Flexible Visualisierung von Entwurfsdaten in CADIC

Jörg Ritter

Diplomarbeit nach einem Thema von Prof. Dr. Günter Hotz
innerhalb der technischen Fakultät, Fachbereich 14 - Informatik,
der Universität des Saarlandes, Saarbrücken

April 1997

Inhaltsverzeichnis

Einleitung

Die Anwendung der elektronischen Datenverarbeitung in Wissenschaft, Industrie und Verwaltung wird unter anderem dadurch gekennzeichnet, daß sie sich von einem anfangs nur beiläufig genutzten Hilfsmittel nach und nach weiterentwickelt hat zu einem Instrument, von dem wir für manche Tätigkeiten ganz und gar abhängig geworden sind. Schrittmacher dieser Entwicklung ist die Herstellung von immer leistungsfähigeren Schaltkreisen. Mittlerweile können nicht nur Mikroprozessoren und Speicher in großen Stückzahlen, sondern auch anwendungsspezifische Schaltkreise rentabel produziert werden. Grundlage dafür ist die Fertigung von immer feineren Strukturen auf den Chips und die gestiegene Ausbeute einerseits sowie drastisch gesunkene Entwicklungskosten durch rechnergestützten Entwurf andererseits.
Die Möglichkeit, höchstintegrierte Schaltkreise für einen ganz speziellen Einsatzzweck herstellen zu können, eröffnet ein breites Spektrum neuer Anwendungsgebiete und zieht einen entsprechend hohen Bedarf an solchen Chips nach sich.

Daraus ergibt sich die Notwendigkeit, den Entwurf von Schaltkreisen auch Nichtspezialisten zugänglich zu machen. Entwurfssysteme, die ausgehend von der Schaltungsbeschreibung alle notwendigen Daten zur Fertigung erzeugen können, sind allerdings auf relativ kleine oder sehr regelmäßige Schaltungen beschränkt.
Der Entwurf komplexer Schaltkreise wird deshalb in verschiedene Teilaufgaben zerlegt. Zu jedem der Teilschritte korrespondieren Entwurfswerkzeuge, mit Hilfe derer ein Designer die zugehörige Teilaufgabe (meist interaktiv) bearbeiten kann. Problematisch ist die Ausführung mehrerer solcher Werkzeuge hintereinander, da diese im allgemeinen auf verschiedenen Schaltkreissichten und Datenformaten arbeiten. Daraus ergibt sich die Forderung nach Entwurfsumgebungen, welche den Einsatz verschiedener Werkzeuge innerhalb eines Systems ermöglichen.

An der Universität des Saarlandes wurde im Rahmen der Sonderforschungsbereiches 124 "VLSI-Entwurf und Parallelität" das Schaltkreisentwurfssystem `CADIC` entwickelt. Dieses System verfügt über eine konfigurierbare Arbeitsumgebung mit graphischer Benutzeroberfläche und ist nach dem Prinzip einer Werkbank aufgebaut. Es können also je nach Bedarf verschiedene Werkzeuge eingesetzt werden. Ist der Benutzer einmal mit den grundlegenden Eigenschaften der Werkbank vertraut, wird die Akzeptanz neuer Werkzeuge verbessert.

Die hier vorgestellte Arbeit befaßt sich mit der Generierung und der Konfiguration ausdrucksstarker Methoden zur Visualisierung von Entwurfsdaten in der graphischen

Arbeitsumgebung von `CADIC`. Mit Hilfe dieser Visualisierung können die berechneten Entwurfsdaten der Werkzeuge flexibel graphisch aufbereitet und an der Schaltungsbeschreibung angezeigt werden. Dazu stehen unter anderem textuelle Anzeigeformen, Punkt- und Liniendiagramme, Histogramme und farbliche Gestaltungsmöglichkeiten zur Verfügung.
Der Benutzer von `CADIC` kann somit den Ablauf der verschiedenen Werkzeuge besser verfolgen und die Resultate direkt am Entwurf der Schaltung ablesen.
Durch optisch ansprechende Visualisierung können nach Anwendung eines Entwurfswerkzeugs direkt die Bereiche der Schaltung, wo gute bzw. schlechte Ergebnisse erzielt wurden, lokalisiert werden. Der Benutzer wird so in die Lage versetzt, interaktiv in den Entwurfsprozeß einzugreifen, um zum Beispiel die zugrundeliegende Schaltung zu modifizieren oder das Werkzeug mit neuen Vorgaben zu starten.

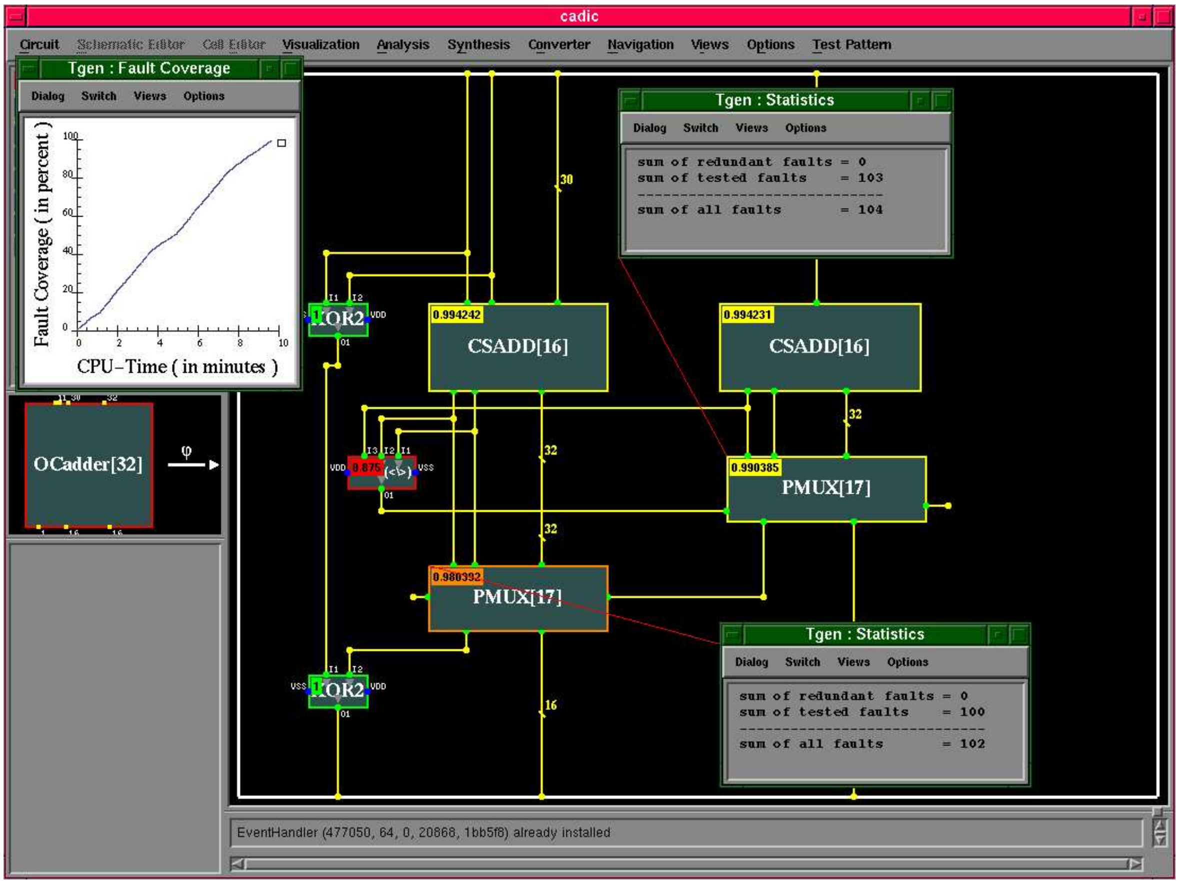

Abbildung 0.1: Visualisierung von Entwurfsdaten während der Testmustergenerierung

Der Leser kann sich einen Eindruck von den vielfältigen Visualisierungsformen anhand der Abbildung 0.1 verschaffen. Gezeigt wird hier eine Konfiguration bei der Anzeige von Informationen während der Testmustergenerierung. In dem Diagramm ist die Fehlerüberdeckungskurve für den Gesamtschaltkreis zu sehen. An jedem Zellvorkommen kann man die prozentuale Fehlerüberdeckung direkt im Schaltkreisbild ablesen.

Die an den Zellvorkommen geöffneten Dialoge (durch ein rotes Gummiband erkennt man die Zugehörigkeit) zeigen die Anzahl der bereits als redundant erkannten, der bereits getesteten und die Gesamtzahl der eingepflanzten Fehler in diesem Modul. Durch zusätzliche Einfärbung entsprechend der bisher berechneten Fehlerüberdeckung (rot–schlecht, orange/gelb–mittel, grün–in Ordnung, d.h. $\geq$99,5%) kann man sofort Bereiche mit schlechten Ergebnissen lokalisieren. Dies wird durch das Zusammenwirken von Visualisierung und Navigation durch die Schaltungshierarchie noch verbessert. Beim Betreten einer neuen Hierarchieebene werden die dort aktuellen Informationen angezeigt und die Daten der verlassenen Ebene ausgeblendet.

Der Benutzer nimmt bei der Visualisierung aber nicht nur eine passive Rolle ein. Vielmehr besteht die Möglichkeit, aktiv auf die Anzeige der Entwurfsdaten Einfluß zu nehmen. Beispielsweise kann man zwischen der Darstellung verschiedener Informationen umschalten, Dialoge öffnen und Einfärbungen bzw. direkt an die Schaltung geheftete Entwurfsdaten an- bzw. abschalten. Ist der Benutzer einmal mit den Bedienungs- und Einstellungsmöglichkeiten vertraut, kann er diese bei allen integrierten Werkzeugen anwenden.

Für den Werkzeugentwickler bietet die hier vorgestellte Visualisierung ebenfalls entscheidende Vorteile. Bisher mußte der Entwickler eines in die `CADIC`-Oberfläche zu integrierenden Werkzeugs, eigene Funktionen zur Anzeige seiner Entwurfsdaten erstellen. Durch den Einsatz der Visualisierung kann man dem Werkzeugentwickler diese Aufgabe weitgehend abnehmen, wodurch die Integration erheblich beschleunigt wird. Seine Aufgabe besteht nur noch darin, die anzuzeigenden Informationen vom Entwurfswerkzeug an die Visualisierung zu schicken und zu beschreiben, wie diese Daten graphisch aufbereitet werden sollen.
Ein besonderer Vorteil ist dabei die klarere Abgrenzung zwischen den Entwurfswerkzeugen und der graphischen Oberfläche, wodurch die Modularität von `CADIC` verbessert wird.

Als Erweiterung zur bisherigen direkten Integration von Entwurfswerkzeugen wird durch die Visualisierung auch die Anbindung von Werkzeugen, die als eigenständige Programme realisiert sind, ermöglicht. Diese Erweiterung erlaubt eine interaktive Steuerung von externen Programmen nach dem Client-Server-Modell. Dabei können die graphische Oberfläche und das externe Werkzeug auf verschiedenen Rechnern arbeiten, falls diese durch ein Netzwerk verbunden sind.
Die so angebundenen Programme können ihre Informationen analog zu den direkt integrierten Werkzeugen visualisieren. Voraussetzung ist, daß der Quellcode des externen Programms vorliegt. Anderenfalls können nur die berechneten Ergebnisse aufbereitet und verschickt werden. Arbeiten die eigenständigen Entwurfswerkzeuge nicht auf der `CADIC`-Datenstruktur, ist eine Abbildung zwischen den verwendeten Strukturen notwendig.
Realisiert wurde eine solche Abbildung für die Datenstruktur `test_circ`, welche speziell für Werkzeuge zum Testen von Schaltkreisen entwickelt wurde.

Gliederung der Arbeit

In Kapitel 1 wird zunächst das Schaltkreisentwurfssystem `CADIC` vorgestellt und die in der Arbeit verwendeten Begriffe und Schreibweisen definiert. Im Anschluß daran beschreibt Kapitel 2 die dem Werkzeugentwickler zur Verfügung gestellten Darstellungsformen seiner Entwurfsdaten. Das dritte Kapitel skizziert die Vorgehensweise zur Konfiguration der Visualisierung und beschreibt die Schnittstelle zu den Entwurfswerkzeugen. Weiterhin wird der Leser mit den Bedienungs- und Einstellungsmöglichkeiten zur Handhabung der Visualisierung vertraut gemacht. Im vierten Kapitel wird die Erweiterung der Schnittstelle zu den Entwurfswerkzeugen vorgestellt, welche eine interaktive Steuerung von externen Programmen nach dem Client-Server-Modell erlaubt. Kapitel 5 beschäftigt sich mit der Weiterverarbeitung der Entwurfsdaten innerhalb der Visualisierung. Darunter fällt insbesondere die Berechnung statistischer Informationen und die Interpretation von numerischen Werten als Index einer Farbskala. Die Konfiguration der Anzeige für mehrere Entwurfswerkzeuge gleichzeitig wird in Kapitel 6 behandelt. Dazu wird aus den lokalen Beschreibungen durch geeignetes "Zusammenmischen" eine Gesamtkonfiguration erzeugt. Im abschließenden Kapitel 7 wird anhand von ausführlichen Beispielen die Integration von Entwurfswerkzeugen erläutert.

Danksagungen

Mein Dank gilt zunächst Herrn Prof. Dr. Günter Hotz für die interessante Aufgabenstellung. Besonders möchte ich mich bei Herrn Dr. Thomas Burch und Herrn PD Dr. Uwe Sparmann für die sorgfältige Betreuung meiner Arbeit bedanken. Schließlich gilt mein Dank allen Mitarbeitern und Kommilitonen am Lehrstuhl, die mich bei der Implementierung durch Anregungen und mit Hilfestellungen unterstützt haben.

Kapitel 1

Grundlagen

Das Schaltkreisentwurfssystem `CADIC` und die zugehörige graphische Arbeitsumgebung bilden die Grundlage für die vorliegende Arbeit. Die hier beschriebene Visualisierung stellt eine Erweiterung der graphischen Arbeitsumgebung dar. Durch geeignete Visualisierung von Entwurfsdaten soll die Integration von Werkzeugen erleichtert werden.
In diesem Kapitel wird der Leser zunächst mit dem Schaltkreisentwurfssystem `CADIC` und der zugehörigen graphischen Arbeitsumgebung vertraut gemacht, soweit dies zum Verständnis der Arbeit nötig ist. Im Anschluß werden die grundlegenden Konzepte und Vorgehensweisen bei der Visualisierung sowie die für diese Arbeit notwendigen Begriffe und Schreibweisen vorgestellt.

1.1 Das Schaltkreisentwurfssystem `CADIC`

Ein Großteil der heute entwickelten Schaltkreise besitzt eine sehr regelmäßige Struktur und läßt sich daher mittels rekursiver Gleichungen beschreiben. `CADIC`, ein Schaltkreisentwurfssystem, das an der Universität des Saarlandes im Rahmen der Sonderforschungsbereiches 124 "VLSI-Entwurfsmethoden und Parallelität" entwickelt wurde, erlaubt die Spezifikation von Schaltkreisen durch rekursive Gleichungen.
Diesem Entwurfssystem liegt ein wohldefinierter mathematischer Kalkül zugrunde [Hot65], [Hot74]. Aufbauend auf diesem Kalkül werden Schaltkreise durch *logisch-topologische Netze* beschrieben [Mol86], [Mol88]. Solche Netze bestehen aus Bausteinen und Verbindungen zwischen diesen in der Ebene. Überkreuzung und Abzweigungen sind ebenfalls Bausteine. Operatoren sind das Übereinandersetzen und das Nebeneinandersetzen zweier logisch-topologischer Netze.
Die Spezifikation von Schaltkreisen erfolgt durch rekursive Netzgleichungen [Kol86], indem aus einer Menge von Grund- und Verdrahtungsbausteinen und den Operatoren neue logisch-topologische Netze gebildet werden. Die durch die Netzgleichungen gebildeten Teilstrukturen können nun mehrfach referenziert werden. Dadurch erhält man die Möglichkeit zur hierarchischen Beschreibung von Schaltkreisen. Die Netzgleichungen lassen sich parametrisieren, so daß mit wenigen Gleichungen eine ganze Familie von Schaltungen beschrieben werden kann. Dabei sind beliebig viele Parameter bei

Verbindungen als auch bei Bausteinen erlaubt.

Nachteilig bei der Beschreibung mit Hilfe von Netzgleichungen ist die textuelle Eingabe der mathematischen Ausdrücke. Schon bei relativ kleinen Schaltungen wird die Darstellung durch Netzgleichungen unübersichtlich und ist anfällig für Schreibfehler.
Zum Erstellen eines solchen Gleichungssystems wird man jedoch zunächst ein Bild der Schaltung malen und anhand dessen die Schaltung gemäß den Operatoren zerlegen. Durch Verwendung eines graphischen Editors kann dieser Prozeß automatisiert werden. Ein solcher Editor wurde in [Bur88] vorgestellt. Mit Hilfe des dort beschriebenen Editors können Schaltkreise als Bild eingegeben und Netzgleichungen erzeugt werden. Die Auswertung der von den Entwurfswerkzeugen berechneten Ergebnisse ist jedoch durch den Übergang von der graphischen zur textuellen Beschreibung wieder problematisch.
Um die Anbindung der anderen Werkzeuge an die graphische Beschreibung zu verbessern, wurde eine allen Entwurfswerkzeugen zugrundeliegende Datenstrukutur und eine Benutzeroberfläche entwickelt [Bur95]. Diese Arbeitsumgebung bildet die Basis für die hier vorliegende Arbeit.

1.2 Die graphische Entwurfsumgebung von CADIC

Die graphische Entwurfsumgebung von CADIC ist nach dem Prinzip einer Werkbank aufgebaut. Je nach Bedarf können Werkzeuge zur Analyse, zur Synthese und zum Testen von Schaltungen integriert werden. Als elementares Werkzeug enthält jede Konfiguration einen *Schematic Editor*. Mit Hilfe dieses Editors können Schaltkreisspezifikationen erstellt, modifiziert und abgespeichert werden. Abbildung 1.1 zeigt die graphische Benutzeroberfläche von CADIC, basierend auf dem kommerziellen Graphikpaket *OSF/Motif* [HF94]. Im Arbeitsfenster ist eine entworfene Netzgleichung für ein Sortiernetzwerk nach dem Odd-Even-Merge-Prinzip dargestellt.
Dieser Entwurf ist rekursiv und parametrisiert in Leitungen und Bausteinen beschrieben.
Zur Bearbeitung des Entwurfs mit den integrierten Werkzeugen, muß die gesamte Spezifikation eventuell durch Angabe von Parametern geladen werden. Nachdem das entstandende Gleichungssystem aufgelöst wurde, sind alle Parameter an Leitungen und Bausteinen mit den entsprechenden Werten belegt. Die gesamte, aus dem Gleichungssystem resultierende Schaltungsbeschreibung wird intern in einer zentralen Datenstruktur gespeichert, die einem gerichtetem azyklischen Graphen mit Rückverweisen entspricht. Jede verwendete Teilschaltung korrespondiert zu einem Knoten des Graphen. Verschiedene Vorkommen eines Teilschaltkreises verweisen auf diesen Knoten. Als Grund- oder Basiszellen werden Bausteine aus einem zugrundegelegten Zellkatalog verwendet. Damit abstrahiert man von konkreten physikalischen Realisierungen und kann verschiedene Fertigungstechniken nutzen. Eine aus den Basiszellen und entsprechenden Verdrahtungskanten und -knoten entworfene Teilschaltung bildet eine *Makrozelle*, die im folgenden wie eine Grundzelle verwendet werden kann. Die Vorkommen von Basis- oder

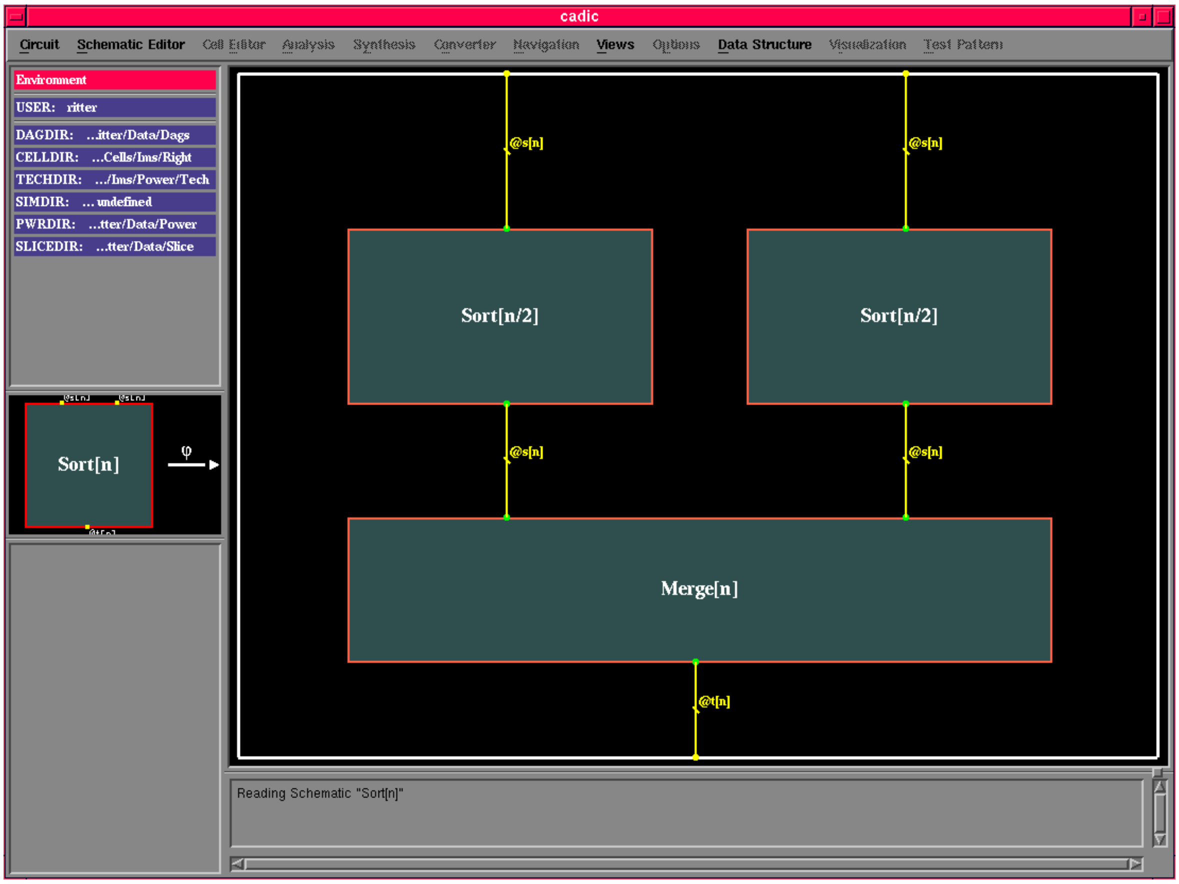

Abbildung 1.1: Die graphische Arbeitsumgebung von CADIC

Makrozellen bezeichnet man als *Instanzen*.

Nachdem eine Schaltungsspezifikation insgesamt geladen wurde (eventuell nach Angabe der Parameter), wird zunächst deren oberste Hierarchiestufe angezeigt. Auf der linken Seite ist der geladene Entwurf mit seiner äußeren Anschlußbelegung und im Arbeitsfenster die zugehörige Beschreibung zu sehen.

Funktionen zur Navigation erlauben das Durchmustern der Hierarchie durch Auswählen und Anklicken einer Bausteininstanz (*Trace - Down*) bzw. durch Mausklick im Fenster mit der äußeren Anschlußbelegung (*Trace - Up*).

Die Schaltkreisdarstellung im Arbeitsfenster kann skaliert und gescrollt werden.

Neben dem *Schematic Editor* besteht die Möglichkeit, andere Entwurfswerkzeuge in die Arbeitsumgebung einzubinden. Zur Integration eines Entwurfswerkzeugs kann die Arbeitsumgebung entsprechend konfiguriert werden. Dazu wird mit Hilfe einer Menubeschreibungssprache *magics* [Bur95] die Bedienung der graphischen Oberfläche um die für das Werkzeug erforderliche Ansteuerung erweitert. Beim Start eines so integrierten Entwurfswerkzeugs erhält dieses als Eingabe den geladenen, hierarchisch gegliederten Schaltkreis und falls notwendig, zusätzliche Parameter. Zur Eingabe der Parameter oder auch zur interaktiven Abfrage von Benutzerdaten stehen vorgefertigte (und teil-

weise konfigurierbare) Dialogmasken zur Verfügung.
Die Bereitstellung von Schnittstellenfunktionen und die Konfiguration der graphischen Oberfläche allein reichen jedoch nicht aus, um die Werkzeuge vollständig zu integrieren, da die berechneten Resultate nicht an der Schaltung ausgewertet werden können. Dazu benötigt man eine auf das jeweilige Entwurfswerkzeug zugeschnittene graphische Aufbereitung der berechneten Zwischenergebnisse und Resultate. Die Flexibilität bei der Integration der Werkzeuge hängt also in entscheidender Weise von den vorhandenen Visualisierungsmöglichkeiten ab.

1.3 Flexible Visualisierung von Entwurfsdaten

Gegenstand dieser Arbeit ist die Erweiterung der graphischen Arbeitsumgebung von `CADIC` um ein Modul Visualisierung. Dieses Modul soll in der Lage sein, die berechneten Ergebnisse von Entwurfswerkzeugen graphisch aufzubereiten und an der Benutzeroberfläche anzuzeigen.
Die Visualisierung kann somit kein statischer Teil der Arbeitsumgebung sein, da jedes Werkzeug andere Anforderungen an die Darstellung der berechneten Entwurfsdaten stellt. Flexibilität ist auch bezüglich der den Entwurfswerkzeugen zugrundeliegenden Schaltkreissichten erforderlich, um dem Werkzeugentwickler das Implementieren von Konvertierungen zu ersparen. Weiterhin sollen auch die Designer von Schaltkreisen Einfluß auf die Gestaltung der Anzeige haben.
Ziel ist es also, eine konfigurierbare Visualisierung mit ausdrucksstarken Methoden zur Verfügung zu stellen.

