch untereinander anzuordnen. Die digitalen Schaltungen werden jeweils durch eine generische Zeichnung repräsentiert. Generische Zeichnungen werden in DEViL im Zusammenhang mit dem Formularmuster verwendet. Die grafische Darstellung der Ausprägung einer Formular-Darstellung lässt sich mit einer generischen Zeichnung ausdrücken. Dazu existiert in DEViL ein Editor, mit dem diese Zeichnungen erstellt werden können (siehe Abb. 3.6).

Die gateView-Sicht ist dagegen um einiges komplizierter. Grundlage dieser Sicht ist das Mengen-Muster (VPSet) und das Verbindungs-Muster (VPConnection). Als Wurzel der Sichtspezifikation der gateView-Sicht dient ein GateDiagramm-Knoten, der in der abstrakten Struktur unter der Wurzel hängt. Der obere Teil des Baums bleibt in der Sicht verborgen.

Die Leitungen der Schaltung werden durch das VPConnection-Muster modelliert. Sie können innerhalb der VPConnectionArea frei bewegt und mit einem VPConnectionEndpoint verbunden werden. Die Wurzel bzw. das GateDiagramm bildet die ConnectionArea; die Ein- und Ausgänge der Gatter bilden jeweils den VPConnectionEndpoint. Die Gatter werden durch Vererbung der VPSetElement-Rolle zu frei positionierbaren Objekten.

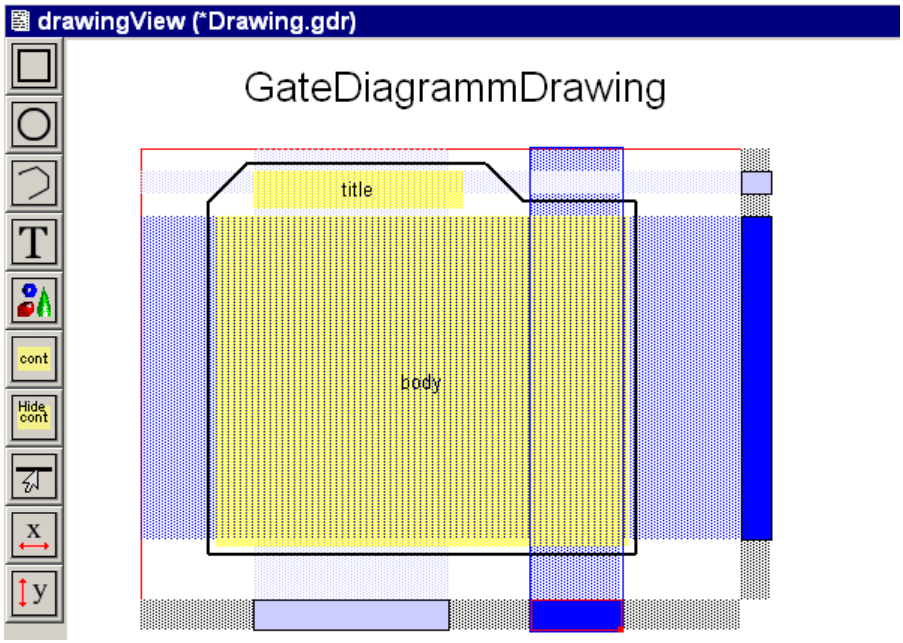

Abbildung 3.6: Generische Zeichnung

Alle generischen Zeichnungen, z.B. die Gatter, die Ein- und Ausgänge usw. werden über das Formular-Muster eingebunden. Die Ein- und Ausgänge der Gatter werden je nachdem, ob diese negiert sind oder nicht, mit einem Kreis oder ohne diesen gezeichnet.

Die grafische Darstellung der Gatter erfolgte in Anlehnung an die DIN EN 60617-12 [1].

3.3 Simulation

Eine Simulation kann auf Basis einer fertigen Zeichnung der Schaltung erfolgen. Die Simulation kann unter dem Menüpunkt „simulate“ im „Process“ Menü gestartet werden. Daraufhin erscheint ein Fenster, mit dem die Simulation gesteuert werden kann (siehe Abb. 3.7).

Abbildung 3.7: Simulationssteuerung

Die Oberfläche ähnelt der eines typischen Wiedergabegerätes wie z.B. eines CD-Spielers. Mit dem grünen Start-Button kann die Simulation gestartet werden. Der Button neben dem Start-Button ist vorhanden, um nur einen Simulationsschritt

auszuführen. Die folgenden Stop- und Pause-Buttons können die Simulation abbrechen bzw. anhalten. Der letzte Button öffnet das Konfigurationsfenster. Mit diesem können Einstellungen für die Animation vorgenommen werden. Eine detaillierte Beschreibung des Konfigurationsfensters folgt in Abschnitt 3.4.1.

Mit der Funktion „simulateNext“ aus dem „Process“ Menü können die nachfolgenden Werte einer Simulation berechnet werden. Soll beispielsweise ein Flip Flop (ein Bauelement oder eine kleine Schaltung, die Werte speichert) simuliert werden, so kann dies nur in mehreren Simulationsläufen mit mehreren Eingabewerten geschehen. Im ersten Simulationslauf könnte z.B. ein Wert in das Flip Flop gespeichert und im nächsten Lauf wieder ausgelesen werden. Um den zweiten Simulationslauf zu starten, muss in diesem Fall die Simulation mit „simulateNext“ weitergeführt werden. Die Benutzung der Funktion „simulate“ würde an dieser Stelle dazu führen, dass die Schaltung neu initialisiert werden würde und die Speicherung des Wertes damit verloren wäre.

