

- `CreateFSM` erzeugt einen endlichen Automaten aus dem **PSFCfg**.
- `CreateNet` transformiert die virtuelle Netzliste in eine **PSFCellInst**.  
Zu diesem Zweck werden die zwei Listen, die die virtuelle Netzliste darstellen durchlaufen (`moduleTypeList`, `dataTypeList`). Für jeden Instanz Deskriptor wird ein Netzlistenelement erzeugt. Anschließend wird überprüft, ob in den `PSFVirtMuxDescr` mehr als ein Eingang beschrieben wird. Ist dies der Fall, so wird ein Multiplexer instantiiert. Die Schnittstelle des “umgebenden” Moduls wird um die Zahl der zur Ansteuerung der Multiplexer benötigten Selectleitungen erweitert, und der Selecteingang des Multiplexers mit diesen Ports verbunden.  
In einem zweiten Durchgang werden die Eingangsports aller Module (Register, Funktionseinheiten) bzw. ihrer Eingangsmultiplexer mit ihren Eingängen verbunden.  
Im dritten und letzten Schritt wird die Schnittstelle des Designs um die Ausgangsports erweitert, an denen die Statussignale angelegt werden (z.B. die Ergebnisse von Vergleichsoperationen zur Steuerung von bedingten Verzweigungen).

### 7.4.2.2 SimpleFUBinding

Die während des Scheduling vorgenommenen Zuordnung von Operationsknoten zu Modultypen wird in dieser Methode verfeinert. Jeder Operationsknoten wird genau einem Instanzdeskriptor des Modultyps zugeordnet.

Die Zuordnung ist Greedy. Das bedeutet in diesem Zusammenhang, daß keine weiteren Analysen, z.B. bezüglich der entstehenden Kosten für die Interkonnektion, vorgenommen werden. Entscheidend für die Zuordnung eines Operationsknotens zu einer Modulinstanz ist lediglich, daß die Instanz in dem Zeitabschnitt, in den die Operation geschedult wurde, noch nicht anderweitig verwendet wird.

### 7.4.2.3 LifetimeAnalysis

Um eine optimale Ausnutzung der Register zu erreichen, muß die Lebensdauer der Variablen bekannt sein.

Für die Analyse der Lebenszeit wird für jeden Datenknoten die Zeitspanne zwischen der Initialisierung des Knotens (der `Cstep`, in dem die Quelle geschedult wurde) und der Verwendung des Wertes (`Cstep` der Senke - 1) ermittelt. Die Lebenszeit einer Variablen ergibt sich aus dem Minimum und dem Maximum aller Werte der Knoten, die das gleiche `PSFCsel` referenzieren.

### 7.4.2.4 SimpleRegisterAllocationAndBinding

Auch bei dieser Zuordnung werden keine Optimierungen bezüglich der Leitungslängen oder anderen Kriterien vorgenommen. Dies hat zur Folge, daß zwar eine minimale Zahl an Registern verwendet wird, das resultierende Design jedoch nicht unbedingt die kleinstmögliche Realisierung darstellt.

### 7.4.2.5 InterconnectionBinding

Beim Interconnection Binding wird die "Verdrahtung" der in der virtuellen Netzliste enthaltenen Instanzen ermittelt.

Der `PSFDfg` wird durchlaufen und für jeden Knoten getestet, ob die annotierte Modulinstanz bereits mit der Modulinstanz der Quelle verbunden wurde. Die Verbindungsinformation wird durch Pointer dargestellt, die von den "Eingängen" (`PSFInputDescr`) der `PSFVirtMuxDescr` auf den Instanz Deskriptor der Quelle zeigen. Die Zeitpunkte (symbolische Zustände), zu denen der Datentransport stattfindet, werden ebenfalls gespeichert. Durch Auslesen dieser Zeitinformationen können bei der Erzeugung des endlichen Automaten, die Werte der Enable- und Select-Signale in den einzelnen Zuständen bestimmt werden.

### 7.4.2.6 CreateNet

Diese Methode ist zweigeteilt:

## 7.4 Die Klasse PSFSynthese

### 7.4.1 Member von PSFSynthese

Im wesentlichen besteht ein Objekt der Klasse **PSFSynthese** aus zwei Datenelementen: `moduleTypeList` und `dataTypeList`. Die beiden Listen enthalten Elemente vom Typ `PSFFUTypeDescr` bzw. `PSFDataTypeDescr`. Sie stellen die während der Synthese aufgebaute virtuelle Netzliste dar. Weitere Listen sind vorhanden für eine typbezogene Verarbeitung der einzelnen Knoten des **PSFCdfg**. So gibt es Listen für Operationen, einfache Datentypen, Arrays, etc. Diese Listen werden im Konstruktor initialisiert.

### 7.4.2 Methoden von PSFSynthese

#### 7.4.2.1 StaticListScheduler

In seiner jetzigen Version verarbeitet der Scheduler keine Informationen über die Ausführungszeiten eines Moduls. Das bedeutet, daß Chaining und Behandlung von Mehr-Zyklus Instruktionen nicht möglich sind.

Der Scheduler erhält als Eingabe (indirekt über die member-Variable `moduleList`) eine Liste von Modultypen, die er zur Realisierung der Operationen des **PSFDfgs** verwenden darf. Zu jedem dieser Modultypen ist festgelegt, wie viele Instanzen von ihnen maximal verwendet werden dürfen. Durch diese Angabe wird die Zahl der Operationen eines Typs begrenzt, die in einem Zyklus parallel abgearbeitet werden kann. Können in einem Schritt mehr Operationen gescheduled werden, als Module zur Verfügung stehen, so erfolgt die Auswahl der Operationsknoten mittels einer Kostenfunktion. Die Kostenfunktion bevorzugt die Knoten, mit der längsten verbleibenden Pfadlänge bis zum Ende des Basic-Blocks.

Die durch den Scheduler festgelegte Ausführungsreihenfolge der einzelnen Operationen basiert auf der Zuordnung eines jeden Operationsknotens zu einem bestimmten Modultyp. Jeder Knoten wird mit dieser Zuordnung annotiert (in Form eines Pointers auf den entsprechenden Modultypdeskriptor).

Die `Csteps`, mit denen die **PSFDfgs**-Knoten annotiert werden, sind lokal. Das bedeutet, in jedem Basic Block werden die `Csteps` wieder bei 1 beginnend durchgezählt. Im **PSFCfgs** werden die `Csteps` der korrespondierenden **PSFDfgs** aufaddiert. Man erhält einen global eindeutigen `Cstep` eines **PSFDfgs** Knotens **K** durch Addition des lokalen Wertes mit dem akkumulierten `Cstep` des **PSFCfgs** Knotens von **K**.

Neben den `Csteps` werden im Scheduler eindeutige symbolische Zustände vergeben. Dies ist sinnvoll, damit sich unterschiedliche Pfade im **PSFCfgs** auch durch unterschiedliche Zustände im **PSFDfgs** widerspiegeln.

Jeder Deskriptor für eine Modulinstanz stellt ein späteres Element der “realen” Netzliste dar. In ihm werden während der High Level Synthese Informationen über seine Belegung und seine Verbindungen gesammelt. Beide Instanz-Deskriptoren, `PSFDataInstanceDescr` und `PSFFUInstanceDescr`, werden von der Klasse `PSFInstanceDescr` abgeleitet. In den Objekten dieser Klasse existiert ein Backpointer auf den jeweiligen Typdeskriptor und ein Pointer auf die Zelle in der realen Netzliste, die dieser Beschreibung entspricht. Weiterhin ist beiden Deskriptoren gemeinsam, daß sie eine Beschreibung ihres Eingangsmultiplexers enthalten (bei nicht monadischen Funktionen sind es natürlich mehrere).

Zusätzlich werden während der HLS für die Funktionsmodule die symbolischen Zustände eingetragen, in denen das jeweilige Modul verwendet wird. Bei multifunktionalen Einheiten wird auch der Wert des Funktion-Selectsignals in den einzelnen symbolischen Zuständen ermittelt.

Für die Datenmodule wird die Belegungszeit durch eine bestimmte Variable in der Klasse `PSFDataUsage` eingetragen. Diese erhält neben der reinen Zeitinformation in globalen `csteps` auch einen Rückverweis auf den Knoten im `PSFDfg`, der diese Variable repräsentiert.

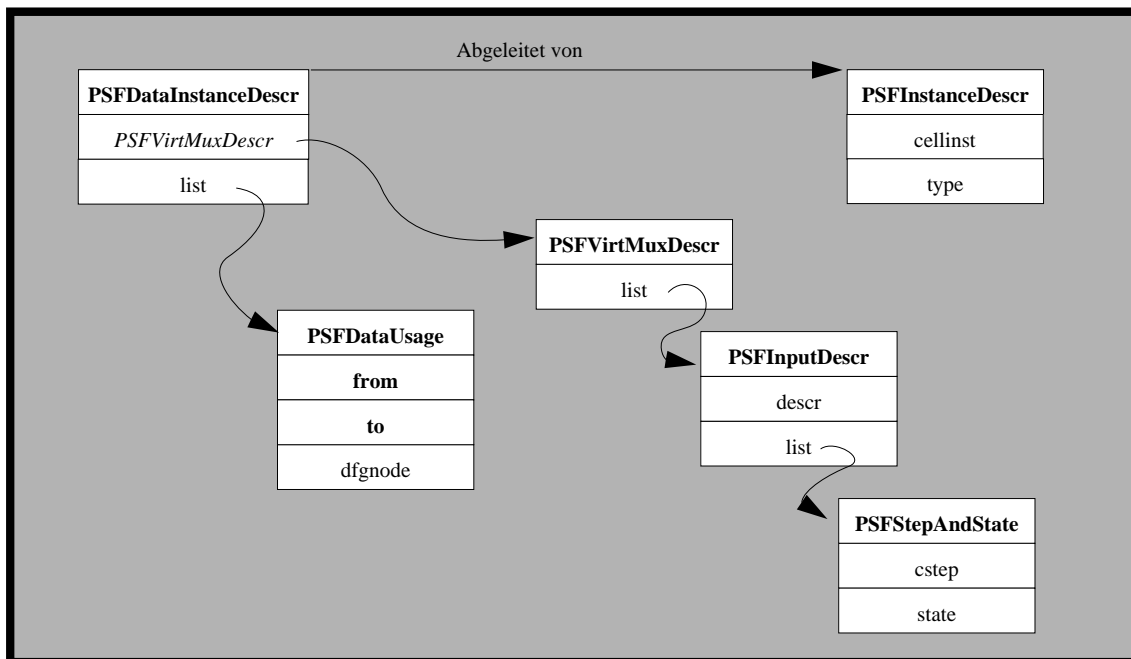


Abb. 40: Definition des Instanz-Deskriptors für ein Datenelement

`int used`, Zahl der in ein einem Schdulingschritt bereits belegten Einheiten). Beide Klassen zur Beschreibung von Modultypen haben als Datenelement eine Liste, die Zeiger auf die Deskriptoren der Instanzen des jeweiligen Modultyps enthält.

Solange im PSF noch kein Registerfile synthetisiert werden kann, hat der `PSFDataTypeDescr` eine Doppelbedeutung. Zum einen verwaltet er die Spezifikation eines Registers und die Instanzen der Module mit dieser Spezifikation. Seine zweiter “Verwendungszweck” ist die Beschreibung der “globalen” Daten eines Arrays. Dies bedeutet, daß der `PSFDataTypeDescr` zur Zeit ein Flag beinhaltet, durch den die von ihm verwalteten Instanzen als Arrayelemente identifiziert werden können. Ist dies der Fall, verwaltet der `PSFDataTypeDescr` zusätzlich die Beschreibungen der Multiplexer und Demultiplexer, über die die Arrayelemente mittels eines Variablenwertes referenziert werden können (siehe Abbildung 37). Da dies keine endgültige Lösung darstellt, wird in den Abbildungen sowie in den folgenden Abschnitten nicht näher darauf eingegangen.

Die Beschreibung für Multiplexer und Demultiplexer erfolgt durch die Klasse `PSFVirtMuxDescr`. Da jedem Modul, bzw. dessen Deskriptor, Eingangsmultiplexer zugeordnet werden, wird in den Objekten dieser Klasse die gesamte Verbindungsinformation der Netzliste verwaltet. Die Definition der Klasse `PSFVirtMuxDescr`, sowie Ihre Stellung in der virtuellen Netzliste, geht aus den Abbildungen 39 und 40 hervor.

### 7.3.2 Deskriptoren für Modulinstanzen

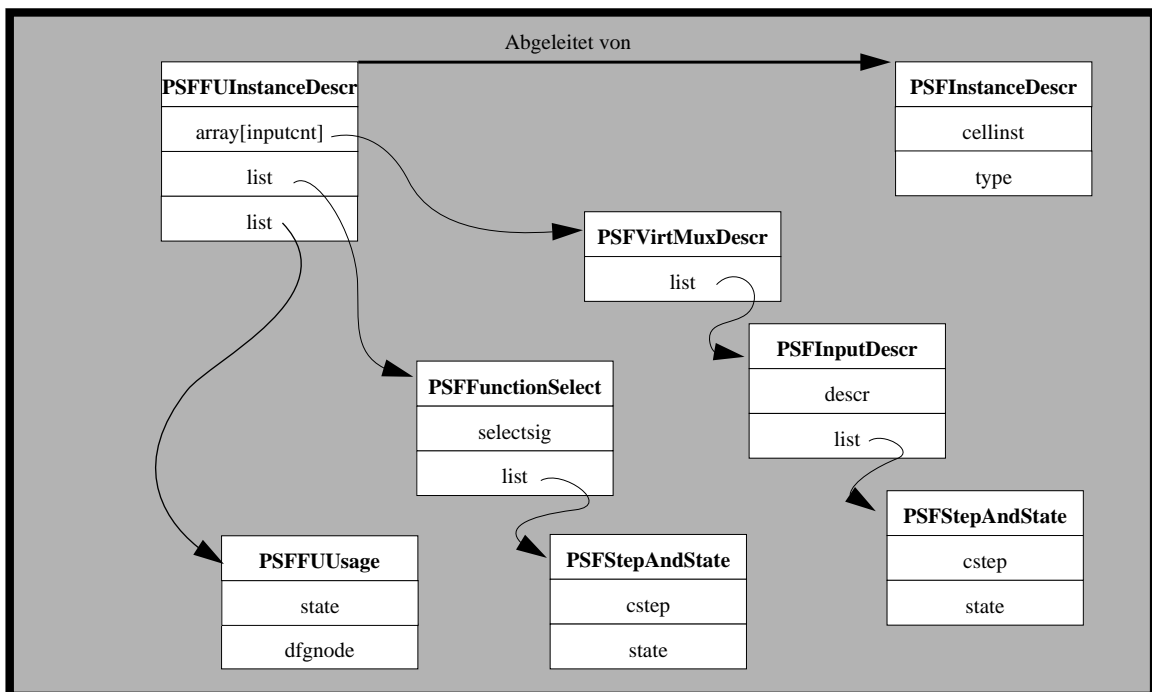


Abb. 39: Definition des Instanz-Deskriptors für eine Funktionale Einheit

eine weitere Bearbeitung des Designs auf Register-Transfer-Level besser zu handhaben ist, bzw. auf durch die sie erst ermöglicht wird. Als Algorithmus auf RT-Ebene sei als Beispiel Retiming genannt.

Bei der Transformation der Darstellung von einem **PSFCdfg** in eine Netzliste mit dazugehörigem endlichen Automaten wird eine Zwischenstruktur aufgebaut, die wir als virtuelle Netzliste bezeichnen. Dabei handelt es sich um Datenstrukturen, aus denen sich alle Informationen zum Aufbau der Netzliste ableiten lassen, die aber voll generisch sind. Dies wird dadurch erreicht, daß im Gegensatz zur Netzliste keine Instantiierungsinformationen verwaltet werden. Außerdem wird die Verbindungsinformation über Pointer und nur in einer Richtung aufgebaut. Als Beispiel soll die Beschreibung von Eingangsmultiplexern einer FU betrachtet werden: durch die "einseitige" Beschreibung der Verbindungsinformation von der Senke aus, lassen sich zwei Eingänge einer FU durch eine "lokale" Operation vertauschen (einfach durch Vertauschung der Pointer). Die Datenstrukturen zur Beschreibung der Eingänge (Quellen) werden für diese Operation nicht benötigt. Erst bei der Erzeugung der "echten" Netzliste wird entschieden, ob die FU tatsächlich Eingangsmultiplexer benötigt.

Die virtuelle Netzliste stellt eine Erweiterung des Konzeptes der Synthese durch Annotierung dar. Die Beschreibung einer Funktionalen Einheit kann in sofern als Annotierung interpretiert werden, als das jeder Operatorknoten im **PSFDFg** einen Pointer auf den Deskriptor des Moduls enthält, von dem seine Funktion ausgeführt wird.

### 7.3.1 Deskriptoren für Modultypen

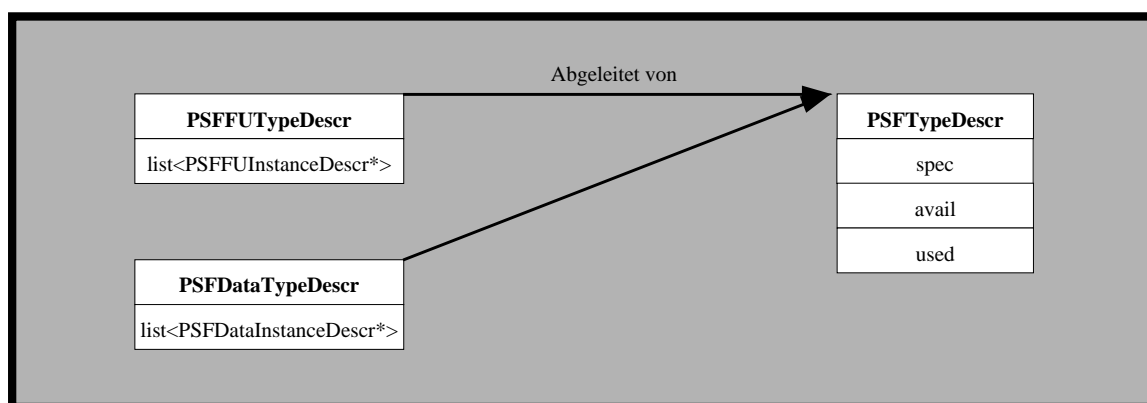


Abb. 38: Definition der Deskriptoren für Modultypen

Die für die Synthese im PSF verwendeten Module lassen sich in drei Typen unterscheiden: Datenmodule (Register), Funktionsmodule und Steuermodule (z.B. Multiplexer). Zur Beschreibung der ersteren beiden Typen dienen die Klassen `PSFFUTypeDescr` und `PSFDataTypeDescr`, die von der gemeinsamen Basisklasse `PSFTypeDescr` abgeleitet werden. Die Elemente der Basisklasse dienen der Verwaltung der Typbeschreibung (`PSFHWSpec* spec`) und der Verwaltung der verfügbaren Module (`int avail`, Gesamtzahl der Module;

## 7.2 High Level Synthese

Ausgangspunkt der High Level Synthese (HLS) im PSF ist die Beschreibung eines Verhaltens als **PSFCdfg**. Bevor eine Umsetzung dieser Beschreibung in den **PSFCdp** erfolgen kann, benötigt man folgende Informationen:

- Welche Module stellt die Bibliothek zur Verfügung und welche Charakteristika haben diese Module.
- Wie groß sind die verfügbaren Ressourcen an Funktionalen Einheiten (FU Allocation).
- In welchem Takt wird eine Operation ausgeführt (Scheduling).
- Auf welcher FU wird die Operation X ausgeführt (FU Binding).
- Wieviele Register werden benötigt (Register Allocation).
- In welchem Register liegt das Datum A (Register Binding).
- Welche Verbindungen (Leitungen und Multiplexer) werden benötigt, damit das Datum A aus Register R im Takt n an der FU X anliegt (Interconnection Binding).

Es ist allgemein bekannt, daß die Algorithmen der HLS große Abhängigkeiten untereinander aufweisen. Es existieren Algorithmen, die eine Menge der Teilprobleme (Scheduling, Register- und FU-Allocation und Binding, Interconnection Binding) gleichzeitig bearbeiten. Oftmals basieren diese Algorithmen auf dem Lösen linearer Gleichungssysteme. Mit diesem Ansatz ist jedoch i.a. die Bearbeitung von Designs realer Größe nicht in annehmbarer Zeit erreichbar. Daher wurde versucht, die auf dem PSF implementierten High-Level-Synthese Algorithmen soweit wie möglich zu modularisieren. Dadurch ist der Benutzer in der Lage, eine auf seinen Bedarf abgestimmte Abfolge von Algorithmen ausführen zu lassen, je nach Zielarchitektur, vorgegebenen Constraints, etc.. Wenn z.B. nur die Auswirkungen zweier Transformationen auf dem **PSFCdfg** bezüglich der Größe des Controllers verglichen werden sollen, so muß nicht die gesamte Netzliste erzeugt werden, was eine enorme Zeitersparnis zur Folge hat. Neben der erhöhten Flexibilität des Systems für den Anwender, verbessert dieser Ansatz auch die Wiederverwendbarkeit des Codes.

In der ersten Version des Systems PSF wird eine Primitiv-Synthese gewählt. Register-Allocation und Binding werden bei dem gewählten Algorithmus gleichzeitig vorgenommen. Als Schedulingmethode ist das List-Scheduling implementiert.

## 7.3 Das Konzept der virtuellen Netzlisten

Die Konvertierung zwischen den beiden Abstraktionsebenen des PSF stellt nichts anderes dar, als eine HLS mit anschließender Transformation der Darstellung in eine Form, die für

### 7.1.1 Synthese von Arrays

Wie bereits in der Einleitung zu diesem Abschnitt bemerkt, können in der aktuellen Version des PSF keine Registerfiles synthetisiert werden. Auch ein weiteres Feature, mit dem man die Synthese von Arrays optimieren könnte, ist im Augenblick noch nicht implementiert: es werden keine Algorithmen zur Analyse der Zugriffe zur Verfügung gestellt. Dies bedeutet, daß bei einer Referenzierung der Arrayelemente durch Variablen, der Bereich des Zugriffes nicht abgeschätzt werden kann. Daher geht der Algorithmus zur Synthese von Arrays immer vom schlechtesten Fall aus. Hat z.B. ein Operator ein, über eine Variable dereferenziertes Arrayelement als Operanden, so wird der entsprechende Eingang der Funktionalen Einheit mit einem Multiplexer verbunden, der die Ausgänge aller Arrayelemente, bzw. der sie darstellenden Register, zusammenfasst.

Der Scheduler läßt für ein Array in jedem Takt nur einen schreibenden Arrayzugriff zu, der über eine Variable indiziert wird. Der zu schreibende Wert wird über einen Demultiplexer an das entsprechende Arrayelement geführt. Vor dem Demultiplexer befindet sich ein weiterer Multiplexer, über den die Quelle der Zuweisung selektiert wird.

Um gleichzeitige schreibende Zugriffe auf durch Konstanten indizierte Arrayelemente zu erlauben, erhält jedes der Arrayelemente einen Eingangsmultiplexer. Dies ist vor allem für eine effiziente Initialisierung von Bedeutung.

Die Netzlistenstruktur zur Realisierung eines Arrays zeigt Abbildung 37.

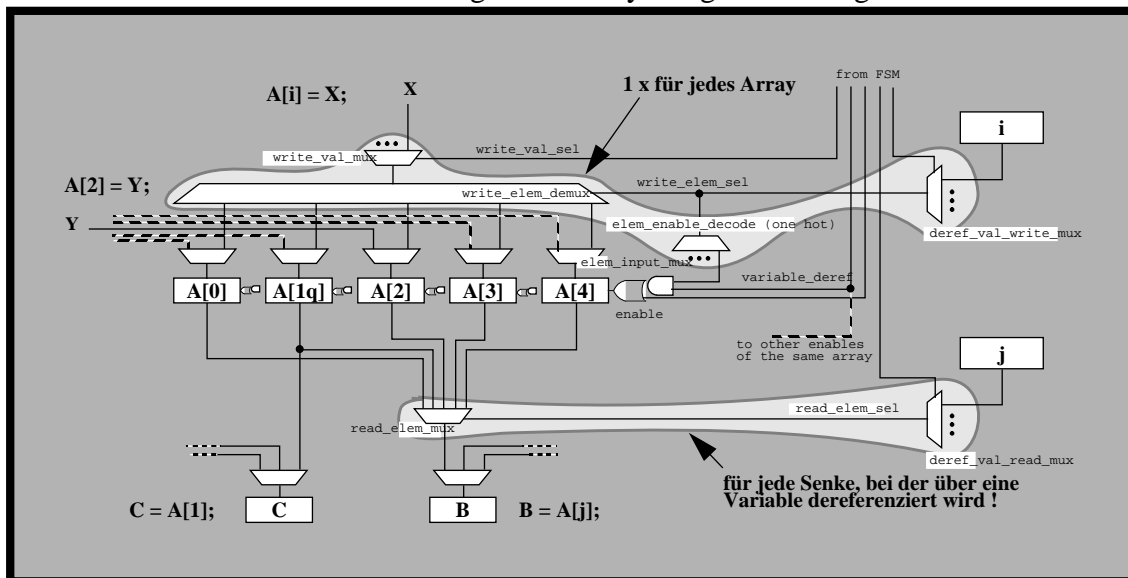


Abb. 37: Realisierung von Arrayzugriffen in der Netzliste

## 7 Die High Level Synthese im PSF

In diesem Kapitel wird beschrieben, wie innerhalb des PSF die Umsetzung einer Beschreibung vom **PSFCdfg** in den **PSFCdp** erfolgt. Diese Umsetzung erfolgt im wesentlichen durch Annotierung des **PSFCdfg** durch die High Level Synthese und anschließendes Auslesen der Informationen zum Aufbau der Netzliste und des endlichen Automaten.

Dieses Kapitel ist wie folgt organisiert. In Abschnitt 7.1 wird ein Überblick über die besonderen Merkmale der erzeugten Designs gegeben. Daran schließen sich Erläuterung der grundlegenden Konzepte der High Level Synthese im PSF an. In Abschnitt 7.3 wird der Begriff der virtuellen Netzliste eingeführt. Diese stellt eine Erweiterung des Konzepts der Annotierung dar. Abschnitt 7.4 beschreibt die Klasse **PSFSynthese**, die die Methoden zur High Level Synthese und zur Konvertierung beinhaltet.

### 7.1 Die Architektur des zu synthetisierenden Systems

Dieser Abschnitt soll einen Überblick darüber geben, nach welchem Modell die Synthese im PSF erfolgt. Die grundlegenden Merkmale sind

- multiplexerbasierter Entwurf, es werden keine Busse synthetisiert.
- Realisierung aller Register durch MS-FFs.
- Speicherung aller Daten in einzelnen Registern. Diese Einschränkung ist nicht Bestandteil des Konzeptes des PSF, sondern spiegelt nur den augenblicklichen Stand der Implementierung (Dezember 1993) wider. In einer der nächsten Versionen wird es auch möglich sein, mehrere Register in Registerfiles zusammenzufassen.