Zunächst soll die grundlegende Vorgehensweise zur Anzeige der Informationen skizziert werden.
Zur Integration eines Werkzeuges beschreibt dessen Entwickler, wie die Anzeige seiner Entwurfsdaten erfolgen soll. Dazu muß er den Typ der Informationen sowie die Art und Weise der Darstellung festgelegen. Dies wird über eine noch zu definierende Beschreibungssprache ermöglicht (siehe Kapitel 3).
Während des Programmablaufs werden dann die anzuzeigenden Informationen mit Hilfe von Schnittstellenfunktionen vom Entwurfswerkzeug an das Modul Visualisierung geschickt. Die Entwurfsdaten werden dort kopiert und nach den Vorgaben des Werkzeugentwicklers zur Anzeige gebracht.
Durch diesen Ansatz verbessert sich der modulare Aufbau der graphischen Entwurfsumgebung. Desweiteren wird eine interaktive Steuerung der Werkzeuge ermöglicht, da durch geeignete Kommunikation zwischen der Benutzeroberfläche und dem Werkzeug dessen Ablauf und somit auch die Anzeige der Informationen kontrolliert werden kann. Erweitert man das Verschicken der Entwurfsdaten, so daß diese über ein Netzwerk übertragen werden können, erhält man die Möglichkeit zur Anbindung externer Entwurfswerkzeuge an `CADIC` nach dem *Client-Server-Modell* (siehe Kapitel 4).
In den nächsten Abschnitten werden die für diese Arbeit benötigten grundlegenden Begriffe und Schreibweisen definiert.

1.4 Die Objektklassen der Schaltkreisdarstellung

Um Informationen an der Benutzeroberfläche von CADIC anzuzeigen, muß zunächst die Schaltkreisspezifikation dahingehend untersucht werden, an welchen Stellen der graphischen Darstellung eine Visualisierung sinnvoll ist. Am Beispiel eines Volladdieres soll verdeutlicht werden, welche Teile des Entwurfs für die Anzeige der Informationen in Frage kommen. Die Abbildung 1.2 entspricht der Beschreibung einer Hierarchieebene eines Volladdierers in CADIC.

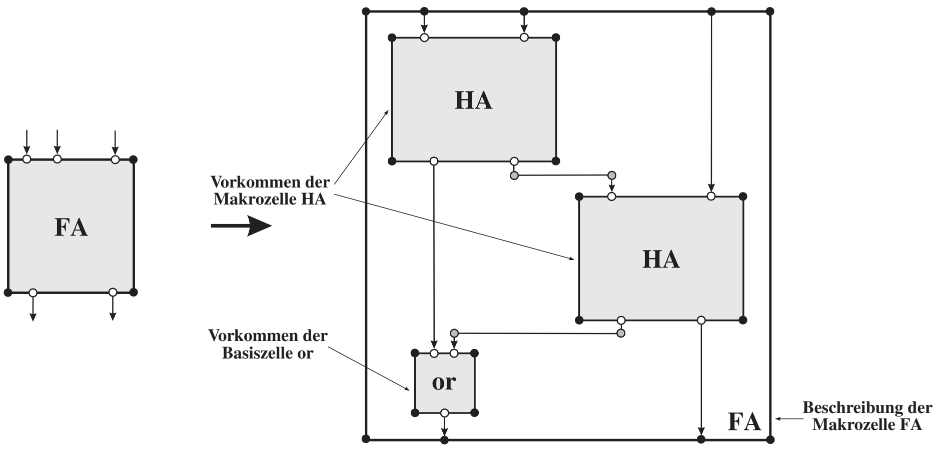

Abbildung 1.2: graphische Darstellung eines Volladdieres in CADIC

Die Beschreibung des verwendeten Halbaddierers vervollständigt die Spezifikation (siehe Abbildung 1.3). Die dargestellten Hierarchieebenen bestehen aus Verdrahtungsknoten und -kanten sowie Randknoten und -kanten. Die verwendeten Makro- und Basiszellen setzen sich wiederum aus Randknoten und -kanten sowie Verdrahtungsknoten zusammen. Die Anzeige von Informationen kann also grundsätzlich an den Knoten, den Kanten oder den Bausteinen der Schaltungsdarstellung erfolgen.

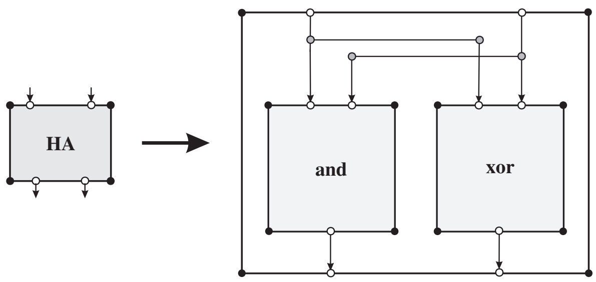

Abbildung 1.3: Definition eines Halbaddieres

Als Werkzeugentwickler oder -anwender im Bereich Schaltkreisentwurf betrachtet man die Knoten, Kanten und Bausteine aber unter verschiedenen Gesichtspunkten. Man unterscheidet zwischen Vorkommen (Instanzen) und den Beschreibungen von Bausteinen, zwischen Anschlüssen auf dem äußeren Rand und den Ein- und Ausgängen von

Bausteinvorkommen usw. Andererseits faßt man eventuell mehrere Komponenten der Darstellung zusammen. Zum Beispiel betrachtet man bei der Analyse von Schaltungen alle miteinander verbundene Verdrahtungskanten als logische Signalnetze.
Unter dem Gesichtspunkt der Visualisierung bedeutet dies aber, daß die Grundkomponenten der Schaltkreisdarstellung teilweise mehrfach überlagert sind. Man kann zum Beispiel eine Kante als Verdrahtungskante oder als Repräsentanten eines Signalnetzes auffassen.

Überlagerungen entstehen auch durch verschiedene Schaltkreissichten (hierarchisch gegliederte oder flache Darstellung). In der graphischen Arbeitsumgebung von `CADIC` wird stets eine Hierarchieebene der Spezifikation angezeigt. Je nachdem welche Schaltkreissicht zugrundeliegt, betrachtet man diese Ebene als Beschreibung eines Bausteins der hierarchisch gegliederten Schaltung bzw. als das durch den zu dieser Ebene zurückgelegten Weg bestimmte Zellvorkommen der flachen Schaltung. Dadurch kann man einen Knoten, eine Kante oder auch ein Vorkommen einer Zelle sowohl als Element der hierarchisch gegliederten als auch der "flachen" Schaltkreisdarstellung interpretieren.

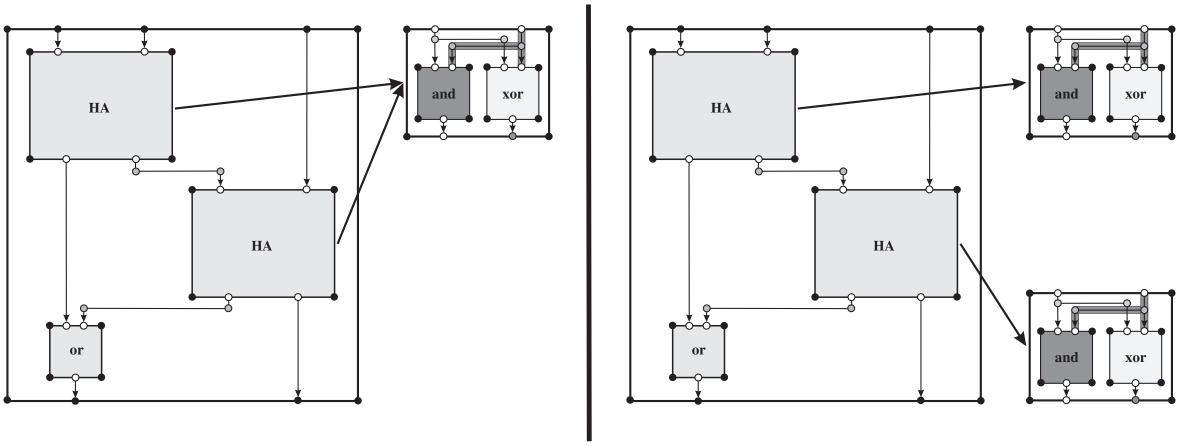

Abbildung 1.4: hierarchisch gegliederte und flache Schaltkreissicht

Abbildung 1.4 stellt die beiden Schaltkreissichten am Beispiel des Volladdierers gegenüber. Der linke Teil der Abbildung korrespondiert zur hierarchisch gegliederten Schaltkreisdarstellung. Beide Vorkommen des Halbaddieres verweisen auf dessen Beschreibung. Würde man beispielsweise Informationen am **and**-Gatter im Halbaddierer anzeigen wollen, wären diese in beiden Vorkommen identisch und würden nur einmal gespeichert.

Betrachtet man im Gegensatz dazu die flache Schaltkreisdarstellung im rechten Teil der Abbildung 1.4, wird hier zwischen den Vorkommen der Zellen unterschieden. Daraus resultiert, daß jedes der beiden **and**-Gatter mit unterschiedlichen Informationen versehen werden kann, die folglich auch separat gespeichert werden. Diese Beobachtung gilt nicht nur für das **and**-Gatter, sondern für alle in der Beschreibung des Halbaddieres vorkommenden Elemente. In der Abbildung sind beispielhaft ein lokales Signalnetz und ein Anschluß auf dem Rand zusätzlich hervorgehoben.

Um dem Werkzeugentwickler die Möglichkeit einzuräumen, basierend auf seiner Schaltkreissicht an einem bestimmten Element der Darstellung Informationen zu visualisieren, werden im folgenden **Objektklassen** definiert. Eine solche Objektklasse faßt zum Beispiel alle Verdrahtungskanten bezüglich der hierarchisch gegliederten Schaltkreisdarstellung zusammen. Durch diese Klassenbildung können Informationen aus der dem Werkzeugentwickler gewohnten Sicht visualisiert werden. Die Menge der Objektklassen wird wie folgt definiert.

Definition 1.1: *Seien OK, OKF und OKH die im folgenden definierten Mengen.*

$$\begin{aligned} \mathrm{OKF} &= \{\texttt{PIN}, \texttt{PAD}, \texttt{FANOUT}, \texttt{CNODE}, \texttt{WIRE}, \texttt{CEDGE}, \texttt{NET}, \texttt{SIGNAL}, \texttt{INSTANCE}\} \\ \mathrm{OKH} &= \{\texttt{HRY\_PIN}, \texttt{HRY\_PAD}, \texttt{HRY\_FANOUT}, \texttt{HRY\_CNODE}, \texttt{HRY\_WIRE}, \texttt{HRY\_CEDGE}, \texttt{HRY\_NET} \\ &\quad\ \texttt{HRY\_INSTANCE}, \texttt{CELL}\} \\ \mathrm{OK} &= \quad \mathrm{OKH} \cup \mathrm{OKF} \cup \{\texttt{GLOBAL}, \texttt{TEMPORARY}\} \end{aligned}$$

Ein Element okl $\in$ OK heißt *Objektklasse*.
Ein Element okl $\in$ OKF (OKH) heißt *flache* (*hierarchische*) *Objektklasse*.

□

Durch die Bereitstellung der Objektklassen wird auch die Anzeige von Informationen an zusammengefaßten Elementen der Darstellung, wie Signalnetzen ermöglicht, ohne daß diese an jede Einzelkomponente propagiert werden müssen. Weiterhin bieten die Objektklassen den Vorteil, die Knoten und Kanten der Beschreibung nach ihren Eigenschaften getrennt zu behandeln. Eigenschaften sind dabei zum Beispiel Verdrahtung oder Rand. Im folgenden wird erläutert, welche Teile der Darstellung durch die einzelnen Objektklassen erfaßt werden.

Betrachten wir zunächst die in der Spezifikation von *FA* verwendeten Makro- und Basiszellen. Schließt man dabei die Definition des Bausteins *HA* mit ein, besteht der gesamte Entwurf aus insgesamt fünf Bausteinen, den Basiszellen *and*, *xor* und *or* sowie den Makrozellen *HA* und *FA*. Alle verwendeten Bausteine faßt man zur Objektklasse `CELL` zusammen. Unterscheidet man zwischen den Vorkommen von Bausteinen nur innerhalb der Hierarchieebenen, faßt man Informationen zu dieser hierarchischen Sicht von Zellvorkommen durch die Objektklasse `HRY_INSTANCE` zusammen. Entfaltet man die hierarchische Schaltkreisdarstellung und unterscheidet nun zwischen den Vorkommen von Makro- und Basiszellen rekursiv in allen Hierarchieebenen, ordnet man diese der Objektklasse `INSTANCE` zu. Der Leser betrachte nochmals Abbildung 1.4. Alle sichtbaren Zellvorkommen im linken Teil des Bildes gehören der Klasse `HRY_INSTANCE`, die Zellvorkommen im rechten Teil der Klasse `INSTANCE` an.
Die Knoten und Kanten des Netzgraphen sind in `CADIC`, wie schon erwähnt, durch verschiedene Eigenschaften bzw. Attribute gekennzeichnet. Bei Knoten unterscheidet man zwischen den Anschlüssen auf dem Rand von Makrozellen (*Pads*), Ein- und Ausgängen von Instanzen (*Pins*), Verzweigungsknoten (*Fanoutknoten*), Verdrahtungsknoten und Randknoten (siehe Abbildung 1.5). Zur Visualisierung werden die Knoten entsprechend in Objektklassen eingeteilt. Der Klasse `CNODE` bzw. `HRY_CNODE` gehören alle Knoten unabhängig von ihrem Attribut an, je nachdem ob die flache oder hierarchisch gegliederte Schaltkreisdarstellung zugrundegelegt wird. Die Anschlüsse auf dem äußeren Rand

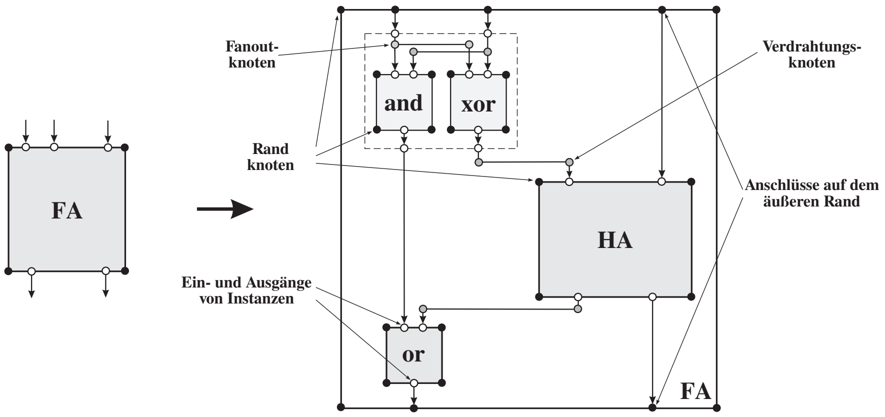

Abbildung 1.5: Klassifizierung der Knoten der Beschreibung

von Makrozellen gehören der Objektklasse `PAD` bzw. `HRY_PAD`, die Ein- und Ausgänge von Instanzen der Klasse `PIN` bzw. `HRY_PIN` und die Verzweigungsknoten der Klasse `FANOUT` bzw. `HRY_FANOUT` an.

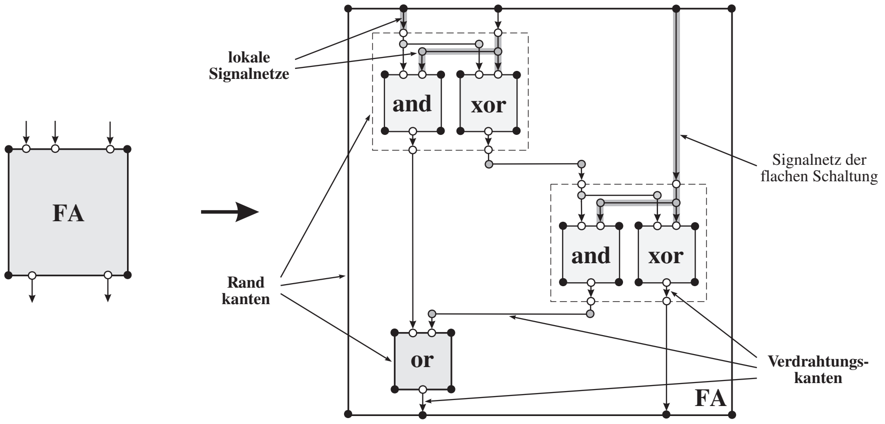

Abbildung 1.6: Klassifizierung der Kanten der Beschreibung

Die Kanten des Netzgraphen können die Attribute Verdrahtungskante und Randkante besitzen (vergleiche Abbildung 1.6). Analog zu den Objektklassen der Knoten existiert für alle Kanten unabhängig von ihrem Attribut die Klasse `CEDGE` bzw. `HRY_CEDGE` . Die Verdrahtungskanten bilden die Klasse `WIRE` bzw. `HRY_WIRE`.

Zusammenhängende Verdrahtungskanten auf einer Hierarchieebene (*lokale Signalnetze*) faßt man zur Objektklasse `NET` bzw. `HRY_NET` zusammen. In Abbildung 1.6 sind lokale Signalnetze durch graue Umrandung der zugehörigen Verdrahtungskanten dargestellt. Interpretiert man die lokalen Signalnetze als Teil von Signalnetzen der flachen Schaltung, faßt man diese zur Objektklasse `SIGNAL` zusammen. Ein Signalnetz der flachen Schaltung besteht aus verschiedenen Komponenten lokaler Signalnetze, die an den Hierarchiegrenzen miteinander verbunden sind. Als Beispiel betrachte man in Abbildung 1.6 das Signalnetz am rechten Eingang des Volladdieres. Dieses erstreckt sich über

zwei Hierarchieebenen. Man beachte, daß die gleichen lokalen Netze im Halbaddierer zu verschiedenen Signalnetzen der flachen Schaltung gehören.
Neben den Objektklassen, die direkt zu den Elementen der Schaltkreisdarstellung korrespondieren, stehen zusätzlich die Klassen `GLOBAL` und `TEMPORARY` zur Verfügung. Die Objektklasse `GLOBAL` dient der Repräsentation von Informationen, welche man dem Gesamtschaltkreis zuordnen würde. Warn- und Hilfsmeldungen an den Benutzer sowie andere temporäre Entwurfsdaten werden durch die Objektklasse `TEMPORARY` erfaßt.
Alle verfügbaren Objektklassen sind im Anhang A nochmals in einer Tabelle aufgelistet und kurz beschrieben.
Nach der Einteilung der signifikanten Stellen des Entwurfs in Objektklassen muß jetzt angegeben werden, wie zwischen den einzelnen Elementen dieser Klassen unterschieden werden kann.

1.5 Ordnungsnummern und Selektoren

In diesem Abschnitt wird der Leser mit den in der `CADIC`-Datenstruktur vorhandenen Numerierungen von Makro- und Basiszellen, deren Vorkommen sowie von Knoten und Kanten vertraut gemacht. Mit Hilfe dieser Numerierungen und darauf aufbauender Definitionen wird anschließend der Begriff der Objektposition vorgestellt. Durch Objektpositionen kann man zwischen den Elementen einer Objektklasse unterscheiden.

In der `CADIC`-Datenstruktur sind alle verwendeten Makro- und Basiszellen von 0 bis Anzahl aller Teilschaltungen - 1 durchnumeriert. Dadurch erhält jede Beschreibung einer Zelle einen eindeutigen Schlüssel (im folgenden **Zellenordungsnummer**).

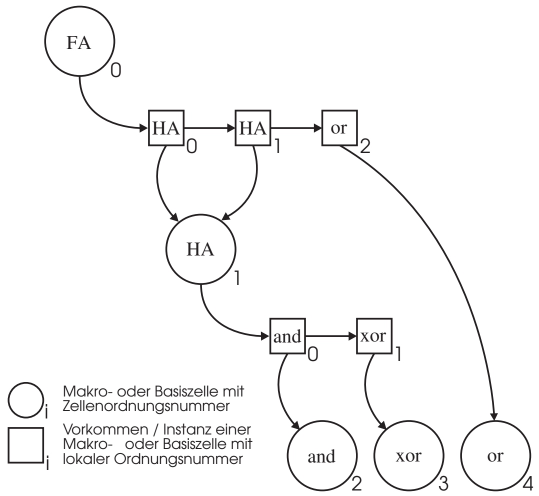

Abbildung 1.7: schematischer Aufbau der Datenstruktur am Beispiel des Volladdierers

Die Vergabe der Zellenordnungsnummern wird am Beispiel des Volladdieres in Abbildung 1.7 gezeigt. Neben den Zellenordnungsnummern sind jeweils die Knoten, die Kanten, die Zellvorkommen und die lokalen Signalnetze lokal bezüglich einer Hierarchieebene durchnumeriert :

- alle Knoten einer Hierarchieebene von 0 bis Anzahl der Knoten - 1
- alle Kanten einer Hierarchieebene von 0 bis Anzahl der Kanten - 1
- alle Zellvorkommen einer Hierarchieebene von 0 bis Anzahl der Zellvorkommen - 1 (siehe Abbildung 1.7)
- alle lokalen Signalnetze einer Hierarchieebene von 0 bis Anzahl der Signalnetze - 1 (diese Numerierung ist nach optionaler Berechnung der hierarchischen Netzliste verfügbar)

Die genannten Indizes bezeichnet man als **lokale Ordnungsnummern**. Die lokalen Ordnungsnummern in Verbindung mit den Zellenordnungsnummern erlauben eine eindeutige Kennzeichnung für jedes Element der hierarchischen Objektklassen, z.B. kann ein Pin durch Angabe der Zellenordnungsnummer der betreffenden Beschreibung und der lokalen Knotenordnungsnummer eindeutig bestimmt werden.
Entfaltet man die hierarchische Darstellung, reichen die obigen Numerierungen zur Lokalisierung der Elemente nicht mehr aus, da nicht zwischen verschiedenen Vorkommen von Makro- und Basiszellen unterschieden werden kann. Eine Möglichkeit zur Unterscheidung wäre, alle Vorkommen von Teilschaltungen der flachen Schaltung durchzunumerieren. Eine andere Variante zur Unterscheidung der verschiedenen Instanzen besteht darin, ein Vorkommen durch den Pfad von der Wurzel der Hierarchie zu dieser Instanz zu kennzeichnen.

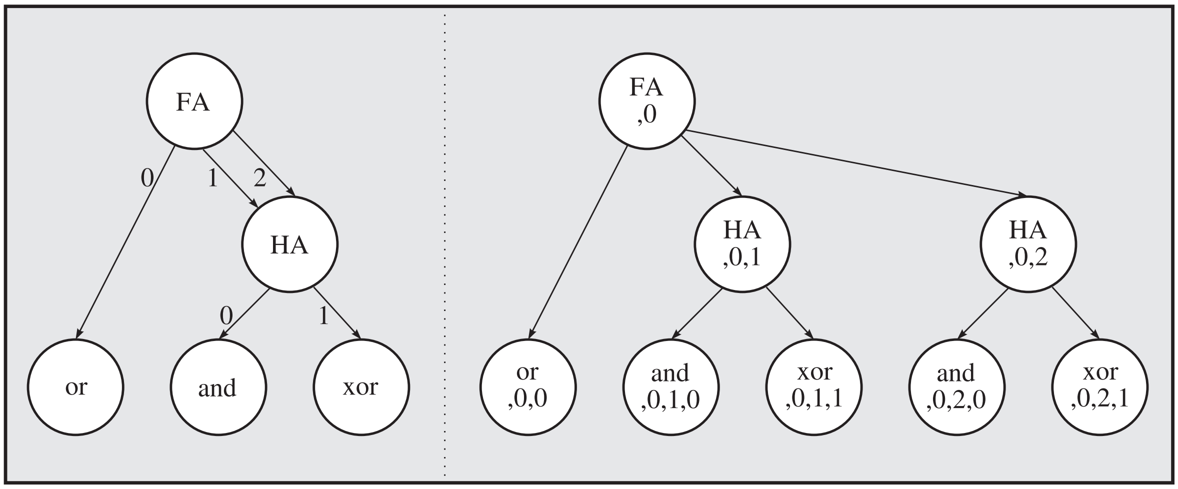

Abbildung 1.8: Selektoren am Beispiel eines Volladdierers

Abbildung 1.8 zeigt auf der linken Seite die hierarchisch gegliederte Spezifikation des Volladdieres. Das Mehrfachvorkommen des Halbaddieres ist durch die zwei Verweise angedeutet. Die in der `CADIC`-Datenstruktur vergebenen lokalen Instanzordnungsnummern sind an den Pfeilen dargestellt. Mit Hilfe dieser Indizes kann der Weg von der

Wurzel zu einer Instanz der Schaltung eindeutig beschrieben werden. Die durch die Indizes enstehende Beschriftung der von der Wurzel ausgehenden Wege bezeichnet man als Selektoren. Auf der rechten Seite der Abbildung 1.8 wird die Vergabe der Selektoren gezeigt. Das Vorkommen der Wurzel erhält den Selektor ",0". Jede Instanz erhält als Selektor den Selektor der umgebenden Instanz verlängert um die eigene lokale Ordnungsnummer. Den Begriff Selektor kann man also folgendermaßen definieren :

Definition 1.2: *Sei S eine Schaltkreisspezifikation und is eine Instanz einer in S verwendeten Makro- oder Basiszelle. Die innerhalb von is vorkommenden Instanzen seien mit $i_1, \ldots, i_n$ $(n \in \mathbb{N}_0)$ bezeichnet. O.B.d.A. habe dabei die Instanz i_j die lokale Ordnungsnummer j $(1 \leq j \leq n)$.*

Der Selektor sel einer Instanz wird definiert als

i) sel(Vorkommen der Wurzel von S) = ",0"
ii) Für $i_k \in \{i_1, \ldots, i_n\}$: $sel(i_k) = sel(is) \cdot$ "," $\cdot\, k$ $\quad (1 \leq k \leq n)$
(" $\cdot$ " = Konkatenation von Zeichenketten)

□

Mit Hilfe der Selektoren und den lokalen Ordnungsnummern kann nun auch zwischen den Elementen der flachen Objektklassen unterschieden werden. Zur eindeutigen Adressierung der Elemente einer Objektklasse wird im folgenden der Begriff der **Objektposition** defininiert.