3.3.1 Simulationsmodell

Wie bereits in 2.2.2 vorgestellt, ist es für die Simulation von digitalen Schaltungen wichtig, ein Modell zu erstellen. Das Modell soll die physikalischen Eigenschaften einer Schaltung abstrahieren. Das in dieser Arbeit aufgestellte Modell soll an dieser Stelle vorgestellt werden. Wie auch in 2.2.2 betrachten wir im Folgenden die Verzögerung der Gatter, die Signalzustände und zusätzlich die Simulationszeit.

3.3.1.1 Verzögerungen

Wie bereits erwähnt, hängt die Auswahl des Verzögerungsmodells stark davon ab, wie genau die Realität simuliert werden soll. In 3.1 sind als Anwendungsmöglichkeit Lehrzwecke angegeben worden. Unter diesem Gesichtspunkt wurde ein einfaches Verzögerungsmodell für die Simulation ausgewählt. Um eine einfache Verzögerung zu simulieren, wird das Unit-Delay Modell benutzt. Dieses Modell reicht aus, um schwingende Schaltungen zu simulieren oder Hazards aufzudecken. Es berücksichtigt allerdings nicht verschiedene Verzögerungen bei an- oder absteigenden Flanken. Dies ist im Lehrkontext allerdings meist auch nicht von sehr großer Bedeutung. Ein Modell mit verschiedenen Verzögerungen für die einzelnen Eingänge kommt allein deswegen nicht in Frage, weil hier nur Grundbausteine simuliert werden. Diese Grundbausteine wie das UND- oder das ODER-Gatter haben jeweils die gleiche Verzögerung an beiden Eingängen. Zwar wäre es möglich, dass ein Eingang eines Gatters negiert wird, allerdings kommt es in diesem Fall auf die interne Schaltung an, ob dieser Eingang eine andere Verzögerung hat als der nicht negierte Eingang. Das Unit-Delay-Modell scheint also auszureichen.

Jeder Gattertyp hat im Modell die gleiche Verzögerung. Diese Annahme ist eine weitere Vereinfachung und könnte in professionellen Simulatoren nicht gemacht werden. Das hier erstellte Programm verwendet allerdings nur einfache Gattertypen, die in der Realität zumindest ähnliche Verzögerungen aufweisen. Aus diesem Grund

ist diese Vereinfachung besonders in Bezug auf den Anwendungszweck akzeptabel. Jedes Gatter hat eine Verzögerung von einer Simulationszeiteinheit.

3.3.1.2 Signalzustände

Für die Darstellung der Signale wird eine dreiwertige Logik mit den Zuständen 0, 1 und X verwendet. Die dreiwertige Logik bietet den Vorteil, dass undefinierte Zustände modelliert werden können. Dies wird u.a. benutzt, um die Schaltung zu initialisieren. Die Initialisierung findet vor der eigentlichen Simulation statt. Jeder Ein- und Ausgang der Logikbausteine wird vor dem Start der Simulation auf X gesetzt. Der Zustand X wird weiterhin auch bei der Berechnung der Gatter verwendet. Hierzu werden die in Abb. 3.8 vorgestellten Wahrheitstabellen benutzt.

AND

	0	X	1
0	0	0	0
X	0	X	X
1	0	X	1

OR

	0	X	1
0	0	X	1
X	X	X	1
1	1	1	1

NOT

0	X	1
1	X	0

Abbildung 3.8: Wahrheitstabellen Dreiwertiger Logik Quelle: [7]

In diesem Zusammenhang ist mit dem Zustand X nicht eine fehlerhafte Spannung am Eingang eines Gatters gemeint (wie er in der Elektrotechnik teilweise benutzt wird), sondern ein Zustand, von dem nicht sicher ist, ob er 0 oder 1 ist. Die Wahrheitstabellen entsprechen dabei auch der logischen Funktion dieser Gatter. Ist z.B. bei einem UND-Gatter ein Eingang 0, so ist es egal, ob am anderen 0 oder 1 anliegt, der Ausgang ist in jedem Fall 0. Aus dieser Tatsache heraus kann, wenn an einem Eingang 0 und am anderen X anliegt, der Ausgang nur 0 liefern.

Die Verwendung der dreiwertigen Logik hat im Gegensatz zur zweiwertigen Logik den Vorteil, dass diese Logik eine Simulation von Schaltungen mit Rückkopplung zulässt. Weil Schaltungen mit Rückkopplung besonders interessant für Lehrzwecke sind, ist an dieser Stelle die Entscheidung auf die dreiwertige Logik gefallen. Die Benutzung einer vierwertigen Logik mit einem hochohmigen Zustand ist in diesem Programm wenig sinnvoll, weil keine Tristate-Buffer modelliert werden. Das Programm ist generell nicht für die Simulation von Bussystemen gedacht, so dass der hochohmige Zustand gar nicht erst auftreten würde.