Dabei bedeuten:

- `from` : Nummer des Quellknotens
- `to` : Nummer des Zielknotens

- `void create_pointer()` erzeugt einen Zeiger. Ein Aufruf sieht so aus:

- `create_pointer(int type, CFGnode from, DFGnode to)`
- `create_pointer(int type, DFGnode from, CeSEL to)`
- `create_pointer(int type, DFGnode from, CFGnode to)`

Dabei bedeutet:

- `type` : Der Typ des Zeigers wird hier angegeben. Es gibt vier verschiedene Typen:
- `PTYPE_DFG` zeigt von einem **PSFCfgNode** auf den `DFG_BEGIN`-Knoten des zugehörigen DFGs.
- `PTYPE_EXPR` zeigt von einem **PSFCfgNode** (z.B. `IFNODE`) auf einen **PSFDfgNode** mit dem Ergebnis eines Ausdrucks.
- `PTYPE_CSEL` zeigt von einem **PSFDfgNode** auf ein dazugehöriges **PSFCsel**.
- `PTYPE_CFG` zeigt von einem **PSFCfgNode** auf den dazugehörigen **PSFDfgNode**. Dieser Zeiger wird allerdings schon durch die Erzeugung eines **PSFDfgNodes** gesetzt. Dieser Typ wird also normalerweise nicht benötigt.

Dabei bedeuten:

- name : Name des Operators
- Rückgabewert : eindeutige Nummer des **PSFCsels**.

Die Funktionsaufrufe für die weiteren Typen der PSFCSELID sind noch nicht definiert.

- void create\_edge() erzeugt eine Kante zwischen beliebigen Knoten oder erweitert eine bereits vorhandene (Hyper-)Kante um Quell- bzw. Zielknoten. Ein Aufruf der Funktion sieht so aus:

- create\_edge(CFGnode from, CFGnode to)
- create\_edge(CFGnode from, CFGlist \*to)
- create\_edge(CFGlist \*from, CFGnode to)
- create\_edge(CFGlist \*from, CFGlist \*to)
- create\_edge(DFGnode from, DFGnode to)
- create\_edge(DFGnode from, DFGnode to, int new)
- create\_edge(DFGnode from, DFGList \*to)
- create\_edge(DFGnode from, DFGList \*to, int new)
- create\_edge(DFGList \*from, DFGnode to)
- create\_edge(DFGList \*from, DFGnode to, int new)
- create\_edge(DFGList \*from, DFGList \*to)

Dabei bedeuten:

- from : Nummer des Quellknotens bzw. einer Liste von Quellknoten
  - to : Nummer des Zielknotens bzw. einer Liste von Zielknoten
  - new : wird hier NEW\_TARGET übergeben, werden ab hier keine weiteren Ziele an eine evtl. bereits zum Zielknoten führende Kante angehängt, sondern eine neue Kante erzeugt. Wird NEW\_SOURCE übergeben, werden keine weiteren Quellen angehängt, sondern eine neue Kante erzeugt. Wird NOT\_NEW oder nichts angegeben, so wird an eine bestehende Hyperkante eine neue Quelle oder ein neues Ziel hinzugefügt. Existiert noch keine solche Kante, so wird sie neu erzeugt.
- void delete\_edge() entfernt eine Kante zwischen beliebigen Knoten oder entfernt Quell- bzw. Zielknoten einer Kante. Ein Aufruf der Funktion sieht so aus:
- delete\_edge(CFGnode from, CFGnode to)
  - delete\_edge(DFGnode from, DFGnode to)

- `create_node(TYPE)` : erzeugt ein Element des **PSFCdfgs**. Was für ein Element erzeugt werden soll, wird durch *TYPE* festgelegt. *TYPE* kann vom Typ `PSFCFGNODEID`, `PSFDFGNODEID` oder `PSFCSELID` sein). Danach folgt je nach *TYPE* eine unterschiedliche Anzahl von Argumenten verschiedenen Typs. Die verschiedenen Varianten sind im folgenden aufgelistet:
- **CFG-Knoten:** folgende Funktionsaufrufe sind möglich:
  - `CFGnode create_node(PSFCFGNODEID)`
  - `CFGnode create_node(PSFCFGNODEID, int line_no)`.
 Dabei bedeuten:
  - `line_no`: Zeilennummer der zugehörigen VHDL-Anweisung im Quelltext
  - Rückgabewert: eindeutige Nummer des Knotens
- **DFG-Knoten:** folgende Funktionsaufrufe sind möglich:
  - `DFGnode create_node(PSFDFGNODEID, CFGnode daddy)`
  - `DFGnode create_node(PSFDFGNODEID, CFGnode daddy, int line_no)`
 Dabei bedeuten:
  - `daddy` : Nummer des zu diesem Knoten gehörenden **PSFCfgNodes**
  - `line_no` : Zeilennummer der zugehörigen VHDL-Anweisung im Quelltext
  - Rückgabewert: eindeutige Nummer des Knotens
- **CSELS:**
- Typ `PSFVarID`: `CSEL create_node(PSFVarID, char *name)`  
 Dabei bedeuten:
  - `name` : Name der Variablen
  - Rückgabewert: eindeutige Nummer des CSELS
- Typ `PSFConstID`: `CeSEL create_node(PSFConstID, int value)`  
 Dabei bedeuten:
  - `value` : Wert der Konstanten. Es sind momentan nur Konstanten vom Typ `Integer` möglich.
  - Rückgabewert : eindeutige Nummer des CSELS
- Typ `PSFOpID`: `CeSEL create_node(PSFOpID, char *name)`

Das macht eine strikte Trennung der Methoden des **PSFCdfg** und des SPI erforderlich. Daher wird zur Erzeugung des PSF-Graphen eine programminterne Schnittstelle verwendet. Diese ist durch folgende Funktionen realisiert:

- `init_graph()`
- `build_graph()`
- `create_node(IDTYPE, ...)`
- `create_edge()`
- `delete_edge()`
- `create_pointer()`

Die **PSF.ID** legt den Typ des zu erzeugenden oder zu löschenden Knotens fest. Danach folgt eine je nach Typ unterschiedliche Anzahl von Angaben zur Beschreibung des Elements (s.u.).

Hier wird die Möglichkeit des Überladens von Funktionen ausgenutzt. Dadurch ist es auch möglich, die Anzahl der beschreibenden Argumente zu erhöhen ohne vorhandenen Code ändern zu müssen.

Ein weiteres Problem ist die Handhabung von Hyperkanten. Innerhalb des Konverters werden immer nur die beiden zu verbindenden Knoten angegeben und einige Verbindungen aufgrund des verwendeten Algorithmus wieder gelöscht. Außerdem werden Kanten gelöscht, welche nicht erzeugt wurden. Daher wird nicht sofort der Graph erzeugt, sondern zunächst die gewünschten Elemente in einer Liste gespeichert. Das Löschen von Elementen kann dann direkt aus dieser Liste erfolgen. Zu Beginn der Übersetzung wird diese Liste mit `init_graph()` initialisiert. Danach können Elemente mit `create_node` eingefügt, mit `create_edge()` oder `create_pointer()` verbunden und mit `delete_edge()` entfernt werden. Nach der vollständigen Übersetzung des VHDL-Programms wird dann aus dieser Liste mit `build_graph()` der PSF-Graph erzeugt.

Die Funktionen im einzelnen:

- `void init_graph()`: muß vor dem Aufbau des PSF-Graphen aufgerufen werden und initialisiert die notwendigen Strukturen.
- `CDFG *build_graph()`: baut den durch die Funktionen `create_node()`, `create_edge()` und `delete_edge()` beschriebenen Graphen auf und gibt ihn als Ergebnis zurück.

- `void free_CFGList(struct CFGList *&list)` gibt eine CFGList (Liste von **PSFCfgNode**, z.B. für CDFG\_Conn) frei.
- `void free_DFGList(DFGList *&list)` gibt eine DFGList frei.
- `void add_cfg(CFGList *&list, CFGnode node)` `void add_dfg(DFGList *&list, DFGnode node)` erweitert die gegebene Liste von **PSFCfgNode** oder **PSFDfgNode** um einen Knoten.
- `void add_cfg(CFGList *&list, CFGList *newlist)` `void add_dfg(DFGList *&list, DFGList *newlist)` hängt eine CFG/DFG-Liste an eine andere CFG/DFG-Liste an.
- `struct CDFG_Conn *new_conn()` erzeugt eine neue CDFG\_Conn-Struktur.
- `void free_conn(struct CDFG_Conn *&conn)` löscht sie wieder.
- `void merge_conn(CDFG_Conn *cdfg, CDFG_Conn *addcdfg)` hängt den **PSFCdfg** addcdfg an den **PSFCdfg** cdfg an. addcdfg wird dabei gelöscht.
- `void free_bblast(struct BB_Varlist *&list)` entfernt eine BB\_Varlist (für die Basic Block-Verwaltung).
- `struct BB_Varlist *find_bbvar(struct BB_Varlist *list, char *name)` sucht die durch den Namen gegebene Variable in der angegebenen BB\_Varlist und gibt einen Zeiger auf die entsprechende Struktur zurück. Ist die Variable noch nicht in der Liste enthalten, wird NULL zurückgegeben.
- `void add_varlist(struct BB_Varlist *&list, DFGnode node, char *name)` hängt ein neues Element an die BB\_Varlist an.
- `CeSEL get_const_csel(int value)` gibt das **PSFCsel** der gegebenen Konstanten zurück. Falls es noch kein **PSFCsel** gibt, wird eines erzeugt.
- `void end_constlist()` beendet die Liste aller **PSFConstCsels**.

### 6.2.3 Schnittstelle zum PSF-Graph

Datei `graph.c`

Da es im SPI und in dem **PSFCdfg** einige Bezeichner gibt, welche in beiden Umgebungen gleich sind, darf jeweils nur der **PSFCdfg** oder nur das SPI eingebunden werden.

1.  $a$  ist in der `bb_varlist` vorhanden und gültig, also wird dieser Knoten als erster Operand benutzt. Für  $b$  als zweiten Operanden gilt das gleiche.
2. Die Operatorknoten '\*' und ':=' werden eingerichtet.
3. Der Knoten  $c$  wird eingerichtet und als gültig erklärt an die `bb_varlist` angehängt.

#### 6.2.2.4 Expressions

Datei `expr.c` In diesem Modul werden die DFGs erzeugt. Dazu werden die im DLS schon als Baum vorliegenden, aufgelösten Ausdrücke rekursiv in einen DFG eingefügt. Es wird dabei besonders auf die korrekte Behandlung der "Basic Blocks" geachtet.

- `void make_dfg_assignment(CFGnode stmt, Node source, Node target, struct StmtListData *sld)` erzeugt aus den gegebenen Ausdrücken (`source`, `target`) einen DFG für eine Zuweisung. `stmt` ist der korrespondierende **PSFCfgNode**, `sld` die `StmtListData`-Struktur, die zur Behandlung der Basic Blocks benötigt wird. Der **PSFDfg** wird daher ggfs. in einen bereits vorhandenen **PSFDfg** eingehängt.
- `void make_dfg(CFGnode stmt, Node expr, DFGBEGINnode *first, DFGENDnode *last, struct StmtListData *sld)` erstellt einen DFG für Ausdrücke, die z.B. für Auswertungen in `if`- oder Schleifenkonstrukten vorkommen. Daher wird nur ein (DLS-)Knoten übergeben, nämlich der `expression-node` selbst.
- `void make_dfg_declaration(CFGnode stmt, Node source, Node target, struct StmtListData *sld)` dient zur Erzeugung eines DFG für die Zuweisung während einer Deklaration.
- `void make_dfg_compare(CFGnode stmt, Node op1, Node op2, struct StmtListData *sld, char *op)` wird in Zählschleifen benötigt und erstellt einen DFG aus zwei Operanden `op1` und `op2`, die mit einem Operator `op` verknüpft werden.

#### 6.2.2.5 Lists

Datei `lists.c` Hier werden Funktionen für die Verwaltung von verschiedenen Listen und anderen Datentypen zur Verfügung gestellt.

- `void init_oplist()` `void end_oplist()` initialisiert bzw. beendet eine Liste aller Operatoren.
- `CeSEL find_opcsel(char *op)` ermittelt das **PSFCsel** des gegebenen Operators. Ist noch kein **PSFCsel** vorhanden, wird eine erzeugt.

- CDFG\_Conn \*process\_statement(Node stmt, struct StmtListData \*sld) erzeugt den **PSFCdfg** für einen einzelnen VHDL-Befehl. Bei Befehlen, die eine oder mehrere weitere Anweisungsfolgen enthalten (z.B. die verschiedenen Zweige beim if-Befehl, Schleifen oder die process-Anweisung, werden diese durch process\_statementlist() rekursiv in weitere Teil-CDFGs konvertiert.

Nun zur Behandlung von Basic Blocks: Die Basic Blocks bestehen nur aus Zuweisungen (im DLS: AssignmentStatement). Wenn eine Anweisung behandelt wird, gibt es drei Möglichkeiten:

1. der Befehl ist eine Zuweisung, aber der vorige war keine Zuweisung: in diesem Fall wird im **PSFCdfg** ein STM\_BLOCKNODE erzeugt, im **PSFDFg** der dazu passende Teil-DFG. Weiterhin wird eine Struktur StmtListData angelegt, welche Informationen über diesen DFG enthält (bb\_dfgbegin, bb\_dfgend, bb\_stmt) sowie in den Element bb\_varlist eine Liste mit den bisher in diesem DFG verwendeten Variablen. Hier wird u.a. gespeichert, ob die Variable gültig ist, d.h. ob eine im Programm folgende Zuweisung sich auf den Knoten dieser Variable beziehen kann oder einen neue Knoten anlegen muß (Datenabhängigkeit).
2. der Befehl ist eine Zuweisung und der vorige Befehl war auch eine Zuweisung: Jetzt wird die neue Zuweisung in den DFG eingehängt, der durch die angelegte StmtListData-Struktur bestimmt ist. Dabei wird durch den Eintrag bb\_varlist festgestellt, welche Variablen und Konstanten bereits in diesem Block vorkommen.
3. der Befehl ist keine Zuweisung: die Variablenliste in der StmtListData-Struktur wird gelöscht.

Ein Beispiel soll nun das Vorgehen bei der Erzeugung der Basic Blocks erläutern (ohne die Erzeugung von Kanten):

- erster Befehl:  $a := a + b;$
1. Die Knoten  $a$  und  $b$  werden angelegt: sie werden an die bb\_varlist angehängt und für gültig erklärt.
  2. Es wird festgestellt, daß die Zielvariable ( $a$ ) schon vorhanden ist, daher wird sie für ungültig erklärt. Dies geschieht, da man von nun an keine Kanten mehr von dem Operanden  $a$  ziehen darf, weil ja  $a$  ein neuer Wert zugewiesen wird.
  3. Die Operator-knoten '+' und ':=' werden eingerichtet.
  4. Da  $a$  ungültig ist, wird ein neuer Knoten dafür angelegt (expr.c, Zeile 57) und eine Kante vom alten  $a$  zu diesem neuen Knoten gezogen. Das alte  $a$  wird in der StmtListData-Struktur durch das neue ersetzt und dieses für gültig erklärt.
- zweiter Befehl:  $c := a * b;$

Die Erzeugung der **PSFDFgs** erfolgt zu jedem Knoten im **PSFCfgs** einzeln durch eine passende Funktion im Modul `expr.c`.

## 6.2.2 Module

Hier werden für jedes Modul die öffentlichen Schnittstellen erläutert, sowie die Funktion der wichtigsten Programmteile.

### 6.2.2.1 Konverter

Datei `convert.c`

- `CDFG *VHDL_in(char *entity, char *architecture, bool debug_on)` nimmt Initialisierungen vor und erzeugt den Graph zur gegebenen VHDL-Beschreibung. Wird der Parameter `debug_on` auf `TRUE` gesetzt, werden während des Programmablaufs Informationen über die Konvertierung ausgegeben.

### 6.2.2.2 Declarations

Datei `decl.c`

- `struct CFG_Conn *process_declarationlist(List list)` deklariert eine Liste von Objekten. Dies können Variablen, Signale oder auch weitere Programmblöcke sein, die wiederum aus Deklaration und Anweisungen bestehen. Daher wird ein **PSFCdfg** zurückgegeben.

### 6.2.2.3 Statements

Datei `stmt.c`

In diesem Modul werden die verschiedenen VHDL-Anweisungen behandelt. Dazu gibt es zunächst die beiden Funktionen:

- `CDFG_Conn *process_block(Node block)`: hier wird ein Block, bestehend aus Deklarationen und Anweisungen, in einen **PSFCdfg** umgewandelt.
- `CDFG_Conn *process_statementlist(List list)`: konvertiert eine Folge von Anweisungen in einen **PSFCdfg**. Dabei werden Basic Blocks erkannt und nach der unten beschriebenen Methode erzeugt. Für alle Befehle wird jeweils eine eigene Behandlungsroutine aufgerufen.

## 6.2 Implementierung des Konverters

In diesem Kapitel soll auf die programmtechnische Realisierung des Konverters eingegangen werden. Dabei wird zunächst das Konzept erläutert und anschließend die verschiedenen Module mit ihren Schnittstellen vorgestellt.

### 6.2.1 Konzept

Das VHDL-Quellprogramm wird zunächst mit dem im CLSI-Paket enthaltenen Compiler übersetzt und dabei im DLS (*Design Library System*) abgelegt. Das Programm wird im DLS-eigenen Format abgelegt, auf das mit Hilfe des SPI (*Software Procedural Interface*), welches als Library realisiert ist, zugegriffen werden kann. Das DLS-Format benutzt zwar einen objektorientierten Ansatz, ist aber in der Sprache C implementiert.

Die Arbeitsweise des Konverters wird durch das Datenformat des DLS-Systems bestimmt. Da der Graph nicht direkt im PSF erzeugt werden kann, werden alle Knoten und **PSFCsel**s durchnummeriert. Dies erleichtert auch z.B. das Wiedererkennen bereits bearbeiteter Elemente durch das Abspeichern dieser Nummer in ein Feld `qToolInfo`, welches in jedem DLS-Element vorhanden ist.

Ein wichtiger Datentyp ist die Struktur `CDFG_Conn`, welche einen in sich abgeschlossenen **PSFCdfg** beschreibt. Diese Struktur enthält jeweils eine Liste von Knoten für die einlaufenden Kanten eines **PSFCfgs** und **PSFDfgs** und die Knoten, an welche auslaufenden Kanten gehängt werden können. Mit der Funktion `merge_conn(CDFG_Conn *cdfg, CDFG_Conn *addcdfg)` können zwei solcher Teil-CDFGs verschmolzen werden.

Hierauf aufbauend lassen sich nun einige Funktionen entwickeln, die jeweils einen Teil-CDFG erzeugen und zurückgeben. Dies sind:

- `CDFG_Conn *process_block(Node block)` setzt einen Block in einen **PSFCdfg** um; ein Block enthält Deklarationen und eine Anweisungsfolge.
- `CDFG_Conn *process_statementlist(List list)` setzt eine Anweisungsfolge (im DLS: `StatementList`) in einen **PSFCdfg** um.
- `CDFG_Conn *process_statement(Node stmt, StmtListData *sld)` liefert den **PSFCdfg** einer Anweisung zurück. Jedoch werden bei Zuweisungen wegen der Besonderheit der Basic Blocks diese nötigenfalls in den **PSFDfg** der letzten Zuweisung eingefügt. Informationen darüber stehen in der Struktur `StmtListData`. Bei Befehlen, wie `if` oder Schleifen wird dann zur Erzeugung des **PSFCdfg**s für den Schleifenrumpf wieder `CDFG_Conn *process_statementlist(List list)` benutzt.

## 6 VHDL-Front-End

Dieser Konverter dient dazu, in VHDL erstellte Programme in den **PSFCdfg** zu übersetzen. Dabei wird die VHDL-Entwicklungsumgebung CLSI benutzt und auf das VHDL-Programm über das Software Procedural Interface, SPI des CLSI-Systems zugegriffen. In diesem Kapitel werden die unterstützten VHDL-Sprachelemente sowie die Implementation erläutert. Dabei wird besonders auf die Teile des Programms eingegangen, welche zur Erweiterung des Sprachumfangs notwendig sind.

### 6.1 Unterstützte VHDL-Sprachelemente

#### 6.1.1 Befehle

- Zuweisungen Variablen (`:=`, nicht in der Deklaration !) Signale (`<=`)
- nebenläufige Prozesse `process`
- Schleifen `while ... loop`, `for ... loop`
- Bedingte Verzweigungen `if ... then ... elsif ... else`

#### 6.1.2 Datentypen

- Numerisch `Integer`, `Natural`, `Bit`, `Bit_Vector`

#### 6.1.3 Operatoren

Es werden sämtliche Operatoren unterstützt, ebenfalls sind beliebige Klammerungen möglich.

- Arraydeklaration: Es ist aufgrund des niedrigen Niveaus der RTL nur unter erheblichem Aufwand möglich z.B. den Beginn und das entsprechende Ende eines Datenfeldes wiederzufinden. Das macht Konvention aus Abbildung 36 bei der Verwendung von Felder nötig.

```
#include "../PSF/BASE/INC/PSF/RTL_C_Constraints.H"

int a[10];

a[0] = ARRAY_BEGIN;
a[9] = ARRAY_END;
```

Abb. 36: Initialisierung von arrays

In dem Modul `RTL_C_Constraints.H` sind die Werte für `ARRAY_BEGIN`, sowie `ARRAY_END` definiert. Diese werden von dem Front-End später zur leichten Feldidentifikation verwendet und einem speziellen Konstrukt des **PSFdfg** verwaltet (siehe Abschnitt 2.2.2).

### 5.3.3 Einlesen einer C/C++ Spezifikation in das PSF

Zur Erzeugung eines RTL-Dumps zu einem C/C++ Sourcecode wird innerhalb der Funktion UNIX-Systemfunktionen verwendet, welche für den Benutzer nicht sichtbar ist. Diese Funktionen aktivieren den GNUCC zur Erstellung des RTL-Dump-Files. Nachdem das File erzeugt wurde, wird es durch den Konverter in den Control-Data-Flow-Graph abgebildet. Danach ist das RTL-Dump-File nicht mehr erforderlich und wird daher durch UNIX-Systemaufrufe gelöscht. Für den Benutzer erscheint dies als ein Vorgang, ihm bleibt die Benutzung des GNUCC verborgen.

Diese beide Punkte sind die einzigen Anforderungen, die neben denen aus Abschnitt 5.3.2 zu beachten sind.

Zur Verwendung der Front-Ends stehen Libraries zur Verfügung, welche die gesamte Funktionalität des Front-Ends beinhaltet. Die Funktion, welche den Transfer einer C/C++ Spezifikation in den **PSFCdfg** leistet, steht in dem Modul `RTLConverter.C`. Die Funktion liest eine Datei, mit der Extension ".C" als Input und hat als Output einen Verweis auf den erzeugten **PSFCdfg**.

Im letzten Abschnitt dieses Kapitels soll erläutert werden, wie das C/C++-Front-End zu verwenden ist.

## 5.3 Verwendung des C/C++-Front-End

Dieser Abschnitt beschreibt die Verwendung des Front-Ends für das PSF. Die hohe Funktionalität des Front-Ends besteht darin, daß theoretisch alle Konstrukte der Sprache C/C++ in das PSF abgebildet werden können. Dazu sind allerdings Beschränkungen bei der Implementation einer C/C++ Spezifikation vorzunehmen, um eine einwandfreie Konvertierung einer C/C++ Spezifikation zu gewährleisten. Grund dafür ist, daß der Aufwand für die Implementation des Front-Ends gering gehalten werden soll. Ziel des Front-Ends ist es, die Funktionalität einer Spezifikation in das PSF zu transferieren. Es geht bei der Implementation des Front-End nicht darum alle Implementationsmöglichkeiten einer Programmiersprache abzufangen. Dieser Aufwand ist unnötig. Spezifikationen, können bei gleichbleibender Funktionalität umgeschrieben werden. Dies rechtfertigt es, Beschränkungen für die Spezifikations-syntax eines C/C++ Programms vorzunehmen (siehe Abschnitt 5.3.2).

### 5.3.1 Zustand des Konverter im Dezember 1993:

Der Konverter verarbeitet folgende C Konstrukte: einfache hierarchische Programme mit Funktionsaufrufen, mit `integer (array)`-Variablen sowie `while`-Schleifen und `if-then-else` Konstrukte.

### 5.3.2 Beschränkung bei der Spezifikation von C/C++

Nun sollen einige Beschränkungen bei der Spezifikation durch C/C++ Programm dargelegt werden, um das Front-End einsetzen zu können.

- **Variablendeklaration:** C++ erlaubt Variablen an einer beliebigen Stelle des Sourcecodes einzuführen. Innerhalb des RTL hat das keine Bedeutung. Das Front-End hingegen verlangt, daß Variablen zu Beginn einer Funktion deklariert werden. Dies erleichtert den Implementationsaufwand des Front-Ends enorm, wogegen die Funktionalität des Sourcecodes nicht eingeschränkt ist.

Um nun aufzuzeigen, wie mit diesen Elementen der **PSFCdfg** aufgebaut wird, soll die Verwendung in der folgenden Abbildung dargestellt werden. Als RTL-Dump, ist das aus Abbildung 33 zu nehmen.

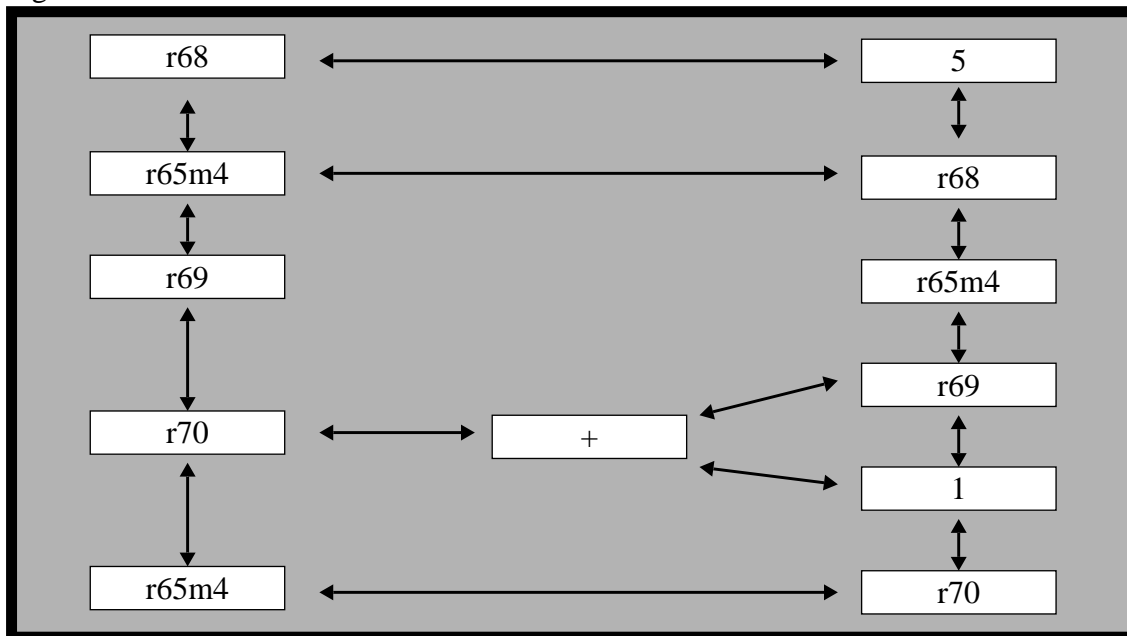


Abb. 35: **RTLAssignElem** Verwendung

In der Abbildung haben die Elemente und Pfeile folgende Bedeutung. Die linke Seite ist eine doppelt verkettete Liste von **RTLAssignElem**-Objekten. Diese Objekte werden in einer RTLDump-Instruktion beschrieben. Die Namen, die innerhalb der Rahmen stehen, ergeben sich aus den Instruktionen aus Abbildung 33. Die rechte Seite ist ebenfalls eine doppelt verkettete Liste, die die lesende Register und Konstanten aller RTL-Dump Instruktion enthält. Die Verweise von der linken in die rechte Liste und umgekehrt, bilden dann eine Instruktion. So wird z.B. in der `insn 11` dem Register 69 der Konstantenwert 5 zugewiesen. Dies findet sich in der obigen Abbildung als erste Elemente wieder. Dieser Verweis wird in das Datenelement `ptr_elem_ass` eines **RTLAssignElem**-Objekts eingetragen. In der Instruktion `insn 16` wird eine Addition definiert. Diese findet sich durch ein Operatorelement zwischen den Listen wieder. Dieses **RTLAssignElem**-Objekt wird ebenfalls über das Datenelement `ptr_elem_ass` durch die entsprechenden Elemente beider Liste referenziert.