Definition 1.3: *Sei $okl \in OK \setminus \{\texttt{GLOBAL}, \texttt{TEMPORARY}\}$ eine Objektklasse und $key \in \mathbb{N}_0$ eine Zellenordnungsnummer. Sei weiter sel ein Selektor und $nr \in \mathbb{N}_0$ eine lokale Ordnungsnummer.*

Eine Objektposition *pos_{okl} eines Elementes $o \in okl$ ist ein Tupel*

$pos_{okl}(o) = (sel, nr)$	für alle $okl \in OKF$
$pos_{okl}(o) = (key, nr)$	für alle $okl \in OKH \setminus \{\texttt{CELL}\}$
$pos_{okl}(o) = (key)$	für $okl = \texttt{CELL}$.

□

Die Objektklassen `GLOBAL` und `TEMPORARY` bestehen jeweils nur aus einem Element, sodaß die Angabe einer Objektposition hier überflüssig ist.

Zur Veranschaulichung sind in Abbildung 1.9 auszugsweise einige lokale Knoten-, Kanten-, Instanzordnungsnummern sowie lokale Numerierungen von Signalnetzen dargestellt. Betrachtet man zum Beispiel den Fanoutknoten im Halbaddierer mit der lokalen Ordnungsnummer 26. Die Objektpositionen dieses Knotens bzgl. der Klassen `HRY_CNODE` und `HRY_FANOUT` heißen jeweils :

$$pos_{\texttt{HRY\_CNODE}}(node) = (1, 26) \text{ und } pos_{\texttt{HRY\_FANOUT}}(node) = (1, 26)\ ,$$

da die Makrozelle HA die Zellenordnungsnummer 1 (vergleiche Abbildung 1.7) besitzt. Legt man die entfaltete Schaltkreisdarstellung zugrunde, existieren zwei Vorkommen dieses Knotens und demzufolge jeweils zwei Elemente der Klassen `CNODE` und `FANOUT`. Die zugehörigen Objektpositionen lauten :

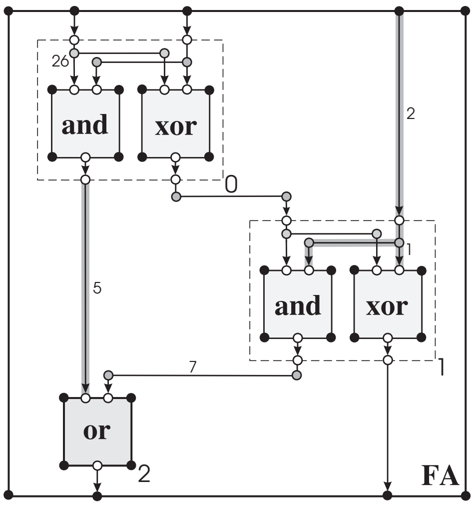

Abbildung 1.9: lokale Ordnungsnummern

$pos_{\mathtt{CNODE}}\ (node_1) = (\ ``,0,0"\ ,\ 26\ )$ bzw. $pos_{\mathtt{CNODE}}\ (node_2) = (\ ``,0,1"\ ,\ 26\ )$
$pos_{\mathtt{FANOUT}}(node_1) = (\ ``,0,0"\ ,\ 26\ )$ bzw. $pos_{\mathtt{FANOUT}}(node_2) = (\ ``,0,1"\ ,\ 26\ )$

Die Objektpositionen zu Kanten, Zellvorkommen und Netzen werden analog gebildet. Dabei werden jedoch die entsprechenden lokalen Kanten-, Instanz- bzw. Netzordnungsnummern zugrundegelegt. Als Beispiel sei noch die Objektposition des grau umrahmten lokalen Netzes im rechten Halbaddierer angegeben, falls dieses als Teil eines Signalnetzes der flachen Schaltung interpretiert wird :

$pos_{\mathtt{SIGNAL}}(net) = (\ ``,0,1"\ ,\ 1)$

Man beachte, daß sich das dazu korrespondierende Signalnetz der flachen Schaltung über zwei Hierarchieebenen erstreckt. Das lokale Signalnetz mit der Ordnungsnummer 2 im Volladdierer kann ebenfalls als Element der Klasse `SIGNAL` interpretiert werden. Die zugehörige Objektposition lautet :

$pos_{\mathtt{SIGNAL}}(net) = (\ ``,0"\ ,\ 2)$

Es existieren somit zwei Elemente der Klasse `SIGNAL`, die zu einem Signalnetz der flachen Schaltung korrespondieren.

1.6 Visualisierbare Datentypen

In den vorangegangenen Abschnitten wurde gezeigt, wie Interpretationen von Teilen der Schaltkreisdarstellung durch Objektklassen und zugehörige Objektpositionen eindeutig addressiert werden können. In diesem Abschnitt wird festgelegt, welche Entwurfsdaten bzw. welche Typen von Informationen man dort visualisieren kann.
Dazu wird eine Menge T von Datentypen definiert, die direkt, d.h. ohne aufwendige Konvertierung, dargestellt werden können. Die in T enthaltenen Datentypen sind an die Typdeklarationen der Programmiersprache C angelehnt.
Als Grundtypen stehen `int`, `long`, `float`, `double` und `char` zur Verfügung. Diese können vorzeichenbehaftet oder vorzeichenlos verwendet werden. Aufbauend auf diesen Grundtypen besteht die Möglichkeit, mit Hilfe von Zeigern, Feldern und Strukturen komplexere Datentypen zu bilden. Alle zu verwendenden Strukturen müssen deklariert werden (zur Deklaration siehe Abschnitt 3.1.1).
Dynamisch angelegte Felder und Zeiger werden nur in Spezialfällen unterstützt, die aber im allgemeinen ausreichend sind. Notwendig zum internen Kopieren der Informationen ist die Größenbestimmung des dynamisch angelegten Speicherbereichs zur Laufzeit. Dies ist zum Beispiel bei Zeichenketten (`char *`) möglich, da deren Länge durch das abschließende '0'-Byte bestimmt wird.
Ein weiteres Beispiel sind Felder numerischen Typs, die folgende Eigenschaft besitzen :

$$(\text{a}[0]=\text{k} \; ; \; \text{a}[1]=a_1, \ldots, \text{a}[\text{k}]=a_k \text{ für } [a_1, \ldots, a_k] \; ; \; \text{a}[0] = \text{Länge des Feldes a})$$

Ein solches Feld der Länge k benötigt Speicherplatz für $k+1$ Elemente, wobei in der 0-ten Komponente die Länge des Feldes und von der ersten bis zur k-ten Komponente die Feldelemete abgelegt sind.
Eine weitere Möglichkeit zur Größenbestimmung dynamisch angelegter Speicherbereiche besteht bei Strukturkomponenten, falls diese über eine andere Strukturkomponente erfolgen kann. Diese andere Komponente muß von numerischen Typ sein. Zur Veranschaulichung soll das folgende Beispiel dienen.

Beispiel 4: *dynamische angelegte Felder in* `CADIC`

```
struct Netzliste
  {
  long          anz_net;    /* Anzahl der Netze */
  struct Netze *netze;      /* Liste der Netze */
  ...
  };
```

□

Die Struktur entspricht der internen Darstellung einer Netzliste bezüglich einer Hierarchieebene. Der Übersicht halber wurde nur der hier relevante Teil der Strukturkomponenten aufgelistet und auf die Deklaration der Struktur `Netze` verzichtet.
Die Struktur `Netzliste` kann visualisiert werden, da die Größe des dynamisch angelegten Feldes bestimmbar ist. Die Länge des Feldes `Netze` wird durch die Komponente `anz_net` angegeben.
Wie dynamisch angelegte Felder im einzelnen deklariert und benutzt werden, wird in

Abschnitt 3.1.1 besprochen. Falls die Größenbestimmung nicht auf obige Weise erfolgen kann, ist zumindest die Visualisierung des durch den Zeiger referenzierten Objekts möglich.
Trotz der genannten Einschränkung können komplexe Datenstrukturen visualisiert werden, beispielsweise auch die interne `CADIC`-Datenstrukur (siehe Abschnitt 2.1).

Zum Einfärben von Elementen wurde ein abstrakter Datentyp `Color` eingerichtet. Diesem Typ kann man Farbnamen zuweisen. Intern wird die Farbe aus Speicherplatz- und Effizienzgründen als Pixelwert abgespeichert.

Die nachfolgende Auflistung enthält zur Übersicht nochmals alle Datentypen aus T :

- int, long, float, double (signed, unsigned)
- char (signed, unsigned)
- Zeiger
  - auf Strukturen
  - char*
  - void*
- Felder
  - konstante oder dynamische (mit obigen Eigenschaften) Länge
- Strukturen (geschachtelt, über Zeiger auch rekursiv verkettet)
- Color (Abstrakter Datentyp für Farben)

1.7 Speicherung der Informationen

Die anzuzeigenden Entwurfsdaten werden von den Werkzeugen über Funktionsaufrufe an das Modul Visualisierung geschickt. Die eigentliche Daten werden dabei per Referenz übergeben. Aufgabe der Visualisierung ist nun, die erhaltenen Informationen zu kopieren und zu speichern. Die Speicherung der Daten ist vorallem für die graphische Aufbereitung von Bedeutung, da hier i.a. eine ganze Folge von Informationen zusammen dargestellt werden soll. Andererseits gibt es Informationen, die nur solange gespeichert werden müssen bis neue, aktualisierte Werte vorliegen.
Daraus kann man als erste Anforderung an die Speicherung der Daten ableiten, daß die Maximalzahl der zu speichernden Elemente einstellbar sein muß. Ein weiteres Kriterium entsteht dadurch, daß in `CADIC` kann jeder Knoten, jede Kante und demzufolge auch jedes lokale Signalnetz eine Einzelleitung oder einen ganzes Bündel von Leitungen (Bus) repräsentieren. Die Anzahl der Einzelkomponenten eines Busses wird mit Busbreite bezeichnet.
Zur Einfärbung einer Busleitung insgesamt reicht es zum Beispiel, eine entsprechende

Farbe zu speichern. Die Anzeige von logischen Signalwerten dagegen erfordert für jede Komponente eines solchen Busses genau einen Signalwert, da die Einzelleitungen im allgemeinen zu verschiedenen Signalnetzen der flachen Schaltung gehören. Wahlweise wird man demzufolge entweder nur eine oder busbreite viele Informationen an die Elemente heften, welche einen Bus repräsentieren können. Die Tabelle im Anhang A verdeutlicht, welche Elemente der Objektklassen dies betrifft.

Der im folgenden definierte abstrakte Datentyp *Informationspuffer* dient der flexiblen Speicherung der Daten, und gestattet gleichzeitig effizienten Zugriff auf die abgelegten Elemente.

Definition 1.5: *Sei $P^{t,l}$ ein abstrakter Datentyp zur Speicherung von l Elementen des Typs $t \in T$ ($l \in \mathbb{N}$), welcher die folgenden Operationen*

- *Einfügen eines Eintrages vom Typ t am Ende von $P^{t,l}$ in konstanter Zeit*
- *Zugriff auf den ersten Eintrag in konstanter Zeit*
- *Zugriff auf den letzten Eintrag in konstanter Zeit*
- *Zugriff auf den Nachfolger eines Eintrags in konstanter Zeit*
- *Zugriff auf den Vorgänger eines Eintrags in konstanter Zeit*
- *Löschen aller Einträge von $P^{t,l}$*

unterstützt. In $P^{t,l}$ können maximal l Einträge vom Typ t gespeichert werden. Enthält $P^{t,l}$ bereits l Elemente, wird beim Einfügen eines weiteren Elements der erste bzw. älteste Eintrag gelöscht.

Unter einem **Informationspuffer der Breite** *k versteht man k Referenzen auf paarweise verschiedene Objekte vom Typ $P^{t,l}$ ($k \in \mathbb{N}$).*
Als abkürzende Schreibweise wird $Ip^{t,l}[0 : k\text{-}1]$ verwendet.

□

Ein Informationspuffer der Breite k könnte also durch k doppelt verkettete Listen mit Längenbeschränkung und entsprechender Einfügeoperation realisiert werden. Zu beachten ist, daß zwei Informationspuffer teilweise auf gleiche Objekte vom Typ $P^{t,l}$ verweisen können. Dies ist vorallem bei der Objektklasse `SIGNAL` von Bedeutung, da hier unter Umständen mehrere Elemente der Schaltkreisdarstellung zu einem Signalnetz der flachen Schaltung gehören. Durch die so definierten Informationspuffer werden die Entwurfsdaten nur einmal pro Signalnetz gespeichert, können aber an verschiedenen Stellen angezeigt werden.
Ein Informationspuffer zur Speicherung von Farbdaten zur Einfärbung eines Elements lautet beispielsweise $Ip^{Color,1}[0 : 0]$. Die Speicherung von logischen Signalwerten zur Anzeige von Timing-Diagrammen erfordert demgegenüber einen Informationspuffer $Ip^{char,1000}$[0 :Busbreite des lokalen Signalnetzes], falls der Signalverlauf über 1000 Zeitschritte hinweg verfolgt werden soll.

Den Elementen der Objektklassen können mehrere Informationspuffer zugeordnet werden, um verschiedene Datentypen zu visualisieren. Zur Unterscheidung zwischen den

Informationspuffern werden diesen Namen zugewiesen, die den in Programmiersprachen verwendeten Bezeichnern entsprechen.

Definition 1.6: *Sei* $okl \in OK$ *eine Objektklasse,* $Ip^{t,l}[0:k\text{-}1]$ *ein Informationspuffer,* $L \in \{A, B, ..., Z, a, b, ..., z, \_\}$ *ein Zeichen und* $D \in \{0, 1, ..., 9\}$ *eine Ziffer.*
Seien weiter $C \in \{A, B, ..., Z, a, b, ..., z\} \cdot (L \cup D)^*$, $Identifier \in C \cup C\cdot$"::"$\cdot C$ *und* δ_{okl} *eine injektive Abbildung, die jedem Informationspuffer einen eindeutigen Identifier zuordnet.*

Dann heißt $\delta_{okl}(Ip^{t,l}[0:k\text{-}1])$ *der* **Name des Informationspuffers** $Ip^{t,l}[0:k\text{-}1]$ *bezüglich der Objektklasse okl.*

□

Die Menge *Identifier* umfaßt normale und zusammengesetzte Bezeichner (mit dem Trennzeichen "::" verbunden). Führende Unterstriche oder Ziffern sind durch die Definition bei den Bezeichnern nicht erlaubt. Die Notwendigkeit von zusammengesetzten Bezeichnern wird im Kapitel 6 erläutert. Die Namen der Informationspuffer besitzen nur Gültigkeit innerhalb einer Objektklasse. Dadurch hat der Werkzeugentwickler einen ähnlichen Vorteil wie durch lokale Variablenbenennung in Prozeduren bei den Programmiersprachen.

1.8 Die Objekte der Visualisierung

Mit Hilfe der eingeführten Begriffe Objektklasse, Objektposition und Informationspuffer kann nun der für die Visualisierung zentrale Begriff des "Objekts" definiert werden. Als Objekte werden die Elemente der Objektklassen bezeichnet. Die Objekte korrespondieren zu einem bestimmten Teil der Schaltungsdarstellung, d.h. zum Beispiel zu einem speziellen Knoten bezüglich einer Schaltkreissicht. Ein Objekt verwaltet die dort anzuzeigenden Daten. Um dies zu ermöglichen, werden jedem Objekt verschiedene Informationspuffer zur Speicherung dieser Daten zugeordnet. Der Begriff des Objektes wird folgendermaßen definiert :

Definition 1.7: *Seien* $okl \in OKF \cup OKH$ *eine Objektklasse,* pos_{okl} *eine zugehörige Objektposition, o das durch* pos_{okl} *bestimmte Element der Schaltkreisdarstellung und* $Ip_1^{t_1,l_1}[0:k_1\text{-}1], ..., Ip_n^{t_n,l_n}[0:k_n\text{-}1]$ *n Informationspuffer (*$t_i \in T, l_i \in \mathbb{N}, k_i \in \{1,$ *Busbreite von o*$\}, n \in \mathbb{N}, 1 \le i \le n$*).*

Ein Tupel $O = (\quad okl \quad, pos_{okl}, Ip_1^{t_1,l_1}[0:k_1\text{-}1], ..., Ip_n^{t_n,l_n}[0:k_n\text{-}1]\ )$ *heißt* **Objekt**.

Die Tupel $O = (\ \texttt{GLOBAL}\ ,\ *\ ,\ Ip_1^{t_1,l_1}[0:0], ..., Ip_n^{t_n,l_n}[0:0]\ )$ *und*
$O = (\ \texttt{TEMPORARY},\ *\ ,\ t_1, ..., t_n\ )$

heißen **globales** *bzw.* **temporäres Objekt**.

($*$ = *don't care*)

□

Ein Objekt $O \in OKF \cup OKH$ faßt also mehrere Informationspuffer an dem durch die

Objektposition bestimmten Element der Darstellung zusammen. Repräsentiert dieses Element einen Bus der Breite k, können die einzelnen Informationspuffer die Breite 1 oder k besitzen.
Die Objektklassen `GLOBAL` und `TEMPORARY` bestehen jeweils aus einem Element. Die Angabe einer Objektposition ist deshalb nicht erforderlich.
Dem temporären Objekt sind keine Informationspuffer sondern eine Menge von Datentypen zugeordnet, da die Informationen hier nur angezeigt und nicht gespeichert werden. Die Hauptanwendung des temporären Objekts ist die Anzeige von Warn- und Hilfsmeldungen (siehe auch Abschnitt 3.4.3), um die Interaktion zwischen den Entwurfswerkzeugen und dem Benutzer zu unterstützen.

Kapitel 2

Visualisierungsmöglichkeiten

Dieses Kapitel beschäftigt sich mit der eigentlichen Visualisierung der Entwurfsdaten, d.h. der Anzeige der Informationen an der graphischen Oberfläche von `CADIC`.
Dazu stehen eine Vielzahl von Darstellungsformen zur Verfügung, welche in den folgenden Abschnitten erläutert werden. Die Darstellungsformen unterscheiden sich nicht nur in der reinen Anzeige, sondern auch durch das Vorkommen innerhalb der graphischen Oberfläche. So können Informationen zum Beispiel direkt in das Arbeitsfenster geschrieben oder in Dialogfenstern angezeigt werden. Letztere kann der Benutzer durch Anwahl mit der Maus am betreffenden Objekt öffnen (siehe auch Abschnitt 3.4). Damit wird verhindert, daß durch eine Vielzahl von plötzlich erscheinenden Fenstern die Anzeige überladen wird.
Grundsätzlich wird für jede Objektklasse eine **Standardausgabe** der deklarierten Informationspuffer erzeugt. Diese enthält für jeden Puffer dessen Namen und den aktuellen Wert. Die Standardausgabe ist unabhängig von den anderen gewählten Darstellungsformen immer verfügbar und wird stets im Dialogfenster beim betreffenden Objekt angezeigt. Der Vorteil der Standardausgabe besteht darin, daß auch ohne Definition von Anzeigeformen eine Darstellung der Informationen ermöglicht wird.
Alle anderen definierbaren Visualisierungsmöglichkeiten werden in den nächsten Abschnitten vorgestellt.

2.1 Textuelle Darstellung

Die naheliegendste Art der Anzeige von Informationen ist die textuelle Darstellung. Diese kann bei allen Objektklassen gewählt werden.
Der Werkzeugentwickler kann die textuelle Darstellung aus beliebigen konstanten Texten und den Werten der deklarierten Informationspuffer zusammenstellen. In Abbildung 2.1 sind den Werten erläuternde Texte vorangestellt, zum Beispiel "\nslack\t =". Ein solcher konstanter Text kann Tabulatoren und Zeilenwechsel ('\t' und '\n') enthalten. Die aktuellen Inhalte der Informationspuffer werden entsprechend ihrem Typ in Zeichenketten konvertiert und zusammen mit den konstanten Zusatztexten angezeigt. Die so gebildeten Informationen können je nach Vorgabe direkt im Arbeitsfenster von

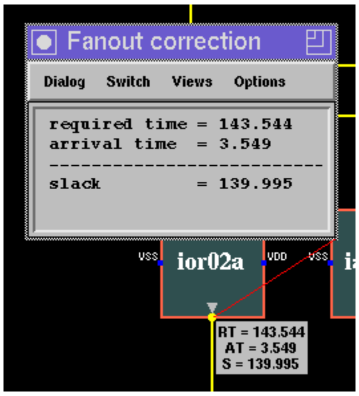

Abbildung 2.1: Textuelle Darstellung im Arbeitsfenster und im Dialogfenster

`CADIC` am zugehörigen Objekt bzw. in einem Dialogfenster dargestellt werden. Um die Übersichtlichkeit aufrecht zu erhalten, sollten im Arbeitsfenster nur Kurzinformationen an die Objekte geschrieben werden. Dies gilt vorallem bei Busleitungen mit großen Busbreiten (die Anzeige von ganzen Zahlen an jeder Einzelleitung eines Bus der Breite 32 ist z.B. unzweckmäßig). Wird wie in Abbildung 2.1 mehrzeilige Information zusammengestellt, kann man zwischen linksbündigem, zentriertem bzw. rechtsbündigem Format wählen.
Strukturen werden wie in der Programmiersprache **C** üblich formatiert :

```
struct example
    {
    typ1 member1;
    ...
    typn membern;
    }
```

Falls geschachtelte Strukturen verwendet wurden, setzt sich dieses Schema fort. Für Informationspuffer vom Typ Zeiger auf Struktur steht zur Visualisierung eine **hypertextähnliche** Anzeige zur Verfügung. Jeder Zeiger ist darin farblich hervorgehoben. Durch Anklicken mit der Maus wird die durch den Zeiger refernzierte Struktur angezeigt.
Abbildung 2.2 zeigt die Vorgehensweise beim Durchmustern einer verketteten Datenstrukur. Die dargestellten Strukuren sind die Repräsentationen von Makrozellen und deren Vorkommen in der internen `CADIC`-Datenstrukur. Der Knopf 'Previous Topic' im Aktionsfeld des Dialoges ermöglicht das "Zurückhangeln" nach verfolgten Hypertextverweisen.
Informationspuffer vom Typ konstantes oder dynamisches Feld werden je nach Elementtyp verschieden angezeigt. Felder vom Typ `char` werden wie fortlaufender Text behandelt. Sind die Elemente eines Feldes vom Typ `char*` oder Struktur, werden diese untereinander angeordnet. Alle anderen Felder werden als eine durch Semicolon getrennte Liste der Elemente dargestellt.
Visualisierung in textueller Form kann auch an der **Baumdarstellung der Hierar-**

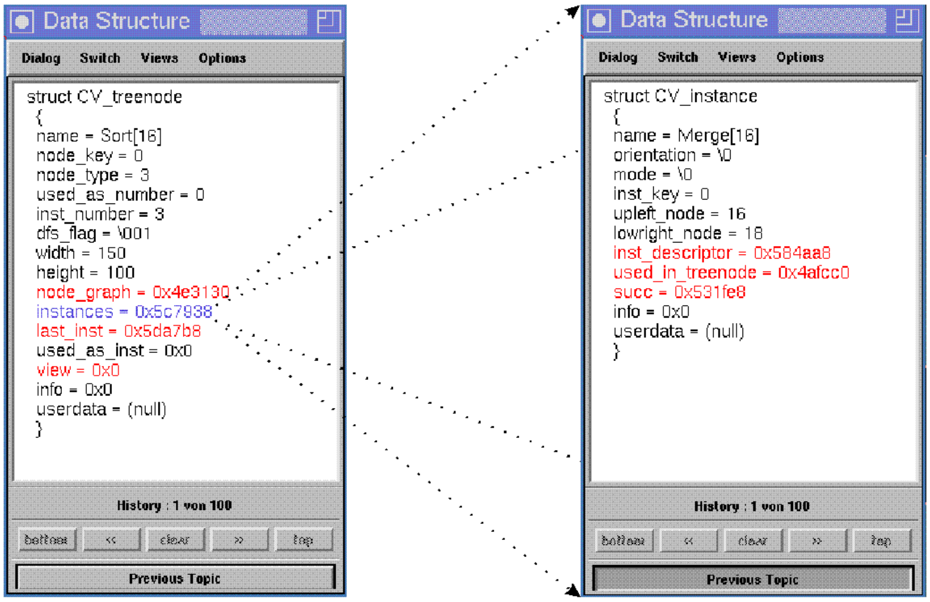

Abbildung 2.2: hypertextähnliche Darstellung der internen `CADIC`-Datenstrukur

chie erfolgen. Diese spezielle Art der Anzeige wird im Abschnitt 2.2.4 erläutert.

2.2 Graphische Anzeigemöglichkeiten

Neben den textuellen Darstellungsformen gibt es die Möglichkeit, gespeicherte Entwurfsdaten graphisch aufzubereiten. Dazu werden mindestens zwei Informationspuffer benötigt. Die Werte von genau einem der Puffer werden als x-Koordinaten interpretiert. Die Werte der anderen Puffer bilden zusammen mit den x-Koordinaten Paare der Form <x,y>. Jeder neu abgespeicherte y-Wert wird mit dem letzten (also aktuellstem) x-Wert kombiniert. Auf diese Weise erhält man Punkte in einem Koordinaten-System. Durch Verwendung von mehr als zwei Informationspuffern für die y-Werte kann man parallel auch mehrere Punktpaarfolgen anzeigen.
Die Aufbereitung dieser Daten kann nun auf verschiedene Weise erfolgen. In den folgenden Abschnitten werden die möglichen Diagrammtypen mit ihren Optionen vorgestellt.

