Objektorientierter Simulator für Logikschaltungen

Rolf Schlagenhaft

Diplomarbeit

Lehrstuhl für Rechnergestütztes Entwerfen
Technische Universität München
Prof. Dr.-Ing. K. Antreich

Ausgabe des Themas:	3. August 1992
Tag der mündlichen Prüfung:	16. Februar 1993
Betreuer:	Dipl.-Ing. Herbert Bauer

Rolf Schlagenhaft
Lerchenfeldstraße 31
8000 München 22

München, den 16. Februar 1993

Kurzbeschreibung

Die vorliegende Diplomarbeit stellt eine objektorientierte Klassenbibliothek für ereignisgetriebene Logiksimulation zur Verfügung. Besonderes Augenmerk gilt dabei der Möglichkeit, zukünftige Erweiterungen problemlos implementieren zu können. Dazu zählen zum Beispiel neue Signalwerte oder neue Schaltungsprimitive.

Die Vorgehensweise gliedert sich in zwei Schritte. Zuerst wird ein allgemeines Konzept für ereignisgetriebene Schaltungssimulation entwickelt. Dies beinhaltet die Darstellung der Schaltungstopologie und die Ereignisverwaltung.

Erst im zweiten Schritt kommen die Besonderheiten der Logiksimulation zum Tragen. Dabei wird die Funktionalität des bereits vorhandenen, prozeduralen Simulators LDSIM (*L*ogic *D*esign *SIM*ulator) nachgebildet. Dies gilt als Nachweis der Funktionstüchtigkeit der Klassenbibliothek.

Durch den objektorientierten Ansatz ergibt sich die flexiblere Verwendbarkeit der Bibliothek im Vergleich zum listenorientierten LDSIM. Dies wird unter anderem durch die Einführung eines neuen Signaltyps (Bussignal) deutlich.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

Der Umfang heutiger Softwareprodukte wächst in gleichem Maße, wie die Leistungsfähigkeit neuer Rechnergenerationen. Diese besitzen mehr Speicherkapazität und arbeiten schneller als ihre Vorgänger.

Dadurch ergeben sich bei der Softwareentwicklung neue Problemstellungen. Wurden früher Programme oft von Einzelpersonen oder kleinen Mitarbeitergruppen entwickelt und betreut, so sind heute immer mehr Menschen daran beteiligt. Um Fehler zu vermeiden sind deshalb saubere Programmschnittstellen notwendig. Des weiteren entsteht aus dem größeren Umfang der Projekte die Anforderung, daß der entstandene Quellcode eine möglichst hohe „Lebensdauer“ besitzt. Es ist nicht mehr vertretbar, daß bereits nach wenigen Jahren ein komplettes Programm-Redesign nötig wird.

Aus diesen Gründen hat sich in den letzten Jahren eine neue Programmiertechnik herausgebildet, die sogenannte objektorientierte Vorgehensweise. Durch ihre Anwendung werden die Anforderungen für umfangreiche Programme im allgemeinen besser erfüllt als durch bisherige Techniken. Ein Umstieg auf objektorientierte Programmierung (OOP) bedeutet allerdings einen einmaligen Mehraufwand an Programmierarbeit. Außerdem ist oft eine neue Sichtweise für das zu lösende Problem nötig. Das sind die Gründe, warum sich der neue Stil noch nicht auf breiter Front durchgesetzt hat.

Lohnend ist der Umstieg zu einem Zeitpunkt, an dem größere Änderungen durchzuführen sind oder ein Redesign nötig ist, da Veränderungen nicht mehr unter vertretbarem Aufwand möglich sind. Beim Logiksimulator LDSIM ist dieser Punkt erreicht. Deshalb wird in dieser Diplomarbeit der Umstieg auf OOP vollzogen. Die objektorientierte Realisierung des Simulators wird außerdem so durchgeführt, daß sich seine Bestandteile auch als Basis für andere Simulationsaufgaben verwenden lassen.

Kapitel 2

Objektorientierte Softwareentwicklung

2.1 Begriffe

In den folgenden Abschnitten werden einige Begriffe der objektorientierten Softwareentwicklung erklärt, die in den weiteren Kapiteln öfter verwendet werden.

Erst durch die Verinnerlichung der neuen Begriffswelt ist es möglich, den alten Programmierstil abzulegen und die neuen Konzepte voll auszunutzen. Eine Vermischung von prozeduraler und objektorientierter Vorgehensweise ist nach Möglichkeit zu vermeiden, da dadurch ein Projekt sehr unübersichtlich wird. Hiervon ausgenommen sind natürlich Standard-Bibliotheksroutinen der verschiedenen Programmiersprachen.

2.1.1 Objekt

In Anlehnung an die Wirklichkeit besteht ein OOP-Programm aus Einzelteilen mit bestimmten Eigenschaften und Fähigkeiten. Diese Teile werden als Objekte bezeichnet.

Die Eigenschaften werden oft auch Attribute genannt und stellen den Datenaspekt eines Objekts dar. Die Fähigkeiten heißen Methoden. Sie repräsentieren den funktionalen Aspekt der Objekte und bilden den eigentlichen „Programmtext“, der zur Laufzeit zur Ausführung kommt.

Die Besonderheit der Objekte besteht nun darin, daß sie zusammengehörige Attribute

und Methoden untrennbar miteinander verknüpfen. Dies ist nicht als Einschränkung anzusehen, sondern als Möglichkeit, Problemstellungen realitätsnäher darzustellen.

Die beiden Abbildungen 2.1 und 2.2 verdeutlichen die Unterschiede zwischen prozeduraler Vorgehensweise und OOP.

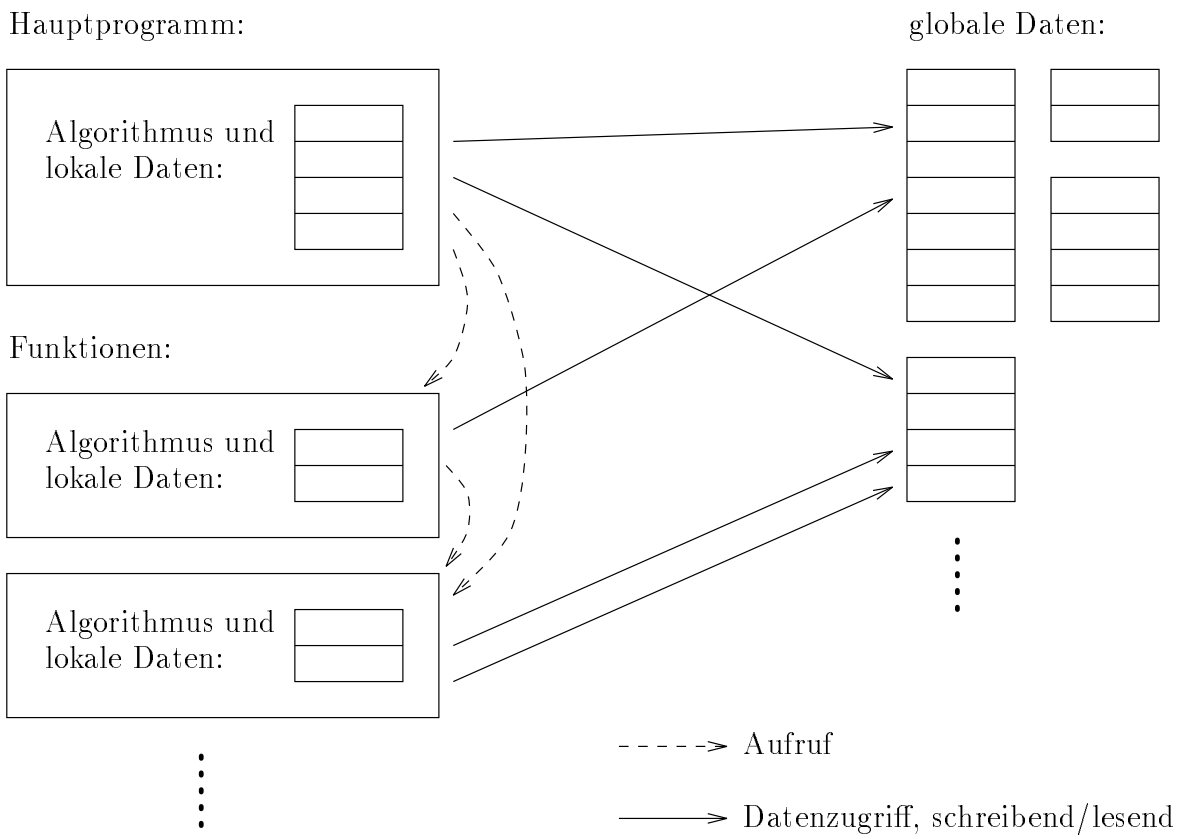


Abbildung 2.1: Prozeduraler Stil

Der Buchtitel „Algorithms + Data Structures = Programs“ von Niklaus Wirth [11] spiegelt im Prinzip genau die alte Vorgehensweise wider. Ein Programm besteht demnach aus Datenstrukturen (meist global) und Funktionen (vgl. Abb. 2.1). Diese rufen sich gegenseitig auf und verändern gemäß ihrer Aufgabe die Daten. Auf globale Daten haben prinzipiell alle Funktionen Zugriff, wodurch deren Konsistenz gefährdet ist. In den Funktionen können auch lokal Daten definiert werden, die allerdings in den meisten Anwendungsfällen nur temporär sind. Das heißt, sie existieren zur Programmlaufzeit nur während der Ausführung einer Funktion und werden selten zur Darstellung von

relevanten Informationen verwendet.

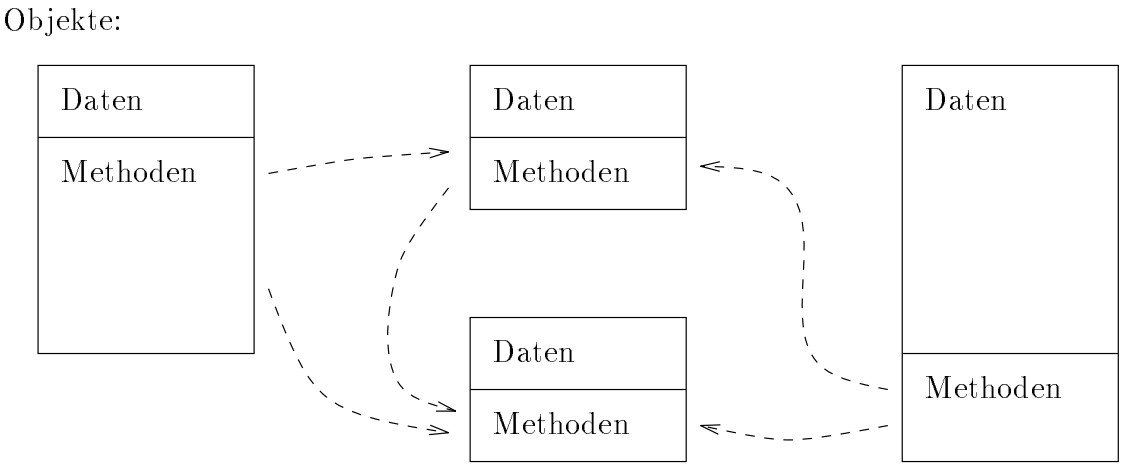


Abbildung 2.2: Objektorientierter Stil

Bei der objektorientierten Realisierung einer Problemstellung (Abb. 2.2) existieren zur Laufzeit eines Programms idealerweise nur die oben vorgestellten Objekte. Sie vereinigen die Darstellung von Informationen (Daten) und deren Funktionalität (Methoden). Der Programmablauf besteht in gegenseitigen Methodenaufrufen. Diese verändern dann ihre zugehörigen Daten. Es ist auch die direkte Modifikation von Daten eines fremden Objekts möglich, wenn dieses es explizit erlaubt. Von dieser Möglichkeit sollte aber nur dann Gebrauch gemacht werden, wenn dadurch keine Konsistenzprobleme entstehen.

2.1.2 Datenkapselung

Ein großer Vorteil der OOP ist die Möglichkeit, Information in Objekten in beliebiger Form darzustellen und nach außen unzugänglich zu machen. Dies bezeichnet man als Datenkapselung. Veränderungen können nur mittels Methodenaufrufen erwirkt werden. Sind diese Methoden sauber implementiert, ist die Konsistenz der Daten immer gewährleistet. Dieses Verfahren erhöht die Sicherheit des erzeugten Programms.

Ein weiterer Vorteil ist die freie Wahl der Implementation der Teilprobleme eines Projekts. Sie kann sogar später unabhängig vom restlichen Programmcode wieder verändert werden, da man sich darauf verlassen kann, daß sämtliche Kommunikation mit dem entsprechenden Objekt nur über die definierten Schnittstellen (Methoden) stattfindet.

Mittels des unter Abschnitt 2.1.4 vorgestellten Vererbungsprinzips ist es sogar möglich, sich erst zur Laufzeit des Programms für eine der verschiedenen Implementierungen zu entscheiden.

2.1.3 Klasse

Bei einer Klasse handelt es sich um eine Typdefinition. Instanzen dieses Typs sind die unter Abschnitt 2.1.1 eingeführten Objekte. Sie werden erst zur Laufzeit eines Programms erzeugt. Alle Objekte eines Typs besitzen die gleichen Methoden, während sie sich im Inhalt ihrer Daten unterscheiden können.

Die wichtigste Aufgabe von Klassendefinitionen ist die Festlegung der Fähigkeiten einer Klasse. Dies geschieht durch die Spezifikation ihrer Methoden.

Um eine Klasse instanziieren zu können, sind weitere Schritte nötig. Einerseits muß die Art der internen Informationsdarstellung festgelegt werden (Klassenvariablen). Andererseits ist die Implementation der angebotenen Methoden notwendig.

Klassen, bei denen dies nicht geschieht, werden abstrakt genannt und können nicht instanziiert werden. Die Aufgabe solcher abstrakter Klassen besteht nur in der allgemeinen Festlegung ihres Verhaltens. Auf ihnen aufbauend sind weitere Klassendefinitionen möglich (siehe Abschnitt 2.1.4).

2.1.4 Vererbung

Diese wohl mächtigste Eigenschaft der OOP stellt den Hauptunterschied zum prozeduralen Stil dar. Aus ihr erwachsen eine ganze Reihe neuer Möglichkeiten.

Vererbung heißt, daß neu erzeugte Klassen auf bereits bestehenden aufbauen können, ohne daß diese verändert werden müssen. Beide Versionen können nebeneinander im gleichen Programm existieren.

Die auf diese Art und Weise neu erzeugten Klassen „erben“ zunächst bestimmte Eigenschaften und Methoden ihrer „Vorfahr“-Klassen, der sogenannten Basisklassen. Dabei kann es sich um eine oder mehrere Basisklassen handeln (Mehrfachvererbung). Darüberhinaus ist es möglich, neue Fähigkeiten zu implementieren.

Aus abstrakten Klassen lassen sich nur mittels Vererbung Klassen erzeugen, die instanziierbar sind. Die in der abstrakten Basisklasse definierten Methoden werden erst in der

abgeleiteten Klasse implementiert.

Mit diesem Mechanismus ist eine Klassenhierarchie möglich. Aufbauend auf relativ allgemein und einfach gehaltenen Basisklassen können in mehreren Stufen neue Klassen entwickelt werden. Diese erfüllen immer speziellere Aufgaben. Durch Vererbung entlang verschiedener „Linien“ ist auch eine Anpassung der allgemeinen Basisklassen an verschiedene Problemstellungen möglich.

2.1.5 Überladen / Polymorphie

Bis jetzt wurde nur die Möglichkeit erwähnt, die noch nicht implementierten Methoden einer abstrakten Basisklasse in abgeleiteten Klassen verschieden zu implementieren. Man kann aber sogar noch einen Schritt weiter gehen und bereits in der Basisklasse implementierte Methoden in abgeleiteten Klassen neu realisieren (=Überladen). Das ist möglich, wenn die ursprüngliche Methode als virtuell definiert ist oder eine andere Parameterliste als die neue Methode besitzt. Im ersten Fall stellt die alte Methode dann gewissermaßen nur einen Prototyp dar.

Nach außen zeigen alle Objekte einer Klassenhierarchie gemeinsame, in der Basisklasse festgelegte Eigenschaften. Bei einem Methodenaufruf wird aber automatisch die entsprechende Realisierung ausgewählt und jedes Objekt führt seine eigenen Aktionen durch. Diese Eigenschaft bezeichnet man als Polymorphie.

Das Besondere an den vorgestellten Möglichkeiten ist die Tatsache, daß die verschiedenen Methodenrealisierungen nebeneinander existieren können. Dadurch ist es nicht notwendig, alte Programmteile zu verwerfen, wenn Veränderungen einzubauen sind. Dies trägt auch zur Wiederverwendbarkeit von Programmcode bei. Software läßt sich so relativ dynamisch entwickeln und erweitern.

2.1.6 Beispiel

Die vorgestellten Begriffe lassen sich anhand eines Beispiels (Abb. 2.3 und 2.4) aus der Logiksimulation nochmal verdeutlichen. Um es einfach und überschaubar zu halten, entsprechen die nun vorgestellten Klassen und Methoden allerdings nicht der tatsächlichen Realisierung im neuen Simulator.

Bei der Klasse `G2` (Abb. 2.3) handelt es sich um eine abstrakte Basisklasse. Sie stellt ein logisches Gatter mit zwei Eingängen und einem Ausgang dar. Die Variablen `i1,i2` und `o` des Datenteils von `G2` stellen die entsprechenden Signalwerte dar.

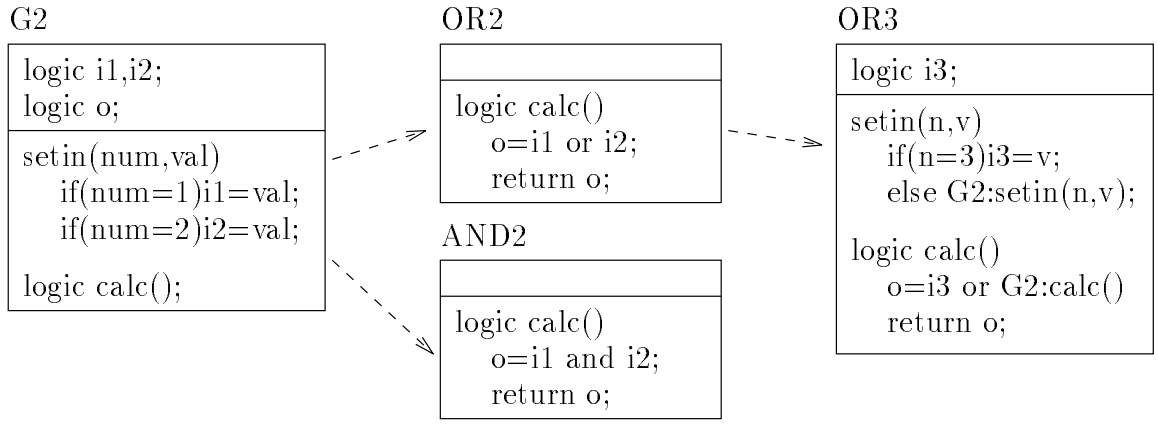


Abbildung 2.3: Klassenhierarchie-Beispiel

Mit der Methode `setin` lassen sich die Eingangssignale mit Werten belegen. Da diese Routine für alle verschiedenen Gatter gleich ist, läßt sie sich bereits hier implementieren.