3.3.1.3 Simulationszeit

Um die Zeit während einer Simulation zu modellieren, wird eine Simulationszeit verwendet. Die Simulationszeit stellt, wie in Kap. 2.2.2.2 beschrieben, die kleinste Zeiteinheit während der Simulation dar. Sie dient dazu, allen Ereignissen einen Zeitpunkt zuzuordnen, zu dem diese auftreten, und dazu eine Verzögerung der Gatter zu modellieren. Die Simulationszeit ist am Anfang einer Simulation 0 und wird in

jedem Simulationsschritt erhöht. Ein Simulationsschritt umfasst die Abarbeitung aller Ereignisse innerhalb einer Simulationszeiteinheit.

3.3.2 Simulationslauf

Hauptbestandteil des interpretierenden Verfahrens ist in dieser Arbeit eine Ereignisliste. In dieser Liste werden alle Ereignisse, die während der Simulation auftreten, gespeichert. Ereignisse sind in diesem Zusammenhang Änderungen der Werte am Ausgang eines Gatters oder am Ausgang eines Eingangs der Schaltung. In einem Ereignis wird sowohl die Simulationszeit, in der es auftritt, wie auch eine Referenz auf das Gatter, bei dem es auftritt, gespeichert. Die Ereignisse werden nach der Simulationszeit, in der sie auftreten, sortiert in der Ereignisliste gespeichert. Ändert sich also während der Simulation der Ausgang eines Gatters, wird dieses der Ereignisliste hinzugefügt.

Während der Konvertierung werden alle Eingänge der Schaltung als Ereignis zum Simulationszeitpunkt 0 in die Ereignisliste geschrieben. Somit werden alle Eingänge im ersten Simulationsschritt einmal bearbeitet. Während der Simulation werden in einem Simulationsschritt nacheinander alle Ereignisse der aktuellen Simulationszeit aus der Ereignisliste geholt. Das zum aktuellen Ereignis gehörende Gatter wird aus dem Ereignis ausgelesen. Die diesem nachfolgenden Gatter werden nacheinander abgearbeitet. Dabei wird der zugehörige Eingang dieser mit dem neuen Wert (der Wert, der am Ausgang des Gatters anliegt, welches dem aktuellen Ereignis zugeordnet ist) belegt. Falls sich der Ausgang eines dieser Gatter nach der Belegung mit dem neuen Wert am Eingang ändert, so wird dieses Gatter in die Ereignisliste eingetragen. Die Simulationszeit der Änderung eines Gatter-Ausgangs ergibt sich aus der Simulationszeit der Eingabe-Änderung plus der Verzögerungszeit des Gatters. Die Simulation läuft solange, bis die Ereignisliste leer ist oder die Simulation abgebrochen wird.

3.3.3 Umsetzung

Für die Simulation wird das Modell, wie es der Editor benötigt, in ein anderes Modell umgewandelt. Diese Konvertierung wurde mit Hilfe von attributierten Grammatiken spezifiziert. Hierzu wurden Mechanismen verwendet, die in DEViL normalerweise für die Codegenerierung vorgesehen sind. Bei der Attributauswertung wird eine C++ Datenstruktur erzeugt. Diese Datenstruktur wird dem Simulator übergeben, der die Simulation durchführt. Die Datenstruktur stellt die digitale Schaltung als gerichteten Graph dar. Die Knoten des Graphen sind Gatter, die Kanten die Leitungen zwischen diesen. Jedes Gatter enhält demnach eine Liste der direkten Nachfolger im Graphen. Gatter sowie ein Ein- und Ausgänge der Schaltung werden durch C++ Objekte repräsentiert. Die Leitungen sind nur noch als Zeiger auf die nachfolgenden Objekte vorhanden. Ein Teil der Informationen, z.B. die Position der Elemente, geht bei der Konvertierung verloren. In der neuen Struktur werden allerdings auch Informationen gespeichert, die in der abstrakten Struktur nicht vorhanden sind. Dies sind z.B. die

Zwischenwerte der Gatter. Die neue Struktur enthält außerdem noch Referenzen auf die alte Struktur, um eine Beziehung zwischen diesen beiden Strukturen herzustellen. Diese Beziehung wird bei der Animation benötigt.

Während der Konvertierung des einen Modells in das andere werden dem Simulator gleichzeitig auch die Eingabedaten der Schaltung mitgeteilt. Wird die Funktion „simulateNext„ verwendet, findet keine Konvertierung sondern nur eine Mitteilung der Eingabedaten statt.

Die Konvertierung des Modells ist notwendig, weil das Modell, das dem Editor zugrunde liegt, ungeeignet für die Simulation ist. Für die Simulation sind immer die nachfolgenden Gatter am wichtigsten. Das Modell des Editors ist allerdings nicht dazu geeignet, diese schnell zu finden, sondern die abstrakte Struktur einfach darzustellen. Eine C++ Datenstruktur wurde gewählt, weil Lido, die Sprache in der die Konvertierung spezifiziert wurde, eine Schnittstelle zu C++ enthält und sich diese Sprache somit anbot. Auch die Implementierung des Simulationsprogrammes als solches erfolgte komplett in C++.