2.2.1 Punkt- und Liniendiagramme

Zunächst werden Diagramme in Form von **Punkt- und Liniendiagrammen** betrachtet (siehe Abbildung 2.3).
Der Werkzeugentwickler kann durch geeignete Wahl von Parametern das Aussehen des Diagramms selbst gestalten. So sind alle Texte wie Titel oder Beschriftung der x- und y-Achse sowie Vorder- und Hintergrund und die Linienfarben einstellbar. Das Aussehen der Kurven verändert man durch die Auswahl von Punkt- und Linienstil. Beispielsweise

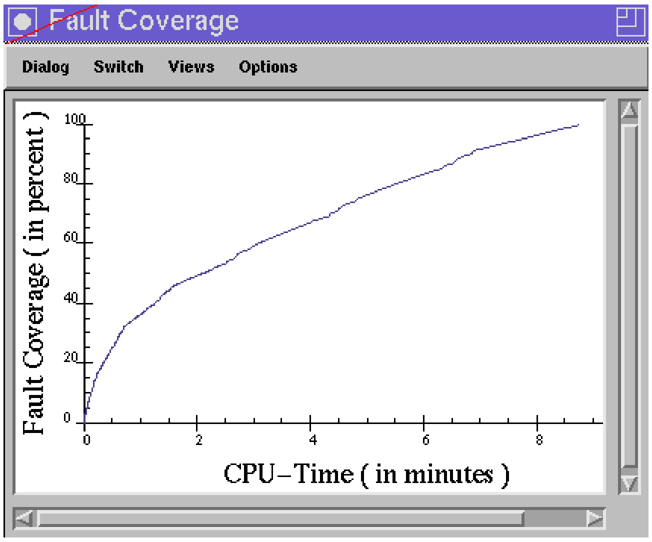

Abbildung 2.3: Fehlerüberdeckungskurve bei Anwendung eines Testmustergenerators

kann man die Punktpaarfolgen direkt abtragen, oder diese zu Kurven verbinden lassen. Dabei können die Punkt- und die Linienformen nach Bedarf eingestellt werden. In Abbildung 2.3 wurde als Punktart '`NONE`' und als Linienart '`SOLID`' gewählt, d.h. daß die Punkte selbst nicht, aber Geraden zwischen den Punkten gezeichnet werden.

2.2.2 Timing-Diagramme

Als besondere Form der graphischen Aufbereitung kann der Werkzeugentwickler **Timing-Diagramme** erzeugen. Dies ist vor allem bei der Analyse von Schaltkreisen äußerst hilfreich.

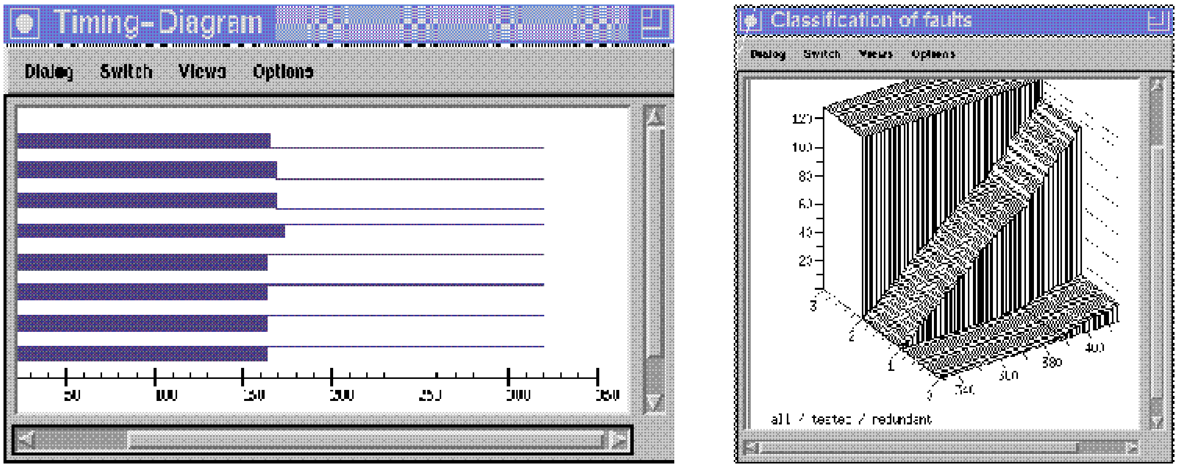

Abbildung 2.4: Beispiele für Timing-Diagramme und Histogramme

Der Wertebereich ist bei diesem Diagrammtyp auf die Konstanten `SIGNAL_LOW`, `SIGNAL_HIGH` und `SIGNAL_UNDEF` eingeschränkt. Es können mehrere Signale gleichzeitig dargestellt werden. Abbildung 2.4 zeigt ein Timing-Diagramm, welches zu einem Bus der Breite 8 korrespondiert. Wie bei den anderen Diagrammen auch, kann man den Titel, die Farben und die Achsenbeschriftungen nach Bedarf einstellen.

2.2.3 Histogramme

Der Werkzeugentwickler kann als weitere Anzeigeform **Histogramme** wählen. Dazu steht eine räumlich animierte Darstellung in Form eines Balkendiagramms zur Verfügung (siehe Abbildung 2.4). Durch die räumliche Anzeige können mehrer Kurven hintereinander abgebildet werden. Das Histogramm kann mit der Maus in allen Richtungen gedreht werden (dadurch ist auch eine 2-dimensionale Sicht einstellbar). Weiterhin kann jede Koordinatenachse gedehnt oder gestaucht werden.

2.2.4 Baumdarstellung der Schaltungshierarchie

In `CADIC` wird im Arbeitsfenster stets nur die jeweils aktuelle Hierarchieebene angezeigt. Zwar kann man mit Hilfe der zur Verfügung stehenden Navigationsmöglichkeiten die geladene Spezifikation inspizieren, einen Gesamtüberblick über die Schaltungshierarchie erhält man damit jedoch nicht. Zu diesem Zweck kann sich der Werkzeugentwickler eine baumartige Darstellung der Schaltungshierarchie erzeugen lassen (siehe Abbildungen 2.5).
Jeder Knoten entspricht einem Vorkommen einer Basis- oder Makrozelle in der geladenen Schaltung. Es kann zwischen gefalteter, teilweise expandierter und vollständig expandierter Darstellung gewechselt werden. Die Baumdarstellung eignet sich sehr gut zur Visualisierung von Zell- oder Instanzinformationen. In Abbildung 2.5 ist zum Beispiel die Anzahl der getesteten, bereits als redundant erkannten und aller eingepflanzten stuck-at-Fehler an den Zellvorkommen dargestellt (während des Ablaufs eines Testmustergenerators). Die jeweils angezeigten Textinformationen und Einfärbungen entsprechen den direkt im Arbeitsfenster dargestellten Daten bzw. deren Vorder- und Hintergrundfarben.

2.3 Einfärben von Objekten

Eine weitere Möglichkeit Entwurfsdaten zu visualisieren, besteht im **Einfärben von Objekten**. Darunter versteht man die Färbung eines Objekts der Schaltkreisdarstellung selbst bzw. die Änderung der Vorder- oder Hintergrundfarbe einer Anzeigeform bei diesem Objekt.
Bei kleinen Objekten wie Ein- und Ausgängen von Zellen, Verdrahtungs- oder Fanoutknoten zeigt ein farbiger Rahmen um das Objekt die jeweilige Farbe an. Zellen und Instanzen werden nur am äußeren Rand eingefärbt. Eine vollständige Färbung würde

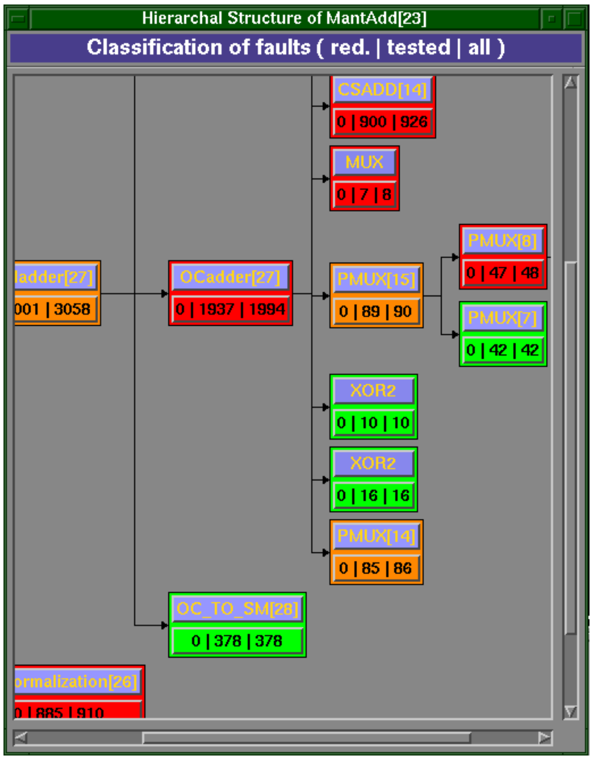

Abbildung 2.5: Textuelle Anzeige an der Baumdarstellung der Hierarchie

mit der Selektion einer Zelle per Maus in Konflikt treten, da bei dieser Selektion das Innere der am nächsten gelegenen Zelle farblich hervorgehoben wird.
Neben der Färbung eines Objektes selbst, kann man auch die Vorder- oder Hintergrundfarbe einer Anzeigeform von einem Informationspuffer abhängig machen. Dies gilt für textuelle und graphische Darstellungsarten.
In Abbildung 2.6 sind zum Beispiel die Instanzen eingefärbt und der Hintergrund der Anzeige der klassifizierten Fehler pro Modul farblich hervorgehoben. Fehlerüberdeckungen von weniger als 95 Prozent werden zum Beispiel mit rot, von mehr als 99,5 Prozent mit grün (also akzeptabel) und Zwischenwerte mit orange und gelb charakterisiert.

2.3.1 Trace-View

In `CADIC` gibt es zur Veranschaulichung des zurückgelegten Pfades während der Navigation durch die Hierarchie der Schaltung eine Darstellungsform **Trace-View**. Wie in Abbildung 2.7 zu sehen, werden die besuchten Hierarchieebenen des verfolgten Pfades nacheinander aufgereiht. Auf jeder Ebene ist das Zellvorkommen, in welches abgestiegen wurde, hervorgehoben. Die Anzeige Trace-View reagiert auf weitere Navigation entsprechend. An jeder der Ebenen kann die Färbung der Objekte betrachtet werden. Die Anzeige anderer Informationen an der Trace-View-Darstellung wird aus Gründen der Übersichtlichkeit nicht unterstützt.

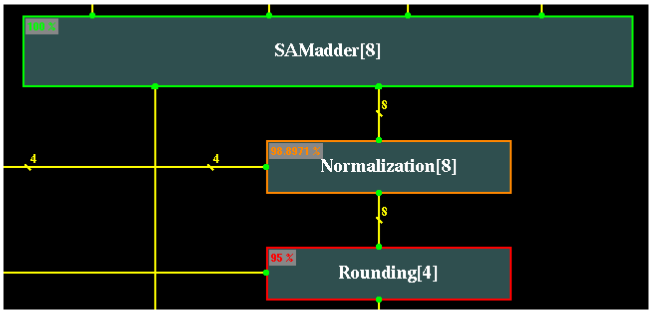

Abbildung 2.6: Visualisierung durch Farben

Im Beispiel ist die Färbung von Verdrahtungskanten durch einen in `CADIC` integrierten Schichtzuweisungsalgorithmus [Gra92], [Mol93] zu sehen. Die Verdrahtungskanten der Schaltung sind entsprechend ihrer zugwiesenen Schicht (grün - Schicht 1 , rot - Schicht 2 , grau - Busleitung mit Einzelleitungen verschiedenener Schichten) eingefärbt sowie alle eingefügten Schichtwechselknoten markiert.

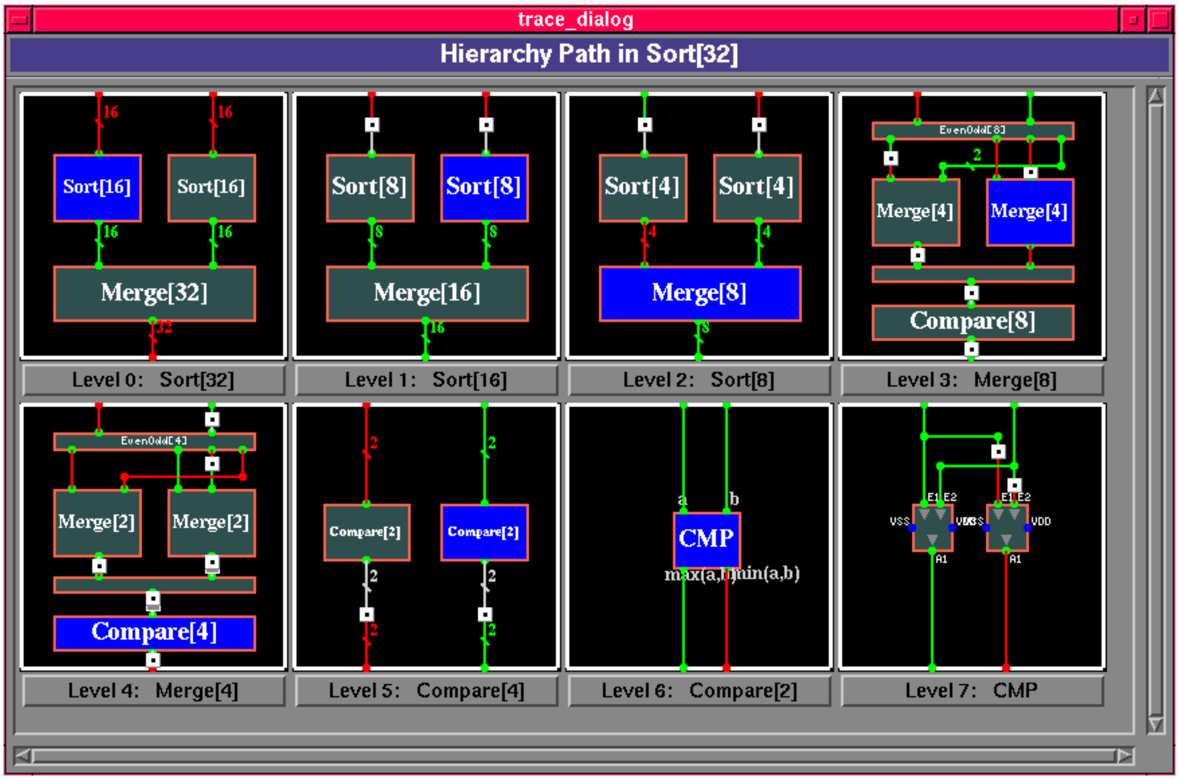

Abbildung 2.7: Trace-View : farbliche Anzeige der zugewiesenen Schicht

Kapitel 3

Konfigurierbarkeit

Die verschiedenen Entwurfswerkzeuge stellen sehr unterschiedliche Anforderungen an eine Visualisierung und jeder Benutzer hat seine individuellen Vorstellungen über die Anzeige der Daten.

Die Visualisierung sollte deshalb konfigurierbar sein, d.h. jeder Werkzeugentwickler aber auch jeder Benutzer kann die Anzeige nach seinen Wünschen gestalten. Der Programmierer nimmt die grundlegenden Einstellungen für sein Werkzeug vor. Die späteren Nutzer der Entwurfstools können unter den vorhandenen Anzeigevarianten auswählen und haben durch die Bedienung ebenfalls Einfluß auf die Gestaltung der dargestellten Informationen.

Eine der zentralen Eigenschaften von `CADIC` ist der modulare Aufbau, d.h. die zugrundeliegende Datenstrukur, der Schematic Editor, die graphische Oberfläche und jedes einzelne Entwurfswerkzeug sind in separaten Funktionsbibliotheken realisiert. Dadurch können einige Werkzeuge wahlweise mit oder ohne graphische Oberfläche eingesetzt werden.

Um diese Modularität zu erhalten, muß die Visualisierung selbst durch eine solche, klar abgegrenzte Funktionsbibliothek implementiert werden. Visualisierung ohne graphische Oberfläche macht natürlich wenig Sinn, so daß eine Kopplung dieser beiden Teile nötig ist. Umgekehrt muß die Visualisierung jedoch von der graphischen Oberfläche getrennt werden können. Analoges gilt für die Beziehung zu den Entwurfswerkzeugen.

Zur Anzeige der Informationen braucht man aber andererseits eine intensive Kommunikationen zwischen der Visualisierung und den angeschlossenen Werkzeugen.

Zusammenfassend teilt sich die Konfiguration insgesamt in die zwei folgenden Teile :

1. die Beschreibung der zu visualisierenden Informationen und der Anzeige zu einem Entwurfswerkzeug

2. die Bereitstellung einer Schnittstelle, um die Kommunikation zwischen den Entwurfswerkzeugen und der Visualisierung zu ermöglichen

Der erste Teil wird im folgenden Abschnitt behandelt, während die Schnittstelle zu den Entwurfswerkzeugen in Abschnitt 3.2 beschrieben wird.

3.1 Konfiguration des Moduls Visualisierung

Die Konfiguration der Visualisierung umfaßt einerseits die Beschreibung des Werkzeugentwicklers, welche Datentypen an welchen Objekten in welcher Art und Weise angezeigt werden sollen. Die andererseits vorhandenen Gestaltungs- und Einflußmöglichkeiten auf die Anzeige durch die Benutzer von `CADIC` werden im Abschnitt 3.4 besprochen. Im folgenden sollen die auftretenden Probleme skizziert werden, die durch die Auswertung einer solchen Beschreibung durch einen Rechner entstehen.

Die klare Trennung zwischen den Entwurfswerkzeugen und der Visualisierung erzwingt eine Duplizierung und Speicherung der übermittelten Daten. Das Kopieren der Informationen verlangt aber die Kenntnis über deren Datentyp. Betrachtet man die im Kapitel 1 vorgestellte Menge T an unterstützten Datentypen, enthält diese insbesondere Strukturen. Zur Anzeige von Informationen des Typs Struktur oder Zeiger auf Struktur müssen die entsprechenden Deklarationen zugänglich sein. Ein Teil der Konfiguration muß also in der **Deklaration von Strukturen** bestehen.
Nachdem die Daten zum Modul Visualisierung geschickt wurden, müssen diese ihrem Objekt zugeordnet, kopiert und gespeichert (außer bei temporären Informationen) werden. Ein Objekt kann mehrere Informationspuffer besitzen (siehe Seite 22). Demzufolge ist eine **Zuordnung der Daten zu den Informationspuffern** in der Beschreibung erforderlich.
Der Hauptteil der Beschreibung besteht aus der **Konfiguration der gewünschten Anzeigeformen** bei den Objekten. Ziel ist dabei die einfache Konstruktion von textuellen und graphischen Darstellungen (wie z.B. von Diagrammen) aus den gespeicherten Entwurfsdaten.

Eine naheliegende Idee zur Realisierung der Konfiguration ist die Definition einer aus Anwendersicht einfachen Sprache.
Der Werkzeugentwickler gestaltet in dieser "Beschreibungssprache" die Visualisierung nach seinen Vorstellungen.

3.1.1 Die Beschreibungssprache

Die ersten Schritte eines Werkzeugentwicklers zur Visualisierung von Entwurfsdaten bestehen darin, seine Vorstellungen über die Darstellung der Informationen in Konstrukte der Beschreibungssprache umzusetzen. In diesem Abschnitt werden die Gliederung und die wichtigsten Konstrukte dieser Beschreibungssprache erläutert. Die vollständige Syntax der Beschreibungssprache ist im Anhang C wiedergegeben. Die der Sprache zugrundeliegenden Grammtik ist vom Typ LR(1) und wird durch einen automatisch generierten Scanner/Parser analysiert [LMB92].
Die Beschreibung zu einem Entwurfswerkzeug wird in einer Datei

<Werkzeug_Name>.config

gespeichert. Zu Beginn wird eine Kennung vergeben, um diese Beschreibung von denen anderer Werkzeuge zu unterscheiden (siehe auch Kapitel 6). In der Syntax der Sprache benutzt man dazu :

```
MODUL = "LayerAssignment"
```

Anschließend gliedert sich die Beschreibung in drei Hauptteile. Davon sind der erste und der dritte Hauptabschnitt optional.

1. Hauptabschnitt : Deklaration von Strukturen

Der erste Abschnitt dient ausschließlich der Deklaration von Strukturen. Falls keine Informationen vom Typ Struktur visualisiert werden sollen, bleibt dieser Teil leer. Betrachten wir nochmals das Beispiel aus Kapitel 1.

Beispiel 1: *Strukturdeklarationen*

```
struct Netzliste
  {
  long          anz_net;
  struct Netze  netze[anz_net];
  ...
  }
```

□

Das dynamische Feld ist hier entsprechend der Syntax der Beschreibungssprache deklariert worden. Wird die Länge eines Feldes innerhalb einer Struktur durch eine andere Strukturkomponente bestimmt, wird deren Name in eckigen Klammern dem Feld angehängt (betrachte dazu das Feld von Netzen der Länge `anz_net`).
Felder numerischen Typs, deren erste Komponente die Länge des Feldes angibt, werden durch das Konstrukt

```
puffer_typ puffer_name[INDEX_0];
```

deklariert. Durch die Modifikationen aufgrund von dynamischen Feldern ändert sich die interne Darstellung der Struktur gegenüber dem Original. Deshalb werden intern die Namen der Strukturen mit Präfixen versehen, um Namenskonflikte zu vermeiden.

2. Hauptabschnitt : Definition von Informationspuffern

Im zweiten Abschnitt der Beschreibung soll der Werkzeugentwickler die erforderlichen Informationspuffer bei den Objekten einrichten können. Zur Charakterisierung eines Informationspuffers muß dessen Name, Typ und Länge festgelegt werden. Betrachtet man die die üblichen Variablendeklarationen,

<variablen_typ> <variablen_name>;
<variablen_typ> <variablen_name> [...];

so sind diese bis auf die Längeangabe gut zur Deklaration von Informationspuffern geeignet. Dazu werden folgende Blöcke verwendet :

```
HISTORY = 35                    /* Festlegung der Länge der folgenden */
  {                             /* Informationspuffer */
  puffer_typ1 puffer_name1;
  ...
  puffer_typn puffer_namen;
  }
```

Alle Deklarationen innerhalb eines solchen Blocks erhalten die angegebene Länge (hier 35). Deklariert man Informationspuffer außerhalb eines Blockes wird standardmaßig der Wert 100 als Länge eingestellt. Informationspuffer, die von der Busbreite des betreffenden Elements abhängen, werden nach dem folgenden Schema deklarariert :

```
puffer_typ puffer_name[BUSWIDTH];
```

Problematisch ist nun die Zuordnung der Informationspuffer zu den Objekten. Schon eine einfache Schaltung kann mehrere zehntausende Instanzen und somit ein Vielfaches mehr an Objekten enthalten. Die Festlegung von Informationspuffern bei einzelnen Objekten ist also nicht durchführbar. Sie ist auch nicht sinnvoll, da ein Entwurfswerkzeug im allgemeinen auf beliebigen Spezifikationen eingesetzt werden wird, die Objektpositionen zur eindeutigen Bestimmung eines Objekts aber entwurfsspezifisch sind.
Zur Vereinfachung werden deshalb die Informationspuffer den Objektklassen statt den einzelnen Objekten zugeordnet. Dazu bildet man für jede Klasse einen Block wie im folgenden Beispiel :

Beispiel 2: *Deklaration von Informationspuffern*

```
HRY_WIRE
  {
  HISTORY = 1
    {
    Color layer;
    }
  }
```

□

Hier wird bei allen Objekten der Klasse `HRY_WIRE` ein Informationspuffer der Länge 1 vom Typ Color eingerichtet, um die Verdrahtungskanten entsprechend der gespeicherten Farbe zu kennzeichnen.