Der Algorithmus zur Erzeugung eines **PSFDFg** zu einem Basic-Block funktioniert nun folgendermaßen: beginnend mit dem Anfang der linken Liste, wird diese durchsucht, bis ein Register mit dem Attribut `mem` gefunden wird. Dieses Register bezeichnet eine Variable des Sourcecodes. Dann wird die Zuweisungskette über beide Listen solange zurückverfolgt, bis entweder eine Konstante oder ein weiteres Register auf der rechten Liste gefunden wurde. Wenn das der Fall war, wurde eine gesamte Anweisung des Sourcecodes in den Data-Flow-Graphen übersetzt. Der Algorithmus, der dieses leistet, ist rekursiv implementiert.

gültigen Graphen eines Basic-Block zu erzeugen. Die Definition der Klasse ist in Abbildung 34 dargestellt.

```

class RTLAssignElem {

private:

    int insn_nr;
    PSFCSELID idtype;
    Reg_info ptr_reg;
    Const_info ptr_const;
    Op_info ptr_optor;
    Pc_info ptr_pc;
    Func_info ptr_func;
    LNODE ptr_lnode;
    RTLAssignElem *ptr_ass_elem;
    LNODE dfgbegin;
    LNODE dfgend;
    RTLBasicblock * basicblock;

}

```

Abb. 34: C++ Definition **RTLAssignElem**

Die einzelnen Members der Klasse **RTLAssignElem** haben folgende Bedeutung.

`int insn_nr`: Die Instruktionsnummer des RTLDumps, welche durch dieses Element dargestellt wird.

`PSFCSELID idtype`: Typ des Element

`Reg_info ptr_reg`: Information zu dem Register, wenn ein Register in dieser Anweisung beschrieben oder gelesen wird.

`Const_info ptr_const`: Information zu der Konstanten, die in dieser RTL Instruktion gelesen wird.

`Op_info ptr_optor`: Information über den Operator der Anweisung.

`Pc_info ptr_pc`: Informationen über den Programm Counter bei unbedingten Sprüngen.

`Func_info ptr_func`: Information über die Schnittstellenparameter einer Funktion.

`LNODE ptr_lnode`: Verweis zu dem LNODE, wenn ein Knoten zu dem Element erzeugt wird.

`RTLAssignElem *ptr_ass_elem`: Zuweisungselemente einer Anweisung

`LNODE dfgbegin`: Verweis zu dem LNODE des PSFDFgBBBeginNode.

`LNODE dfgend`: Verweis zu dem LNODE des PSFDFgBBBeginEndNode.

`RTLBasicblock * basicblock`: Verweis zum Basic-Block, zu dem dieses Element gehört.

Das Front-End ist dreiteilig realisiert.

1. Das RTL-Dump wird zeichenweise in Strukturen des GNUCC eingelesen. Dies ermöglicht, daß implementierte Funktionen des GNUCC verwendet werden können, ohne einen neuen Parser zu codieren, der die RTL-Dump Struktur interpretiert.
2. Die Strukturen des RTL auf eine neue Struktur von **RTLAssignElem** abgebildet (siehe Abschnitt 5.2.1), welche die relevante Information zur Erzeugung des **PSFDfg** aufnehmen. Es handelt sich dabei um zwei Listen, die auf der einen Seite die zu beschreibende Elemente, auf der anderen Seite die zu lesenden Elemente einer Anweisung der RTL-Dumps beinhalten. Während diese Struktur aufgebaut wird, erfolgt eine gleichzeitige Analyse des Kontrollflusses innerhalb der RTL-Dump Struktur. Ein RTL-Dump besteht weiterhin aus konditionalen und unkonditionalen Sprüngen, sowie Basic-Blocks. Daher wird während des Aufbaus der Beginn und das Ende eines Basic-Block, sowie dessen Art (Basic-Block, Ausdruck für die Berechnung einer eventuellen Verzweigung) erkannt. Ist ein solcher Basic-Block erkannt, wird der **PSFDfg** für diesen Basic-Block aus der **RTLAssignElem** Struktur erzeugt und Speicherplatz freigegeben. Ebenfalls wird der entsprechende **PSFCfgNode** erzeugt. Die Verweise zwischen dem Basic-Block-Dfg und dem **PSFCfgNode** werden gesetzt. Es wird weiterhin die Information der Sprünge gespeichert, um im dritten Schritt den komplette **PSFCfg** zu erstellen.
3. Alle Data-Flow-Graphen der Basic-Blocks sind erzeugt. Es ist nun erforderlich, die Knoten des Control-Flow-Graphen, sowie die Basic-Blocks untereinander zu verbinden. Im vorherigen Schritt wurde die Information über die Sprünge gesammelt und nun nur noch ausgelesen um die Kanten in den jeweiligen Graphen zu setzen. Somit wird in dieser Phase der komplette **PSFCfg** und **PSFDfg** bzw. der **PSFCdfg** erstellt.

Es erscheint der Aufbau des Data-Flow-Graph, der komplexere Teil des Front-Ends zu sein. Daher soll der folgende Abschnitt eine kurze Einführung in die Vorgehensweise geben.

### 5.2.1 Zwischenstruktur - **RTLAssignElem**

Diese Elemente nehmen die für die Erstellung des **PSFDfg** notwendigen Informationen des RTL-Dumps auf. Auf diesen Strukturen sind dann Algorithmen implementiert, die den end-

struktionen `insn 11` und `insn 12` nachgebildet. Dabei fällt auf, daß das Register mit der Nummer 68, vom Typ `SI` ist. `SI` bezeichnet `integer`-Werte des Sourcecodes. Durch andere Attributierung werden verschiedene datenbreiten eines Registers definiert. Diesem Register 68 wird der Konstantenwert 5 zugewiesen. In der darauffolgenden Anweisung erhält das Register mit der Nummer 65, mit dem negativen Offset 4 und dem Attribut `mem`, den Wert des Registers 68, also den Konstantenwert 5. Dieses Attribut `mem` bezeichnet immer eine Speicherstelle, die mit dem Register 65 und entsprechendem Offset angesprochen werden kann. Eine solche Stelle enthält den Wert einer Variablen des Sourcecodes (`int a`) und wird im folgenden mit dem Identifikator `r65m4` bezeichnet.

- das “m” steht für das Attribut `mem`,
- `r65` für das Register 65 und
- 4 für den negativen Offset.

Die zweite Anweisung des Sourcecodes enthält zusätzlich einen Operator. Der Wert, der in der vorherigen Anweisung auf die Speicherstelle `r65m4` geschrieben wurde, wird in der ersten RTL-Instruktion (`insn 15`) auf das Register 69 zugewiesen. Die Anweisung (`insn 16`) addiert mit dem Operator `plus` den Wert des Register 69 und die Konstante 1 und schreibt das Ergebnis in das Register 70. Dies wird dann erneut an die Speicherstelle `r65m4` geschrieben (`insn 18`).

Man erkennt folgende Eigenschaften des RTL-Dump:

- Eine Variable des Sourcecode belegt eine Speicherstelle, die mit einem speziellen Register mit bestimmten Offset erreicht werden kann: (`int a -> r65m4`)
- Die Register ohne das Attribut `mem` haben keine direkte Korrespondenz zu Variablen des Sourcecodes. Sie bilden über Instruktionsketten die Anweisung des Sourcecodes nach: (`int a = 5; -> insn 11, insn 12`)
- Operatoren finden sich im RTL-Dump wieder: (`insn 16`)
- Der Kontrollfluß ergibt sich aus der Analyse der Sprunginstruktionen. In dem Beispiel aus Abbildung 33 befinden sich keine Sprünge, die für die Erzeugung des Control-Flow-Graphs von Bedeutung sind. Es handelt sich nur um einen Basic-Block.

## 5.2 Technische Realisierung des C/C++ Front-Ends

Ziel des Front-Ends ist es, den gesamten Sprachschatz von C/C++ für die Synthese verfügbar zu machen. Als Eingabe soll eine Spezifikation in C/C++ innerhalb einer Datei sein, die Ausgabe des Front-Ends ein Verweis auf den durch die Konvertierung erzeugten Control-Data-Flow-Graph der Spezifikation.

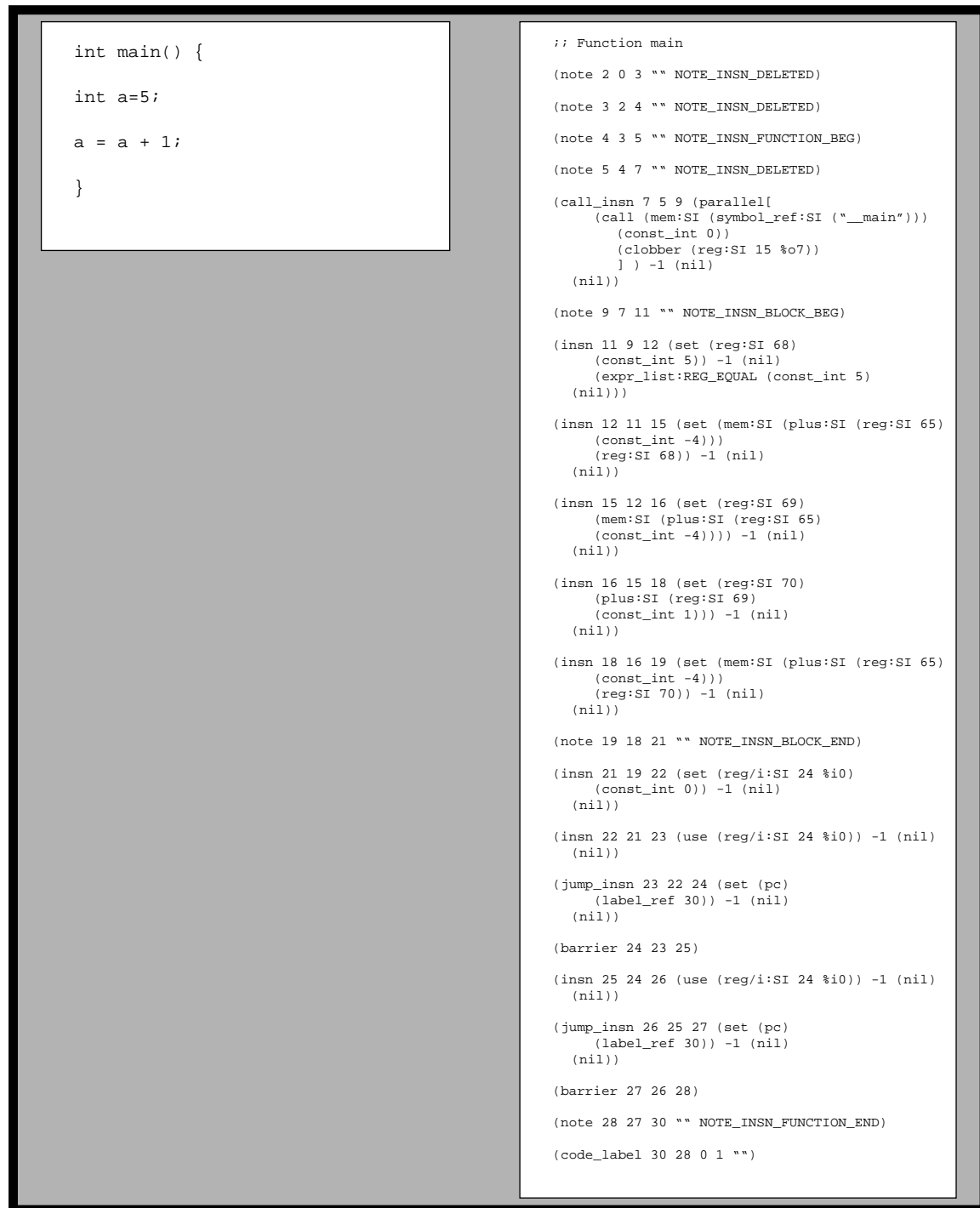


Abb. 33: RTL-Dump für C Code

Die obige Abbildung zeigt auf der linken Seite den C Code einer simplen Funktion und das dazugehörige RTL-Dump. Der Transfer geschieht nach folgendem Prinzip. Die Anweisungen, die mit `insn` beginnen, bilden den Datenfluß. Nach der Instruktion (`note 9`), beginnt der Basic-Block der Funktion. Dabei wird erste Zuweisung des Sourcecodes, durch die In-

## 5.1 Register-Transfer-Language RTL

Dieser Abschnitt beschreibt die Struktur der RTL. Der Übersetzungsprozeß des GNUCC bildet einen C/C++-Sourcecode und seine die “hohen” Strukturen der Programmiersprache auf Register-Transfer Anweisungen ab. Man kann die RTL, als Assembleranweisungen einer virtuellen Maschine ansehen. Die Sprache besteht im wesentlichen aus komplexen Strukturen, die je eine Anweisung der virtuellen Maschine darstellen.

Ein RTLDump existiert im aus fünf Typen von Anweisungen:

1. `note` - Instruktionen: Diese Instruktionen sind Kommentare, die für die Interpretation von Positionen innerhalb der Anweisungen verarbeitet werden. Sie zeigen z.B. den Beginn eines Basic-Blocks an.
2. `call_insn` - Instruktionen: Eine solche Anweisung ist für die Verarbeitung von Funktionsaufrufen relevant.
3. `jump_insn` - Instruktionen: Diese Anweisungen, sind für den Kontrollfluß des Programms verantwortlich. Es handelt sich dabei um konditionale, sowie unbedingte Sprünge innerhalb des RTL. Durch Analyse der Sprungziele wird festgestellt, ob es sich um Schleifen, einfache Verzweigungen, etc. handelt. Die Analyse ist zur Erstellung des **PSFCfg** nötig.
4. `insn` - Instruktionen: Die Instruktionen bilden auf feinem granularem Niveau den Datenfluß des Sourcecodes nach. Es sind Zuweisungen zwischen den Registern der virtuellen Maschine. Diese Instruktionen, sind essentiell zur Erstellung des **PSFDfg** (siehe Abschnitt 5.2.1).
5. `barrier`: Die Instruktionen bilden Grenzen zwischen Basic-Blocks. Desweiteren bilden sie Sprungziele für die Sprunginstruktionen.

Um nicht zu detailliert das Format der RTL zu beschreiben, ist am deutlichsten, ein einfaches Beispiel wie in Abbildung 33 anzugeben. Anhand des Beispiels werden ein paar grundlegende Eigenschaften eines RTL-Dumps deutlich.

mantischen Analyse. Dieses Prinzip verwendet der GNU CC für verschiedene Eingabesprachen.

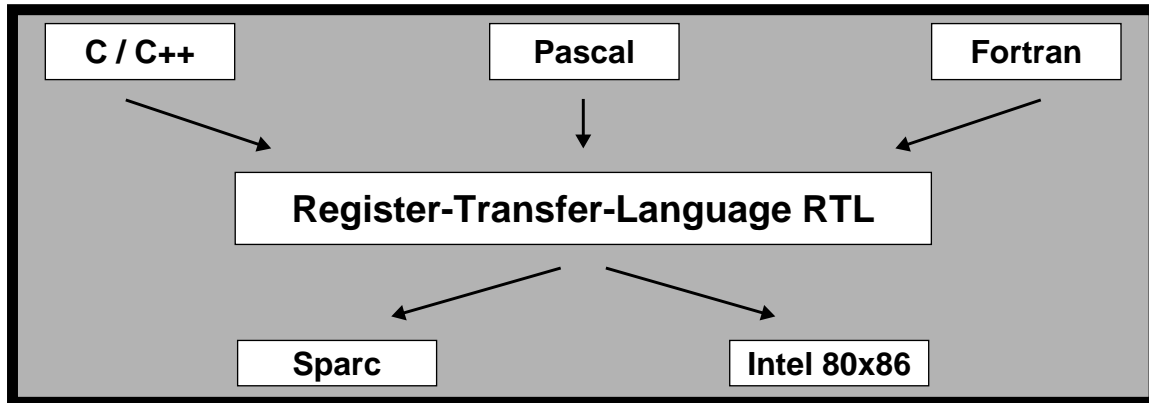


Abb. 32: GNU CC

Die Register Transfer Language bildet die Schnittstelle für die Code-Generierung des GNUCC. Ein Programmkonstrukt einer beliebigen, höheren Programmiersprache wird in Register Transfer Strukturen abgebildet.

Dies bietet eine Möglichkeit, C/C++ in das PSF zu übersetzen. Der verwendete Weg zur Konvertierung von C/C++ in das PSF geht nach folgendem Prinzip vor:

1. Ein Programm wird in C/C++ codiert.
2. Durch Aufruf des GNUCC wird das codierte C/C++ Programm auf seine Syntax und Semantik überprüft und in die GNU RTL abgebildet. Durch GNUCC-Optionen wird ein Dump-File der RTL-Anweisungen erzeugt.
3. Das Front-End des PSF nimmt dieses Dump-File als Input und bildet es auf den **PSF-Cdfg** ab.

Durch diesen Prozeß ist es möglich, ein durch C/C++ spezifiziertes System durch den Syntheseprozess des PSF zu verarbeiten. Der Transfer von Strukturen der Eingabespezifikation in die Strukturen des **PSFCdfg** wurde in Kapitel 2 beschrieben.

Dieses Kapitel ist wie folgt strukturiert:

1. Die Register Transfer Language wird beschrieben.
2. Die Realisierung des Front-Ends wird beschrieben.
3. Die Verwendung des C/C++-Front-Ends wird beschrieben.

## 5 C/C++-Front-End

Das PSF bietet die Möglichkeit, Spezifikationen aus verschiedenen Eingabesprachen zu synthetisieren. Um die Anwendung des Syntheseprozesses auf Beschreibungen von großer Allgemeingültigkeit zu haben, wurde als Systemspezifikationsprache C bzw. seine objektorientierte Erweiterung C++ gewählt. Die High-Level-Synthese von in C/C++ geschriebenen Programmen, machte die Implementation eines Front-Ends zur Konvertierung einer Spezifikation in das PSF erforderlich.

Es war Ziel, den enormen Sprachschatz der Sprache C++ für die Synthese uneingeschränkt zu lassen, denn dies erhöht die allgemeine Verwendbarkeit. Es stand zur Wahl einen Parser, bzw. einen kompletten Übersetzer mit Zielstruktur **PSFCdfg** zu implementieren, oder sich dabei der Verwendung von Public Domain Software zu behelfen. Dabei bietet sich ein häufig verwendetes Tool an, da dies die Zuverlässigkeit des Werkzeugs erhöht. Der am häufigsten verwendete C/C++-Compiler ist der Public Domain Compiler GNU CC.

Die eingeschlagene Vorgehensweise bedient sich des GNU CC und dessen Zwischensprache, Register-Transfer-Language, RTL. Jeder Compiler, der für die Code Erzeugung verschiedener Rechner implementiert wurde, besitzt eine Zwischensprache, auf die der Compileprozeß nach der semantischen Analyse abbildet. Von dieser Zwischensprache ausgehend, wird dann für die entsprechende Zielmaschine, die Code-Generierung vorgenommen. Dies erlaubt für verschiedene Maschinen, nur eine Implementation der lexikalischen, syntaktischen und se-

Vom Generator werden die Bits des zu shiftenden Wertes über Demultiplexer an die entsprechenden Ausgabestellen verteilt und dort über OR-Gatter zusammengefaßt.

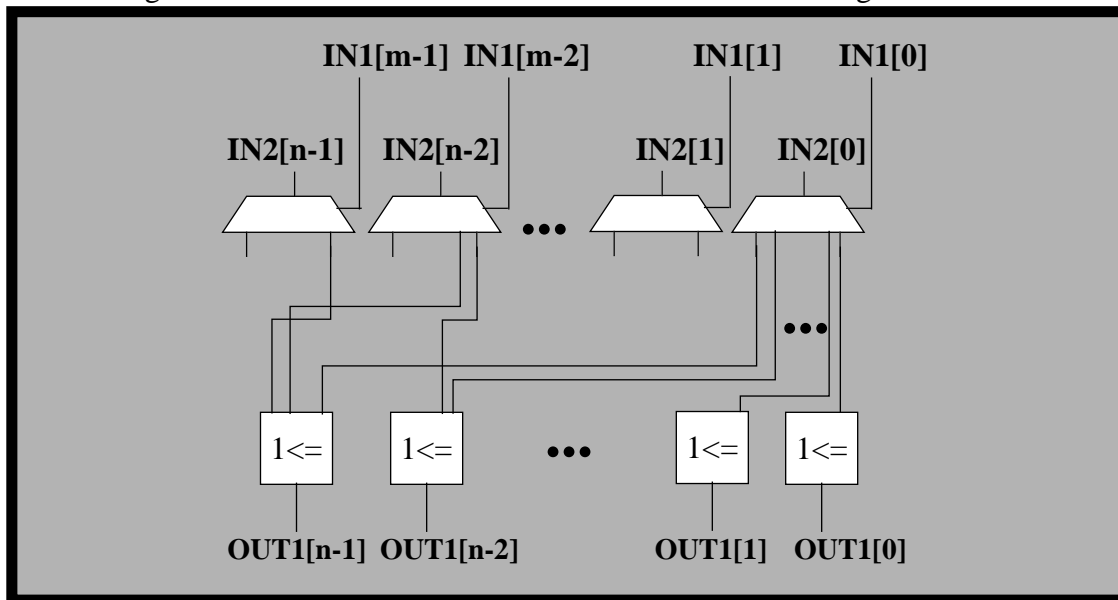


Abb. 31: Realisierung des (logischen) Shift-Left-Operators **SHL**.

#### 4.2.4.6 SHR

Analog dem **SHL**-Operator ist der **SHR**-Operator als dyadischer, logischer Shift-Operator implementiert.

## 4.2.4 Logische Elemente

### 4.2.4.1 AND

Es wird eine Menge von Gattern ausgegeben, die eine bitweises logisches UND der entsprechenden Eingänge realisieren. Dazu ist es erforderlich, daß die Bitbreiten der beiden Eingänge übereinstimmen. Entspricht dem C-Operator `&`.

### 4.2.4.2 OR

Es wird eine Menge von Gattern ausgegeben, die eine bitweises logisches ODER (siehe AND) der entsprechenden Eingänge realisieren. Dazu ist es erforderlich, daß die Bitbreiten der beiden Eingänge übereinstimmen. Die Bitbreite des Ausgangs hat ebenfalls mit der Bitbreite des Einganges übereinzustimmen. Entspricht dem C-Operator `|`.

### 4.2.4.3 XOR

Realisiert das bitweise exklusive ODER der beiden Eingänge. Beide Eingänge und der Ausgang müssen die gleiche Bitbreite haben. Der Generator legt für jeweils zwei an der gleichen Position befindliche Bits ein XOR-Gatter an.

### 4.2.4.4 NOT

Bestimmt das bitweise, logische NICHT des Einganges. Es ist darauf zu achten, daß die Bitbreiten des Einganges und des Ausgangs, wie bei allen monadischen Operationen übereinstimmen. Der Generator erzeugt für jedes Eingangs-Bit einen Inverter.

### 4.2.4.5 SHL

Der (logische) Shift-Left-Operator ist entsprechend der C-Semantik als dyadischer Operator implementiert. Eingaben sind der zu shiftende Wert und die Anzahl der zu shiftenden Stellen.

#### **4.2.3.1 EQ**

Realisiert den Vergleichsoperator  $==$ . Die Realisierung des Vergleichers erfolgt durch Bitweise Berechnung der XNOR-Funktion. Die Ergebnisse werden dann UND-verknüpft. Die Ausgabe des UND-Gatters ist dann die Ausgabe des Vergleichers.

#### **4.2.3.2 NE**

Realisiert den Vergleichsoperator  $!=$ . Die Generatorfunktion ist mit der der **EQ**-Komponente bis auf den Inverter am Ausgang identisch.

#### **4.2.3.3 LE**

Realisiert den C-Vergleichsoperator  $<=$ . Noch nicht implementiert.

#### **4.2.3.4 LT**

Realisiert den C-Vergleichsoperator  $<$ . Noch nicht implementiert.

#### **4.2.3.5 GE**

Realisiert den C-Vergleichsoperator  $>=$ . Noch nicht implementiert.

#### **4.2.3.6 GT**

Realisiert den C-Vergleichsoperator  $>$ . Noch nicht implementiert.

#### **4.2.3.7 RiCompare**

Hilfsgenerator.

#### **4.2.3.8 RiLE**

Hilfsgenerator.

#### **4.2.3.9 RiLT**

Hilfsgenerator.

#### **4.2.3.10 RiGE**

Hilfsgenerator.

#### **4.2.3.11 RiGT**

Hilfsgenerator.

Verbindung zusammengeschlossen, so ist es erforderlich, ein Element, welches die beiden Leitungen verschmilzt, zu instantiiieren. Der entsprechende Blif-Generator erzeugt einen Treiber zwischen einem Eingang und dem entsprechenden Ausgang des Elementes. Die Matchfunktion überprüft, daß die Summe  $m + n$  der Bitbreiten der beiden Eingänge der Bitbreite des Ausganges entsprechen.

#### 4.2.2.8 SPLIT

Dieses Element erlaubt die Aufspaltung der Verbindung (Wire) in zwei Verbindungen, deren Summe der Bitbreiten  $m + n$  der Bitbreite des Einganges entspricht.

#### 4.2.2.9 SWAP

Diese Komponente erlaubt das Vertauschen zweier Verbindungen mit unterschiedlichen Bitbreiten. Die Matchfunktion überprüft, daß die Bitbreite des ersten Eingangs mit der Bitbreite des zweiten Ausgangs und die des zweiten Eingangs mit der des ersten Ausgangs übereinstimmt. Zur Vertauschung der entsprechenden Leitungen werden wiederum Treiber verwendet.

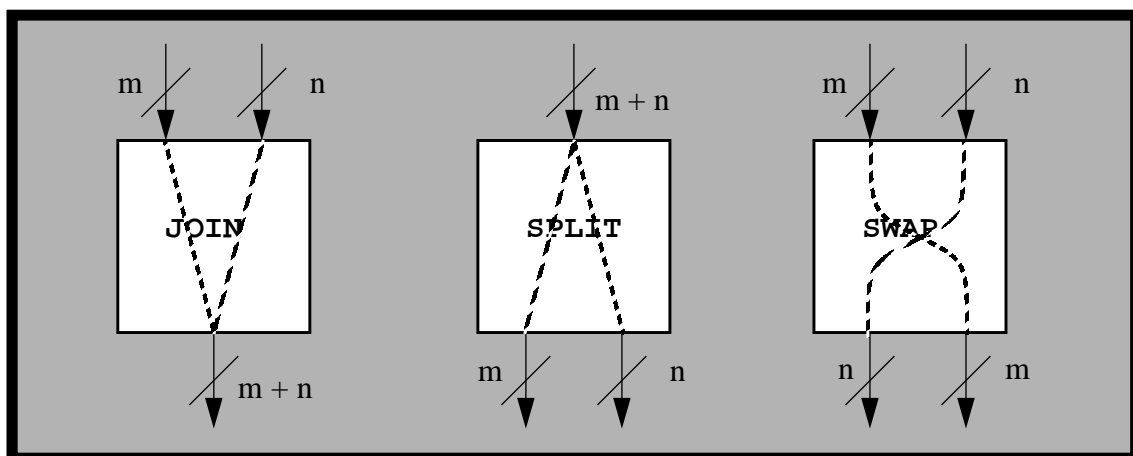


Abb. 30: Komponenten zum Verschmelzen, Aufspalten und Vertauschen von Leitungen durch **JOIN**, **SPLIT** und **SWAP**.