Das Simulationsprogramm wird im Anschluss an die Konvertierung direkt gestartet. Dieses wiederum öffnet das Kontrollfenster. Die Simulation ist fast unabhängig von dem Editor. Mit einigen Veränderungen könnte die Simulation auch als eigenständiges Programm laufen.

Die Entscheidung für ein Simulationsverfahren ist auf das interpretierende Verfahren gefallen. Dieses Verfahren ist zwar langsamer als das compilerorientierte, allerdings ist dies bei der Größe der Schaltungen, die innerhalb von Lehrzwecken benutzt werden, irrelevant. Für eine Simulation von sehr großen Schaltungen wären ohnehin weitere Mechanismen notwendig, wie z.B. das Modularisieren von Schaltungen, die dieses Programm nicht unterstützt. Der Effizienzvorteil wäre also nicht erkennbar. Bei diesem Anwendungsszenario überragen die Vorteile des interpretierenden Verfahren also deutlich die Nachteile des compilerorientierten. Dieses sind vor allem die Simulierbarkeit von Verzögerungen und die einfachere Implementierung, da alles in einem Prozess auf dem Rechner ausgeführt wird. Um trotz des interpretierenden Verfahrens möglichst schnell zu simulieren wird ereignisorientiert simuliert.

3.4 Animation

3.4.1 Überblick

Die Animation einer Schaltung kann in dem hier entwickelten Programm auf mehrere Weisen geschehen. Wie die Animation durchgeführt wird, hängt davon ab, welche Einstellungen im Konfigurationsfenster vorgenommen wurden (siehe Abb. 3.9). Das Fenster besteht aus drei Teilen: einem allgemeinen Auswahlfeld „flüssige Animation“ und zwei Rahmen mit den Überschriften „Animation“ und „keine Animation“. Im allgemeinen Auswahlfeld kann entschieden werden, ob die Simulationsergebnisse durch eine flüssige Animation dargestellt werden oder nicht. Davon abhängig wird entweder der obere Rahmen „Animation“ oder der untere Rahmen „keine Animati-

on“ aktiv. Der andere ist jeweils inaktiv (grau hinterlegt).

Wird die flüssige Animation ausgewählt, kann im oberen Rahmen über die Art der Animation entschieden werden. Zurzeit sind die Optionen „fliegende Einsen“ und „dicker werdendes Kabel“ auswählbar, wobei nur die Option „fliegende Einsen“ funktionsfähig ist. Wird die Option „fliegende Einsen“ ausgewählt, so fliegt bei jeder Änderung eines Wertes an einer Leitung dieser Wert an der Leitung entlang. Entlang-Fliegen bedeutet, dass der Wert der Leitung sich in kleinen Schritten vom Ausgang eines Gatters zum Eingang des nächsten Gatters fortbewegt (siehe Abb. 3.14). Bei der anderen flüssigen Animationsart „dicker werdendes Kabel“ (die noch nicht implementiert ist), sollte bei der Änderung eines Wertes an einer Leitung statt eines fliegenden Wertes die Leitung langsam dicker werden. Damit ist gemeint, dass die Leitung Stück für Stück vom Ausgang bis zum Eingang des nächsten Gatters breiter wird.

Wählt man im allgemeinen Auswahlfeld „Nein“, können im unteren Rahmen verschiedene Einstellungen vorgenommen werden. Diese Einstellungen sind „Einzelne Schritte anzeigen“, „Ergebnisse aller Gatter anzeigen“ und die „Verzögerungszeit“. Die Einstellung „Verzögerungszeit“ ist im Zusammenhang mit der Einstellung „Einzelne Schritte anzeigen“ zu betrachten. Wird die Einstellung „Einzelne Schritte anzeigen“ augewählt, wird jeder Simulationsschritt und damit auch die Animation von diesem verzögert ausgeführt. Über die „Verzögerungszeit“ kann eingestellt werden, wieviele Sekunden nach einem Simulationsschritt gewartet wird, bis der nächste ausgeführt wird. Mit der Einstellung „Ergebnisse aller Gatter anzeigen“ kann ausgewählt werden, ob an jedem Gatter die aktuellen Werte der Ein- und Ausgänge dargestellt werden oder nicht (siehe Abb. 3.11).

Welche Art der Animation gewählt werden sollte, hängt von dem Anwendungsszenario ab. Ist beispielsweise nur das Endergebnis des Simulationslaufes interessant, kann die Standard-Einstellung (Einstellung wie Abb. 3.9) beibehalten werden. Diese Einstellung führt dazu, dass nur die Ausgänge der Schaltung animiert werden, ohne dass eine Verzögerung zwischen den Simulationsschritten stattfindet (siehe Abb. 3.10).

Bei der Beobachtung von größeren Schaltungen ist es auch interessant, wie das Endergebnis zustande kommt, d.h. an welchem Gatter welcher Wert anliegt. Um dies zu beobachten muss zusätzlich die Einstellung „Ergebnisse aller Gatter anzeigen“ aktiviert sein (siehe Abb 3.11).