3. Hauptabschnitt : Auswahl und Konfiguration der Anzeigeformen

Im letzten Abschnitt der Beschreibungsdatei kann der Anwender Darstellungsformen definieren und konfigurieren. Dabei können alle bereits deklarierten Informationspuffer der betreffenden Objektklasse sowie von `GLOBAL` benutzt werden. Unterbleibt die Beschreibung der Anzeige zu einer Klasse, wird zumindest eine Standardausgabe textueller Art ermöglicht. Diese besteht aus der kombinierten Anzeige aller Informationspuffer der betreffenden Klasse (zeilenweise nacheinander angeordnet). Wie in Abschnitt 2 werden die Anzeigemöglichkeiten für jede Objektklasse separat in einem Block beschrieben.

```
DISPLAY ( SIGNAL )
  {
  /* Auswahl und Konfiguration der Anzeigeformen zur Objektklasse SIGNAL */
  }
```

Innerhalb eines solchen Blocks können nacheinander verschiedene Anzeigeformen deklariert werden. Diese lassen sich in 5 Gruppen einteilen :

- Anzeige direkt in der Schaltkreisdarstellung (`DIRECT`)
- Einfärben der Objekte (`COLOR`)
- textuelle Darstellung im Dialogfenster am Objekt (`ITEM`)
- graphische Darstellung im Dialogfenster am Objekt
  - (`TIMING_DIAGRAM` , `DIAGRAM`, `HISTOGRAM`)
- gemischte Darstellung im Dialogfenster am Objekt
  - (`MATRIX(column)` , `ROW`, `COLUMN`)

Jedes der aufgeführten Konstrukte besitzt wieder selbst einen Rumpf, worin die zugehörigen Einstellungen vorgenommen werden können.
Unter der gemischten Anzeige im Dialogfenster ist zu verstehen, daß textuelle Darstellungen und Grafiken zeilen-, spalten,- oder matrizenförmig angeordnet werden. Im Rumpf der Konstrukte `MATRIX` , `ROW`, oder `COLUMN` können also die Elemente der dritten und vierten Gruppe verwendet werden. Eine Matrix wird zeilenweise unter Beachtung der angegebenen Spaltenzahl aufgefüllt. Da nicht jede der aufgeführten Anzeigeformen bei allen Objektklassen definiert ist, zeigt eine Tabelle im Anhang B die Verfügbarkeit bei den einzelnen Objektklassen.
Beispielhaft sollen jetzt die Anzeigeformen `DIRECT`, `ITEM`, `COLOR` und `HISTOGRAM` erläuert werden.
Man vergleiche den folgenden Auszug aus einer Beschreibungsdatei mit der Abbildung 2.1 auf Seite 26.

```
MODUL = FanoutCorrection

PIN
  {
  double arr_time;          /* 3 Informationspuffer des Typs */
  double req_time;          /* double werden bei der */
  double slack;             /* Objektklasse PIN eingerichtet */
  }
```

```
DISPLAY( PIN )                /* Konfiguration der Anzeigeformen */
  {                           /* zur Objektklasse PIN*/
  DIRECT                      /* Konstrukt schreibt direkt in das */
    {                         /* Arbeitsfenster von CADIC */
    NAME = “FanoutCorrection: Fanout AT/RT/Slack“
    FOREGROUND = “black“
    BACKGROUND = “white“
    “RT = “  req_time  “\nAT = “  arr_time  “\nS = “  slack
    }
  ITEM                        /* Konstrukt erzeugt eine Anzeigeform im */
    {                         /* Dialogfenster bei dem betreffenden Objekt */
    NAME = “FanoutCorrection: Fanout correction“
    “required time \t= “  req_time  “\n“
    “arrival time \t= “  arr_time  “\n“
    “-------------------------\n“
    “slack \t= “  slack
    }
  }
```

Dies entspricht genau der Konfiguration, mit Hilfe derer der Bildschirmabzug auf Seite 26 erstellt wurde. Das zugehörige Entwurfswerkzeug optimiert die Laufzeit von Schaltungen durch Einfügen von Treiberbäumen [Kre96].
Die textuellen Anzeigeformen wurden aus konstanten Texten (in Hochkomma, z.B. “`required time \t=` “) und den aktuellen Inhalten der angeführten Informationspuffer kombiniert. Jeder Darstellungsform kann ein Name vergeben werden. Definiert man mehrere Anzeigeformen des gleichen Typs, kann so zwischen diesen umgeschaltet werden (siehe Abschnitt 3.4). Die Festlegung von Vorder- und Hintergrundfarbe ist optional. Standardmäßig wird schwarz für den Vordergrund und weiß für den Hintergrund verwendet.
Die Abbildung 2.4 auf Seite 28 wurde durch folgenden Teil einer Beschreibungsdatei erzeugt.

```
...
INSTANCE
  {
  ...
  HISTORY = 1 { long tic; }  /* Zähler */
  HISTORY = 500
    {
    long sum_of_faults;       /* diese Informationspuffer speichern */
    long sum_of_tested;       /* die Summe der getesteten, redundaten */
    long sum_of_red;          /* und aller eingepflanzten Fehler */
    }                         /* innerhalb eines Zellvorkommens */
  ...
  }
...
```

```
DISPLAY( INSTANCE )
  {
  ...
  HISTOGRAM                  /* erzeugt ein Histogramm */
    {
    NAME = "Testgenerator: Classification of faults"
    XVAR = tic
    YVAR = sum_of_faults, sum_of_tested , sum_of_red
    YLABEL = "all / tested / redundant"
    }
  ...
  }
...
```

Bei den graphischen Darstellungsformen werden die Informationspuffer einer Achse des Koordinatensystems zugeordnet. Im obigen Beispiel werden Punktpaare $<x, y_1, y_2, y_3>$ gebildet, wobei x die Werte von "`tic`" und die y_i die jeweiligen Werte der Informationspuffer von `YVAR` annehmen.
Die anderen Anzeigeformen sind analog zu den bereits besprochenen zu konfigurieren. So wird zum Beispiel eine Verdrahtungskante mit

```
...
DISPLAY( HRY_WIRE )
  {
  ...
  COLOR
    {
    NAME = "LayerAssignment: Color of Edge"
    col
    }
  ...
  }
...
```

einfärbt, wenn der Informationspuffer `col` vom Typ `Color` ist.

3.1.2 Generierung der Funktionsbibliotheken

Nachdem der Werkzeugentwickler die Vorstellungen über die Anzeige seiner Informationen in Konstrukte der Beschreibungssprache umgesetzt hat, besteht nun noch die Aufgabe, daraus die benötigten Funktionen zu generieren, welche die eigentliche Visualisierung im Programm übernehmen.
Schematisch soll dies durch die Abbildung 3.1 verdeutlicht werden. Aus einer werkzeugspezifischen Beschreibungsdatei wird durch Anwendung einer Scanner/Parser-Kombination Quellcode erzeugt, der den konfigurierten bzw. dynamischen Teil der Visualisierung ausmacht. Alle von der Konfiguration unabhängigen Bestandteile sind

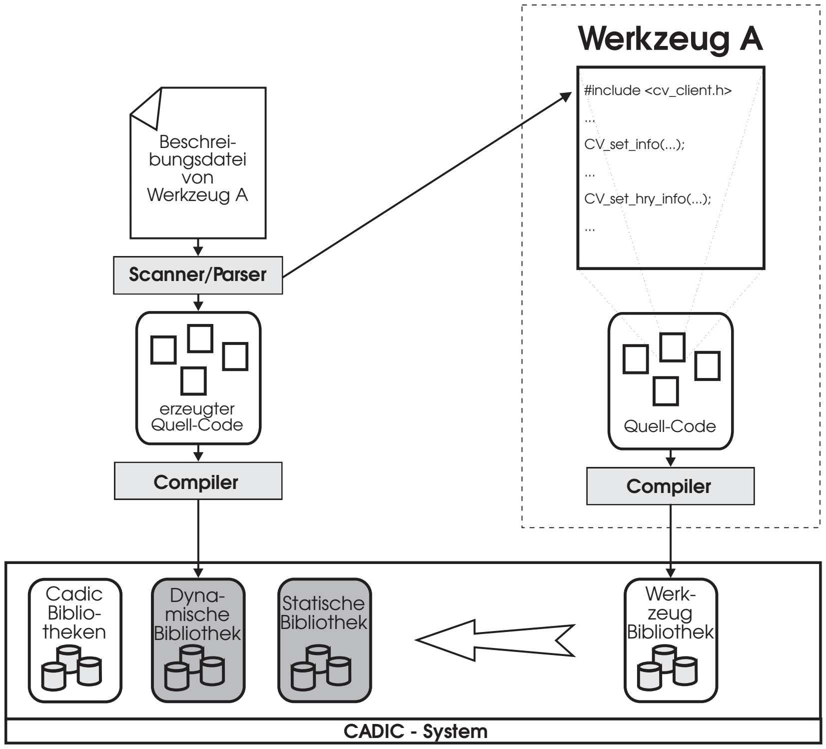

Abbildung 3.1: Integration eines Entwurfswerkzeuges

in der statischen Bibliothek enthalten. Der Aufruf der Scanner/Parser-Kombination lautet wie folgt :

CV_parser <Beschreibungsdatei>
<m_include> <c_include> <h_output> <l_output>

Das 2. und 3. Argument entspricht jeweils einem Pfadnamen, welcher angibt, wo die Motif-Header- bzw. die `CADIC`-Header-Dateien zu finden sind. Die beiden letzten Argumente bestimmen, in welchen Unterverzeichnissen die generierten Dateien (Header-Dateien und Bibliotheken) abgelegt werden sollen.

Neben der Kommandozeilenversion besteht auch die Möglichkeit, die Erzeugung der benötigten Dateien durch die Konfigurationsoberfläche `Cavis` vorzunehmen (siehe auch Abschnitt 3.3). Nachdem die zur Generierung notwendigen Verzeichnisse einmalig eingestellt wurden, können die Bibliothken nach Änderungen in der Beschreibung per Knopfdruck erzeugt werden.

3.2 Die Schnittstelle zum Entwurfswerkzeug

Der zweite wesentliche Punkt bei der Konfiguration der Visualisierung besteht in der Bereitstellung einer Schnittstelle zu den Entwurfswerkzeugen. Die Funktionen dieser

Schnittstelle sollen es dem Werkzeugentwickler ermöglichen, die anzuzeigenden Informationen auf einfache Art und Weise zur Visualisierung zu schicken. Notwendige Parameter zum Versenden von Informationen sind

- eine Objektklasse, zur Festlegung, an welchem Element der Schaltkreisdarstellung visualisiert werden soll,
- eine Objektposition, zur Adressierung eines bestimmten Objekts,
- ein Name eines Informationspuffers, zur Auswahl eines solchen bei dem Objekt,
- und die zu visualisierenden Daten.

Die Informationen werden per Referenz an die Visualierung übergeben. Dadurch kann als allgemeiner Typ für die Entwurfsdaten `void*` festgelegt werden.
Die Schnittstelle muß Funktionen enthalten, welche die Informationen von integrierten und von externen Entwurfswerkzeugen absenden können. Bei eigenständigen Programmen, sind zunächst die Voraussetzungen zur Kommunikation zwischen dem Werkzeug und der graphischen Oberfläche von `CADIC` zu schaffen. Diese Grundlagen werden in Kapitel 4 behandelt. Die Funktionen der Schnittstelle bauen darauf auf und erlauben das Versenden der Entwurfsdaten auf einfache Weise.
Falls Entwurfswerkzeuge wahlweise mit oder ohne graphische Oberfläche bzw. Visualisierung eingesetzt werden sollen, können die Schnittstellenfunktionen und die notwendigen Headerdateien durch Präprozessordirektiven eingeschachtelt werden :

```
#ifdef CADIC_VISUALIZATION
  ...
#endif /* CADIC_VISUALIZATION */
```

3.2.1 Schnittstelle für integrierte Entwurfswerkzeuge

Entwurfswerkzeuge, die in `CADIC` integriert sind, können wahlweise Informationen zum hierarchisch gegliederten oder zum flachen Schaltkreis visualisieren.

1. Schnittstellenfunktion zum Versenden von Informationen zum hierarchisch gegliederten Schaltkreis

```
void CV_set_hry_info( long zell_ord_nr,
                      long lokale_ord_nr,
                      long bus_komp,
                      unsigned long hier_obj_klasse,
                      char* puffer_name,
                      void* data );
```

2. Schnittstellenfunktion zum Versenden von Informationen zum flachen Schaltkreis

```
void CV_set_info( selector* sel,
                  long lokale_ord_nr,
                  long bus_komp,
                  unsigned long obj_klasse,
                  char* puffer_name,
                  void* data );
```

Da die Objektpositionen bei hierarchischen und flachen Objektklassen jeweils verschiedene Parameter benötigen, wurde zwei Funktionen in die Schnittstelle aufgenommen. Die ersten drei Parameter entsprechen dabei den Objektpositionen. Die Angabe des Parameters `bus_komp` ist nur dann notwendig, falls der Informationspuffer bei dieser Klasse von der Busbreite des Elements abhängt. Bei den Sonderfällen `GLOBAL` und `TEMPORARY` sind die ersten drei Parameter redundant.
Die beiden letzten Argumente bestimmen den Namen eines Informationspuffers bei der Objektklasse und eine Referenz auf die zu übermittelnden Entwurfsdaten. Die Objektklassenbezeicher sind Konstanten, die durch die eingebundenen Header-Dateien definiert werden. Sie entprechen den Namen der Objektklassen mit vorangestelltem "`CV_`" (z.B. `CV_PIN` für die Objektklasse `PIN`). Diese Ergänzung wurde vorgenommen, da häufig Konstanten wie `PIN` oder `PAD` bei der Implementierung von Entwurfswerkzeugen verwendet werden. Die Verwendung einzelnen Schnittstellenfunktionen wird im Kapitel 7 nochmals an Beispielen gezeigt.

3.2.2 Schnittstelle für externe Entwurfswerkzeuge

Die Anbindung externer Programme kann für Werkzeuge, welche auf der `CADIC`-Datenstruktur als auch für Werkzeuge, die auf der `test_circ`-Datenstruktur [KHB93] basieren, erfolgen. Die zu visualisierenden Informationen werden über die folgenden Schnittstellenfunktionen abgesendet :

- Externes Programm basierend auf der `CADIC`-Datenstruktur

```
void CV_send_hry2cadic( long zell_ord_nr,
                        long lokale_ord_nr,
                        long bus_komp,
                        unsigned long hier_obj_klasse,
                        char* puffer_name,
                        void* data );

void CV_send2cadic( selector* sel,
                    long lokale_ord_nr,
                    long bus_komp,
                    unsigned long obj_klasse,
                    char* puffer_name,
                    void* data );
```

- Externes Programm basierend auf der `test_circ`-Datenstruktur

```
void CV_tcsend2cadic( long c_ord,
                      long pin_nr,
                      long bus_komp,
                      unsigned long obj_klasse,
                      char* puffer_name,
                      void* data );
```

Die Bedeutung der Parameter bei Programmen basierend auf der `CADIC`-Datenstruktur wurde bereits besprochen. Die bei externen Entwurfswerkzeugen anzuwendenden Schnittstellenfunktionen unterscheiden sich von den anderen Funktionen, da diese die Informationen über ein Netzwerk an die graphische Oberfläche übertragen.
Für Werkzeuge basierend auf der `test_circ`-Datenstruktur sind andere Parameter anzugeben. Bevor deren Bedeutung erklärt wird, folgt zunächst eine kurzer Überblick zu dieser Datenstruktur.

Die `test_circ`-Datenstruktur wird hauptsächlich von Werkzeugen zum Testen von Schaltkreisen verwendet. Die verwendete Schaltkreisdarstellung entspricht einer flachen Netzliste. Der Bezug zur ursprünglich hierarchisch gegliederten Schaltung ist nicht mehr vorhanden. Um Entwurfsdaten an der graphischen Oberfläche von `CADIC` anzuzeigen, mußte ein Werkzeugentwickler bisher selbst die Korrespondenz zwischen den verschiedenen Datenstrukturen und auch den Bezug zur Hierarchie wieder herstellen. Durch die Erweiterung der Schnittstelle um Funktionen zur Anbindung von Werkzeugen basierend auf der `test_circ`-Datenstruktur, wird dem Entwickler ein erheblicher Implementierungsaufwand erspart.

Externe Programme mit `test_circ`-Schaltkreisdarstellung können Informationen zu Signalnetzen, Zellen und deren Vorkommen visualisieren (`SIGNAL`, `CELL`, `INSTANCE`). Dazu existiert eine Abbildung von den globalen Ordungsnummern in `test_circ` zu den Selektoren in `CADIC` ($0 \leq c\_ord < no\_of\_cells()$). Daten zu den Zellen werden durch Verwendung der Beschreibung des durch die Abbildung gefundenen Zellvorkommens übermittelt. Signalnetze addressiert man durch Angabe einer Zellordnungsnummer und der Nummer eines Pins ($0 <= pin\_nr < no\_of\_pins(cell\_id)$) dieser Zelle, mit welchem das Signal verbunden ist. Das 2. Argument ist demzufolge nur zur Adressierung von Signalen notwendig. Das 3. Argument ist stets mit 0 zu besetzen.

3.3 Einfache Konfiguration durch Cavis

Zur einfachen Handhabung der Konfiguration der Visualisierung wurde eine kleine graphische Oberfläche `Cavis` implementiert (siehe Abbildung 3.2).
Mit Hilfe dieses Programms kann der Benutzer

- Beschreibungsdateien laden, editieren und speichern,

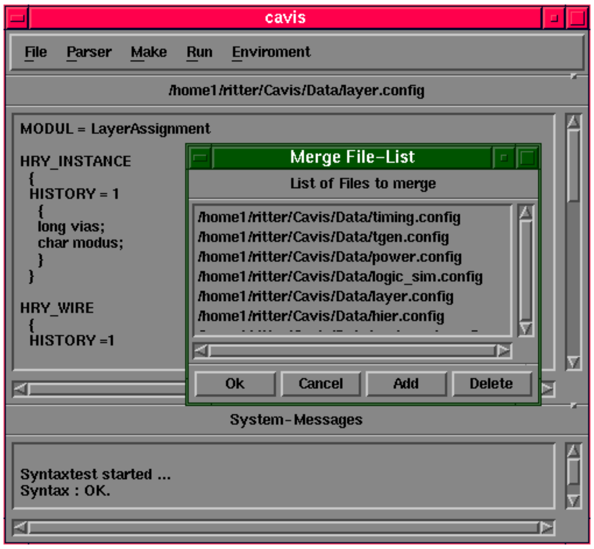

Abbildung 3.2: die Konfigurationsoberfläche `Cavis`

- aus verschiedenen Beschreibungsdateien eine neue Gesamtbeschreibung bilden (Erläuterung im Kapitel 6),
- die geladene Beschreibungsdatei auf korrekte Syntax testen,
- aus der geladenen Beschreibungsdatei die Header-Dateien und die Funktionsbibliotheken erzeugen,
- die Funktionsbibliotheken übersetzen und
- `CADIC` bzw. externe Programme binden und starten.

Nachdem die Beschreibung zur Visualisierung für ein Werkzeug erstellt wurde, kann der Entwurf auf syntaktische Fehler getestet werden. Sind tatsächlich Fehler enthalten, wird beim Syntaxtest die fehlerhafte Zeile markiert, und diese wenn nötig in das sichtbare Fenster gescrollt. Dadurch kann man Fehler schnell lokalisieren und mit Hilfe der ausgegebenen Hilfsmeldungen auch korrigieren. Aus der entworfenen Beschreibung können die benötigten Funktionsbiblioteken und Headerdateien generiert werden.
Abbildung 3.2 zeigt auch, wie man Gesamtkonfigurationen erstellt. Dazu 'mischt' man mehrere Einzeldateien zusammen. Die Auswahl dieser Dateien erfolgt in dem dargestellten Dialog. Die resultierende Gesamtbeschreibung wird im Arbeitsfenster angezeigt.

3.4 Bedienung durch den Benutzer

Dieser Abschnitt soll dem Leser einen Eindruck von der Arbeit mit der Visualisierung in `CADIC` geben. Im folgenden werden die vorhandenen Möglichkeiten vorgestellt, mit Hilfe derer man während der Laufzeit auf die Visualisierung Einfluß nehmen kann. Dazu gehören eine Reihe von globalen Einstellungsmöglichkeiten , Funktionen zur Koordination der Labels und der Färbung der Objekte sowie die Handhabung der Objektdialoge. Weiterhin wird das Zusammenspiel der Visualisierung mit den Navigationsmöglichkeiten durch die Hierarchie erläutert. Zunächst wird auf die globale Funktionalität in der graphischen Oberfläche eingegangen.

3.4.1 Globale Funktionalität

In jeder Konfiguration der graphischen Oberfläche von `CADIC` ist ein Menüeintrag "Visualization" enthalten. Sobald die ersten Entwurfsdaten zur Anzeige geschickt worden sind, können die Funktionen dieses Menüs ausgeführt werden :

- Öffnen der Dialoge zu einer bestimmten Objektklasse
- An- oder Abschalten aller Labels bzw. der Labels einer bestimmten Objektklasse
- An- oder Abschalten aller Einfärbungen bzw. der Einfärbungen bezüglich einer Objektklasse
- Löschen der gesamten empfangenen Information bzw. der Information zu einer bestimmten Objektklasse

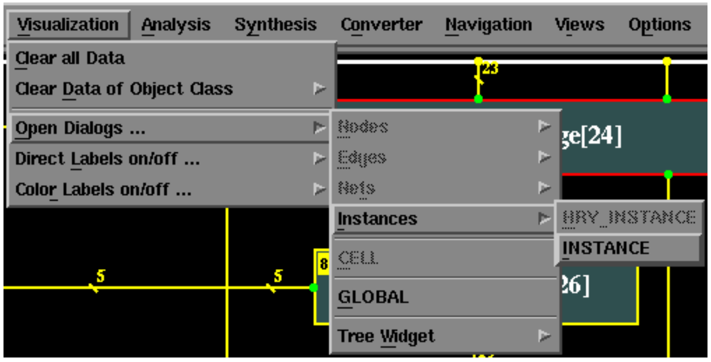

Abbildung 3.3: Menü "Visualization"

Zum Öffnen eines Dialoges wählt man den entsprechenden Menüpunkt und im anhängenden Untermenü eine Objektklasse aus (siehe Abbildung 3.3). Anschließend kann mit der Maus ein Objekt dieser Klasse selektiert werden. Bewegt man sich in die Nähe eines solchen Objekts, verändert sich der Cursor zu :

i , falls Informationen zu diesem Objekt vorhanden sind, zu

√ , falls der Objektdialog bereits geöffnet wurde, bzw. zu

 , falls zu diesem Objekt noch keine Daten empfangen worden sind.

Das An- bzw. Abschalten der Labels und Farben bzw. das Löschen von gespeicherten Informationen zu einer Objektklasse erfolgt analog durch Anwählen des entsprechenden Menüpunkts und der Objektklasse im Untermenü.
Bewegt man sich im folgenden durch die Hierarchie der geladenen Schaltung (*Trace-Down, Trace-Up*), werden beim Verlassen einer Ebene die offenen Objektdialoge geschlossen und alle angezeigten Labels verschwinden. An der neu betretenen Hierarchieebene werden die Labels und, falls diese Ebene schon einmal besucht worden ist, auch die eventuell schon geöffneten Objektdialoge wieder angezeigt. Falls die *Trace-View*-Darstellung geöffnet wurde, wird diese beim Absteigen in eine Instanz um diese Instanz erweitert bzw. beim Verlassen einer Hierarchieebene um diese verkürzt.
Zur Illustration langer Berechnungen innerhalb der Entwurfswerkzeuge kann der der Cursor in einen "busy cursor" verwandelt werden. Dies wird durch den Aufruf der Funktionen

```
CV_start_waiting( );
CV_stop_waiting( );
```

vor und nach den rechenzeitintensiven Berechnungen erreicht. Der Benutzer kann während dieser Zeit keine Aktionen auf der Oberfläche ausführen.

3.4.2 Koordination der Labels und Farben

Über Konstrukte der Beschreibungssprache besteht die Möglichkeit, gespeicherte Informationen direkt in das Arbeitsfenster an die Objekte zu schreiben bzw. diese einzufärben. Es existiert dabei jedoch keine Einschränkung in der Hinsicht, daß auch mehrere solcher Labels bzw. verschiedene Objektfärbungen durch den Benutzer angegeben werden. Um die Anzeige nicht zu überladen, kann jeweils nur ein Label und insbesondere nur eine Färbung pro Objekt dargestellt werden. Daraus entsteht die Notwendigkeit, zwischen verschiedenen Darstellungsformen dieser Art umzuschalten.
Zunächst können die Labels und Farben einer bestimmten Objektklasse bzw. alle zusammen, wie in Abschnitt 3.4.1 beschrieben, an- und abgeschaltet werden. Da der Wechsel zwischen den Darstellungsformen bei jedem einzelnen Objekt aufgrund der im allgemeinen großen Anzahl von Objekten nicht sinnvoll erscheint, wird deshalb das Umschalten für die Objekte einer Klasse insgesamt ermöglicht. Dazu benutzt man ein auf dem Arbeitsfenster installiertes Popup-Menü (rechte Maustaste läßt das Menü erscheinen). Darin verzweigt man zunächst nach Labels oder Farben (siehe Abbildung 3.4). Anschließend wählt man bei den Objektklassen unter den zur Verfügung stehenden Darstellungsformen die gewünschte aus. Zur Unterscheidung dienen dabei die in der Beschreibungssprache vergebenen Namen.

3.4.3 Handhabung der Objektdialoge

In diesem Abschnitt werden die Bedienungselemente der Visualisierungsdialoge vorgestellt. Dialogfenster zur Objektklasse `TEMPORARY` bilden eine Ausnahme und werden

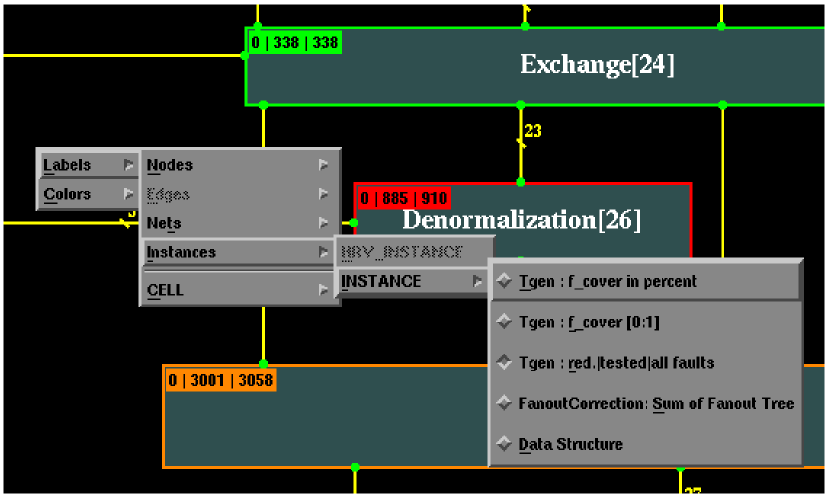

Abbildung 3.4: Popup-Menü zum Umschalten zwischen verschiedenen Informationen

separat am Ende des Abschnitts behandelt.