Alle Gattertypen müssen eine Methode zum Berechnen des Ausgangssignals besitzen. Da aber jedes Gatter eine andere Berechnungsvorschrift hat, wird sie hier nur als virtuelle Methode `calc` definiert. Deshalb handelt es sich bei `G2` um eine nicht instanziierbare, abstrakte Klasse. Es läßt sich kein Objekt dieses Typs `G2` erzeugen.

In den beiden Nachfahren `OR2` und `AND2` wird die Methode `calc` jeweils implementiert. Dadurch entstehen instanziierbare Klassen. Sie erben von `G2` die Methode `setin` und die Variablen `i1,i2` und `o`.

Soll nun ein Oder-Gatter mit drei Eingängen realisiert werden, reichen die Variablen von `OR2` nicht mehr aus. Außerdem ist mit der Routine `setin` nur das Besetzen von zwei Eingangsvariablen möglich. Die Klasse `OR3` muß also eine neue Variable `i3` besitzen und die Methoden `setin` und `calc` von `OR2` überladen.

Die neue Methode `setin` wird dabei unter Wiederverwendung des alten Codes realisiert. Nur die Besetzung der dritten Eingangsvariablen `i3` muß neu programmiert werden. Ähnlich verhält es sich bei der neuen Methode `calc` der Klasse `OR3`. Sie verwendet die Methode `calc` des Vorfahren `OR2`.

In der Abbildung 2.4 ist ein möglicher Programmausschnitt dargestellt, der die eben vorgestellten Klassen verwendet . Es wird vorausgesetzt, daß `HIGH` und `LOW` Variablen vom Typ `logic` sind. In Zeile eins wird ein Feld von Zeigern auf `G2`-Objekte angelegt.

```
1: G2 *g[2];
2: AND2 and;
3: OR3 or;
4:
5: and.setin(1,HIGH);
6: and.setin(2,LOW);
7: or.setin(1,HIGH);
8: or.setin(2,LOW);
9: or.setin(3,LOW);
10:
11: g[0]=&and;
12: g[1]=&or;
13:
14: g[0]→calc();
15: g[1]→calc();
```

Abbildung 2.4: Programm-Beispiel

In den beiden folgenden Zeilen werden zwei Gatter-Objekte erzeugt. Eines ist vom Typ `AND2` und besitzt zwei Eingänge. Das andere ist vom Typ `OR3` mit drei Eingängen.

In den Zeilen fünf bis neun werden die Eingänge mit Werten belegt. Beim And-Gatter kommt die von `G2` geerbte Methode `setin` zur Ausführung, während bei Or-Gatter automatisch die in `OR3` implementierte Methode verwendet wird.

In den Zeilen elf und zwölf wird das Feld `g` mit Zeigern auf die erzeugten Objekte belegt. Die letzten beiden Anweisungen schließlich starten die Berechnung der Ausgangssignale. Dabei ist aus den Aufrufen nicht ersichtlich, um welchen Gattertyp es sich in den beiden Fällen handelt. Trotzdem wird die jeweils richtige Methode `calc` aufgerufen. In Zeile 14 wird der Signalwert am Ausgang des `AND2`-Gatters berechnet, in Zeile 15 der des `OR3`-Gatters. Man spricht hier von später Bindung, da erst zur Laufzeit des Programms feststeht, welche Routine auszuführen ist.

2.2 Vorgehensweise

Die Durchführung eines objektorientierten Projekts gliedert sich in mehrere Schritte. Im ersten Schritt werden die Leistungfähigkeit des geplanten Vorhabens und die zur Problemlösung nötigen Voraussetzungen festgelegt.

In der zweiten Stufe gilt es, die Problemstellung genauer zu analysieren. Hier ist es eventuell schon möglich, eine Auftrennung in Teilaufgaben vorzunehmen.

Es folgt die Abbildung von verschiedenen, möglichst unabhängigen Teilaspekten des Projekts auf Klassen. Dabei ist darauf zu achten, diese weitgehend zu entkoppeln. Dadurch hält sich später der Kommunikationsaufwand zwischen den Objekten bei der Problemlösung in Grenzen. Zusammengehörige Daten werden in der gleichen Klasse untergebracht.

Nun wird festgelegt, welche Methoden die Klassen zur Verfügung stellen müssen, um damit die Lösung des Problems zu ermöglichen. Dabei festgestellte Ähnlichkeiten oder Gemeinsamkeiten sind auf gemeinsame Basisklassen abzubilden.

Durch schrittweise Abstraktion entsteht nach und nach eine Klassenhierarchie. Sind alle Aspekte der Problemstellung durch Objekte dargestellt, kann mit der Implementierung der Methoden und des Hauptprogramms begonnen werden.

Da die Objekte relativ abgeschlossene, unabhängige Einheiten darstellen, lassen sie sich meist auch sehr einfach einzeln testen, um ihre Funktion zu überprüfen.

Stellen sich neue Anforderungen ein, sind diese mittels Vererbung nach Möglichkeit in die bestehende Klassenhierarchie einzubauen.

Kapitel 3

Simulator-Grundprinzip

3.1 Darstellung der Schaltungstopologie

Eine elektrische Schaltung wird allgemein dargestellt durch Elemente, die durch Signalnetze untereinander verbunden sind. Die Funktionen der Elemente können je nach Hierarchiestufe und Schaltungsart stark variieren (Abb. 3.1). Analoge Bauteile zeichnen sich durch anderes Verhalten aus als digitale. Kleine Schaltungen werden meist sehr detailliert dargestellt, während große Schaltungen oft nur mittels Funktionsblöcken wiedergegeben werden.

Für die Simulation ergeben sich somit auch verschiedene Ansätze, die sich in den verwendeten Modellen für Elemente und Signale unterscheiden.

Allen Schaltungsarten gemein ist allerdings die Möglichkeit, sie als Topologie darzustellen, bestehend aus einer Menge an Elementen E_μ und einer Menge an Signalen S_ν. Diese Tatsache wird ausgenutzt, um daraus ein allgemein verwendbares Simulatorkonzept aufzubauen.

Dazu wird die Schaltungstopologie abgebildet auf untereinander verknüpfte Objekte im Rechner. Für jedes funktionale Teil einer Schaltung (Bauteil, Gatter) existiert ein Objekt. Sie können eine beliebige Anzahl von Ein- und/oder Ausgängen besitzen. Die Schaltungsbestandteile sind in der Realität durch Signalnetze miteinander verbunden. Diese werden im Simulator dargestellt durch Objekte, auf die jeweils alle angeschlossenen Elemente Zugriff haben. Bei diesen Signalobjekten kann es sich je nach darzustellendem Signal um einfache Datentypen oder um Instanzen komplexerer Klassen mit eigenen Methoden handeln.

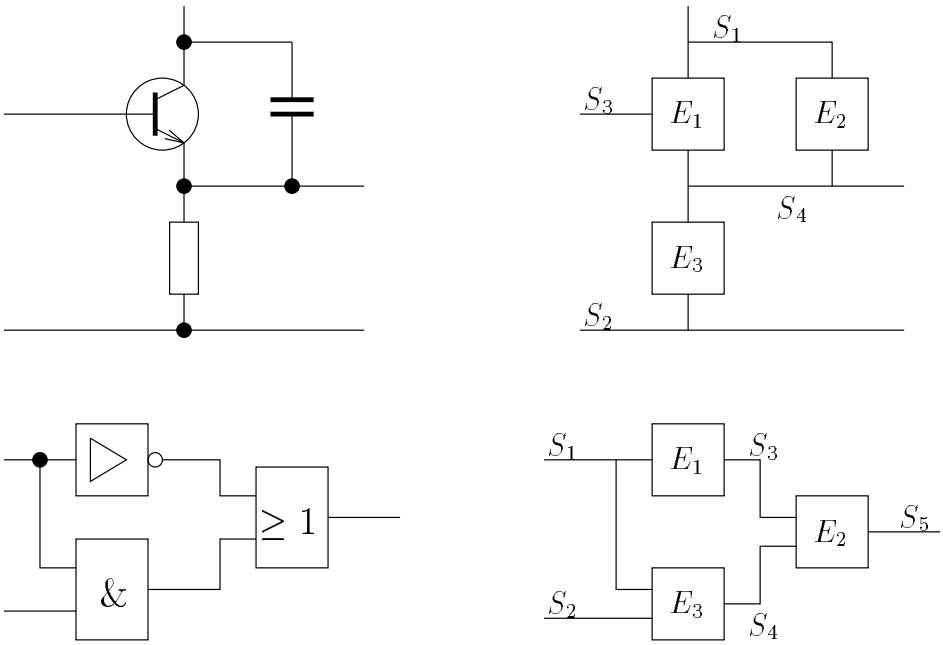


Abbildung 3.1: Elemente, Signale

Zunächst werden nur gerichtete Zweipunktverbindungen betrachtet (unidirektionale Signale). Mehrpunktverbindungen werden durch ein spezielles Objekt in der Schaltungstopologie realisiert, welches später vorgestellt wird. Ungerichtete Verbindungen (bidirektionale Signale) können bei Bedarf zum Beispiel durch zwei entgegengesetzt gerichtete Zweipunktverbindungen dargestellt werden.

Da sich bereits hier an allen vorstellbaren Elementen gemeinsame Eigenschaften erkennen lassen, wird die erste abstrakte Basisklasse `SimObj` eingeführt (vgl. A.3). Sie bildet die Basisklasse für alle Elemente, die sich in das Abbild der Schaltungstopologie einbauen lassen sollen. Ihre wichtigsten Datenelemente sind die Anzahl der Ein- und Ausgangssignale, die Zeiger auf die Speicherbereiche (Objekte), die deren Signalwerte beinhalten und Zeiger auf die Elemente, die mit den Ausgangssignalen verbunden sind.

Die Klasse `SimObj` definiert außerdem Methoden zum Verbindungsaufbau zwischen den Elementen. Da es möglich sein soll, die verschiedensten Signalarten darzustellen, ist es notwendig, daß während des Verbindungsaufbaus eine Überprüfung der Signalart stattfindet. An ein Element, das ein digitales Signal treibt, darf für die Simulation kein Element angeschlossen werden, welches ein analoges Signal erwartet. Standardmäßig geht `SimObj` von einem Digitalsignal nach Kapitel 4.1.1 aus. Sollen andere Signalarten

eingeführt werden, müssen die Methoden zum Verbindungsaufbau überladen werden (z.B. Kapitel 4.1.2 und 4.2.3).

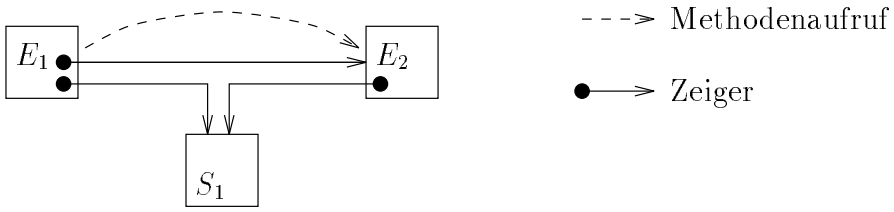


Abbildung 3.2: Gerichtete Zweipunktverbindung

Eine weitere sehr wichtige Methode ist die, mittels derer andere Topologieelemente dem Objekt mitteilen können, daß sich an einem der Eingangssignale eine Veränderung ergeben hat, die sogenannte Aktivierungsmethode. Hierbei handelt es sich um eine virtuelle Methode, da jedes Element anders auf solch eine Nachricht zu reagieren hat. Während der Simulation läuft eine Signalwertänderung folgendermaßen ab (Abb. 3.2): Element E_1 (Objekt) verändert den Signalwert S_1 (für E_1 und E_2 zugängliches Objekt). Anschließend teilt es dem Element E_2 durch den Aufruf der Aktivierungsmethode die Veränderung mit. Wie E_2 nun darauf reagiert, hängt von dessen Typ ab und interessiert E_1 nicht weiter.

Diese Art der Schaltungsabbildung läßt sich nicht nur zur Darstellung von Bauteilen eines klassischen Schaltplans verwenden. Vor allem in Kapitel 4 über die LDSIM-Nachbildung werden einige „Elemente“ eingeführt, die ausschließlich der Simulatorsteuerung dienen. Es ist also notwendig, den Begriff Element auszudehnen. Als Element wird im folgenden jedes Objekt bezeichnet, welches eine Instanz einer von SimObj abgeleiteten Klasse ist. Solche Objekte zeichnen sich dadurch aus, daß sie sich in die Schaltungstopologie einbauen lassen.

Das erste Beispiel hierfür ist die Klasse `SimDist`. Mit ihr lassen sich Mehrpunktverbindungen realisieren. Sie hat nur einen Eingang, der mit dem treibenden Element des Mehrpunktnetzes zu verbinden ist. Die Anzahl der Ausgänge ist parametrisiert. Sie sind mit allen weiteren an das Signalnetz angeschlossenen Elementen zu verbinden. Somit stellt `SimDist` einen Fanoutbaum dar.

Die Realisierung ist relativ einfach (Abb. 3.3). Die Zeiger auf das Eingangssignal und sämtliche Ausgangssignale von SimDist (E_2) referenzieren alle das gleiche Signalobjekt S_1. Somit stellt eine Eingangssignaländerung an SimDist automatisch die Veränderung dessen Ausgangssignale dar. Die Eingänge der Elemente E_3 und E_4 sind nun beide

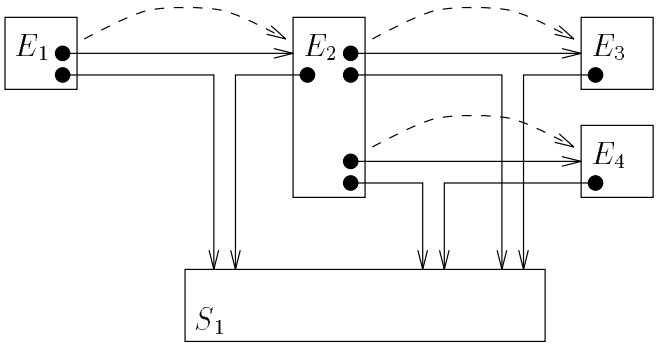


Abbildung 3.3: Mehrpunktverbindung mittels SimDist

indirekt mit dem Ausgang von Element E_1 verbunden. Die Aufgabe der Klasse `SimDist` besteht dann nur noch darin, die Mitteilung über eine Signalwertänderung an deren Eingang an alle angeschlossenen Elemente (hier E_3 und E_4) weiterzuleiten.

3.2 Ereignisgesteuerte Simulation

Den zweiten Schwerpunkt neben der Abbildung der Schaltungstopologie bildet die Darstellung der Zeit im Simulator. Prinzipiell sind hier zwei Vorgehensweisen möglich. Man kann die Schaltung zeitdiskret in festen Zeitschritten simulieren oder ereignisgesteuert.

Bei der ersten Vorgehensweise stellt sich das Problem, wie fein die zeitliche Auflösung gewählt werden soll. Werden relativ kleine Zeitschritte gewählt, wird sich der Simulator oft in Zeitbereichen aufhalten, in denen kaum eine Veränderung in der Schaltung stattfindet (Abb. 3.4). Dies führt zu einer unnötigen Verlängerung der Programmlaufzeit.

Wird dagegen ein große Schrittweite eingestellt, führt dies zu einem Genauigkeitsverlust in den Simulationsergebnissen (Abb. 3.5).

Die ereignisgetriebene Simulation vereint die Vorteile beider Methoden. Jegliche Veränderung in der Schaltung wird durch ein Ereignis dargestellt. Dieses zeichnet sich aus durch einen Zeitpunkt, an dem es stattfinden soll und durch eine Aktion, die dann auszuführen ist.

Zeitabschnitte, in denen wenige Signalwertänderungen stattfinden, bearbeitet der Simu-

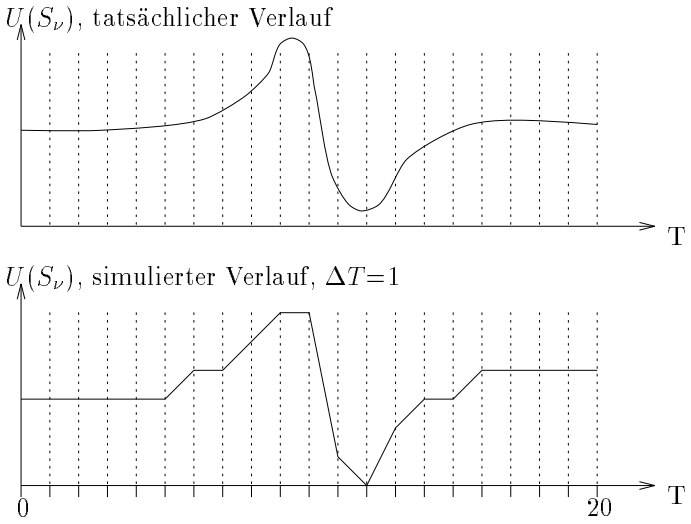


Abbildung 3.4: Zu kleine Zeitschritte

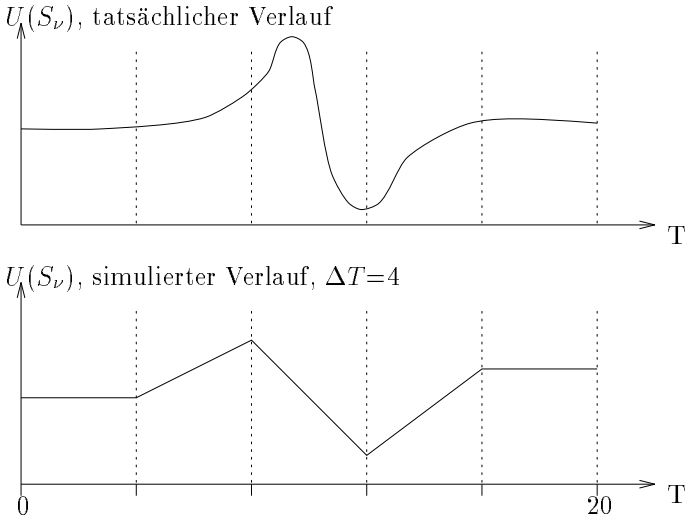


Abbildung 3.5: Zu große Zeitschritte

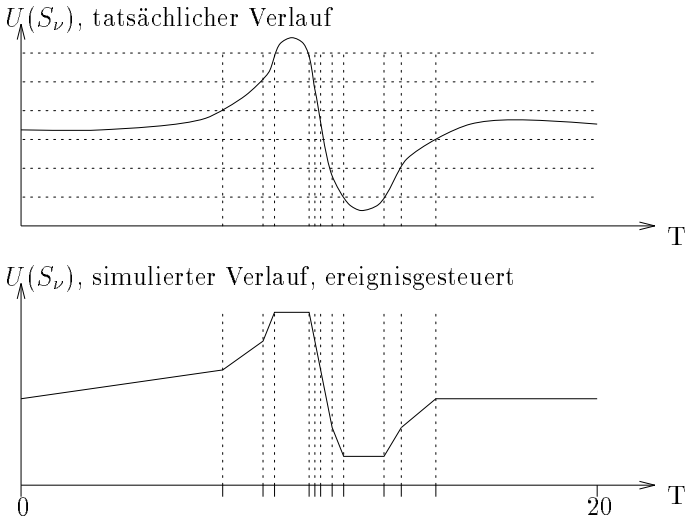


Abbildung 3.6: Zeiteinteilung bei Ereignissteuerung

lator sehr schnell, da wenige Ereignisse zur Bearbeitung anfallen. In Zeitbereichen mit hoher Schaltungsaktivität steigt die zeitliche Auflösung dagegen automatisch durch die Vielzahl der erzeugten Ereignisse an.