Um einen Laufzeiteffekt wie z.B. einen Hazard der Schaltung zu beobachten, muss man die Einstellung „Einzelne Schritte anzeigen“ aktivieren. Gleichzeitig sollte eine Verzögerungszeit eingestellt werden, um den Effekt sichtbar zumachen. Die Verzögerung ist notwendig, um die schrittweise Veränderung der Schaltung zu visualisieren. In Abb. 3.12 sieht man die Simulation und Animation eines Hazards. Um diesen zu erzeugen, musste die Schaltung als erstes mit der Eingabe 0 simuliert werden. Danach konnte mit einer 1 am Eingang die Simulation fortgesetzt werden. Dies geschah mit der Funktion „simulateNext“ aus dem „Process“ Menü.

In Kombination mit der Einstellung „Ergebnisse aller Gatter anzeigen“ kann die Entstehung eines Hazards beobachtet werden. In dieser Kombination kann der Be-

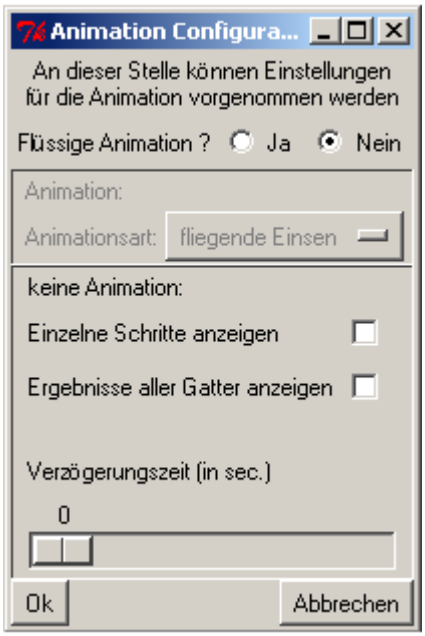

Abbildung 3.9: Animationskonfiguration

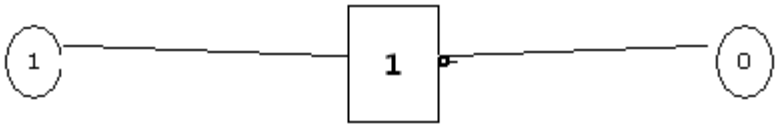

Abbildung 3.10: Schaltung nach Simulation

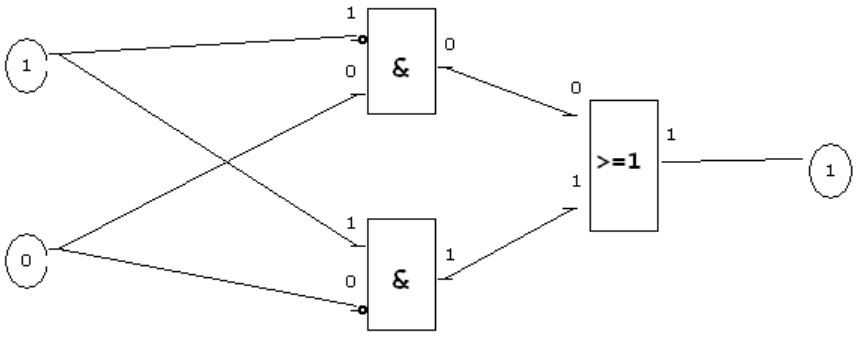

Abbildung 3.11: Schaltung nach Simulation mit allen Zwischenergebnissen

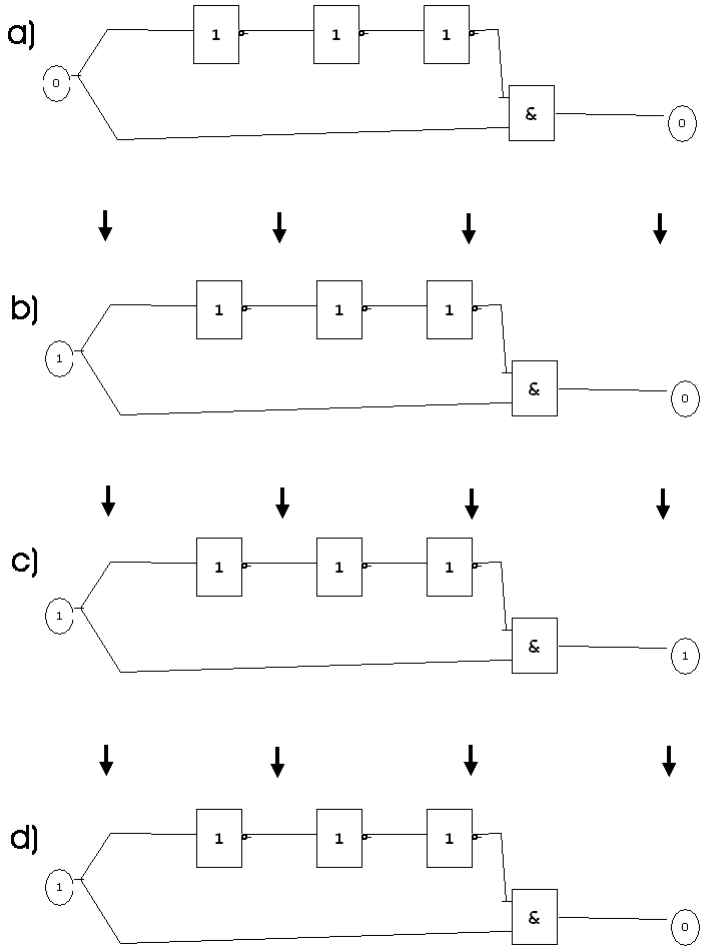

Abbildung 3.12: Hazard: a) Der Ausgangszustand: die Schaltung wurde mit einer 0 simuliert b) Die Eingabe wurde von 0 auf 1 gesetzt und die Simulation mit den neuen Werten fortgesetzt. c) Der Hazard tritt auf d) Das Endergebnis

nutzer des Programms die Fortpflanzung eines Signales durch die Schaltung beobachten (siehe Abb. 3.13).

