Diplomarbeit

Karl Michael Göschka

Firmware Compiler Generation

Ein Ansatz zur Generierung des Backends von Firmwarecompilern

Technische Universität Wien
Institut für Computertechnik
Gusshausstrasse 27–29
A–1040 Wien
Tel: (+43-1) 58801/3822

Siemens AG Österreich
Programm– und Systementwicklung 13
Geusaugasse 17
A–1030 Wien
Tel: (+43-1) 71600/301

DIPLOMARBEIT

Firmwarecompiler Generation

Ein Ansatz zur Generierung des Backends von Firmwarecompilern

ausgeführt am Institut für Computertechnik
der Technischen Universität Wien

unter der Anleitung von o.Univ.Prof. Dr. Richard Eier
und Univ.Ass. Dipl.-Ing. Klemens Urban
als verantwortlich mitwirkendem Universitätsassistenten

durch

Dipl.-Ing. Karl Michael Göschka
Karl–Meißlstraße 7/17, 1200 Wien
Matr.Nr. 8625510

Wien, März 1995 _____

FCG — *Firmwarecompiler Generation*

Ein Ansatz zur Generierung des Backends von Firmwarecompilern.

Kurzfassung

Die Entwicklung mikroprogrammierter Hardware erfordert die parallele Entwicklung von Hardware und Firmware. Der Flaschenhals im daraus resultierenden Entwicklungszyklus ist die „händische“ Anpassung des Firmwarecompilers durch den Compilerentwickler: Jede Änderung an der Hardware erfordert eine Anpassung des Firmwarecompilers, um mit Hilfe der Firmwareprogramme die veränderte Hardware testen zu können.

Die Diplomarbeit zeigt einen Ansatz zur *Generierung* des Firmwarecompilers: Dabei können Frontend und Intermediate Representation als unabhängig von der Hardware angenommen werden, zumindest für eine bestimmte Familie ähnlicher Prozessoren. Zur Erzeugung des Backends hingegen benötigt man zwei wesentliche Eingaben: Erstens eine Hardwarebeschreibung, aus der jene Konflikte abgeleitet werden können, die durch Mehrfachnutzung von Hardwareressourcen entstehen. Zweitens eine formale Beschreibung der Backend-Funktionen, die es ermöglicht, die semantischen Aktionen und die Codeerzeugung auf die Attribute der Intermediate Representation zugreifen zu lassen.

Die erste Anforderung wurde bereits in früheren Arbeiten durch Entwicklung der deskriptiven Hardwarebeschreibungssprache MDL/HD[1] erfüllt: Diese beschreibt die Hardwareressourcen eines Prozessors sowie die Verwendung dieser Ressourcen in den *Mikrooperationen*. Die Sprache ist dabei effizient, flexibel und großzügig, sodaß sie im Rahmen eines Entwurfswerkzeuges Verwendung finden kann. Zu dieser Sprache existiert ein sehr robuster und fehlerstabiler *Analysator-Prototyp*, dessen Aufgabe es ist, eine MDL–Beschreibung einzulesen und daraus eine Tabelle aller *Hardwarekonflikte* zu erzeugen.

Trotz des dadurch gewonnenen Wegfalls der mühevollen Suche nach hardwarebedingten *Konflikten* bleibt jedoch die ineffiziente Formulierung der *semantischen Prüfungen* und der *Codeerzeugung*. Um das Backend des Firmwarecompilers und die prozessorspezifische *Sprachuntermenge* der Mikroprogrammsprache automatisch erzeugen zu können, wird die bestehende MDL/HD um die MDL/BED[2] erweitert. Damit wird die Zuordnung der Attribute der Intermediate Representation sowohl zu den Mikrooperationen zwecks Konfliktprüfung als auch zur Codegenerierung ermöglicht.

Darüberhinaus scheint es durchaus möglich zu sein, die zugrundeliegenden Methoden auch im Bereich des Hardware/Software-Codesign einsetzen zu können. Zu diesem Zweck müssen die bereits bestehenden Backend-Beschreibungssprachen durch einen Hardwareteil erweitert werden, sodaß eine automatische Erzeugung der Compiler-Optimierungen möglich ist. Umgekehrt eröffnet die Generierbarkeit des Firmwarecompilers neue Möglichkeiten im Bereich der High Level Synthese, wobei aufbauend auf einer abstrakten Beschreibung des Controllers die Firmware mit einem impliziten Compiler direkt generiert werden kann.

[1] Microcompiler Design Language/Hardware Description.

[2] MDL/Backend Description.

Danksagung

Besonderer Dank gebührt Herrn Johann Aschenbrenner, der auch nach Beendigung meiner Ferialpraxis bei der Firma Siemens stets Zeit für produktive Diskussionen erübrigen konnte.

Danken möchte ich auch Herrn Prof. Dr. Richard Eier, der es ermöglicht hat, daß ich neben der Arbeit als Universitätsassistent am Institut für Computertechnik auch mein zweites Studium beenden konnte.

Ich bedanke mich auch bei meinem Kollegen Herrn Dipl.-Ing. Klemens Urban, der — mehr Freund als nur Betreuer — die Mühe auf sich nahm, meine Arbeit kritisch zu durchleuchten.

Mein Dank gilt auch Frau Erika Göschka und Fräulein Gabriele Grossmüller für das Opfer vieler Stunden Korrekturlesens. Meinen Eltern Albert und Erika Göschka schließlich gebührt der allergrößte Dank, da sie mein Studium überhaupt erst ermöglicht haben.

Inhaltsverzeichnis

1 Der Firmwarecompiler im Hardware-Entwicklungsprozeß 1

1.1 Einleitung 1

1.2 Vorangegangene Arbeiten 8

Die Sprache MDL/HD und der Analysator 8

Konsequenzen 14

1.3 Lösungsansatz für die Compilergenerierung 16

1.4 Anforderungen 20

2 Spracherweiterung und Generierung 23

2.1 Frontend und Intermediate Representation 23

Intermediate Representation 23

Frontend 29

2.2 MDL-Spracherweiterung durch Baumgrammatik 31

2.3 Attribute und Semantische Aktionen 36

Konfliktprüfung und MDL/C-Schnittstelle 41

Codeerzeugung und relozierbare Module 43

2.4 Gesamtstruktur und Implementierung 46

Implementierung 46

Überprüfung der Entwurfsziele 46

3 Weiterführende Konzepte und Zukunft 50

3.1 Der allgemeine Entwicklungsprozeß 50

3.2 Hardware-Software Codesign 53

3.3 High Level Synthese 57

3.4 Ausblick und Schluß 61

Anhang

A Lexikalische Regeln und Syntax **62**

A.1 Lexikalische Regeln 62

A.2 Syntax 63

B Baumgrammatik und semantische Aktionen **66**

B.1 Baumgrammatik 66

B.2 Backend Programmfragment 73

Verzeichnisse

Abbildungsverzeichnis **V-1**

Tabellenverzeichnis **V-2**

Beispielverzeichnis **V-3**

Literaturverzeichnis **V-4**

Kapitel 1

Der Firmwarecompiler im Hardware-Entwicklungsprozeß

1.1 Einleitung

Die Entwicklung mikroprogrammierter Prozessoren erfordert die parallele Entwicklung von Hardware und Firmware, was insofern grundsätzlich problematisch ist, als die beiden einander sehr stark beeinflussen. Zum grundlegenden Verständnis zeigt Abbildung 1.1 den ursprünglichen Entwicklungsprozeß vor Einführung des Hardwaresimulators.

Der *Hardwareentwicklungszyklus*, links im Bild, ist geprägt von der Arbeit des Hardwareentwicklers an seinem Laboraufbau: Die Schaltung wird real aufgebaut, anschließend wird die Reaktion auf verschiedene Eingangssignale mit geeigneten Meßgeräten aufgenommen. Der Hardwareentwicklungszyklus wird hier bewußt auf den für die Aufgabenstellung der Diplomarbeit wesentlichen Teil reduziert, das ist das Testen der Hardware mit einem lauffähigen Firmwareprogramm. Die anderen Teile des Hardwareentwicklungszyklus, wie etwa das Entwerfen der Schaltung in Hinblick auf Integration, Partitionierung und Fertigung, wurden bewußt weggelassen, da sie vom Hardwareentwickler allein durchgeführt werden und daher hier nicht Gegenstand der Betrachtung sind.

Als *Firmwareentwicklungszyklus*, rechts im Bild, wird die Arbeit des Firmwareentwicklers am Compiler bezeichnet: Die eingegebenen Mikrobefehle werden übersetzt und entsprechend den ausgegebenen Meldungen solange verbessert, bis eine lauffähige Version zustande kommt.

Es gibt nun zwei Verbindungen zwischen dem Hardwareentwicklungszyklus und dem Firmwareentwicklungszyklus: Die eine besteht in der Umsetzung des vom Compiler erzeugten Bitmusters in einen geeigneten Code, der schließlich in ein EPROM[1] einprogrammiert wird. Dieses wird im Laboraufbau vom Hardwareentwickler verwendet, um seine Probeschaltung mit lauffähigen Mikroprogrammen zu testen.

[1] *Eraseable Programable Read Only Memory*

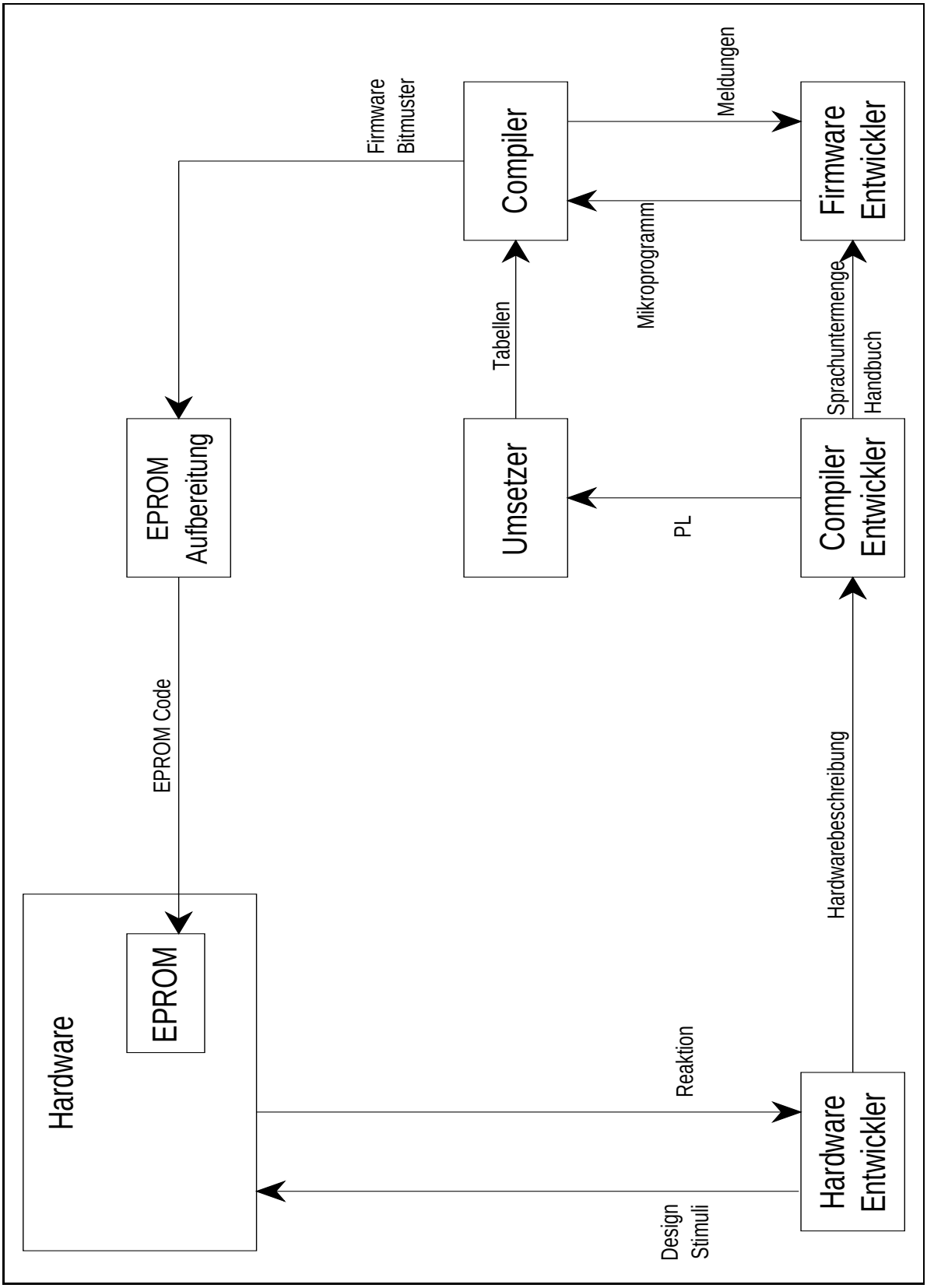

Abbildung 1.1: Der ursprüngliche Entwicklungszyklus.

Die zweite Verbindung ist jene über den *Compilerentwickler*, im Bild in der Mitte unten: Dieser erhält vom Hardwareentwickler eine nicht formalisierte Hardwarebeschreibung, zum Beispiel anhand eines Blockschaltbildes, sowie eine Erläuterung, die gegebenenfalls durch Rückfragen ergänzt werden muß. Als Feedback erhält der Hardwareentwickler vom Compilerentwickler Information, zum Beispiel über mögliche hardwarebedingte Konflikte, wobei es im wesentlichen um die Frage geht, welche Mikrooperationen sich in einem Mikrobefehl vereinen lassen. Da dieses Feedback aber nur gering ausgeprägt ist, wurde es im Bild weggelassen. Der Compilerentwickler trifft dann die Auswahl der Sprachelemente aus dem vorgegebenen Sprachrahmen und stellt sie dem Firmwareprogrammierer zur Verfügung. Außerdem formuliert der Compilerentwickler die notwendigen semantischen Einschränkungen in Form einer Problemsprache PL[2], welche von einem Umsetzer zu Tabellen aufbereitet wird. Diese Tabellen steuern dann die semantischen Prüfungen im Firmwarecompiler.

Damit ist der Kreis geschlossen: Jede Hardwareänderung muß vom Compilerentwickler aufbereitet werden, damit der Firmwareentwickler auf der veränderten Hardware weiterarbeiten kann. Die lauffähigen Mikroprogramme wiederum werden vom Hardwareentwickler benötigt, um die Funktion der veränderten Hardware anhand der lauffähigen Firmware zu testen. Aus diesem Testlauf ergeben sich im allgemeinen wieder Hardwareänderungen und der Zyklus beginnt von neuem. Es ist daher leicht einzusehen, daß die Firmwareentwicklung immer hinter der Hardwareentwicklung nachhinkt. Da die Entwicklung der Hardware insgesamt ohne Simulator relativ langsam war, hat es sich aber nicht negativ ausgewirkt, daß eine Facette der Hardwareentwicklung, nämlich das Testen mit der Firmware, relativ langsam war.

Das hat sich aber schlagartig geändert, als die Entwicklung digitaler Hardware fast zur Gänze auf *Simulator* umgestellt wurde: Nun wird nicht mehr gelötet und gemessen, sondern das Design der Schaltung in ein Programm eingegeben und alles weitere nur noch simuliert. Lediglich solche Hardwareteile, deren Beschreibung zu kompliziert ist, werden weiterhin hardwaremäßig aufgebaut und an den Simulator angeschlossen. Mit zunehmender Rechenleistung der Simulatoren verliert dieser Sachverhalt aber an Bedeutung. Abbildung 1.2 zeigt den veränderten Entwicklungszyklus. Als Fazit wurden die anderen Hardwareentwicklungszyklen, wie etwa Integration, Layout oder Partitionierung, mittels CAD[3] und CAM[4] so stark beschleunigt, daß der Zyklus mit dem Firmwaretest in Relation dazu plötzlich viel zu langsam ist. Daran konnte auch die Einführung eines Simulatorteiles, der speziell dem Testen der simulierten Schaltung mit fertigen Firmwarebitmustern gewidmet ist, nichts ändern: Die Compilerentwicklung ist einfach zu langsam.

Die Ursache dafür ist zunächst im hohen *Kommunikationsaufwand* zwischen Hardwareentwickler und Firmwareentwickler begründet: Gerade diese zwischenmenschliche „Schnittstelle“ ist ja während des Hardwareentwicklungsprozesses laufend Änderungen unterworfen. Da sie aber keinen formalen Charakter besitzt, also in keiner Weise genormt ist, ist sie als Basis denkbar ungeeignet. Der zweite Bremsschuh ist die Entwicklung des Compilers selbst: Eigentlich sollte möglichst rasch nach dem Ende des Hardwareentwur-

[2] *Problem Language*
[3] *Computer Aided Design*
[4] *Computer Aided Manufacturing*

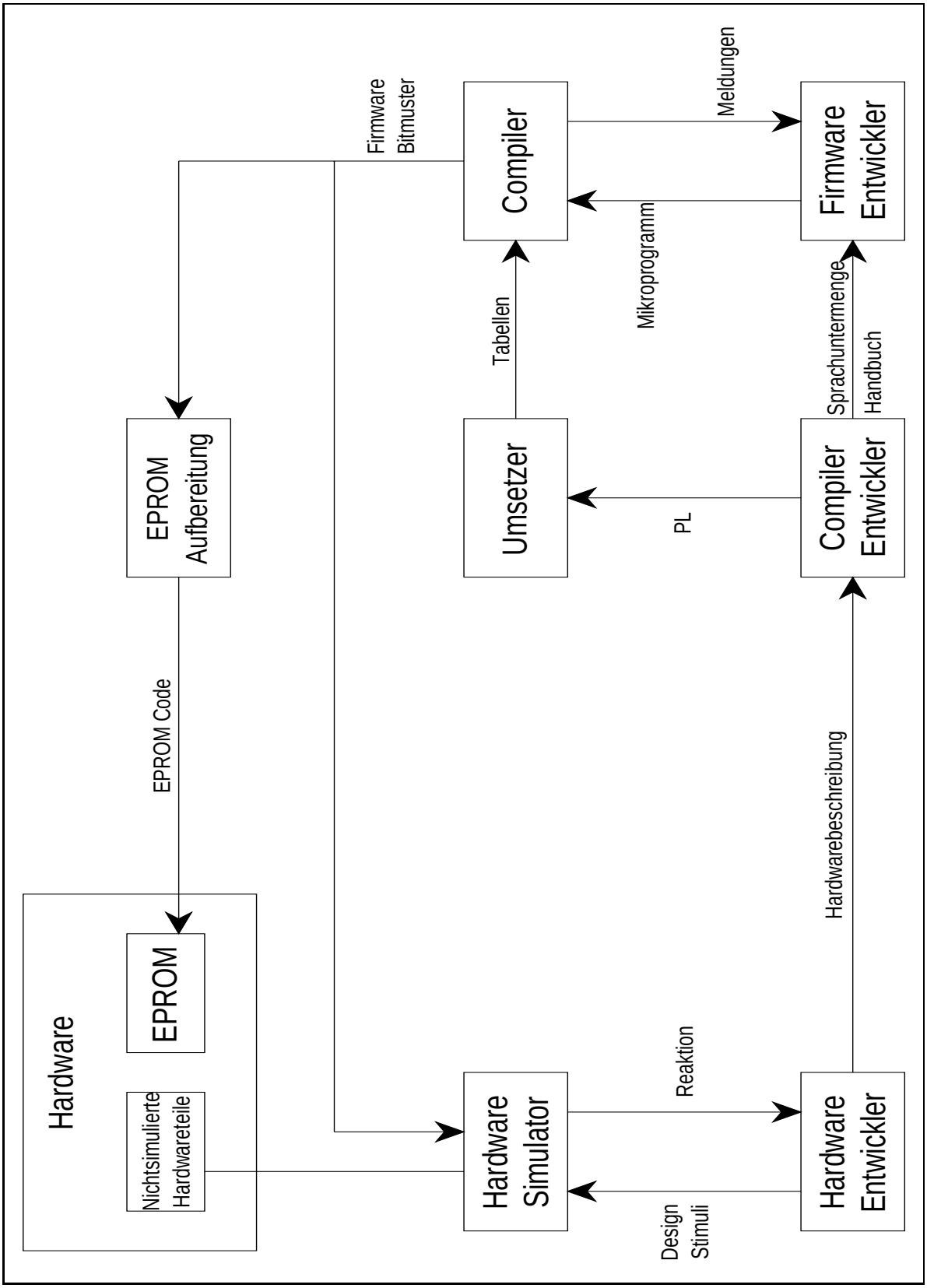

Abbildung 1.2: Der Entwicklungszyklus nach Einführung des Hardwaresimulators.

fes ein funktionierender *Mikroprogrammcompiler* zur Verfügung stehen, um die Firmware entwickeln zu können, mit der die Hardware getestet werden soll. Dies ist aber nicht möglich, da die Anpassung des Compilers sehr mühsam und fehleranfällig ist. Zum besseren Verständnis wird nun kurz der Aufbau des Compilers in seiner bisherigen Form erläutert, die Beschreibung bezieht sich auf Abbildung 1.3.

Da der Compiler *hardwareabhängig* ist, muß er an jede Hardwareänderung angepaßt werden. Dabei wird aber nicht das Programm selbst angepaßt, sondern die Tabellen, die den Ablauf des Übersetzungsvorganges steuern: Jeder Mikrobefehl wird zunächst lexikalisch und syntaktisch überprüft. Da der Sprachrahmen und somit auch die Syntax fix vorgegeben sind, sind die Syntaxtabellen kaum hardwareabhängig, es sei denn, der Sprachrahmen muß an die Erfordernisse einer speziellen Prozessorfamilie angepaßt werden. Der semantische Vorlauf prüft die verwendeten variablen Bezeichner für Register und setzt sie gegebenenfalls um. Die hiezu verwendeten Deskriptortabellen enthalten alle für einen bestimmten Prozessor erlaubten Bezeichner und ihre Bedeutung und sind daher bereits stark hardwareabhängig.