Der Nachteil dieser Vorgehensweise ist darin zu sehen, daß eine Ereignisverwaltung notwendig wird, deren Aufgabe es ist, die für zukünftige Zeitpunkte generierten Ereignisse zwischenzuspeichern und gemäß ihrer zeitlichen Reihenfolge zur Ausführung zu bringen.

Für die vorliegende Arbeit wird das Konzept der ereignisgesteuerten Simulation verwendet. Da die Basisklasse `SimEvent` für Ereignisse sehr allgemein gehalten ist, lassen sich aus ihr nicht nur Klassen zur Darstellung von Signalwertänderungen ableiten, sondern auch solche, die den Simulationsablauf selbst steuern (Abb. A.1 im Anhang). Jede Aktion, die während der Simulation zu einem bestimmten Zeitpunkt stattfinden soll, läßt sich in Form eines entsprechenden Ereignis-Objekts darstellen. Die verschiedenen Ereignistypen bilden eine eigene Klassenhierarchie. In der Basisklasse `SimEvent` sind unter anderem die Eigenschaften und Methoden festgelegt, die von der Ereignisverwaltung (Kap. 3.4) benötigt werden, um die Ereignisse unabhängig von ihrem Typ zeitlich zu sortieren und zu verwalten.

3.3 Simulationsablauf

Der Simulationsablauf läßt sich anhand von Abbildung 3.7 erläutern. Zu Beginn der Simulation wird die Schaltungsbeschreibung aus einer Datei eingelesen. Daraus erzeugt der Simulator die Elemente der Schaltung in Form von Objekten der entsprechenden Klassen. Zusätzlich werden Element-Objekte generiert, die zu Kontroll- und Steuerungzwecken mit in die Topologie eingebaut werden sollen.

Anschließend werden alle Objekte entsprechend der Schaltungsbeschreibung miteinander verbunden. Dabei generieren die signaltreibenden Elemente die Objekte, die diese Signale darstellen selbst. Das andere an der jeweiligen Zweipunktverbindung beteiligte Element überprüft dann seinerseits, ob es sich um eine erlaubte Verbindung handelt, das heißt, ob der gelieferte Signaltyp mit dem erwarteten übereinstimmt.

Der Simulatorkern selbst muß also keinerlei Information über den Typ der simulierten Signale besitzen.

Ist die Schaltung im Speicher des Rechners vollständig erzeugt, beginnt das Einlesen der Stimulidatei. Aus den darin enthaltenen Informationen werden die entsprechenden Ereignisse erzeugt und sofort der Ereignisverwaltung übergeben.

Nun beginnt die eigentliche Simulation. Der Simulatorkern holt sich dazu von der Ereignisverwaltung die Menge der zum nächsten Zeitpunkt auszuführenden Ereignisse. Diese können von völlig unterschiedlichem Typ sein. Vom Simulator wird nur jeweils die Methode zur Ereignisausführung aufgerufen. Welche Aktionen dadurch hervorgerufen werden, hängt von dem jeweiligen Ereignis ab. In den meisten Fällen wird es sich um Signalwertänderungen handeln. Solche Ereignis-Objekte stehen in Verbindung mit einem Element in der Schaltungstopologie. Dort wird dann die Veränderung am entsprechenden Signal vorgenommen und alle an dieses Signalnetz angeschlossenen Elemente davon in Kenntnis gesetzt (Aktivierungsmethode). Wie diese darauf reagieren, hängt wiederum von deren Typ ab. Sie können einerseits sofort weitere Signale verändern und die Nachfolgeelement aktivieren oder weitere Ereignisse erzeugen, die sie der Ereignisverwaltung zur Aufbewahrung übergeben.

Sind alle Ereignisse des ersten Zeitpunktes abgearbeitet, holt der Simulator die des nächsten Zeitpunktes von der Ereignisverwaltung ab und bringt sie zur Ausführung. Dies wird so lange iterativ wiederholt, bis keine weiteren Ereignisse anstehen oder der Simulator durch ein spezielles Steuerereignis vorher angehalten wird.

Die Simulationsergebnisse werden nicht vom Simulatorkern protokolliert, sondern von speziellen Elementen in der Schaltungstopologie. Dadurch ist es möglich, den Simulator

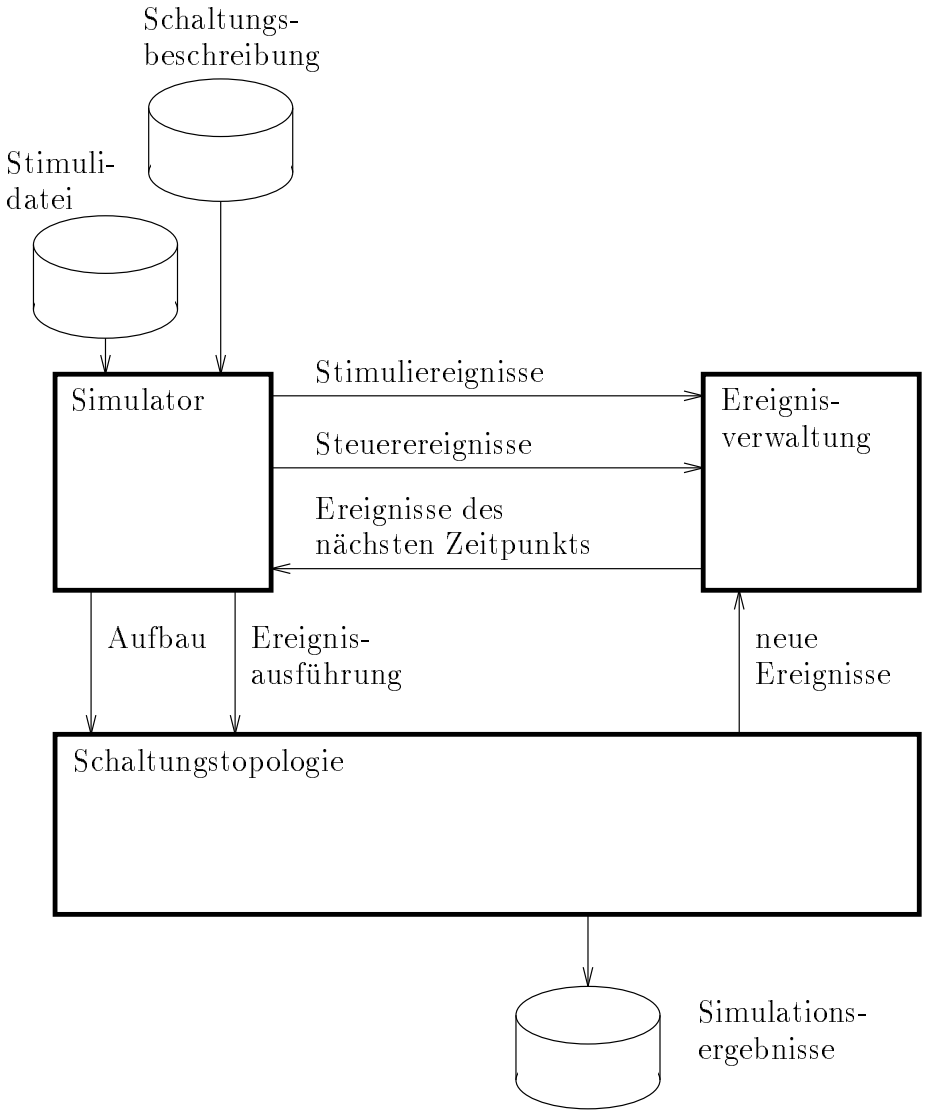


Abbildung 3.7: Zusammenspiel der Komponenten

frei zu halten von Information über die verwendeten Signaldarstellungen. Die einzige Aufgabe des Kerns in diesem Zusammenhang ist es, in der Protokolldatei Zeitmarker zu setzen.

3.4 Ereignisverwaltung

3.4.1 Allgemeine Anforderungen

Die Aufgabe der Ereignisverwaltung wurde bereits in Kapitel 3.2 angesprochen. Sie besteht darin, Ereignisse für zukünftige Simulationszeitpunkte zwischenzuspeichern. Prinzipiell gelten dabei folgende Besonderheiten:

- Nur die Ereignisse mit der jeweils kleinsten Ausführungszeit müssen von der Zeitverwaltung zurückgeliefert werden.
- Die Ausführungszeit neu generierter Ereignisse liegt meist zwischen der aktuellen Zeit und dem Ausführungszeitpunkt des am weitesten in der Zukunft liegenden Ereignisses.
- Oft sind Ereignisse gleicher oder nahe beieinander liegender Ausführungszeiten direkt nacheinander in die Verwaltung einzufügen.
- Ereignisse sind einzeln in die Verwaltung einzufügen, dagegen als Gruppe von Ereignissen mit gleicher Ausführungszeit abzuliefern.

Es handelt sich deshalb um einen Spezialfall einer Priority Queue.

Um hier Spielraum für verschiedene Implementationen zu haben wird zunächst nur eine abstrakte Basisklasse `SimTime` gebildet. Sie legt das grundsätzliche Methodenangebot einer Ereignisverwaltung fest. Die drei wichtigsten Methoden sind das Einfügen eines einzelnen neuen Ereignisses in die Verwaltung, das Zurückliefern der Menge Ereignisse mit kleinster Ausführungszeit und die Abfrage, ob weitere Ereignisse anstehen. Beim Aufnehmen eines neuen Ereignisses ist außerdem eine Kausalitätsüberprüfung notwendig. Es dürfen keine Ereignisse für bereits simulierte Zeitpunkte erzeugt werden.

3.4.2 Realisierung als binärer Baum

Diese Anforderungen lassen sich mittels eines nach Ausführungszeiten sortierten binären Baums realisieren. Er ist geeignet für schnelles Einsortieren. Da es mehrere Ereignisse gleicher Ausführungszeit geben kann, verwaltet jeder Knoten eine verkettete Liste von Ereignissen. Die weiteren oben genannten Besonderheiten werden durch die Einführung von Hilfs-Knotenzeigern und zusätzlicher Datenelemente in den Knoten berücksichtigt. Dadurch wird eine zusätzliche Beschleunigung erreicht.

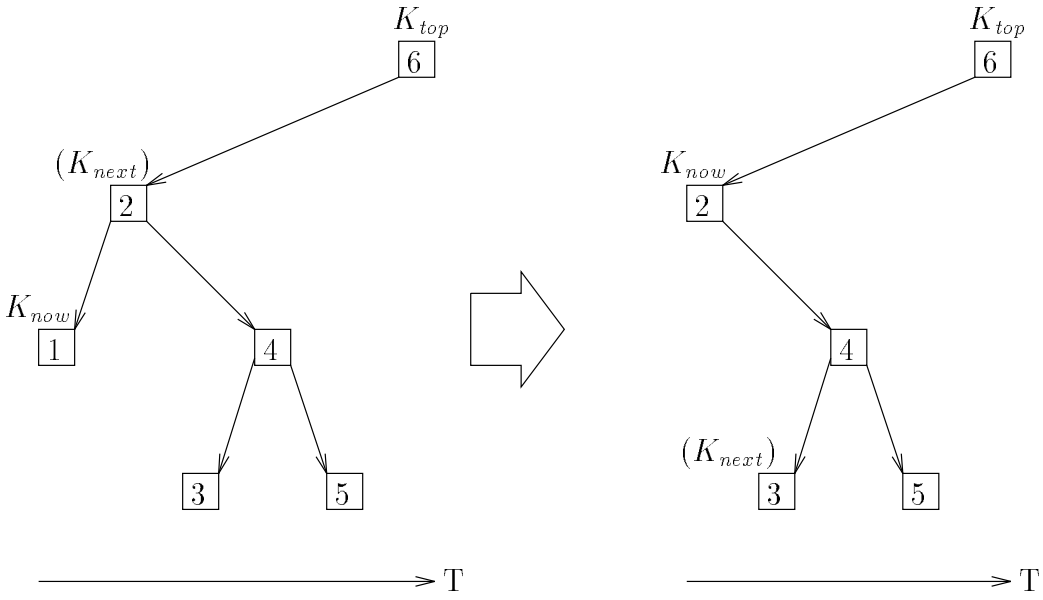


Abbildung 3.8: Suche nach den nächsten auszuführenden Ereignissen (Fall A)

Die Suche nach dem zweitkleinsten Knoten K_{next} (= Liste der als nächstes auszuführenden Ereignisse) beginnt sinnvollerweise nicht bei der Wurzel des Baums K_{top} , sondern beim kleinsten Knoten K_{now} (= Liste der gerade abgearbeiteten Ereignisse), da dieser K_{next} näherliegt. K_{now} bildet also den ersten Hilfs-Knotenzeiger. Mit ihm wird Punkt eins der allgemeinen Anforderungen Rechnung getragen.

Die Suche nach der nächsten auszuführenden Ereignisliste läßt sich anhand Abb. 3.8 und 3.9 erklären. Die Ziffern in den Knoten stellen die Ausführungszeit der darin gespeicherten Ereignisse dar. Da K_{now} immer auf den kleinsten Knoten zeigt, können sich niemals links von ihm weitere Knoten befinden, sonst läge eine Kausalitätsverletzung

vor. Es sind also prinzipiell nur zwei Fälle zu unterscheiden: K_{now} besitzt keinen rechten Nachfolgeknoten (Fall A) oder K_{now} besitzt einen rechten Nachfolgeknoten (Fall B).

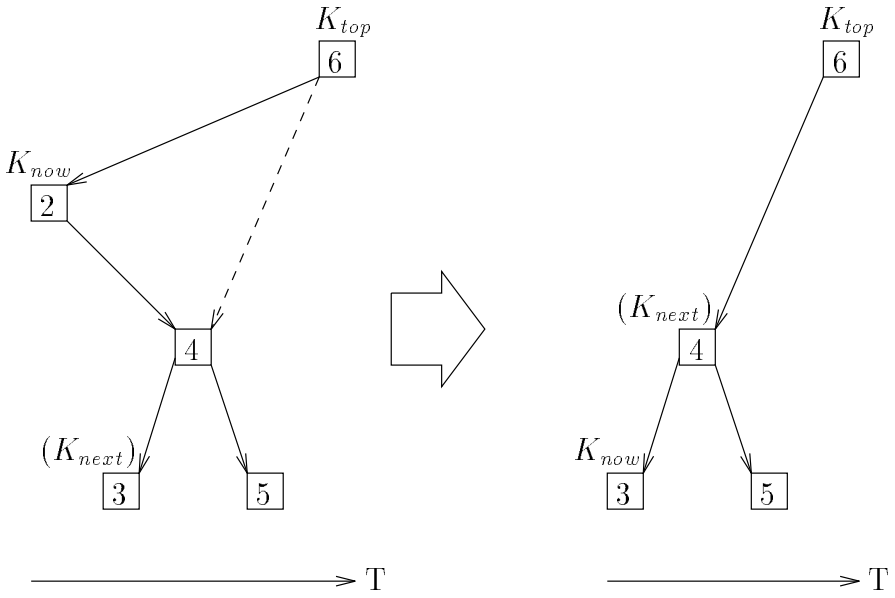


Abbildung 3.9: Suche nach den nächsten auszuführenden Ereignissen (Fall B)

Im Fall A ist der Knoten des nächsten aktuellen Zeitpunkts (K_2) sehr einfach zu finden. Es handelt sich immer nur um den Vorgängerknoten von K_{now}. Es ist also einfach der Hilfszeiger K_{now} umzusetzen, der alte aktuelle Knoten (K_1) zu löschen und der neue (K_2) als Suchergebnis zurückzuliefern.

Fall B gestaltet sich etwas komplexer. Hier sind zunächst zwei knoteninterne Zeiger umzusetzen, da vor dem Löschen von K_{now} die Baumstruktur lokal verändert werden muß. Der Zeiger auf den linken Nachfolgeknoten des Vorgängerknotens von K_{now} (K_6) ist auf den rechten Nachfolgeknoten von K_{now} (K_4) zu setzen. Außerdem ist der Vorgängerknotenzeiger des rechten Nachfolgeknotens von K_{now} (K_4) auf dessen Vorgängerknoten (K_6) umzusetzen. Nun stellt K_{now} kein Baumelement mehr dar und kann gelöscht werden. Die Suche nach der nächsten auszuführenden Ereignisliste kann nun vom ehemaligen rechten Nachfolgeknoten von K_{now} (K_4) aus nach links gestartet werden. Dabei stößt man auf K_3, den neuen aktuellen Knoten.

Zusätzlich sind einige Spezialfälle zu berücksichtigen. Bei einem leeren Ereignisbaum (K_{top} unbelegt) wird von der Suchfunktion für die nächste aktuelle Ereignisliste ein Null-

zeiger zurückgeliefert. Ist der Zeiger K_{now} noch unbelegt (erstmalige Suche), muß der nächste aktuelle Knoten von K_{top} aus nach links gesucht werden. Außerdem ist sicherzustellen, daß keiner der Hilfszeiger auf einen gelöschten Knoten zeigt. Dazu sind diese jedesmal sinnvoll neu zu besetzen, falls sie auf einen abgearbeiteten, zu löschenden Knoten zeigen.

Punkt zwei und drei der allgemeinen Anforderungen werden bei der Implementierung der Methode zur Aufnahme eines neuen Ereignisses berücksichtigt. Die Suche nach dessen Einfügepunkt wird nicht von der Baumwurzel aus gestartet, sondern vom Ort des zuletzt gespeicherten Ereignisses, auf den der neue Hilfsknotenzeiger K_{new} weist. Gleiche oder nahe beieinanderliegende Einfügepunkte nacheinander generierter Ereignisse werden auf diese Art und Weise sehr schnell gefunden.