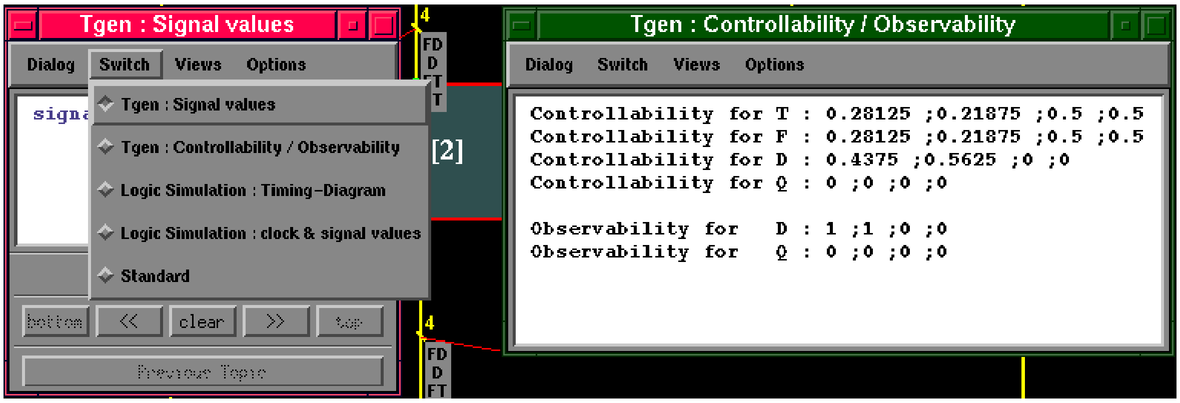

Abbildung 3.5: Beispiel für Visualisierungsdialoge

Die Dialoge aller anderen Klassen besitzen eine Menüzeile mit folgender Funktionalität (siehe Abbildung 3.5) :

- Umschalten zwischen verschiedenen Anzeigeformen (z.B. zwischen einem Diagramm und einer textuellen Darstellung)
- Ein- und Ausblenden einer Funktionsleiste zum
  - Vor- und Zurückblättern in der Folge von gespeicherten Daten (<<, >>)
  - die ältesten bzw. die aktuellste Information einblenden (bottom, top)
  - Löschen der gespeicherten Daten (clear)
  - schrittweisen "Zurückhangeln" nach verfolgten Hypertextverweisen (Previous Topic)

- Erstellen von PostScript-Dateien der aktuell angezeigten Information
- Scalierung von grafischen Darstellungen
- Schließen des Dialogs

Nach dem Öffnen eines Objektdialoges wird die erste Darstellungsform angezeigt, bei der schon gespeicherte Information vorliegt. Anschließend kann man zwischen den vorhandenen Darstellungen umschalten. Zur Identifikation werden die Namen der Anzeigeformen benutzt.

Gummiband

Nach dem Öffnen mehrerer Dialogfenster verliert man schnell den Überblick, welcher Dialog zu welchem Objekt gehört. Als Abhilfe wurde dazu zwischen den Dialogen und den zugehörigen Objekten jeweils ein "**Gummiband**" gespannt (siehe auch Abbildung 3.5). Dieses erscheint nach Öffnen des Dialogs und reagiert selbständig auf Bewegen, Vergrößern, Verkleinern und Schließen des Fensters, womit der Bezug zum zugehörigen Objekt stets erhalten bleibt.

Temporäre Dialoge

Die Visualisierung von Informationen ohne Speicherung ist nur bei dem temporären Objekt möglich. Beim Eintreffen von temporären Entwurfsdaten werden diese direkt angezeigt. Der Benutzer muß also nicht explizit einen Dialog öffnen, sondern es erscheint ein Fenster mit den Informationen. Nach Kenntnisnahme kann man dieses mit Hilfe des integrierten "Dismiss"-Pushbuttons wieder schließen.

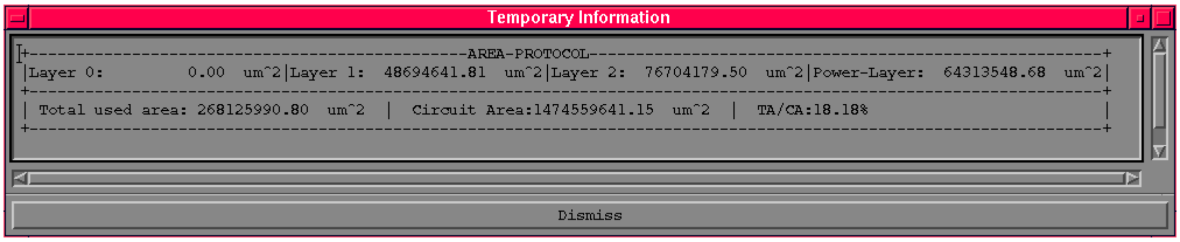

Abbildung 3.6: Statistische Angaben nach Generierung des Layouts

Die Anzeige von Informationen über das temporäre Objekt ist sehr gut für Warn- und Hilfsmeldungen an den Benutzer geeignet. Insbesondere nutzt man dabei die Eigenschaft aus, daß die Daten sofort auf dem Bildschirm erscheinen. Als Beispiel soll hierfür die Anzeige von statistischen Angaben nach der Generierung des Layouts dienen (siehe Abbildung 3.6).

Kapitel 4

Werkzeugintegration nach dem Client-Server-Modell

Neben der direkten Integration in die CADIC-Oberfläche können auch Entwurfswerkzeuge, die als eigenständige Programme realisiert sind, ihre Informationen an der Oberfläche von CADIC visualisieren. Grundsätzlich ist dabei eine Kommunikation zwischen dem angeschlossenen Programm und der Oberfläche nötig. Dabei ist zwischen Entwurfswerkzeugen deren Quellcode zur Verfügung steht, und solchen, die nur als ausführbares Programm vorliegen, zu unterscheiden.

Bei Entwurfswerkzeugen, die nur als ausführbares Programm vorliegen, beschränkt sich die Integration auf das Starten, das Versorgen mit entsprechenden Parametern und dem Einlesen der Resultate nach Abschluß der Berechung. Eine direkte Kommunikation über ein Netzwerk findet also nicht statt. Meist werden die Ergebnisse von Dateien eingelesen, falls das Werkzeug Ausgabedateien erzeugt hat bzw. dessen Ausgabe dahin umgeleitet werden kann. Beim Einlesen der berechneten Daten sind im allgemeinen Transformationen nötig, die die Resultate zur CADIC-Datenstruktur in Beziehung setzen und über die Schnittstellenfunktionen zur Visualisierung senden.
Eine kontrollierte Steuerung des Werkzeugs ist nicht möglich, da nicht in den Programmablauf eingegriffen werden kann. Einzig kann während der Berechnung hart abgebrochen werden. Hier ist jedoch mit dem Verlust der bis dahin erzeugten Daten zu rechnen. Problematisch ist bei rechenzeitintensiven Werkzeugen auch die Kontrolle über den aktuellen Stand der Bearbeitung.
Unter den genannten Voraussetzungen ist nur eine Anbindung an die graphische Benutzeroberfläche, aber keine echte Integration solcher Entwurfswerkzeuge möglich.

Steht bei einem separatem Programm dessen Quellcode zur Verfügung, wäre es sehr elegant, wie bei einem an die Oberfläche angebunden Werkzeug, die Informationen durch Schnittstellenfunktionen aus dem externen Programm abzuschicken. Weiterhin soll eine komfortable Steuerung des Programmes möglich sein, um die Visualisierung auch bei bei interaktiven Entwurfswerzeugen zu ermöglichen. Dazu benötigt man eine beidseitige Verbindung zwischen den beteiligten Prozessen. Im folgenden Abschnitt wird das der Kommunikation zugrundeliegende Modell vorgestellt.

4.1 Client-Server-Modell

Allgemein kann man eine verteilte Anwendung nach dem Client-Server-Modell durch die folgenden Charakteristika beschreiben :

- es existiert i.a. eine sogenannte Server-Anwendung (Server) und mehrere Client-Anwendungen (Clients)
- Clients fordern Dienste von Servern an, Server erbringen diese Dienste für die Clients
- Clients bauen Verbindungen aktiv auf, Server akzeptieren - passiv - Verbindungsaufbauwünsche der Clients

Bezogen auf die Visualisierung von Entwurfsdaten eigenständiger Programme, stellen die externen Werkzeuge die Clients und die graphische Benutzeroberfläche den Server dar. Die Arbeitsumgebung akzeptiert die Verbindungsaufbauwünsche der Entwurfswerkzeuge und kann gleichzeitig mehrere solche Werkzeuge verwalten. Nachdem sich ein Werkzeug angemeldet hat, und eine Verbindung durch die Benutzeroberfläche akzeptiert wurde, besteht eine beidseitige, band-orientierte Verbindung zwischen den Anwendungen. Diese Verbindung unterstützt den Daten- und Befehlsaustausch in beiden Richtungen über ein Netzwerk, insbesondere also auch den Transport der darzustellenden Informationen zur Visualisierung.
Das der Verbindung zugrundeliegende Protokoll erlaubt das Verschicken von Daten und Befehlen, wobei jede Information genau einmal und in der durch das Abschicken vorgegebenen Reihenfolge beim Empfänger eintrifft. Treten Fehler in der Datenübertragung auf, wird der Server und der betreffende Client darüber informiert. Je nach Typ des Fehlers können diese entsprechend reagieren (z.B. erneuter Versuch zum Verschicken oder Empfangen, Fehlermeldung verschicken, Kommunikation abbrechen usw.).
Der beschriebene Austausch von Daten und Befehlen zwischen zwei Programmen beruht auf einer bandorientierte Interprozeßkommunikation [Mic91]. Der Vorteil der implementierten Verbindung liegt insbesondere in der Systemunabhängigkeit, so daß die beteiligten Programme auf verschiedenen Betriebssystemen und verschiedenen Rechnertypen ablaufen können.
Der Leser kann den Aufbau der Verbindung anhand der Abbildung 4.1 nachvollziehen. Dargestellt sind die beteiligten Prozesse (ein Server und zwei Clients) und die zugehörigen Deskriptor-Tabellen.
Der Server-Prozeß akzeptiert Anfragen von Clients über einen Deskriptor s. Dieser besitzt eine für den betreffenden Recher eindeutige Nummer zur Addressierung (port_nr). Der Client Nr. 1 hat sich bereits zur Kommunikation angemeldet, die Verbindung wurde akzeptiert (durch s) und aufgebaut (durch Erzeugung von c_1). Client 2 meldet sich gerade an. Seine Verbindung wurde noch nicht akzeptiert, da noch kein weiterer Deskriptor vom Server erzeugt wurde. Die beteiligten Deskriptoren werden in diesem Zusammenhang als "sockets" bezeichnet.

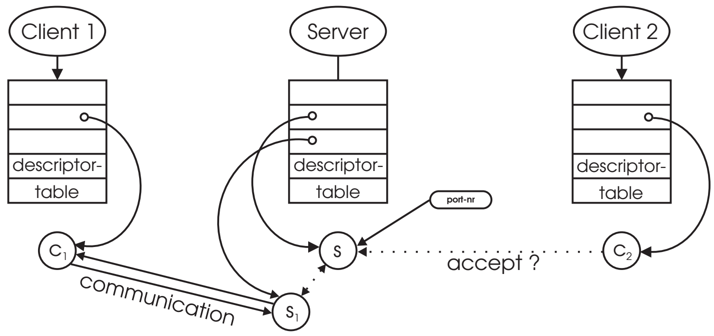

Abbildung 4.1: interne Realisierung einer Client-Server-Anwendung

4.2 Anbindung externer Entwurfswerkzeuge

Die Visualisierung von Entwurfsdaten externer Werkzeuge unterscheidet sich nicht von der bisherigen Vorgehensweise bei integrierten Werkzeugen. Zum Übertragen der Informationen werden ebenfalls die in Abschnitt 3.2.2 vorgestellten Schnittstellenfunktionen verwendet. Die Konfiguration der Anzeige erfolgt wieder durch eine Beschreibungsdatei.
Zusätzlich zu den bisher erzeugten Funktionsbibliotheken, wird in diesem Fall eine weitere Bibliothek generiert und an das externen Werkzeug gebunden. Die darin enthaltenen Schnittstellenfunktionen sind so konfiguriert, daß sie die Entwurfsdaten über ein Netzwerk an die graphische Benutzeroberfläche abschicken. Die Abbildung 4.2 zeigt die schematische Vorgehensweise bei der Integration externer Entwurfswerkzeuge. Neben der Übertragung der Entwurfsdaten zur Anzeige können beidseitig weitere Daten und insbesondere Befehle zwischen den beteiligten Prozessen ausgetauscht werden (siehe Abschnitt 4.3).

4.2.1 Notwendige Vorarbeiten auf Seiten der graphischen Arbeitsumgebung

Die graphische Arbeitsumgebung erfüllt nach dem Start die Anforderungen an einen Server, nämlich passiv auf Anmeldungen von Clients (externe Werkzeuge) zu reagieren. Die Aktivierung einer Verbindung geht wie schon beschrieben von den externen Werkzeugen aus. Bevor diese eine Verbindung aufnehmen können, müssen sie zunächst von Hand bzw. mit der folgenden Funktion :

```
int CV_ipc.start_external(char* hostname, char* username,
                          char* commands, char* path,
                          char* program, ... )
```

gestartet werden. Die beiden ersten Parameter spezifizieren den Zielrechner und einen Benutzer auf diesem Recher. Mit dem dritten Parameter können vor dem eigentlichem Aufruf des externen Programms eventuell notwendige andere Programme (Shell-Scripte) gestartet werden (mit Semicolon getrennt). Falls dies nicht erforderlich ist,

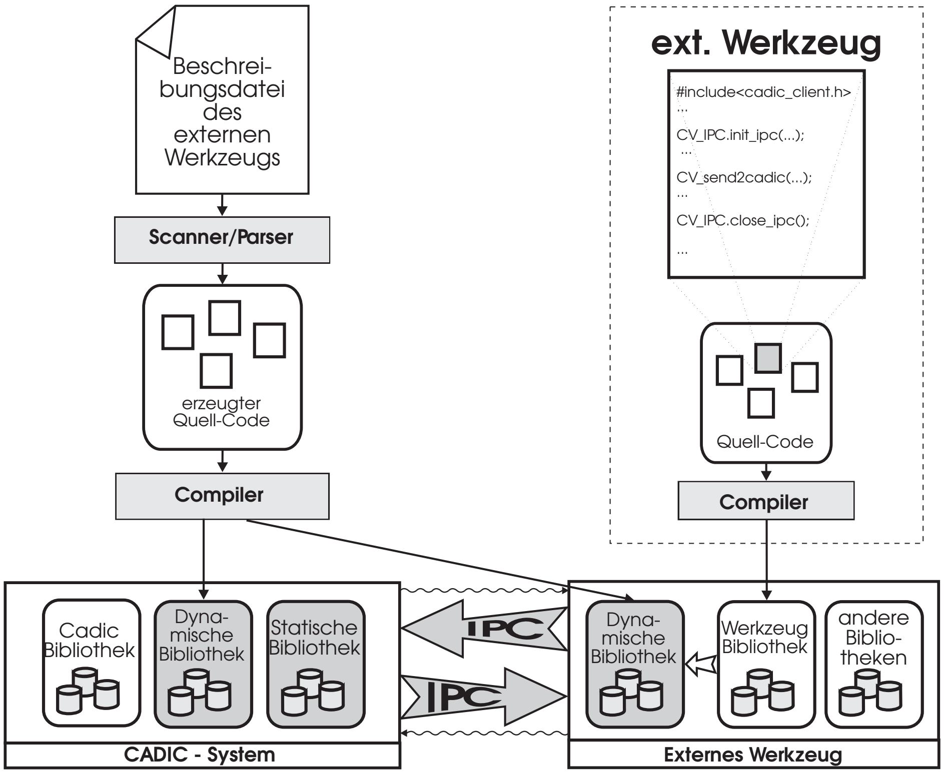

Abbildung 4.2: schematische Vorgehensweise zur Integration externer Werkzeuge

kann `NULL` übergeben werden. Durch den Parameter `path` wird zuerst in das entsprechende Verzeichnis gewechselt, bevor das externe Programm mit Namen `program` aufgerufen wird. Der Pfad kann ebenfalls auf `NULL` gesetzt werden. Nach den festen Parametern besteht die Möglichkeit, im Anschluß eine Liste von Kommandozeilenparametern, beendet mit `NULL`, an das externe Programm zu übermitteln. Als Beispiel wäre folgender Aufruf denkbar :

```
int key = CV_IPC->start_external( "minisun.cs.uni-sb.de",
                                  "testen",
                                  "cd ~/testcirc/",
                                  "~/Tgen/objekt/",
                                  "tgen",
                                  "-b 10000 -IPC MantAdd[23]",
                                  NULL );
```

Der Rückgabewert der Funktion dient im folgenden als Schlüssel, um zwischen verschiedenen angeschlossenen Programmen zu unterscheiden (zum Beispiel beim Verschicken von Befehlen, siehe Abschnitt 4.3).
Die Funktion zum Starten externer Programme sowie alle anderen Funktionen zur Interprozeßkommunikation sind in einer Klasse `CV_ipc` zusammengefaßt. Das Starten der externen Anwendungen aus der Arbeitsumgebung heraus hat den Vorteil, daß

die gesamte Bedienung der Werkzeuge innerhalb der graphischen Oberfläche erfolgen kann. Insbesondere wird so kontrolliert, ob alle gestarteten Prozesse beim Beenden von `CADIC` ebenfalls beendet wurden. Zum Testen der Anbindung eines externen Entwurfswerkzeugs kann auch aus Gründen der Einfachheit und Schnelligkeit das Starten von Hand hilfreich sein.

Nachdem vom externen Programm die Verbindung angemeldet und von der Oberfläche akzeptiert wurde, werden die übertragenen Entwurfsdaten an das Modul Visualisierung übergeben. Zum Schließen einer bestehenden Verbindung von Seiten der Oberfläche aus, benutzt man die Funktion :

```
int CV_ipc.close_connection( int key );
```

Im Falle eines interaktiven externen Programm stehen weitere Funktionen der Interprozeßkommunikation zur Verfügung (siehe Abschnitt 4.3).

4.2.2 Notwendige Vorarbeiten auf Seiten des externen Entwurfswerkzeugs

Auf Seiten des externen Entwurfswerkzeugs muß vor dem Aufruf der Schnittstellenfunktionen zunächst die Kommunikation zur graphischen Benutzeroberfläche aufgebaut werden. Dazu verwendet man wieder die Funktionen der Klasse `CV_ipc`. Diese ist im dynamisch generierten Modul der Visualisierung enthalten (vergleiche Abbildung 4.1). Zum Aufbau der Verbindung verwendet man den folgenden Aufruf :

```
int CV_ipc.init_ipc( int argc, char** argv );
```

Die Funktion zum Starten der externen Programme hat die zum Aufbau der Verbindung notwendigen Daten in den Kommandozeilenparametern abgelegt. Falls das Werkzeug per Hand gestartet wurde, müssen diese Daten vom Werkzeugentwickler eingetragen werden :

```
-ipc_host_name <server_host_name> -ipc_port_nr <port_nr>
```

Die aktuelle `port_nr` kann in der graphischen Arbeitsumgebung abgefragt werden. Im Anschluß an diese Initialisierung kann der eigentliche Datenaustausch erfolgen. Dazu benutzt man die schon in Abschnitt 3.2.2 vorgestellten Funktionen zum Versenden der Informationen. Zum Schließen der Verbindung benutzt man :

```
int CV_ipc.close_ipc();
```

4.3 Interaktive externe Programme

In den vorangegangenen Abschnitten wurde das Verschicken der zu visualisierenden Informationen von den externen Entwurfswerkzeugen zur graphischen Benutzeroberfläche beschrieben. Neben diesem Hauptdatenstrom besteht zusätzlich die Möglichkeit, Befehle und andere, zur gegenseitigen Steuerung notwendige Daten zwischen der graphischen Oberfläche und dem angeschlossenen Werkzeug auszutauschen. Dadurch können interaktive externe Entwurfswerkzeuge angesteuert werden. Im folgenden wird die dabei auftretende typische Situation geschildert :

> Nach dem Aufbau der Verbindung wartet das angeschlossene Werkzeug auf Befehle der Arbeitsumgebung von `CADIC`. Löst deren Benutzer einen solchen Befehl aus, reagiert das Entwurfswerkzeug mit entsprechender Bearbeitung des Befehls. Dabei können Informationen zur Anzeige gebracht werden oder auch Parameter angefordert werden. Nach Beendigung der Befehlsausführung wird ein Meldung an die graphische Oberfläche gesendet, daß die Bearbeitung abgeschlossen wurde oder weitere Angaben erforderlich sind. Der Benutzer konnte dort entweder weiterarbeiten, oder wurde durch die Anzeige einer Eieruhr (busy cursor) zwischenzeitlich blockiert.

Zur Durchführung des geschilderten Ablaufs der Kommunikation sind Funktionen zum Versenden und Empfangen von Befehlen und Daten bereitzustellen.
Unter Datenaustausch in diesem Sinne versteht man die Übermittlung von zusätzlichen Informationen zu Befehlen, da ein Befehl nur der Größe von einem Byte entspricht. Zum Abschicken von Befehlen verwendet man :

```
int CV_ipc.send_cmd( char cmd );           /* Ext. Werkzeug */
int CV_ipc.send_cmd( int key, char cmd ); /* CADIC */
```

Innerhalb von `CADIC` muß durch zusätzliche Angabe des Schlüssels die gewünschte Verbindung selektiert werden.
Der Empfang von Kommandos wird durch das Eintragen von benutzerdefinierten Funktionen in eine Liste ermöglicht. Die Funktionen dieser Liste werden beim Eintreffen eines Befehls nacheinander ausgeführt. Zum Eintragen in eine solche Liste lauten die Funktionen :

```
int CV_ipc.add_cmd_function( CV_ipc_fct fct, void* client_data );
int CV_ipc.add_cmd_function( int key, CV_ipc_fct fct, void* client_data );
```

Dabei ist `CV_ipc_fct` eine Typdeklaration für Funktionen mit Rückgabewert `void`, die nach dem obligatorischen Einbinden der Header-Dateien definiert ist. Die benutzerdefinierten Funktionen müssen folgendermaßen deklariert werden :

void userfunction(char cmd, void* client_data);

Durch den Zeiger `client_data` kann man beim Eintragen in die Liste jeder dieser Funktionen Zeiger auf eigene Daten übergeben. Sobald ein Befehl auf Seiten des externen Werkzeugs bzw. auf Seiten der graphischen Oberfläche eintrifft, werden die Funktionen in der Liste ausgeführt und erhalten als Parameter den entsprechenden Befehl.

Das Übertragen von Daten wird durch die überladenen Operatoren << und >> der Funktionsklasse `CV_ipc` ermöglicht. Dabei können alle Datentypen der Menge T ohne Konvertierung verschickt werden, insbesondere auch verkettete Strukturen. Innerhalb von `CADIC` wird das Entwurfswerkzeug, zu dem Daten geschickt bzw. von welchem Daten gelesen werden sollen, mit den Manipulatoren

```
CV_IPC << write_to( key ) und
CV_IPC >> read_from( key )
```

selektiert.

Die Kombination von Befehls- und Datenaustausch soll anhand des folgenden Beispiels erläutert werden.

Beispiel 1: *Sei A ein externes Entwurfswerkzeug, das auf die Befehle u und w reagiert. Zur Ausführung von w wird zusätzlich ein Parameter x vom Typ* `int` *benötigt. Das externe Werkzeug A schickt nach Bearbeitung der Befehle die Anwort r.*
Die graphische Oberfläche von `CADIC` *verschickt die Befehle u und w folgendermaßen :*

Versenden von Befehl u :

```
CV_IPC.send_cmd( key_A, 'u');
```

Versenden von Befehl w :

```
CV_IPC.send_cmd( key_A, 'w');
CV_IPC << write_to(key_A) << x;
```

Das externe Werkzeug A reagiert mit der folgenden Funktion auf die Befehle :

```
void action( char cmd, void* client_data ) {
   int x;
   switch( cmd ) {
     case 'u' :
       /* Reaktion auf den Befehl 'u' */
       CV_IPC.send_cmd('r');
       break;
     case 'w' :
       CV_IPC >> x;
       /* Reaktion auf den Befehl 'w' mit Parameter x */
       CV_IPC.send_cmd('r');
       break;
     default :
       /* Befehl unbekannt */
       break;
   }
}
```

□

Es besteht die Möglichkeit, den Benutzer der graphischen Benutzeroberfläche während der Ausführung eines Befehls im Entwurfswerkzeug zu blockieren. Der Cursor verändert sich dabei in eine Eieruhr (busy cursor). Die notwendigen Funktionen wurden schon in Abschnitt 3.4.1 vorgestellt.

Kapitel 5

Transformation und Interpretation gespeicherter Daten

Neben der Anzeige der erhaltenen Informationen, besteht auch die Möglichkeit, die gespeicherten Daten zunächst weiterzuverarbeiten und anschließend zu visualisieren bzw. die Daten durch geeignete Interpretation graphisch aufzubereiten. Dieses Kapitel widmet sich dieser Transformation und Interpretation gespeicherter Informationen.
Die Weiterverarbeitung der Daten kann man in drei Kategorien einteilen :

- die Berechung statistischer Informationen,
- den Aufruf benutzerdefinierter Funktionen und
- die farbliche Interpretation numerischer Werte.

Jede der aufgeführten Kategorien wird in den nachfolgenden Abschnitten einzeln beschrieben und anhand eines durchgehenden Beispiels wird erklärt, wie ein Werkzeugentwickler diese Möglichkeiten für seine Visualisierung nutzen kann.

5.1 Berechung statistischer Informationen

Unter der Berechung statistischer Informationen über die Hierarchie einer Spezifikation versteht man das rekursive "Aufsummieren" von Entwurfsdaten innerhalb einer Zelle bzw. einer Instanz. Im Ergebnis erhält jede Zelle bzw. Instanz einen weiteren Informationspuffer, der die Summe der aufzusammelnden Informationen dieser Hierarchieebene plus die Summen der Informationen verwendeter Zellen bzw. Instanzen enthält.
Diese Vorgehensweise soll durch die beiden Zahlenbeispiele in Abbildung 5.1 verdeutlicht werden.
An jeder Zelle bzw. Instanz ist rechts unten die aufzusummierende Information und an den Pfeilen die berechnete Summe (fett) dargestellt. Zum Beispiel ergibt sich die Summe bezüglich der Instanz HA mit dem Selektor ",0,1" zu $88 = 14 + 51 + 23$, die Summe zur Zelle FA zu $140 = 9 + 64 + 64$.