Das größte Problem aber bereitet die *semantische Analyse* selbst: Deren Eingabe ist eine linearisierte und attributierte Form des ursprünglichen Mikrobefehls, die Ausgabe der Bitcode. Mittels der Problemsprachenbeschreibung PL werden die semantischen Regeln und Prüfungen unter Bezugnahme auf die linearisierten Mikrobefehle beschrieben. Aus der PL–Beschreibung erzeugt ein Umsetzer dann die PL–Tabellen, welche die semantischen Prüfungen im Zuge des Firmwarecompilers steuern. Dabei gibt es nun zwei Probleme:

Erstens müssen *Konflikte*, die unmittelbar die Hardwarestruktur betreffen, vom Hardwareentwickler selbst herausgefunden und dem Compilerentwickler als notwendige Einschränkungen mitgeteilt werden. Der Compilerentwickler muß sich dann darüberhinausgehende Konflikte erarbeiten, die zum Beispiel dadurch entstehen können, daß manche Sprachelemente aufgrund der Hardwarestruktur oder der Bedeutung der Bits im Mikrobefehl keine Bitcodeentsprechung besitzen. Dabei fällt auf, daß alle Konflikte von Entwicklern erkannt werden müssen. Dadurch wird die experimentelle Entwicklungsarbeit durch das ständige „händische“ Suchen nach Konflikten stark gebremst. Alle Konflikte gemeinsam werden dann mittels Problemsprache implementiert.

Diese PL–Beschreibung nun ist der zweite Grund für den hohen Aufwand der Compileranpassung an eine Hardwareänderung: Durch die hohe *Redundanz*, die der relativ simplen PL–Beschreibung innewohnt, ist das Erstellen einer PL–Beschreibung nicht nur gedanklich sehr aufwendig, sondern auch im höchsten Maße fehleranfällig. Hier liegt also einer der wichtigsten Ansatzpunkte für eine Verbesserung.

Das Hauptprogramm des Compilers besteht im wesentlichen aus einer Schleife über alle Mikrobefehle, wobei Aufgaben der Blockverwaltung und der Symboltabellenverwaltung wahrgenommen werden. Das vorläufige Endergebnis ist ein Rumpfmodul, welches durch Adreßprüfung und Adreßvergabe in ein fertiges Modul umgeformt wird. Diese Adreßbestimmungen sind ebenfalls hardwareabhängig. Mehrere Module werden schließlich vom Linker zu einer Phase gebunden, die dann schließlich dem Mikroprogrammspeicher zugeführt wird.

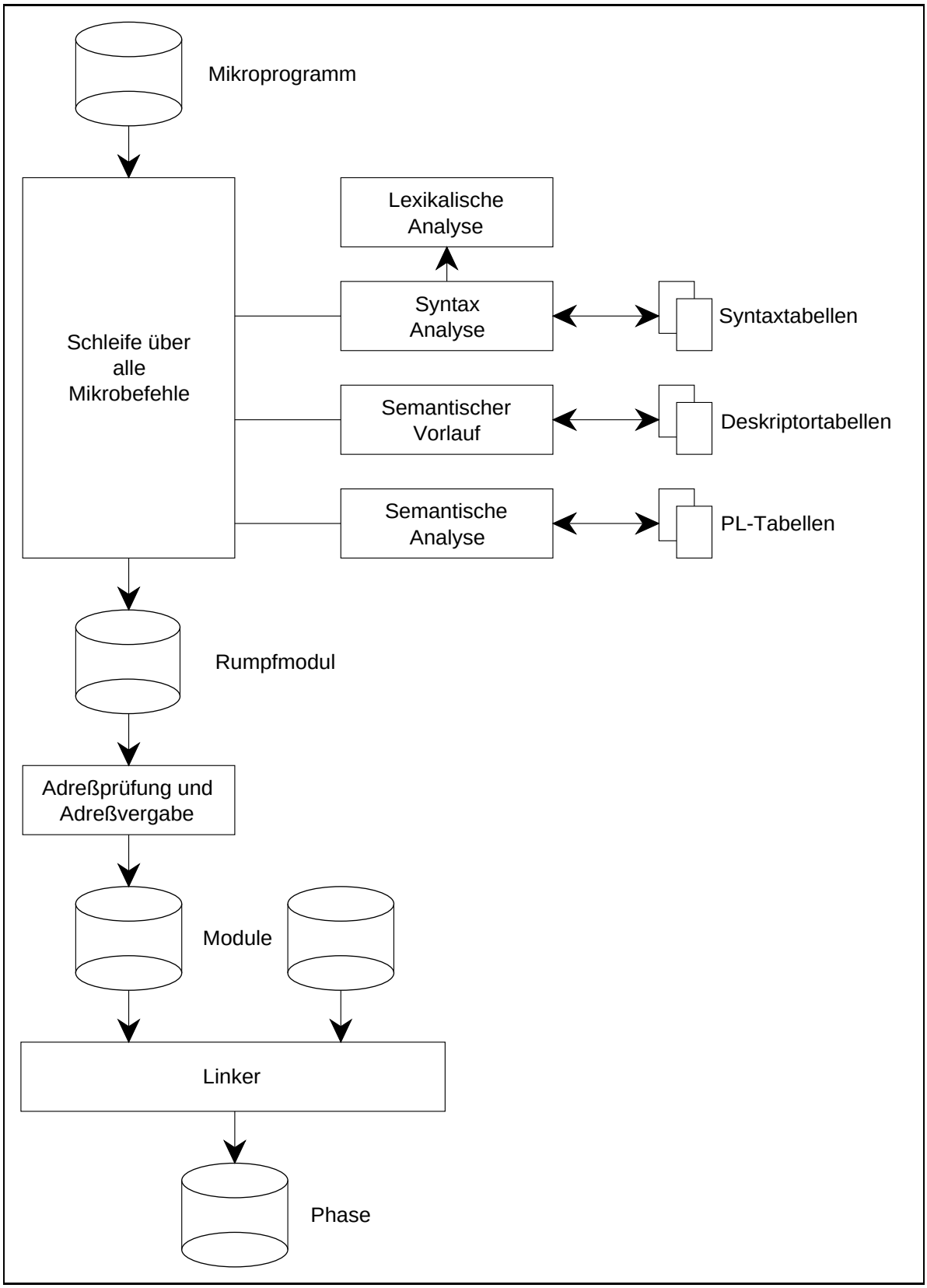

Abbildung 1.3: Der bisherige Aufbau des Firmwarecompilers.

Zusammenfassend kann also festgestellt werden: Die größten Hemmnisse für eine rasche Compilerentwicklung sind die ungeeignete Schnittstelle zwischen Hardwareentwickler und Compilerentwickler, sowie die mühevolle Suche nach Konflikten und die Formulierung der semantischen Prüfungen mittels PL–Beschreibung. Wie nun diese Hemmnisse durch die MDL beseitigt wurden, wird im nächsten Abschnitt erläutert.

1.2 Vorangegangene Arbeiten

Als erster Schritt wurde die *Schnittstelle* zwischen Hardwareentwickler und Compilerentwickler formal klar definiert [4]. Den veränderten Entwicklungszyklus zeigt Abbildung 1.4. Die Sprache MDL[5] wurde entwickelt, um die Hardware so zu beschreiben, daß alle für die Compilerentwicklung notwendigen Informationen dargestellt werden können. Es können durch die Beschreibung alle möglichen Konflikte erkannt werden, die sowohl bei der Hardwareentwicklung, als auch bei der Übersetzung der Mikroprogramme selbst entstehen können. Diese Aufgabe erfüllt der Analysator, der automatisch entsprechende Konfliktabellen erstellt: Der Hardwareentwickler muß die Hardwarebeschreibung mittels MDL/HD[6] durchführen und an den Analysator weiterreichen. Dieser liefert die MDL/HD–Beschreibung, ergänzt durch die Tabellen von erkannten möglichen Konfliktfällen, an den Compilerentwickler weiter. Dieser definiert nun, so wie auch bisher, die prozessorspezifische Sprachuntermenge und erstellt die PL–Beschreibung. Andererseits erhält der Hardwareentwickler sein Feedback jetzt nicht mehr vom Compilerentwickler, sondern vom Analysator, und zwar ebenfalls in Form von Konfliktabellen. Dabei handelt es sich um Diagnoselisten über die gegenseitige Blockierung einzelner Mikrooperationen mit Angabe der Blockierungsursache. Die dabei verwendete MDL/HD ist jene Untermenge der MDL, die der Beschreibung der unmittelbaren Hardwareinformationen dient.

Die Sprache MDL/HD und der Analysator

In diesem Abschnitt werden nun die Eigenschaften der MDL/HD sowie die Funktionalität des Analysators kurz zusammengefaßt, Details sind in [4] nachzulesen. Zweck der Beschreibung eines Mikroprozessors mit der Sprache MDL ist die Lösung der in den vorangegangenen Abschnitten aufgezeigten Probleme, wobei die Untermenge MDL/HD die folgenden Punkte betrifft:

- Normung der Schnittstelle zwischen Hardwareentwickler und Compilerentwickler und Verminderung des Kommunikationsaufwandes.
- Automatische Auflistung der hardwarebedingten Konflikte zum Zwecke eines zusätzlichen Hardwareentwicklungszyklus mit der Möglichkeit, auch auf einer abstrakten Maschine entwickeln zu können.

Die Sprache MDL ermöglicht daher die Beschreibung folgender Sachverhalte:

1. Die *Hardwareressourcen* des Prozessors. Das ist die Netzstruktur bestehend aus Hardwareelementen und -verbindungen. Anders als bei anderen Hardwarebeschreibungssprachen sind aber hier weder die funktionalen Eigenschaften der Hardwareelemente noch die Breite der Verbindungen interessant. Die Beschreibung der Hardwareressourcen erfolgt im Deklarationsteil: Die Elemente werden mit

[5] Microcompiler Design Language.
[6] MDL/Hardware Description.

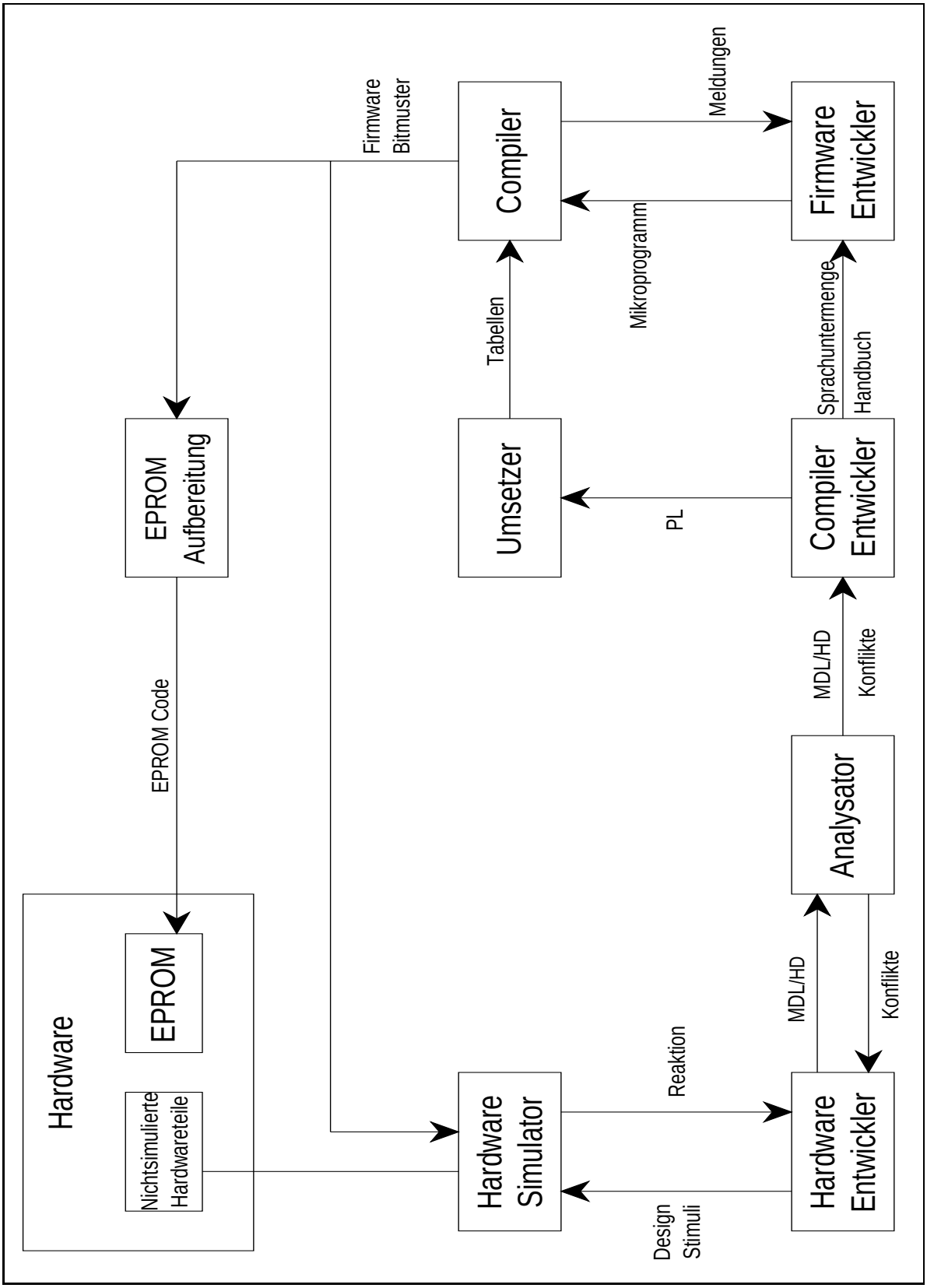

Abbildung 1.4: Der Entwicklungszyklus mit MDL.

den Schlüsselwörtern `DEVICE` und `REGISTER` definiert, die Verbindungen mit dem Schlüsselwort `PATH`. Als besondere Ressource ist auch die Beschreibung des Mikrobefehlsfeldes von großer Bedeutung, diese wird mit dem Schlüsselwort `FIELD` gehandhabt. Außerdem bieten die Schlüsselwörter `SET` und `GROUP` noch die Möglichkeit der Gruppierung von Ressourcen.

2. Die Verwendung der Hardwareressourcen in den *Mikrooperationen* wird im Operationsteil beschrieben: Das Schlüsselwort `MICROP` definiert die Mikrooperationen, mit dem Schlüsselwort `USING` wird die Verwendung der Ressourcen und mit dem Schlüsselwort `DATAFLOW` die Verwendung der benötigten Verbindungen beschrieben. Die Verwendung der Mikrobefehlsfelder wird mit dem Schlüsselwort `FIELD` beschrieben. Das Schlüsselwort `TIME` dient der Angabe der Zeitangaben, mit den Schlüsselwörtern `READ` und `WRITE` wird die Art der Benutzung von Registern angegeben. Die Handhabung der subtilen Konflikte zweiter Art (vergleiche Seite 21) erfolgt mit den Schlüsselwörtern `VAR` und `CONDITION`.

Neben der Frage, *was* eine Sprache beschreiben kann, ist auch die Frage, *wie* es beschrieben werden kann, von Bedeutung. Folgende Eigenschaften besitzt die Sprache MDL:

deskriptiv: Im Gegensatz zu logischen, funktionalen oder prozeduralen Sprachen ist die MDL rein deskriptiv, also beschreibend: Die funktionalen Eigenschaften der Hardwareelemente und der Mikrooperationen werden nicht dargestellt. Das liegt vor allem daran, daß für die Entwicklung des Firmwarecompilers Syntax und Semantik der Mikroprogrammsprache ausreichen, erst der Firmwareentwickler benötigt zusätzliche Information über die funktionalen Zusammenhänge. Da jedoch in den meisten Fällen Hardwareentwickler und Firmwareentwickler sehr eng zusammenarbeiten, wenn nicht sogar ein und dieselbe Person sind, stellt das kein Problem dar.

effizient: Diese Eigenschaft ist gleichbedeutend mit geringer Redundanz: Die Sprache ist so beschaffen, daß jene Elemente und Zusammenhänge, die in der Praxis sehr oft beschrieben werden müssen, besonders einfach und übersichtlich beschrieben werden können. Hand in Hand damit gehen leichte Änderbarkeit und geringe Fehleranfälligkeit. Die geringe Redundanz wird vor allem durch die mächtigen Möglichkeiten der Schlüsselwörter `FIELD`, `MAP` und `ALIAS`, aber auch durch `GROUP` und `SET` erreicht. Lediglich eine gewisse Redundanz besteht zwischen dem Simulator, der ja bereits eine Hardwarebeschreibung enthält, und der Hardwarebeschreibung durch die MDL. Wie im Abschnitt 1.3 ab Seite 16 aufgezeigt, kann diese Redundanz durch Einführung eines Filters überwunden werden.

einfach: Damit ist vor allem die Einfachheit der syntaktischen Struktur gemeint: Die Sprache soll eine möglichst übersichtliche und einheitliche Syntax besitzen, um leicht und rasch erlernbar zu sein. Darüber hinaus wurde die Programmiersprache C als syntaktisches und lexikalisches Vorbild gewählt. Durch die weite Verbreitung dieser Programmiersprache wird auf diesem Weg der Zugang zur MDL erleichtert.

flexibel: Das bedeutet, daß trotz der eben angesprochenen Einfachheit die Sprache eine hohe Mächtigkeit in der Möglichkeit und Vielfalt des Beschreibbaren besitzt. Flexibilität und Mächtigkeit sind insofern gegeben, als die Sprache keine Beschränkung auf eine spezielle Prozessorfamilie enthält. Dennoch ist es denkbar, daß sie für bestimmte Prozessoren noch erweitert werden müßte.

großzügig: Dieser Punkt ist die Voraussetzung für die Verwendung der Sprache im Rahmen eines Entwurfswerkzeuges: Eine MDL–Beschreibung ist korrekt und bearbeitbar, auch ohne Festlegung aller Einzelheiten bis ins kleinste Detail. Eine Menge von mannigfachen impliziten Annahmen ergänzt Unvollständiges oder Fehlendes im Sinne des Benutzers. Die Sprache selbst erlaubt dabei die korrekte Darstellung von sehr mangelhaften Informationen, sodaß die Möglichkeit der Beschreibung einer abstrakten Maschine ohne Festlegung von Bezeichnern, Bitcodes oder ähnlichem möglich ist. Hier soll vor allem darauf aufmerksam gemacht werden, daß bei den Schlüsselwörtern `DEVICE`, `REGISTER`, `FIELD`, `GROUP`, `SET` und `MICROP` bis auf den Namen alle Angaben optional sind!

Die Abstraktionsebene, auf der die MDL–Beschreibung angesiedelt ist, befindet sich etwa zwischen der Architekturebene[7] und der Organisationsebene[8]. Das liegt daran, daß zwar die Hardwarestruktur auf Organisationsebene beschrieben wird, andererseits aber die Mikrooperation auf Architekturebene das elementare Objekt zur Konflikterkennung und Syntax/Code–Umsetzung darstellt. Beispiel 1.1 auf Seite 12 zeigt einen exemplarischen Ausschnitt aus einer MDL/HD-Beschreibung. Dieser Ausschnitt dient in weiterer Folge auch als Basis für die Beispiele der Ergänzungen und Erweiterungen der MDL.

Nachdem nun die Eingenschaften der Sprache erläutert wurden, soll auch die Funktionalität des *Analysators* dargestellt werden:

- Einlesen einer MDL/HD–Beschreibung und Prüfung der syntaktischen Korrektheit. Beim Auftreten von Syntaxfehlern wird die weitere Analyse mit entsprechenden Fehlermeldungen abgebrochen. Wenn hingegen die MDL–Beschreibung semantisch fehlerhaft oder unvollständig ist, wird die Analyse mit Hilfe impliziter Annahmen nach Möglichkeit fortgesetzt. Dies entspricht der Forderung nach einem großzügigen Entwurfswerkzeug.

- Ausgabe einer Tabelle aller verwendeten Bezeichner, die dem semantischen Vorlauf zwecks Prüfung der hardwarespezifischen Namen zur Verfügung gestellt wird.

- Ausgabe einer Tabelle aller überhaupt möglichen Konflikte (bottom-up) sowie Erstellung von konkreten Konfliktanalysen für bestimmte Anfragen (top-down), zum Beispiel eine konkrete Kombination von Mikrooperationen.

[7] *Instruction set level*
[8] *Register transfer level*

```
/*

SAMPLE PROCESSOR P95

*/

/* Devices */

DEVICE #2 alu;
DEVICE a_bus;
DEVICE b_bus;
DEVICE c_bus;

/* Registers */

MAP wbytes {
      <#8> 0,1,2,3  MAP bdigits { <#4> h,l; };
      };
MAP wdigits {
      <#4> 0h,0l,1h,1l,2h,2l,3h,3l;
      };
MAP word OF wbytes, wdigits;

REGISTER ut[0:15]<0:31> MAP word;  /* row 0 */
REGISTER gr[0:15]<0:31> MAP word;  /* row 1 */
REGISTER fr[0:15]<0:31> MAP word;  /* row 2 */
REGISTER ux[0:15]<0:31> MAP word;  /* row 3 */

REGISTER reg[0:15]<0:31> MAP word; /* extra */
pt0 ALIAS reg[2]; /* pointer register 0 */
pt1 ALIAS reg[3]; /* pointer register 1 */

ALIAS pt0 {
      <#8> p0, p1, p2, p3 MAP bdigits;
};
ALIAS pt1 {
      <#8> p4, p5, p6, p7 MAP bdigits;
};
SET pointer { pt0, pt1 };

/* Fields */

MAP adress {
      <0,2>  art; /* address */
      <2,3>  dl;  /* digit length */
      <5,1>  d;   /* digit offset high / low */
      <6,4>  w;   /* word offset */
      <10,2> b;   /* byte offset */
      <2,2>  bl;  /* byte length */
      <4,2>  r;   /* register row */
      <6,3>  wi;  /* word indirect pointer */
      <9,1>  x;   /* xor bit */
```

Beispiel 1.1: Beispiel für eine MDL-Beschreibung.

```
      <6,3>  dw;  /* double word offset */
      <10,2> bi;  /* byte indirect pointer */
      <3,1>  a;   /* direction */
      };

FIELD fac<8:19>  MAP address; /* AC operand */
FIELD fb<20:31>  MAP address; /* B  operand */
FIELD fgroup<0:1>;            /* instruction group */
FIELD ffunction<3:7>;         /* ALU function */
FIELD fwait;                  /* wait bit */

/* time granularity: 1 = 1ns */

#define T 25 /* clock cycle 25ns, clock frequency 40MHz */
#define T1 0,T
#define T2 T,T
#define T3 2*T,T
#define T4 3*T,T
#define Strobe 1000

/* Microinstructions */

MICROP  addition;
VAR     p1,p2 PARTOF pointer;
        reg1, reg2 PARTOF ut, gr, fr, ux;
FIELD   fgroup, fwait, ffunction, fac, fb;
USING
        READ p1,p2 TIME (0,12);
        READ reg1, reg2 TIME (T1);
        a_bus, b_bus TIME (T,10);
        alu TIME (T2;T3);
        c_bus TIME (Strobe+T,4;Strobe+2*T,4);
        WRITE reg2 TIME (T4);
DATAFLOW
        reg2 -> a_bus -> alu -> c_bus -> reg2;
        reg1 -> b_bus -> alu;
CONDITION
        ! reg1 <= pointer;
        ! reg2 <= pointer;
ENDOP;
```

Beispiel 1.1 (Fortsetzung): Beispiel für eine MDL-Beschreibung.