### 4.2.3 Komparatoren

Die folgenden Komponenten realisieren die C-Vergleichsoperatoren ( $==$ ,  $!=$ ,  $<$ ,  $<=$ ,  $>$ ,  $>=$ ).

Für alle Komparatoren gilt die Restriktion, daß die Bitbreiten der beiden Eingänge übereinstimmen müssen. Die Ausgänge haben Bitbreite 1.

### 4.2.2.3 MUX

Multiplexer sind generisch über die Anzahl  $n$  der Eingänge und deren Bitbreite. Der Generator realisiert also einen  $m$ -Bit  $n$ -zu-1 Multiplexer. Für jeden der  $m$  Kanäle wird ein Baum von  $\log n$  2-zu-1-Multiplexern aufgebaut.

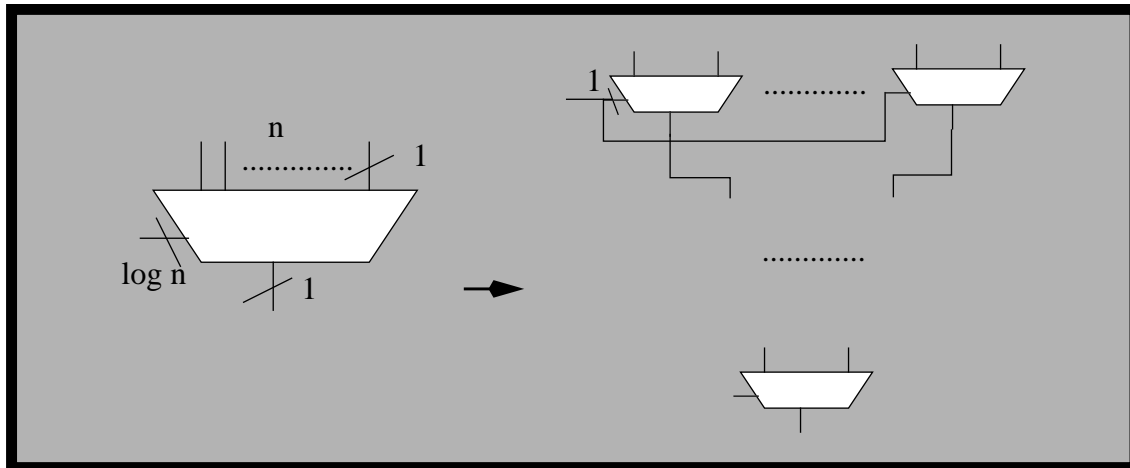


Abb. 29: Realisierung 1 Bit  $n$ -zu-1-Multiplexers.

### 4.2.2.4 DEMUX

Demultiplexer sind generisch in der Anzahl der Eingänge und deren Bitbreite. Der Generator baut analog zur Implementation des Multiplexers einen Baum von 1-zu-2-Demultiplexern auf, d.h. Vergabe von Namen für die Darstellung von internen Signalen ist im wesentlichen gleich.

### 4.2.2.5 CONST

Konstanten-Speicherung und Ausgabe.

### 4.2.2.6 REG

Beschreibt ein Register mit Load-Enable-Eingang (dies ist der erste Input). Die Match-Funktion überprüft, ob Eingang und Ausgang die gleiche Bitbreite besitzen. Der Load-Enable-Eingang muß Bitbreite 1 haben. Der Generator erzeugt einen  $n$  Bit 2-zu-1 Multiplexer, dessen Steuerleitung an das Load-Enable-Signal und dessen Eingänge durch den Eingang des Registers und die Ausgänge einer Bank von  $n$  Flip-Flops gebildet werden. Das Bündel von Ausgangsleitungen der Flip-Flops bildet gleichzeitig den Ausgang des Registers.

### 4.2.2.7 JOIN

Innerhalb der Netzliste ist möglich, Leitungen (**PSFWire**) einer Bitbreite zuzuordnen. Der Anschluß einer solchen Leitung an einen Pin (**PSFPin**) oder Port (**PSFPinInst**) erfordert, daß dieser die gleiche Bitbreite aufweist. Werden zwei Leitungen der Breite  $n$  und  $m$  zu einer

### 4.2.2.1 OneHot2Bin

Diese Funktion konvertiert einen one-hot-kodierten Bitstring in die entsprechende Binärdarstellung.

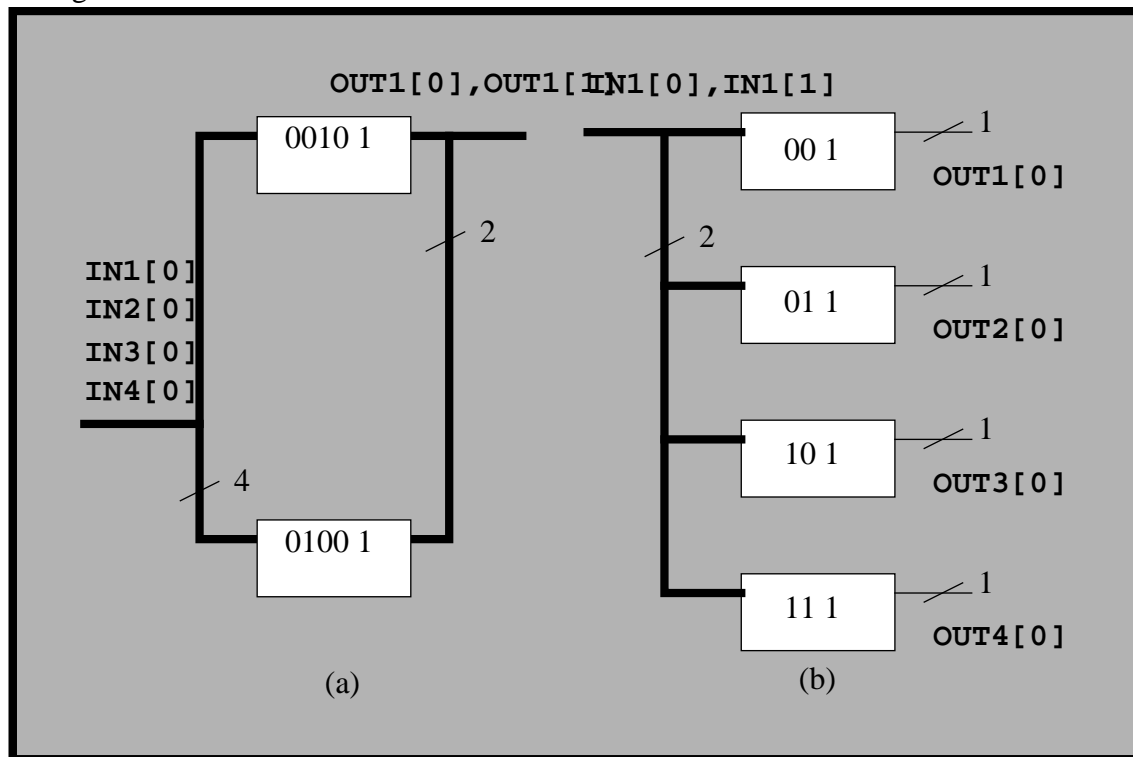


Abb. 28: Umkodierungsfunktionen: **OneHot2Bin** (a) und **Bin2OneHot** (b) werden als jeweils als Bündel von Tafeln dargestellt (Beispiele für Ausgangs- bzw. Eingangs-Bitbreite 2). Die Abbildung der Werte für das obige Beispiel erfolgt nun folgender Maßen: 00 (binär)  $\leftrightarrow$  0001 (1-aus-n), 01  $\leftrightarrow$  0010, 10  $\leftrightarrow$  0100 und 11  $\leftrightarrow$  1000.

### 4.2.2.2 Bin2OneHot

Diese Funktion konvertiert einen Binärstring in einen Onehot-kodierten 7String und stellt damit die Umkehrfunktion von **OneHot2Bin** dar.

#### 4.2.1.4 DIV

Dazu wird ein Feld von Divisionszellen  $D$  aufgebaut, in der jede Zelle einen Einbit-Dividierer darstellt.

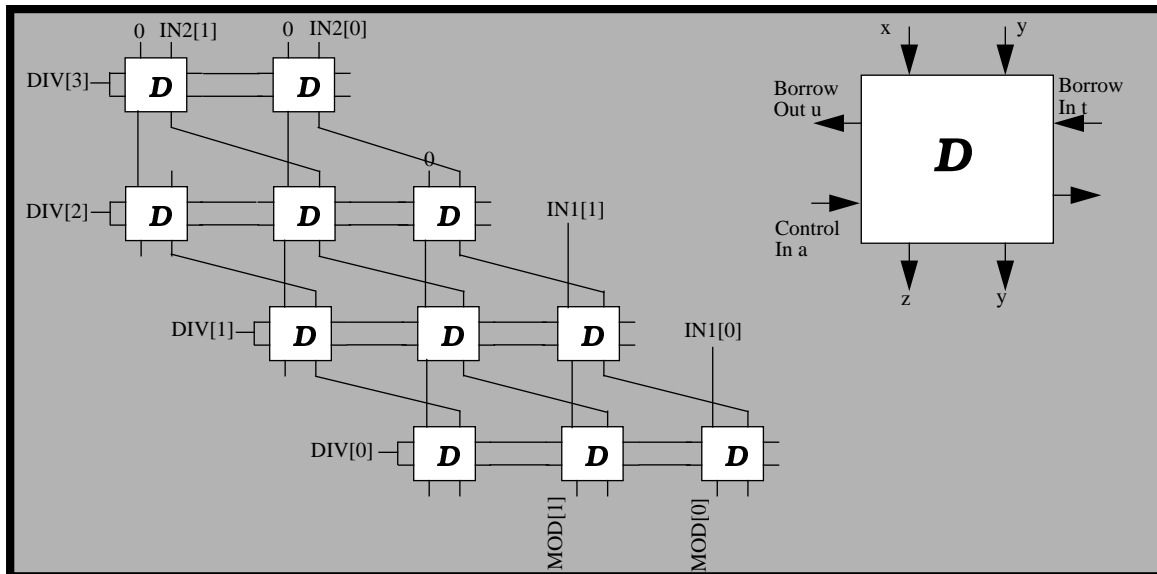


Abb. 27: Kombinatorische Divisionszelle (3 Bit) und Realisierung eines Divisions-Arrays.

In obiger Divisionszelle wird  $z = x - (y + t)$  (falls  $a = 0$ ) und  $z = x$  (sonst) gesetzt. Die Ausgänge werden daher durch die booleschen Gleichungen  $z = x \oplus \bar{a} (y \oplus t)$  und  $u = \bar{x} y + \bar{x} t + y t$  beschrieben. Als Disjunktion der Produkt-Terme (Elimination des Exor) ergibt sich:  $z = xa + x \bar{y} \bar{t} + xyt + \bar{x} y \bar{a} \bar{t} + \bar{x} y \bar{a} t$ . Für jede instanzierte Divisions-Zelle werden diese beiden Ausgänge als Gatter beschrieben.

#### 4.2.1.5 MOD

Wird wie der DIV-Operator implementiert. Statt der Betrachtung des Divisionsergebnisses wird nun aber der Rest der Integer-Division betrachtet und an die Ausgänge gegeben.

#### 4.2.1.6 ALU1

Noch nicht implementiert.

### 4.2.2 Kontroll Strukturen

Diese Elemente werden zur Herstellung von Verbindungen zwischen RT-Bausteinen eingesetzt.

Bei der Instanziierung einer Zelle aus der Hardware-Spezifikation eines Bibliotheks-Elementes ist die Benennung der Eingänge und Ausgänge wie folgt:

```
IN1[0], . . . . , IN1[<bitsize-1>], IN2[0], . . . . , OUT1[0], . . . .
```

Dies ist bei der Erweiterung der Library um neue Komponenten zu beachten. Im folgenden werden die einzelnen Bausteine nach der Funktion aufgelistet.

## 4.2.1 Arithmetische Komponenten

Hier werden Komponenten, die die von höheren Programmiersprachen her bekannten arithmetischen Operationen (+, -, ++, --, \*, /, %) entsprechen, nachgebildet.

### 4.2.1.1 PLUS

Realisiert den dyadischen C-Operator +, d.h. es wird davon ausgegangen, daß es zwei Inputs und einen Output gibt. Der Generator verwendet den Generator des unten beschriebenen Ripple-Carry-Addierer **RiPLUS**. Ein Übertrag wird weder ausgegeben noch als Eingang verwertet.

### 4.2.1.2 MINUS

Realisiert den dyadischen C-Operator -, d.h. es wird wiederum davon ausgegangen, daß es zwei Inputs und einen Output gibt und die Bitbreiten der Eingänge mit denen des Ausgangs übereinstimmen. Der Generator verwendet ebenfalls den Generator des unten beschriebenen Ripple-Carry-Addierer **RiPLUS**.

### 4.2.1.3 MULT

Noch nicht implementiert.

## 4.2 Beschreibung der Bibliothekselemente

In Bibliothekselementen werden statisch und generisch Daten verwaltet. In der untenstehenden Abbildung wird der Inhalt der Klassen dargestellt.

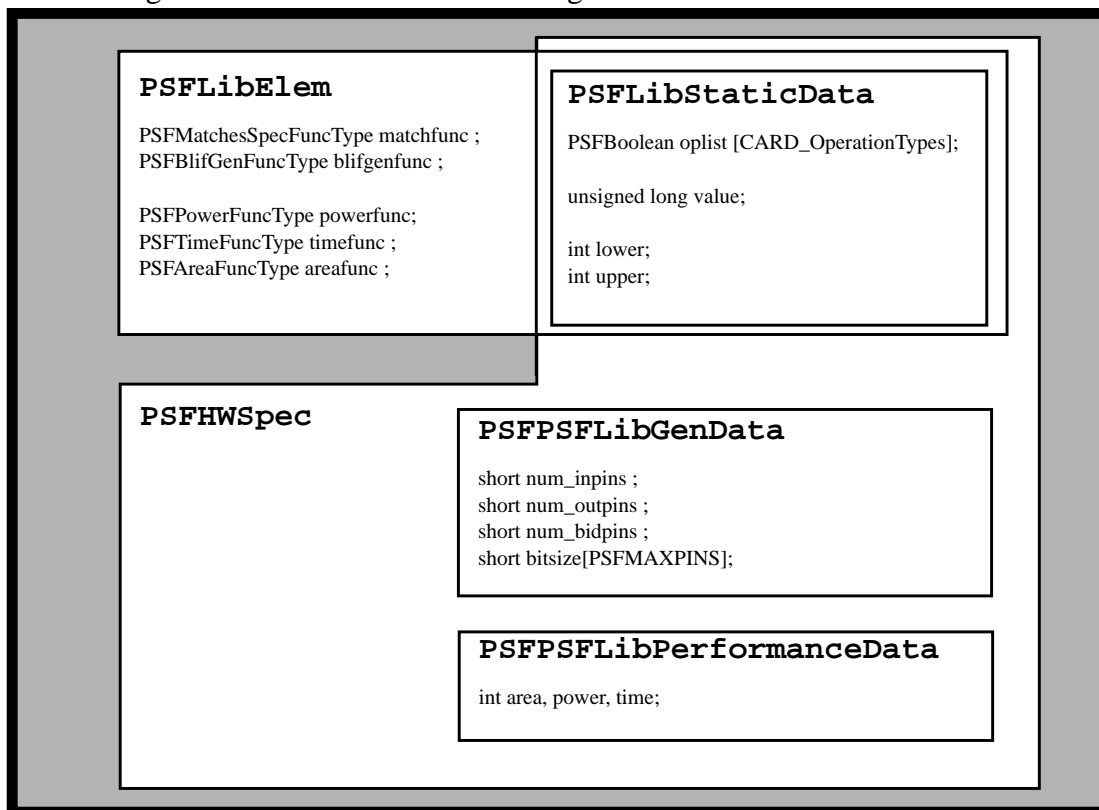


Abb. 26: Klassen für Bibliothekselemente.

Für jedes Element gibt es zwei wesentliche Funktionen:

1. Eine Funktion **matchfunc**, die bei Eingabe einer Hardware-Spezifikation **PSFHWSpec** überprüft, ob das Element die Spezifikation erfüllt oder nicht. Diese Funktion wird auf der Liste der Module in **findMatchingElems()** für jedes Bibliothekselement aufgerufen. So ist zum Beispiel für einen Multiplexer, der über die Anzahl der Eingänge und über deren Bitbreiten generisch ist, zu überprüfen, daß genügend Steuerleitungen spezifiziert wurden. Dies betrifft also den generischen Datenanteil.
2. Einen Blif-Generator, der die entsprechende Beschreibung im Blif-Format generiert. Hierbei werden die generischen Daten (Anzahl der Ein-/Ausgänge und deren Bitbreite berücksichtigt).

Daneben gibt es noch Funktionen, die Abhängigkeit der generischen Daten **PSFPSFLibGenData** einer Hardware-Spezifikation die Fläche, die Leistungsaufnahme und die Laufzeitverzögerung angeben.

## 4 PSFLib

Die Klasse **PSFLib** stellt die Modul-Bibliothek zur Datenpfad-Synthese zur Verfügung. Eine globale Variable dieses Typs ist in der Datei `net.C` instantiiert. Bei der Synthese und Ausgabe des synthetisierten Entwurfes, der als Strukturbeschreibung in Form einer Netzliste vorliegt, wird diese Variable benutzt, um Elemente zu matchen oder die entsprechenden Zell-Generatoren aufzurufen.

Die in der Modul-Bibliothek verfügbaren, d.h. in der Netzliste instanziiierbaren, Komponenten vom Typ **PSFLibElem** werden in einer Liste **modulelist** verwaltet. Bei der Ausgabe einer hierarchischen Netzliste war darauf zu achten, daß jedes Library-Element nur einmal ausgegeben wird. Dies wird durch zwei weitere Member-Variablen sichergestellt.

### 4.1 Beschreibung der Funktion der Klasse **PSFLib**

Die Ausgabe der Library wird durch die oben angesprochenen Member-Variablen gesteuert. Vor einer Ausgabe der Modul-Bibliotheks-Komponenten, ist die Ausgabe durch Aufruf der Methode **initBlifOut()** zu aktivieren. Entsprechend wird die Ausgabe durch Aufruf der Methode **exitBlifOut()** deaktiviert und der Ausgabe-Stack zurückgesetzt.



### 3.3.6 Probleme

Da im Moment noch nicht getestet wird, ob die Hierarchie konsistent ist - d.h. der Hierarchie-Baum in dem die Knoten Zellen und Kanten Instanziierungen von Zellen darstellen, ein DAG ist - kommt es beim Einlesen von Blif-Beschreibungen, beim Expandieren und beim Kollabieren von Zellinstanzen zu Problemen, wenn rekursive Referenzierungen vorhanden sind.

Speziell beim Einlesen von Blif-Beschreibungen kommt es zu Problemen, wenn Modellnamen mit intern vergebenen Namen übereinstimmen. Dies sind Namen der Form 'table\_cell*i*' bzw. 'latch\_cell*i*', die intern für Gatter bzw. Flipflops vergeben werden, wobei *i* eine natürliche Zahl ist.

Danach erfolgt dann das Verdrahten, daß hier nur für den Ausgang als Beispiel aufgeführt ist:

```
PSFPin *1st_input = design->addPin(1, "in1", 'I');
PSFPin *2nd_input = design->addPin(1, "in2", 'I');
PSFPin *output = design->addPin(1, "in2", 'O');
PSFPinOrInst *from = and->getOutPins().head();
PSFPinOrInst *to = output;
design->addWire (from, to, "Ausgang");
```

Die Methode `addPin()` benötigt als Parameter die Angaben über die Bitbreite (auch ein 32Bit breiter Ein-, oder Ausgang läßt sich auf diese Weise mit einem Aufruf bewerkstelligen), einen eindeutigen Namen und die vorgesehene Signalrichtung, d.h. ob dieser Pin als Eingang ('I'), als Ausgang ('O'), oder Bidirektional ('B') fungieren soll. die für dieses Beispiel gewählte Funktion `addWire()` ist auf das Anlegen eines Drahtes mit einer Quelle und einer Senke vorgesehen. Die Referenz auf das Objekt `PSFPinOrInst` erlaubt dabei die Vernachlässigung der Unterscheidung von einem Pin und einer Instanz eines Pins. Die Methode `addWire()` ist mehrfach überladen, so existieren auch Methoden, die z.B ein Anlegen eines Drahtes mit mehreren Senken erlauben.

In dem folgenden Beispiel soll die Anwendung der hierarchischen Funktionen `cut/collectCell()` erläutert werden. Gegeben sei das unter '**Hierarchische Operationen/Funktionen**' gegebene Design der "Vaterzelle", welches die Instanzen "latch0", "table0" und "table1" enthält. Die oben als Grafik verdeutlichte Aktion von `collectCell()` und `cutCell()` lassen sich dann wie folgt bewerkstelligen:

```
PSFCell *design // Referenz auf die "Vaterzelle" sei gegeben

PSFCellInstList ci_list; // Sammelnde Liste für Zellinstanzen
ci_list.append (design->getCellInst ("latch0");
ci_list.append (design->getCellInst ("table1");

// Herausschneiden von Zellinstanzen
PSFCell *new_cell = cutCell (*design, ci_list, "Neuzelle");

// alternativ: das Sammeln von Zellinstanzen
PSFCell *new_cell = new PSFCell ("Neuzelle");
PSFWireList interface = cutCell (*design, *new_cell, ci_list);
```

Zu bemerken ist, das die elementarere Funktion `collectCell()` eine bereits generierte, leere Zelle voraussetzt. Der Rückgabewert dieser Funktion ist eine Liste der Drähte, die in dem Design "Vaterzelle" mit den Zellinstanzen verbunden sind, die in "Neuzelle" gesammelt wurden. Dies hat sich aus der Notwendigkeit ergeben, `collectCell()` als Subfunktion in `cutCell()` verwenden zu können.

### 3.3.4 Abhängigkeiten

```

PSFCellBehaviour: PSFCell,PSFBlifType
PSFCell: PSFCellInst,PSFPinInst,PSFCellBehaviour
PSFCellInst: PSFCell,PSFPinInst
PSFPinOrInst: PSFPin,PSFPinInst
PSFPin: PSFWire,PSFCell
PSFPinInst: PSFCellInst,PSFPin,PSFWire,PSFCellInst
PSFWire: PSFCell,PSFPin,PSFPinInst

```

### 3.3.5 Benutzung der Klasse

Der Konstruktor wird durch Angabe der Inputs und Outputs und durch Spezifikation der Funktion der Zelle aufgerufen:

```

PSFHWSpec specAND ;
specAND.oplist[opAND] = true ;
specAND.num_inpins = 2 ;
specAND.num_outpins = 1 ;
PSFCellBehaviour* andbehav = new PSFCellBehaviour (&specAND) ;
PSFCell* and = new PSFCell (specAND.makeUpCellName (), *andbehav) ;

```

Im obigen Beispiel wird eine AND-Zelle spezifiziert. Das Format der Library-Funktion wird innerhalb der Klasse **PSFLib** erklärt (siehe auch Kapitel 4). Daneben ist es noch möglich, Zellen nur durch Angabe eines Names anzulegen oder eines benutzerdefinierten Typen aufzurufen. Im ersten Fall handelt es sich dann um hierarchische Komponenten, die aus anderen Objekten zusammengesetzt sind, und im zweiten Fall um Wahrheits-Tafeln **PSFBlifTable**, generische Latches **PSFBlifLatch** und endliche Automaten **PSFFSM**.

Das Instanzieren einer Zelle erfolgt kombiniert mit dem Anlegen dieser Instanz in einem bestehenden Design:

```

PSFCell *design = new PSFCell ("Vaterzelle");
PSFCellInst *new_ci = design->addCellInst (and, "NeueInstanz");

```

In diesem kurzen Beispiel wird zunächst eine leere Zelle - "Vaterzelle" - generiert. In der zweiten Zeile wird dann die Zelle \*and aus dem vorherigen Beispiel instanziiert und zugleich der Vaterzelle als Komponente \*new\_ci zugewiesen.

In dem folgenden Beispiel soll die instanziierte Zelle "NeueInstanz" direkt mit den Eingängen und dem Ausgang der Vaterzelle verbunden werden. Da die Vaterzelle mit einem einfachen Konstruktor erzeugt wurde, fehlen ihr allerdings noch diese Ein- und Ausgänge. Sie müssen zunächst mit den entsprechenden Methoden angelegt werden.

```

.latch [2] LatchOut_v4 1
.start_kiss
.i 2
.o 3
.p 8
.s 8
.r s1
10 s1 s2 010
00 s2 s3 001
-1 s3 s4 000
-1 s4 s5 000
10 s5 s6 100
-1 s6 s7 000
10 s7 s8 100
-- s8 s1 100
.end_kiss
.latch_order LatchOut_v2 LatchOut_v3 LatchOut_v4
.code s1 101
.code s2 010
.code s3 011
.code s4 100
.code s5 001
.code s6 110
.code s7 000
.code s8 111
.names LatchOut_v2 LatchOut_v3 LatchOut_v4 [0]
00- 1
0-1 1
111 1
.names LatchOut_v2 LatchOut_v3 LatchOut_v4 [1]
00- 1
101 1
010 1
.names LatchOut_v2 LatchOut_v3 LatchOut_v4 [2]
-00 1
111 1
010 1
.names LatchOut_v2 LatchOut_v3 LatchOut_v4 OUT_0
00- 1
111 1
.names LatchOut_v2 LatchOut_v3 LatchOut_v4 OUT_1
101 1
.names LatchOut_v2 LatchOut_v3 LatchOut_v4 OUT_2
010 1
## output of cell/fsm interface
.names IN_0[0] IN_0
1 1
.names IN_1[0] IN_1
1 1
.names OUT_0 OUT_0[0]
1 1
.names OUT_1 OUT_1[0]
1 1
.names OUT_2 OUT_2[0]
1 1
.end

```

Die Treiber am Ende der Datei dienen der Bindung der Pins an die vom Zustandskodierer generierten Namen der Ein- bzw. Ausgänge.

```

    CLK => CLK,
    Pin_gout1 (0) => WIRE_gout1(0),
    Pin_gin2 (0) => WIRE_gin2(0)
  );
  ...
end STRUCTURAL;

```

- Benutzer-definierte Typen (Wahrheits-Tafeln, Latches, FSMs) werden als Prozesse innerhalb der `architecture`-Direktive behandelt. Die Ausgabe von endlichen Automaten in VHDL erfolgt im Synopsys-Dialekt für *endliche Mealy-Automaten* (Vgl. VHDL-Compiler-Handbuch, A-5).

```

architecture STRUCTURAL of table_cell10 is
begin
  process (Pin_i1 , Pin_i4 )
  begin
    Pin_gout1 (0) <= (Pin_i1(0) AND '1') OR
      ('1' AND Pin_i4(0) );
  end process;
end STRUCTURAL;

```

- Für gemappte Strukturen wird der entsprechende VHDL-Generator, der zu der Bibliotheks-Komponente gehört, aufgerufen. Diese Generatoren sind zur Zeit noch nicht verfügbar.