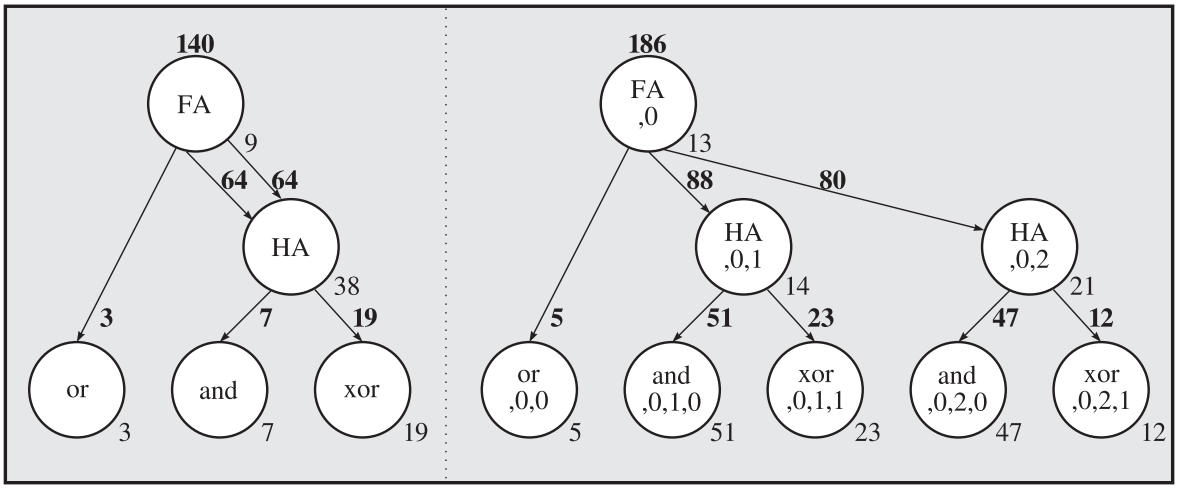

Abbildung 5.1: "Aufsummieren" von Zell- bzw. Instanzinformation

Eine naheliegende Anwendung der Summenbildung ist die Erstellung von Statistiken. Betrachtet man zum Beispiel die Anzahl der berechneten Schichtwechsel innerhalb einer Instanz nach Anwendung eines Schichtzuweisungsalgorithmus, kann man nun schon auf der obersten Hierarchieebene die Verteilung der Schichtwechsel in den einzelnen Teilmodulen erkennen. Weiterhin kann man sich auch während der Navigation durch die Hierarchie diese Verteilung anzeigen lassen, da jede Instanz die Summe der Schichtwechsel im darunterliegenden Baum enthält.

Die Transformation "Aufsummieren" kann auch für Entwurfswerkzeuge, die auf der flachen Schaltkreisdarstellung arbeiten, sehr nützlich sein. Diese Werkzeuge besitzen im allgemeinen keinen Bezug mehr zur ursprünglichen hierarchischen Gliederung des Schaltkreises. Heftet man jedoch an die Basiszellen bzw. deren Instanzen Informationen und läßt diese anschließend durch die Visualisierung aufsummieren, kann der Bezug zur hierarchischen Darstellung wieder hergestellt werden.

Als Beispiel dafür soll nun auf die Visualisierung der berechneten Fehlerüberdeckung beim Ablauf eines Testmustergenerators eingegangen werden. Dazu wurden bei jeder Instanz einer Basiszelle Informationspuffer für die Anzahl der bereits getesteten, der bereits als redundant erkannten und aller eingepflanzten Fehler eingerichtet. Nachfolgend ist der entsprechende Ausschnitt der Beschreibungsdatei aufgeführt :

```
MODUL = "Test_Pattern"
...
INSTANCE
  {
  ...
  HISTORY = 1000
    {
    int tested;
    int red;
```

```
    }
  HISTORY = 1
    {
    int faults;
    long sum_of_tested    = SUM ( INSTANCE, tested );
    long sum_of_red       = SUM ( INSTANCE, red );
    long sum_of_fauts     = SUM ( INSTANCE, faults );
    }
  ...
  } ...
```

Zum "Aufsummieren" verwendet man also das folgende Konstrukt :

```
num_typ sum = SUM ( okl, arg );

wobei  num_typ ∈ { int, long, float, double }
       sum = Name des neu gebildeten Informationspuffers
       okl ∈ { OKF ∪ OKH } \ SIGNAL
       arg ∈ { Menge der Namen der Informationspuffer
               von okl }
```

Dabei ist zu beachten, daß der Informationspuffer *arg* bei der Objektklasse okl bereits deklariert wurde und daß bei abweichenden Datentypen zwischen Argument und Summe entsprechend den üblichen Regeln konvertiert wird. Das Konstrukt `SUM` ist natürlich nur innerhalb der Objektklassen sinnvoll, die einer Box entsprechen. Das trifft für die Klassen `INSTANCE` und `CELL` zu. Informationspuffer der Objektklasse `SIGNAL` können als Argument für `SUM` nicht verwendet werden, da sich die Elemente dieser Klasse i.a. über mehrere Zellen bzw. Instanzen erstrecken, was eine lokale Summenbildung ausschließt. Die auf diese Weise neugebildeten Informationspuffer kann man wie üblich weiterverwenden.
Auf der obersten Hierarchieebene bekommt man jetzt schon einen Überblick über die Anzahl der getesteten, redundanten und die Zahl aller Fehler pro Modul.
Eine prozentuale Anzeige der Fehlerüberdeckung ist jedoch nicht möglich. Die Realisierung dessen wird erst durch die Kombination von berechneten statistischen Informationen und dem Aufruf anwenderspezifischer Funktionen möglich.

5.2 Aufruf benutzerdefinierter Funktionen

Eine weitere Möglichkeit zur Transformation gespeicherter Daten ist der **Aufruf benutzerdefinierter Funktionen**. Hiermit wird der Werkzeugentwickler in die Lage versetzt, Informationen, die erst durch die Visualisierung erzeugt wurden bzw. Entwurfsdaten die von anderen Entwurfswerkzeugen stammen, weiterzuverarbeiten. Führt man das Beispiel aus Abschnitt 5.1 weiter, kann durch eine benutzerdefinierte Funktion die Fehlerüberdeckung pro Modul berechnet werden. In diesem Fall berechnet die

Funktion einfach das Verhältnis von getesteten und redundanten Fehler zur Gesamtzahl der Fehler. Man beachte, daß die Argumente zur Berechnung dieses Verhältnisses erst durch das "Aufsummieren" der Visualisierung verfügbar geworden sind. Ein Entwurfswerkzeug basierend auf der flachen Schaltkreisdarstellung hat selbst keinen Zugang zu diesen Daten.
In der Beschreibungssprache werden benutzerdefinierte Funktionen, wie in der Programmiersprache C üblich, deklariert :

```
MODUL = "Test_Pattern"
...
INSTANCE
  {
  ...
  HISTORY = 1
    {
    float f_cover = calc_f_cover(sum_of_tested, sum_of_red, sum_of_faults );
    }
  ...
  }
...
```

Sobald alle angegebenen Argumente vorliegen, bzw. sobald sich anschließend ein Argument ändert, wird die deklarierte Funktion aufgerufen. Diese muß vom Werkzeugentwickler entsprechend den verwendeten Datentypen definiert werden. Im Beispiel würde die Funktion also lauten :

```
#include<Cadic/cv_userf.h>

float calc_f_cover( long t, long r, long f )
        {
        float ft=t, fr=r, ff=f;
        return (ft+fr)/ff;
        }
```

Bei der Definition und Deklaration dürfen alle Datentypen aus T verwendet werden, Strukturen allerdings nur dann, falls diese in der Beschreibungsdatei deklariert wurden. Die Datei `Cadic/cv_userf.h` enthält alle notwendigen Deklarationen und wird während der Konfiguration erzeugt.

5.3 Farbliche Interpretation numerischer Werte

Eine sehr ausdrucksstarke Methode zur Visualisierung ist die Einfärbung von Objekten durch Interpretation numerischer Werte. Dazu werden die Objekte bzw. die Vorder- bzw. Hintergrundfarbe von Labels und Dialogen nicht direkt durch das Versenden

von Farbnamen eingefärbt, sondern es wird eine Farbe aufgrund anderer numerischer Entwurfsdaten berechnet. Dazu definiert man Intervalle in der Ebene, die sich nicht überlappen dürfen. Jedem Intervall wird eine Farbe zugeordnet. Trifft ein zu interpretierender Wert ein, wird getestet, in welchem Intervall dieser Wert liegt. Das Objekt wird dann mit der diesem Intervall zugeordneten Farbe gezeichnet. Bereiche in der Ebene, die nicht durch Intervalle überdeckt werden, korrespondieren zu einer einstellbaren Standardfarbe.
Hauptanwendung ist die übersichtliche Darstellung von Bereichen, in denen ein Entwurfswerkzeug "gute" bzw. "schlechte" Resultate geliefert hat. Als Beispiel soll die berechnete Fehlerüberdeckung aus Abschnitt 5.2 zur Einfärbung der Instanzen benutzt werden :

```
MODUL = "Test_Pattern"
...
INSTANCE
  {
  ...
  HISTORY = 1
    {
    Color col = SCALE ( f_cover, "red")
        {
        [ 0.00  , 0.98  ) "red"    /* schlechte Fehlerüberdeckung */
        [ 0.98  , 0.99  ) "orange" /* mittlere Fehlerüberdeckung */
        [ 0.99  , 0.995 ) "yellow" /* gute Fehlerüberdeckung */
        [ 0.995 , 1.00  ] "green"  /* sehr gute Fehlerüberdeckung */
        }
    }
  ...
  }
...
```

Das verwendete Konstrukt `SCALE` erhält als Parameter einen Informationspuffer numerischen Typs, einen Standardfarbwert und eine Menge von Paaren, die jeweils aus einem Intervall und der zugeordneten Farbe bestehen. Fällt der Wert des angegebenen Informationspuffers in einen der Bereiche, wird die entsprechende Farbe eingestellt, anderenfalls der Standardfarbwert. Die Intervalle können abgeschlossen, rechts bzw. links offen oder offen sein und werden auf Überlappung und inkonsistente Intervallgrenzen überprüft. Der resultierende Farbwert kann wie im Beispiel einem weiteren Informationspuffer zugeordnet werden, oder er wird zur farblichen Gestaltung von Vorder- oder Hintergrundfarbe benutzt. Dazu wird das gleiche Konstrukt verwendet :

```
     FOREGROUND = SCALE ( ... )  ...
     BACKGROUND = SCALE ( ... )  ...
```

Mit Hilfe dieser farblichen Interpretation sind Bereiche mit guter und schlechter Fehlerüberdeckung leicht zu lokalisieren.

Kapitel 6

Gesamtkonfigurationen

In den vorangegangenen Kapiteln wurde die Konfiguration der Visualisierung stets lokal für jedes Entwurfswerkzeug beschrieben. Stellt man eine Oberfläche von `CADIC` zusammen, die mehrere Entwurfswerkzeuge kombiniert, muß eine Konfiguration erzeugt werden, welche allen integrierten Werkzeugen gerecht wird.
Eine naheliegende Möglichkeit dazu ist, eine neue große Beschreibungsdatei aus den lokalen Beschreibungen zu bilden. Aus dieser Datei werden analog zur Vorgehensweise bei einem einzelnen Werkzeug die entsprechenden Bibliotheken und Headerdateien erzeugt. Voraussetzung ist die syntaktische Korrektheit und paarweise verschiedene Modulnamen der zu mischenden Einzelbeschreibungen.

6.1 Auftretende Probleme

Erzeugt man die kombinierte Beschreibung durch einfaches Aneinanderhängen der einzelnen Dateien, entspricht diese nicht mehr der Syntax der Sprache (die Modulnamen der einzelnen Dateien werden nicht in die neue Beschreibung übernommen). Dies liegt an der Gliederung jeder Beschreibungsdatei in

1. den Abschnitt mit den Strukturdeklarationen,
2. den Abschnitt mit den Deklarationen der Informationspuffer bei den Objektklassen und
3. den Abschnitt mit den Anzeigedeklarationen bei den Objektklassen.

Zur Bildung einer korrekt gegliederten Konfiguration muß nacheinander jeder Hauptabschnitt aus allen zu kombinierenden Dateien extrahiert werden. Dazu wurde für jeden Abschnitt ein Parser implementiert. Jeder der Parser erhält als Eingabe die zu mischenden Beschreibungen und schneidet den entsprechenden Hauptabschnitt aus. Die beiden Parser, die zum zweiten und dritten Abschnitt korrespondieren, müssen zusätzlich gewährleisten, daß jede Objektklasse mit genau einem Deklarationsblock pro Abschnitt vertreten ist. Deklarieren zum Beispiel zwei Beschreibungen jeweils Informationspuffer zur Klasse `CELL`, dann sind alle Informationspuffer dieser Klasse zu sammeln und in einem Block der kombinierten Beschreibung zusammenzufassen.

Nach obiger Vorgehensweise erhält man eine korrekt gegliederte neue Beschreibungsdatei ohne Modulnamen. Diese Mischbeschreibung kann jedoch weitere Syntaxfehler enthalten.

Betrachtet man zunächst den ersten Abschnitt mit den Strukturdeklarationen, können durch die Kombination mehrerer Beschreibungsdateien Überschneidungen bei den Strukturnamen auftreten, wie folgendes Beispiel zeigt :

Beispiel 1: *Überschneidungen bei Strukturnamen*

```
MODUL = Werkzeug_A
  ...
  struct list
    {
    char  item;
    struct list * next;
    }
  ...
```

```
MODUL = Werkzeug_B
  ...
  struct list
    {
    int  item;
    struct list * prev;
    struct list * next;
    }
  ...
```

□

Diese Überschneidungen versucht man durch Voranstellen der jeweiligen Modulkennung mit nachfolgendem Unterstrich vor die Strukturnamen aufzulösen. Im Beispiel würden sich die folgenden Namen in der kombinierten Beschreibung ergeben :

`struct Werkzeug_A_list { ... }` und
`struct Werkzeug_B_list { ... }` .

Dadurch sind die Namen im allgemeinen eindeutig bezüglich der neuen Beschreibung. Eine Ausnahme bildet der folgende Fall :

Seien u,w zwei Modulnamen mit $u = x \cdot \text{'\_'} \cdot y$ *und* $w = x$ *(nicht präfixfrei).*
Seien s_1, s_2 *zwei gültige Strukturnamen aus den Modulen u und w.*

Für $s_2 = y \cdot \text{'\_'} \cdot s_1$ *gilt :* $\quad u \cdot \text{'\_'} \cdot s_1 = w \cdot \text{'\_'} \cdot s_2.$

(*„·" = Konkatenation von Zeichenketten*)

Im Falle nicht präfixfreier Modulnamen können also Überschneidungen bei den Strukurbezeichnern auftreten, falls diese die obige Eigenschaft besitzen. Um die Eindeutigkeit der Strukturnamen weiterhin zu gewährleisten, kann man Präfixfreiheit der Modulnamen fordern. In Hinsicht auf die Unabhängkeit zwischen den Entwurfswerkzeugen ist eine bessere Lösung, Strukturnamen mit anführendem Unterstrich zu verbieten und beim Bilden der neuen Namen alle verwendeten Unterstriche zu verdoppeln. Damit existiert genau ein Vorkommen von Unterstrichen mit ungerader Anzahl, wobei der letzte davon das Trennzeichen zwischen Modulkennung und Strukturnamen bildet. Aus dem Auffinden dieses Trennzeichens ergibt sich die Eindeutigkeit der Zerlegung. Da die Mo-

dulnamen als paarweise verschieden vorausgesetzt wurden, sind die so gebildeten neuen Bezeichner eindeutig bezüglich der Gesamtbeschreibung.
Angewandt auf das Beipiel ergeben sich folgende Strukturnamen :

```
struct Werkzeug__A_list { ... }  und
struct Werkzeug__B_list { ... } .
```

Es existiert genau ein Vorkommen von Unterstrichen mit ungerader Anzahl, nämlich der Unterstrich vor `list`.

Im zweiten Hauptabschnitt können wie bei den Strukturdeklarationen Informationspuffer in verschiedenen Modulen gleich benannt worden sein. Um eine bezüglich der neuen Beschreibungsdatei eindeutige Vergabe der Informationspuffernamen sicherzustellen, wird diesen der Modulname wie ein Klassenname den Memberfunktionen in der Programmiersprache **C++** vorangestellt. Betrachtet man folgendes Beispiel,

Beispiel 2: *Namenskonflikte bei Informationspuffern*

MODUL = Werkzeug_A	MODUL = Werkzeug_B
...	...
SIGNAL	SIGNAL
{	{
char signal_value;	int signal_value;
}	}
...	...

□

erhält man in der Gesamtbeschreibung folgende Namen der Informationspuffer :

```
char Werkzeug_A::signal_value;
int  Werkzeug_B::signal_value;
```

Da der Doppelpunkt nicht bei den einfachen Bezeichnern verwendet werden kann, ist diese Namensbildung eindeutig bezüglich der kombinierten Beschreibung.

Weitere Probleme beim Zusammenmischen mehrerer Beschreibungsdateien entstehen durch die Verwendung von benutzerdefinierten Funktionen und durch die Berechnung statistischer Informationen über die Hierarchie (siehe auch Kapitel 5). Beiden Konstrukten ist gemeinsam, daß durch sie neue Informationspuffer in Abhängigkeit von bereits deklarierten entstehen.

Beim Aufruf benutzerdefinierter Funktionen innerhalb einer Objektklasse können als Parameter Informationspuffer dieser Objektklasse bzw. der Klasse `GLOBAL` angegeben werden. Daraus folgt, daß im zweiten Abschnitt, falls vorhanden, zuerst die Informationspuffer zu `GLOBAL` deklariert werden müssen. Bei den lokalen Beschreibungen ist dies garantiert, da man Informationspuffer erst nach entsprechender Deklaration verwenden kann.

Durch die Berechnung statistischer Informationen über die Hierarchie resultiert eine weitere Anforderung an die Reihenfolge der Deklarationsblöcke.

Das Konstrukt `SUM(okl, puffer_name)` findet nur bei den Objektklassen `CELL` und

`INSTANCE` Anwendung. Als Parameter können aber Informationspuffer der folgenden Objektklassen

- bei Verwendung innerhalb von `CELL`
  - `HRY_PIN`, `HRY_PAD`, `HRY_FANOUT`, `HRY_CNODE`, `HRY_WIRE`, `HRY_CEDGE`, `HRY_NET`, `HRY_INSTANCE`, `CELL`
- bei Verwendung innerhalb von `INSTANCE`
  - `PIN`, `PAD`, `FANOUT`, `CNODE`, `WIRE`, `CEDGE`, `NET`, `INSTANCE`

angegeben werden. Folglich müssen die beiden Deklarationsblöcke von `CELL` und `INSTANCE` an das Ende des zweiten Abschnitts verschoben werden. Die Reihenfolge untereinander spielt keine Rolle, da kein Informationspuffer der beiden Klassen als Parameter bei der anderen verwendet werden kann.

6.2 Erzeugen von Gesamtkonfigurationen

Zusammenfassend verdeutlicht Abbildung 6.1 nochmals die Vorgehensweise beim Mischen mehrerer Beschreibungsdateien.

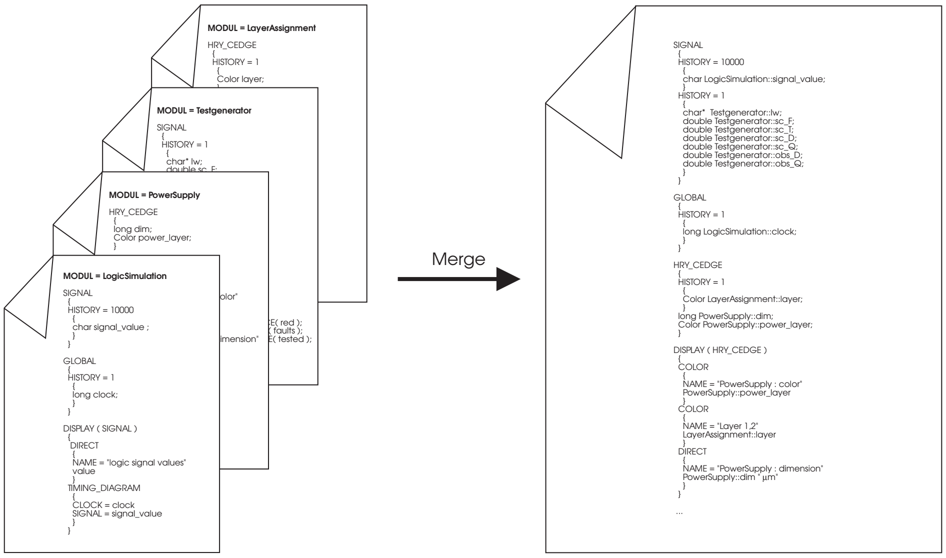

Abbildung 6.1: Mischen von Beschreibungsdateien

Zum einfachen Erstellen von Gesamtkonfigurationen steht die Funktion

CV_merge $file_1$, ..., $file_n$

zur Verfügung.
Diese Funktion schreibt die aus den Dateien $file_1$, ..., $file_n$ entstandene kombinierte Beschreibung auf die Standardausgabe. Durch Umlenken der Ausgabe in eine

Datei kann das Resultat abgespeichert werden. Vor dem Mischen wird jede der einzelnen Dateien auf syntaktische Korrektheit geprüft. Der Mischvorgang selbst löst die angesprochenen Konfliktsituationen auf und liefert eine syntaktisch korrekte Gesamtbeschreibung.

Das Erstellen von Mischbeschreibungen kann auch durch die Konfigurationsoberfläche `Cavis` erfolgen (siehe Abschnitt 3.3). Dazu stellt man zunächst die zu mischenden Dateien in einer Liste zusammen. Anschließend kann der Mischvorgang gestartet werden. Die resultierende Gesamtkonfiguration wird im Arbeitsfenster angezeigt. Diese kann jetzt gegebenenfalls weiter editiert und abgespeichert werden.

Kapitel 7

Beispiele

Dieses Kapitel soll dem Leser anhand von Beispielen einen Eindruck davon verschaffen, wie man ausgehend von den Vorstellungen über die Anzeige der Entwurfsdaten eines Werkzeuges zu einer selbst konfigurierten Visualisierung gelangt. Dabei wird insbesondere auf die Erstellung von Beschreibungsdateien und auf die notwendigen Modifikationen des Quellcodes der anzuschließenden Werkzeuge eingegangen.

7.1 Logik-Simulation

Zunächst soll die Konfiguration der Visualisierung anhand des in `CADIC` integrierten Analyse-Werkzeugs 'Logik-Simulation' besprochen werden.

Diese Logiksimulation arbeitet auf der Netzlistendarstellung der Schaltung. Jeder Zelle wird ein logisches und zeitliches Verhalten zugeordnet. Die Ausgangssignale werden entsprechend dem logischen Verhalten der Zelle berechnet und können um eine vorgegebene Zeit verzögert werden. Diese Verzögerung entspricht der Zeit, die ein Gatter benötigt, um nach Änderung der Eingangswerte seinen Ausgangswert zu berechnen. Ein Wert am Ausgang wird nach Ablauf einer bestimmten Zeit zurückgesetzt. (auf einen Standardwert oder undefiniert). Das zeitliche Verhalten wird dazu durch interne Zähler beschrieben. Die Verzögerungen der Signalnetze werden nicht betrachtet, da die exakten Leitungslängen erst nach Berechnung eines Layouts bekannt sind. In einem Zeitschritt simulieren alle Zellen ihr zugewiesenes Verhalten 'parallel'.

Für den Benutzer ist die Anzeige der logischen Signalwerte an den Signalnetzen und insbesondere die graphische Aufbereitung in Form von Timing-Diagrammen von Interesse.

Zur Konfiguration der Visualisierung ist zunächst die Beschreibung der Anzeige zu erstellen. Dazu legt man eine Datei an, um die Beschreibung zu speichern, zum Beispiel :

    log_sim.config

Zur Abgrenzung zu anderen Entwurfswerkzeugen wird zu Beginn der Beschreibung eine Modulkennung vergeben :

    MODUL = "Logic_Simulation"

Der erste Abschnitt der Beschreibungsdatei mit den Strukturdeklarationen bleibt leer, da keine Daten dieses Typs angezeigt werden sollen. Im zweiten Hauptabschnitt werden Informationspuffer bei den Objektklassen `SIGNAL` und `GLOBAL` eingerichtet. Zur Darstellung der logischen Signalwerte (0-`SIGNAL_LOW`, 1-`SIGNAL_HIGH`, X-`SIGNAL_UNDEF`) reicht ein Informationspuffer vom Typ `char` aus. Der Zähler wird durch einen Informationspuffer vom Typ `long` repräsentiert und bei der Objektklasse `GLOBAL` deklariert, da dieser Zähler für alle Signalnetze der flachen Schaltung gleichzeitig gültig ist. Der zweite Abschnitt der Beschreibungsdatei sieht also folgendermaßen aus :

```
SIGNAL
  {
  HISTORY = 10000 { char signal_value; }
  }
GLOBAL
  {
  HISTORY = 1 { long clock; }
  }
```

Die Länge des Informationspuffers `clock` wurde auf eins beschränkt. Dadurch überschreiben die inkrementierten Werte stets den älteren Eintrag. Von den Signalwerten werden dagegen möglichst alle gespeichert, um die Signalverläufe über viele Zeitschritte hinweg in Form von Timing-Diagrammen verfolgen zu können (deshalb Länge 10000).