Konsequenzen

Aus der Einführung der MDL/HD ergeben sich drei Konsequenzen: Erstens wird der *Kommunikationsaufwand* auf ein Minimum reduziert, indem sichergestellt wird, daß exakt die benötigte Information in klarer Darstellung übermittelt wird, nicht mehr und nicht weniger. Als zweite Konsequenz wird der Compilerentwickler von den *Konflikttabellen* des Analysators unterstützt, sodaß er nicht mehr alle Konflikte „händisch“ suchen muß, was ihm einen Teil seiner Arbeit abnimmt: Er konsultiert nunmehr die Konflikttabelle und verwendet sie als Basis für einen Teil der semantischen Regeln. Deren Implementierung in Form der PL–Beschreibung ist aber nach wie vor reformbedürftig, da ein Mensch für diese Aufgabe eigentlich zu schade ist.

Als dritte Konsequenz erhält der Hardwareentwickler einen zusätzlichen *Entwicklungszyklus*: Bisher war die Reaktion auf eine Hardwareänderung viel zu langsam, um spielerisch experimentieren zu können. Außerdem mußten bisher Schaltung und Bitcode sowie die Verwendung der Hardwareelemente in den Mikrooperationen bereits genau feststehen, da ja in weiterer Folge daraus der Compilerentwickler die notwendigen semantischen Einschränkungen für den Firmwarecompiler hergeleitet hat. Nun aber kann der Hardwareentwickler relativ schnell verschiedene Ansätze und Varianten durchspielen: Die Konflikte werden stets automatisch und damit rasch und unkompliziert erzeugt. Wenn die Funktion des Analysators sichergestellt ist, wird auch die Fehlerquote geringer sein, es werden kaum noch Fehler übersehen werden. Darüberhinaus wird es dem Hardwareentwickler ermöglicht zu experimentieren, sogar ohne sich in bezug auf Bitcodemuster, Bitfelder oder Bezeichner allzusehr festzulegen: Da viele dieser Angaben erst bei der Compilergenerierung benötigt werden, beeinträchtigt ihr Fehlen nicht die Funktion des Analysators, insbesondere die Konflikterkennung. Aber auch der Analysator selbst muß großzügig und flexibel genug sein, um bei fehlenden Vorgaben sinnvolle Annahmen zu treffen.

Vor allem diese Möglichkeit, auf einer *abstrakten Maschine* mit der Entwicklung zu beginnen, bietet dem Hardwareentwickler einen zusätzlichen Freiheitsgrad bei der Wahl von Designalternativen. Ein Beispiel: Er kann zunächst mit breiten, horizontalen Bitmustern beginnen, die zwar schnell, aber auch teuer sind. Dann kann er die Auswirkung der Vertikalisierung von Mikrobefehlen unmittelbar studieren und so einen Kompromiß aus Kosten und Geschwindigkeit finden. Erst dann muß überhaupt erst die Befehlsbreite festgelegt werden und erst zum Schluß der Bitcode selbst. Auch mit der Festlegung der Register kann sich der Hardwareentwickler bis zuletzt Zeit lassen. Erst wenn er eine Variante gefunden hat, die ihm wirklich brauchbar erscheint, muß er den Compilerentwickler bemühen. Diese Möglichkeit, eine MDL/HD–Beschreibung für eine abstrakte Maschine zu entwerfen, setzt also den Kommunikationsaufwand zusätzlich weiter herab, da viele Konflikte bereits vom Hardwareentwickler erkannt und entsprechend behandelt werden können. Daher muß nicht jedesmal der Compilerentwickler bemüht werden, der vergleichsweise immer noch relativ langsam ist, da er nach wie vor die umständliche PL–Beschreibung der semantischen Regeln erstellen muß.

Die Vorteile dieses ersten Ansatzes sind also eine genormte Schnittstelle, die rasche Überprüfbarkeit des Entwurfes im Hinblick auf Konflikte durch den Hardwareentwickler selbst, die Möglichkeit, zuerst auf einer abstrakten Maschine zu entwickeln, und die

automatische Suche der Konflikte als Basis für einen Teil der semantischen Regeln. Der größte Nachteil ist die umständliche Implementierung der semantischen Regeln mittels PL–Beschreibung.

1.3 Lösungsansatz für die Compilergenerierung

Aufgrund der verbleibenden Probleme ist im Zuge der Diplomarbeit als weiterer Schritt geplant, daß der Compilerentwickler die erhaltene MDL/HD–Beschreibung um eine MDL/BED[9]–Beschreibung erweitert, damit daraus der Firmwarecompiler *generiert* werden kann. Das bedeutet, daß die MDL/BED–Beschreibung vor allem die semantischen Regeln enthalten muß und die Auswahl der prozessorspezifischen Sprachelemente aus dem vorgegebenen Sprachrahmen sowie die Syntax/Bitcode–Zuordnung ermöglichen muß. Diese Ergänzung der erhaltenen MDL/HD–Beschreibung zur vollen MDL–Beschreibung ersetzt dann die Erstellung der PL–Beschreibung. Abbildung 1.5 zeigt den weiter beschleunigten Entwicklungszyklus.

Wodurch wird dabei die Verbesserung erreicht? Die Redundanz der PL–Beschreibung entsteht nicht durch Überflüssiges, sondern durch Ähnliches. Das bedeutet, daß etwa 80% der in der Praxis vorkommenden Fälle so geartet sind, daß eine PL–Beschreibung im höchsten Maße redundant ist. Die MDL/BED–Beschreibung hingegen muß spezielle Mechanismen vorsehen, um diese in der Praxis häufiger auftretenden Fälle auf eine vereinfachte Weise zu beschreiben. Effizienz einer Sprache bedeutet ja nichts anderes, als die Sprachmittel für jene Elemente kurz zu halten, die am häufigsten beschrieben werden. Daher kann Effizienz aber auch nur für einen gewissen Anwendungsbereich garantiert werden. Da somit die MDL/BED–Beschreibung auf einer höheren Abstraktionsebene als die PL–Beschreibung stattfindet und außerdem kaum redundant ist, wird die Anpassung an eine Hardwareänderung drastisch vereinfacht.

Das Ziel dieses Ansatzes ist die automatische Erzeugung der semantischen Prüfungen und der Codegenerierung für den Mikroprogrammcompiler. Die Vorteile liegen jetzt zusätzlich in der Verlagerung der Arbeiten zur Erzeugung eines prozessorspezifischen Compilers in die Labors, in der schnelleren Verfügbarkeit der semantischen Prüfungen und letztendlich in der Einsparung von Personalkapazität für Anpassungen: Selbst wenn sich zusätzliche semantische Einschränkungen als notwendig erweisen sollten, ist dennoch ein neuer Compiler wieder in kürzester Zeit verfügbar. Dies ist unter anderem dann denkbar, wenn sich entweder durch Simulation oder aber durch Testen des Prototyps Einschränkungen ergeben, die auf Realisierungsbedingungen zurückgeführt werden können.

Der *Generator* ermittelt die Sprachuntermenge und generiert die hardwareabhängigen Tabellen und Unterprogrammfunktionen für den Firmwarecompiler. Der Compilerentwickler wird deutlich entlastet, er steuert den Generierungsprozeß und überwacht das Ergebnis. Sein Aufgabenbereich hat sich also drastisch verändert, weg von der „Knochenarbeit“ hin zum kreativen Design. Damit ist die Compilerentwicklung genauso rasch geworden wie die anderen Teile des gesamten Entwicklungszyklus. Da jede Kette nur so stark ist wie ihr schwächstes Glied, steht erst jetzt ein effizienter Prozessorentwicklungszyklus zur Verfügung.

Ein Maß für die Effizienz einer Sprache ist die enthaltene Redundanz. Daher stellt der Übergang von der PL–Beschreibung zur MDL/BED–Beschreibung eine wesentliche Verbesserung der Effizienz des Entwicklungszyklus dar. Dennoch ist noch ein gewisses Maß

[9] MDL/Back End Description

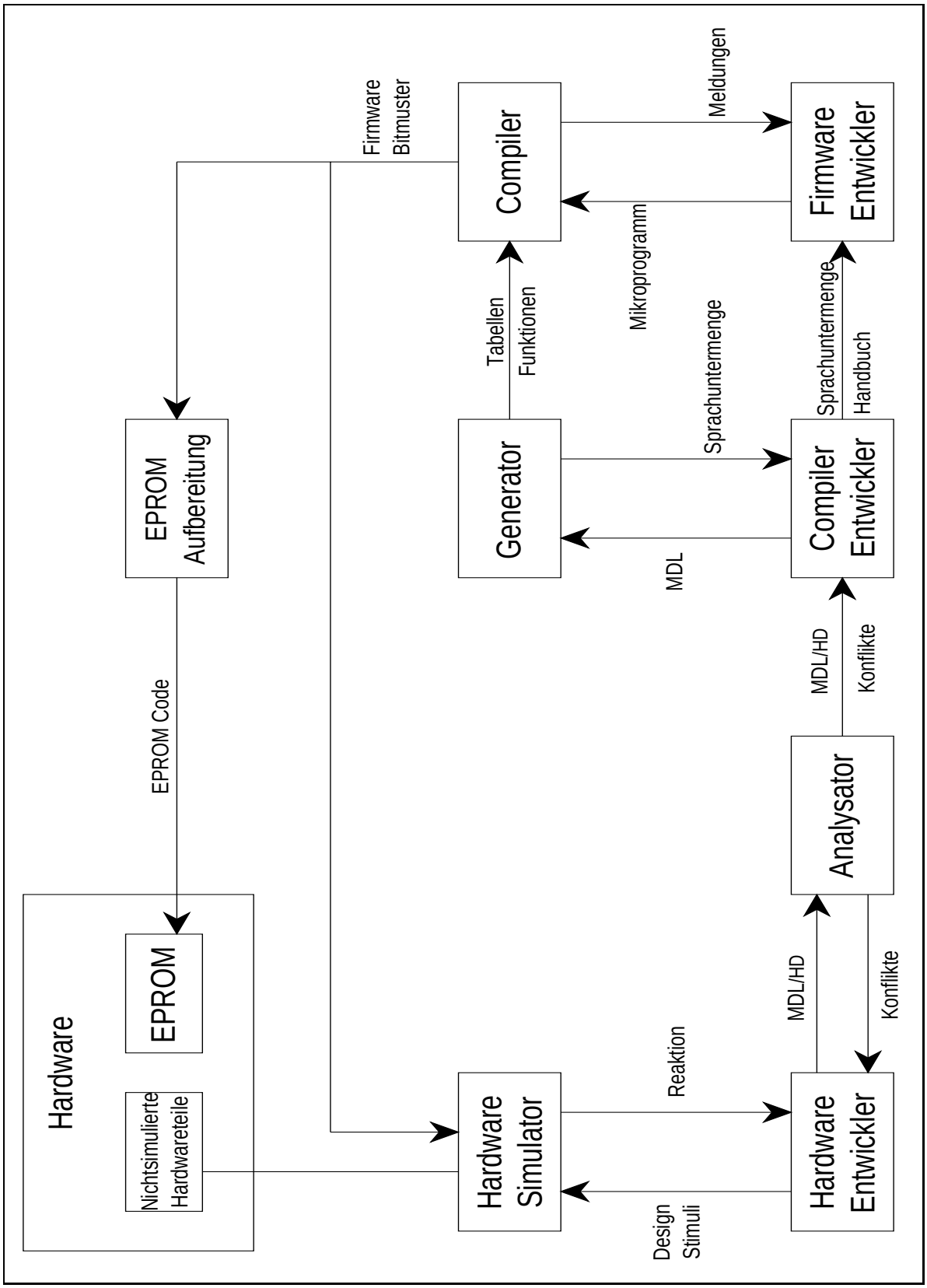

Abbildung 1.5: Der Entwicklungszyklus mit erweiterter MDL und Compilergenerator.

an Redundanz in der MDL/HD–Beschreibung enthalten: Der Hardwaresimulator, auf dem der Hardwareentwickler arbeitet, enthält bereits intern eine Prozessorbeschreibung in irgendeiner Hardwarebeschreibungssprache HDL[10]. Diese HDL enthält aber bereits explizit und implizit die Masse der Informationen, die in der MDL–Beschreibung enthalten sind. Daher wäre es noch sinnvoll, mittels eines *Filters* die MDL/HD–Beschreibung aus der simulatorinternen HDL–Beschreibung zu generieren. Mit der damit verbundenen Redundanzverminderung wäre eine weitere Effizienzsteigerung verbunden, die im Endeffekt zu einer zusätzlichen Beschleunigung des Entwicklungszyklus führt. Abbildung 1.6 zeigt den aus der Sicht der Firmwareerstellung ideal beschleunigten Entwicklungszyklus. Der Hardwareentwickler kann die MDL/HD zunächst für eine abstrakte Maschine selbst entwerfen; wenn dann später bereits der Hardwaresimulator in Verwendung ist, wird die MDL/HD–Beschreibung einfach extrahiert. Gegebenenfalls kann der Hardwareentwickler die extrahierte Beschreibung dann variieren, um verschiedene Möglichkeiten zu vergleichen. Er muß aber nicht mehr selbst dafür Sorge tragen, daß die MDL/HD–Beschreibung mit der Simulator–HDL übereinstimmt. Genau diese Redundanz ist mit Einführung des Filters beseitigt.

Nun hätte man gleich eine bestehende Hardwarebeschreibungssprache wie etwa VHDL [18], ARCHI [19] oder ISPS [28] heranziehen können, um sowohl die Schnittstelle zu formalisieren als auch den Compiler zu generieren. Diese Vorgangsweise hat aber einige Nachteile. Vor allem ist es sehr wahrscheinlich, daß diese bestehende HDL hätte erweitert werden müssen, womit der Vorteil der Verwendung einer bestehenden Sprache wieder verloren gewesen wäre. Zweitens ist die HDL–Schnittstelle zum Simulator ganz allgemein eher kurzfristig und mannigfaltig veränderlich, sodaß eine enge Bindung an eine solche HDL eine unnötige Einschränkung bewirkt hätte. Drittens schließlich dient die MDL als ruhige und überprüfbare Basis für die Firmwarecompilererstellung und enthält exakt jene Information, die dafür notwendig und hinreichend ist. Das hat den Vorteil, daß im Zuge der Entwicklung der MDL deutlich geworden ist, welche Information nun tatsächlich erforderlich ist. Aufbauend auf diese MDL können nun Filter sozusagen als Adapter für bestehende Hardwarebeschreibungssprachen verwendet werden, wobei vermutlich nur die MDL/HD–Beschreibung generiert werden kann, die MDL/BED–Beschreibung muß nach wie vor vom Compilerentwickler erstellt werden, denn genau das ist ja seine neue Aufgabe. Die Anpassung an eine neue HDL bedeutet dann jedenfalls lediglich die Änderung des Adapters.

[10] *Hardware Description Language*

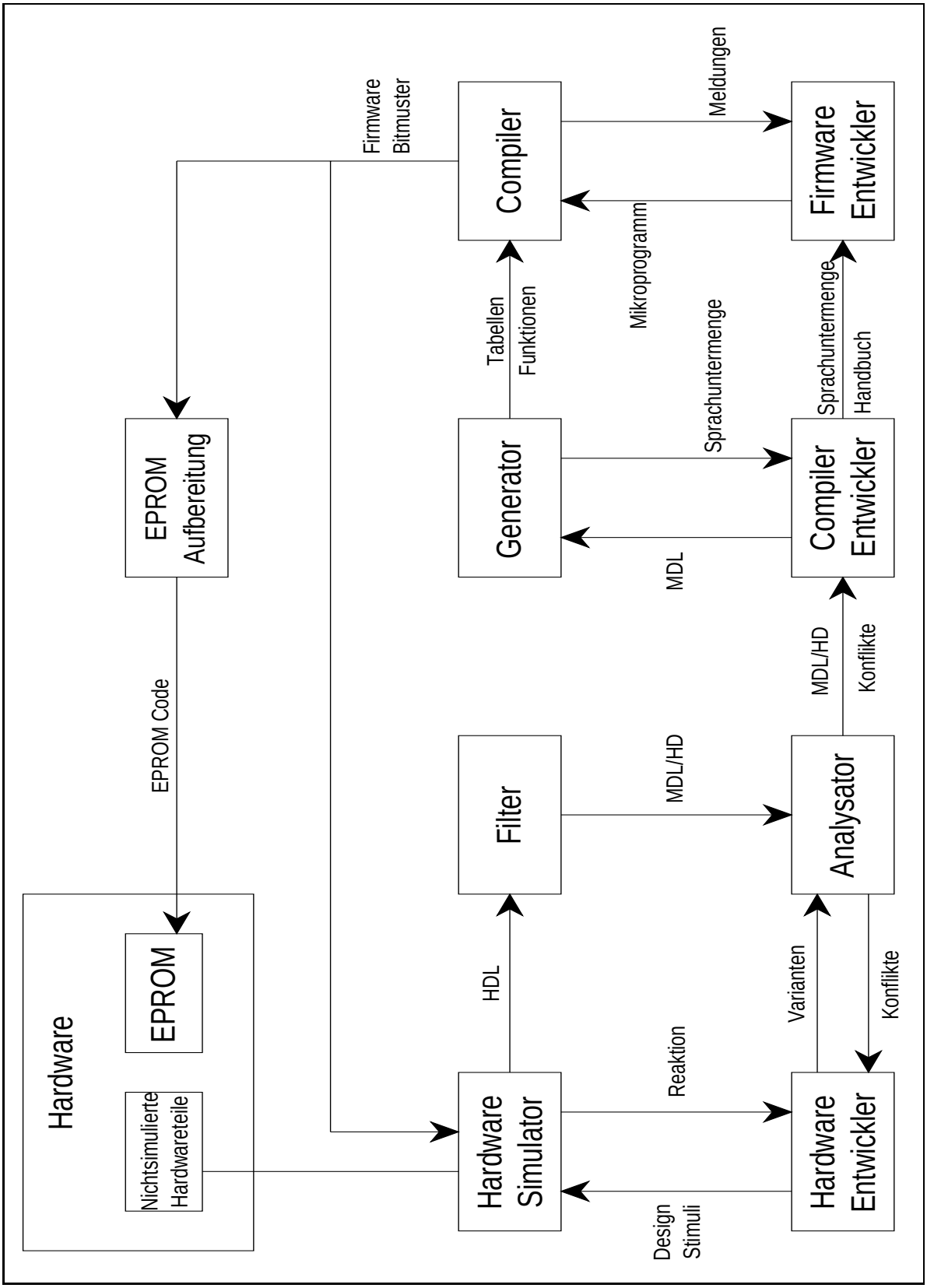

Abbildung 1.6: Der Entwicklungszyklus mit teilweise generierter MDL.

1.4 Anforderungen

Nun werden die Anforderungen an die MDL/BED-Spracherweiterung sowie an den Generator zusammengefaßt. Im abschließenden Abschnitt „Zielüberprüfung“ des folgenden Kapitels wird dann verglichen, wie sowohl die Spracherweiterung, als auch der Generator die Anforderungen erfüllen. Vor allem die folgenden Punkte sind für den Teil MDL/BED wichtig:

- Automatische Generierung der hardwarebedingten semantischen Regeln, d.h. Generierung der Konfliktprüfung von Ressourcenkonflikten.
- Effiziente Implementierung der sonstigen semantischen Prüfungen sowie der Codeerzeugung.
- Auswahl der prozessorspezifischen Sprachuntermenge aus dem fest vorgegebenen Sprachrahmen.

Die Spracherweiterung MDL/BED muß daher die Beschreibung folgender Sachverhalte ermöglichen:

1. Die Formulierung der semantischen Aktionen unter Verwendung der Attribute der Mikroprogrammsprache. Zwar wird die Syntax/Code–Zuordnung eine funktionale Beschreibung erfordern, da es sich dabei aber nicht um die Beschreibung von Hardwarefunktionen handelt, ist die MDL bezüglich der Hardware dennoch rein deskriptiv.
2. Die Zuordnung der *Syntaxelemente* der Mikroprogrammsprache zu den Mikrooperationen, wobei die Abhängigkeit der Verwendung einzelner Hardwareressourcen von den konkreten Mikrobefehlen zu beachten ist. Dies dient der Erkennung von Konflikten auf Hardwareressourcen.
3. Die Beschreibung der Codegenerierung.

Als Ausgangsbasis ist zu beachten, daß bereits mit den Schlüsselwörtern `REGISTER`, `MAP` und `ALIAS` Information angegeben wird, die für die Syntax/Code–Zuordnung relevant ist, und zwar für die Deskriptoren und ihre Alias-Namen.

Daraus ergeben sich die Anforderungen an den *Generator*, die hier ebenfalls aufgezählt werden:

- Einlesen einer MDL/BED–Beschreibung und syntaktische Prüfung.
- Erzeugung von Konflikttabellen in einer Form, die zur Steuerung der semantischen Prüfungen geeignet ist.

- Auswahl der Sprachuntermenge aus dem Sprachrahmen und Angaben über Einschränkungen, die der Firmwareprogrammierer selbst berücksichtigen muß, weil sie sich der Überprüfung durch den Compiler entziehen. Dabei kann es sich zum Beispiel um Konflikte handeln, die sich erst zur Laufzeit ergeben, wie etwa bei der indirekten Adressierung.
- Generierung von Tabellen und Funktionen zur Steuerung der Umsetzung der Syntaxelemente in den Bitcode.
- Bereitstellen einer Schnittstelle, über die der Compilerentwickler steuernd in den Analysevorgang eingreifen kann, etwa um Sonderfälle abzuhandeln.