Eine ganz andere Möglichkeit die Schaltungssimulation darzustellen ist die flüssige Animation. Die Animationsart ist dabei zwangsläufig „fliegende Einsen“. Dies visualisiert einen Fluss der Daten, wie er in der Wirklichkeit vorkommt. Eine Spannung ist nicht im gleichen Augenblick, in den Sie an eine Leitung angelegt wird auch an dem anderen Ende der Leitung. Diese Eigenschaft kann mit der flüssigen Animation gut dargestellt werden, obwohl sie in der Simulation nicht berücksichtigt wird.

3.4.2 Umsetzung

Die Spezifikation der Animation erfolgt in der Sichtspezifikation. Weil in DEViL die visuelle Darstellung vollständig in einer Sicht-Spezifikation definiert wird, sollte in dieser Sicht auch die Animation spezifiziert werden. Alle Konzepte, die versuchen würden, die Animation außerhalb der Sichtdefinition zu spezifizieren, verletzen das Konzept der Sichten. DEViL stellt die notwendigen Konstrukte bereit, um Daten zu visualisieren, die nicht in der abstrakten Struktur vorkommen. Die Werte der Ein- und Ausgänge der Schaltung sind nur während der Laufzeit vorhanden und nicht in der abstrakten Struktur.

Die zu animierenden Daten werden mit der Simulation mittels eines Definitionsmoduls ausgetauscht. Dieses Definitionsmodul ermöglicht es, Daten mit bestimmten Schlüsseln zwischen den verschiedenen Sichten oder einer Sicht und dem Simulator auszutauschen (siehe Abb. 3.15). Damit bildet es die Grundlage für die Animation in DEViL. Die Schlüssel bestehen aus einer Referenz auf einen Baumknoten und einem beliebigen Schlüsselwort. Es besitzt die gleiche Schnittstelle wie das vlProperty-Modul in DEViL.

Die Simulation schreibt die Ergebnisse des Simulationslaufes in das Modul und ruft eine Funktion auf, die die Sicht aktualisiert. Während der Attributauswertung der Sicht werden die Daten aus dem Modul ausgelesen und auf dem Bildschirm unter Verwendung von Mustern präsentiert. Um alle Daten richtig zuordnen zu können, sind im Modell der Simulation eindeutige Identifikationsnummern gespeichert, die vom Modell des Editors erstellt werden. Über diese Identifikationsnummern kann sowohl die Sichtspezifikation als auch die Simulation das gleiche Objekt eindeutig identifizieren, obwohl zwei unterschiedliche Modelle benutzt werden.

Die flüssige Animation verwendet eine eigens entwickelte Funktion, um eine flüssige Bewegung zu erzeugen. Um eine Bewegung an einer Leitung durchzuführen, wird unter Verwendung des Connection Musters und einigen Attributberechnungen der Weg des zu animierenden Objekts berechnet. Die Datenübernahme stammt wie bei der nicht flüssigen Animation aus dem Definitionsmodul.