Bei der Blif-Ausgabe war darauf zu achten, daß in einer Blif-Datei nur eine bereits synthetisierte Maschine ausgegeben werden kann. Die Ausgabe erfolgt hier über die Funktion `wri - teBlifFSM( )`, die den Automaten auf eine temporäre Kiss-Datei schreibt, auf dieser dann JEDI (Zustandskodierung) laufen läßt, und die synthetisierte Datei auf das Ausgabe-File schreibt. So wird für die Kiss-Datei, die in einer Zelle `Mealy_Machine` realisiert wird,

```

.start_kiss
.i 2
.o 3
.s 8
.r s1
10 s1 s2 010
00 s2 s3 001
-1 s3 s4 000
-1 s4 s5 000
10 s5 s6 100
-1 s6 s7 000
10 s7 s8 100
-- s8 s1 100
.end_kiss

```

der folgende Blif-Code generiert:

```

## Generated by BLIF writer (heiyo) at Thu Jan 6 15:54:59 1994

.model Mealy_Machine
.inputs IN_0[0] IN_1[0]
.outputs OUT_0[0] OUT_1[0] OUT_2[0]
.latch [0] LatchOut_v2 1
.latch [1] LatchOut_v3 0

```

keine zugehörige Zelle beim Einlesen eines Subcircuits, so wird eine Zelle zunächst generiert, instanziiert und entsprechend der Parameterliste verdrahtet. Da die Parameterliste nicht notwendig vollständig ist, ist eine Modifikation der Anschlüsse der zugehörigen Zelle beim Einlesen des entsprechenden Modells später noch möglich.

Die Ausgabe wird durch die Funktionen `writeBlif()` und `writeVHDL()` realisiert, die jeweils in den Modulen `writeVHDL.[CH]` bzw. `writeBlif.[CH]` implementiert sind.

Alle Komponenten werden mit dem gleichen synchronen Taktsignal angesteuert. Um die Ausgabe zu vereinfachen, wurde die Abfrage, ob es sich um ein sequentielles Bauteil oder eine rein kombinatorische Schaltung handelt, eingespart. D.h. alle Komponenten verwenden das gleiche synchrone Taktsignal. Bei der VHDL-Ausgabe sind folgende Unterscheidungen zu treffen:

- Netzlisten, die durch nicht-gemappte oder benutzerdefinierte Zellen entstehen, werden als Mengen von Entities ausgegeben. Alle Entities werden in einer einzelnen Datei (single-output) oder der Übersichtlichkeit halber pro physikalischer Datei mit nur einem Entity ausgegeben; der Name der Datei ergibt sich im zweiten Fall durch `<cell-name>.vhd1`. Instanziierungen von Zellen werden durch die `component`-Anweisung und dem entsprechenden Port-Mapping abgearbeitet. Für die internen Verbindungen innerhalb der Zelle werden Signale definiert, die, um eine Unterscheidung zu den gleichnamigen Pins zu erlauben, mit dem Präfix `WIRE_` beginnen. Eine entsprechende Namens-Konvention gibt es auch für die Pins der auszugebenden Zelle:

```
-- Generated by VHDL writer (heiyo) at Mon Dec 27 15:33:22 1993

library IEEE;

use IEEE.std_logic_1164.all;
...

entity TEST is
  Port (
    CLK : in Bit;
    Pin_il : in Bit_vector(0 downto 0);
    ...
  );
end TEST;

architecture STRUCTURAL of TEST is
  signal WIRE_gout1 : Bit_vector (0 downto 0);
  ...

  component latch_cell0
  Port (
    CLK : in Bit;
    Pin_gout1 : in Bit_vector(0 downto 0);
    Pin_gin2 : out Bit_vector(0 downto 0)
  );
end component;
.....
begin
  latch_inst0: latch_cell0
  Port Map (
```

archie beschränkt. Ein zusätzlicher Parameter von `expandCellInst()` erlaubt die Angabe der Hierarchie-Stufen auf denen die Zellinstanz rekursiv expandiert werden soll.

### 3.3.3.3 Interfaces

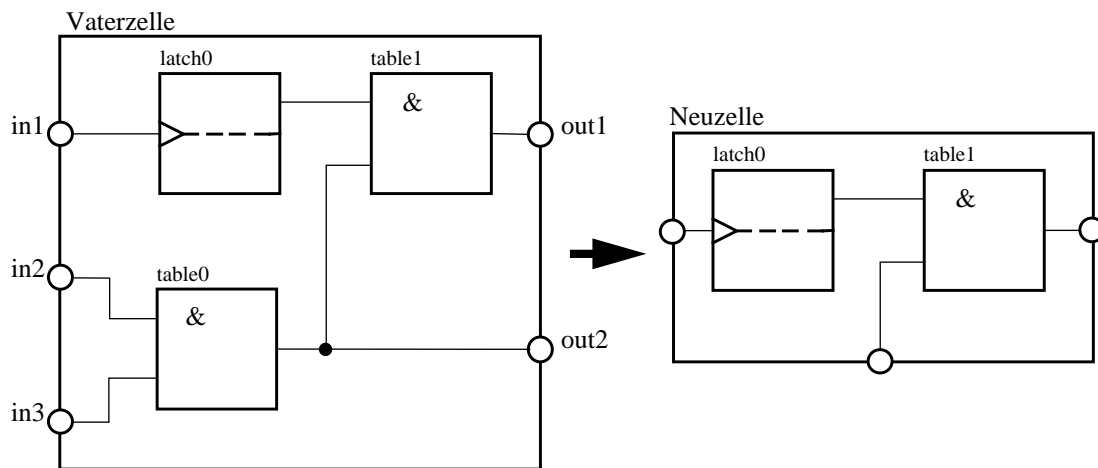
Eine Ausgabe eines Entwurfes ist momentan in zwei verschiedenen Formaten möglich: zum einen im Blif-Format und zum anderen als VHDL-Strukturen (S-VHDL). Daneben ist es noch möglich, Blif-Beschreibungen einzulesen (`readBlif()`).

Das Einlesen wird durch die Funktion `readBlif()` realisiert, die in dem Modul `readBlif.[CH]` implementiert ist.

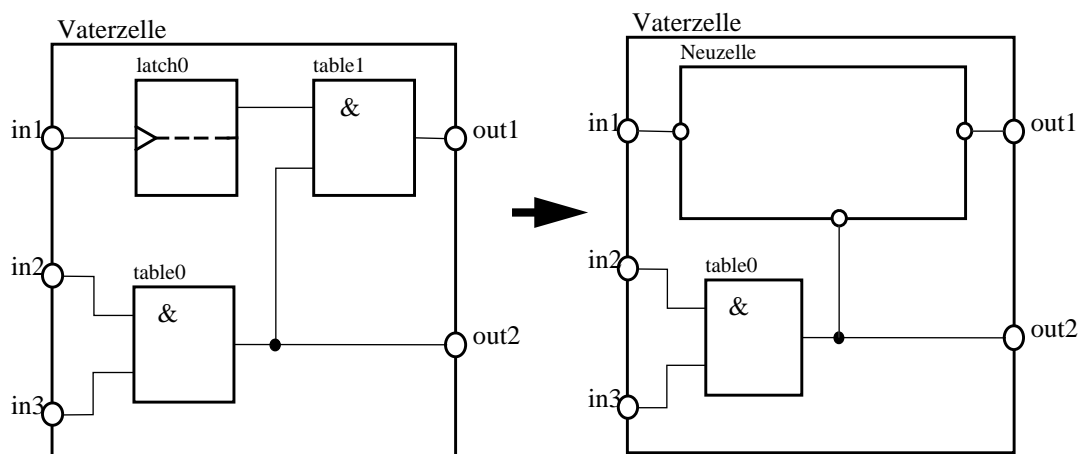
Im Moment werden allerdings nicht alle Möglichkeiten, die eine Blif-Beschreibung bietet, umgesetzt. So werden Beschreibungen eines Automaten im Kiss-Format noch ignoriert. D.h. Textstellen, beginnend mit `.start_kiss` und endend mit `.end_kiss`, sowie `.latch_order` und `.code` (wie weiter unten beschrieben) bleiben beim Einlesen unberücksichtigt. Es werden jedoch Hierarchien in Form von Subcircuits (`.subckt`) eingelesen. Generell ist folgendes zu berücksichtigen:

1. Für jedes Modell, welches in der einzulesenden Blif-Beschreibung vorkommt, wird eine Zelle angelegt. Die Funktion `readBlif()` gibt eine Referenz auf die Zelle zurück, die textuell das erste Modell darstellt. Diesen Zellen werden dann durch das weitere Einlesen von Gattern, Flipflops oder Subcircuits Zellinstanzen angefügt, die über Wires miteinander, und zu den Anschlüssen der Zelle selbst, in Verbindung gebracht werden.
2. Für jedes Gatter (`.names`) sowie Flipflop (`.latch`) wird eine Zelle generiert. Diese Zelle wird instanziiert und in die Zelle eingefügt, die als Modell dieses Gatter bzw. Flipflop umfaßt. Dabei werden die durch die Beschreibung gegebenen Verbindungen in Form von Drähten in der umfaßenden Zelle hergestellt. Bei der Generierung von Zellen, die auf Gattern bzw. Flipflops basieren werden intern Namen vergeben, die nicht in der Blif-Beschreibung als Modellnamen verwendet werden dürfen. Dies sind die Namen `'table_celli'` für Gatter, sowie `'latch_celli'` für Flipflops, wobei *i* eine natürliche Zahl ist.
3. Hierarchien sind in der Blif-Beschreibung durch die Deklaration von Subcircuits (`.subckt`) gekennzeichnet. Zu jedem Subcircuit muss eine Beschreibung des zugehörigen Modells in der Blif-Beschreibung existieren. Dabei ist es vollkommen unerheblich, ob in der Blif-Beschreibung zuerst das Modell beschrieben wird, oder das Modell als Subcircuit Verwendung findet, oder eine Kombination von beiden zur Anwendung kommt. Existiert bereits eine zugehörige Zelle zu einem Subcircuit, so wird beim Einlesen des Subcicuits diese Zelle instanziiert, und entsprechend der Parameterliste verdrahtet (mit Hilfe der Methoden `PSFCell::addWire()`, `PSFCell::addWireSource()`, `PSFCell::addWireTarget()`). Existiert noch

1. `collectCell()` wird auf einer Zelle mit einer Liste von darin enthaltenen Zellinstanzen aufgerufen. Es wird eine neue Zelle generiert, deren interner Aufbau durch die aufgelisteten Zellinstanzen induziert wird. An der Vaterzelle selbst finden keine Änderungen statt. Die folgende Abbildung zeigt das ‘Sammeln’ der Zellinstanzen ‘latch0’ und ‘table1’ der Vaterzelle. Die neu generierte Zelle trägt in der Abbildung den Namen ‘Neuzelle’.



2. `cutCell()` impliziert die Funktion `collectCell()`. Darüberhinaus werden in der Vaterzelle die Zellinstanzen, mit denen `cutCell()` aufgerufen wird, durch eine Instanz der neu generierte Zelle ersetzt. Die folgende Abbildung zeigt die Vaterzelle vor und nach dem Aufruf von `cutCell()` mit einer Liste der Zellinstanzen ‘latch0’ und ‘table1’ als Parameter.



3. `expandCellInst()` ist im Prinzip die Umkehrfunktion von `cutCell()`. Diese Funktion expandiert eine Zellinstanz, indem der interne Aufbau der Zellinstanz in die Vaterzelle übertragen und dort Verdrahtet wird. Die expandierte Zellinstanz selbst wird aus der Vaterzelle gelöscht. Das Expandieren ist jedoch nicht auf eine Ebene der Hier-

### 3.3.3.1 Grundfunktionen

Hier werden Methoden

- zum Anlegen neuer Objekte vom Typ **PSFCell** oder Instanzen (**PSFCellInst**) davon (Konstruktoren),
- zum Anlegen einer Verbindung zwischen zwei oder mehreren Instanzen (**PSFWire**)

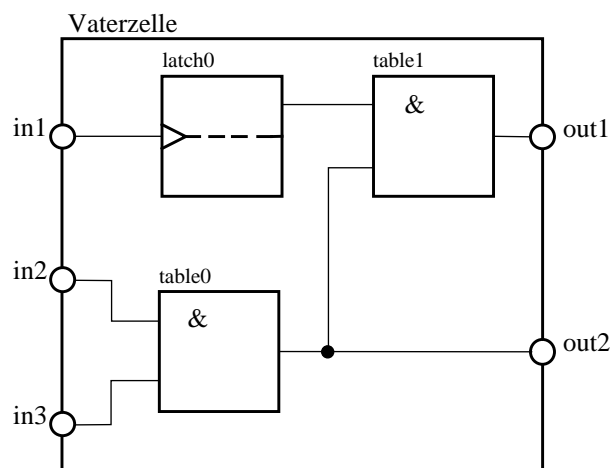
zur Verfügung gestellt. Die Reihenfolge des Aufrufes ist stets so, daß zunächst eine Zelle generiert werden muß; der Aufruf des Konstruktors hat unter Angabe eines Namens und eines Attributes für die interne Beschreibung (Verhalten - Struktur) der Zelle zu erfolgen. Existiert eine Zelle mit dem Namen bereits, so wird ein Fehler gemeldet. Die Schnittstelle der Zelle kann nun durch Hinzufügen von Pins (**PSFPin**) modifiziert werden.

Die Instanziierung einer Zelle erfolgt ebenfalls durch Angabe eines Namens für die neu erzeugte Instanz. Die Erzeugung von Verbindungen zwischen Komponenten ist an die Existenz der Instanzen zu verbindenden Pins gebunden. Beispiele zur Benutzung der Klasse werden oben beschrieben.

### 3.3.3.2 Hierarchische Operationen/Funktionen

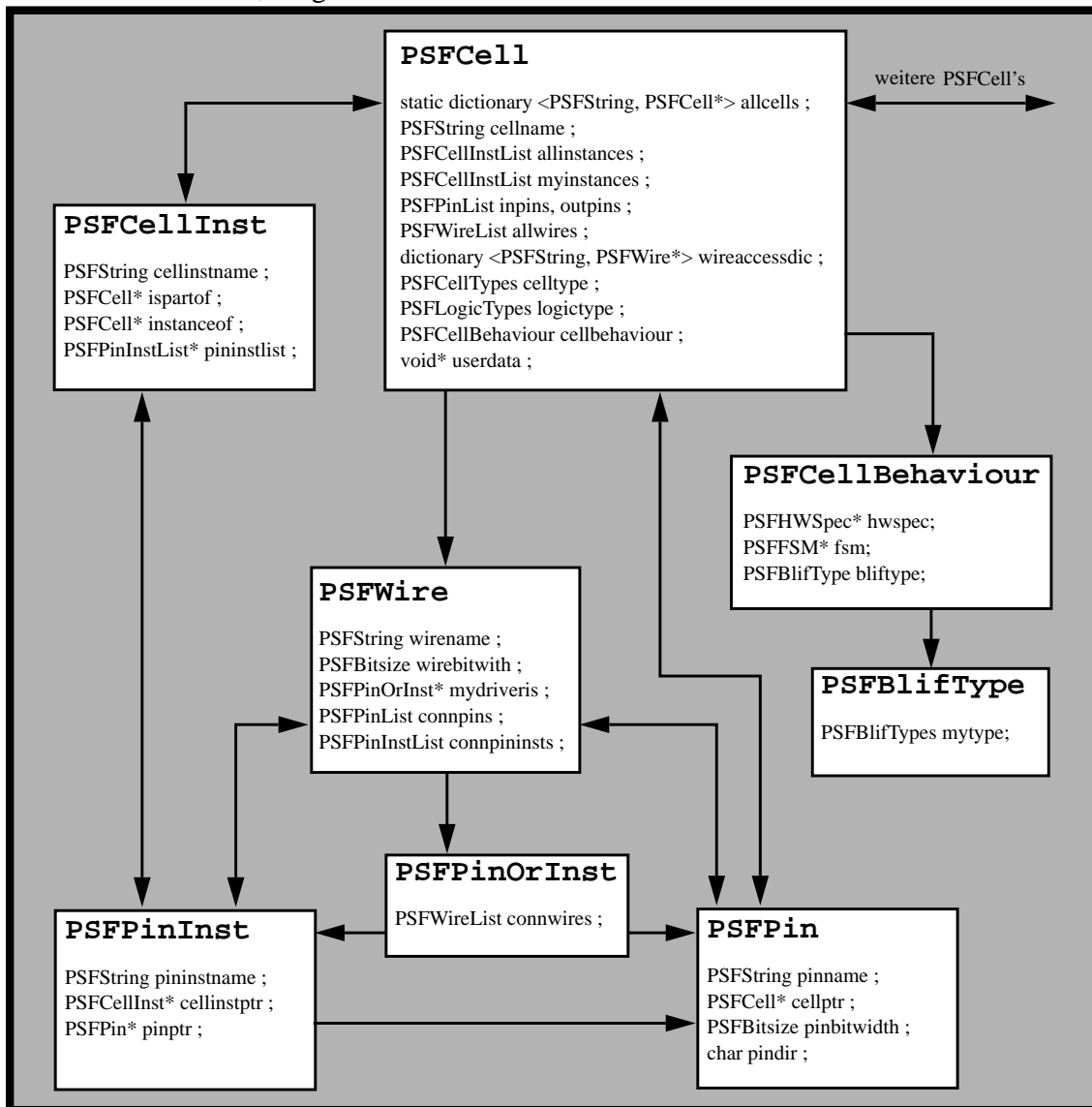
Hier wird das Ausschneiden von Zellen und deren Instanziierung (`cut`, `collect`), sowie das Expandieren von instanziierten Zellen in der Vaterzelle beschrieben.

Die zur Verfügung stehenden hierarchischen Operationen sind `cutCell()`, `collectCell()`, sowie `expandCellInst()`. Sie sind in `netutils.[CH]` implementiert. Im folgenden bezeichne 'Vaterzelle' ein Design vom Typ **PSFCell**, deren interner Aufbau durch Zellinstanzen und deren Verdrahtung untereinander beschrieben ist. Ein Beispiel einer solchen Vaterzelle zeigt die folgende Abbildung.



### 3.3.2 Definition

In der folgenden Graphik werden die wesentlichen Klassen, die in und von der Klasse **PSFCell** benutzt werden, dargestellt.



Die Kanten im Graphen deuten an, von welchen Klassen auf welche Informationen zurückgegriffen werden kann.

### 3.3.3 Realisierung

Im folgenden werden die wesentlichen Funktionen zur Handhabung der Klassen und die Schnittstellen zum Einlesen von Entwürfen vorgestellt. Die Methoden beziehen sich auf die Klasse **PSFCell**.

Ein Hardware-Entwurf wird durch eine Netzliste von verbundenen Komponenten dargestellt, die Instanzen einer Definition sind. Eine solche Definitionen wird **PSFCell** genannt. Instanzen von Zellen, also Zell-Instanzen - **PSFCellInst**, werden über Drähte (**PSFWire**) miteinander verbunden, die an den Ein-/Ausgängen der Zelle (**PSFPin**) oder an den Schnittstellen der innen-liegenden Zell-Instanzen (**PSFPinInst**) fixiert werden können.

### 3.3.1 Kontext und Aufgabe des Moduls

Die Klasse **PSFCell** dient zur Beschreibung und Manipulation von hierarchischen Objekten in einer Netzliste. Neue Zellen können Bottom-Up oder durch Herausschneiden einer Vaterzelle erzeugt werden. Im Moment wird noch nicht getestet, ob die Hierarchie konsistent ist, d.h. der Hierarchie-Baum in dem die Knoten Zellen und Kanten Instanziierungen von Zellen darstellen ein DAG ist.

2. Edit Menu: Um die selektierten Objekte in den Vorder-/bzw. Hintergrund zu bringen ist der Menüpunkt `front` bzw. `back` anzuwählen. `Select all` selektiert alle Transitionen und Zustände des dargestellten Automaten. "Cut" löscht die selektierten Objekte.
3. Pattern Menu: Hat keine Auswirkungen.
4. In-/Output Menu: "Write Blif" erzeugt eine Ausgabe des Automaten im Blif-Format. "Read Blif" liest die Blif-Datei ein und zeigt die korrespondierende FSM in der Zeichenebene an. "Write C" generiert eine C++-Ausgabe des Automaten.

Das Editor-Fenster weist sechs Buttons auf: `Select`, `Move`, `Show Data`, `Show Hierarchy`, `Corresponding Objects`, `Application Edit Node/Edge`.

1. "Select" ist die Default-Einstellung. In diesem Modus können durch Klicken auf die Zustände und Transitionen die korrespondierenden Objekte selektiert.
2. Durch Betätigung des "Move" Buttons können Objekte durch den Benutzer verschoben werden; der linke Maus-Button muß hierbei gedrückt bleiben.
3. "Show Data" macht die Attribute des selektierten Objektes sichtbar.
4. "Show Hierarchy", "Corresponding Objects" sind im Moment noch nicht implementiert.
5. "Application Edit Node/Edge" startet die gleichen Aktionen wie "Show Data".

### 3.2.6 Probleme

Die Methoden im In-/Output Menu können nur aktiviert werden, wenn keine Objekte selektiert sind. Im Moment ist es noch nicht möglich, neue Knoten und Transitionen zu erzeugen. Durch den Layout-Algorithmus kommt es häufiger vor, daß Knoten oder Transitionen übereinander gezeichnet werden, so daß eine Identifikation teilweise erschwert ist.

## 3.3 Netzlisten - Zellen (PSFCe11)

Die dazugehörigen Klassen werden in den Dateien:

```
net.[CH] writeBlif.[CH] blifypes.[CH]
readBlif[CH] writeVHDL[CH]
```

beschrieben.

```
PSFFSM fsm ();
```

Zunächst wird die Anzahl der Ein- und Ausgangssignale spezifiziert:

```
fsm->SetInputNo (2) ;
fsm->SetOutputNo (3) ;
```

Neue Zustände werden durch den parameter-losen Konstruktor erzeugt:

```
State *state1, *state2;
state1 = new State();
state2 = new State();
```

oder durch die entsprechenden Methoden an die FSM angehängt:

```
state1 = fsm.AddState();
state2 = fsm.AddState();
```

Jeder Zustand kann als Attribute einen Namen, eine Kodierung oder ein Flag zur Erkennung des Startzustandes erhalten; diese können wie folgt modifiziert werden:

```
fsm.SetName (state1,"name1") ;
fsm.SetEncoding (state1,"000") ;
fsm.SetStartState (state1) ;
```

state1 wird der Startzustand. Eine neue Transition zwischen state1 und state2 kann nun angelegt werden:

```
Transition *trans;
trans = fsm.AddTransition(state1,state2);
```

Eine Boole'sche Formel (**PSFBoolFormula**) kann jetzt an diese Transition gehängt werden. Diese Formel kann, wie oben dargestellt verschiedene Ausprägungen haben, z.B. als Wahrheitstafel (**PSFTruthTable**).

```
tr->SetFormula(bf);
```

### 3.2.5 Benutzung des Editors (FSMEditor)

Der FSM Editor wird durch Eingabe des Kommandos PSF gestartet. Zur Demonstration erscheint eine FSM im Fenster (linke/obere Ecke).

Der Editor besitzt vier **Pulldown Menus**: File, Edit, Pattern, In-/Output.

1. File Menu: Um den Editor zu verlassen, ist Quit zu aktivieren.

ionen ist implizit: Transitionen, die zu Hyperkanten korrespondieren, stellen *Fork*-Transitionen dar, während “multiple source” Hyperkanten normal genannt werden. Aus technischen Gründen kann eine Transition nicht beides sein.

4. Ein Objekt der Klasse **PSFSTG** stellt einen STG dar, d.h. Knoten korrespondieren zu Zuständen und Kanten zu Transitionen.
5. Die Klasse **PSFHierState** ist abgeleitet von der Klasse **PSFState** und bildet auf komplexe Zustrände ähnlich der *STATECHART* Semantik ab. Ein Teil der Information, die an den Knoten gebunden ist, betrifft den hierarchischen Verweis auf den Sub-Graphen und den Graphen zu dem der Knoten gehört.
6. Die Klasse **PSFHierPath** stellt einen Mechanismus zur Verbindung von hierarchischen Zuständen bereit. Ein Pfad in der Hierarchie ist ein Menge von hierarchischen Zuständen (d.h. Instanzen von Automaten) dar, die relativ zum aktuellen Zustand zu durchlaufen sind.

Ein einfacher oder komplexer Zustand enthält Informationen über die Kodierung, den Namen, die interne Nummer, den Start-Zustand und Referenzen auf Knoten im **PSFCdfg** (siehe Kapitel 2). Außerdem können die symbolischen Namen zur Referenzierung benutzt werden.

In jedem STG oder komplexen Zustand repräsentiert ein ausgezeichneter Zustand den Startzustand des Sub-Automaten. Wie oben ausgeführt wird eine Transition mit einer Boole’schen Formel annotiert. Die unterliegende Graph Datenstruktur ist weitgehend verborgen (**PSF\_GRAPH**). Die Erzeugung hierarchischer Zustände ist durch die `Collect` Methode möglich.

### 3.2.3 Abhängigkeiten

**PSFTransition** hängt stark von den Klassen **PSFBoolFormula** und **PSFSTG** ab, da eine Transition mit einer booleschen Formel annotiert wird und der STG aus einer Menge von Kanten besteht.

```

State:           PSFSTG
PSFHierState:  PSFState, PSFSTG
PSFTransition: PSFFSM, PSFSTG, Timing, PSFBoolFormula
PSFFSM:        PSFSTG, State, PSFHierState
PSFBoolFormula: PSFString

```

### 3.2.4 Benutzung der Klasse

Der Konstruktor wird ohne Argumente aufgerufen:

Sequentielle Abläufe, zum Beispiel zur Steuerung von Datenpfaden, werden durch die hier beschriebene Klasse für endliche Automaten dargestellt. Neben der Programmier-Schnittstelle ist es noch möglich, über die Klasse **FSMEditor** Automatenbeschreibungen zu erzeugen und in verschiedenen Formaten ausgeben zu lassen (C, Blif/Kiss, VHDL).

### 3.2.1 Kontext und Aufgabe des Moduls

Die Klasse **PSFFSM** erlaubt die hardware-orientierte Beschreibung endlicher Automaten. Mit diesen Beschreibungen lassen sich Zellen in der Netzliste annotieren und damit komplexe Steuerungsaufgaben realisieren.

### 3.2.2 Definition

Die hier realisierte Automaten-Darstellung ist vom *Mealy-Typ*, d.h. die Ausgabe-Funktion  $\lambda: S \times I \rightarrow O$  hängt vom aktuellen Zustand  $s \in S$  und vom Input  $i \in I$  ab. Daneben ist es noch möglich, hierarchische Steuerungen, wobei sich die Hierarchie auf den Zustandsübergangsgraphen bezieht, zu definieren, so wie sie in der High-Level-Synthese auftreten. Unter bestimmten Bedingungen ist es möglich, sog. *Fork-Transitionen* zwischen normalen und hierarchischen Zuständen einzubauen. Zusätzlich wird noch eine *Join-Transition* zur Rückkehr aus parallel arbeitenden Teilautomaten, die durch hierarchische Zustände dargestellt werden, angeboten.