Um später näher darauf eingehen zu können, wie *Konflikte* analysiert werden, wird hier zunächst eine Einteilung der Konfliktarten getroffen. Konflikte zwischen Mikrooperationen entstehen ganz allgemein dann, wenn Hardwareressourcen zu einem bestimmten Zeitpunkt mehrfach genutzt werden. Typische Konfliktursachen sind zum Beispiel Register und Busbenutzung oder überlappende Bitfelder im Mikrobefehl. Grundsätzlich kann man aus der Sicht der Compilergenerierung drei Arten von Konflikten unterscheiden:

Totale Konflikte entstehen, wenn zwei Mikrooperationen unabhängig von einem konkreten Mikroprogramm einen Ressourcenkonflikt verursachen. Solche Konflikte *erster Art* entstehen zum Beispiel dann, wenn Bitfelder im Mikrobefehlsfeld doppelt genutzt werden, wenn zwei Mikrooperationen die einzige ALU[11] zur selben Zeit benötigen oder ähnliches. Solche Konflikte bedeuten für den Compilerentwickler beziehungsweise später für den Generator die absolute Unvereinbarkeit dieser Mikrooperationen in einem Mikrobefehl. Wenn hingegen bereits innerhalb einer Mikrooperation ein totaler Konflikt auftritt, so handelt es sich um einen Fehler, den der Analysator entsprechend melden und behandeln muß und der bereits vom Hardwareentwickler beseitigt werden muß.

Partielle Konflikte sind abhängig von einer konkreten Mikroprogrammeingabe, man könnte sie auch als Übersetzungskonflikte bezeichnen, da sie erst zum Zeitpunkt der Übersetzung eines Mikroprogrammes im Firmwarecompiler konkret geprüft werden können. Dazu ein Beispiel: Wenn zwei Mikrooperationen je ein Register aus einer Menge von zum Beispiel 64 Registern benutzen, so handelt es sich nur um einen möglichen Konflikt, der nicht zwingend eintreten muß. Der Compiler muß dann für eine konkrete Eingabe prüfen, ob ein Konflikt vorliegt, weil beide Mikrooperationen dasselbe Register beanspruchen, oder nicht. Für den Compilerentwickler beziehungsweise den Generator bedeutet ein solcher Konflikt *zweiter Art* die Implementierung von entsprechenden semantischen Einschränkungen, die wesentlich komplexer sind, als die Überprüfung von totalen Konflikten. Im Konfliktfall gibt es dann grundsätzlich die zwei Möglichkeiten, entweder die Übersetzung mit einer Fehlermeldung abzubrechen, oder einer der beiden Mikrooperationen den Vorrang einzuräumen und die andere zu ignorieren.

[11] *Arithmetical Logical Unit*

Laufzeitkonflikte können vom Compiler überhaupt nicht überprüft werden, da sie zur Übersetzungszeit noch nicht feststehen. Ein typisches Beispiel bietet der Registerzugriff wie vorhin, jetzt aber mittels indirekter Adressierung: Der Inhalt der Pointerregister steht zur Übersetzungszeit nicht fest, daher kann nicht überprüft werden, ob die beiden Mikrooperationen dasselbe Register beanspruchen oder nicht. Solcherlei Konflikte sollten eigentlich von der darunterliegenden Hardware selbst erkannt und richtig behandelt werden. Sollte dies aber nicht der Fall sein, dann muß die Möglichkeit solcher Konflikte dem Firmwareprogrammierer im Handbuch mitgeteilt werden. Diese Vorgangsweise bietet sich auch als Notlösung für solche Konflikte an, die strenggenommen zwar nicht zu den Konflikten *dritter Art* gehören, aber für eine Behandlung im Compiler zu komplex sind.

Abschließend noch eine grundsätzliche Bemerkung: Insgesamt soll die Diplomarbeit eine Studie darstellen, welche die *Machbarkeit* anhand eines Prototyps zeigt, Möglichkeiten und Varianten diskutiert und die wichtigsten in der Praxis vorkommenden Fälle abdeckt. Hingegen wurde bei der Realisierung keinerlei Wert auf Optimierungen aller Art, wie etwa Geschwindigkeit oder Speicherplatz, gelegt. Auch die Abdeckung *aller* auch noch so ausgefallenen Möglichkeiten würde den Rahmen der Diplomarbeit bei weitem übersteigen.

Im nächsten Kapitel wird nun die Struktur des Firmwarecompilers in ihrer Gesamtheit vorgestellt sowie die dafür notwendige Spracherweiterung MDL/BED beschrieben. Das letzte Kapitel schließlich analysiert zusammenfassend die Anwendbarkeit der vorgestellten Methoden im Bereich des Hardware/Software-Codesign sowie im Zuge der High-Level-Synthese.

Kapitel 2

Spracherweiterung und Generierung

Die Spracherweiterung MDL/BED soll also jene Informationen enthalten, die notwendig sind, damit der Generator die hardwareabhängigen Teile des Firmwarecompilers generieren kann. Doch bevor dieser Vorgang im Detail erläutert wird, soll die Struktur des Firmwarecompilers erläutert werden, vor allem die hardwareunabhängigen Teile.

2.1 Frontend und Intermediate Representation

Wie die meisten Compiler läßt sich auch der zu generierende Firmwarecompiler grob in Frontend und Backend unterteilen. Durch eine geeignete Wahl der Intermediate Representation, in weiterer Folge als IR abgekürzt, können Frontend und Backend so entkoppelt werden, daß das Frontend prinzipiell hardwareunabhängig wird. Aus diesem Grund wird zunächst das Design der IR vorgestellt, dann erst wird darauf aufbauend das Frontend kurz erläutert.

Intermediate Representation

Beim Design einer IR hat man drei grundsätzliche Möglichkeiten:

Orientierung der IR an der Frontendsprache: In diesem Fall stellt die IR ein Spiegelbild der grammatikalischen Struktur der Frontendsprache dar. Dies ermöglicht die einfachste Implementierung des Frontend, weil die IR-Struktur 1:1 bei der Übersetzung des Mikroprogrammes aufgebaut werden kann.

Orientierung der IR an der Hardware: Dieser Ansatz bewirkt analog dazu ein relativ einfaches Backend.

Unabhängiges Design der IR: In diesem Fall müssen sowohl Frontend als auch Backend die mitunter komplexe Zuordnung zur IR-Struktur enthalten. Dennoch hat dieser Ansatz große Vorteile, was die Flexibilität betrifft, wie in weiterer Folge noch erläutert wird.

Da gerade die Abhängigkeit des Firmwarecompilers von der Hardware das große Problem ist, ergibt sich offensichtlich, daß es völlig widersinnig wäre, die zweite Variante zu wählen: In diesem Fall wäre die IR ja als stabile Basis zum hardwareabhängigen, generierten Backend völlig ungeeignet.

Die erste Variante ist an sich nicht grundsätzlich schlecht, hat aber den Nachteil, daß die IR von der verwendeten Firmwareprogrammiersprache abhängig ist. Darüberhinaus neigt man zu relativ komplexen IR-Strukturen, wenn man die Grammatik der Firmwareprogrammiersprache in der IR nachzubilden versucht. Dies wiederum erschwert die Referenzierung der IR-Strukturen und ihrer Attribute im Zuge der Baumgrammatik: Um die vom Frontend vorgegebenen Implementierungsdetails der IR zu verstecken, muß man mittels Makros[1] eine Kapselung der IR vornehmen. Zugriffe auf die Attribute könnten dann etwa so aussehen: `REG_NAME(...)`, `RIGHT_CHILD(node)` oder `EXPR_OP(VAR_INDEX(x))`. Dennoch bleibt immer eine gewisse gegenseitige Abhängigkeit von Frontend und Backend bestehen, die sich dann besonders unangenehm auswirkt, wenn es viele verschiedene Frontend-Backend-Kombinationen gibt.

Aus diesen Gründen wurde die dritte Variante gewählt, das Design der IR ist die stabile Basis, Frontend und Backend müssen sich daran orientieren. Abbildung 2.1 auf Seite 25 zeigt die Zusammenhänge anschaulich: Das Frontend muß allen möglichen Syntaxelementen mindestens eine IR-Struktur zuordnen. Es kann ein Syntaxelement je nach Attributen auch auf mehrere IR-Strukturen abgebildet werden oder mehrere Syntaxelemente auf eine IR-Struktur. Umgekehrt müssen jedoch nicht alle IR-Strukturen eine Syntax-Entsprechung besitzen. Wie später erläutert, referenziert die Baumgrammatik in der MDL/BED die IR-Strukturelemente und definiert damit jene IR-Strukturen, die eine Entsprechung in einer Hardwarefunktion finden. Damit wird also die Abbildung dieser IR-Strukturen auf die Mikrooperationen, also die möglichen Hardwarefunktionen des Prozessors, gesteuert. Dabei muß jede Mikrooperation einer oder (bei unterschiedlichen Attributen) auch mehreren IR-Strukturen entsprechen, während umgekehrt nicht alle IR-Strukturen auf eine Mikrooperation abbildbar sein müssen. Allerdings kann auch eine IR-Struktur je nach Attributen auf mehrere Mikrooperationen abgebildet werden.

Durch die Abbildung des Frontends auf die IR und die Referenzierung der IR durch die Baumgrammatik der MDL/BED wird implizit auch die hardwarespezifische Sprachuntermenge der Firmwareprogrammiersprache festgelegt: Jedes Syntaxelement mit mindestens einem Pfad zu einer Mikrooperation gehört zur hardwarespezifischen Sprachuntermenge dazu. Dabei können zwei problematische Situationen auftreten:

1. Das IR-Element 3 in der Abbildung 2.1 wurde von einem Syntaxelement erzeugt, besitzt aber keine zugeordnete Hardwarefunktion. Dies ergibt eine Fehlermeldung

[1] Die ebenfalls vom Frontend zur Verfügung gestellt werden müssen!

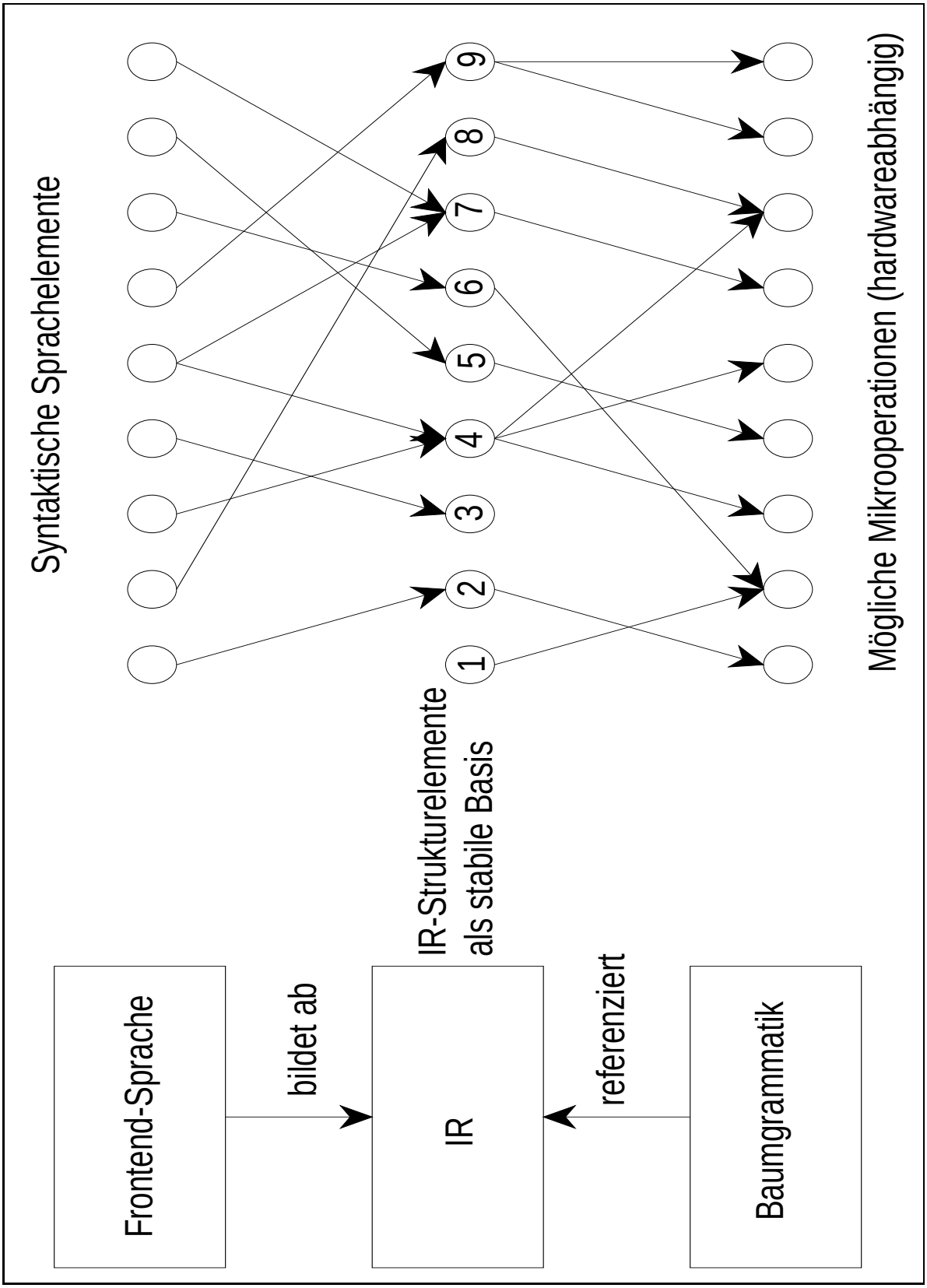

Abbildung 2.1: Die IR als stabile Basis.

dahingehend, daß das verwendete Syntaxelement auf der speziellen Hardware unzulässig ist.

2. Das IR-Element 1 hingegen entspricht zwar einer Hardwarefunktion, das wurde in der Baumgrammatik definiert. Leider gibt es aber kein Syntaxelement, welches das gegebene IR-Element erzeugen kann und somit keine Möglichkeit, die zugehörige Hardwarefunktion von der Firmwaresprache aus anzuwenden. Das deutet auf eine unglückliche Formulierung der IR-Referenz hin: Will man die Frontendsprache unverändert lassen, muß man die IR-Referenz umformulieren. Dies ist stets möglich, da die IR-Struktur `[Builtin]` die Formulierung all jener Hardwarefunktionen gestattet, welche die Mächtigkeit der Frontendsyntax übersteigen[2]. In der Abbildung 2.1 ist eine alternative Möglichkeit über IR-Element 6 angedeutet.

Da also die IR die Basis des Systems ist, würde eine Änderung der IR-Struktur eine Anpassung aller Frontends und aller Backends erfordern. Daher ist das Design der IR sehr sorgfältig vorzunehmen: Es muß so allgemein und flexibel sein, daß (nahezu) alle geeigneten Frontendsprachen und alle denkbaren Hardwarekonstellationen kompatibel sind. Als relativ einfache Struktur wurde aus diesen Gründen der *binäre Baum* gewählt. Als Variante wurde auch der allgemeine Baum untersucht, der zwar bei listenartigen Strukturen Vorteile besitzt, jedoch aufgrund der vielfältigen Freiheitsgrade in der Struktur zu Problemen bei der Referenzierung der IR im Rahmen der Baumgrammatik geführt hat.

Die Struktur der IR nun im Detail: Jeder Knoten des binären Baumes enthält neben den beiden Zeigern auf den linken und rechten Nachfolger noch zwei Informationen: Den *Typ* des Knotens und sein *Attribut*. Das Attribut enthält die semantische Information des Knotens, der Typ die syntaktisch-strukturelle. Beide Informationen sind als ganze Zahlen implementiert, wobei über Tabellen die ausführliche Attributinformation referenziert werden kann. Bei einem Register zum Beispiel kann über die Nummer der Name des Registers aus der Tabelle abgefragt werden. Da dies aber nur eine Darstellungsfrage ist, die überdies implementierungstechnisch leicht zu lösen ist, kann in weiterer Folge dieser Unterschied ignoriert werden. Daher wird in weiterer Folge z.B. „... der Name des Registers..." statt „... die Registernummer..." geschrieben.

Wiewohl die *Struktur* der IR völlig unabhängig sowohl von der Hardware als auch von der Frontendsprache ist, so sind dennoch die möglichen *Typen* der Knoten und Unterbäume an den *grundsätzlichen Erfordernissen* mikroprogrammierter Hardware orientiert. Folgende Typen sind vorgesehen:

`Area` Beschreibt einen Bereich, wobei Offset und Länge als rechter und linker Nachfolger vom Typ `[Constant]` auftreten, der Knoten selbst enthält kein Attribut.

`Assignment` Auch dieser Knoten besitzt kein Attribut, linker Nachfolger ist das Ziel der Zuweisung, rechter Nachfolger der zuzuweisende Ausdruck.

[2]Solange die Frontendsyntax Builtin-Funktionen zuläßt. Ansonsten kann es zu einer unzureichenden Frontendsyntax kommen und eine Erweiterung derselben wird unvermeidlich.

`Binop` Attribut ist der Typ der Operation, linker und rechter Nachfolger sind die zu verknüpfenden Ausdrücke.

`Builtin` Attribut ist der Name der Builtin-Funktion, der linke Nachfolger ist ein einzelner Parameter oder eine Liste von Parametern, es gibt keinen rechten Nachfolger. Die Builtin-Funktion ist unter anderem das „Hintertürchen“ für all jene Hardwarefunktionen, die keine syntaktische Entsprechung besitzen. Wenn irgendeine Frontendsyntax die Builtin-Funktion erlaubt, dann können bereits sicher alle möglichen Hardwarefunktionen beschrieben werden, wenn auch mitunter nicht sehr komfortabel.

`Call` Enthält kein Attribut und keinen rechten Nachfolger, der linke Nachfolger beschreibt das Sprungziel.

`Condition` Kein Attribut, linker Nachfolger ist die Bedingung, rechter Nachfolger ist stets ein Knoten vom Typ `[Then_Else]`.

`Constant` Attribut ist der Wert der Konstanten, keine Nachfolger.

`Jump` Wie bei `[Call]` kein Attribut und kein rechter Nachfolger, der linke Nachfolger beschreibt das Sprungziel.

`Label` Attribut ist der Name des Labels, keine Nachfolger.

`List` Mit Hilfe dieses Knotens werden listenartige Strukturen linksrekursiv beschrieben: Der rechte Nachfolger ist stets ein Listenelement, der linke Nachfolger kann entweder ein Knoten vom Typ `[List]` sein oder ein Listenelement. Im zweiten Fall ist das Ende der Liste erreicht. Abbildung 2.2 veranschaulicht diese Struktur.

`Return` Keine Nachfolger, kein Attribut.

`Then_Else` Linker Nachfolger ist der `Then`-Zweig, rechter Nachfolger der `Else`-Zweig der übergeordneten Bedingung. Der rechte Nachfolger kann auch fehlen. Der Knoten besitzt kein Attribut.

`Unop` Attribut ist der Typ der Operation, linker Nachfolger der Operand. Da es sich um eine unäre Operation handelt, gibt es keinen rechten Nachfolger.

`Variable` Attribut ist der Name des Registers. Die Bezeichnung „Variable“ kommt daher, daß in der Denkweise der Mikroprogrammierung ein Register als Variable betrachtet wird. Dabei handelt es sich aber eben nicht um frei definierbare Variablen im Sinne einer Hochsprache, sondern um vorgegebene Hardwareressourcen. Der linke Nachfolger ist der Index[3] des Registerfeldes. Der rechte Nachfolger kennzeichnet den Bitbereich des angesprochenen Registers. Typische Unterbäume beginnen entweder mit einem Knoten vom Typ `[Area]` bei direkten Adressierungsarten oder mit einem Knoten vom Typ `[Variable]` bei indirekten Adressierungsarten über Pointer-Register. Es sind aber auch beliebig komplexe Substrukturen wie z.B. `[Binop]` möglich.

[3]Dabei sind auch mehrdimensionale Registerfelder leicht zu beschreiben, etwa als `[List]`-Strukturen von `[Area]`-Knoten, ggf. sogar durchmischt mit beliebigen indirekten Adressierungsarten.

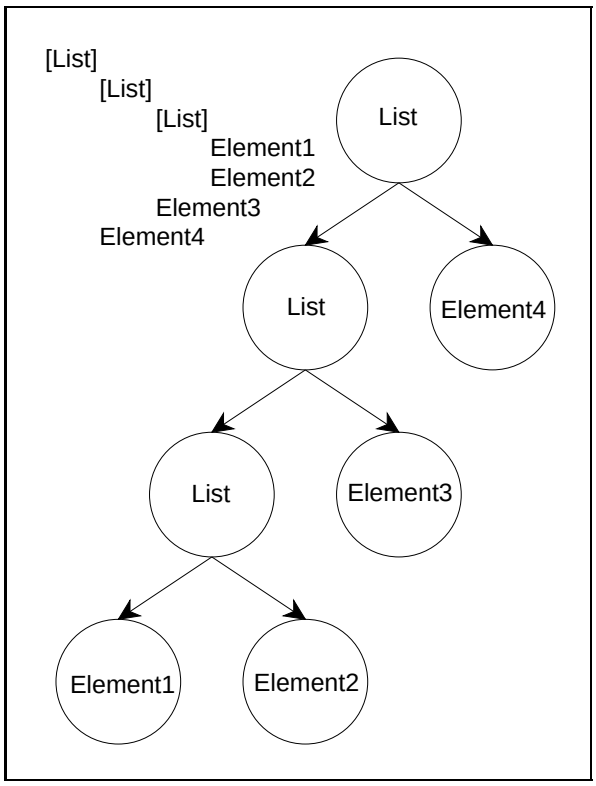

Abbildung 2.2: Unter Verwendung des Knotentyps `[List]` können listenartige Strukturen linksrekursiv dargestellt werden. Links oben im Bild ist die zugehörige textuelle Darstellung der IR-Struktur angegeben, wie sie später noch öfters verwendet werden wird.

Betrachtet man die so definierten möglichen IR-Strukturen, so gibt es außer der impliziten Annahme linksrekursiver Verschachtelung[4] und den wenigen obgenannten Einschränkungen[5] keine Beschränkungen der Flexibilität.