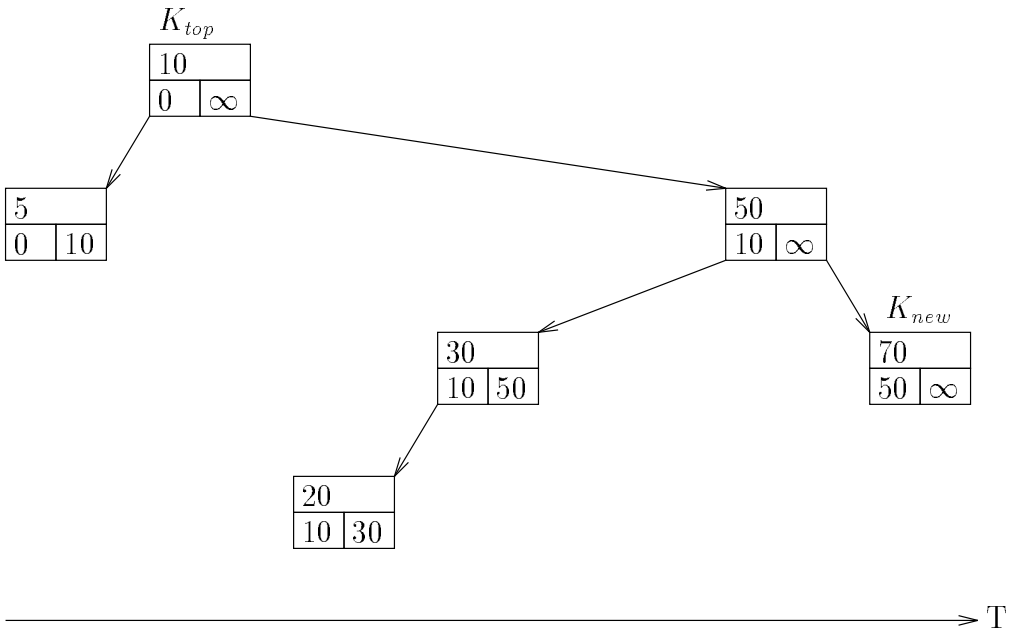


Abbildung 3.10: Ereignisverwaltung vor dem Einfügen

Allerdings entsteht durch diese Vorgehen die Notwendigkeit, in bestimmten Fällen auch in Richtung der Baumwurzel nach einem Einfügepunkt zu suchen. Um dies zu vereinfachen werden in jedem Knoten zwei neue Variablen min und max eingeführt (in Abb. 3.10 und 3.11 die beiden unteren Zahlen in den Knoten). Sie stellen die mögliche Unter- und Obergrenze für den Ausführungszeitpunkt aller Nachfolgeknoten dar.

Die Suche nach dem neuen Einfügepunkt gestaltet sich nun recht einfach (vgl. Abb. 3.10

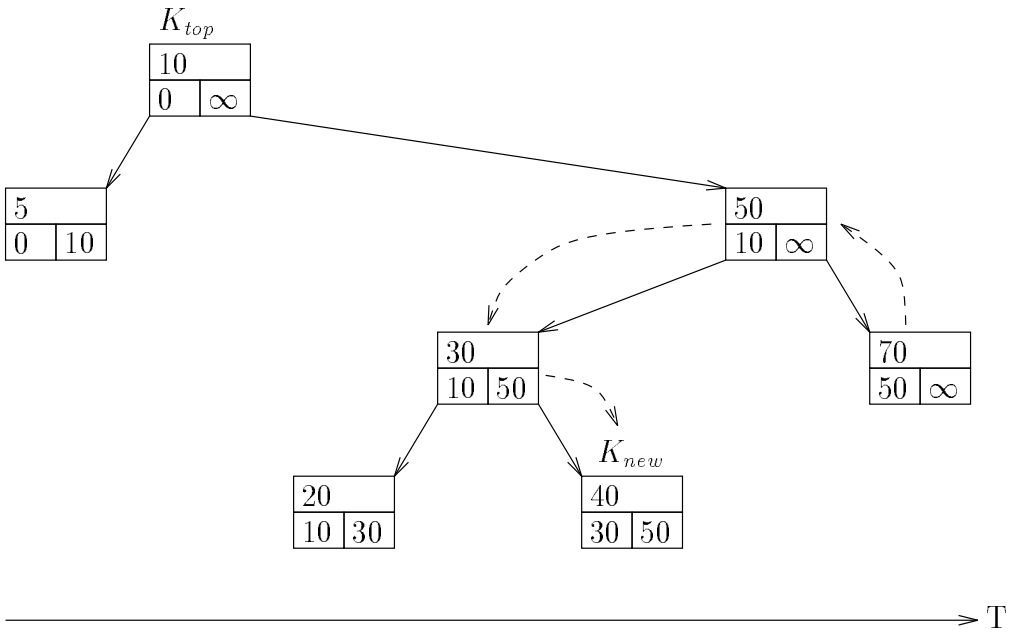


Abbildung 3.11: Ereignisverwaltung nach dem Einfügen

und 3.11). Stimmt der Ausführungszeitpunkt des neuen Ereignisse mit dem des zuletzt eingefügten überein, steht der Hilfszeiger K_{new} schon auf dem richtigen Knoten. Das Ereignis muß nur an die dort gespeicherte Liste angefügt werden.

Ist dies nicht der Fall (z.B neues Ereignis mit $T_{Exe} = 40$), wandert man zunächst innerhalb des Baumes in Richtung Wurzel (K_{70}, K_{50}), bis der Ausführungszeitpunkt des neuen Ereignisses mit dem des jeweiligen Knotens übereinstimmt (Ereignis wird hier eingefügt und K_{new} aktualisiert) oder innerhalb dessen Grenzen min und max liegt ($K_{50} : min = 10 < T_{Exe} = 40 < max = \infty$). Dies ist ein Zeichen dafür, daß der Einfügepunkt ein Nachfolger dieses Knotens ist.

Deshalb wird ab jetzt in entgegengesetzter Richtung weitergesucht. Man entfernt sich wieder von der Baumwurzel (K_{50}, K_{30}). Findet man dabei den nötigen Einfügepunkt, wird das neue Ereignis an die dortige Liste angehängt und K_{new} aktualisiert. Bleibt die Suche erfolglos (K_{30} hat keinen rechten Nachfolgeknoten), muß hier ein neuer Knoten erzeugt werden, der dann das neue Ereignis aufnimmt und K_{new} darstellt (K_{40}). Dabei sind natürlich auch die Werte min und max des neuen Knotens richtig zu setzen.

Kapitel 4

LDSIM-Nachbildung

4.1 Signaldarstellung

Wie bereits in Kapitel 3.1 erläutert, stehen die Element-Objekte der im Rechner dargestellten Schaltungstopologie durch Signal-Objekte untereinander in Verbindung. Sie stellen unter anderem die Signalnetze der simulierten Schaltung dar und sind nur für die jeweils daran angeschlossenen Elemente sichtbar. Diese Signal-Objekte können prinzipiell beliebig komplex sein, wodurch eine größtmögliche Freiheit zur Signaldarstellung erreicht wird.

Für die Nachbildung des LDSIM reichen zwei relativ einfache Modelle aus. Das erste entspricht der im LDSIM allgemein verwendeten sechswertigen Logik [2]. Das zweite ist der im LDSIM-Busprimitiv intern verwendeten Bussignaldarstellung nachempfunden, welche zusätzlich einen hochohmigen Zustand modelliert. Durch die Einführung dieses zweiten allgemeinen (nicht nur elementinternen) Signals wird eine größere Flexibilität zur Simulation von Bussen erreicht. Beide Signalarten lassen sich durch einfache Integerzahlen darstellen (nach OOP-Terminologie können auch elementare Datentypen als Objekte bezeichnet werden).

4.1.1 Sechswertige Logik

Ausgangspunkt für die Logiksimulation ist zwangsläufig die binäre Logik. Die ausschließliche Verwendung von logisch Null und Eins ist allerdings ungeeignet, um Unsicherheiten und Signalwertübergänge darzustellen.

Tupel	Interpretation	Basiswert	Wertigkeit bei der Codierung (oktal)
(0,0)	logisch Null	l	1
(1,1)	logisch Eins	h	2
(1,0)	Übergang von Eins auf Null	f	4
(0,1)	Übergang von Null auf Eins	r	10

Tabelle 4.1: Basiswerte

Aus diesem Grund werden aus den Werten Null und Eins in der ersten Stufe der Modellierung Zweiertupel gebildet (Tabelle 4.1). Damit lassen sich bereits Signalwertübergänge darstellen. Das erste Tupelelement stellt den Ausgangs-Zustand dar, während das zweite Tupelelement den Ziel-Zustand repräsentiert. Die vier Kombinationsmöglichkeiten werden als Basiswerte bezeichnet. Es existieren demnach jeweils zwei Übergangsbasiswerte (f,r) und statische Basiswerte (l,h).

Die zweite Modellierungs-Stufe besteht in einer Mengenbildung aus den Basiswerten. Dadurch lassen sich dann Unsicherheiten ausdrücken. Ist zum Beispiel nicht bekannt, ob ein Signalwert logisch Eins oder Null ist, wird dies durch die Basiswertmenge $Z = \{h, l\}$ zum Ausdruck gebracht und als Unknown ($= U$) bezeichnet. Grundsätzlich ließen sich auf diese Art 16 verschiedene Signalwerte bilden. Da aber nicht alle Kombinationen sinnvoll sind, benötigt man drei Zusatzbedingungen zur Mengenbildung. Die erste schließt die leere Menge aus, da jedes Signal einen Wert besitzen muß (Formel 4.1). Die zweite (Formel 4.2) und dritte Bedingung (Formel 4.3) fordern, daß eine Menge, die einen Übergangsbasiswert enthält, auch explizit dessen Ausgangs- und Ziel-Zustand enthalten muß.

$$\{\} \subset Z \subseteq \{l, r, h, f\} \tag{4.1}$$

$$r \in Z \Longrightarrow h \in Z \wedge l \in Z \tag{4.2}$$

$$f \in Z \Longrightarrow h \in Z \wedge l \in Z \tag{4.3}$$

Dadurch sind nur noch die in Tabelle 4.2 wiedergegebenen Signalwerte möglich. Sie bilden den Wertevorrat für die Simulation von unidirektionalen Logiksignalen.

Die Codierung als Integer-Zahl geschieht mittels der Basiswerte. Sie ist ebenfalls den Tabellen 4.1 und 4.2 zu entnehmen.

Basiswertmenge	Symbol	Interpretation	Codierung (oktal)
{l}	L	Low	1
{h}	H	High	2
{h,l}	U	Unknown	3
{h,f,l}	F	Falling	7
{l,r,h}	R	Rising	13
{h,f,r,l}	C	Changing	17

Tabelle 4.2: Signalwerte

4.1.2 Erweiterung um Zustand „Hochohmig“

Tupel	Interpretation	Bus-Basiswert
(0,0)	logisch Null	l
(1,1)	logisch Eins	h
(Ω,Ω)	Hochohmig	t
(0,1)	Übergang von Null auf Eins	r
(0,Ω)	Übergang von Null auf Hochohmig	${}_r$t
(1,0)	Übergang von Eins auf Null	f
(1,Ω)	Übergang von Eins auf Hochohmig	${}_f$t
(Ω,0)	Übergang von Hochohmig auf Null	t$_f$
(Ω,1)	Übergang von Hochohmig auf Eins	t$_r$

Tabelle 4.3: Bussignal-Basiswerte

Zur Simulation von Bussen ist es nötig, nicht nur den Wert eines Signals anzugeben, sondern auch eine Aussage über dessen Signalstärke zu machen.

Die Grundwerte für Logiksimulation Null und Eins werden deshalb hier durch den Zustand Hochohmig (Ω) ergänzt. Er besitzt keinen expliziten Signalwert, sondern stellt nur eine im Vergleich zu Null und Eins schwächere Signalstärke dar. Aus diesen drei Grundwerten läßt sich nun genau wie unter 4.1.1 in zwei Stufen ein zur Simulation von Bussen geeigneter Wertevorrat entwickeln.

In der ersten Stufe werden wieder 2-Tupel gebildet, woraus sich die in Tabelle 4.3 wiedergegebenen neun Bus-Basiswerte ergeben. In der zweiten Stufe bildet man aus diesen Tupeln wiederum unter Berücksichtigung von Zusatzbedingungen Mengen. Dadurch ergeben sich 79 verschiedene, gültige Signalwerte, welche nicht mehr mit Namen bezeichnet werden.

Da der Grundwertevorrat der sechswertigen Logik nach 4.1.1 eine Teilmenge des erweiterten Grundwertevorrats bildet und die Vorgehensweise (Basiswertbildung, Mengenbildung) zur Bildung von Signalwerten in beiden Fällen die gleiche ist, stellt der neue Signalwertevorrat eine Übermenge des Wertevorrats {H,L,R,F,C,U} dar.

Die Codierung erfolgt nicht wie bei den Signalen der sechswertigen Logik ausschließlich dadurch, daß den Basiswerten bestimmte Bits in der Integerdarstellung zugeordnet werden. Die Bedeutung der einzelnen Bits beim Bussignal ist Tabelle 4.4 zu entnehmen.

Bit	Wertigkeit (oktal)	Bedeutung
1	1	Basiswert l
2	2	" h
3	4	" f
4	10	" r
5	20	Dieses Bit zeigt an, ob der Zustand Hochohmig an diesem Signal noch möglich ist. Ist der Signalwert zum Beispiel sicher High, ist dieses Bit nicht gesetzt.
6	40	Basiswert $_f$t
7	100	" $_r$t
8	200	" t$_f$
9	400	" t$_r$
10	1000	Ist diese Bit gesetzt, ist der Signalwert sicher Low.
11	2000	Signalwert ist sicher High.

Tabelle 4.4: Bussignal-Codierung

4.2 Schaltungs-Primitive

Die nachzubildenden Gatterprimitive des LDSIM lassen sich grob klassifizieren. Man unterscheidet dabei solche mit rein kombinatorischer Funktion und solche mit internen Zuständen (sequentielle Primitive). Diese Unterscheidung spiegelt sich auch in der Klassenhierarchie der Topologieelemente (Abb. A.3 im Anhang) wieder. Eine Sonderstellung nehmen die beiden Busprimitive ein. Ihre Funktion kann auf mehrere Topologieelemente aufgeteilt werden, da sich nun Bussignale allgemein und nicht nur primitivintern darstellen lassen. Die Klasse `SimPrim` stellt die gemeinsame Basisklasse für alle Primitive dar.

Im Gegensatz zu den LDSIM-Primitiven findet in den hier vorgestellten Primitiven noch keine Verzögerungsbehandlung statt. Sie wird mittels spezieller Topologieelemente (Kapitel 4.4) primitiv-unabhängig realisiert, da es sich hierbei nach OOP-Denkweise um zwei unterschiedliche, trennbare Aufgabenstellungen handelt.

Die Funktion der hier vorgestellten Primitive besteht nur in der Simulation des logischen Verhaltens der entsprechenden Gatter. Bekommmt ein Objekt einer Primitiv-Klasse durch den Aufruf der Aktivierungsmethode eine Veränderung an einem Eingangssignal mitgeteilt, fügt es sich zunächst nur in einen Puffer auszuwertender Elemente ein (Kapitel 4.6). Erst wenn alle bisher bekannten, in Ereignissen codierten Signalwertwechsel des aktuellen Zeitpunkts ausgeführt wurden, beginnt die Auswertung der in diesem Puffer zwischengespeicherten Primitive. Sie verändern ihre Ausgangssignale und setzen die Nachfolgeelemente mittels der Aktivierungsmethode davon in Kenntnis. Würden die Gatter bei jedem Eingangssignalwechsel sofort ausgewertet (ohne Zwischenspeicherung in einem Puffer), käme es bei Veränderungen an mehreren Eingängen zu mehrfachen und deshalb überflüssigen Elementauswertungen während eines Simulationszeitpunktes. Die Pufferung wird genauer in Kapitel 4.6 erläutert.

4.2.1 Kombinatorische Primitive

Da sich die Funktion dieses Gattertyps auf eine Abbildung der Eingangssignale

$$\underline{i} = (i_1, i_2, i_3, ..., i_n) \tag{4.4}$$

auf die Ausgangssignale

$$\underline{o} = (o_1, o_2, o_3, ..., o_m) \tag{4.5}$$

beschränkt (Gleichung 4.6), müssen in den entsprechenden Klassen keine zusätzlichen Daten neben den geerbten eingeführt werden. Es genügt, die geerbte virtuelle Methode zur Primitiv-Auswertung dem jeweiligen Fall angepaßt zu implementieren (Funktion f).

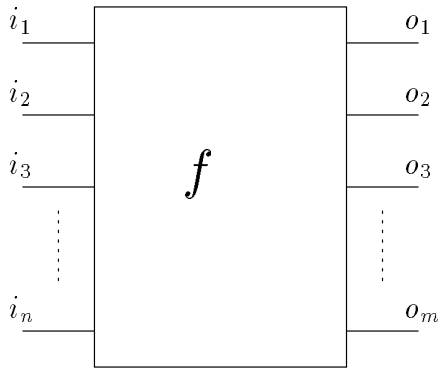


Abbildung 4.1: Kombinatorischer Primitiv

$$\underline{o} = f(\underline{i}) \tag{4.6}$$

Sie kann bei den meisten geforderten Gattern (Tabelle B.1 im Anhang) durch wenige Tabellenzugriffe realisiert werden. Bei der Erstellung dieser Tabellen ist zu beachten, daß es sich, bedingt durch die Verwendung der sechswertige Logik, um keine Boolesche Algebra mehr handelt. Das Distributivgesetz und das Gesetz des komplementären Elements sind verletzt ([2] S.43), während Kommutativ- und Assoziativgesetz weiterhin gelten. Zudem ist zu beachten, daß die Ausgangssignale eines Gatters (un-)sicherer werden müssen, das heißt sie müssen (mehr) weniger Basiswerte enthalten, wenn ein Eingangssignal (un-)sicherer wird. Die verwendeten Tabellen sind in Tab. 4.5 angegeben.

Ein Nand-Gatter mit drei Eingängen wertet man beispielsweise mit folgenden drei Tabellenzugriffen aus:

$$o_1 = NOT[AND[AND[i_1, i_2], i_3]] \tag{4.7}$$

Die Auswertung der vier Primitive ADD, MULT (Addierer und Multiplizierer), ADDC und MULTC (Add. und Mult. mit Konstante) kann nicht ausschließlich durch Tabellenzugriffe erfolgen. Die Erzeugung der Ausgangssignale erfolgt hier durch einen Algorithmus, wie er schon im LDSIM Verwendung fand.

Die in Tabelle B.1 (Anhang) in der rechten Spalte angegebenen Klassen fassen jeweils eine Gruppe von Gattern zusammen, die sich nicht in ihrer Funktion, sondern nur in der Anzahl der Eingänge unterscheiden. Dies ist möglich durch eine entsprechende Pa-

UND	H	L	U	F	R	C
H	H	L	U	F	R	C
L	L	L	L	L	L	L
U	U	L	U	F	R	C
F	F	L	F	F	C	C
R	R	L	R	C	R	C
C	C	L	C	C	C	C

ODER	H	L	U	F	R	C
H	H	H	H	H	H	H
L	H	L	U	F	R	C
U	H	U	U	F	R	C
F	H	F	F	F	C	C
R	H	R	R	C	R	C
C	H	C	C	C	C	C

EXOR	H	L	U	F	R	C
H	L	H	U	R	F	C
L	H	L	U	F	R	C
U	U	U	U	C	C	C
F	R	F	C	C	C	C
R	F	R	C	C	C	C
C	C	C	C	C	C	C

NICHT	H	L	U	F	R	C
	L	H	U	R	F	C

Tabelle 4.5: Verknüpfungstabellen für sechswertige Logik

rametrisierung. Bei jeder Erzeugung eines Objekts des entsprechenden Klassentyps ist die gewünschte Anzahl Eingänge mit anzugeben.