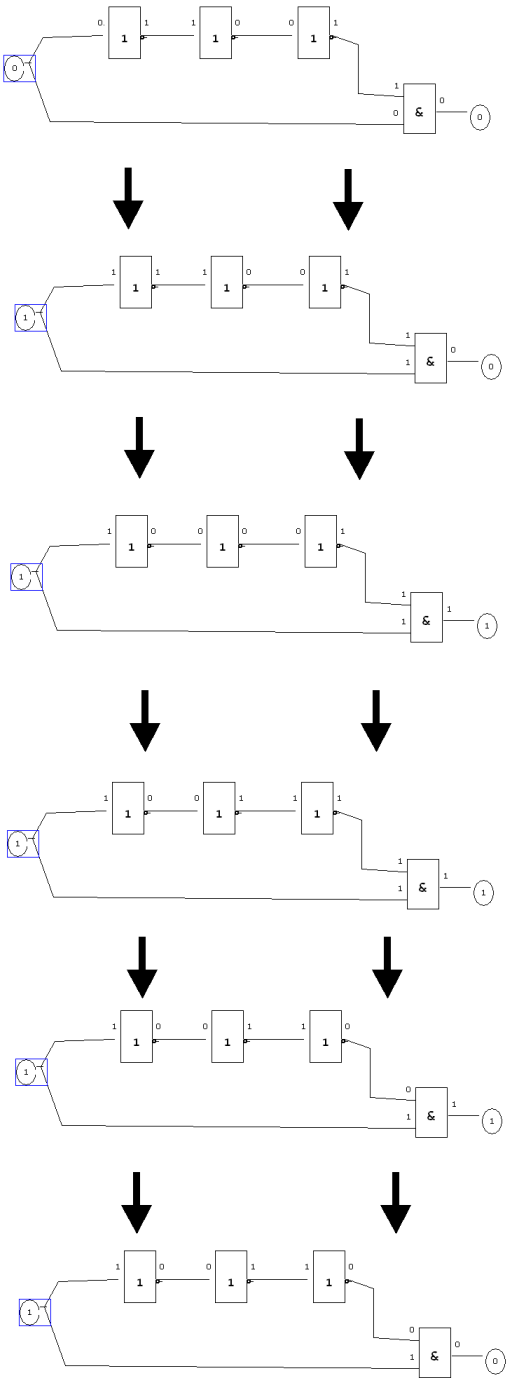

Abbildung 3.13: Hazard mit Anzeige der Zwischenergebnisse. Vorgehen wie in Abb. 3.12

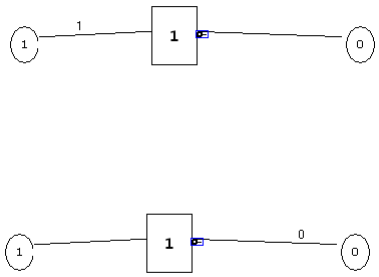

Abbildung 3.14: Bei der flüssigen Animation bewegen sich die Werte über den Leitungen

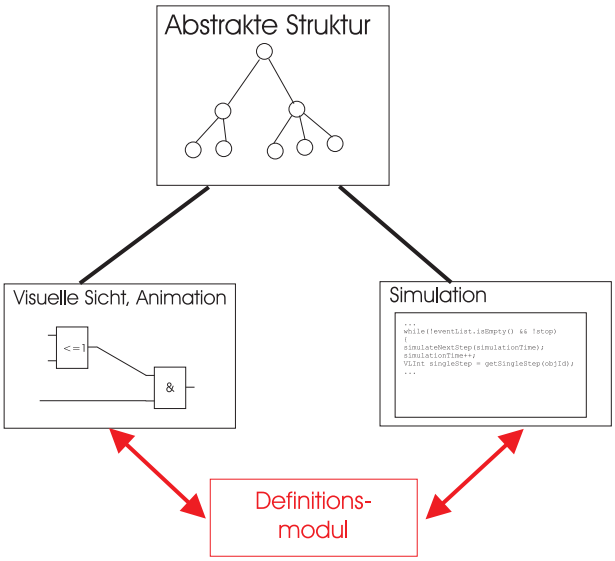

Abbildung 3.15: Datenaustausch Modul

4 Ergebnisse

Zwei Ergebnisse hat diese Arbeit hervorgebracht. Zum einen ein Programm zur Simulation und Animation von Logikbausteinen und zum anderen die Animation in DEViL. Im folgenden sollen diese beiden Ergebnisse vorgestellt werden.

4.1 Das Programm

Das Programm, das in dieser Arbeit entwickelt wurde, ist im Augenblick eher ein Prototyp. Die in 3.1 beschriebene Zielsetzung des Programms im Lehreinsatz ist zwar denkbar, allerdings müsste zu diesem Zweck das Programm verfeinert und die Benutzerführung an einigen Stellen verbessert werden. Die Funktionalität reicht allerdings jetzt schon aus, um kleinere Schaltungen vorzuführen.

Um DEViL auf die Möglichkeit der Generierung von Animationsumgebungen zu testen, war dieses Programm recht gut geeignet. Die Probleme auf dem Weg von einer visuellen Sprache bis hin zu einer visuellen Sprache, die animierbar ist, konnten erkannt werden.

4.2 Animation in DEViL

Ziel dieser Arbeit war es, DEViL auf die Möglichkeit zur Generierung von Animationumgebungen zu untersuchen. Mit dem Programm zur Animation und Simulation von Logikbausteinen ist es gelungen, mit DEViL ein Programm zu generieren, das die Animation von Logikbausteinen ermöglicht. Diese Tatsache heißt allerdings noch nicht, dass DEViL bestens dafür geeignet ist. An dieser Stelle sollen verschiedene Funktionen vorgestellt werden, die zur Realisierung von Animation in DEViL integriert werdem sollten oder bereits während dieser Studienarbeit realisiert wurden.

Das Definitionsmodul Um eine Animation von Daten vorzunehmen, die nicht innerhalb des Attributauswerters der Sicht erzeugt werden, müssen die Daten zu dieser Sicht gelangen. Liegt der Animation eine Simulation zugrunde, so ist es sinnvoll, die Simulation nicht über den Attributauswerter der Sicht zu starten. Um eine Animation zu gewährleisten, muss demnach eine Schnittstelle zum Datenaustausch zwischen dem Simulationsmodul und der Sicht-Berechnung existieren. Diese Weiterleitung sollte darüber hinaus asynchron verlaufen, da die Animation unabhängig vom Erzeugungszeitpunkt der Daten sein sollte. Weil in DEViL eine solche Möglichkeit noch nicht existierte, wurde DEViL um ein globales Definitionsmodul erweitert, auf

dessen Daten alle Attributauswerter zugreifen können. Mit dem Definitionsmodul ist es nun möglich, Daten zwischen den verschiedenen Attributauswertern auszutauschen. Dies bildet die Grundlage für die Animation.