So können etwa durch beliebige Kombination indirekter und direkter Adressierungsarten sowohl beim Index als auch beim Bitbereich der Register auch die komplexesten Adressierungsarten beschrieben werden, zur Not immerhin noch unter Zuhilfenahme der Builtin-Funktionen. Man könnte etwa eine indirekte Adressierungsart über *zwei* Pointer-Register der Form `reg(p0,p1)` durch eine Erweiterung der Frontendsyntax handhaben, sodaß das Frontend die richtige IR-Struktur[6] erzeugt. Noch einfacher ist aber die Beschreibung unter Zuhilfenahme einer Builtin-Funktion, etwa `reg(quad(p0,p1))`, womit der bereits bestehenden Frontend-Syntax Genüge getan werden kann: Pseudofunktionen zur Implementierung komplexerer Adressierungsarten werden erst im Backend umgesetzt und sind daher unabhängig vom Frontend. Auch spezielle komplexere arithmetisch-logische Operationen sind Kandidaten für die Verwendung von Builtin-Funktionen. Als Minimalforderung an die Frontendsyntax ergibt sich daraus jedoch die Forderung nach einem Sprachelement für die Builtin-Funktion.

[4]Nicht nur bei Listen, sondern auch z.B. bei mehrfachen binären Operationen.
[5]Wie z.B., daß der rechte Nachfolger einer `[Condition]` stets ein `[Then_Else]` ist.
[6]In diesem Fall eine Liste mit zwei Knoten vom Typ `[Variable]`.

Die Anwendung der Builtin-Funktion bewirkt also eine stärkere semantische Entkoppelung von Frontend und Backend. Wiewohl *syntaktisch entkoppelt*, können Frontend und Backend voneinander *semantisch abhängig* sein: Im obigen Beispiel der indirekten Adressierungsart `reg(p0,p1)` einer speziellen Hardware müßte im Frontend eine syntaktische Ausprägung dieser Adressierungsart definiert werden, um diese sodann in *jene* IR-Struktur umsetzen, die von der Baumgrammatik für diese Adressierungsart *erwartet* wird[6]. Die Verwendung von zusätzlichen Builtin-Funktionen hat jedoch *gar keinen* Einfluß auf das Frontend, solange die Frontendsprache ein Sprachelement für die Builtin-Funktion enthält.

Bei genauer Überlegung erkennt man, daß man leicht mit den Typen `[Builtin]`, `[Variable]`, `[List]`, `[Label]` und `[Constant]` das Auslangen finden könnte, wenn man die übrigen Typen, je nach Erfordernissen, unter dem Typ `[Builtin]` als Pseudo-Builtin-Funktionen implementiert. Durch die Definition der zusätzlichen Typen `[Area]`, `[Assignment]`, `[Binop]`, `[Call]`, `[Condition]`, `[Jump]`, `[Return]`, `[Then_Else]` und `[Unop]` wird jedoch die semantische Struktur eines IR-Baumes deutlicher hervorgehoben, was wiederum die Referenzierung im Zuge der Baumgrammatik erleichtert.

Frontend

Nachdem die IR-Struktur exakt definiert wurde, können darauf aufbauend die Aufgaben des Frontend exakt festgelegt werden. Folgende Punkte müssen im Bereich des Frontend behandelt werden:

Bezeichner für Variablen und Funktionen: Im Gegensatz zu Compilern von Hochsprachen sind diese Bezeichner von der Hardware vorgegeben: Variablenbezeichner sind eigentlich Namen für Register, Funktionen beschreiben bestimmte Hardwarefunktionalitäten, auch Builtin-Funktionen genannt. Daher wird auch vom Firmwarecompiler keine Symboltabelle aufgebaut, sondern es wird eine hardwareabhängige Deskriptortabelle *konsultiert*, um zu prüfen, ob und wie bestimmte Bezeichner für die jeweilige Hardware definiert sind. Diese Deskriptoren werden vom Generator der MDL-Beschreibung entnommen und dem Frontend zur Verfügung gestellt.

Aliasdefinitionen: Aliasnamen für Register und Registerteile werden ebenfalls hardwarespezifisch definiert und müssen dem Frontend z.B. in Tabellenform zur Verfügung stehen. Das Frontend kann dann die Aliasnamen zu den eigentlichen Hardwarebezeichnern umsetzen. Dies erlaubt eine hohe Flexibilität bei der Definition von Hardwarenamen, ohne die IR-Referenzierungen im Backend zu komplex werden zu lassen.

Zusätzlich deklarierte Bezeichner: Die meisten Frontendsprachen erlauben die Deklaration zusätzlicher oder alternativer Bezeichner für bestimmte Hardwarebezeichner. Dabei muß das Frontend die zusätzlich deklarierten Bezeichner selbst zu den Originalbezeichnern umsetzen, das Backend darf davon nicht berührt werden.

Adreßinformation: Label, Marken und Sprungziele müssen ebenfalls vom Frontend vorbearbeitet werden. Dabei muß im Sinne der Modularisierung auch ein Mechanismus vorgesehen werden, der die Verwendung externer Label ermöglicht. Zu diesem Zweck wird als Information in den Knoten von Typ `[Label]` nur ein Verweis in die Label-Tabelle abgelegt. In der Labeltabelle sind dann die für den Linker und Locater relevanten Informationen gespeichert, die im Zuge der Codeerzeugung noch ergänzt werden, z.B. in welchem Bereich des Mikrobefehlsfeldes dann die endgültig berechnete Adresse als Bitmuster abgelegt werden soll. Ebenso wie die IR ist auch die Labeltabelle unabhängig vom Front- und Backend definiert, es handelt sich dabei um eine einfache Tabellenstruktur, auf die aber hier nicht weiter eingegangen werden soll.

Intermediate Representation: Wenn das Mikroprogramm syntaktisch richtig ist, wird vom Frontend eine IR aufgebaut, die exakt den Spezifikationen genügen muß, die semantische Information wird in den Attributen hinterlegt. Im Unterschied zu Hochsprachen, wo ein großer Syntaxbaum für ein ganzes Programm-Modul aufgebaut werden muß, wird bei der Firmwareprogrammierung ein kleiner IR-Baum für jede Mikrooperation aufgebaut. Da mit Firmware i.a. sehr Hardware-nahe programmiert wird, erlauben gängige Firmwareprogrammiersprachen stets eine exakte Zuordnung, welche Syntaxelemente zu einer Mikrooperation gehören.

Noch eine Bemerkung zur Umformung der Aliasnamen: Wie in [4] beschrieben, ermöglicht die `ALIAS`-Definition nicht nur eine simple Umbenennung von Registernamen, sondern ermöglicht gemeinsam mit der `MAP`-Deklaration eine komplexe Strukturierung der Register. Daher wird bei der Ersetzung eines Alias-Namens ein IR-Teilbaum durch einen anderen ersetzt. Die Regeln der Alias-Umsetzung sind in [4] sehr ausführlich beschrieben und werden hier daher nicht mehr angeführt. Wichtig ist als Fazit nur, daß die Struktur, die durch die Umsetzung eines Alias-Namens entsteht, stets so beschaffen ist, daß sie auch unter Verwendung eines mittels `REGISTER`-Definition deklarierten Namens hätte erzeugt werden können. Damit wiederum ist aber bewiesen, daß eine Teilbaum-Ersetzung in allen denkbaren Fällen widerspruchsfrei möglich ist. Die Information für diese Umsetzungen ist bereits in MDL/HD enthalten und muß, analog zu den Deskriptortabellen, in Form von Alias-Tabellen dem Frontend zur Verfügung gestellt werden.

In [1] ist die Implementierung eines Frontends auf einer UNIX-Plattform unter Verwendung der Compiler-Generator-Werkzeuge *lex* und *yacc* vorgestellt. Dabei handelt es sich um die Firmwareprogrammiersprache MPL-Y, die in [9] genau spezifiziert ist.

2.2 MDL-Spracherweiterung durch Baumgrammatik

Mit der stabilen Definition der IR wird das Frontend unabhängig von der Hardware, es benötigt nur die Deskriptor- und Aliastabellen zur Umsetzung der Bezeichner. Damit muß als einziger Hardware-abhängiger Teil des Compilers das Backend generiert werden. Dieses Backend hat nun die Aufgabe, eine IR einzulesen und mit den Attributen dieser IR semantische Aktionen durchzuführen. Dabei wird zunächst die IR je nach Struktur und Inhalt den entsprechenden Mikrooperationen zugeordnet, um dann hardwarebedingte Konflikte zwischen mehreren Mikrooperationen zu überprüfen und den Bitcode für den gesamten Mikrobefehl zu erzeugen. Und genau diese *Zuordnung* der IR-Struktur zu bestimmten Mikrooperationen ist die schwierigste Aufgabe des Backends.

Im Bereich der Compiler für Hochsprachen gibt es zu diesem Zweck *Backend-Generatoren*, die als Eingabe eine *Baumgrammatik* erhalten und daraus das Backend eines Compilers generieren. Die Baumgrammatik beschreibt dabei, welche Unterbäume des IR-Syntaxbaumes auf welche Maschinenbefehle abgebildet werden. Dieser Vorgang wird als *Tree-Parsing* bezeichnet und ist nicht immer eindeutig: Mit Kostenvektoren wird dann versucht, optimale Lösungen zu finden. Dabei werden heuristische Methoden angewandt, da die entstehenden Optimierungsaufgaben nicht geschlossen lösbar sind. Einige Anregungen zu diesem Thema finden sich in [11] und [16].

Offensichtlich erfordert auch die Generierung des Backends des Firmwarecompilers eine Baumgrammatik, welche die Zuordnung bestimmter IR-Strukturen zu den Mikrooperationen beschreibt. Dabei gibt es grundsätzlich drei Möglichkeiten:

1. Verwendung eines fertigen Backend-Generators: Dabei muß die Baumgrammatik für den Firmwarecompiler direkt in einer Form beschrieben werden, die ein bestehender Backend-Generator als Eingabe akzeptiert. Da aber die bestehenden Backend-Generatoren sehr gezielt auf Hochsprache-Compiler ausgelegt sind, kann es bei der Formulierung firmwarespezifischer Aktionen zu Problemen kommen. Der Nachteil liegt auf der Hand: Man kann keine selbstdefinierte Baumgrammatik verwenden, die für den Einsatz mit Firmwarecompilern eventuell besser geeignet ist.

2. Indirekte Einbindung eines bestehenden Backend-Generators: Es wird zwar eine eigenständige Beschreibungsform definiert, die dann aber in eine solche Form umgesetzt wird, daß sie einem bestehenden Backend-Generator als Eingabe dienen kann. Der Vorteil liegt in der Verwendung bestehender Programme, der Nachteil ist ein gewisser „Overkill“, da die meisten Backend-Generatoren viel mächtiger sind, als es für die Generierung eines Firmwarecompiler-Backends notwendig wäre.

3. Entwicklung eines speziellen Firmwarecompiler-Backend-Generators: Dieser Ansatz wurde bevorzugt, da auch die Umsetzung einer selbstgewählten Beschreibungsform zu einer Eingabe für einen bestehenden Backend-Generator bereits sehr viel Komplexität enthält. Darüberhinaus ist im Firmwarecompiler-Backend kein *Tree-Parsing*, sondern nur ein *Tree-Pattern-Matching* erforderlich. Wie bereits erwähnt,

S36	syntax : GRAMMAR [gram_head] { rule } ENDGRAMMAR

Abbildung 2.3: Ein Baumgrammatik-Teil im Zuge der MDL-Beschreibung wird stets von den Schlüsselwörtern GRAMMAR und ENDGRAMMAR eingeschlossen. Im optionalen gram_head ist die Definition der Typen der semantischen Attribute der Token angeführt, danach kann eine, keine oder es können beliebig viele Regeln (rule) deklariert werden.

wird ja mit Firmware i.a. sehr Hardware-nahe programmiert und daher ermöglichen gängige Firmwareprogrammiersprachen stets eine exakte Zuordnung, welche Syntaxelemente zu einer Mikrooperation gehören. Daher wird *ein* kleiner IR-Baum für jede Mikrooperation aufgebaut, und es ist keine Erkennung von Unterbäumen im IR-Baum mehr notwendig: Jeder „Mini"-IR-Baum entspricht genau einer Mikrooperation. Allerdings werden i.a. sehr wohl mehrere Mikrooperationen in einem Mikrobefehl vereint und gemeinsam in einer Befehlszeile codiert.

Es bleibt also die Aufgabe des *Tree-Pattern-Matching*, also die Überprüfung, ob ein IR-Baum mit der Baumgrammatik verträglich ist. Eine Übereinstimmung liegt dann vor, wenn

- die Struktur übereinstimmt, also die Anzahl und Anordnung der Knoten im binären Baum.
- alle Knoten vom Typ her übereinstimmen.
- alle Attribute in den Knoten die vorgegebenen Bedingungen erfüllen.

Daher muß die Baumgrammatik genau diese drei Sachverhalte beschreiben.

Im Rahmen einer MDL-Beschreibung können beliebig viele Teile der Baumgrammatik enthalten sein, günstigerweise ordnet man die jeweiligen Baumgrammatik-Stücke um die zugehörigen Mikrooperationen an. Jedes Stück Baumgrammatik beginnt mit dem Schlüsselwort GRAMMAR und endet mit dem Schlüsselwort ENDGRAMMAR, bei der Bearbeitung durch den Backend-Generator werden diese Stücke dann zu einer gesamten Grammatik umgesetzt, sodaß sich logische Zusammenhänge über mehrere Baumgrammatik-Stücke erstrecken können. Abbildung 2.3 zeigt die zugehörige Syntax. In Anhang A ab Seite 62 befindet sich dann eine geschlossene Darstellung jener Syntaxregeln, die die Spracherweiterung beschreiben.

Jede Regel (rule) ist eine Aufzählung von verschiedenen möglichen Formulierungen. Diese Formulierungen beschreiben entweder den gesamten Syntaxbaum einer Mikrooperation (ROOT) oder nur einen Teilbaum (SUBTREE). Die Definition der Teilbäume hat zweierlei Sinn:

S38

```
rule
  : ( ROOT | SUBTREE Identifier )
    ":" formulation { "|" formulation } END
```

Abbildung 2.4: Eine Regel, die mit dem Schlüsselwort `ROOT` beginnt, beschreibt die Formulierungen der gesamten IR-Struktur einer Mikrooperation. Beginnt eine Regel jedoch mit dem Schlüsselwort `SUBTREE`, gefolgt von einem `Identifier`, so wird ein Teilbaum der IR-Struktur beschrieben, der `Identifier` enthält den eindeutigen Namen dieses Teilbaumes. Danach folgt eine Aufzählung von Formulierungen, abgeschlossen wird eine Regel mit dem Schlüsselwort `END`.

S39

```
formulation
  : { gram_line }+
```

Abbildung 2.5: Jede Formulierung besteht aus einer Reihe von Zeilen (`gram_line`), wobei jede Zeile genau einen Knoten im IR-Baum referenziert.

1. Ein Unterbaum, der mehrmals in der IR-Struktur einer oder verschiedener Mikrooperationen vorkommt, wird auf diese Weise nur einmal definiert und mehrmals in die Hauptstruktur eingebunden.

2. Durch die Ausgliederung von Unterbäumen kann die Übersichtlichkeit der Baumgrammatik erhöht werden, indem nicht allzutiefe Strukturen angegeben werden.

Abbildung 2.4 beschreibt die zugehörige Syntaxregel.

Jede Formulierung beschreibt eine mögliche Ausprägung eines IR-(Teil-)-Baumes und besteht aus einer Reihe von Zeilen (`gram_line`), wobei jede Zeile einen Knoten des IR-Baumes referenziert. Abbildung 2.5 zeigt die zugehörige Syntaxbeschreibung. Die Syntax der Referenzierung eines Knotens im IR-Baum wird schließlich in Abbildung 2.6 dargestellt. Dabei ist zu beachten, daß von der Eingabe her die Einrückungstiefe[7] der Zeile die Tiefe des Knotens im Baum ausdrückt. Dies bewirkt, daß die Baumgrammatik auch rein optisch die *Baumstruktur* wiederzugeben imstande ist. Im Zuge der lexikalischen Analyse wird jedoch die durch die Einrücktiefe der Zeile bestimmte Tiefe des Knotens im IR-Baum in einen Integer zwischen eckigen Klammern umgeformt, der nun grammatikalisch jede `gram_line` zwingend einleitet. Für jeden Knoten wird entweder sein Typ festgelegt, oder der Name einer Unterbaumbeschreibung angegeben, die dann die weitere Substruktur des Baumes von dem entsprechenden Knoten weg beschreibt. Damit ist die reine Beschreibung der Baumstruktur abgeschlossen. Um jedoch auf die semantischen Attribute des IR-Baumes zugreifen zu können, kann man dem Knoten einen symbolischen

[7] Präzise ausgedrückt, die Anzahl der unmittelbar davorliegenden Tabulatoren.

S40

```
gram_line
  : "[" integer "]"
    [ LEFT | RIGHT ] ( Identifier | "[" Identifier "]" )
    [ IS Identifier ]
    [ VALUE String_constant ]
    [ TO  ( integer { "," integer } )
        | ( "(" integer "," integer ")" ]
```

Abbildung 2.6: Jede Zeile referenziert genau einen Knoten des IR-Baumes. Zu Beginn steht eine ganze Zahl zwischen eckigen Klammern, die die Tiefe des Knotens im IR-Baum angibt. Diese Zahl wurde von der lexikalischen Analyse aus der Einrückung der Zeile erzeugt. Danach folgt optional eines der Schlüsselwörter `LEFT` oder `RIGHT`. Damit wird festgelegt, ob der aktuelle Knoten der linke oder rechte Nachfolger seines unmittelbaren Vorgängers ist. Implizit wird nach einem Knoten stets der linke vor dem rechten Nachfolger angegeben. Wenn jedoch nur einer der beiden Nachfolger angegeben wird, könnten bei einer tieferen Baumstruktur Mehrdeutigkeiten entstehen, die auf diese Weise aufgelöst werden können. Nun folgt der Typ des Knotens zwischen eckigen Klammern. Wird hingegen an der Stelle des Knotens ein Unterbaum eingefügt, so ist der Name des Unterbaumes statt dem Knotentyp angegeben. In jedem Fall kann dem Knoten oder dem Unterbaum mit dem optionalen Schlüsselwort `IS` ein Name gegeben werden, der der Verwendung der Attribute im Zuge der semantischen Aktionen dient. Das ebenfalls optionale Schlüsselwort `VALUE` definiert eine Menge von möglichen Attributwerten für einen bestimmten Knoten. Diese Werte werden aber nicht in Form der Integer angegeben, die tatsächlich im IR-Baum als Attribute enthalten sind, sondern in Form der Original-Zeichenketten, z.B. `"+"` oder `"reg"`. Dies vereinfacht die IR-Referenzierung im Zuge der Baumgrammatik. Bei der Umsetzung der Baumgrammatik zum Parser, der dann die IR einliest und überprüft, werden diese Zeichenketten ganz analog zu dem entsprechenden Vorgang im Frontend mittels Deskriptor- und Aliastabellen zu den entsprechenden Integern umgeformt. Das letzte Schlüsselwort `TO` ist auch optional. Es ermöglicht beim Auftreten eines zugelassenen Attributes aus der mit `VALUE` definierten Menge den Ersatz des ursprünglichen Attributwertes durch einen Integer-Wert. Dies vereinfacht die im Zuge der semantischen Aktionen typischen Zuweisungen und steigert so insgesamt die Übersichtlichkeit der Baumgrammatik. Anstelle einer simplen Aufzählung der zuzuweisenden Werte ermöglicht die zweite Form der `TO`-Definition in geschwungenen Klammern die Angabe eines Startwertes sowie einer Schrittweite, da mit dieser effizient abkürzenden Symbolik sehr viele der in der Praxis vorkommenden Fälle abgedeckt werden können.

Namen geben, man kann eine Menge erlaubter Attributwerte angeben und man kann beim Auftreten eines Elementes aus dieser Menge einen bestimmten Wert das ursprüngliche Attribut im Knoten ersetzen lassen. Details dazu werden im folgenden Abschnitt 2.3 erklärt.

Die so beschriebene Baumgrammatik wird dann vom Generator eingelesen, der daraus einen Backend-Parser generiert. Dieser Backend-Parser liest eine IR-Beschreibung ein, muß die mittels Baumgrammatik beschriebenen IR-Strukturen erkennen und dann die semantischen Aktionen ausführen, die bei den Baumgrammatik-Regeln angeführt sind.

2.3 Attribute und Semantische Aktionen

Die semantischen Aktionen in einem Backend haben drei wesentliche Aufgaben:

1. Syntaxorientierte Semantikprüfungen, wenn zum Beispiel eine Konstante nur eine Potenz von 2 sein darf.
2. Hardwarebedingte Semantikprüfungen, hier vor allem die Vereinigung mehrerer Mikrooperationen in einem Mikrobefehl ohne Ressourcenkonflikte.
3. Die Bitcodierung des Mikrobefehlsfeldes.

Mit der Baumgrammatik wurden diese semantischen Aktionen an das Auftreten bestimmter IR-Strukturen gekoppelt. Wie nun die semantischen Aktionen *selbst* implementiert werden und wie sie auf die Attribute der IR-Bäume zugreifen, soll hier gezeigt werden. Um dabei auch die im vorigen Abschnitt vorgestellte Syntax der Baumgrammatik zu vertiefen, wird in weiterer Folge das Beispiel aus dem Anhang B.1 ab Seite 66 herangezogen. Anhand einzelner herausgegriffener Teile werden die typischen Möglichkeiten der Baumgrammatik beschrieben.

Beispiel 2.1 zeigt die Beschreibung einer erlaubten IR-Struktur für bestimmte arithmetische Operationen. Die Details sind beim Beispiel selbst beschrieben, wesentlich ist aber das Konzept, wie die Übergabe der Attribute des IR-Baumes an die semantischen Aktionen stattfindet. Nicht alle Attribute des IR-Baumes müssen von den semantischen Aktionen verwendet werden, manche Knotentypen haben ja überhaupt keine Attribute. Will man jedoch einen Attributwert im Zuge der semantischen Aktionen verwenden, sei es zur Überprüfung der Hardwarekonflikte oder für die Codeerzeugung, so muß diesem Attribut mit dem Schlüsselwort `IS` ein Name zugewiesen werden. Unter diesem Namen kann das Attribut dann im C-Code der semantischen Aktionen wie eine C-Variable verwendet werden. Dabei muß aber dem Namen ein `$` vorangestellt werden, um ihn von den anderen C-Variablen zu unterscheiden.