4.2.2 Sequentielle Primitive

Diese Art von Schaltungsprimitiven (Abb. 4.2) zeichnet sich dadurch aus, daß der Wert ihrer Ausgangssignale nicht ausschließlich von der Belegung der Eingangssignale zum entsprechenden Zeitpunkt abhängt. Sie besitzen interne Zustände (Vektor $\underline{s}$), die die Ausgänge zusätzlich beeinflussen. Die Auswertung eines solchen Primitivs besteht in der Berechnung der Funktion f_s (Gleichung 4.9) für eine bestimmte Eingangsbelegung und bestimmte interne Zustände.

$$\underline{s} = (s_1, s_2, s_3, ..., s_l) \tag{4.8}$$

$$\underline{o} = f_s(\underline{i}, \underline{s}) \tag{4.9}$$

Zusätzliche Variablen in den entsprechenden Objekten repräsentieren die internen

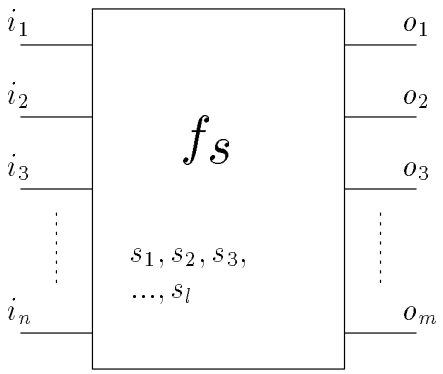


Abbildung 4.2: Sequentieller Primitiv

Zustände dieser Gatter-Primitive. Entsprechend den feststellbaren Gemeinsamkeiten lassen sich hier zwei neue Basisklassen einführen (vgl. Abb. A.3 im Anhang).

Die erste ist eine Basisklasse für alle sequentiellen Primitive (`SimPrimDigState`). Sie enthält unter anderem Variablen für einen internen Zustand und dessen Initialwert bei Simulationsbeginn. Die zweite Basisklasse (`dffcommon`) bildet die Grundlage für alle D-Flip-Flop-Versionen. Die Auswertemethoden dieser sequentiellen Primitive sind von Fall zu Fall verschieden. Sie lassen sich erst in den Klassen am Ende der Hierarchie implementieren.

4.2.3 Busprimitiv

Um den LDSIM exakt nachzubilden, sind auch dessen Busprimitive zu realisieren. Diese modellieren jeweils einen kompletten Bus. Das heißt, sie besitzen eine parametrisierte Anzahl Eingangssignalpaare, die jeweils aus einem Dateneingang und dem dazugehörigen Enableeingang bestehen und einen Bustreiber darstellen. Außerdem besitzen sie ein bis zwei Ausgänge. Einer liefert den Buszustand als sechswertiges Logiksignal (Primitiv BUS und BUSC), der andere zeigt einen eventuellen Buskonflikt an (nur Primitiv BUSC). Mit diesen großen Primitiven lassen sich Bussysteme nur relativ unbefriedigend simulieren, deshalb wird hier eine weitere Aufteilung der Funktion auf mehrere Topologieelemente durchgeführt. Der erste notwendige Schritt hierfür war die Definition des Bussignals in Kapitel 4.1.2 als allgemein verwendbares Signal.

Hier werden nun zwei zusätzliche Topologieelemente eingeführt: Ein Bustreiber (Klasse `SimPrimBusdrv`) und ein Element zur Busauswertung (Klasse `SimPrimBuscoll`).

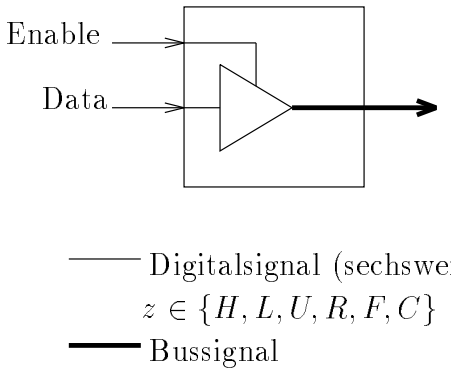


Abbildung 4.3: Bustreiber

Der Bustreiber (Abb. 4.3) besitzt einen Daten- und einen Enable-Eingang. Beide können nur an ein sechswertiges Logiksignal angeschlossen werden. Als einziges Ausgangssignal liefert der Bustreiber ein Bussignal. Es handelt sich hierbei um ein Element ohne interne Zustände. Die Abbildung der Eingangssignalwerte auf den Ausgangssignalwert ist Tabelle B.3 im Anhang zu entnehmen. Da für die 79 möglichen Bussignalwerte keine sinnvollen Namen mehr eingeführt werden konnten, kann darin nur die Oktalcodierung angegeben werden.

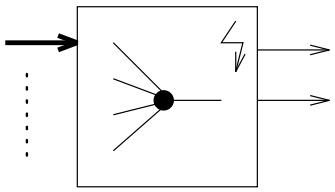


Abbildung 4.4: Busauswerter

Das Topologieelement zur Busauswertung (Abb. 4.4) besitzt eine beliebige Anzahl von Eingängen. Sie müssen mit Bussignalen verbunden werden. Die Aufgaben dieses Elements sind, den Bussignalwert abzubilden auf ein Signal sechswertiger Logik und Buskonflikte festzustellen und anzuzeigen. Der Busauswerter besitzt deshalb zwei Ausgänge, die jeweils ein sechswertiges Logiksignal liefern. Ein Ausgang stellt das abgebildete Bussignal dar, der andere meldet einen eventuellen Buskonflikt. Die Berechnung dieser beiden Signale geschieht analog zur internen Berechnung im LDSIM-Busprimitiv (keine internen Zustände). Auf welchen Signalwert der Buszustand „Hochohmig“ abgebildet

werden soll, läßt sich bei der Erzeugung des Busauswerte-Objekts im Programm als Parameter angeben.

Mittels der beiden neuen Primitive lassen sich nun die LDSIM-Busprimitive durch eine Klasse nachbilden. Da alle nötigen Funktionen für die logische Funktion des Busses bereits in den eben vorgestellten neuen Topologieelementen enthalten sind, bildet die Klasse `bus` nur noch eine Hülle, die nach außen wie einer der alten Busprimitive aussieht, intern aber aus mehreren Bustreibern und einem Busauswerter besteht (Abb. 4.5). Ein Objekt der Klasse `bus` ist nur während des Aufbaus der Schaltungstopologie (Kapitel 3.1) nötig. Nach dem Start der eigentlichen Simulation tritt es in den Hintergrund und es werden nur noch Methoden der in ihm enthaltenen Objekte (Bustreiber, -auswerter) aufgerufen.

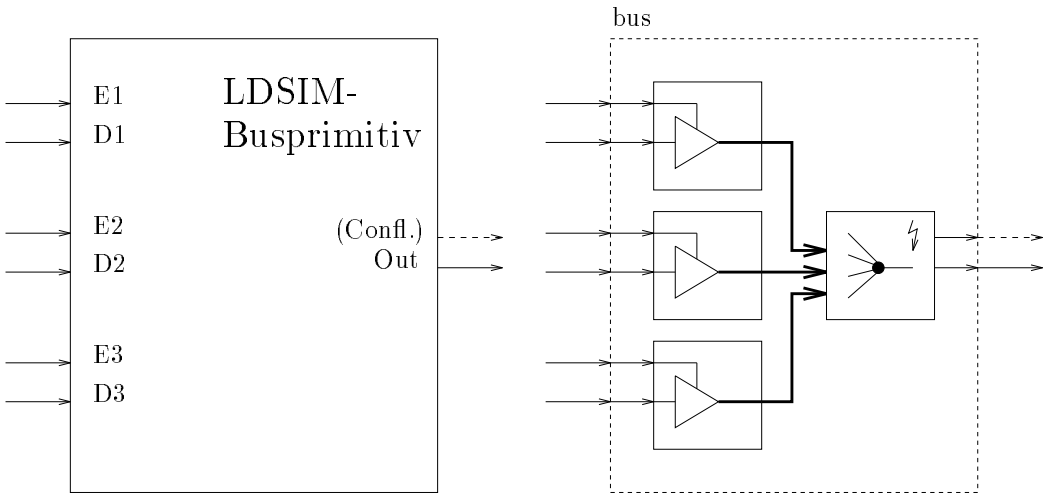


Abbildung 4.5: Busprimitiv-Nachbildung

Die Aufgabe der „Hüllenklasse“ ist es also, sämtliche Methodenaufrufe zum Verbindungsaufbau von/nach außen an die richtigen internen Objekte weiterzuleiten und andererseits intern die Bustreiber mit dem Busauswerter zu verbinden. Damit ist gewährleistet, daß während der Simulation einerseits bei einem Eingangssignalwertwechsel automatisch die internen Objekte aktiviert werden und andererseits bei einem Ausgangssignalwertwechsel die internen Objekte sofort die nachfolgenden externen Topologieelemente aktivieren.

4.3 Signaleingabe, -ausgabe

Signaleingabe bedeutet, daß die Stimulis eingelesen werden müssen und an die Schaltungstopologie angelegt werden müssen. Die Signalausgabe besteht in der Protokollierung der Simulationsergebnisse.

4.3.1 Stimulis

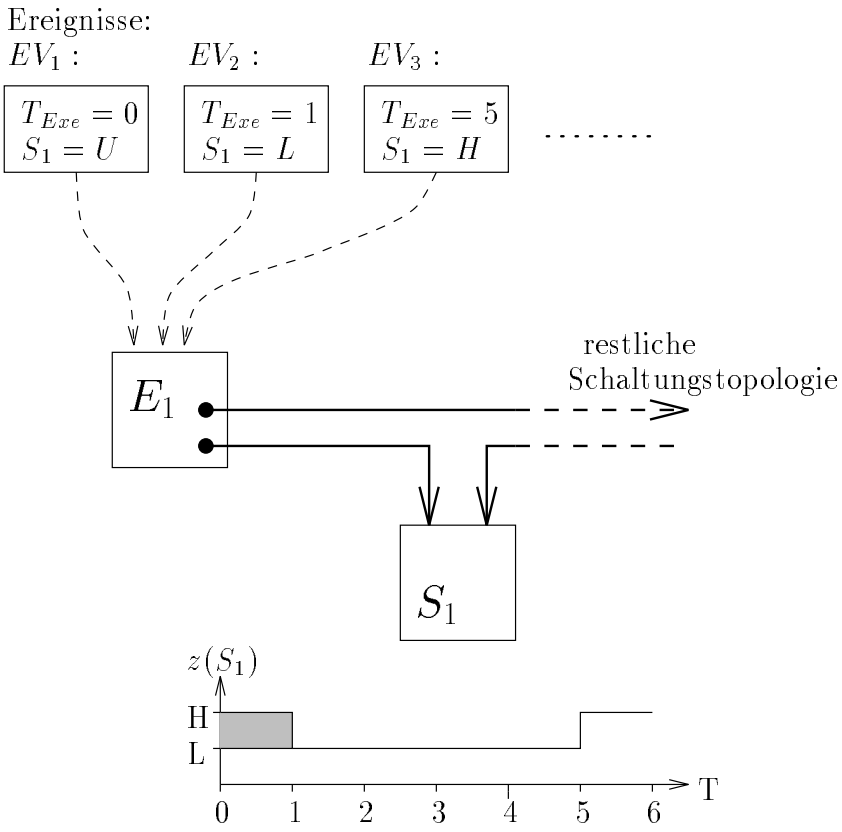


Abbildung 4.6: Darstellung eines Primäreingangs als Topologieelement

Anders als beim LDSIM werden alle Stimulis bereits vor der Simulation eingelesen und in Form von speziellen Ereignissen (Klasse `SimEvDiginp`) der Ereignisverwaltung übergeben (siehe Kap. 4.9.3). Damit die Ereignisse nun auf die Schaltungstopologie wirken können, wird eine neue Topologieelement-Klasse `SimInDig` eingeführt. Ein Objekt dieser Klasse (E_1 in Abb. 4.6) besitzt nur einen Ausgang und keinen Eingang und wird von

den zugehörigen Ereignisobjekten ($EV_1, EV_2, EV_3, ...$) gesteuert. Solch ein Eingangssignalereignis beinhaltet also einen Zeiger auf das zugehörige Topologieelement (Objekt der Klasse `SimInDig`, E_1), den neuen Signalwert und natürlich den Zeitpunkt, ab dem dieser gelten soll, was dem Ausführungszeitpunkt (T_{Exe}) des Ereignisses entspricht.

Kommen nun während der Simulation die Eingangssignalereignisse zur Ausführung, ändern sie den Ausgangssignalwert ihres zugehörigen Topologieelements und bilden so den gewünschten Signalverlauf (S_1) am entsprechenden Primäreingang.

Hier ist nun ein kleiner Vorgriff auf das Halbschrittverfahren (Kapitel 4.7) notwendig. Alle Ereignisse des Simulators, die Signalwertänderungen betreffen, werden in zwei Schritten ausgeführt. Im ersten Schritt können die zugehörigen Signale nur unsicherere Werte annehmen (mehr Basiswerte), während sie in der zweiten Stufe nur sicherer werden können (weniger Basiswerte). Die Begründung hierfür ist in Kapitel 4.7 beschrieben. Diese Zweistufigkeit muß auch bei Primäreingangssignalen eingehalten werden.

Im ersten Halbschritt der Ereignisausführung werden deshalb einfach die Basiswertmengen des neuen und alten Ausgangsignalwerts des Topologieelements vereinigt. Entsteht dadurch eine Basiswertmenge, die keinen Signalwert darstellt, ist diese durch geeignete Basiswerte zu ergänzen, so daß ein gültiger Signalwert entsteht. Dieser bildet das vorläufige Ausgangssignal, von dem alle Nachfolgeelemente in Kenntnis gesetzt werden.

Erst der zweite Halbschritt setzt das Ausgangssignal endgültig auf den neuen Signalwert. Auch hierüber werden die Nachfolgeelemente mittels deren Aktivierungsmethoden informiert.

4.3.2 Simulationsergebnisse

Auch zur Protokollierung der Simulationsergebnisse wird ein neues Topologieelement (SimRecDig) eingeführt. Es besitzt je einen Ein- und Ausgang sechswertiger Logik, kann an jeder beliebigen Stelle in die Topologie eingebaut werden und den dortigen Signalverlauf in einen ihm zugeordneten Stream protokollieren. Damit der zeitliche Bezug dieser Ergebnisse nicht verlorengeht, müssen im gleichen Stream vom Simulator selbst Zeitmarker gesetzt werden. Ein solcher Zeitmarker besteht aus einer Integerzahl. Der Wert dieser Zahl ist die negative aktuelle Simulationszeit. Die negative Darstellung wird gewählt, um den Marker von anderen Einträgen im Protokollstream unterscheiden zu können.

Im Normalfall werden alle Elemente ihre Protokolle in den gleichen Stream lenken, der dann sämtliche Simulationsergebnisse enthält. In der vorliegenden Realisierung wird

allen Protokollelementen der Standardausgabestream zugeordnet. Es ist aber auch vorstellbar, einzelne Signale oder Signalgruppen in verschiedenen Dateien festzuhalten.

Da die Protokoll-Topologieelemente keine Signalwerte verändern, sondern nur lesend darauf zugreifen, sobald sie über eine Veränderung an ihnen informiert werden (Aktivierungsmethode), läßt sich die Klasse `SimRecDig` von `SimDist` (Kapitel 3.1) ableiten. Ein Objekt der Klasse `SimDist` zeichnet sich ja dadurch aus, daß der Zeiger auf sein Eingangssignal mit dem auf sein Ausgangssignal identisch ist.

Da es durch den Befehlssatz des Logiksimulators LDSIM möglich ist, die Protokollierung von Signalen erst während der Simulation zu starten oder zu beenden und außerdem die Initialisierungsphase des Simulators (Kapitel 4.9.2) auf keinen Fall protokolliert werden soll, werden in der Klasse `SimRecDig` zwei Flag-Variablen eingeführt, die die Protokollierbereitschaft steuern und von speziellen Ereignissen (siehe Kapitel 4.8.2) gesetzt beziehungsweise gelöscht werden können.

Wird nun ein Objekt der Klasse `SimRecDig`, dessen Flags die Protokollierung erlauben, aktiviert, schreibt es einen Protokolleintrag in den ihm zugeordneten Stream. Dieser Eintrag besteht aus zwei Integerzahlen. Die erste stellt die Signalnummer (positiv) des betreffenden Signals dar, die zweite ist der gemäß Tabelle 4.2 codierte Signalwert. Anschließend aktiviert das Objekt sein einziges (nur ein Ausgang) Nachfolgeelement.

4.4 Verzögerungen

Da die eingeführten Primitive nur logische Funktion haben, ist zusätzlich ein Verzögerungsobjekt für Signale sechswertiger Logik nötig, um das zeitliches Verhalten der Gatter zu modellieren. Die Kombination eines Primitiv-Topologieelements mit entsprechenden Verzögerungselementen an dessen Ausgängen bildet einen LDSIM-Primitiv erst komplett nach.

Diese Aufteilung wurde gewählt, da die Verzögerungsfunktion bei allen Primitiven prinzipiell gleich ist und mit der logischen Funktion nichts zu tun hat. Außerdem entsteht dadurch die Möglichkeit, überall in der Schaltungtopologie Signale sechswertiger Logik zu verzögern, zum Beispiel auch an Primitiveingängen.

Bei dem Verzögerungselement (Klasse `SimDelDig`) handelt es sich vorerst um das einzige Topologieelement, das Ereignisse während der Simulation erzeugt. Es besitzt je einen Ein- und Ausgang und verwendet das Laufzeitmodell „variable Verzögerungszeit“. Dieses Modell beruht darauf, daß steigenden und fallenden Flanken (Basiswerte r und f) am Eingang jeweils eine minimale (T_{Rmin}, T_{Fmin}) und maximale Verzögerung (T_{Rmax}, T_{Fmax})

zugeordnet wird. Im Ausgangssignalverlauf können dadurch neue Wertebereiche entstehen.