Als Ausgangs-Basis für die Darstellung des Automaten wurde ein Graphen-Modell gewählt, welches die Darstellung von Zuständen durch Knoten und Transitionen durch Mengen von Kanten vorsieht. Das Erzeugen von Zuständen und Transitionen wird damit auf das Anlegen ebensolcher Objekte zurückgeführt. Nach außen ist diese Realisierung weitgehend gekapselt, so daß der Benutzer lediglich mit der Handhabung der obengenannten Objekt-Klassen zu tun hat.

1. Die Klasse **PSFFSM** enthält das Interface des Automaten, das aus der Maske für die Inputs und Outputs besteht, die für den Aufbau der Zustandstabelle verwendet wird. Dieses Interface ist einheitlich für alle Transitionen in allen Hierarchieebenen des Automaten. Der Automatengraph (*state-transition graph* - STG) existiert als Zeiger in der Klasse.
2. Die Klasse **PSFState** enthält alle erforderlichen Informationen, um einen *einfachen* Zustand (d.h. einen Zustand, der nicht hierarchisch, also aus anderen Zuständen zusammengesetzt, ist) des Automaten.
3. Die Klasse **PSFTransition** modelliert Transitionen im Automaten, d.h. Übergänge von einem in einen anderen Zustand im Automaten und die Bedingungen, die zu einer solchen Transition führen. Die Unterscheidung zwischen *normalen* und *Fork-Transit-*

Ausgänge für diese Belegung den Wert 0 annehmen. Daneben ist es auch möglich, daß ein Ein- oder Ausgang undefiniert ist. Ein Beispiel ist unten angegeben:

```
--1 10
1-0 01
```

### 3.1.3 Abhängigkeiten

Eine Zeile in der Wahrheitstabelle wird durch einen Bitvektor (**PSFBitvector**) realisiert. Ein solcher Bitvektor stellt einen String über ACSII-Zeichen dar. Damit wird es ermöglicht, leicht auf eine n-wertige Logik umzustellen. Wird ein String über Zeichen (char), die von 0,1,-,~ verschieden sind, definiert, so wird keine Warnung ausgegeben. Lediglich bei der Ausgabe als PLA erfolgt eine Fehlermeldung.

### 3.1.4 Benutzung der Klasse

Im Konstruktor werden die Anzahl der Ein- und Ausgänge angegeben:

```
PSFTruthTable(3,2);
```

Modifikationen können durch inkrementelles Anhängen einer Zeile an die bisherige Formel getätigt werden. Eine solche Erweiterung (Zeile) ist vom Typ **PSFBitvector**. Beispiel:

```
bf.addRow( "--110" ); bf.addRow( "1-001" ); bf.addRow( "0-100" );
```

Weitere Funktionen, wie der Zugriff auf die i-te Zeile der Tafel etc. werden unterstützt. Dabei ist es möglich, sowohl auf die Bedingung und Belegung der Ausgänge zusammen oder auch separat zuzugreifen. Desweiteren ist es möglich, logische Operationen auf den Tafeln auszuführen (logische Operationen: shift, OR, AND, NOT etc.).

### 3.1.5 Probleme

Bisher nicht bekannt.

## 3.2 Finite State Machines - PSFFSM

Die Implementation der hier beschriebenen Klassen und Funktionen befindet sich in den Dateien:

```
kiss.[CH] stg.[CH] cwrite.[CH] timing.H transition.[CH]
fsm.[CH] writeVHDLFSM.[CH] hierpath.[CH] state.[CH]
```

## 3 PSFCdp

Im Folgenden werden die Klassen und Interfaces zur Beschreibung der strukturellen Ebene dokumentiert. Neben einem Anschluß an das Berkeley SIS-Tool über eine Blif-Schnittstelle, ist es auch ermöglicht, Struktur-Beschreibungen im Synopsys VHDL-Dialekt auszugeben.

### 3.1 Boolesche Formeln - **PSFBoolFormula**

Zur Darstellung Boole'scher Formeln dienen die Dateien:

```
truthtable.[CH] boolean.[CH]
```

#### 3.1.1 Kontext und Aufgabe der Klasse

Die Klasse **PSFBoolFormula** stellt einen Datentyp zur Darstellung und Manipulation von Boole'schen Formeln mit einem oder mehreren Ausgängen zur Verfügung. Die Klasse **PSFTruthTable** ist abgeleitet von der Klasse **PSFBoolFormula**, in der nur das Interface (Anzahl der Eingänge und Ausgänge und deren Namen) verwaltet wird, und beschreibt die Realisierung einer Formel durch eine Wahrheits-Tafel. Daneben soll es später noch möglich sein, diese Beschreibung durch BDDs zu ersetzen.

#### 3.1.2 Definition

Das Format einer Formel in der Klasse **PSFTruthTable** ist vergleichbar mit dem ES-PRESSO-PLA-Format, d.h. ist eine Bedingung nicht definiert, so wird angenommen, daß die

PSFDfg: LLIST(LNODE) \*dfgbeginnodelist: Liste aller Knoten vom Typ PSF-BBBeginNode. Diese Liste ist sehr hilfreich, um über alle Blöcke des **PSFDfg** zu iterieren.

PSFDfg: PSFConstTable \*consttab

PSFDfg: PSFOptorTable \*optortab

PSFDfg: PSFSymbolTable \*localsymtab

PSFDfg: PSFSymbolTable \*globalsymtab

PSFDfgNode: PSFCsel\*csel: Für hierarchische Knoten ist das korrekte **PSFFuncsel** zu generieren.

PSFDfgNode list<FCTINTERFACE> \*fct\_interface: Diese Liste realisiert die Zuordnung der aktuellen Parameter zu den formalen Parametern. Der Datentyp FCTINTERFACE ist definiert durch Abbildung 25.

```
typedef struct fctinterface {
    LNODE call;    // aktueller Parameter
    LNODE func;   // formaler Parameter
    VARATTRIB va; // Reference oder Value Parameter
} *FCTINTERFACE;
```

Abb. 25: Definition der Functionsschnittstelle

Dieses Kapitel beschrieb die softwareorientiert Ebene des PSF, den Control-Data-Flow-Graph, **PSFCdfg**. Sein Prinzip und die technische Realisierung wurde beschrieben.

Die vom Abstraktionsgrad niedrigere Ebene ist der Control-Data-Path. Seine Beschreibung erfolgt im folgenden Kapitel.

PSFCdfg \*sub\_cdfg: Dieser Pointer ist die Referenz vom aufrufenden zum gerufenen Modul. Auch der Zugriff auf den **PSFCfg** bzw. **PSFDfg** des Submoduls ist hierüber realisiert.

string funcname: Dieser *LEDA*-String speichert den Namen der aufgerufenen Funktion, also des Submoduls.

## 2.2.5 Implementation des Hierarchiekonzepts

Die Implementation des Hierarchiekonzeptes ergibt sich aus der korrekten Verwaltung der einzelnen **PSFCdfg**-Konstrukte. Es wird keine Klasse für Hierarchiedaten angelegt. Aber die einzelnen Konventionen des **PSFCdfg**-Datenformates, müssen für alle Hierarchie-Aktionen (Create-, Delete-Submodul) beibehalten werden. Eine Liste der Datentrukturen, die durch die betreffenden Konventionen betroffen sind, wird im Folgenden angegeben.

PSFCdfg: PSFDfg \* dfg

PSFCdfg: PSFCfg \* cfg

PSFCdfg: string name

PSFCdfg: LNODE first\_cfg\_node

PSFCdfg: LNODE first\_dfg\_node

PSFCfg: PSFCdfg \*cdfg;

PSFCfgNode: PSFCdfg\* sub\_cdfg: Hier muß ein Pointer auf das Submodul eingetragen werden, wenn der Knoten hierarchisch ist. Anderfalls wird der Pointer auf NULL gesetzt (default value).

PSFDfg: list <CALLINFO>\* fct\_interface: Das Funktionsinterface wird durch Elemente vom Typ CALLINFO beschrieben, die in einer Liste zusammenfasst werden. Der Typ CALLINFO ist definiert durch Abbildung 24.

```
typedef struct callinfo {
    LNODE n;          // formaler Parameter
    VARATTRIB va;    // Reference oder Value Parameter
} *CALLINFO;
```

Abb. 24: Definition CALLINFO

PSFDfg: LLIST(LNODE) \*hiernodelist: Liste aller hierarchischen Knoten des aktuellen DFGs.

PSFDfg: LLIST(LNODE) \*exitlist: Liste aller PSFBBEndNodes, von denen aus der aktuelle **PSFDfg** verlassen werden kann.

### 2.2.3.4 Definition PSFOptorCsel

Innerhalb des **PSFDfg** bestehen im wesentlichen Verbindungen zwischen Variablen und Operatoren. Zur Beschreibung der **PSFDfgOptorNodes** wird das **PSFOptorCsel** verwendet, welches in Abbildung 22 dargestellt ist.

```

class PSFOptorCsel : public PSFCsel {

    private:

        OPTOKEN optor_id;
        ALGPROP algprop;
        int bitsize;

    public:

        PSFOptorCsel(OPTOKEN optor_id,int size=32,
                    ALGPROP prop = NOPROP);
}

```

Abb. 22: C++ Definition **PSFOptorCsel**

Die Datenelemente haben folgende Bedeutung.

**OPTOKEN optor\_id**: Der Typ des Operators wird hier eingetragen. Die Identifikatoren stehen in **PSFOptoken\_def.H**.

**ALGPROP algprop**: Die algebraischen Eigenschaften des Operators werden hier als Konstanten eingetragen. Diese sind aus dem Modul **PSFTypes.H** zu entnehmen.

**int bitsize**: Die Anzahl der Bits, auf der der Operator arbeitet, ist hier einzutragen.

### 2.2.4 Definition PSFFunCsel

Informationen über den Funktionsnamen und den **PSFCdfg** der Funktionsimplementation sind an das spezielle **PSFFunCsel** des hierarchischen Knoten gebunden. Abbildung 23 zeigt die **PSFFunCsel**-Definition.

```

class PSFFunCsel : public PSFCsel {

    private:

        PSFCdfg *sub_cdfg; // pointer to submodul
        string funcname; // name of function

    public:

}

```

Abb. 23: C++ Definition **PSFFunCsel**

### 2.2.3.3 Definition **PSFConstCsel**

Es gibt verschiedene Arten von Konstanten in einem Programm. Um diese innerhalb einer Struktur darzustellen, wurde das **PSFConstCsel** entwickelt, welches in Abbildung 21 dargestellt ist.

```
class PSFConstCsel : public PSFCsel {  
  
    private:  
  
        DATYPE type;  
        union {  
            long i;  
            double r;  
            char c;  
        } val;  
        int bitsize;  
}
```

Abb. 21: C++ Definition **PSFConstCsel**

Die Datenelemente des **PSFConstCsel** haben folgende Bedeutung.

**DATYPE type**: Der Datentyp der Konstanten wird hier eingetragen. Die Kennzeichnung des Typs steht in dem **PSFTypes.H** und ist mit denen zur Definition der **PSFTypeC-sels** identisch.

**union {..} val**: Der Wert der Konstanten wird entsprechend seines Datentyps in dieser Struktur abgelegt.

**int bitsize**: Die Anzahl der Bits, die durch die Konstante belegt werden, wird in diesem Datenelement abgelegt.

Als letztes **PSFCsel** soll in dem nächsten Abschnitt, das **PSFOptorCsel** beschrieben werden.

`int bitsize`: Die Bitbreite der Variablen wird eingetragen.

`PSFVarCsel()`: Der Konstruktor des **PSFVarCsel** ruft den Konstruktor der Basisklasse auf und trägt dort die entsprechende Daten ein. In der Basisklasse erhält dadurch als Datum des `idtype` dem Wert `PSFVarID`. Andere solche Definitionen entsprechend der abgeleiteten Klassen, stehen in dem Modul `PSFCselID_def.H`. Der `char* idt` enthält den Namen der Variable.

Zum Beschreiben des Variablentyps wird ein **PSFTypeCsel** definiert und auf dieses wird von dem **PSFVarCsel** verwiesen. Die Definition befindet sich im folgenden Abschnitt.

### 2.2.3.2 Definition PSFTypeCsel

Diese abgeleitete Klasse wird vielfältig eingesetzt, um Variablentypen zu spezifizieren.

```
class PSFTypeCsel : public PSFCsel {
    private:
        DATYPE datatype;
        int lo_range;
        int hi_range;
    public:
        PSFTypeCsel(char* idt,DATYPE st,
            int lo = 0,int hi = 0);
}
```

Abb. 20: C++ Definition **PSFTypeCsel**

Die Datenelemente der abgeleiteten Klasse **PSFVarCsel** haben folgende Bedeutung.

`DATYPE datatype`: Hier wird eine spezielle Kennzeichnung für den darzustellenden Datentyp eingetragen. Die genaue Definition dieser Kennzeichnung steht in dem Modul `PSFTypes.H`.

`int lo_range`: Bei komplexen Datentypen wie Feldern wird an dieser Stelle die untere Bereichsgrenze eingetragen.

`int hi_range`: Entsprechend wird hier die obere Bereichsgrenze eingetragen.

`PSFTypeCsel()`: Der Konstruktor des **PSFTypeCsel** ruft den Konstruktor der Basis-klasse auf und trägt dort die entsprechende Werte ein. Der `char* idt` enthält den Namen der Variable.

Zur Beschreibung von Konstanten existiert ein spezielles **PSFConstCsel**, welches im folgenden Abschnitt beschrieben wird.

Nun sollen einige abgeleitete Klassen des **PSFCsel** erläutert werden. Es gibt folgende verschiedene abgeleitete Klassen: **PSFArrTypeCsel**, **PSFConstCsel**, **PSFComponentCsel**, **PSFFunCsel**, **PSFOptorCsel**, **PSFModuleCsel**, **PSFRectTypeCsel**, **PSFTypeCsel** und **PSFVarCsel** deren genaue Definition in dem Modul **PSFCsel.H** steht. Um nicht alle abgeleiteten Klassen aufzuführen, werden hier nur die wichtigsten erläutert.

### 2.2.3.1 Definition PSFVarCsel

Der **PSFDfg** besteht im wesentlichen aus Variablen und Operatoren. Dazu sind spezielle **PSFCsel** definiert. Das in Abbildung 19 aufgeführte **PSFVarCsel** ist als abgeleitete Klasse von **PSFCsel**. Durch einen Verweis aus dem **PSFDfgNode** wird es zur detaillierten Beschreibung einer Variablen referenziert. Zu jeder Variablen der zu synthetisierende Spezifikation, existiert nur ein **PSFVarCsel**, welches von verschiedenen **PSFDfgNodes** referenziert wird.

```
class PSFVarCsel : public PSFCsel {
    private:
        PSFTypeCsel *typescel;
        VARATTRIB attrib;
        int blkno;
        int bitsize;
    public:
        PSFVarCsel( char* idt,
                   VARATTRIB attr = ATTNONE,
                   PSFTypeCsel *ty = NULL,
                   int blk = BLK_LOCAL,
                   int size = 0);
};
```

Abb. 19: C++ Definition **PSFVarCsel**

Die Datenelemente der abgeleiteten Klasse **PSFVarCsel** haben folgende Bedeutung.

**PSFTypeCsel \*typescel**: Durch den Verweis auf das **PSFTypeCsel** wird der Variablentyp beschrieben (siehe Abbildung 20).

**VARATTRIB attrib**: Die Variable wird hier attribuiert, um zu erkennen, ob es eine Portvariable zum Ein- oder Auslesen eines Werte an externe Devices ist. Die Werte für die Belegung ist aus dem Modul **PSFTypes.H** zu entnehmen.

**int blkno**: Der Gültigkeitsbereich einer Variablen ist hier einzutragen. Eine Variable ist entweder lokal oder global definiert oder sie ist eine Schnittstellenvariable. Die entsprechenden Werte liegen als **define** in dem Modul **PSFTypes.H** vor.

Analog zum **PSFCfg** sind auch für den **PSFDfg** verschiedene Graphenmethoden implementiert. Ein Blick in das Modul `PSFDfg.H` gibt einen detaillierten Überblick der Graphenmethoden.

### 2.2.3 Compilation Structured Element - **PSFCsel**

Der Aufbau der Daten, die an einen **PSFDfgNode** gebunden werden, ist durch Objektorientierung effizient möglich. Dies ist bei der Implementation des **PSFCsel** erreicht worden. Aus einer Basisklasse heraus werden verschiedene Datenelemente für einen **PSFDfgNode** effizient beschrieben. Die an den **PSFDfgNode** annotierte Information ist für den Knoten. Die Datenbehandlung steckt vollständig hinter den Methode der Basisklasse **PSFCsel** sowie deren Ableitungen. Ein Objekt, welches nur aus diese Basisklasse besteht, wird es im **PSFDfg** nicht geben. Objekte, die aus diese Basisklasse abgeleitet werden, rufen allerdings immer den Konstruktor zu dieser Basisklasse auf. Die Basisklasse **PSFCsel** ist wie in Abbildung 18 definiert.

```
class PSFCsel {  
  
    private:  
  
        PSFCSELID idtype;  
        char* ident;  
        int birth;  
        int death;  
  
}
```

Abb. 18: C++ Definition **PSFCsel**

Die Datenelemente der Basisklasse aus Abbildung 18 haben folgende Bedeutung.

**PSFCSELID idtype**: Ähnlich des **idtype** des **PSFDfgNode** wird **idtype** benutzt, um das **PSFCsel** eindeutig zu spezifizieren. Dies einzelnen Kennzeichnung der **PSFCsels**, ist aus dem Modul `PSFCsel_ID_def.H` zu entnehmen. Allerdings wird in diesem ein abgeleitetes Objekt entsprechend der Typkennzeichnung durch den Konstruktor erzeugt.

**char\* ident**: Hier steht ein Verweis auf den Namen des **PSFDfgNode**. Für Variablen, wird hier der Name derselben eingetragen. Er ist mit dem **char\* ident** des **PSFDfgNode** identisch. Für Operatoren wird der Libraryname des Operators eingetragen. Diese ergibt sich aus dem `OPTOKEN_LIBSTRING` des Moduls `PSFOptoken.C`.

**int birth**: Hier wird der Konstrollschritt, des ersten Auftretens des **PSFCsel** in der Synthese eingetragen.

**int death**: Die Konstrollschrittnummer des letzten Auftretens des **PSFCsel** während der Synthese wird hier annotiert.

Die Datenelemente der **PSFDfgEdge** sind bisher noch leer. Daraus ergibt sich die Definition für den **PSFDfg** wie in Abbildung 17.

```

class PSFDfg : public LGRAPH(PSFDfgNode *,PSFDfgEdge *) {
    private:
        PSFCdfg *cdfg;
        PSFSymbolTable *globalsymtab;
        PSFSymbolTable *localsymtab;
        PSFOptorTable *optortab;
        PSFConstTable *conststab;
        list<LNODE> *dfgbeginnodelist;
        list<LNODE> *exitlist;
        list<LNODE> *hiernodelist;
        dictionary<string, Array_info>rtl_array_dict;

    public:
        PSFCfg* GetCfg();

        // several graph methodes are implemented
}

```

Abb. 17: C++ Definition **PSFDfg**

**PSFDfg** ist eine abgeleitete Klasse von LGRAPH.

PSFSymbolTable \*globalsymtab: Verweis auf die globalen Symbole des **PSFDfg**. Die Definition dieser und der folgenden Tabellen findet sich in dem Modul PSFTable.H.

PSFSymbolTable \*localsymtab: Verweis auf die lokalen Symbole des **PSFDfg**.

PSFOptorTable \*optortab: Verweis auf die Operatoren des **PSFDfg**.

PSFConstTable \*conststab: Verweis auf die Konstanten des **PSFDfg**.

list<LNODE> \*dfgbeginnodelist: Verweis auf eine Liste, die alle PSFDfgBBBeginNodes des **PSFDfg** enthält.

list<LNODE> \*exitlist: Verweis auf alle Ausstiegsstellen aus dem **PSFDfg** (Blätter)

list<LNODE> \*hiernodelist: Verweis auf auf alle hierarchischen Knoten des **PSFDfg**.

dictionary<string, Array\_info> rtl\_array\_dict: Dieses Dictionary ist für das C++Front-End nötig (siehe Kapitel 5). Für andere Anwender des **PSFDfg** hat es keine Bedeutung.

PSFCdfg \*cdfg: Verweis auf den zugehörigen **PSFCdfg**.

PSFDfg \*GetCfg(): Diese Methode ermittelt den korrespondierenden **PSFCfg**.

`int alap_cstep`: Kontrollschritt, der bei dem Syntheseprozess (siehe Kapitel 7) während des ALAP-Scheduling ermittelt wird.

`int asap_cstep`: Kontrollschritt, der bei dem Syntheseprozess während des ASAP-Scheduling ermittelt wird.

`int cstep`: Kontrollschritt, der bei dem Syntheseprozess während des entgeltigen Scheduling ermittelt wird.

`int state`: Zustandsnummer für den zu synthetisierende endlichen Automaten. Dieser Wert wird während des Syntheseprozesses annotiert.

`PSFTypeDescr *modtype`: Verweis auf den Modultyp, der den **PSFDfgNode** in der Netzliste darstellen wird. Diese Auswahl wird für Variablen durch die Register-Allocation und für Operatoren durch die FU-Allocation der Synthese vorgenommen.

`PSFInstanceDescr *inst`: Verweis auf die Instanz des Modultyps, der den **PSFDfgNode** in der zu synthetisierenden Netzliste repräsentiert. Diese Auswahl geschieht während des Binding der Synthese.

Knoten innerhalb des Graphen werden durch Kanten verbunden. Die Definition der Kante für den Control-Flow-Graph ist in Abbildung 16 gegeben.

```
class PSFDfgEdge {  
  
    public:  
  
        PSFDfgEdge();  
        ~PSFDfgEdge();  
  
}
```

Abb. 16: C++ Definition der **PSFDfgEdge**

## 2.2.2 Data-Flow-Graph - PSFDfg

Der Data -Flow-Graph besteht **PSFDfgNodes** und **PSFDfgEdges**. Die Deklaration der Klassen ist in den Abbildung 15 und Abbildung 16 dargelegt.

```

class PSFDfgNode {
    private:
        PSFDfgNODEID idtype;
        PSFCsel *csel;
        LNODE cfgnode;
        LNODE dfgnode; (attrib ???)
        int src_line_no;
        int alap_cstep;
        int asap_cstep;
        int cstep;
        int state;
        PSFTypeDescr *modtype;
        PSFInstanceDescr *inst;
        void* userdata;
}

```

Abb. 15: C++ Definition des **PSFDfgNode**

Die Datenelemente der Abbildung 15 erfüllen folgende Aufgaben.

**PSFDfgNODEID idtype**: Es existieren im Datenfluß verschiedene Arten von **PSFDfgNode**. Basic-Blocks werden durch **PSFDfgBBBeginNodes** und **PSFDfgBBEndNodes** gekapselt (siehe Abbildung 5). Zwischen diesen beiden Typen von **PSFDfgNodes** bestehen Verbindungen zwischen im wesentlichen Operatoren und Variablen. Diese verschiedenen Typenkennzeichnungen werden als Datenelement gespeichert. Die Definition ist aus dem Modul **PSFDfgNodeID\_def.H** zu entnehmen.

**PSFCsel\* csel**: Entsprechend des **idtype** sind unterschiedliche Informationen, wie Operortyp oder Variablenausprägung an den **PSFDfgNode** zu binden. Das Konzept der abgeleiteten **PSFCsels** ermöglicht dies effizient (siehe Abschnitt 2.2.3). Sämtliche knotenspezifische Informationen werden in einem solchen **PSFCsel** abgelegt und durch einen Verweis an den **PSFDfgNode** gebunden. Innerhalb eines **PSFDfg** treten Variablen i.a. häufiger als einmal auf. Es wird daher für eine Variable mehrere **PSFDfgNodes** geben, die auf dasselbe **PSFCsel** verweisen.

**LNODE cfgnode**: Dieser Rückverweis von **PSFDfgNode** zum entsprechenden **PSFCfgNode** wird nur von dem **PSFDfgBBBeginNode** gesetzt. Somit existiert eine 1:1 Korrespondenz zwischen dem **PSFDfg**-Basic-Block und dem **PSFCfgNode**.

**LNODE dfgnode**: Alle **PSFDfgNodes** eines Basic-Blocks verweisen auf den **PSFDfgBBBeginNode** desselben. Der **PSFDfgBBBeginNode** verweist mit diesem Zeiger auf den **PSFDfgBBEndNode** des Basic-Blocks.

**int src\_line\_no**: Nummer der Anweisung des Sourcecodes, welche durch diesen **PSFDfgNode** repräsentiert wird.

**PSFCdfg \* sub\_cdfg:** Bei existierendem hierarchischen **PSFCdfg** zeigt dieser Verweis auf den unterliegenden **PSFCdfg**.

Knoten innerhalb des Graphen werden durch Kanten verbunden. Die Definition der Kante für den Control-Flow-Graph ist in Abbildung 13 gegeben.

```
class PSFCfgEdge {
    public:
        PSFCfgEdge();
        ~PSFCfgEdge();
};
```

Abb. 13: C++ Definition **PSFCfgEdge**

Die Datenelemente der **PSFCfgEdge** sind bisher noch leer.

Daraus ergibt sich die Definition für den **PSFCfg** wie in Abbildung 14.

```
class PSFCfg : public LGRAPH (PSFCfgNode*,PSFCfgEdge*) {
    protected:
        PSFCdfg *cdfg;
    public:
        PSFDfg *GetDfg();
        // several graph methodes are implemented
}
```

Abb. 14: C++ Definition **PSFCfg**

**PSFCfg** ist eine abgeleitete Klasse des *LEDA* **LGRAPH**.

**PSFCdfg \*cdfg:** Verweis auf den zugehörigen **PSFCdfg**.

**PSFDfg \*GetDfg():** Diese Methode ermittelt den korrespondierenden **PSFDfg** zum **PSFCfg**.

In Anlehnung an *LEDA* sind verschieden Graphmethoden als `public`-member auf dem **PSFCfg** implementiert. Methoden, die **PSFCfgNodes** erzeugen oder diese durch ein **PSFCfgEdge** verbinden sind klassische Beispiele. Solche Methoden kapseln zum einen *LEDA*, so daß der **PSFCfg** Benutzer sich nicht um *LEDA*-Graphmethoden kümmern muß und zweitens existieren zusätzliche Methoden, die den **PSFCfg** behandeln. Dazu ist ein Blick in das Modul **PSFCfg.H** nötig, um die Flexibilität der Methoden zu erkennen. Diese Methoden sind in der Definition als Platzgründen nicht aufgeführt.

```

class PSFCfgNode {

    protected:

        PSFCFGNODEID idtype;
        LNODE dfgnode;
        LNODE exprnode;
        int src_line_no;
        LNODE nodepair;
        CfgStepCnt alap_cstep;
        cfgStepCnt asap_cstep;
        CfgStepCnt cstep;
        PSFCfg * sub_cdfg;
        void* userdata;

}

```

Abb. 12: C++ Definition **PSFCfgNode**

Die Datenelemente erfüllen folgende Aufgaben

**PSFCFGNODEID idtype**: Es existieren in dem Kontrollfluß verschiedene Strukturen. z.B. werden konditionale Verzweigungen, Basic-Blocks oder Schleifen dargestellt. Um diese verschiedenen Knoten zu kennzeichnen, wird der **PSFCfgNode** durch verschiedene Kennzeichnungen beschrieben, die aus dem Modul `PSFCfgNodeID_def.H` zu entnehmen sind.