Dabei ist zu beachten, daß die Angabe der semantischen Operationen in einer Programmiersprache erfolgt, im konkreten Fall wurde C gewählt. Dabei wird der C-Code vom Generator an den C-Compiler weitergereicht, lediglich die Metasymbolik für den Zugriff auf die Attribute wird vorher aufgelöst. Grundsätzlich kann jede Computersprache zur Beschreibung der semantischen Aktionen verwendet werden, also auch etwa PASCAL oder MODULA. Der Programmcode wird eben an den entsprechenden Compiler weitergereicht, nachdem die Attribut-Referenzen aufgelöst worden sind. Daher wurde auch die Grammatik der semantischen Aktionen nicht in den bisherigen Syntaxdiagrammen zur Baumgrammatik aufgezählt. Für den Generator ist nur wichtig, daß

1. die semantischen Aktionen zwischen geschwungenen Klammern eingeschlossen sind und
2. die Ausdrücke der Form `$name` umgeformt werden müssen auf Zugriffe auf die Attribute des IR-Baumes in der jeweiligen Programmiersprache, in C z.B. `root->n_succ2->n_value` statt `$opcode`.

```
ROOT
/*
<AC> = AND (<AC>,<B>);;
<AC> = OR (<AC>,<B>);;
<AC> = XOR (<AC>,<B>);;
<AC> = NXOR (<AC>,<B>);;
*/
:   [Assignment]
        Operand IS AC_Op
        [Builtin] IS opcode VALUE "AND","OR","XOR","NXOR"
                          TO 0b10111, 0b10101, 0b10011, 0b10001
            [List]
                Operand IS AC_Op,
                Operand IS B_Op
    { /* conflict check */
      alu_conflict($AC_Op,$B_Op);
      /* encoding */
      encode("fgroup",$0b00);
      encode("ffunction",$opcode);
      adr_encode($AC_Op,$B_Op);
    }
.....
END
```

Beispiel 2.1: Die Referenz des gesamten IR-Baumes. Da die Regel mit dem Schlüsselwort `ROOT` beginnt, beschreibt sie eine mögliche Ausprägung des *gesamten* IR-Baumes. In diesem Beispiel ist nur eine Formulierung der Regel angeführt. Welche Mikrooperationen diese Formulierung beschreibt, ist in Form von Kommentaren angemerkt, es handelt sich um einfache arithmetisch-logische Funktionen. Der Wurzelknoten ist vom Typ `[Assignment]`, sein Attribut ist daher irrelevant. Der linke Nachfolger ist ein Unterbaum, der mit dem Namen `AC_Op` belegt wird. Der rechte Nachfolger ist eine Builtin-Funktion, die den Namen opcode erhält. Hinter `VALUE` stehen nun die für diesen Knoten erlaubten Attributwerte. Mit dem Schlüsselwort `IN` werden nun bei Auftreten der erlaubten Werte diese durch die angegebenen Binärkonstanten ersetzt. So wird etwa beim Auftreten des Attributes `XOR` dieses Attribut durch die Binärkonstante `0b10011` ersetzt. Im Zuge der semantischen Aktionen, zwischen den geschwungenen Klammern eingeschlossen, wird weiter unten mittels $opcode dieser zugewiesene Attributwert im C-Programmtext weiterverwendet. Die Builtinfunktion schließlich hat nur einen linken Nachfolger in Form einer Liste, die wiederum zwei Unterbäume `AC_Op` und `B_op` vom Typ `Operand` enthält. Die in den semantischen Aktionen verwendeten Unterprogramme sind in Anhang B.2 ab Seite 73 angeführt.

Somit kann die funktionale Beschreibung der semantischen Aktionen in einer vertrauten Hochsprache stattfinden, über die Metasymbolik erhält man Zugriff auf die Attribute des IR-Baumes.

Der Typ eines solchen elementaren Attributes ist daher stets ein Integer, es werden Unterprogramme zur Verfügung gestellt, die eine wechselseitige Umwandlung zwischen den Darstellungen eines Attributes als Integer oder als Zeichenkette erlauben. Wurde dem Attribut mit dem Schlüsselwort `TO` hingegen ein neuer Integer-Wert zugewiesen, so ist dieser enthalten und keine Umsetzung erforderlich. Im Beispiel 2.1 wurde das Attribut `opcode` des Builtin-Knotens bereits durch die zu codierenden Binärzahlen ersetzt. Da solche direkten Umwandlungen einen Großteil der semantischen Aktionen ausmachen, wurde eben das Schlüsselwort `TO` zur effizienten Darstellung eingeführt.

Bei der Darstellung der erlaubten Attribute hinter dem Schlüsselwort `VALUE` fällt auf, daß die textuellen Repräsentationen angegeben werden, und nicht die Integer. Dies dient der Vereinfachung und besseren Lesbarkeit der Baumgrammatik insgesamt. Bei der Umsetzung zum Backend-Parser werden jedoch diese Namen in die zugehörigen Nummern umgewandelt. Die Vorgangsweise ist dabei analog zum Frontend und dort wie hier gilt, daß es sich dabei letztendlich nur um eine Darstellungsfrage handelt. Daß diese nicht ganz trivial ist, zeigt die Umsetzung ganzer Alias-Strukturen in ihre Original-Formen. Dabei werden ganze Teilbäume durch andere ersetzt, die Vorgangsweise ist aber ebenfalls identisch zu der im Frontend, weshalb hier nicht mehr näher darauf eingegangen werden soll.

Etwas anders sieht die Situation bei den Unterbäumen aus. Wenn ein Unterbaum in einer Formulierung verwendet wird, muß er an anderer Stelle mit dem Schlüsselwort `SUBTREE` auch definiert werden. Will man die Attribute eines Unterbaumes ansprechen, muß man ihm ebenfalls einen Namen zuweisen. Allerdings ist der Typ der Attributinformation eines Unterbaumes nicht a priori als einfacher Integer festgelegt, sondern kann vom Entwickler der Baumgrammatik frei definiert werden. Meist wird man entsprechende Strukturen definieren, um die untergeordnete Attributinformation des Teilbaumes bereits aufbereitet nach oben weiterzureichen.

Die Schlüsselwörter `TYPE` und `UNION` dienen der Typdefinition von Unterbäumen, Beipiel 2.2 zeigt eine mögliche Variante. Wie man erkennt, besitzt das Attribut des Unterbaumes vom Typ `Operand` den Attributtyp `t_var`, der seinerseits als `struct variable *` definiert wurde. Diese in der Programmiersprache C definierte Struktur wiederum besitzt eine Komponente mit dem Namen `direction`, sodaß also im Besipiel 2.1 mittels `$AC_Op->direction` auf diese Komponente zugegriffen werden könnte.

Besitzen zwei Knoten denselben *Namen*, so wird implizit vom Generator auf Gleichheit der Attribute geprüft. Dies funktioniert auch mit ganzen Unterbäumen, im Beispiel 2.1 tritt `AC_Op` an zwei Stellen im IR-Baum auf. Beim ersten Auftreten wird die Erfüllung aller Bedingungen überprüft, während beim zweiten Auftreten auf exakte Gleichheit des gesamten Unterbaumes geprüft wird!

Mit der Verwendung von Unterbäumen findet also das Weiterreichen der Attributinformation wie folgt statt: Aufgrund der Attribute der elementaren Knoten werden

```
UNION
    int                 t_int;
    struct range *      t_ran;
    struct variable *   t_var;
END

TYPE <t_var> Operand
TYPE <t_var> Simple_Operand
TYPE <t_ran> Index
TYPE <t_ran> Range
TYPE <t_int> Pointer
```

Beispiel 2.2: Die Typdefinition der Unterbäume. Diese Definition wird analog zur Typdefinition von Grammatikbegriffen im Zusammenhang mit der Verwendung des Parsergenerators *yacc* verwendet. Details sind in [20] nachzulesen.

selbstdefinierte Datentypen aufgebaut und an die Unterbäume zugewiesen. In der übergeordneten Struktur kann dann auf diese Attribute zugegriffen werden, es werden eventuell einige Bearbeitungsschritte gemeinsam mit anderen Attributen durchgeführt und die entstehenden vorbearbeiteten Daten dann in wieder anderer Form an die nächste Struktur weitergegeben, bis die Wurzel erreicht ist. Spätestens dort, aber gegebenenfalls auch früher, muß die Prüfung der Hardware-bedingten Konflikte eingeleitet sowie die Codeerzeugung durchgeführt werden. Die Methodik der Attributsynthese ist an die des Parsergenerators *yacc* angelehnt, nähere Informationen können in [20] nachgelesen werden.

Als mögliche Variante für die Synthese der Attribute zeigt Beispiel 2.3 in Fortsetzung von Beispiel 2.1 die beiden Unterbaum-Definitionen für `Operand` und `Simple_Operand`.

Im Zuge der Baumgrammatik wird also die Struktur der erlaubten IR-Bäume exakt beschrieben, es können Bedingungen für die möglichen Attribute angegeben werden und zwar komfortabel in der textuellen Repräsentation, und es kann mit dem Schlüsselwort `SUBTREE` eine mächtige Strukturierung der Beschreibung stattfinden. Die semantischen Aktionen werden mit einer funktionalen Programmiersprache wie z.B. C oder PASCAL implementiert, mit einer Metasymbolik kann auf die Attribute zugegriffen werden. Diese Codesegmente werden später im Backend bei Auftreten einer passenden IR-Struktur aufgerufen. Dabei kann die Struktur der Attributinformation von Unterbäumen frei nach Bedürfnis festgelegt und eine effiziente Attributsynthese verwendet werden. Auf diese Weise werden auch syntaxorientierte Semantikprüfungen direkt implementiert. Mit Hilfe dieser aufbereiteten Attributinformation müssen nun die Ressourcenkonflikte untersucht werden, die bei der Vereinigung mehrerer Mikrooperationen in einem Mikrobefehl entstehen können. Im Anschluß soll auch die Codefestlegung durchgeführt werden.

```
SUBTREE Operand
:   Simple_Operand IS Operand
    { $Operand->address = make_address($Operand,0);
      $$ = $Operand; }
|   [Builtin] VALUE "HSTR"
        [List]
            Simple_Operand IS Operand
            [Constant] IS digits VALUE "1","2","3","4","5","6","7","8" TO (1,1)
    { $Operand->bits->length = $digits * 4; /* explizite Verarbeitungslaenge */
      $Operand->address = make_address($Operand,1);
      $$ = $Operand; }
|   [Builtin] VALUE "BSTR"
        [List]
            Simple_Operand IS Operand
            [Constant] IS bytes VALUE "1","2","3","4" TO (1,1)
    { $Operand->bits->length = $digits * 8; /* explizite Verarbeitungslaenge */
      $Operand->address = make_address($Operand,0);
      $$ = $Operand; }
|   [Builtin] VALUE "BSTR"
        [List]
            Simple_Operand IS Operand
            [Builtin] IS direction VALUE "L","R" TO 1,0
    { $Operand->direction = $direction;
      $Operand->address = make_address($Operand,3);
      $$ = $Operand; }
END /* SUBTREE Operand */

SUBTREE Simple_Operand
:   [Variable] IS reg
    { $$ = make_operand($reg,implicit_index($reg),implicit_bits($reg),0); }
|   [Variable] IS reg
        LEFT Index IS field
    { $$ = make_operand($reg,$field,implicit_bits($reg),0); }
|   [Variable] IS reg
        LEFT Index IS field
        RIGHT [Area]
            [Constant] IS offset
            [Constant] IS length
    { $$ = make_operand($reg,$field,make_range(0,0,0,$offset,$length,0)); }
|   [Variable] IS reg
        RIGHT Range IS bits
    { $$ = make_operand($reg,implicit_index($reg),$bits); }
END /* SUBTREE Simple_Operand */
```

Beispiel 2.3: Die Typdefinition der Unterbäume `Operand` und `Simple_Operand`. Man erkennt, daß die mittels $-Metasymbolik verwendeten Attributnamen im C-Programmcode wie normale C-Variablen verwendet werden können. $$ steht für den Grammatikbegriff, der auf der linken Seite der aktuellen Regel steht, damit wird also die Synthese der Attribute, also die Weitergabe an die übergeordneten Strukturen, eingeleitet. Die in den semantischen Aktionen verwendeten Unterprogramme und C-Definitionen sind in Anhang B.2 ab Seite 73 angeführt.

Konfliktprüfung und MDL/C-Schnittstelle

In Abschnitt 1.4 ab Seite 20 wurden die möglichen Ressourcenkonflikte bereits nach Konflikten erster, zweiter und dritter Art klassifiziert. Anhand dieser Einteilung ist es offensichtlich, daß die Konflikte dritter Art vom Compiler nicht überprüft werden können. Zur Überprüfung der Konflikte erster Art hingegen gibt es bereits die Top-Down-Konfliktüberprüfung, die in [4] in Abschnitt 3.2 ab Seite 77 beschrieben ist. Der Grundgedanke für die Überprüfung der Konflikte zweiter Art durch den Firmwarecompiler liegt nun in der Anpassung der Top-Down-Konfliktanalyse dahingehend, daß die *Ressourcenvariablen* der Mikrooperationen mit konkreten Ressourcen belegt werden, die sich aus den Attributen des IR-Baumes ergeben.

In Beispiel 1.1 ab Seite 12 ist ein Ausschnitt aus der MDL/HD-Beschreibung eines Prozessors dargestellt. Die Mikrooperation `addition` beschreibt dabei die Hardware-Abläufe der arithmetisch-logischen Funktionen, Details sind in [4] nachzulesen. Wesentlich ist die Definition der Ressourcenvariablen `p1`, `p2`, `reg1` und `reg2` mit dem Schlüsselwort `VAR`. Diese Ressourcenvariablen stehen jede für genau eine Ressource aus der *Trägermenge* der möglichen Ressourcen, deren Deklaration mit dem Schlüsselwort `PARTOF` eingeleitet wird. Hinter dem Schlüsselwort `USING` wird dann angegeben, welche Ressource oder Ressourcenvariable von der Mikrooperation in welchen Zeitbereichen benötigt wird. Daher muß umgekehrt für jede syntaxgesteuerte Ressourcennutzung eine Ressourcenvariable definiert werden. Nicht durch die Syntax beeinflußbare Ressourcenvariablen bedeuten, daß erst zur Laufzeit festgelegt wird, welche Ressource konkret benutzt wird. Das kann zum Beispiel bei der indirekten Adressierung passieren und stellt die Basis für Konflikte dritter Art dar.

Da in der MDL/HD keine konkreten Syntax/Code-Belange enthalten sind, handelt es sich bei den Mikrooperationen genau genommen um *Klassen* von Hardwareoperationen, die eine ähnliche Ressourcennutzung, aber unterschiedliche Syntax/Code-Zuordnung besitzen. Dadurch wird die MDL/HD-Beschreibung deutlich entlastet, sie kann vom Hardwareentwickler benutzt werden, um die Funktionalität der Hardware in Hinblick auf die Ressourcenkonflikte zu beurteilen. Vom Firmwareentwickler wird sie dann erst um die Baumgrammatik MDL/BED ergänzt, die der Generierung des Firmwarecompilers dient.

Beispiel 2.4 zeigt eine Möglichkeit für die Eingabe der modifizierten Top-Down-Konfliktanalyse. Es werden konkrete Ressourcen in MDL-Syntax an die Variablen zugewiesen, die Reihenfolge der Zuweisung entspricht der Reihenfolge bei der Definition der Ressourcenvariablen. Mit dem Schlüsselwort `NIL` wird bestimmt, daß die Ressourcenvariable gar nicht benutzt wird, mit dem Schlüsselwort `UNDEF`, daß sie unbestimmbar bleibt, wie im Beispiel der indirekten Adressierung. Im Zuge der Abarbeitung dieser Konfliktprüfung durch den Analysator sind für jede Mikrooperation drei Aufgaben durchzuführen:

1. Es muß überprüft werden, ob die an die Variablen zugewiesenen Ressourcen in der mit `PARTOF` definierten Trägermenge enthalten sind.

2. Es müssen die mit dem Schlüsselwort `CONDITION` definierten Bedingungen über den konkreten Inhalten der Ressourcenvariablen ausgewertet werden. Komplexere Bedingungen, die nicht einfach und *syntaxunabhängig* mit dem Schlüsselwort

```
COMPOSE
MICROP addition VAR p5, NIL, UNDEF, fr[4];
MICROP branch VAR ....;
MICROP ....
END
```

Beispiel 2.4: Konfliktprüfung mittels Top-Down-Analyse. Als erste Mikrooperation wurde `addition` verwendet, vergleiche mit Beispiel 1.1 ab Seite 12. Das zugehörige Sprachelement des Firmwareprogrammes lautet `FR(P5) = XOR(FR(P5),FR(4))`. Daher wird das Register `fr[4]` an Stelle der Ressourcenvariable `reg2` sicher benötigt, die Ressourcenvariable `p2` hingegen überhaupt nicht und daher mit `NIL` befüllt. Aufgrund der indirekten Adressierung wird der Ressourcenvariablen `p1` das Pointerregister `P5` zugewiesen, während die Ressourcenvariable `reg1` unbestimmt bleiben muß (`UNDEF`), weil es sich um eine indirekte Adressierung handelt und das konkrete Register daher erst zur Laufzeit feststeht.

`CONDITION` beschrieben werden können, müssen funktional im Zuge der semantischen Aktionen behandelt werden.

3. Es müssen die Ressourcenvariablen mit den konkreten Ressourcen befüllt werden. Damit werden dann die mittels `USING` definierten Ressourcennutzungen in einer Tabelle eingetragen.

Die Eingabe für die Top-Down-Analyse wird mit dem Schlüsselwort `END` abgeschlossen. Im Anschluß wird die Zusammensetzung der Mikrooperationen zu einem Mikrobefehl abgehandelt und die Konfliktanalyse durchgeführt. Danach erst wird die Codierung des Mikrobefehls in das relozierbare Modul eingetragen, welches die Ausgabe des Firmwarecompilers darstellt. Details dazu werden im nächsten Abschnitt gebracht.

Der generierte Firmwarecompiler kann nun ebenfalls die Top-Down-Konfliktanalyse des Analysators benutzen. Diese Verbindung wird als MDL/*C-Schnittstelle* bezeichnet. Bei einer UNIX-Implementierung kann die Schnittstelle leicht durch eine `pipe` realisiert werden, Beispiel B.2 ab Seite 74 enthält in der Prozedur `backend()` den erforderlichen Systemaufruf. Zu diesem Zweck muß also im Zuge der semantischen Aktionen eine Zeichenkette zusammengebaut werden, die der Syntax der Top-Down-Analyse entspricht. Die Schnittstelle wird dann mit dieser Zeichenkette beschickt. Der Analysator „merkt“ dabei gar keinen Unterschied, er führt eine normale Top-Down-Konfliktüberprüfung durch, wie sie eben besprochen wurde. Ebenfalls in Beispiel B.2 ist in der Prozedur `alu_conflict` die Zusammensetzung einer solchen Zeichenkette und die Verwendung der Schnittstelle angedeutet.

Auf diese Weise wird also die Abbildung der Attribute der IR auf die Ressourcenvariablen bewerkstelligt: Aus den Attributen werden die Parameter der Konflikt-Abfrage über die MDL/C-Schnittstelle ermittelt und diese dann als Zeichenkette an den Analysator weitergegeben. Das Ergebnis der im Analysator stattfindenden Top-Down-Konfliktanalyse wird dann vom Analysator in gewünschter Weise zurückgeliefert oder

gegebenenfalls als Fehlermeldung ausgegeben. Diese Vorgangsweise implementiert eine Mini-Client-Server-Struktur: Der Analysator fungiert als MDL-Server, der generierte Firmwarecompiler bedient sich als Client der Möglichkeiten zur Konfliktabfrage und Codierung.

Zur konkreten Implementierung der MDL/C-Schnittstelle gibt es folgende Variante: Statt eine Zeichenkette zusammenzubauen, könnte man auch die Top-Down-Befehle direkt in den C-Code der semantischen Aktionen einfügen, durch ein spezielles Symbol vom eigentlichen C-Code getrennt. Innerhalb des Top-Down-Befehls benötigt man dann eine weitere Metasymbolik, um C-Variablen als Konstante innerhalb des Top-Down-Befehls nutzen zu können. Die Vorgangsweise ist dabei ähnlich zur SQL[8]/C-Schnittstelle, wie sie z.B. in [2] beschrieben ist. Ein Präcompiler muß dann aus den Top-Down-Befehlen die entsprechenden Aufrufe der MDL/C-Schnittstelle über Zeichenketten generieren.

Als Alternative zur MDL/C-Schnittstelle überhaupt erzeugt der Analysator Bottom-Up-Konflikttabellen. Diese enthalten die Information, welche Mikrooperationen auf den Ressourcen welche Konflikte verursachen. Der generierte Firmwarecompiler könnte nun die Tabellen konsultieren, um eine konkrete Kombination von Mikrooperationen auf Konflikte zu prüfen, was vordergründig ein günstigeres Laufzeitverhalten zu versprechen scheint. Dieser Ansatz hat jedoch den entscheidenden Nachteil, daß das Einbringen der Ressourcenvariablen die Tabellen und die Funktionen zur Tabellenverwaltung sehr rasch so komplex werden läßt, daß der scheinbar gewonnene Vorteil beim Laufzeitverhalten in der Praxis mehr als zunichte gemacht wird.

Codeerzeugung und relozierbare Module

Aufgabe der Codeerzeugung ist es, bestimmten Feldbereichen im Mikrobefehlsfeld die Bitmuster zuzuordnen. Dabei müssen drei Einschränkungen überprüft werden:

1. Es dürfen nur solchen Feldern Bitcodes zugeordnet werden, die innerhalb der Definition der Mikrooperation mit dem Schlüsselwort `FIELD` deklariert wurden. Damit kann es *innerhalb* einer Mikrooperation zu keinen Feldüberlappungen kommen.

2. Es dürfen die Felder *mehrerer* Mikrooperationen einander nicht überlappen, wenn diese Mikrooperationen in einem Mikrobefehl vereint werden sollen.

3. Die Bitcodes müssen der Größe nach in das jeweilige Feld „hineinpassen“.

Die Berechnung der Bitcodes erfolgt im Rahmen der semantischen Aktionen. Eine besondere Erleichterung bietet dabei das Schlüsselwort `TO` innerhalb der Baumgrammatik: Mit ihm können sehr effizient bereits die Bitcodes die identifizierten Attribute aus einer vorgegebenen Attributmenge ersetzen, sodaß diese veränderten Attribute dann später nur noch 1:1 an die Codierungs-Funktion übergeben werden müssen. Beispiel 2.1 auf Seite 37 zeigt die Anwendung dieser Vorgangsweise.