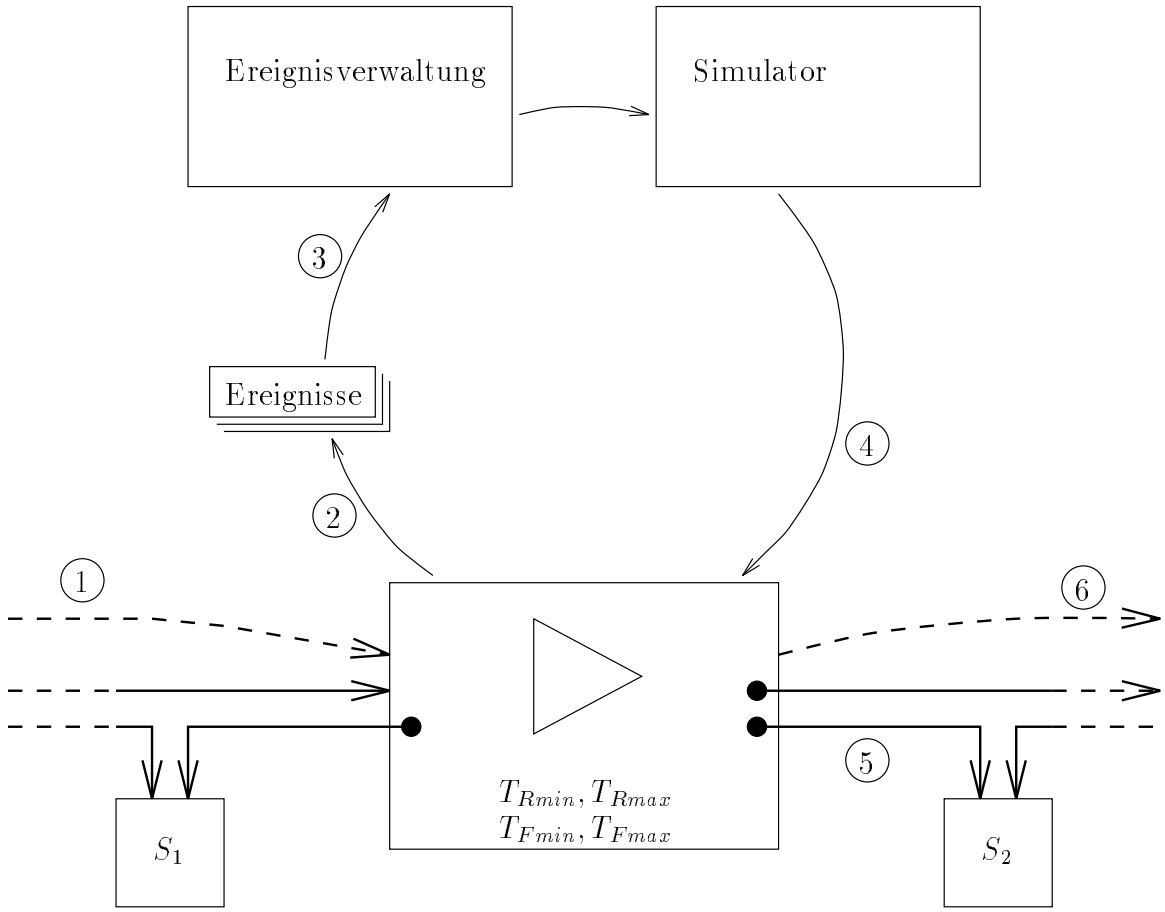


Abbildung 4.7: Verzögerungselement

Die von diesem Verzögerungs-Topologieelement erzeugten Ereignisse unterscheiden sich grundlegend von den in Kapitel 4.3 zur Darstellung von Eingangssignalverläufen eingeführten. Während in den Eingangssignalereignissen direkt Signalwerte gespeichert sind, werden in den Verzögerungsereignissen (Klasse `SimEvDig`) Signalwertwechsel festgehalten. Ein Signalwertwechsel läßt sich darstellen durch eine Menge hinzuzufügender Basiswerte und durch eine Menge zu entfernender Basiswerte. Weitere Datenelemente dieser Ereignisse sind ein Zeiger auf das betroffene Verzögerungsobjekt, die Ereignisgenerationszeit und natürlich der Ereignis-Ausführungszeitpunkt T_{Exe}.

Wird ein Verzögerungsobjekt nach einer Veränderung an dessen Eingangssignal S_1 aktiviert (Ziffer 1 in Abb. 4.4), erzeugt es aus dem alten und neuen Eingangssignalwert unter Berücksichtigung der Verzögerungszeiten $T_{Rmin}, T_{Rmax}, T_{Fmin}$ und T_{Fmax} bis zu vier neue Ereignisse (Ziffer 2), die der Ereignisverwaltung übergeben werden (3). Das Ausgangssignal bleibt vorerst unverändert. Die Ausführung eines der erzeugten Ereignisse durch den Simulator (4) führt zu erneuter Aktivität des Verzögerungsobjekts. Es verändert jetzt unter bestimmten Nebenbedingungen seinen Ausgangssignalwert (5) und aktiviert das nachfolgende Element in der Schaltungstopologie (6). Die Verfahren zur Ereignisgenerierung, Ereignisausführung und Spikebehandlung (Spike = kurzer Signalimpuls, dessen Dauer in der Größenordnung der Verzögerungszeit liegt) sind genauso realisiert wie im LDSIM und sind in [2] ausführlich beschrieben.

4.5 Abfangen von Oszillationen

In Schaltungen, die Rückkopplungsschleifen in ihrer Topologie enthalten, kann es während der Simulation zu Oszillationen kommen. Diese stellen zwar ein richtiges Simulationsergebnis dar, aber oft sind ihr genauer Verlauf und ihre Periodendauer nicht von Interesse. Trotzdem führen sie zu starker Simulatortätigkeit. Besonders negativ wirkt sich dies auf die Simulationsdauer aus, wenn die Periodendauer der Oszillation im Vergleich zum Abstand zwischen zwei Primäreingangsänderungen relativ klein ist oder sehr viele Topologieelemente an der Schwingung beteiligt sin. Um den unnötigen Zeitverbrauch zu verhindern, muß man solche Oszillationen erkennen und abfangen.

Es ist nicht erlaubt, nach dem Detektieren und Lokalisieren einer solchen Oszillation, sie einfach durch Verwerfen der zugehörigen Ereignisse zu unterbinden, da dadurch der Simulator fälschlicherweise Stabilität vortäuschen würde. Die einzige Möglichkeit ist, die beteiligten Signale auf den Signalwert Changing (C) zu setzen. Dadurch wird der Tatsache Rechnung getragen, daß sich dieses Signal in Wirklichkeit laufend ändern würde. Es entstehen also keine falschen Simulationsergebnisse, sondern nur ungenauere.

Um nun die Schwingungen festzustellen und in der Topologie zu lokalisieren, werden zwei Arten von Zählern eingeführt. Der erste, C_e existiert nur einmal im Simulator und zählt die seit der letzten Primäreingangsänderung ausgeführten Ereignisse. Überschreitet er eine vorgegebene Schranke L_e, beginnt die zweite Art von Zählern ($C_{s,i}$) zu arbeiten. Sie zählen an jedem Gatterausgangssignal die dort stattfindenden Signalwertwechsel. Überschreitet nun auch einer dieser Zähler seine vorgegebene Schranke $L_{s,i}$, wird das betreffende Signal i als oszillierend angenommen und statt auf den geforderten Signalwert auf Changing gesetzt. Damit ist der Kreislauf an Signalwertänderungen durchbrochen, da alle daran beteiligten Signale nach wenigen Simulationsschritten ebenfalls

den Signalwert Changing annehmen werden. Ab diesem Zeitpunkt finden keine weiteren Signalwechsel mehr statt und der Simulator kann die nächsten, nicht an der Oszillation beteiligten Ereignisse abarbeiten. Das Zurücksetzen der Zähler und Freigeben der blockierten Signale erfolgt bei der nächsten Primäreingangsänderung.

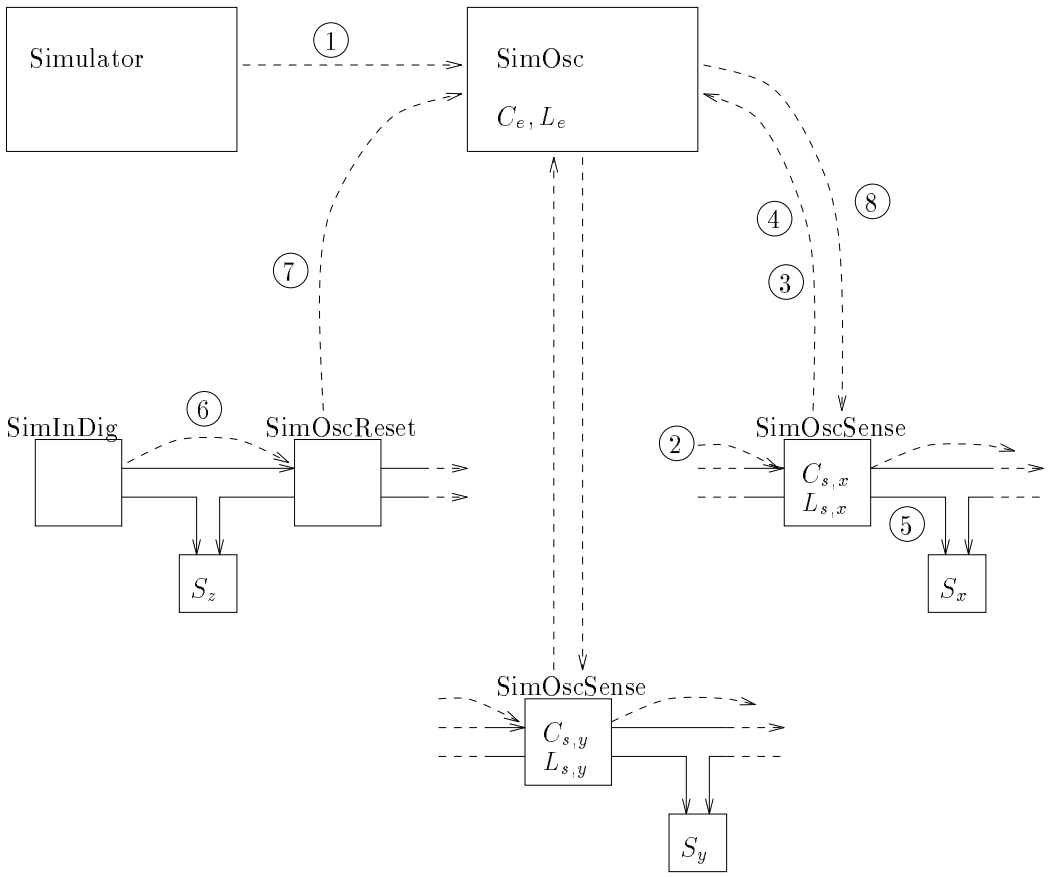


Abbildung 4.8: Oszillationsbehandlung

Für das Zähler-Rücksetzen und das Zählen der Signalwertwechsel lassen sich Topologieelemente einführen (Klassen `SimOscSense` und `SimOscReset`). Der Ereigniszähler ist jedoch nirgends in der Schaltung lokalisiert. Er wird deshalb durch eine außenstehende Klasse (`SimOsc`) realisiert, die mit den Rücksetz- und Zählobjekten in der Topologie in Verbindung steht.

Anhand von Abb. 4.8 läßt sich die Zusammenarbeit der Objekte erläutern. Der globale Ereigniszähler wird vom Simulator über jede Ereignisausführung informiert und zählt

diese (Ziffer 1). Wird ein Objekt zum Zählen von Signalwertwechseln in der Schaltungstopologie aktiviert (2), frägt es zuerst beim Ereigniszähler nach, ob der seine Schranke L_e schon erreicht hat (3). Nur wenn dies der Fall ist, wird der Zähler $C_{s,i}$ erhöht und das betroffenen Topologieelement teilt der Oszillationsüberwachung mit, daß hier sich eine Oszillation ankündigt (4). Hat darüberhinaus auch $C_{s,i}$ seine Schranke $L_{s,i}$ erreicht, wird der Ausgangssignalwert statt auf den gleichen Wert wie das Eingangssignal auf den Signalwert C gesetzt (5).

Die Objekte der Klasse `SimOscReset`, die in der Topologie direkt hinter jedem Primäreingang eingebaut sind, verhalten sich bei Aktivierung anders (6). Sie benachrichtigen den globalen Ereigniszähler von der Primäreingangsänderung (7). Dieser setzt dann sämtliche Zähler ($C_e, C_{s,i}$) zurück auf Null und löst bei allen oszillierenden Signalen deren Blockierung auf den Signalwert C auf (8).

4.6 Rangweise Elementauswertung

Wie schon in Kapitel 4.2 erwähnt, werden Schaltungsprimitive bei einer Veränderung an einem Ihrer Eingangssignale nicht sofort ausgewertet, sondern erst in einem Puffer abgelegt. Dies verhindert unnötige Mehrfachauswertungen von Primitiven zu einem Simulations-Zeitpunkt. Der implementierte Primitivpuffer (Klasse `SimCalcbuf`) wird hier vorgestellt.

Allen Primitiven in der Schaltung ist ein Rang zugeordnet. Er ist bereits in der binären Schaltungsbeschreibung enthalten und wird beim Aufbau der Schaltungstopologie mit eingelesen. Die Rangvergabe dient dazu, die Anzahl unnötiger Elementauswertungen weiter zu verringern, wenn in der Schaltung Gatter mit Ausgangsverzögerungen von Null enthalten sind (siehe [2] S.49 ff.). Dazu müssen die Elementauswertedurchläufe nach Rängen geordnet durchgeführt werden. Da sie vom Primitivpuffer aus gestartet werden, ist es günstig, auch diesen nach Rängen zu organisieren. Er besitzt für jeden vorkommenden Rang eine Liste, in die er die sich zur Auswertung anmeldenden Gatter eintragen kann.

Der Ablauf eines Zeitpunktes ($T = 10$) im Simulator wird anhand Abb. 4.9 erklärt. Tabelle 4.6 gibt alle dabei vorkommenden Ereignisse wieder. Das in Kapitel 4.7 vorgestellte Halbschrittverfahren bleibt dabei noch unberücksichtigt.

Zunächst sind nur die Ereignisse eins bis drei vorhanden. Die Ausführung von Ereignis eins bewirkt eine Veränderung von Signal S_1 durch das Verzögerungselement E_{t1}. Dieses aktiviert (Ziffer 1) außerdem sofort das Topologieelement E_{p1}, welches einen Primitiv darstellt. Er wird nicht sofort ausgewertet, sondern erst in den Puffer P eingetragen (2).

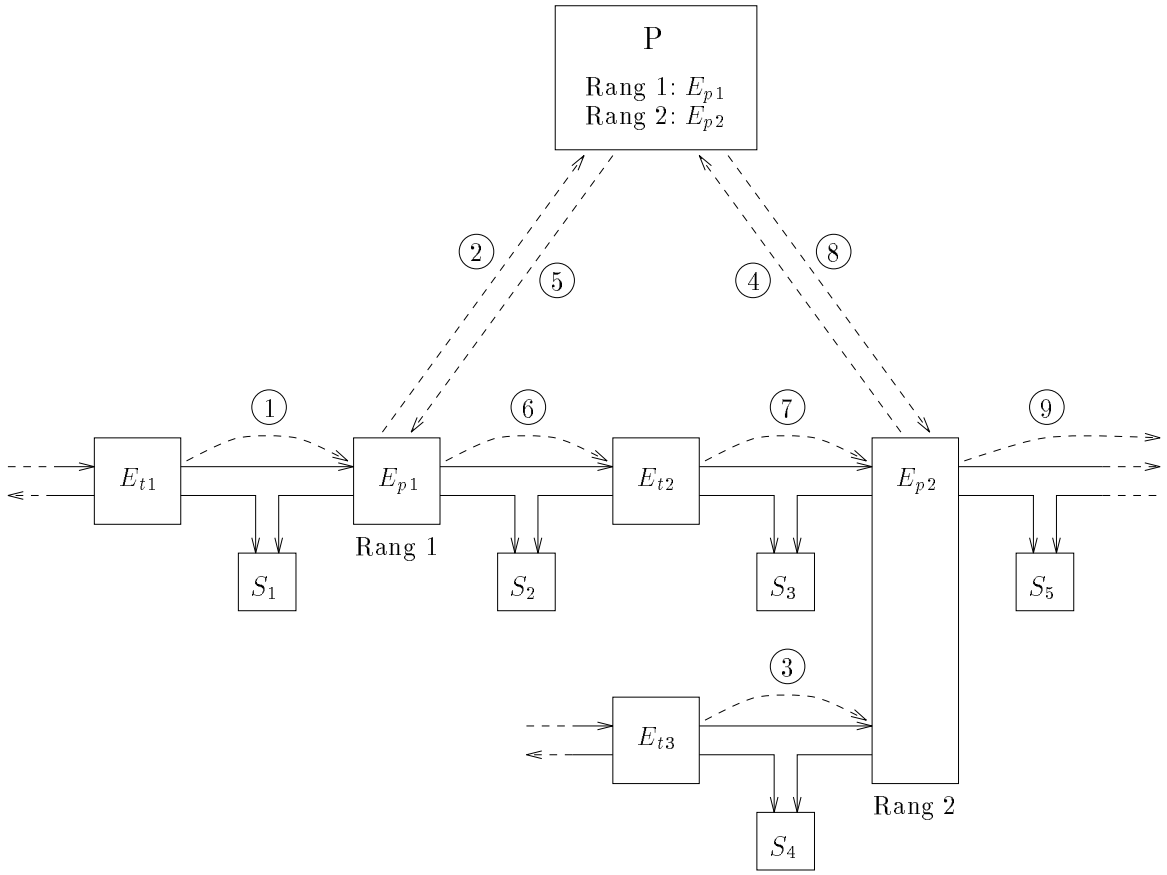


Abbildung 4.9: Rangweise Elementauswertung durch Primitivpuffer

Jetzt wird Ereignis zwei ausgeführt. Dabei wird S_4 verändert und E_{p2} aktiviert (3). Auch dieser Primitiv wird in P eingetragen (4). Ereignis drei startet jetzt die Auswertung der Primitive des nächsten Rangs (Rang eins). Im Puffer P ist unter Rang eins nur Primitiv E_{p1} eingetragen (5). Bei dessen Auswertung wird Signal S_2 verändert und das zweite Verzögerungselement E_{t2} aktiviert (6). Dieses enthält eine Verzögerungszeit von Null und generiert deshalb das Ereignis vier für den aktuellen Zeitpunkt. Dies wird vom Simulator bemerkt, weshalb er nicht sofort mit der Auswertung des nächsten Rangs fortfährt, sondern Ereignis fünf erzeugt und Ereignis vier ausführt. Dieses bewirkt die Veränderung von S_3 und die Aktivierung von Primitiv E_{p2} (7). Er muß nicht in P eingetragen werden, da er hier schon erfaßt ist. Nun wird der Puffer P von Ereignis fünf erneut dazu aufgefordert, den nächsten Rang auszuwerten. Rang zwei mit dem Primitiv E_{p2} ist an der Reihe (8). Dessen Auswertung bewirkt eine Veränderung von S_5 und die Aktivierung seines Nachfolgeelements (9).

Hier wird nun das Beispiel abgebrochen. Man sieht, daß Primitiv E_{p2} zweimal ausgewertet worden wäre, wenn der Primitivpuffer P nicht rangweise organisiert wäre.