**LNODE dfgnode**: Verweis auf den **LNODE** des **PSFDfgBBBeginNodes**, welcher zu diesem **PSFCfgNode** korrespondiert.

**LNODE exprnode**: Verweis auf **LNODE** des **PSFDfgOptorNode** welcher im **PSFDfg** das Ergebnis für eine konditionale Verzweigung auswertet. D.h. dieser Verweis ist für einen **PSFCfgIfNode** oder **PSFCfgWhileNode** gesetzt jedoch **NULL** für einen **PSFCfgBasicBlockNode**.

**int src\_line\_no**: Nummer der Anweisung des Sourcecodes, welche durch diesen **PSFCfgNode** repräsentiert wird.

**LNODE nodepair**: Verweis der bei konditionalen Verzweigungen zwischen den entsprechenden **PSFCfgNodes** gesetzt wird; sonst **NULL**.

**CfgStepCnt alap\_cstep**: Kontrollschritte, die bei dem Syntheseprozess (siehe Kap. 4) des ALAP-Scheduling des ermittelt werden.

**cfgStepCnt asap\_cstep**: Kontrollschritte, die bei dem Syntheseprozess des ASAP-Scheduling des ermittelt werden.

**CfgStepCnt cstep**: Hier werden die Nummer der Zustände für die zu synthetisierenden endlichen Automaten (siehe Kapitel 7) eingetragen.

Es wird das erweiterte Graphenpaket der *Library of efficient Data Types and Algorithms*, *LEDA* verwendet. *LEDA* erlaubt Templatemechanismen, welche effiziente Datenverarbeitung in Graphen ermöglicht. Ein *LEDA*-Graph besteht Knoten, `LNODEs` und Kanten, `LEGDEs`, welche den Graphen repräsentieren. Diese *LEDA*-spezifischen Knoten werden durch den Templatemechanismus mit verschiedenen Datenelementen attribuiert. Dieses wird sowohl für den `PSFCfg` als auch für den `PSFDfg` genutzt. Festzuhalten ist an dieser Stelle, daß sämtliche Graphenalgorithmien auf den Knoten und Kanten durch *LEDA* unterstützt werden.

Der `PSFCdfg` besteht aus zwei Graphen, die zueinander in enger Beziehung stehen. Ein Objekt der Klasse `PSFCdfg` beschreibt den Kontroll-Daten-Fluß einer Sourcecode-Funktion.

Die Klasse `PSFCdfg` ist wie in Abbildung 11 definiert:

```
class PSFCdfg {
    private:
        PSFDfg * dfg;
        PSFCfg * cfg;
        string name;
        LNODE first_cfg_node;
        LNODE first_dfg_node;
}
```

Abb. 11: C++ Definition `PSFCdfg`

Die Datenelemente der Klasse haben folgende Bedeutung:

`PSFDfg* dfg`: Verweis auf den `PSFDfg`, der durch den `PSFCdfg` dargestellt wird..

`PSFCfg* cfg`: Verweis auf den `PSFCfg`, der durch den `PSFCdfg` dargestellt wird.

`string name`: Name der Funktion oder des Programms, welches durch den `PSFCdfg` dargestellt wird.

`LNODE first_dfg_node`: Verweis auf den ersten `LNODE` des `PSFDfg`.

`LNODE first_cfg_node`: Verweis auf den ersten `LNODE` des `PSFCfg`.

Die Access-Methoden sowie die Konstruktoren sind hier nicht mit aufgeführt. Dies gilt aus Platzgründen auch für die folgenden Klassenbeschreibungen.

Jedes Element des `PSFCdfg` stellt genau eine Funktion dar. Eine Funktion besteht wiederum aus Kontroll- und Datenfluß. Die Datenstrukturen dazu, `PSFCfg` und `PSFDfg` werden in den beiden folgenden Abschnitten detailliert beschrieben.

## 2.2.1 Control-Flow-Graph - `PSFCfg`

Der Control-Flow-Graph besteht `PSFCfgNodes` und `PSFCfgEdges`. Die Deklaration der Klassen ist in der Abbildung 12 und Abbildung 13 aufgezeigt

In Abbildung 10 ist ein Ausschnitt aus dem **PSFDfg** des Submoduls dargestellt. Es sind der einzige Funktionsparameter `r65m36` (call-by-Reference) und eine lokale Variable `r65m40` erkennbar.

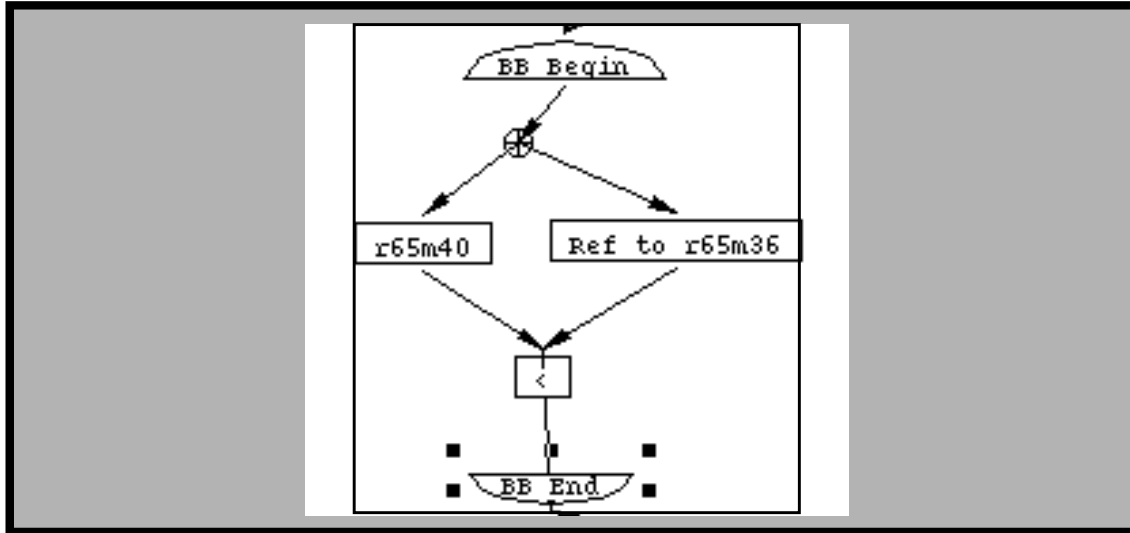


Abb. 10: Ausschnitt aus **PSFDfg** des Submoduls

Der **PSFCfg**-Teil des Submoduls entspricht exakt dem ursprünglichen Kontrollfluß. Der **PSFDfg** des Submoduls weist dagegen einige Veränderungen auf. Alle Knoten, die im Interface des aufrufenden Moduls als Reference-Knoten auftreten, erhalten nun den Knotentyp "Data-Reference-Node". Lokale Variablen und Value-Parameter sind wie gewohnt vom Typ "Data-Node".

Dieses Hierarchie-Konzept ist rekursiv aufzufassen, d.h. jeder **PSFDfg**-Datensatz ist als aufgerufenes Submodul zu betrachten. Die Schnittstelle des **PSFDfg** wird durch eine Liste der formalen Parameter definiert. Die Korrelation der aktuellen Parameter mit den formalen Parametern wird durch eine explizite Zuordnungsliste verwaltet. Diese Liste ist nur für einen Modulaufruf eindeutig. Es ergibt sich die Zugehörigkeit zum hierarchischen Knoten des aufrufenden **PSFDfgs**.

Für den Zugriff auf das Submodul sind Methoden des **PSFCdfg** implementiert. Sowohl der **PSFCfg** des Submoduls als auch der **PSFDfg** oder das Submodul an sich (**PSFCdfg**) kann vom aufrufenden Modul direkt angesprochen werden.

## 2.2 Technische Realisierung - PSFCdfg

Die technische Beschreibung des **PSFCdfg** wird in diesem Abschnitt erfolgen. Das PSF ist in C++ implementiert. Dies ermöglicht den Softwareansatz der Objektorientierung, welche in der Implementation konsequent verwirklicht wurde. Es werden nun die Datenstrukturen der einzelnen Komponenten vorgestellt.

duls. Ein ggf. vorhandener Rückgabewert-Wert wird durch einen Return-Knoten zwischen dem hierarchischen Knoten und dem End-Knoten angegeben.

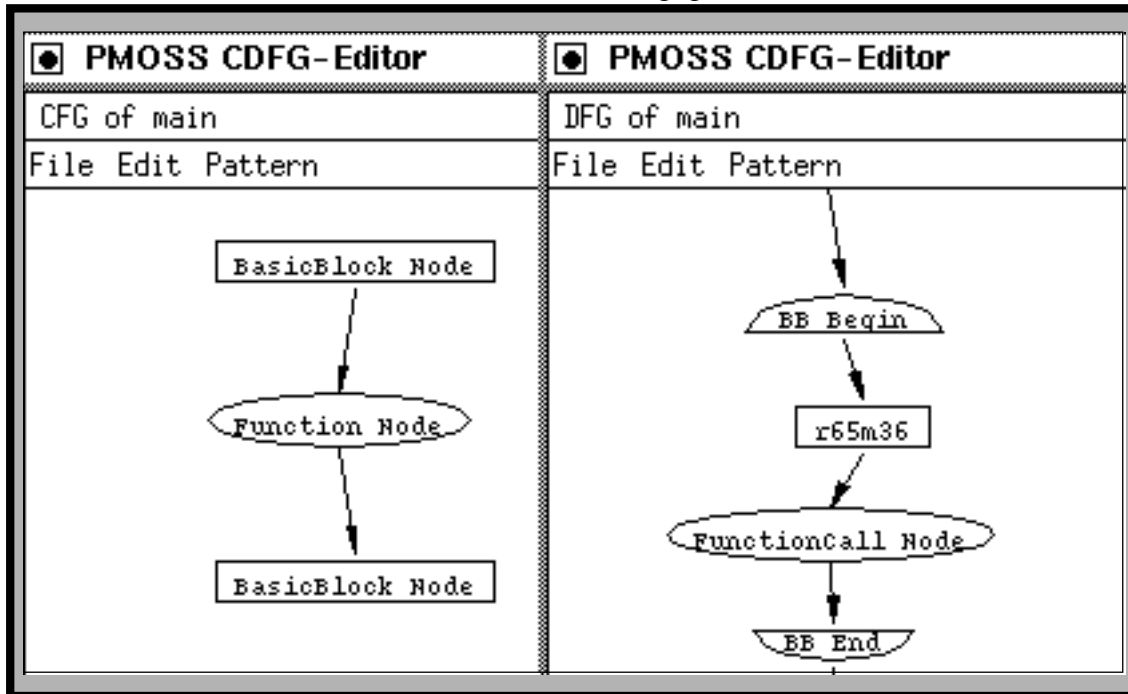


Abb. 9: Hierarchie-Konstrukt des **PSFCdfg** und des **PSFDFg**

Diese Knoten werden im zugehörigen Submodul wieder aufgeführt. Dabei ist zu unterscheiden zwischen

1. Call-by-Reference Variablen: alle Variablen dieser Kategorie werden in der Symboltabelle für globale Daten verwaltet.
2. Call-by-Value Variablen: Variablen dieses Typs werden in die Symboltabelle für lokale Daten eingetragen.
3. lokalen Variablen: diese Variablen werden weiterhin in der Symboltabelle für lokale Daten verwaltet.

- der Typ des Operators.
- Variablen; diese werden spezifiziert
  - nach der Anzahl der Bits, die diese Variable belegt,
  - der Name der Variable,
  - der Typ der Variable (array, integer,...).
- Konstanten; diese werden spezifiziert
  - nach der Anzahl der belegt Bits,
  - dem Wert der Konstanten,
  - Typ der Konstanten (Ganzzahlig, Buchstaben,..).

Für diese Knoten im **PSFDfg** werden entsprechende **PSFCsels** definiert.

Es wurde in dem einleitenden Kapitel erwähnt, daß die Graphen hierarchische Darstellung ermöglichen. Dies wird nun beschrieben.

#### 2.1.4 Hierarchie im PSFCdfg

Ein typisches Software Programm ist modular bzw. hierarchisch implementiert. Um dieser Spezifikationstechnik entgegenzukommen, wurde das folgende Hierarchiekonzept für den **PSFCdfg** entwickelt. Dieses spiegelt die Spezifikationsstrukturierung durch Funktionen wider. Im **PSFCfg** wird ein hierarchischer BasicBlock- Knoten angelegt und im **PSFDfg** wird der hierarchische Funktionsaufruf-Knoten durch Begin- und End-Knoten gekapselt angelegt. Wie in Abbildung 9 angedeutet, enthält der **PSFDfg** die Parameter (r65m36) des Submo-

Die anschließende Abbildung 8 soll die Repräsentation, von Datenabhängigkeiten innerhalb eines Feldes erklären. Diese ist der Datenabhängigkeitsdarstellung einzelnen Variablen im wesentlichen übernommen.

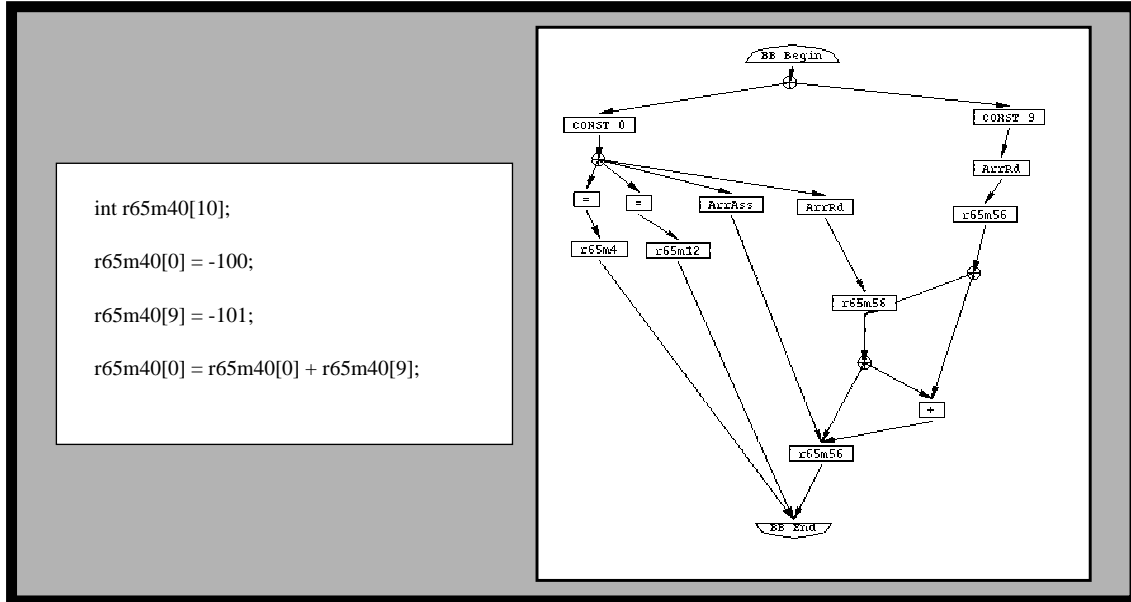


Abb. 8: Datenabhängigkeiten für arrays im **PSFDfg**

- Datenabhängigkeiten zwischen den Feldelementen werden so dargestellt, daß alle Feldelemente, über eine Kette von Datenabhängigkeitskanten verbunden sind. Dies geschieht unabhängig von dem Zugriff auf das Feldelement (lesen, schreiben) sondern von der Reihenfolge des Auftretens innerhalb des Source-Codes.

Es ist offensichtlich, daß ein Operator Knoten oder ein Operanden Knoten im **PSFDfg** verschiedene Daten zu seiner Beschreibung benötigt. Dazu wurde das Compilation Structured Element, **PSFCsel**, definiert, um dieser Beschreibungsanforderung gerecht zu werden.

### 2.1.3 Compilation Structured Element - **PSFCsel**

Als Datenstruktur im **PSFDfg** wird das Compilation Structured Element **PSFCsel** eingeführt. Innerhalb des **PSFDfg** werden Datenabhängigkeiten von Variablen über Operatoren dargestellt. Um die Variablen bzw. Operatoren zu spezifizieren, wird das **PSFCsel** als Basisklasse definiert und der Knoten des **PSFDfg** mit dem entsprechenden **PSFCsel** annotiert. Für spezielle Programmkonstrukte ergeben sich Ableitungen, die eine effiziente Beschreibung von Konstrukten, wie Funktionsaufrufen, variabler Struktur etc. ermöglichen.

In einem Programm existieren für den Datenfluß

- Operatoren; um diese zu spezifizieren ist nötig
  - auf welcher Anzahl von Bits ein Operator arbeitet,

Die Darstellung innerhalb des **PSFDfg** sollen die beiden folgenden Abbildungen verdeutlichen.

In Abbildung 7 liegt der Schwerpunkt auf den Referenzknoten zur Berechnung der einzelnen Feldelemente.

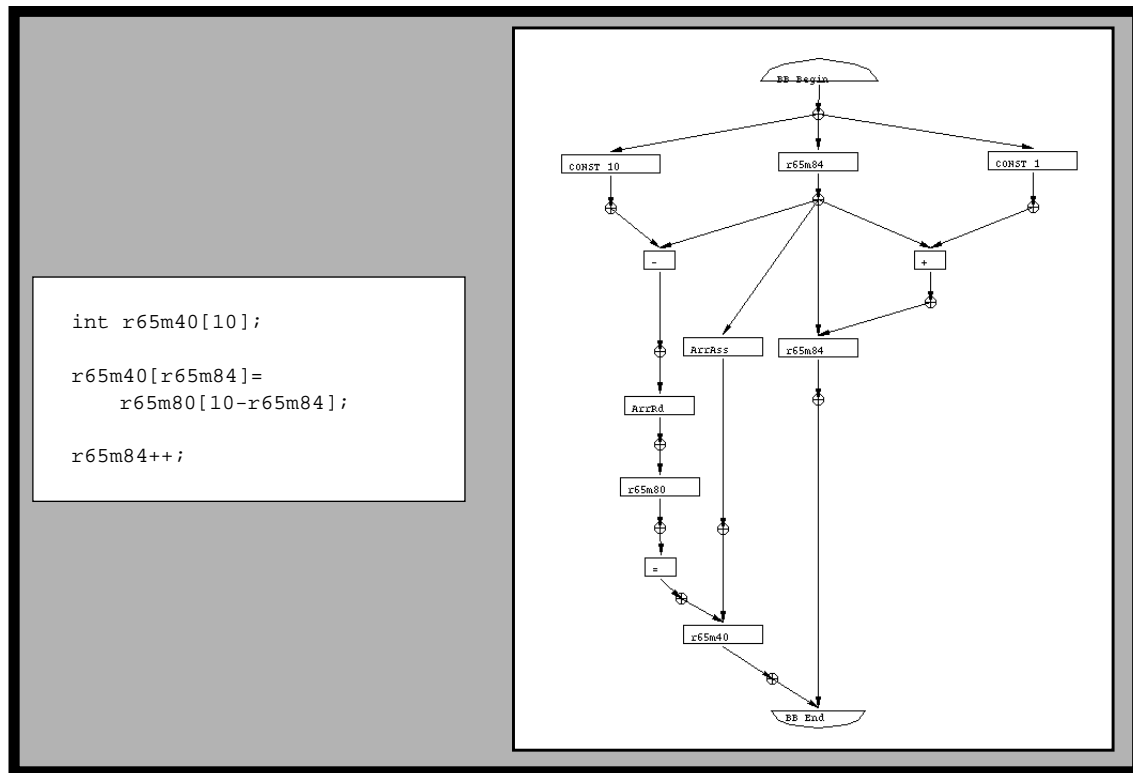


Abb. 7: Felder im **PSFDfg**

In Abbildung 7, die die Darstellung von Feldern in den **PSFDfg** zeigt, ist folgendes zu sehen.

- Die Knoten **Addr Read** und **Addr Ass** sind für die Berechnung der Feldelemente notwendig. An diesen Knoten wird die Berechnung der Feldelemente repräsentiert. Sie werden bei Lesen und Beschreiben eines Feldelements entsprechend angefügt.

### 2.1.2.2 Konditionale Verzweigungen im PSFDFg

Wie in Abschnitt 2.1.1.1 beschrieben, gibt es konditionale Verzweigungen in dem Kontrollfluß eines Programms. Dies schlägt sich im Datenfluß wie folgt nieder.

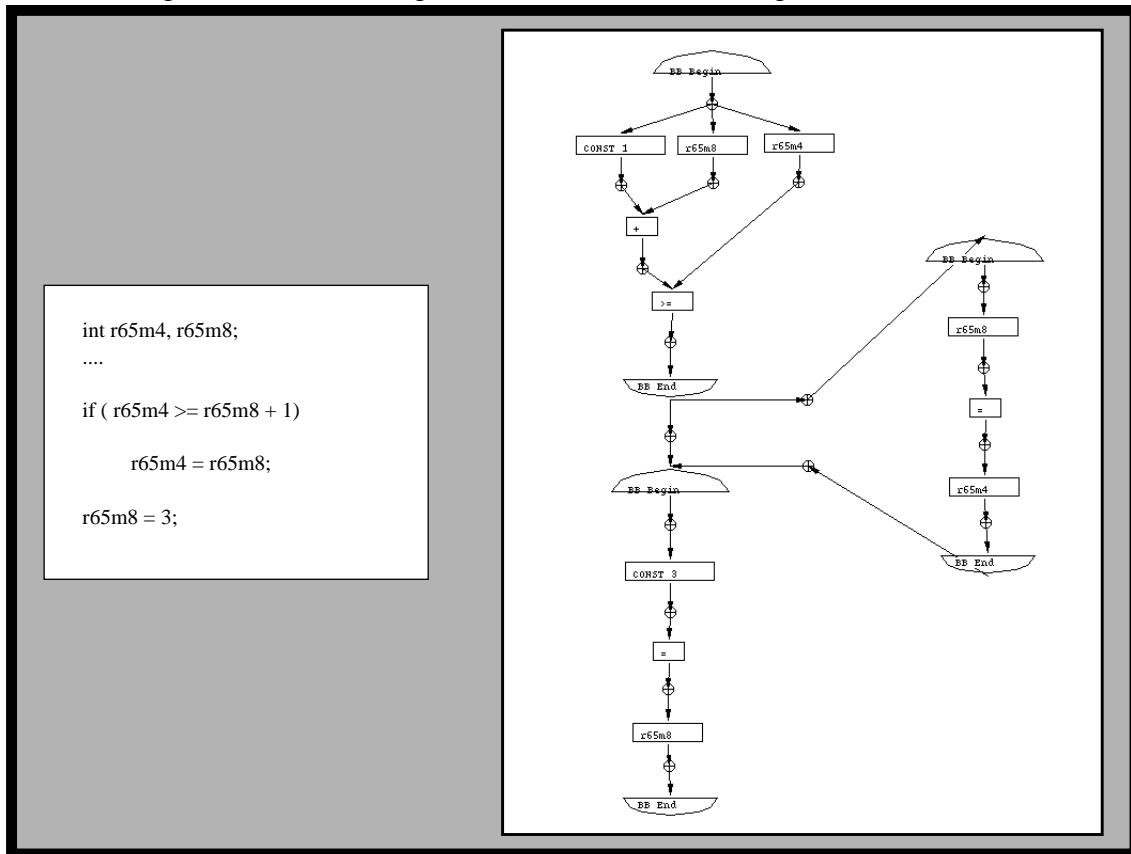


Abb. 6: if-then-else im PSFDFg

Abbildung 6 erklärt folgende allgemeingültige Konventionen.

- Bei einem Ausdruck wird der Operator, der das Ausdrucksergebnis ermittelt, mit dem `BB_End` verbunden (`>=`). Dadurch wird entschieden, wie der weitere Kontrollfluß aussieht.
- Operanden, die zu Ermittlung des Ausdruckswertes gelesen werden, sind mit dem `BB_Begin` verbunden (`Const 1`, `r65m8`, `r65m4`).
- Die `BB_Begin` der weiteren Programmteile sind mit dem `BB_End` des `if`-Ausdruck entsprechend des Kontrollfluß verbunden. Die Kantenreihenfolge, spielt im Gegensatz zum `PSFCdfg` (siehe Abschnitt 2.1.1.1) keine Rolle.

### 2.1.2.3 Felder im PSFDFg

Felder sind häufig auftretende Datenstrukturen. Datenabhängigkeiten und Referenzierung der einzelnen Feldelemente spielen für diese zusammengesetzten Strukturen eine primäre Rolle.

### 2.1.2.1 Zuweisungen im PSFDFg

Der einfachste Fall des Datenflusses, ist eine einfache Zuweisung. Sie wird ein einfaches Datenflußprinzip verdeutlichen.

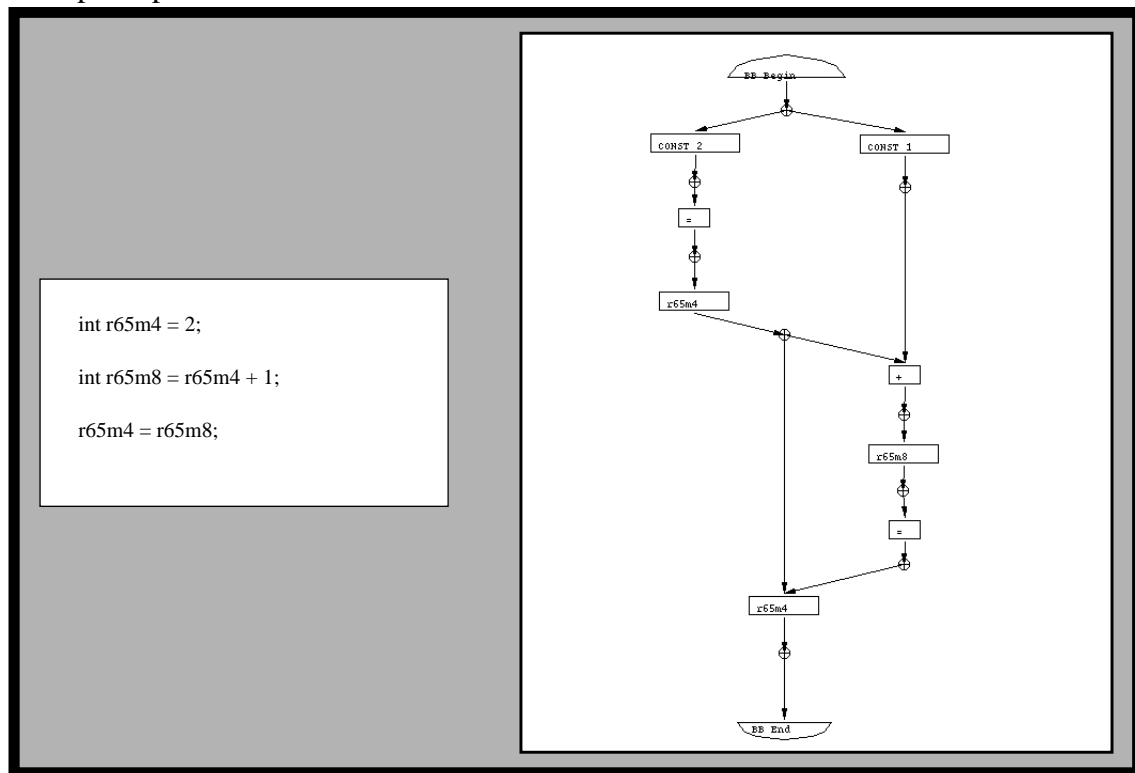


Abb. 5: Zuweisung im PSFDFg

Abbildung 5 soll folgende, allgemeingültige Prinzipien verdeutlichen.