[8] Structured Query Language.

```
COMPOSE
MICROP addition VAR p5, NIL, UNDEF, fr[4];
CODE fgroup TO 0b00;
CODE ffunction TO 0b10011;
CODE fac.art TO 2;
CODE fac.bl TO 3;
CODE fac.r TO 2;
CODE fac.wi TO 5;
CODE fac.x TO 0;
CODE fac.b TO 0;
CODE fb.art TO 1;
CODE fb.bl TO 3;
CODE fb.r TO 2;
CODE fb.w TO 4;
CODE fb.b TO 0;
MICROP branch VAR ....;
CODE fdest TO label<3:12>;
CODE fcond TO 0b011,0x6E;
MICROP ....
END
```

Beispiel 2.5: Codierung über die Top-Down-Analyse. In Beispiel 1.1 ab Seite 12 sind die zugehörigen Felddefinitionen enthalten. Das entsprechende Sprachelement des Firmwareprogrammes lautet `FR(P5) = XOR(FR(P5),FR(4))`. Die Entstehung der Codierung kann durch Vergleich mit den Beispielen B.1 ab Seite 67 und B.2 ab Seite 74 nachvollzogen werden. Interessant ist auch die Zuweisung des Labels `label` an das Feld `fdest`: Die Bits 3 bis 12 der später dem Label zugeordneten Adresse sollen dem Feld `fcond` zugewiesen werden. Erst der Linker kann später die konkrete Adresse eruieren und den gewünschten Ausschnitt als Bitmuster dem vordefinierten Feldbereich zuordnen. Die Codierung des Feldes `fcond` zeigt eine weitere Besonderheit: Die beiden Zahlen werden als Binärzahlen aneinandergehängt und ergeben so insgesamt den Bitcode `0b01101101110`. Damit können auch Bitcodes erzeugt werden, die den Zahlenbereich der Integer-Variablen der in den semantischen Aktionen verwendeten Programmiersprache übersteigen.

Um auch bei der Codierung die in der MDL verwendete komfortable Syntax verwenden zu können, wird die Top-Down-Konfliktprüfung des Analysators noch um Codierungs-Information erweitert, Beispiel 2.5 zeigt in Erweiterung des Beispiels 2.4 eine mögliche Anwendung. Dabei wird schlicht an bestimmte vordefinierte Feldbereiche der entsprechende Bitcode zugeordnet, bei längeren Codes könnnen durch Beistrich voneinander getrennte Zahlen an ein Feld zugewiesen werden, indem sie als Binärzahlen aneinandergehängt werden. Eine Ausnahme bilden die Sprungadressen, die zu diesem Zeitpunkt nicht aufgelöst werden können, da ja ein relozierbares Modul erzeugt wird. Felder, die mit Adressen codiert werden, erhalten daher anstelle des Bitcodes einen Label und Teilbereich eines Labels. Gemeinsam mit der Label-Tabelle kann dann später der Linker die endgültigen Bitmuster in die richtigen Feldbereiche einordnen. Alle `CODE`-Befehle sind der unmittelbar vorangegangenen Mikrooperation zuzuordnen, sodaß leicht überprüft werden

kann, ob nur vordefinierte Felder dieser Mikrooperation benutzt wurden.

Im Zuge der semantischen Aktionen kann also die Codierung ebenso über die MDL/C-Schnittstelle erfolgen, wie die Prüfung der Ressourcenkonflikte: Es wird eine geeignete Zeichenkette aufgebaut und an den Analysator übergeben, der im Zuge der Top-Down-Analyse nun auch die Codierung durchführt. Im Beispiel B.2 ab Seite 74 wurde jedoch die Zeichenkette nicht direkt aufgebaut, sondern eine Funktion `encode(char * string, int bitcode)` verwendet, die dann die Ausgabe des Abfragestrings mittels `fprintf(query,"CODE %s TO %d;\n", string, bitcode)` erzeugt.

Das Endergebnis der Codeerzeugung ist zugleich die Ausgabe des Firmwarecompilers insgesamt: Ein relozierbares Modul und eine Label-Tabelle. Das relozierbare Modul enthält für jeden Mikrobefehl eine Zeilennummer als Offset innerhalb des Modules und dann eine Reihe von Bitbereichen mit den zugehörigen Bitcodes oder den Labelnamen. Die Labeltabelle enthält zu allen Labeln, die im Modul definiert wurden, den Offset und zu allen anderen symbolischen Adressen den Vermerk „extern“. Der Linker verbindet dann mehrere Module, löst alle Referenzen auf externe Label auf und erzeugt ein Gesamtmodul, das immer noch relozierbar ist. Erst der Locater bestimmt aus der Startadresse und dem Offset die physikalische Adresse der Mikrobefehle und kann nun schließlich auch die Label durch die entsprechenden Bitmuster ersetzen und an die vorgegebenen Bitbereiche im Befehlsfeld zuweisen. Dabei muß natürlich überprüft werden, ob die Bitmuster in die Felder „hineinpassen“. Das fertige Firmwareprogramm kann nun in ein EPROM geladen und zum Testen der Hardware verwendet werden.

2.4 Gesamtstruktur und Implementierung

Implementierung

Die Implementierung dient als Machbarkeitsstudie der theoretischen Ansätze. Daher wurde ein Prototyp implementiert, es wurden aber nicht alle Teile fertig ausgeführt, da dies den Aufwand einer Diplomarbeit bei weitem überstiegen hätte. Daher sollen an dieser Stelle auch nur die Implementierungs-*Ansätze* der komplexeren und zentralen Fähigkeiten des Generators angedeutet werden, die Fortsetzung dieser Gedanken zum lauffähigen Generator ist dann reine Programmierarbeit[9].

Die Implementierung des Backend-Parser-Generators für die Baumgrammatik erfolgt so: Aus der Baumgrammatik wird vom Generator eine linearisierte Form erzeugt, die als Eingabe für den Parsergenerator *yacc* [20] dienen kann. Diese Form wird in der Datei `backend.y` abgelegt. Die zugehörige lexikalische Beschreibung der IR wird fix in der Datei `backend.l` festgelegt. Zusätzlich gibt es die Datei `backend.c`, in der die Unterprogramme zu den semantischen Aktionen zusammengefaßt sind. In einem weiteren Schritt bildet der Parsergenerator dann einen tabellengetriebenen Parser für die IR.

Bei der Erzeugung von `backend.y` werden auch die `$name`-Referenzen der Attribute zu entsprechenden `$i`-Anweisungen umgeformt, wie sie von *yacc* akzeptiert werden. Binärzahlen in der Form `$0b010011` werden in Hexadezimalzahlen umgeformt. Ansonsten werden die semantischen Aktionen unverändert übernommen, da sie bereits in einer *yacc*-verträglichen Form abgelegt sind. Die hinter den Schlüsselworten `UNION` und `TYPE` angeführte Information wird ebenfalls in sehr geradliniger Weise für *yacc* aufbereitet. Die mit `VALUE` definierten erlaubten Attributmengen werden durch Ketten von `if-else`-Abfragen ersetzt. Die mit dem Schlüsselwort `TO` definierten neuen Attributwerte ersetzen die originalen im Zuge dieser Abfragen.

Überprüfung der Entwurfsziele

Bevor die Erreichung der Entwurfsziele durch Vergleich mit den in Abschnitt 1.4 gestellten Anforderungen überprüft wird, erfolgt an dieser Stelle noch eine kurze Zusammenfassung des Gesamtsystems. Abbildung 2.7 zeigt die Gesamtübersicht. Der Firmwarecompiler besteht aus den beiden Komponenten Frontend und Backend. Das Frontend akzeptiert ein Mikroprogramm und formt es in eine Intermediate Representation (IR) um. Um die hardwarespezifischen Deskriptoren und Aliasnamen zu erkennen und umzusetzen, benötigt das Frontend Tabellen, die vom Generator erzeugt werden müssen. Der Generator erhält zu diesem Zweck eine MDL-Beschreibung der Hardware und generiert neben den Tabellen auch das Backend des Firmwarecompilers. Das Backend wiederum erhält die IR als Eingabe und produziert ein relozierbares Modul als Ausgabe. Um Konflikte zu erkennen und die Codierung durchzuführen, bedient sich das Backend über die

[9] Diese ist zwar nicht zwingend trivial, hat aber mit der eigentlichen Aufgabenstellung der Diplomarbeit weniger zu tun: Die Probleme liegen dann eher im Auffinden geeigneter Algorithmen und Datenstrukturen.

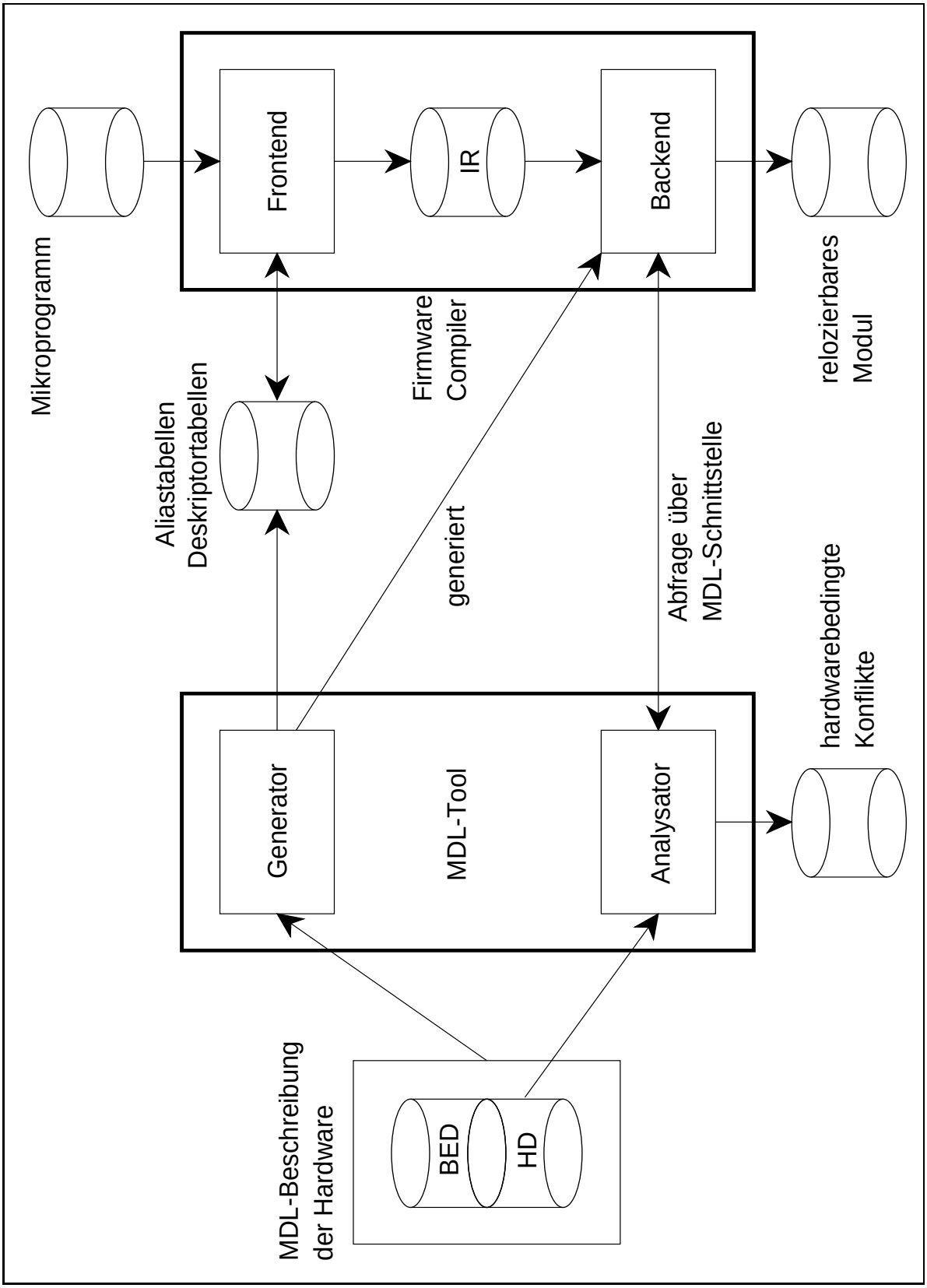

Abbildung 2.7: Der Gesamtaufbau des generierten Firmwarecompilers.

MDL-Schnittstelle der Top-Down-Konfliktanalyse des Analysators. Dieser benötigt nur eine MDL/HD-Beschreibung und erzeugt als Ausgabe der Top-Down-Analyse Meldungen über hardwarebedingte Konflikte. Gesteuert wird die Generierung des Firmwarecompilers von der MDL/BED-Beschreibung, die aufbauend auf die Baumgrammatik der IR die Zuordnung der IR-Attribute an die Ressourcenvariablen und an die Codeerzeugung durchführt. Dieses System ermöglicht eine effiziente Implementierung der komplexen Relationen zwischen Hardwareressourcen, Syntax der Mikroprogrammsprache und Bitcode. Die besonders häufigen 1:1-Beziehungen (80%) werden dabei besonders einfach implementiert, komplexere Beziehungen werden im Zuge der semantischen Aktionen funktional beschrieben.

Abschließend werden noch die Anforderungen an die MDL/BED-Spracherweiterung sowie an den Generator überprüft:

- Effiziente Implementierung der semantischen Prüfungen sowie der Aufbereitung der Attribute für Konfliktprüfung und Codeerzeugung: Das ergibt sich durch die Verwendung der Programmiersprache C zur Beschreibung der semantischen Aktionen sowie durch die Erweiterung der Top-Down-Analyse.
- Automatische Generierung der hardwarebedingten semantischen Regeln, d.h. Generierung der Konfliktprüfung von Ressourcenkonflikten: Dieser Punkt wird durch die Benützung der Top-Down-Konfliktprüfung des Analysators via MDL/C-Schnittstelle erfüllt.
- Die Codeerzeugung erfolgt ebenfalls im Rahmen der semantischen Aktionen über die MDL/C-Schnittstelle.
- Auswahl der prozessorspezifischen Sprachuntermenge aus dem fest vorgegebenen Sprachrahmen: Die eingeschränkte Sprachuntermenge ergibt sich implizit aus der Gesamtmenge aller zugelassenen IR-Strukturen. Für jede Mikrooperation wird also implizit überprüft, ob sie in der prozessorspezifischen Sprachuntermenge des Sprachrahmens enthalten ist.

Um das Backend generieren zu können, ermöglicht daher die Spracherweiterung MDL/BED die Beschreibung folgender Sachverhalte:

1. Die Formulierung der semantischen Aktionen erfolgt in der Programmiersprache C, für die Verwendung der Attribute der Mikroprogrammsprache wird eine Metasymbolik mittels $-Referenzen eingeführt.
2. Die Angabe der erlaubten beziehungsweise codierbaren *Syntaxelemente* der Mikroprogrammsprache sowie die Zuordnung derselben zu den Mikrooperationen wird durch die Verwendung eines binären Baumes als Intermediate Representation und die Referenz dieser IR mittels Baumgrammatik erreicht.
3. Die Beschreibung der Konfliktprüfung erfolgt direkt im Zuge der semantischen Aktionen unter Verwendung der MDL/C-Schnittstelle. Dabei werden die IR-Attribute an die Ressourcenvariablen zugewiesen, um die Konflikte zweiter Art überprüfen zu können.

4. Aus den Attributen kann der Bitcode der einzelnen Mikrooperationen algorithmisch ermittelt werden. Er wird dann den entsprechenden Bitbereichen im Mikrobefehlsfeld zugeordnet, woraus sich insgesamt der Bitcode des Mikrobefehls ergibt.

Um die Vorteile der MDL–Beschreibung voll nutzen zu können, ist es darüber hinaus möglich, innerhalb der Programmcodesegmente auf die `FIELD`-, `REGISTER`-, `MAP`- und `ALIAS`–Definitionen mittels einer Metasymbolik zuzugreifen. Darüberhinaus ist die Verwendung des `multirange` sowie auch der `Binary_constant` möglich.

Kapitel 3

Weiterführende Konzepte und Zukunft

Um die Methoden, die bei der Generierung des Firmwarecompilers verwendet wurden, auch für andere Bereiche zu erschließen, wird zunächst eine allgemeinere Sicht der Problematik dargestellt.

3.1 Der allgemeine Entwicklungsprozeß

Grundsätzlich ergibt sich bei der gemeinsamen Erstellung von Hardware und Software durch den Entwickler eine gegenseitige Abhängigkeit, wie sie in Abbildung 3.1 dargestellt wird. Je nachdem, ob der spätere Anwender eher mit der Hardware oder mit der Software interagiert, je nachdem also, ob die Hardware oder die Software im Vordergrund steht, kommt es zu einer unterschiedlichen Ausprägung der Hardware/Software-Schnittstelle.

Wenn das Gesamtsystem mit dem späteren Anwender über die Benutzerschnittstelle der Software in Kontakt tritt, dann zeigt Abbildung 3.2 die Interaktionen. Typisch für diese Konfiguration sind Information-Engineering-Prozesse, wo also anwenderorientiert programmiert wird, im Vordergrund steht die algorithmische Formulierung der Aufgabe. In einem praxisfremd idealisierten Top-Down-Entwicklungsprozeß würde also die Hardware nach den Erfordernissen der jeweiligen Software generiert werden. Wie bei allen praktischen Entwicklungsprozessen kommt jedoch eine Bottom-Up-Komponente ins Spiel: Es wird standardisierte Hardware als Basis für die Software verwendet, etwa Personal Computer oder Workstations. Die Bottom-Up-Methodik geht sogar noch weiter, sodaß auch Standard-Softwaremodule eingebunden werden. In der effektiven Verschmelzung von Bottom-Up und Top-Down, in der gegenseitigen Beeinflussung von Hardware- und Software-Design befindet sich das *semantic gap*, der semantische Bruch zwischen Hardware und Software. Wie die Methoden der MDL zum Nutzen des Hardware/Software-Codesign verwendet werden können, wird im Abschnitt 3.2 erläutert.

Eine gänzlich andere Situation liegt vor, wenn die Funktionalität der Hardware im Vordergrund steht. Dies ist zum Beispiel beim Entwurf von Mikrocontrollern zur Steue-

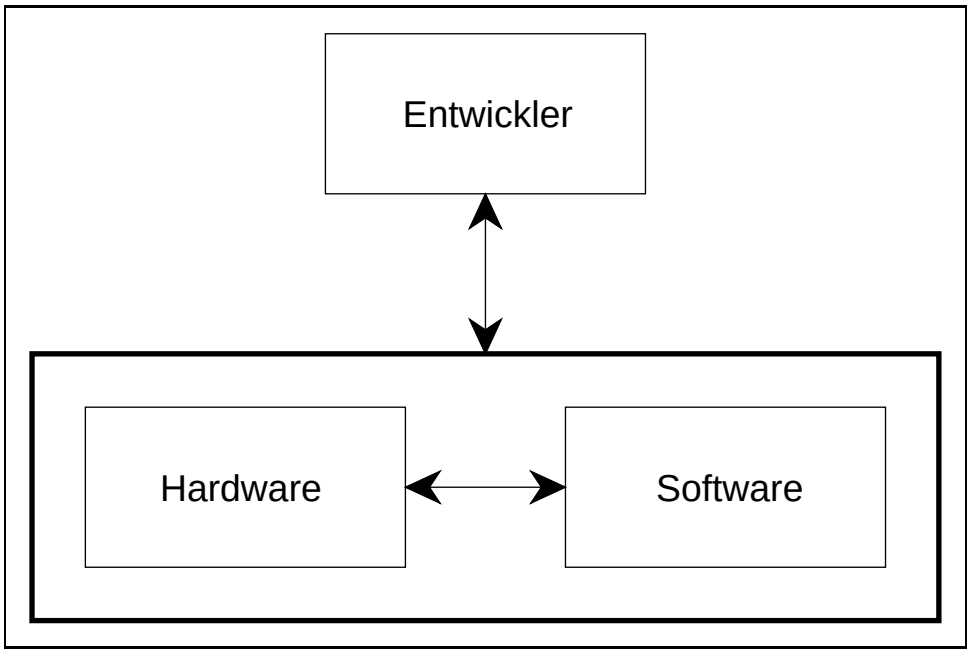

Abbildung 3.1: Der Hardware/Software-Entwicklungsprozeß.

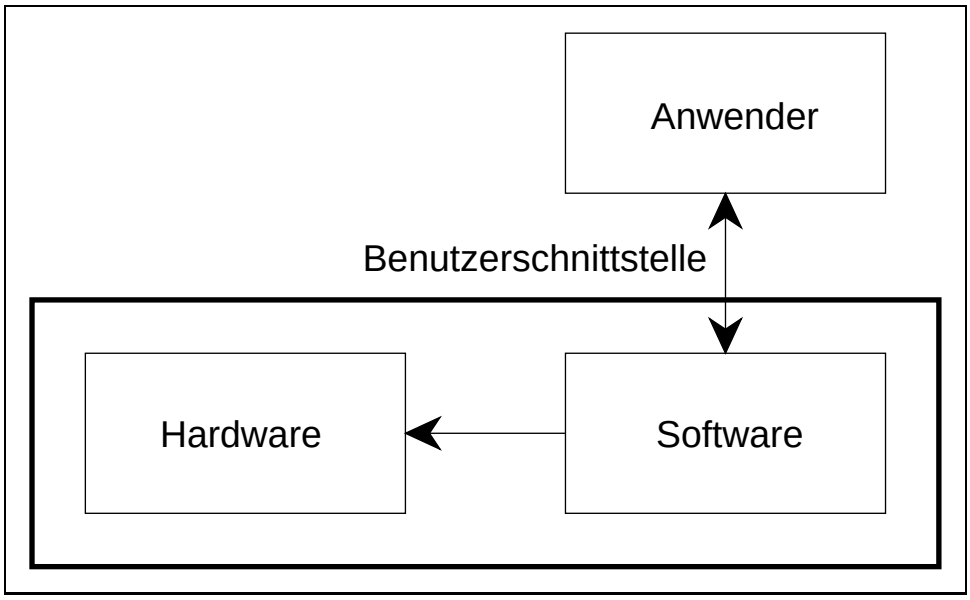

Abbildung 3.2: Die Software steht im Vordergrund.