Ereignisse für Zeitpunkt $T = 10$

Nr.	Typ	Aktion
1	SimEvDig	Signalwert von S_1 ändern
2	SimEvDig	S_4 ändern
3	SimEvCalc	Auswertung der nächsten Ränge
4	SimEvDig	S_3 ändern
5	SimEvCalc	Auswertung der nächsten Ränge

Tabelle 4.6: Ereignisliste

4.7 Halbschrittverfahren

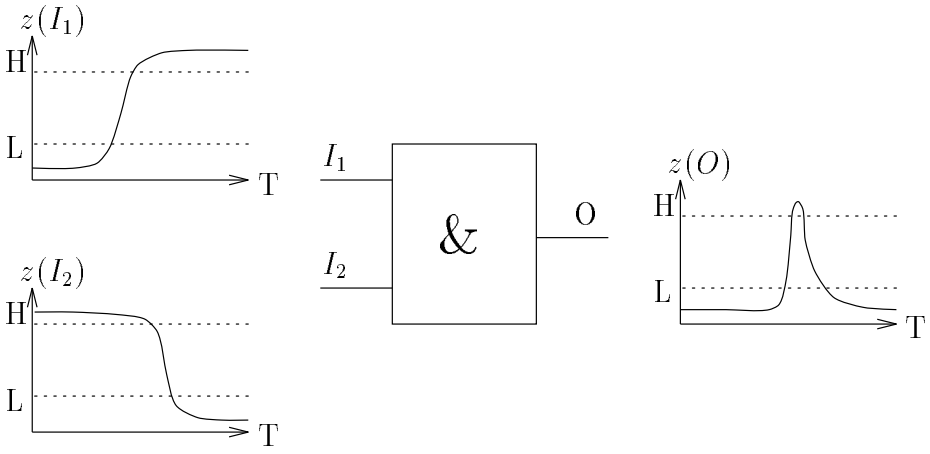


Abbildung 4.10: Hazard an realem Gatter

Das Halbschrittverfahren dient zur Erkennung von Hazards. Ändern sich an einem Gatter an zwei Eingängen die Signalwerte zum gleichen Simulationszeitpunkt, ist dies aufgrund der Zeitdiskretisierung im Simulator nicht gleichbedeutend mit einer echten physikalischen Gleichzeitigkeit. An einem realen Gatter kann durch eine minimale Zeitdifferenz der Eingangsveränderungen am Gatterausgang ein sehr kurzer Puls (Hazard)

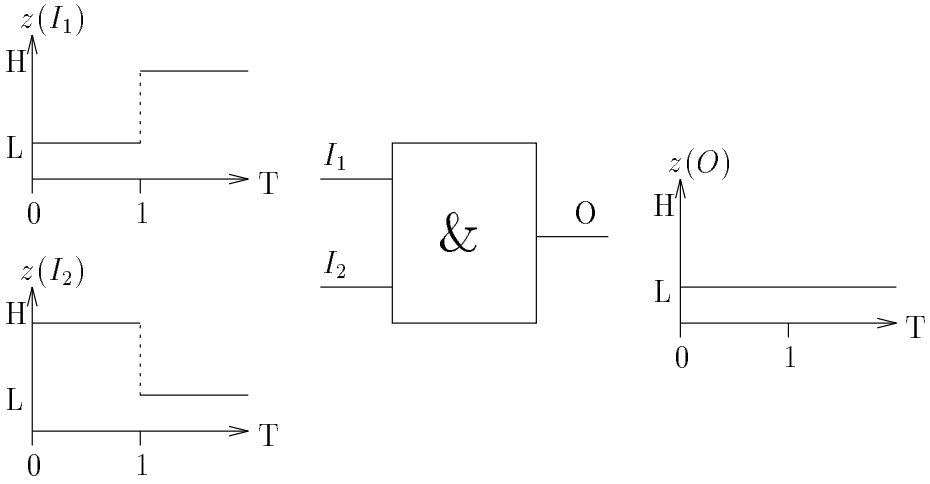


Abbildung 4.11: Kein Hazard an simuliertem Gatter

entstehen (Abb. 4.10). Dieser wird im Simulator ohne Halbschrittverfahren durch die Gleichzeitigkeit der Ereignisse nicht erzeugt (Abb. 4.11).

Da aber viele Schaltungsfehler auf solche Hazards, die zum Beispiel durch Races entstehen können, zurückzuführen sind, ist es unerläßlich, sie bei der Entwurfsverifikation mittels der Simulation zu finden. Hierzu ist das Halbschrittverfahren geeignet (siehe [2] S.56 ff.).

Das Halbschrittverfahren beruht darauf, alle Ereignisse in zwei Schritten durchzuführen. Im ersten Schritt dürfen dabei die Signale durch Hinzufügen von Basiswerten nur unsicherer werden. Im zweiten Schritt nehmen sie dann durch Löschen von Basiswerten ihren endgültigen Wert an. An dem UND-Gatter des obigen Beispiels ergibt sich durch die Eingangssignalverläufe (L,R,H) und (H,F,L) der Ausgangssignalverlauf (L,C,L), wodurch der mögliche Hazard zum Ausdruck kommt (Abb. 4.12).

Darüberhinaus gliedert sich jeder einzelne Halbschritt nochmal in zwei Phasen (vgl. [2] S.83 f.). In der sogenannten Vorbereitungsphase wird nur ein Vorbereitungswert in den Verzögerungselementen verändert. Erst in der zweiten Phase, der Ausführungsphase werden die Ausgangssignalwerte neu gesetzt und die Nachfolgeelemente aktiviert. Dieses Vorgehen ist nötig, da für ein Verzögerungselement mehrere Ereignisse zum gleichen Zeitpunkt zur Ausführung kommen können.

Es ergeben sich also insgesamt vier Stufen, die mit V_I, A_I, V_{II} und A_{II} bezeichnet wer-

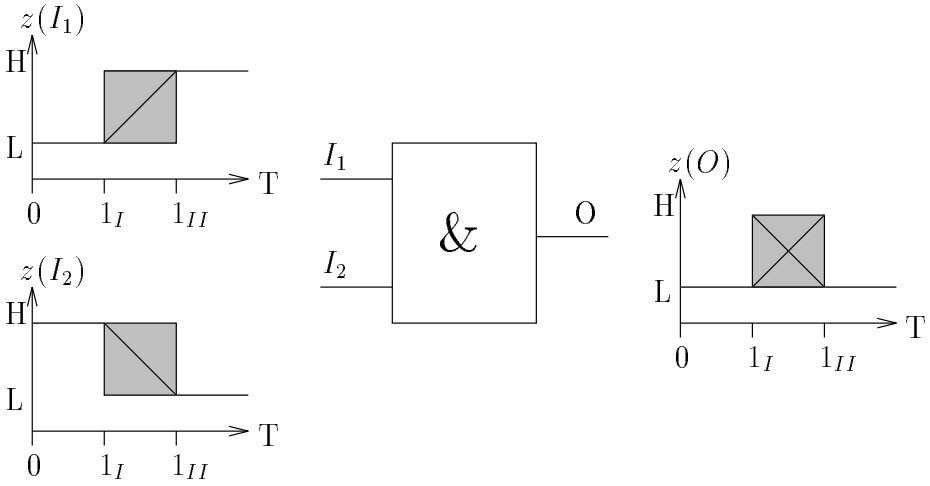


Abbildung 4.12: Erkannter Hazard

den. Die verkettete Liste der Ereignisse des aktuellen Zeitpunkts muß demnach viermal bearbeitet werden.

4.7.1 Normaler Ablauf

Der Simulator geht davon aus, daß bei einer Ereignisausführung ihm das nächste auszuführende Ereignis zurückgeliefert wird. Im Normalfall ist dies das nächste Element der verketteten Ereignisliste. Die viermalige Abarbeitung der gesamten Liste läßt sich somit steuern durch ein spezielles Ereignis (Klasse `SimEvCalc`) am Ende der verketteten Liste, welches bei dessen Ausführung das erste Ereignis der verketteten Liste zurückliefern kann (Abb. 4.13). Das Spezialereignis wird zu Beginn der Simulation eines Zeitpunktes erzeugt und befindet sich deshalb automatisch am Ende der verketteten Liste. Damit nun jedes Ereignis entscheiden kann in welcher Stufe der Simulation eines Zeitpunktes man sich gerade befindet, und welche Aktion deshalb durchzuführen ist, werden in allen Ereignissen Zähler eingeführt, die die Anzahl der Aufrufe mitzählen. Wird ein Ereignis zum ersten Mal aufgerufen, führt es die Aktionen der Vorbereitungsphase des ersten Halbschritts durch (V_1). Beim zweiten Aufruf wird Stufe A_1 durchgeführt, usw..

Das Steuerereignis am Ende der verketteten Liste dient darüberhinaus auch dazu, den Elementepuffer für Primitivauswertungen anzusteuern. Er muß jeweils am Ende von Stufe A_I und A_{II} zur Auswertung der Primitive des nächsten Ranges aufgefordert wer-

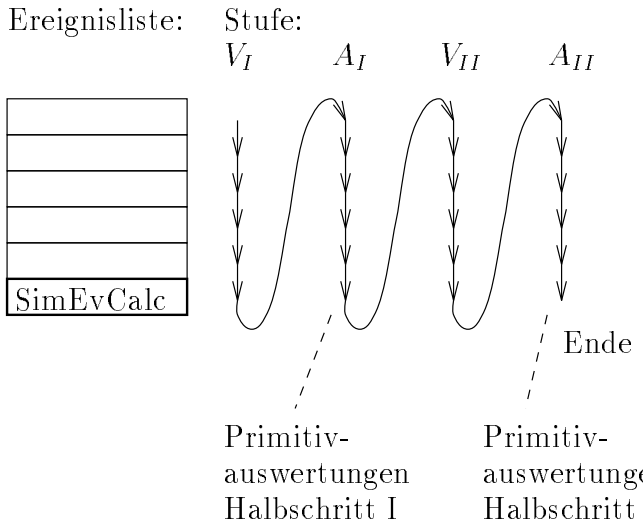


Abbildung 4.13: Vierfache Abarbeitung der Ereignisliste

den. Ist das Steuerereignis schließlich zum vierten Mal ausgeführt worden, ohne daß dabei neue Ereignisse entstanden sind, ist die Simulation diese Zeitpunktes abgeschlossen und das Steuerereignis liefert dem Simulatorkern kein als nächstes auszuführendes Ereignis mehr. Der Simulator selbst bemerkt von der Vierstufigkeit der Simulation eines Zeitpunktes nichts, er führt nur die Ereignisse in der Reihenfolge aus, die er geliefert bekommt.

4.7.2 Neue Ereignisse während Halbschritt I

Schwieriger gestaltet sich die Situation, wenn bei den Elementauswertungen des ersten oder zweiten Halbschritts neue Ereignisse für den aktuellen Zeitpunkt entstehen. Dies kann bei Null-Verzögerungen auftreten. Zunächst wird der Fall betrachtet, daß während Halbschritt I neue aktuelle Ereignisse entstanden sind (* in Abb. 4.14).

In diesem Fall muß das Spezialereignis der Klasse `SimEvCalc` die rangweise Elementauswertung beim nächsten Rang stoppen, ein neues Ereignis `SimEvCalc` am neuen Ende der verketteten Liste erzeugen, sich selbst deaktivieren (es führt bei zukünftigen Aufrufen keine Aktionen mehr durch) und die neuen Ereignisse zur Ausführung bringen. Diese führen nun ihrerseits zuerst wieder die Aktionen von Stufe V_1 aus. Beim neuen Ereignis `SimEvCalc` angekommen, darf dieses nun nicht das Ereignis des Listenkopfs zurückliefern, da hier Stufe A_I bereits ausgeführt wurde. Vielmehr muß beim Nachfolgeereignis

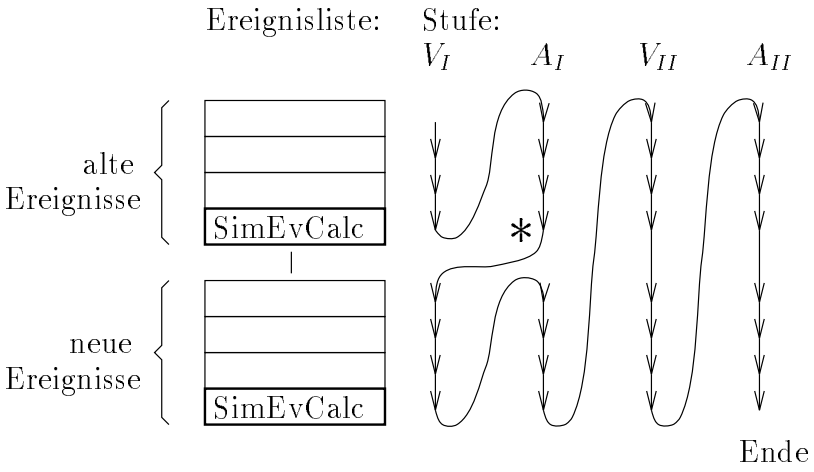


Abbildung 4.14: Neue Ereignisse bei Halbschritt I

des alten Spezialereignisses `SimEvCalc` mit Stufe A_I fortgefahren werden. Dazu besitzt `SimEvCalc` neben dem Zeiger auf den Listenkopf `top` einen weiteren Ereigniszeiger `loop`, der auf dieses erste neue Ereignis zeigt. Dieser wird zurückgeliefert. Wurde nun bei allen (alten und neuen) Ereignissen Stufe A_I ausgeführt, beginnt Stufe V_{II} wieder am Listenkopf. Darauf folgt Stufe A_{II} und die Simulation des Zeitpunktes ist beendet.

4.7.3 Neue Ereignisse während Halbschritt II

In Abbildung 4.15 ist die Situation wiedergegeben, die entsteht, wenn während der Primitivauswertungen des Halbschritts II neue Ereignisse entstehen (markiert durch ein *). In diesem Fall ist ebenfalls die rangweise Primitivauswertung zu stoppen, ein neues Ereignis `SimEvCalc` zu erzeugen und mit Stufe V_I beim ersten neuen Ereignis fortzufahren. Die Stufen V_I und A_I dienen hier allerdings nur noch dem Erhöhen der ereignisinternen Ausführungszähler, da während des Halbschritts II keine Ereignisse entstehen konnten, die Aktionen in Halbschritt I hervorrufen würden. Nach Beendigung von Stufe A_I ist nun im Gegensatz zum vorherigen Fall die Ereignisausführung nicht am Listenkopf fortzusetzen, sondern ebenfalls beim ersten neuen Ereignis.

Die beiden eben beschriebenen Szenarien (Abschnitte 4.7.2 und 4.7.3) dürfen selbstverständlich auch mehrfach und in Kombination auftreten.

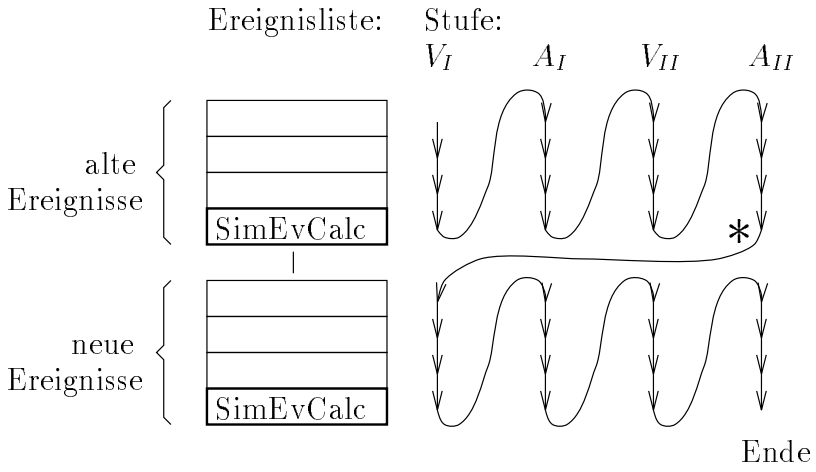


Abbildung 4.15: Neue Ereignisse bei Halbschritt II

4.8 Besondere Ereignisse

Zusätzlich zu den drei bereits vorgestellten Ereignisklassen `SimEvDiginp`, `SimEvDig` und `SimEvCalc` existieren vier weitere. Sie dienen zur Steuerung der Ergebnisprotokollierung und des Simulationsablaufs. Ereignisobjekte dieser neuen Klassen werden vor der Simulation aus der binären Stimulidatei erzeugt und in die Ereignisverwaltung eingefügt.

4.8.1 Vorzeitiger Simulationsabbruch

Mittels eines Ereignisses des Typs `SimLdsimExit` kann der Simulationsablauf vorzeitig abgebrochen werden. Dazu setzt dieses Ereignis bei seiner Ausführung ein Flag in der Ereignisverwaltung, welches ihr anzeigt, daß die noch gespeicherten Ereignisse für zukünftige Zeitpunkte nicht mehr an den Simulator geliefert werden sollen. Der Simulator beendet also den aktuellen Zeitpunkt noch ordnungsgemäß, bekommt anschließend von der Ereignisverwaltung keine neuen Ereignisse geliefert und beendet deshalb die Simulation.

4.8.2 Auswahl der zu protokollierenden Signale

Welche Signale bei der Simulation mittels Objekten der Klasse `SimRecDig` protokolliert werden sollen und welche nicht, läßt sich durch Ereignisse der Klassen `SimLdsimRespr` und `SimLdsimPrint` steuern.

Ein `SimLdsimRespr`-Ereignis setzt bei seiner Ausführung die Flags sämtlicher vorhandener `SimRecDig`-Topologieelemente auf Werte, die die Protokollierung dieser Signale beenden. Tritt ein solches Ereignis auf, werden ab diesem Zeitpunkt vorerst keinerlei Simulationsergebnisse mehr protokolliert.

Ein `SimLdsimPrint`-Ereignis dagegen setzt das Flag eines bestimmten `SimRecDig`-Topologieelements auf einen Wert, der die Protokollierung des zugehörigen Signalverlaufs ab dem Ereignisausführungszeitpunkt ermöglicht.

Grundsätzlich müssen für alle im Laufe der Simulation irgendwann zu protokollierenden Signale `SimRecDig`-Topologieelemente an der betreffenden Stelle bereits beim Topologieaufbau eingebaut werden, da dies nachträglich während der Simulation nicht mehr möglich ist.

4.8.3 Einfügen von Bytefolgen in den Ergebnisstream

In der binären Stimulidatei sind auch Steuerinformationen enthalten, die nicht für den Simulator bestimmt sind sondern für nachfolgende Programme zur Ergebnisvisualisierung. Die entsprechenden Bytefolgen sind in ein spezielles Kommando gekapselt (vgl. Kapitel 4.9.3). Es weist den Simulator an, diese Bytefolge zu einem bestimmten Simulationszeitpunkt in die binäre Ergebnisdatei einzufügen. Aus solch einem Kopierkommando wird beim Einlesen der binären Stimulidatei ein Ereignis des Typs `SimLdsimEvCopy` erzeugt, welches die betreffende Bytefolge zwischenspeichert und zum richtigen Zeitpunkt in den Stream der Simulationsergebnisse einfügt.