Nicht persistente Daten Daten einer Animation, vor allem solche, die während einer Simulation entstanden sind, sind nicht Teil der abstrakten Struktur. Die abstrakte Struktur repräsentiert nur das visuelle Programm. Dies wird über die Laufzeit des Programms hinweg, d.h. persistent, gespeichert. Für die Animation werden meist Daten verwendet, die flüchtig sind, d.h. nach der Laufzeit des Programms gelöscht werden. DEViL stellt zurzeit keine geeigneten Mechanismen zur Verfügung, nicht persistente Daten zu editieren. Dies war für visuelle Sprachen auch nicht notwendig, da für visuelle Programme nur persistente Daten benötigt werden. Eine Möglichkeit diese Daten zu speichern, wäre sie an die abstrakte Struktur zu hängen. Dafür müsste in der Modelldefinition ein Konstrukt geschaffen werden, mit dem man definiert, dass ein Teilbaum nicht persistente Daten enthält. Diese Daten würden bei der Speicherung der abstrakten Struktur in eine Datei nicht berücksichtigt. Beim Öffnen der Datei würden diese Teilbäume initialisiert. Für die nicht persistenten Daten könnte wie für die persistenten Daten ein Dialog zum Ändern der Attribute generiert werden. Der Umgang mit nicht persistenten Daten würde damit sehr viel einfacher.

Flüssige Animation Eine flüssige Animation kann bestimmte Sachverhalte gut darstellen, die ansonsten nur schwer nachvollziehbar wären. Im Augenblick lassen sich in DEViL Objekte flüssig bewegen. Allerdings ist diese Funktion noch nicht ganz ausgereift. Es fehlen Konfigurationsmöglichkeiten, um z.B. die Animationsgeschwindigkeit einzustellen.

Für die Zukunft wäre es interessant, weitere Formen der flüssigen Animation in DEViL zu implementieren, z.B. die Verformung eines Objekts in ein anderes. Dafür müssten bestimmte Standard-Animationen herausgearbeitet werden, die für viele Animationsumgebungen interessant sind. Weiterhin könnten die visuellen Muster um Animationsunterstüzung ergänzt werden. Es wäre z.B. möglich, das Connection-Muster so zu erweitern, dass es den Weg für ein zu animierendes Objekt selber ausrechnet, um eine flüssige Bewegung an einer Linie zu erzeugen. Unter Umständen könnten auch neue Muster entwickelt werden, die mit den vorhandenen Mustern zusammenarbeiten. Mit einer solchen Weiterentwicklung von DEViL wäre es problemlos möglich, innerhalb von kürzester Zeit eine Animationsumgebung zu generieren. Man würde sich dabei nicht nur Zeit sondern auch die Aneignung von Expertenwissen sparen, welches notwendig ist, um eine solche Umgebung zu erstellen.

Generell läßt sich sagen, dass sich durch die Umsetzung der oben genannten Erweiterungen Animation in DEViL gut unterstützen ließe.

Literaturverzeichnis

[1] DIN 60617-12 Grafische Symbole für Schaltpläne, 4 1999.

[2] Christian Schindler Carsten Schmidt. Muster-basierte Generierung von Struktur-Editoren für visuelle Sprachen. Master's thesis, Universität Paderborn, 1 2000.

[3] Prof. Dr. Gerhard Goos. *Vorlesung über Informatik*. Springer Verlag Berlin Heidelberg, 3. edition, 2000. 3-540-67270-2.

[4] Hans-Jochen Schneider (Hrsg.). *Lexikon der Informatik und Datenverarbeitung*. Oldenbourg Verlag GmbH, München, 3. edition, 1991. 3-486-21514-0.

[5] Mario Jeckle, Chris Rupp, Jürgen Hahn, Barbara Zengler, and Stefan Queins. *UML 2 glasklar*. Carl Hanser Verlag München Wien, 2004. 3-446-22575-7.

[6] Prof. Dr. Uwe Kastens. *Übersetzerbau*. Oldenbourg Verlag GmbH, München, 1990. 3-486-20780-6.

[7] Haybatolah Khazar, B.Dittus, Bernhard Kick, W. Roesner, H.Spiro, J.Tatje, and D.Tavangarin. *Simulation und Synthese logischer Schaltungen*. expert Verlag, 1991. 3-8169-0542-0.

[8] Hans Thomas Krodel. *Verfahren zur Logiksimulation komplexer digitaler Schaltungen mit flexibler Modellierung*. PhD thesis, Technische Universität München, 6 1986.

[9] Andreas Mäder. *VHDL Kompakt*. Universität Hamburg.

[10] Stefan Schiffer. *Visuelle Programmierung*. Addison Wesley Longman Verlag GmbH, 1998. 3-8273-1271-X.

[11] Carsten Schmidt. *DEViL Benutzerhandbuch*. Universität Paderborn.

[12] Manfred Seifart. *Digitale Schaltungen*. Verlag Technik Berlin, 5. edition, 1998. 3-341-01198-6.

[13] Universität Paderborn. *Implementation of visual languages using pattern-based specifications*.