- Ein Basic-Block wird durch die Knoten `BBegin` und `BBEnd` begrenzt.
- Operanden, die zu Beginn eines Basic-Block gelesen werden, sind mit dem `BBegin` durch Kanten verbunden (`Const 1`, `Const 2`).
- Operanden, die zum Ende eines Basic-Blocks beschrieben und nicht mehr innerhalb des Basic-Block gelesen werden, sind mit dem `BBEnd` verbunden (`r65m4`).
- Datenabhängigkeiten bestehen zwischen Operatoren und Operanden. Dies wird durch die Kanten des Graphen repräsentiert.
- Datenabhängigkeiten können auch zwischen Operanden bestehen. Wenn ein Operand innerhalb eines Basic-Blocks zuerst gelesen und dann beschrieben wird. Diese Sequenz wird durch eine Kante zwischen den beiden Knoten, die den Operanden darstellen, repräsentiert (`r65m4`).

Der entsprechende Kontrollfluß wird durch die Kanten zwischen den Knoten symbolisiert.

Der **PSFCfg** steht in enger Beziehung zum **PSFDfg**. Sein Prinzip wird in dem folgenden Abschnitt beschrieben.

### 2.1.2 Data-Flow-Graph - PSFDfg

Der Datenflußgraph soll die Datenabhängigkeiten eines Eingabeprogramms durch einen Graphen modellieren. Knoten und Kanten stellen die Abhängigkeiten zwischen Operatoren und Operanden dar. Die Knoten repräsentieren Operatoren und Variablen, Kanten den Datenfluß.

Die Idee des **PSFDfg** ist es, Datenabhängigkeiten innerhalb von Basic-Blocks darzustellen und diese Basic-Blocks zu verknüpfen. Die Sicht ist den Basic-Blocks aus der Compiler-Theorie übernommen. Das bedeutet, daß ein Basic-Block keine Sprünge oder Verzweigungen enthält und daß in einen Basic-Block nicht von außen hineingesprungen werden kann, um nur einen Teil der Anweisungen zu durchlaufen. Datenabhängigkeiten, die über Basic-Block-Grenzen hinausgehen, ergeben sich durch gleiche Registernamen und **PSFCsels**(siehe Abschnitt 2.2.3).

Eine weitere Information ist die Korrespondenz zwischen einem Knoten des **PSFDfg** und der entsprechenden Anweisung im Code bzw. Knoten im **PSFCfg**.

Die Darstellung verschiedener Datenflüsse soll nun anhand einiger Beispiele verdeutlicht werden.

- Der Ausdruck des `if`-Konstrukts des Quell-Codes, welcher über den weiteren Programmfluß entscheidet, wird durch den `If Node` dargestellt.
- Durch Analyse des Ausdrucks ergibt sich, welcher der `BasicBlock Nodes` durchlaufen wird. Die Entscheidung darüber fällt im korrespondierenden Basic-Block des Data-Flow-Graphs. (siehe auch Abbildung 6). Die auslaufenden Kanten werden durchnummeriert, wobei die erste Kante immer der `true`-Pfad repräsentiert
- Am Ende der Verzweigung steht der `End If Node` welche beide Kontrollflußstränge miteinander verknüpft. Der weitere Kontrollfluß schließt sich an.

### 2.1.1.2 Schleifen im PSFCdfg

Schleifen, sind Konstrukte, die essentiell zur effektiven Systembeschreibung sind. Deren Abbildung auf den Control-Data-Flow-Graph wird nun beschrieben.

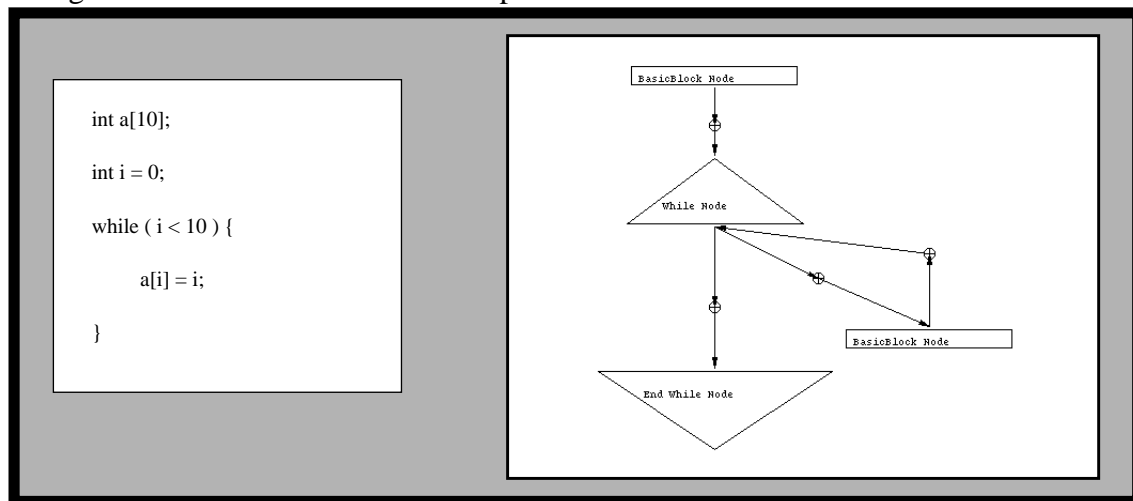


Abb. 4: while - Schleifen im PSFCdfg

In Abbildung 4 sind folgende Programmteile zu beachten.

- Die Initialisierung wird im **PSFCdfg** durch den ersten `BasicBlock Node` dargestellt.
- Der Ausdruck, welcher über den Programmfluß entscheidet, wird durch den `While Node` repräsentiert.
- Wird der Ausdruck zu `true` evaluiert, wird der Rumpf der Schleife durchlaufen. Dieser wird durch den `BasicBlock Node` dargestellt. Er enthält einen Rückverweis zur wiederholten Auswertung des Ausdrucks.
- Bei nicht erfülltem Ausdruck ist das Schleifenende erreicht (`EndWhile Node`). Der weitere Kontrollfluß beginnt am Ende der Schleife.

Der Kontrollfluß eines System wird durch einen Graphen dargestellt. Dieser wird im folgenden Abschnitt erläutert.

### 2.1.1 Control-Flow-Graph - PSFCdfg

Der **PSFCdfg** ist ein gerichteter Graph. Er besteht aus zwei disjunkten, endlichen Mengen von Knoten und Kanten. Der **PSFCdfg** modelliert den Programmablauf. Zur Darstellung der Programmabschnitte sind Knoten im **PSFCdfg** definiert. Ein solcher Knoten wird gekennzeichnet, um eine Unterscheidung der verschiedenen Programmkonstrukte zu ermöglichen. Dadurch wird gewährleistet, daß für bestimmte Sourcecode-Ablaufstrukturen (z.B. Schleifen) entsprechende Objekte innerhalb des **PSFCdfg** zur Verfügung stehen. Kanten zwischen den Knoten stellen den Programmfluß dar. Ein Knoten im **PSFCdfg** stellt immer eine Programmsequenz dar, welche immer nach ihrem Eintritt vollständig durchlaufen wird.

Verschiedene Konstrukte der Spezifikationsprache werden in entsprechenden **PSFCdfg**-Elementen repräsentiert. Dies wird nun an zwei Beispielen aufgezeigt.

#### 2.1.1.1 Konditionale Verzweigung im PSFCdfg

Es ist klar, daß ein Programm verschiedene Programmflüsse besitzt. Durch Ermitteln von Booleschen Ausdrücken, wird über den Programmfluß entschieden. Die Darstellung innerhalb des **PSFCdfg** soll nun erläutert werden.

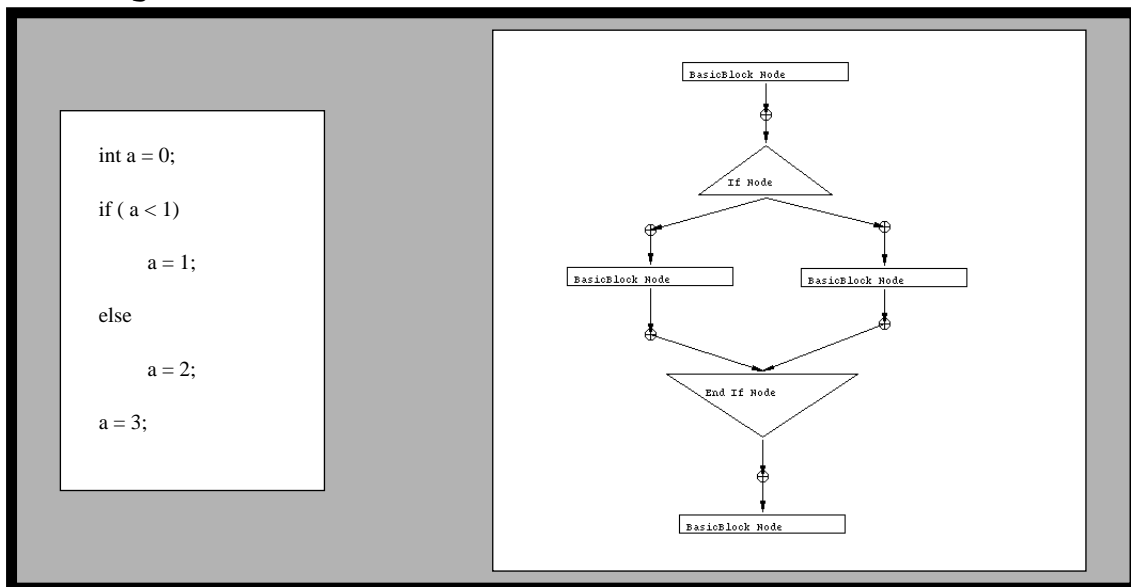


Abb. 3: if-then-else im **PSFCdfg**

Das Beispiel in Abbildung 3 soll die Abbildung von Programmkonstrukten auf die Strukturen des **PSFCdfg** erläutern.

- Die Initialisierung wird im **PSFCdfg** durch den ersten BasicBlock Node dargestellt.

## 2 Control-Data-Flow-Graph - PSFCdfg

Das PSF umfaßt zwei Ebenen mit unterschiedlichem Abstraktionsgrad. Die Ebene der höheren Abstraktion, der Control-Data-Flow-Graph **PSFCdfg** soll hier beschrieben werden. Der Control-Data-Flow-Graph besteht im wesentlichen aus zwei hierarchischen Graphen, die zueinander in enger Beziehung stehen.

Der Control-Data-Flow-Graph bildet die Verhaltensbeschreibung der zu synthetisierenden Systembeschreibung auf zwei hierarchische Graphen ab. Ein System ist in einer beliebigen Sprache spezifiziert, welche Kontrollstrukturen und Datenabhängigkeiten beschreiben kann.

Der **PSFCdfg** stellt die entsprechenden Verweise zwischen Kontrollstrukturen und Datenabhängigkeiten dar.

Der Aufbau dieser Beschreibung ist wie folgt:

1. Die Darstellung eines Programms durch den **PSFCdfg** wird abstrakt beschrieben.
2. Die technische Realisierung des **PSFCdfg** schließt sich daran an.

### 2.1 Abstrakte Beschreibung - PSFCdfg

Dieser Abschnitt soll das Prinzip des **PSFCdfg** erläutern. Es soll die Darstellungsweise eines Control-Data-Flow-Graphen verdeutlichen. Die Darstellung ist software-orientiert, welches für den High-Level-Synthese Prozeß, sowie für HW/SW-Codesign auf System-Level Vorteile bietet.

Die hierarchische Darstellung ist nicht für die einzelnen Graphen unabhängig zu betrachten. Daher wird die Darstellung der Hierarchie explizit in Abschnitt 2.1.4 erläutert.



Der Dateienbaum hat die Struktur wie in Abbildung 2.

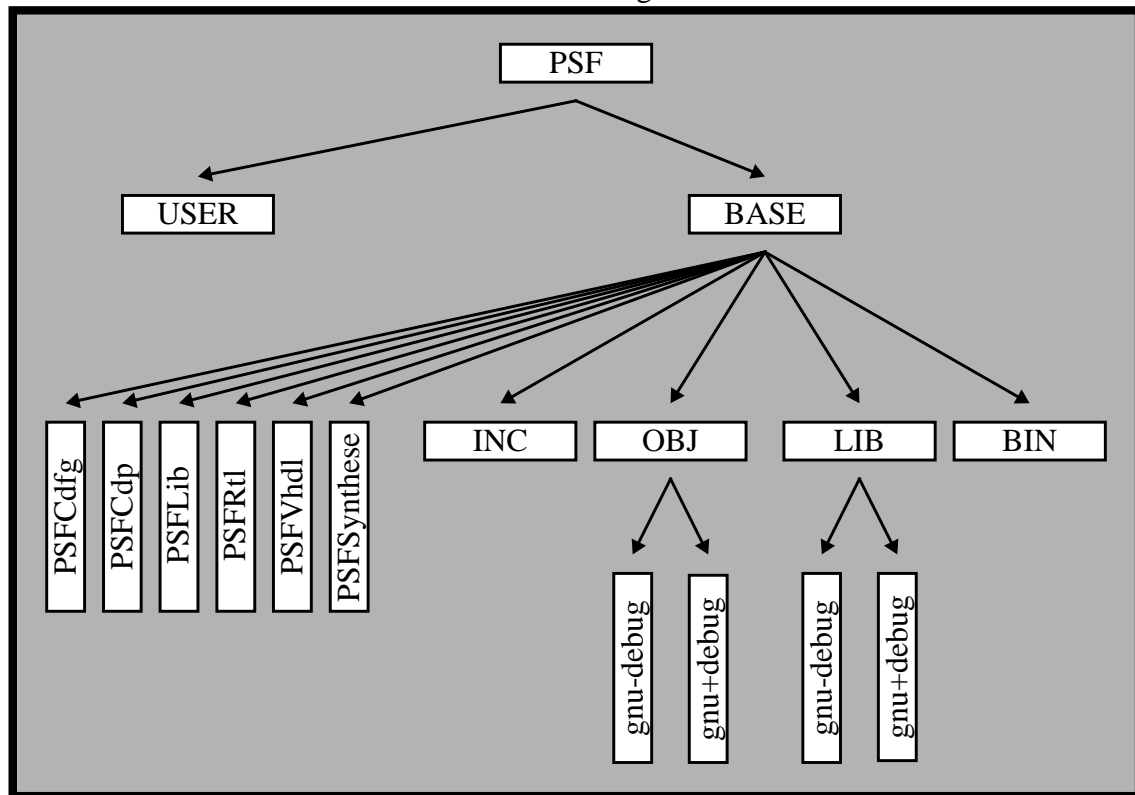


Abb. 2: Dateienbaum des PSF

Jedes Modul, welches sich in dieser Beschreibung wiederfindet, enthält einen Zweig des Baumes unter dem sich die Source- und Header-Dateien befinden. Weiterhin gibt es einen zentralen INC-Ordner, welcher alle Header-Dateien, des PSF beinhaltet. Die erzeugten Object-Dateien stehen entsprechend ihres Übersetzungsmodus (mit und ohne Debug-Information) unter dem OBJ-Ordner. Die Funktionalität der einzelnen Module ist in Bibliotheken zusammenfaßt. Ebenfalls wird auch hier zwischen Bibliotheken in dem Verzeichnis LIB mit und ohne Debug-Information durch entsprechende Unterverzeichnisse unterschieden. Zuletzt existiert ein BIN-Ordner, der die ausführbaren Programme beinhaltet. Unter dem Pfad USER, kann ein Benutzer sich seine eigene Entwicklungs- oder Testumgebung schaffen, ohne die Referenzversion zu stören.

Dadurch wird die Simulation des synthetisierten Hardwaresystems ermöglicht. Die folgende Abbildung 1 soll den Aufbau des PSF verdeutlichen.

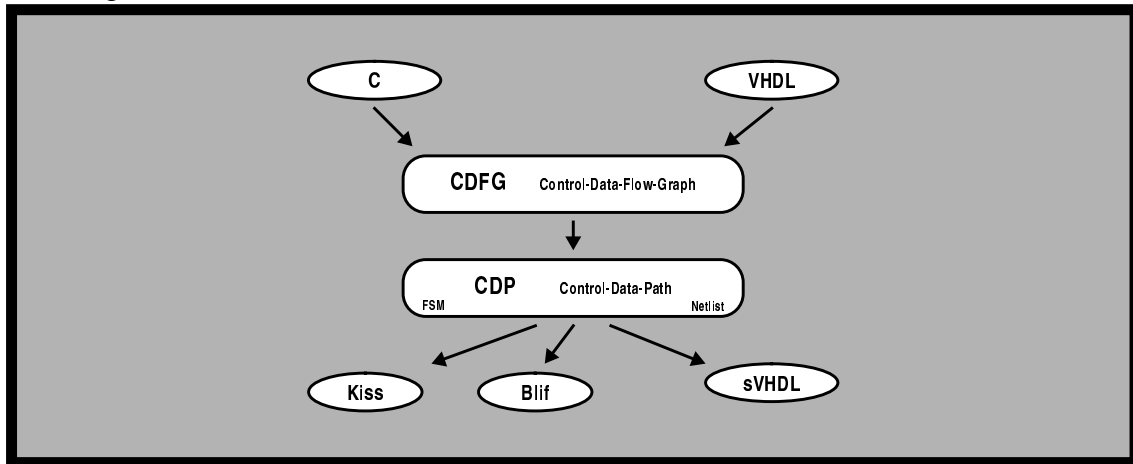


Abb. 1: Paderborn Synthesis Format

Der Control-Data-Flow-Graph ist eine verhaltensorientierte Darstellung des Systems. High-Level-Applikationen der Synthese werden auf ihm implementiert. Der Control-Data-Path ist eine strukturorientierte Darstellung des Systems.

Die Beschreibung richtet sich an Benutzer des Formates, um einen schnellen Überblick über das PSF zu bekommen. Es sollen nicht die Algorithmen der High-Level-Synthese erläutert werden, sondern nur deren Verwendeten innerhalb des implementierten Synthesekonzeptes.

## 1.1 Organisation

Das System ist auf einer UNIX-Plattform in C++ implementiert. Der relevante Compiler ist der GNU CC-2.4.5.

Die einzelnen PSF Systemkomponenten sind in Modulen zusammengefaßt. Aus Gründen der Übersichtlichkeit, wurde die Software und die Dateien strukturiert. Es wird von einer Wurzel ausgegangen, von der aus sich nach der Organisation und Übersichtlichkeitesgründen Teilbäume für Object-Dateien oder Bibliotheken anschließen.

# 1 PSF - Paderborn Synthesis Format

High-Level-Synthese, umfaßt die Umsetzung einer beliebigen algorithmischen Spezifikation in Register-Transfer-Strukturen. Dieser Transfer wird durch mehrere Algorithmen wie, Scheduling, Allocation, Interconnection Binding realisiert. Es gibt verschiedene Spezifikationsmöglichkeiten um ein System zu beschreiben. Die interne Darstellung derselben sollte aber für die High-Level-Synthese von der Spezifikationssprache unabhängig sein. Für HW/SW-Codesign ist eine ähnliche, einheitliche Darstellung von Vorteil.

Mit dem *Paderborn Synthesis Format, PSF*, wird die Idee eines einheitlichen Syntheseformates zur Datenrepräsentation für die High-Level-Synthese verwirklicht. Die Notwendigkeit einer einheitlichen Datendarstellung ist in mehreren Publikationen beschrieben worden.

Das PSF besteht im wesentlichen aus zwei Abstraktionsebenen, die beide in enger Beziehung stehen. Auf höherer Ebene wird das Verhalten des Systems als Kontroll-Daten-Flußgraph modelliert. Die niedere Ebene modelliert die Struktur des Systems als Kontroll-Datenpfad durch eine Netzliste von Modulen und einen endlichen Automaten. Der Informationsgehalt beider Ebenen ist gleich, was eine 1:1 Konvertierung zwischen den Ebenen erlaubt. HW/SW-Codesign ist so auf verschiedenen Abstraktionsgraden möglich.

Durch zwei Front-Ends ist es möglich, Spezifikationen verschiedener Sprachen in das PSF zu konvertieren. Applikationen auf den beiden Ebenen vollführen die Synthese in eine Hardware-Beschreibung. Back-Ends erzeugen Formate, die den Anschluß an bereits existierende Systeme erlauben. Über die Schnittstellenformate Blif/Kiss kann das Logiksynthesesystem SIS die Synthese auf Logikebene weiterführen. Es ist daher nicht erforderlich die Algorithmen der Logiksynthese auf dem PSF zu implementieren. Die Ausgabe eines strukturellen Vhdl-Dialekts erlaubt den Anschluß des kommerziellen Design Analyzer von Synopsys Inc.



---

<b>5</b>	<b>C/C++-Front-End .....</b>	<b>61</b>
5.1	Register-Transfer-Language RTL .....	63
5.2	Technische Realisierung des C/C++ Front-Ends .....	65
5.2.1	Zwischenstruktur - RTLAssignElem .....	66
5.3	Verwendung des C/C++-Front-End .....	69
5.3.1	Zustand des Konverter im Dezember 1993: .....	69
5.3.2	Beschränkung bei der Spezifikation von C/C++ .....	69
5.3.3	Einlesen einer C/C++ Spezifikation in das PSF .....	70
<b>6</b>	<b>VHDL-Front-End .....</b>	<b>71</b>
6.1	Unterstützte VHDL-Sprachelemente .....	71
6.1.1	Befehle .....	71
6.1.2	Datentypen .....	71
6.1.3	Operatoren .....	71
6.2	Implementierung des Konverters .....	72
6.2.1	Konzept .....	72
6.2.2	Module .....	73
6.2.2.1	Konverter .....	73
6.2.2.2	Declarations .....	73
6.2.2.3	Statements .....	73
6.2.2.4	Expressions .....	75
6.2.2.5	Lists .....	75
6.2.3	Schnittstelle zum PSF-Graph .....	76
<b>7</b>	<b>Die High Level Synthese im PSF .....</b>	<b>81</b>
7.1	Die Architektur des zu synthetisierenden Systems .....	81
7.1.1	Synthese von Arrays .....	82
7.2	High Level Synthese .....	83
7.3	Das Konzept der virtuellen Netzlisten .....	83
7.3.1	Deskriptoren für Modultypen .....	84
7.3.2	Deskriptoren für Modulinstanzen .....	85
7.4	Die Klasse PSFSynthese .....	87
7.4.1	Member von PSFSynthese .....	87
7.4.2	Methoden von PSFSynthese .....	87
7.4.2.1	StaticListScheduler .....	87
7.4.2.2	SimpleFUBinding .....	88
7.4.2.3	LifetimeAnalysis .....	88
7.4.2.4	SimpleRegisterAllocationAndBinding .....	88
7.4.2.5	InterconnectionBinding .....	88
7.4.2.6	CreateNet .....	88

---

3.3.3.2	Hierarchische Operationen/Funktionen .....	41
3.3.3.3	Interfaces .....	43
3.3.4	Abhängigkeiten .....	47
3.3.5	Benutzung der Klasse .....	47
3.3.6	Probleme .....	49
<b>4</b>	<b>PSFLib .....</b>	<b>51</b>
4.1	Beschreibung der Funktion der Klasse PSFLib .....	51
4.2	Beschreibung der Bibliothekselemente .....	52
4.2.1	Arithmetische Komponenten .....	53
4.2.1.1	PLUS .....	53
4.2.1.2	MINUS .....	53
4.2.1.3	MULT .....	53
4.2.1.4	DIV .....	54
4.2.1.5	MOD .....	54
4.2.1.6	ALU1 .....	54
4.2.2	Kontroll Strukturen .....	54
4.2.2.1	OneHot2Bin .....	55
4.2.2.2	Bin2OneHot .....	55
4.2.2.3	MUX .....	56
4.2.2.4	DEMUX .....	56
4.2.2.5	CONST .....	56
4.2.2.6	REG .....	56
4.2.2.7	JOIN .....	56
4.2.2.8	SPLIT .....	57
4.2.2.9	SWAP .....	57
4.2.3	Komparatoren .....	57
4.2.3.1	EQ .....	58
4.2.3.2	NE .....	58
4.2.3.3	LE .....	58
4.2.3.4	LT .....	58
4.2.3.5	GE .....	58
4.2.3.6	GT .....	58
4.2.3.7	RiCompare .....	58
4.2.3.8	RiLE .....	58
4.2.3.9	RiLT .....	58
4.2.3.10	RiGE .....	58
4.2.3.11	RiGT .....	58
4.2.4	Logische Elemente .....	59
4.2.4.1	AND .....	59
4.2.4.2	OR .....	59
4.2.4.3	XOR .....	59
4.2.4.4	NOT .....	59
4.2.4.5	SHL .....	59
4.2.4.6	SHR .....	60

---

<b>1</b>	<b>PSF - Paderborn Synthesis Format</b>	<b>5</b>
1.1	Organisation	6
<b>2</b>	<b>Control-Data-Flow-Graph - PSFCdfg</b>	<b>9</b>
2.1	Abstrakte Beschreibung - PSFCdfg	9
2.1.1	Control-Flow-Graph - PSFCfg	10
2.1.1.1	Konditionale Verzweigung im PSFCfg	10
2.1.1.2	Schleifen im PSFCfg	11
2.1.2	Data-Flow-Graph - PSFDfg	12
2.1.2.1	Zuweisungen im PSFDfg	13
2.1.2.2	Konditionale Verzweigungen im PSFDfg	14
2.1.2.3	Felder im PSFDfg	14
2.1.3	Compilation Structured Element - PSFCsel	16
2.1.4	Hierarchie im PSFCdfg	17
2.2	Technische Realisierung - PSFCdfg	19
2.2.1	Control-Flow-Graph - PSFCfg	20
2.2.2	Data-Flow-Graph - PSFDfg	23
2.2.3	Compilation Structured Element - PSFCsel	26
2.2.3.1	Definition PSFVarCsel	27
2.2.3.2	Definition PSFTypeCsel	28
2.2.3.3	Definition PSFConstCsel	29
2.2.3.4	Definition PSFOptorCsel	30
2.2.4	Definition PSFFunCsel	30
2.2.5	Implementation des Hierarchiekonzepts	31
<b>3</b>	<b>PSFCdp</b>	<b>33</b>
3.1	Boolesche Formeln - PSFBoolFormula	33
3.1.1	Kontext und Aufgabe der Klasse	33
3.1.2	Definition	33
3.1.3	Abhängigkeiten	34
3.1.4	Benutzung der Klasse	34
3.1.5	Probleme	34
3.2	Finite State Machines - PSFFSM	34
3.2.1	Kontext und Aufgabe des Moduls	35
3.2.2	Definition	35
3.2.3	Abhängigkeiten	36
3.2.4	Benutzung der Klasse	36
3.2.5	Benutzung des Editors (FSMEditor)	37
3.2.6	Probleme	38
3.3	Netzlisten - Zellen (PSFCell)	38
3.3.1	Kontext und Aufgabe des Moduls	39
3.3.2	Definition	40
3.3.3	Realisierung	40
3.3.3.1	Grundfunktionen	41





UNIVERSITÄT  
GESAMTHOCHSCHULE PADERBORN

Fachbereich 17 • Mathematik - Informatik

# PSF

## Paderborner Synthese Format

---

**Version 1.0**

**Universität Paderborn,  
Fachbereich 17,  
Warburger Strasse 100,  
D-33095 Paderborn**

**Andreas Hoffmann, Heinz-Josef Eikerling, Reiner Genevriere, Wolfram Hardt**