4.9 Simulationsvorbereitung

Die Simulationsvorbereitung gliedert sich grob in drei Schritte: Topologieaufbau, Schaltungsinitialisierung und Einlesen der Stimulis. Erst danach kann die eigentliche Simulation beginnen.

4.9.1 Topologieaufbau

Die Grundlage für den Aufbau der Schaltungtopologie ist die binäre Schaltungsbeschreibungsdatei, die auch der alte Simulator LDSIM verwendete. Sie besteht aus verschiedenen Listen, die LDSIM direkt einlas und zur Simulation verwendete. Für die Simulation mit dem neuen, hier vorgestellten Simulator müssen diese Listen interpretiert werden und daraus die nötigen Topologieobjekte und ihre Verbindungen aufgebaut werden. Tabelle 4.7 gibt die verschiedenen Listen und ihre Bedeutung wieder.

Listenname	Inhalt
Kopf	Anzahl der Primitive, Signale und Ränge; Minimale und maximale Verzögerung; Listenlängen
Modulliste	Primitive: Nummer, Typ, Ausgangsverzögerungen, Nummern der angeschlossenen Signale
Fanoutliste	Fanoutbäume aller Signale
Netzliste	Zuordnung zwischen Signalnummern und Fanoutbäumen
Elementliste	Zeiger auf die einzelnen Primitive in der Modulliste
Outputliste	Angabe der Signalquellen (Primäreingang, Primitive)
Levelliste	Anzahl der Primitive eines Rangs
Rangliste	Ränge aller Primitive

Tabelle 4.7: Listen der binären Schaltungsbeschreibung

Nach dem Einlesen des Kopfs der Schaltungbeschreibung werden zuerst aus der Modulliste die Objekte für alle Primitive, Verzögerungen, Oszillationsüberwachungselemente und Ergebnisrekorder erzeugt. Diese lassen sich untereinander in obiger Reihenfolge verbinden.

Anschließend werden die Objekte für die Primäreingänge, Oszillationszählerrücksetzung und Rekorder der Eingangssignale unter Verwendung der Outputliste erzeugt und ebenfalls untereinander verbunden.

Im nächsten Schritt werden mittels zwischenzeitlich erzeugter Hilfsvariablen die Objekte zur Darstellung der Fanoutbäume generiert und mit den bisher erzeugten Ergebnisrekordern verbunden.

Als letzter Schritt des Topologieaufbaus werden alle Zweige der Fanoutbäume mit den entsprechenden Primitiveingängen verbunden. Dies geschieht wieder mittels der Modulliste.

Abschließend werden noch die Ränge aller Primitive durch Einlesen der Rangliste festgelegt und der Primitivpuffer mittels der Levelliste mit nötigen Parametern versorgt.

```
INPUTS
A B
END

OUTPUTS
D E
END

TYPES
AND2 (DEL1: 0 0 0 0 S) and.
NOR2 (DEL1: 1 1 1 1 N) nor.
INV (DEL1: 3 4 3 4 E) inv.
END

CONNECTELE
and ( D A , C )
nor ( C B , D )
inv ( B , E )
END
```

Abbildung 4.16: Schaltungsbeschreibung

Aus der in den Abbildungen 4.16 und 4.17 wiedergegebenen Schaltung entsteht so die in Abbildung 4.18 dargestellte Schaltungtopologie im Simulator. Als Besonderheit ist anzumerken, daß die Verzögerungszeiten des AND-Gatters alle Null sind und deshalb kein entsprechendes Verzögerungselement in der Topologie eingebaut wird. Außerdem ist nur für das Eingangssignal B ein Fanoutbaum-Element nötig, da es sich bei den anderen Signalen um Zweipunktverbindungen handelt.

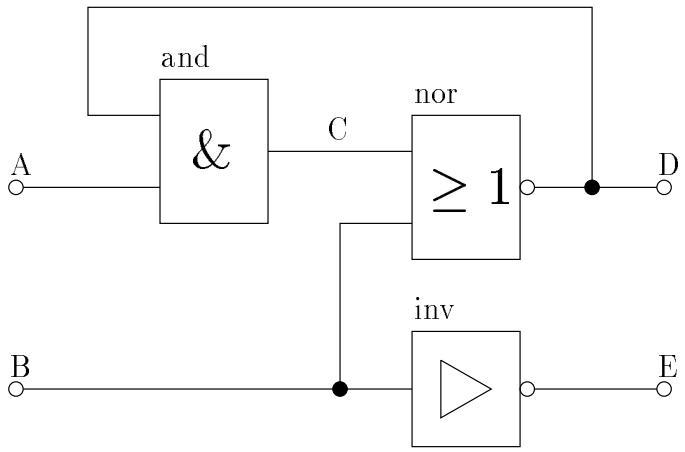
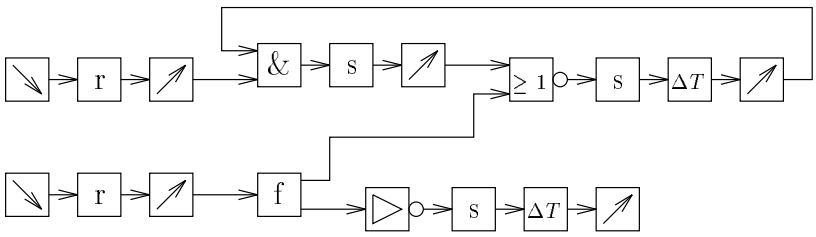


Abbildung 4.17: Schaltplan



Legende:

Symbol:	Klasse:	Symbol:	Klasse:
↘	SimInDig	&	and
↗	SimRecDig	≥ 1 ○	nor
r	SimOscReset	▷ ○	inv
s	SimOscSense	ΔT	SimDelDig
		f	SimDist

Abbildung 4.18: Topologie im Simulator

4.9.2 Schaltungsinitialisierung

Die Initialisierung der Schaltung findet auf zwei Wegen statt. Einerseits sind in der Datei der binären Schaltungsbeschreibung Initialzustände für die sequentiellen Primitive enthalten. Sie werden beim Topologieaufbau bereits berücksichtigt.

Andererseits sind in der Stimulidatei Signalwertwechsel und Steuerkommandos für den Simulationszeitpunkt Null enthalten, welche zur Schaltungsinitialisierung bestimmt sind. Ihre Codierung ist in Kapitel 4.9.3 beschrieben. Dabei können sämtliche Signale der Schaltung gesetzt werden, während nach der Initialisierungsphase nur noch Veränderungen an Primäreingängen erlaubt sind.

Durch die Möglichkeit alle internen Signale mit einem Initialwert zu versehen, kann ein inkonsistenter Schaltungszustand entstehen. Deshalb sind nach dem Setzen der Signalwerte alle Primitive einmal auszuwerten. Entstehen dadurch neue Ereignisse, lag eine Inkonsistenz vor und die Schaltung wird weitersimuliert, bis keine neuen Ereignisse mehr vorliegen. Die dabei entstehenden Signalwertwechsel werden nicht protokolliert. Anschließend wird die Simulationszeit auf Null zurückgesetzt und die Initialisierung der Schaltung ist abgeschlossen.

4.9.3 Einlesen der Stimulis

Nach der Schaltungsinitialisierung müssen nun die restlichen Stimulis (Simulationszeit > 0) eingelesen und daraus die nötigen Ereignisse generiert werden. Diese werden der Ereignisverwaltung übergeben.

Die verwendete LDSIM-Stimulidatei besteht aus dreierlei Bestandteilen: Zeitmarker, Signalwertwechsel und Kommandos. Ein Zeitmarker ist daran erkennbar, daß er aus einer negativen, vier Byte langen Integerzahl besteht. Er bestimmt die Ausführungszeit aller nachfolgenden Signalwertwechsel und Kommandos.

Ein Signalwertwechsel ist codiert in zwei Integerzahlen. Die erste Zahl stellt die positive Signalnummer dar, während die zweite den entsprechend Tabelle 4.2 codierten Signalwert beinhaltet.

Die letzte Kategorie (Kommando) wird durch die Integerzahl Null eingeleitet. Die zweite Zahl legt den Kommandotyp fest (vgl. Tabelle 4.8). Es können je nach Typ weitere Zahlen folgen. Einige Kommandos sind für den normalen Simulationsbetrieb nicht erforderlich und werden daher nicht mehr unterstützt.

Die Kommandos, welche die Aufteilung der Stimulidatei (und Ergebnisdatei) in Blöcke steuern, werden nicht vom Simulator direkt unterstützt. Sie lassen sich durch das Filterprogramm `pipein`, welches die Blockstruktur auflöst, entfernen. Genauso ist es durch ein anderes Filterprogramm (`pipeout`) möglich, der Ergebnisdatei eine Blockstruktur zu geben, um sie mit anderen Programmen des LDSIM-Pakets weiterverarbeiten zu können.

Code	Name	Verarbeitung / Bedeutung
0	EXIT	Simulation beenden (Klasse SimLdsimEvExit)
1	READ	Wird von `pipein` verarbeitet bzw. von `pipeout` erzeugt
2	GO	Wird von `pipein` verarbeitet
3	COPY	Bytefolge in die Ergebnisdatei kopieren (Klasse SimLdsimEvCopy)
4	RESSIMPRINT	Protokollierung aller Signale beenden (Klasse SimLdsimEvRespr)
5	SIMPRINT	Protokollierung eines Signals beginnen (Klasse SimLdsimEvPrint)
6	PRINT	Wird nicht direkt unterstützt, da für die Ergebnisvisualisierung bestimmt
7	DELETE	Wird nicht direkt unterstützt, da für die Ergebnisvisualisierung bestimmt
8	SAVE	Wird nicht unterstützt
9	RECOVER	Wird nicht unterstützt
10	INIT	Kennzeichnung des Beginns der Ergebnisdatei
11	READY	Wird von `pipein` verarbeitet

Tabelle 4.8: Kommandos der Stimuli- und Ergebnisdatei

Kapitel 5

Ausblick

Der vorliegende objektorientierte Simulator kann als Ausgangspunkt für weitere Arbeiten dienen. Im Bereich der Logiksimulation ist zum Beispiel eine Parallelisierung denkbar. Dazu wäre es notwendig, Topologieelemente zu entwerfen, die Partitionsschnittstellen darstellen und für die Kommunikation zwischen den Simulatoren sorgen. Außerdem müßte die Ereignisverwaltung erweitert werden, um Rollbacks zu ermöglichen.

Das Grundkonzept des Simulators (Abbildung der Topologie, Ereignisverwaltung) läßt sich darüberhinaus auch für andere Problemstellungen neben der Logiksimulation verwenden. Denkbar sind Anwendungen in allen Bereichen der ereignisgesteuerten Simulation. Die Topologieelemente und Ereignisse müssen dann gemäß den jeweiligen Anforderungen gestaltet werden.

Literatur

[1] **Kurt Antreich**: *Rechnergestützte Layout- und Testverfahren.* Vorlesungsskript, Lehrstuhl für Rechnergestütztes Entwerfen, Technische Universität München.

[2] **Thomas Krodel**: *Verfahren zur Logiksimulation komplexer digitaler Schaltungen mit flexibler Modellierung.* Dissertation, Lehrstuhl für Rechnergestütztes Entwerfen, Technische Universität München, 1989.

[3] **Thomas Krodel**: *Beschreibung des Logic Design Simulators ldsim.* Lehrstuhl für Rechnergestütztes Entwerfen, Technische Universität München, 1989.

[4] **Michael Hermann, Christian Sporrer**: *Logiksimulation am Beispiel des Entwurfs eines einfachen Mikroprozessors.* Praktikumsanleitung, Lehrstuhl für Rechnergestütztes Entwerfen, Technische Universität München, 1990.

[5] **Helmut Becker**: "Objektorientierte Software Entwicklung". - In: *Informationstechnik*, R. Oldenbourg Verlag, Februar 1992, S. 92-101.

[6] **Timm Grams**: "Denkfallen beim objektorientierten Programmieren". - In: *Informationstechnik*, R. Oldenbourg Verlag, Februar 1992, S. 102-111.

[7] **R.S. Wiener, L.J. Pinson**: *An Introduction to Object-Oriented Programming and C++.* Reading, Massachusetts: Addison-Wesley, 1988.

[8] **Ira Pohl**: *C++ for C Programmers.* Redwood City, California: Benjamin / Cummings, 1989.

[9] **Roman Gericke**: "Crash-Kurs in C++, Schnelleinstieg für Programmierer". - In: *c't Magazin für Computertechnik*, Hannover: Verlag Heinz Heise GmbH & Co KG, Oktober 1991 - Januar 1992.

[10] **H.H. MacDougall**: *Simulating computer systems: techniques & tools.* Massachusetts: MIT Press, 1989.

[11] **Niklaus Wirth**: *Algorithms + Data Structures = Programs.* New Jersey: Prentice-Hall, 1975.

Anhang A

Klassenhierarchien

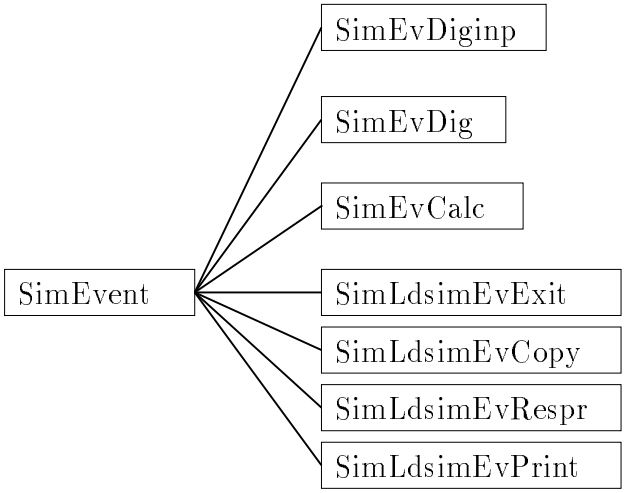


Abbildung A.1: Klassenhierarchie der Ereignisse

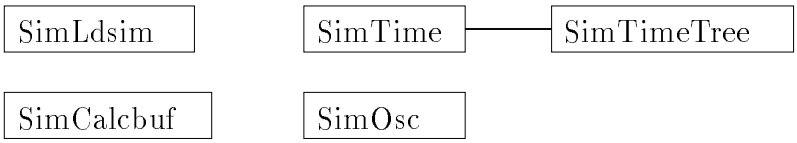


Abbildung A.2: Sonstige Klassen

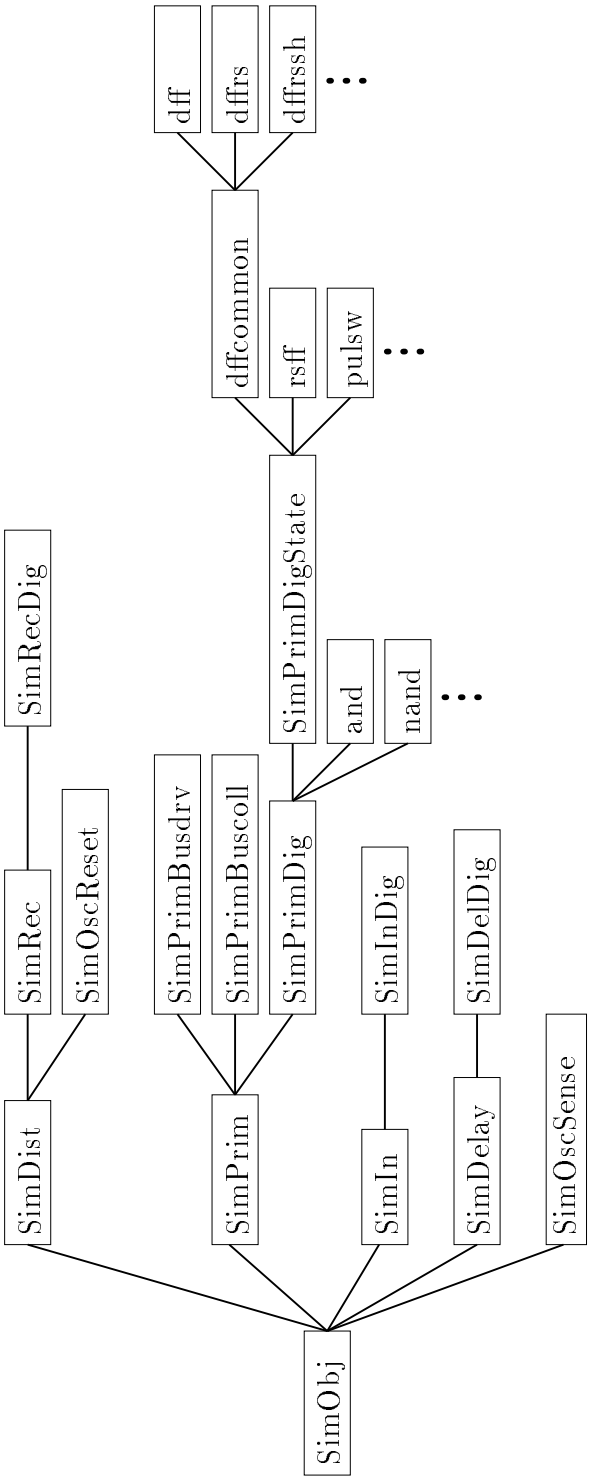


Abbildung A.3: Klassenhierarchie der Topologieelemente

Anhang B

Primitive

LDSIM-Primitiv	Topologieelement-Klasse, abgeleitet aus SimPrimDig
AND2, AND3, AND4, AND5, AND8, ANDN	and
OR2, OR3, OR4, OR5, OR8, ORN	or
NAND2, NAND3, NAND4, NAND5, NAND8, NANDN	nand
NOR2, NOR3, NOR4, NOR5, NOR8, NORN	nor
EXOR2, EXORN	exor
EXNOR2, EXNORN	exnor
INV	inv
BUF, DEL	buf
ADD	add
ADDC	addc
MULT	mult
MULTC	multc
BUS, BUSC	bus (unter Verwendung von SimPrimBusdrv, Simprim-Buscoll)

Tabelle B.1: Liste der kombinatorischen Primitive

LDSIM-Primitiv	Klasse	abgeleitet aus
DFF	dff	dffcommon
DFFR	dffr	
DFFDR	dffdr	
DFFDRS	dffdrs	
DFFRSSH	dffrssh	
DFFDRSSH	dffdrssh	
LATCH	latch	SimPrimDigState
LATCHSP	latchsp	
RSFF	rsff	
RSFFSP	rsffsp	
MSFF	msff	
MSFFD	msffd	
MSFFDSP	msffdsp	
PULSW	pulsw	
SETUP	setup	

Tabelle B.2: Liste der Primitive mit internen Zuständen

	Data					
	H	L	U	F	R	C
Enable H	2022	1021	23	27	33	37
L	0	0	0	0	0	0
U	2	1	3	7	13	17
F	42	101	143	147	153	157
R	402	201	603	607	613	617
C	442	301	743	747	753	757

Tabelle B.3: Verknüpfungstabelle Bustreiber