Der nächste Schritt zur Konfiguration besteht in der Beschreibung der Anzeige. Eine übersichtliche Darstellung der logischen Signalwerte erreicht man durch direktes Anheften an die Signale :

```
DIRECT
  {
  NAME = “Logic Simulation : signal values“
  FOREGROUND = “blue“
  BACKGROUND = “white“
  signal_values
  }
```

Möchte man gleichzeitig den Takt mit anzeigen, kann die Anzeige in den Labels durch den Zähler ergänzt werden :

```
DIRECT
  {
  NAME = “Logic Simulation : clock & signal values“
  FOREGROUND = “black“
  BACKGROUND = “grey“
  clock “:“ signal_values
  }
```

Während des Ablaufs der Logiksimulation kann zwischen den zwei konfigurierten Labels umgeschaltet werden.
Zur Illustration der Signalverläufe wird als nächstes ein Timing-Diagramm aus den Informationspuffern erzeugt. Die dazu notwendige Syntax ist recht einfach :

```
  TIMING_DIAGRAM
    {
    NAME = "Timing-Diagram"
    CLOCK = clock
    SIGNAL = signal_value
    }
```

Verzichtet man wie hier auf die Spezifikation der Vorder- oder Hintergrundfarbe wird standardmäßig blau bzw. weiß eingestellt.
Zusammenfassend erhält man die folgende vollständige Beschreibungsdatei :

```
MODUL = "Logic_Simulation"

/* ----------------- 1. Hauptabschnitt -----------------

  /* leer, da keine Strukturen benötigt werden */

/* ----------------- 2. Hauptabschnitt ----------------- */
  SIGNAL
    {
    HISTORY = 10000 { char signal_value; }
    }
  GLOBAL
    {
    HISTORY = 1 { long clock; }
    }
/* ----------------- 3. Hauptabschnitt ----------------- */
  DISPLAY ( SIGNAL )
    {
    DIRECT
      {
      NAME = "Logic Simulation : signal values"
      FOREGROUND = "blue"
      BACKGROUND = "white"
      signal_values
      }
    DIRECT
      {
      NAME = "Logic Simulation : clock & signal values"
```

```
      FOREGROUND = “black“
      BACKGROUND = “grey“
      clock “:“ signal_values
      }
    TIMING_DIAGRAM
      {
      NAME = “Timing-Diagram“
      CLOCK = clock
      SIGNAL = signal_value
      }
    }
```

Aus der vorliegenden Beschreibung werden durch Anwendung der Scanner-Parser-Kombination bzw. durch die Konfigurationsoberfläche `Cavis` alle benötigten Funktionen zur Visualisierung erzeugt.
Anschließend muß der Transport der Informationen vom Modul Logiksimulation zur Anzeige ermöglicht werden. Zur Übermittlung der Signalwerte und des Zählers sind die folgenden Aufrufe von Schnittstellenfunktionen zu implementieren. Dabei ist darauf zu achten, daß jeweils alle veränderten Signalwerte verschickt werden, bevor der Zähler inkrementiert und ebenfalls abgesendet wird.

```
...
long      clock, net_nr, komp;
selector* slr;
char      sig_values;
...
/* inkrementiere und verschicke den Zähler */
clock++;
CV_set_info( NULL, 0, 0, CV_GLOBAL,
             “Logic_Simulation::clock“,
             (void*) &clock );
...
/* Für alle Instanzen von Basiszellen */
  /* Berechne die Signalwerte der Ausgänge */
    /* Falls gilt : alter Wert ungleich neuer Wert */
    /* Schicke diesen Signalwert zur Anzeige */

       CV_set_info( slr, net_nr, komp, CV_SIGNAL,
                    “Logic_Simulation::signal_value“,
                    (void*) &sig_value) );
...
```

Nach dem Einfügen der Funktionsaufrufe zum Abschicken der Daten muß das entsprechende Modul neu übersetzt werden. Zusammen mit den generierten Bibliotheken wird dieses an das `CADIC`-System angebunden. Abbildung 7.1 zeigt beispielhaft eine Konfiguration beim Ablauf der Logik-Simulation anhand eines 16-bit-Sortierers. Das korrekte

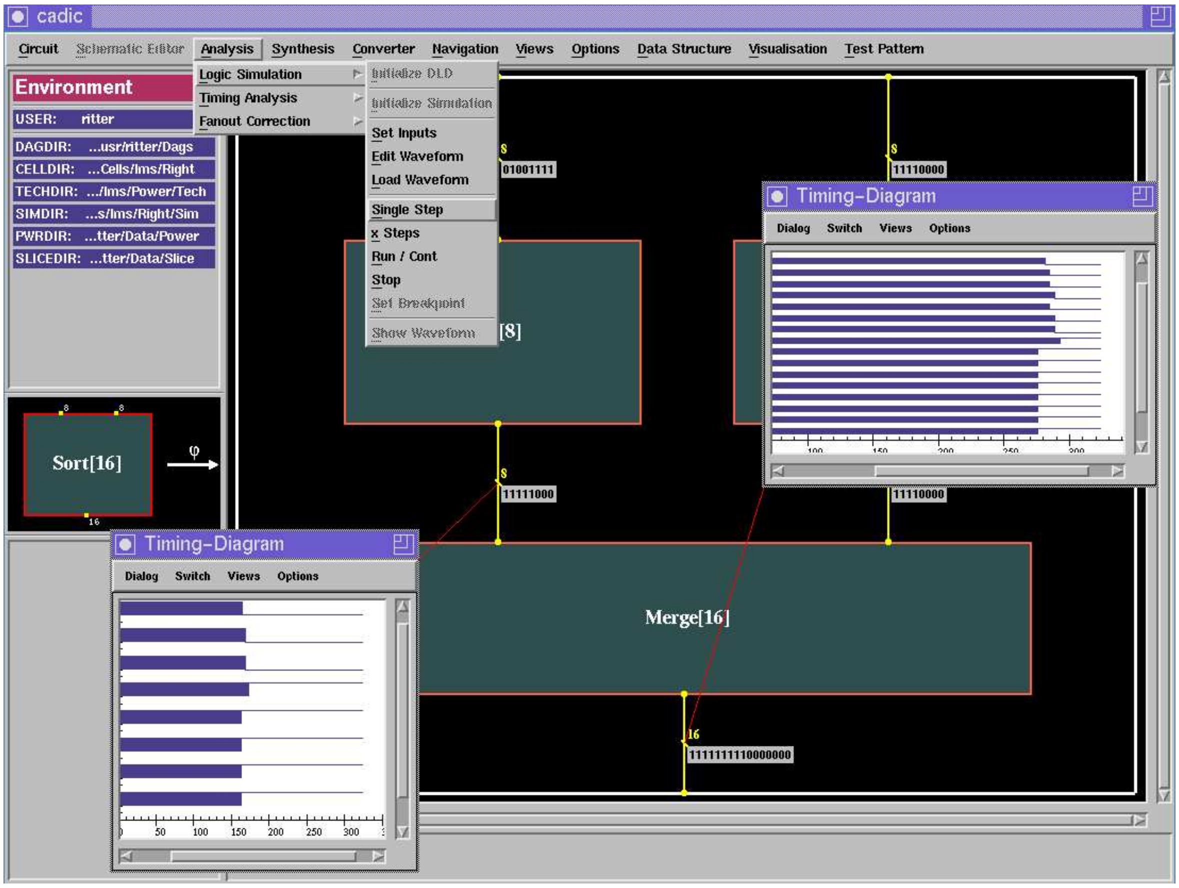

Abbildung 7.1: Visualisierung der logischen Signalwerte

Verhalten der Schaltung bei dieser Eingabe ist durch die visualisierten Signalwerte offensichtlich.

7.2 Schichtzuweisung

Im folgenden wird als weiteres Beispiel die Konfiguration der Visualisierung bei der Schichtzuweisung besprochen werden.

Der zugrundeliegende Algorithmus arbeitet auf der hierarchischen Schaltungsdarstellung und verdrahtet in 2 Schichten (Stromversorgung, falls generiert, ist in einer separaten Schicht verlegt) [Mol93], [Gra92]. Für jede Zelle wird die Schichtzuweisung genau einmal durchgeführt. Die Vorkommen der Zellen erhalten den Status normal bzw. dual, je nachdem ob die Verdrahtungskanten in dieser Instanz wie bei der ursprünglichen Schichtzuweisung der Zelle oder dual dazu verlegt wurden.

Zur Visualisierung sollen die eingefügten Schichtwechsel, die je nach zugewiesener Schicht eingefärbten Verdrahtungskanten, die Zahl der Schichtwechsel innerhalb einer Zelle (rekursiv über die darin verwendeten Zellen aufsummiert), sowie der Modus

('N'-normal / 'D'-dual) eines Zellvorkommens angezeigt werden.
In der nachfolgenden Beschreibung werden deshalb zunächst Informationspuffer bei den Verdrahtungsknoten (Objektklasse HRY_CNODE), den Zellvorkommen (Objektklasse HRY_INSTANCE) und den Verdrahtungskanten (Objektklasse HRY_WIRE) angelegt.

```
MODUL = LayerAssignment

/* ---------------- 1. Hauptabschnitt ----------------

  /* leer, da keine Strukturen benötigt werden */

/* ---------------- 2. Hauptabschnitt ---------------- */

HRY_CNODE
  {
  HISTORY = 1             /* Festlegung der Pufferlänge */
    {
    long v;               /* Informationspuffer vom Typ long */
                          /* zur Identifizierung von Schichtwechselknoten */
    Color col;            /* Informationspuffer vom Typ Farbe */
    }                     /* zur Markierung der eingefügten Schichtwechsel */
  }
HRY_INSTANCE
  {
  HISTORY = 1             /* Festlegung der Pufferlänge */
    {
    long vias = SUM ( HRY_CNODE , v );
                          /* Einrichtung eines Informationspuffers vom Typ
                             long, welcher die rekursive Summe der Schicht-
                             wechsel über die Hierarchie enthält
                             s.a. Abschnitt 5.1 */
    char modus;           /* Einrichtung eines Informationspuffers */
    }                     /* vom Typ char */
  }
HRY_WIRE
  {
  HISTORY =1              /* Festlegung der Pufferlänge */
    {
    Color layer;          /* Einrichtung eines Informationspuffers */
    }                     /* vom Typ Farbe */
  }
/* ---------------- 3. Hauptabschnitt ---------------- */

DISPLAY ( HRY_CNODE )     /* Beschreibung der Anzeige */
  {                       /* zu den Verdrahtungsknoten */
  COLOR                   /* Einfärbung der Schichtwechsel */
```

```
    {
    NAME = "LayerAssignment : Color of Vias"
    col
    }
  }
DISPLAY ( HRY_WIRE )      /* Beschreibung der Anzeige */
  {                       /* zu den Verdrahtungskanten */
  COLOR                   /* Einfärbung der Leitungen */
    {
    NAME = "LayerAssignment : Color of Edge"
    layer
    }
  }
DISPLAY( HRY_INSTANCE )   /* Beschreibung der Anzeige */
  {                       /* zu den Zellvorkommen */
  DIRECT                  /* an die Instanzen wird */
    {                     /* "Vias: 3 ( N )" geschriebden */
    NAME = "LayerAssignment : Vias / Mode"
    "Vias: " vias " ( " modus " ) "
    }
  }
```

Jeder eingefügte Schichtwechselknoten wird mit dem Wert eins und der Farbe weiß markiert. Dazu dienen die Informationspuffer `v` und `col`. Der Informationspuffer `vias` speichert die rekursive Summe der Schichtwechsel in den hierarchischen Zellvorkommen. Der Status der Zellvorkommen wird durch den Informationspuffer `modus` repräsentiert. Die Verdrahtungskanten werden über `layer` eingefärbt.
Zur Beschreibung der Anzeige genügt es, die Labels an den Instanzen und die Einfärbung der Verdrahtungskanten bzw. -knoten zu konfigurieren. Dies wurde im dritten Hauptschnitt vorgenommen.

Während der Schichtzuweisung werden beim Einfügen von Schichtwechselknoten deren Markierung und Farbe sowie für jedes hierarchische Zellvorkommen deren Modus ('N'-normal, 'D'-dual) verschickt. Dazu verwendet man die folgenden Funktionsaufrufe :

```
CV_set_hry_info ( key, node_nr, 0,
                  CV_HRY_CNODE,
                  "LayerAssignment::v", (void*) &vias);
CV_set_hry_info ( key, node_nr, 0,
                  CV_HRY_CNODE,
                  "LayerAssignment::col", (void*) "white");
CV_set_hry_info ( key, inst_nr, 0,
                  CV_HRY_INSTANCE,
                  "LayerAssignment::modus", (void*) &modus );
```

Alternativ hätte die Färbung der Schichtwechselknoten auch direkt durch die Visualisierung erfolgen können. Der Informationspuffer `col` müßte in diesem Fall wie folgt deklariert werden :

```
HRY_CNODE
  {
  HISTORY = 1
    {
    long v;
    Color col = SCALE( v , "white" )
                      { [1,1] "white" }
    }
  }
```

Die Verdrahtungskanten werden zunächst nur für die aktuelle Hierarchieebene eingefärbt :

```
  /* Für alle Verdrahtungskanten */
  CV_set_hry_info( key, edge_nr, 0,
                   CV_HRY_CEDGE,
                   "LayerAssignment::layer", (void*) col );
```

Die Farben werden dabei anhand der Instanzmodi auf dem Weg von der Wurzel der Hierarchie zur aktuellen Ebene berechnet, d.h. bei ungerader Anzahl dualer Vorkommen auf dem Weg muß die duale Färbung verwendet werden.
Nach jedem Betreten einer neuen Hierarchieebene sind die Farben neu zu berechnen und zu verschicken. Busleitungen, deren Einzelkomponenten in verschiedenen Schichten verlegt worden sind, werden grau eingefärbt.
In Verbindung mit der Baumdarstellung der Hierarchie oder dem Trace-View ergibt sich so eine eindrucksvolle Aufbereitung der Schichtzuweisungsinformationen. Ein Beispiel dafür ist das Bild auf Seite 30.

Anhang A

Die verfügbaren Objektklassen

Objektklassen			
Schaltkreisdarstellung		Bus-	Kurzbeschreibung
flach	hierarchisch	breite	
PIN		x	Ein- und Ausgänge von Instanzen
	HRY_PIN	x	
PAD		x	Anschlüsse am äußeren Rand von Makrozellen
	HRY_PAD	x	
FANOUT		x	Verdrahtungsknoten mit Fanout größer 1
	HRY_FANOUT	x	
CNODE		x	Knoten ohne spezielle Attribute
	HRY_CNODE	x	
WIRE		x	Verdrahtungskanten
	HRY_WIRE	x	
CEDGE		x	Kanten ohne spezielle Attribute
	HRY_CEDGE	x	
NET		x	lokale Signalnetze bzgl. einer Hierarchieebene
	HRY_NET	x	
SIGNAL		x	Signalnetze der flachen Schaltung
INSTANCE			Vorkommen von Makro- oder Basiszellen
	HRY_INSTANCE		
	CELL		die Beschreibungen von Makro- oder Basiszellen
GLOBAL			zur Repräsentation globaler Daten
TEMPORARY			zur Repräsentation temporärer Daten

Tabelle A.1: die verfügbaren Objektklassen und Sonderfälle

Die Spalte *Busbreite* gibt an, ob die Objekte der betreffenden Klasse einen Bus repräsentieren können (mit *x* gekennzeichnet).

Anhang B

Anzeigeformen und Objektklassen

Objektklassen	direkt im Schaltkreisbild	im Dialogfenster des betreffenden Objekts	an der Baumdarstellung der Hierarchie	Trace-View
	textuelle Darstellung			
CELL, INSTANCE HRY_INSTANCE	DIRECT	ITEM	DIRECT	-
GLOBAL, TEMPORARY	-	ITEM	-	-
andere	DIRECT	ITEM	-	-
	einfärben von Objekten			
GLOBAL, TEMPORARY	-	-	-	-
andere	COLOR	-	-	COLOR
	farbliche Vorder- und Hintergrunddarstellung			
GLOBAL, TEMPORARY	-	FOREGROUND BACKGROUND	-	-
andere	FOREGROUND BACKGROUND	FOREGROUND BACKGROUND	FOREGROUND BACKGROUND	-
	Punkt-, Liniendiagramme			
TEMPORARY	-	-	-	-
andere	-	DIAGRAM	-	-
	Timing-Diagramme			
TEMPORARY	-	-	-	-
andere	-	TIMING_DIAGRAM	-	-
	Histogramme			
TEMPORARY	-	-	-	-
andere	-	HISTOGRAM	-	-
	hypertextähnliche Darstellung von Strukturen			
alle	-	ITEM	-	-

Tabelle B.1: Darstellungsformen je nach Objektklasse und Anzeigeort

Anhang C

Syntax der Beschreibungssprache

```
Letter     = [A-Za-z_]
Digit      = [0-9]
ID         = [A-Za-z] { Letter | Digit }*
MOD_ID     = ID {::} ID
INTCONST   = {[+-] | ϵ } { Digit }+
REALCONST  = {[+-] | ϵ } INTCONST '.' INTCONST |
             {[+-] | ϵ } INTCONST '.' INTCONST [Ee] {[+-] | ϵ } INTCONST
STRING     = '“'[^“]*'“'

BeschreibungsDatei:          MODUL '=' ID Beschreibung
                           | Beschreibung
                           ;
Beschreibung:                Strukturdeklarationen
                           | Objektpufferdeklarationen
                           | Anzeigekonfigurationen
                           ;
Strukturdeklarationen:       /* empty */
                           | struct ID '{' Strukturkörper '}' Strukturdeklarationen
                           ;
Strukturkörper:              Variablendeklaration
                           | Variablendeklaration Strukturkörper
                           ;
Variablendeklaration:        Typ ID ';'
                           | Typ ID '[' INTCONST ']' ';'
                           | Typ ID '[' INDEX_0 ']' ';'
                           | Typ ID '[' ID ']' ';'
                           ;
Typ:                         elementarerTyp
                           | signed elementarerTyp
                           | unsigned elementarerTyp
                           | char '*'
                           | struct ID
                           | struct ID '*'
```

```
                              | void '*'
                              | Color
                              ;
elementarerTyp:                 int | long | float | double | char
                              ;

Objektpufferdeklarationen:      Objektpufferdeklaration
                              | Objektpufferdeklaration Objektpufferdeklarationen
                              ;

Objektpufferdeklaration :       ObjektKlassenName '{' Deklarationsblöcke '}'
                              ;

Deklarationsblöcke:             Deklarationsblock
                              | Deklarationsblock Deklarationsblöcke
                              ;
Deklarationsblock:              Puffergröße '{' Pufferdeklarationen '}'
                              | Pufferdeklarationen
                              ;

ObjektKlassenName:              PIN | PAD | FANOUT | CNODE | WIRE | CEDGE | NET
                              | HRY_PIN | HRY_PAD | HRY_FANOUT | HRY_CNODE
                              | HRY_WIRE | HRY_CEDGE | HRY_NET | SIGNAL | INSTANCE
                              | HRY_INSTANCE | CELL | GLOBAL | TEMPORARY
                              ;
Pufferdeklarationen:            Pufferdeklaration ';'
                              | Pufferdeklaration ';' Pufferdeklarationen
                              ;
Pufferdeklaration:              Typ identifier
                              | Typ identifier '[' BUSWIDTH ']'
                              | Typ identifier '[' INTCONST ']'
                              | Typ identifier '[' INDEX_0 ']'
                              | Typ identifier '=' ID '(' ParameterListe ')'
                              | Typ identifier '=' Summe
                              | Typ identifier '=' FarbScala
                              ;
Summe:                          SUM '(' ObjektKlassenName ',' identifier ')'
                              ;
Anzeigekonfigurationen:         /* empty */
                              | DISPLAY '(' ObjektKlassenName ')'
                                '{' Anzeigeformen '}' Anzeigekonfigurationen
                              ;
Anzeigeformen:                  Anzeigeform
                              | Anzeigeform Anzeigeformen
                              ;
```

```
Anzeigeform:            MATRIX '(' INTCONST ')' '{' Name Elemente '}'
                      | ROW '{' Name Elemente '}'
                      | COLUMN '{' Name Elemente '}'
                      | Element
                      | COLOR '{' Name identifier '}'
                      ;
Elemente:               Element
                      | Element Elemente
                      ;
Element:                ITEM '{' Optionen formatierteAusgabe'}'
                      | DIRECT '{' Optionen formatierteAusgabe '}'
                      | TIMING_DIAGRAM '{' Optionen Takt Signal '}'
                      | HISTOGRAM '{' Optionen Histogrammblock '}'
                      | DIAGRAM '{' Optionen Diagrammblock '}'
                      ;
Optionen:               Name Vordergrund Hintergrund
                      ;
Vordergrundfarbe:       /* empty */
                      | FOREGROUND '=' STRING
                      | FOREGROUND '=' FarbScala
                      ;
Hintergrundfarbe:       /* empty */
                      | BACKGROUND '=' STRING
                      | BACKGROUND '=' FarbScala
                      ;
Takt:                   TAKT '=' identifier
                      ;
Signal:                 SIGNAL '=' identifier
                      ;
DiagrammBlock:          xAchse yAchse
                      | xAchse yAchse DiagrammOptionen
                      ;
xAchse:                 XVAR '=' identifier
                      ;
yAchse:                 YVAR '=' ParameterListe
                      ;
DiagrammOptionen:       DiagrammOption
                      | DiagrammOption DiagrammOptionen
                      ;
DiagrammOption:         TITLE '=' STRING
                      | XLABEL '=' STRING
                      | YLABEL '=' STRING
                      | LINE_STYLE '=' Linientyp
                      | MARKER_STYLE '=' Punkttyp
                      | XLOG '=' Boolean
                      | YLOG '=' Boolean
```

```
                          | DRAW_MAJOR '=' Boolean
                          | DRAW_MAJOR_TICS '=' Boolean
                          | DRAW_MINOR '=' Boolean
                          | DRAW_MINOR_TICS '=' Boolean
                          | DEGREES '=' Boolean
                          | CARTESIAN '=' Boolean
                          ;
Boolean:                    TRUE | FALSE
                          ;
Linientyp:                  NONE | SOLID | DOTTED | WIDEOUT
                          ;
Punkttyp:                   NONE | CIRCLE | SQUARE | DIAMOND
                          ;
HistogrammBlock:            xAchse yAchse
                          | xAchse yAchse HistogrammOptionen
                          ;
HistogrammOptionen:         HistogrammOption
                          | HistogrammOption HistogrammOptionen
                          ;
HistogrammOption:           TITLE '=' STRING
                          | XLABEL '=' STRING
                          | YLABEL '=' STRING
                          ;
formatierteAusgabe:         STRING
                          | identifier
                          | '<' identifier '>'
                          | STRING formatierteAusgabe
                          | identifier formatierteAusgabe
                          | '<' identifier '>' formatierteAusgabe
                          ;
identifier:                 ID | MOD_ID
                          ;
Name:                       /* empty */
                          | NAME '=' STRING
                          ;
Puffergröße:                HISTORY '=' INTCONST
                          ;
ParameterListe:             identifier
                          | identifier ',' ParameterListe
                          ;
FarbScala:                  SCALE '(' identifier ',' STRING ')'
                            '{' Intervalle '}'
                          ;
Intervalle:                 Intervall STRING
                          | Intervall STRING Intervalle
                          ;
```

```
Intervall:                     '[' REALCONST ',' REALCONST ']'
                             | '[' INTCONST ',' REALCONST ']'
                             | '[' INTCONST ',' INTCONST ']'
                             | '[' REALCONST ',' INTCONST ']'

                             | '(' REALCONST ',' REALCONST ']'
                             | '(' INTCONST ',' REALCONST ']'
                             | '(' INTCONST ',' INTCONST ']'
                             | '(' REALCONST ',' INTCONST ']'

                             | '(' REALCONST ',' REALCONST ')'
                             | '(' INTCONST ',' REALCONST ')'
                             | '(' INTCONST ',' INTCONST ')'
                             | '(' REALCONST ',' INTCONST ')'

                             | '[' REALCONST ',' REALCONST ')'
                             | '[' INTCONST ',' REALCONST ')'
                             | '[' INTCONST ',' INTCONST ')'
                             | '[' REALCONST ',' INTCONST ')'
                             ;
```

Literaturverzeichnis

[Bur88] Th. Burch. Ein grafisches Eingabesystem für CADIC. Master's thesis, Fachbereich Informatik, Universität des Saarlandes, Postfach 15 11 50, 66041 Saarbrücken, FRG, 1988.

[Bur95] Th. Burch. *Eine grafische Arbeitsumgebung zum parametrisierten Entwurf von Schaltkreisen.* PhD thesis, Fachbereich Informatik, Universität des Saarlandes, 1995.

[Gra92] B. Grande. Verfahren zur hierarchischen Schichtzuweisung und ihre Implementierung in CADIC. Master's thesis, Fachbereich Informatik, Universität des Saarlandes, 1992.

[HF94] D. Heller and P.M. Ferguson. *Motif Programming Manual.* O'Reilly & Associates, Inc., 1994.

[Hot65] G. Hotz. Eine Algebraisierung des Syntheseproblems für Schaltkreise. *EIK Journal of Information Processing and Cybernetics*, 1:185–205,209–231, 1965.

[Hot74] G. Hotz. *Schaltkreistheorie.* de Gruyter Lehrbuch. Walter de Gruyter Verlag, 1974.

[KHB93] R. Krieger, R. Hahn, and B. Becker. test_circ: Ein abstrakter Datentyp zur Repräsentation von hierarchischen Schaltkreisen. Technical Report 07/1993, Fachbereich Informatik, Universität Frankfurt, 1993.

[Kol86] R. Kolla. *Spezifikation und Expansion logisch topologischer Netze.* PhD thesis, Fachbereich Informatik, Universität des Saarlandes, 1986.

[Kre96] M. Kreutzer. Das Fanout-Problem auf flachen und hierarchischen Schaltungen. Master's thesis, Fachbereich Informatik, Universität des Saarlandes, Postfach 15 11 50, 66041 Saarbrücken, FRG, 1996.

[LMB92] J.R. Levine, T. Mason, and D. Brown. *Lex & Yacc.* O'Reilly & Associates, Inc., 2nd edition, 1992.

[Mic91] Sun Microsystems. *Referenz Manual, A Advanced Socket-Based Interprocess Communications Tutorial*, 1991.

[Mol86] P. Molitor. *Über die Bikategorie der logisch topologischen Netze und ihre Semantik*. PhD thesis, Fachbereich Informatik, Universität des Saarlandes, Postfach 15 11 50, 66041 Saarbrücken, FRG, 1986.

[Mol88] P. Molitor. Free net algebras in VLSI-theory. *Fundamenta Informaticae, Annales Societatis, Mathematicae Polonae*, XI:117–142, June 1988.

[Mol93] P. Molitor. A Hierarchy Preserving Hierarchical Bottom-up 2-Layer Wiring Algorithm with respect to Via Minimization. *INTEGRATION, the VLSI Journal*, 15:73–95, 1993.