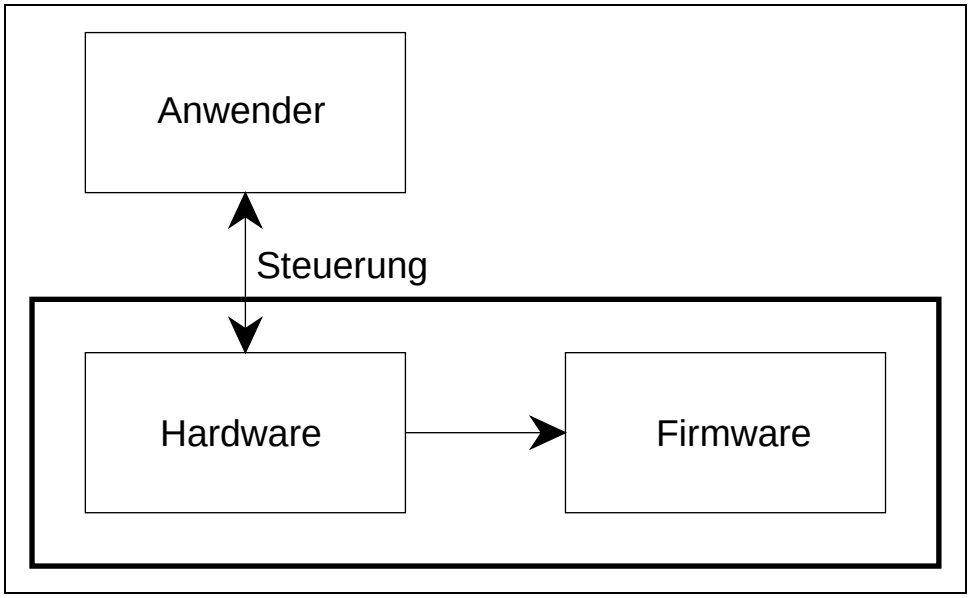

Abbildung 3.3: Die Hardware steht im Vordergrund.

rung der Fall, aber auch ganz allgemein beim Design mikroprogrammierter Hardware, also auch beim Design eines Prozessors, der dann als Bottom-Up-Komponente in einem Softwareentwicklungsprozeß dient. Abbildung 3.3 zeigt symbolisch die Interaktionen. Das Maß aller Dinge ist jetzt die Funktionalität der Hardware, die Software, die jetzt als Firmware bezeichnet wird, dient lediglich der Programmierung dieser Funktionalität. Betrachtet man wieder einen idealisierten Top-Down-Entwicklungszyklus, so bedingt die Spezifikation der Hardware in weiterer Folge die Generierung der Firmware. Während also im Zuge der Entwicklung von MDL das Ziel die Generierung des Firmwarecompilers war, verfolgt die High-Level-Synthese sogar den Ansatz, die Firmware selbst zu generieren. Daß dabei aber auch Methoden zum Zug kommen müssen, die denen der MDL ähnlich sind, wird im Abschnitt 3.3 angedeutet.

3.2 Hardware-Software Codesign

Erste Ideen zur Verwendung der MDL-Methodik im Bereich des Hardware/Software-Codesign wurden bereits in [4] vorgestellt, daher an dieser Stelle eine Zusammenfassung und Erweiterung dieser Ansätze. Es wird die Hardware eines Prozessors so beschrieben, daß daraus die hardwareabhängigen Teile des Compilerbackends für den Hochsprachencompiler generiert werden können. Zwei Vorteile ergeben sich unmittelbar aus einer solchen Methode:

1. Fertige, hochentwickelte Compiler wären mit geringstem Aufwand auf jeder beliebigen Maschine verfügbar. Vor allem jede Verbesserung am Compiler und insbesondere jede hardwareunabhängige Optimierungsmethode könnte sofort auf allen Maschinen eingesetzt werden. Da heutzutage nicht nur Anwenderprogramme, sondern auch Betriebssysteme nahezu ausschließlich in Hochsprachen programmiert werden, reduziert sich das Problem der *Portabilität* auf die Suche nach einer geeigneten Hardwarebeschreibungsmethode.

2. Mit der Möglichkeit, den Compiler auch für eine abstrakte Maschine zu entwickeln, könnte der Compiler bereits in der Entwurfsphase eines neuen Prozessors zur Verfügung stehen. Damit könnte bereits der entstehende Prozessor mit fertigen Programmen getestet werden, der Compiler wird somit zu einem *Entwurfswerkzeug* für den Hardwareentwickler. Das Zitat "A computer architect designes machines to run programs" aus [14] zeigt die Bedeutung dieses Entwurfswerkzeuges als Maßstab für die Prozessorentwicklung: Es kann das Hauptziel des Entwurfsprozesses in jeder Phase der Entwicklung im Auge behalten und überprüft werden, indem stets eine Auswahl typischer Anwenderprogramme mit dem Compiler für die in Entwicklung befindliche Maschine übersetzt und getestet wird. Damit können letztendlich auch Kosten erspart werden, wenn Fehlentwicklungen frühzeitig erkannt und verhindert werden.

Vor allem auf den zweiten Vorteil wird noch etwas näher eingegangen.

Voraussetzung für beide Vorteile ist die gestiegene Bedeutung des *Compilers*: Da fast nur noch in Hochsprachen programmiert wird, ist es vor allem die Ausgabe des Compilers, die von den Prozessoren abgearbeitet wird. Daher stehen Hardwareentwicklung und Compilertechnologie in einem engen Zusammenhang, ja sind sogar gegenseitig voneinander abhängig und die Unterstützung des Compilers wird zu einem wesentlichen Designziel der Hardware. Eine Methode analog zur MDL könnte hier die Brücke zwischen Hardware und Compiler schlagen. Diese Methode wurde in [4] als CIAD – *Compiler Integrated Architecture Design* bezeichnet, die Beschreibungsform, die dieser Methode zugrundeliegt, als HCDL – *High Level Compiler Design Language*. Zunächst wird die Methode des CIAD näher beschrieben, danach werden Anforderungen an die HCDL im Vergleich zur MDL analysiert sowie die Machbarkeit beurteilt.

Um den Entwurf der Hardware während der gesamten Entwicklung zu unterstützen, muß das CIAD hierarchisch gegliedert sein und die HCDL die Beschreibung der Hardware auf mehreren *Abstraktionsebenen* ermöglichen:

Architektur: Auf dieser Abstraktionsebene muß die HCDL die Architektur des Befehlssatzes des Prozessors beschreiben. Daraus muß der Compiler für eine *abstrakte Maschine* generiert werden. Dieser ermöglicht dann das Testen des Befehlssatzes, zum Beispiel die Ausnutzung der einzelnen Befehle durch den entstehenden Code, insbesondere die dynamische Häufigkeit des Auftretens einzelner Befehle. Bisher wurden kurze, künstlich konstruierte Programme händisch übersetzt, um diese Messungen durchzuführen. Der Nachteil war, daß kein noch so subtil konstruiertes Beispiel eine Auswahl realer Anwenderprogramme ersetzen kann, ja manchmal waren gerade diese konstruierten Beipiele für eine Irreführung der Entwickler verantwortlich.

Organisation: Hier muß die funktionelle Organisation der Hardware auf *Register–Transfer–Ebene* beschrieben werden. Es wird die Hardware besonders im Hinblick auf Ressourcenauslastung und Ressourcenkonflikte untersucht. Im Vordergrund steht hier die Messung der erreichbaren Geschwindigkeit, allerdings noch auf Basis des Taktzyklus als abstrakte Größe. Dabei muß berücksichtigt werden, daß ein Design mit niedriger Taktzyklenzahl im Zuge der Implementierung eine Verlängerung der Taktzyklendauer zur Folge haben kann, die den erreichten Vorteil unter Umständen mehr als zunichte macht.

Implementierung: Hier erfolgt eine Beschreibung auf *Gatterebene*, das logische Design, das Packaging und die Entflechtung stehen jetzt im Vordergrund. Jetzt kann auch die *reale* Geschwindigkeit der getesteten Programme als eigentliches Designziel überprüft werden. Ist die Entwicklung des Prozessors abgeschlossen, steht auch sofort ein ausgereifter Compiler zur Verfügung.

Dabei sind die einzelnen Hierarchieebenen nicht isoliert zu sehen. Vielmehr sollte eine geschlossene Beschreibung die Hardware auf allen drei Hierarchieebenen modellieren. Dies ist vor allem dann sinnvoll, wenn sich während der Entwicklung verschiedene Teile des Prozessors in unterschiedlichen Entwurfsstadien befinden. Es soll jedenfalls stets möglich sein, den Einfluß des Compilers zu testen und quantitative Untersuchungen auf Basis der Anwenderprogramme durchzuführen.

Zuletzt noch ein Vergleich zwischen MDL und HCDL, der die Unterschiede und Gemeinsamkeiten untersucht und schließlich die Machbarkeit der CIAD darstellt. Speziell für RISC[1]–Prozessoren gilt: Noch nie gab es eine Klasse von Mikroprozessoren, die untereinander so *ähnlich* gewesen sind. Dies läßt zugleich erwarten, daß *eine* Beschreibungsform gefunden werden kann, mit der alle RISC–Prozessoren beschrieben werden können. Eine weitere Ähnlichkeit zur Mikroprogrammierung ist die oft *fixe* Aufteilung des Befehlsfeldes mit typisch etwa einem bis fünf verschiedenen Befehlsformaten. Dabei werden oft einzelne Bitfelder direkt als Steuerleitungen in die Hardware übernommen. Ein typisches Beispiel ist die Registernummer, die bei RISC–Prozessoren häufig einen fixen Platz im Befehlsfeld besitzt und von dort über einen Decoder direkt der Registerselektion dient.

[1]Reduced Instruction Set Computer.

Eine weitere Analogie findet man vor allem bei RISC–Prozessoren mit den Eigenschaften *superpipelined*[2] oder *superscalar*[3]: Durch parallel ausgeführte Befehle kann es zu Ressourcenkonflikten kommen. Dies verursacht zwar keine Fehler, aber immerhin bewirkt es, daß Befehle auf ihre Ausführung warten müssen, was in einer Verminderung der Gesamtgeschwindigkeit resultiert. Hier ist es eine der wichtigsten Aufgaben des Compilers, die Befehle so anzuordnen, daß solche Verzögerungen vermieden werden. An die Stelle der Konflikte zwischen Mikrooperationen treten also hier die Konflikte zwischen Maschinenbefehlen.

Diese Gemeinsamkeiten lassen von der Machbarkeit der MDL auf eine Machbarkeit der CIAD schließen. Dennoch wird diese Methode deutlich komplexer sein als die MDL, was auf folgende Unterschiede zwischen Mikroprogrammierung und Hochsprache zurückzuführen ist:

- Wenn ein Konflikt erkannt wird, bedeutet das bei der Mikroprogrammierung, daß die beteiligten Mikrooperationen nicht in einem Mikrobefehl vereinbar sind. Bei der Makroprogrammierung hingegen muß nach geeigneten Algorithmen eine *Umordnung* der Befehle gefunden werden, sodaß alle Konflikte beseitigt sind. Anders als bei der Mikroprogrammierung ist es dem Makroprogrammierer nicht zumutbar, sich in der Anordnung seiner Hochsprachenbefehle nach Hardwareerfordernissen zu richten.

- Während bei der Mikroprogrammierung die Konflikterkennung dominiert und die Syntax/Code–Zuordnung eher einfach ist, stellt bei der Makroprogrammierung die Syntax/Code–Zuordnung den wesentlichsten Punkt dar. Vielfältige Optimierungsstrategien wie etwa *instruction scheduling*, *register allocation*, *loop unrolling* oder *software pipelining* müssen unter einen Nenner gebracht werden. Diese Aufgabe wäre so schon schwer genug, dennoch sind auch die Optimierungsstrategien teilweise hardwareabhängig: Beim *register renaming* zum Beispiel ist die Anzahl der verfügbaren Register die elementare Größe.

- Die Syntax/Code–Zuordnung muß mit den Methoden des *tree parsing* die Relation der Zwischensprache zum Maschinencode herstellen, während bei der Mikroprogrammierung das *pattern matching* der Mini-Bäume ausreichend war. Details sind in [16] und [11] beschrieben.

- Die durch die HCDL zu beschreibenden Organisationselemente sind teilweise *komplexer* als die durch die MDL beschriebenen: Man denke etwa an die Beschreibung einer vierstufigen Pipeline oder einer superskalaren *instruction dispatch and branch unit* wie beim MC88110 (siehe [7]).

Alles in allem ist die Methode der CIAD deutlich komplexer als die Verwendung der MDL im Bereich der Mikroprogrammierung. Andererseits würde der Aufwand für die Entwick-

[2]Hohe zeitliche Parallelität durch besonders tiefe Pipeline ermöglicht die Parallelität vieler Befehle auf einer Hardware.

[3]Hohe räumliche Parallelität durch Reduplikation wichtiger Ressourcen ermöglicht einen Durchsatz von mehr als einem Befehl pro Taktzyklus.

lung dieser Methode durch gewaltige Vorteile in den Bereichen der Hardwareentwicklung und der Compilerentwicklung gleichermaßen belohnt werden.

Als Vollendung des Hardware/Software-Codesign sollten schließlich Werkzeuge zur Verfügung stehen, die eine integrierte Entwicklung von Hardware und Compiler ermöglichen, und zwar unter Verwendung bestehender Hardware-Bibliotheken einerseits und unter Einbeziehung bewährter Prinzipien aus dem Compilerbau andererseits. Genaugenommen geht es dabei nur um das Compiler-Backend, das insbesondere durch jüngste Ansätze mittels ANDF[4] [3] noch deutlicher vom Frontend abgetrennt ist.

[4]Architecture Neutral Distribution Format.

3.3 High Level Synthese

Bedeutet der Sieg der RISC–Architektur über die CISC–Architektur auch das endgültige Aus für die Mikroprogrammierung? Nicht unbedingt! Es ist offensichtlich, daß der eigentliche Nachteil der Mikroprogrammierung bei der Geschwindigkeit liegt. Das liegt daran, daß die Mikroprogrammierung von der *Speichertechnologie* abhängig ist: Die Dauer des Taktzyklus ist durch die Zeit beschränkt, die benötigt wird, um einen Mikrobefehl aus dem Mikroprogrammspeicher zu lesen. In den späten 60er und frühen 70er Jahren waren schnelle Halbleiterspeicher für den Mikroprogrammspeicher verfügbar, während der Hauptspeicher noch mit Ferritkernen aufgebaut war. Da sich die beiden Technologien bezüglich der Zyklusdauer um den Faktor zehn unterscheiden, war die Mikroprogrammierung damals eine rentable Vorgangsweise. In den späten 70er Jahren mit dem Aufkommen der Cache–Speicher war aber dieser Vorteil dahin, weil Hauptspeicher und Mikroprogrammspeicher mit derselben Technologie aufgebaut waren. Daher erfolgte der Übergang zur RISC–Architektur.

Allerdings ist die Mikroprogrammierung nach wie vor eng mit der Speichertechnologie verbunden: Es wäre zumindest prinzipiell denkbar, daß irgendwann ROM–Speicher wesentlich schneller als RAM–Speicher werden, oder daß die Cache–Speicher aus irgendeinem Grund ineffizient würden. In diesem Fall würde die Mikroprogrammierung wieder rentabel eingesetzt werden können, auch RISC–Prozessoren könnten wieder in stärkerem Maße mikroprogrammiert werden.

Darüberhinaus gibt es eine große Anzahl von *Spezialprozessoren*, die mit den Methoden der Mikroprogrammierung eine bequeme Anpassung an die Bedürfnisse des Benutzers ermöglichen. Dabei handelt es sich zumeist um solche Programmieraufgaben, deren Komplexität für eine reine Hardware-Steuerung zu hoch ist, die aber dennoch nicht den Aufwand der Erstellung eines Hochsprachen-Compilers lohnen. Als aktuelles Beispiel sei hier der Mikrocontroller SAB 88C166 genannt, der ein Umprogrammieren der Firmware im System und ohne Hardware–Veränderungen ermöglicht (siehe [5]). Auch hier bietet sich der Methode der Generierung des Firmwarecompilers mittels MDL ein weites Betätigungsfeld.

Letztendlich verlagert sich mit der Einführung der RISC–Architektur die Problematik der Firmware in das Backend des Hochsprachen–Compilers: Durch die starke Hardwareabhängigkeit eines RISC–Compilers muß jetzt der Compilerentwickler Probleme bewältigen, die früher im Bereich der Firmware aufgelöst wurden und für die Software unsichtbar blieben. Wie dabei die Methode der MDL in angepaßter Form ebenfalls sinnvoll eingesetzt werden kann, wurde im vorherigen Abschnitt 3.2 angedeutet.

An dieser Stelle soll jedoch ein weiterer Gedanke präsentiert werden, der sich als logische Fortsetzung der Generierung des Firmwarecompilers ergibt: *Die Generierung der Firmware selbst.* Anhand des Y-Chart Diagramms in Abbildung 3.4 soll dieser Ansatz kurz vorgestellt werden, der Grundgedanke dazu wurde aus [6] entnommen, wo auch nähere Details zum Y-Chart-Diagramm und zur High-Level-Synthese nachgelesen werden können.

Das Y-Chart-Diagramm beschreibt den Prozeß des Hardware-Entwurfes auf fünf Ab-

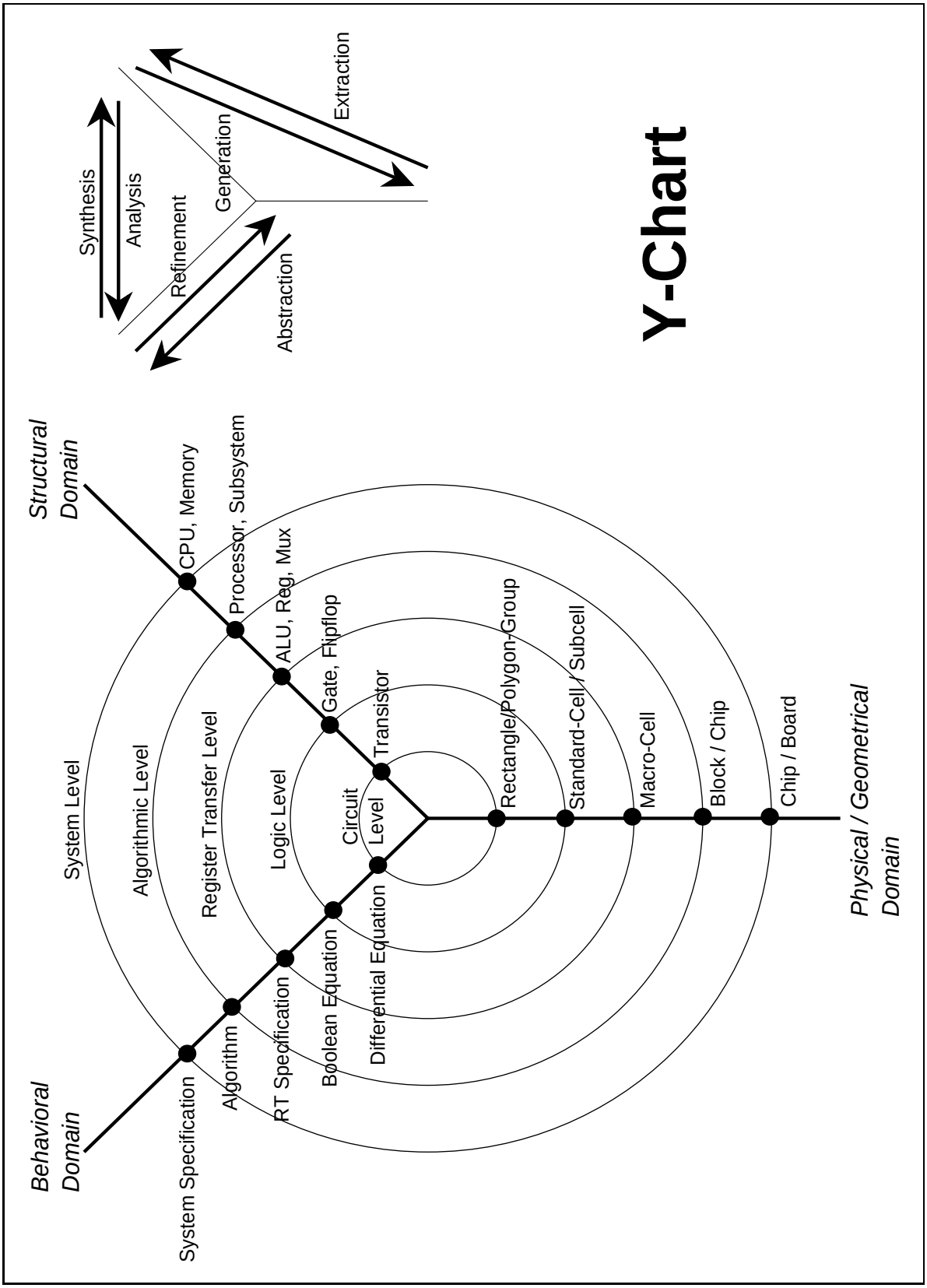

Abbildung 3.4: Das Y-Chart Diagramm.

straktionsebenen und in drei Bereichen oder Sichtweisen. Die drei Bereiche sind das Verhalten, die Struktur und der geometrisch-physikalische Aufbau. Die Abstraktionsebenen sind:

System Level: Das Gesamtsystem wird selten formal spezifiziert, sondern meistens in einer nicht formalisierten Beschreibung. Strukturell besteht die Systemebene aus der CPU, dem Speicher und der Peripherie. Physikalisch besteht das Gesamtsystem aus einem Board mit einem oder mehreren Chips darauf.

Algorithmic Level: Der Algorithmus beschreibt das Verhalten dieser Ebene bereits nach streng formalen Gesichtspunkten. Strukturell aufgeteilt besteht diese Ebene aus einem Prozessor und diversen Subsystemen. Physikalisch handelt es sich um einen Block oder Chip.

Register Transfer Level: Das Verhalten dieser Ebene wird typischerweise mit einer Hardwarebeschreibungssprache wie VHDL spezifiziert. Auch MDL beschreibt die Hardware auf der Register-Transfer-Ebene, wobei jedoch keine *funktionale* Beschreibung enthalten ist. Strukturell besteht diese Ebene aus Registern, ALUs, Multiplexern und den Verbindungswegen wie Busse und Steuerleitungen. Physikalisch wird etwa die Komplexität einer Makro-Zelle von dieser Abstraktionsebene beschrieben.

Logic Level: Das Verhalten kann mit Booleschen Gleichungen und Wahrheitstabellen formuliert werden, die maßgeblichen Elemente der Struktur sind Gatter und Flipflops, die maßgeblichen Elemente aus physikalischer Sicht die Standard-Zellen und Subzellen.

Circuit Level: Das wesentlichste Strukturelement dieser Ebene ist der Transistor neben anderen aktiven und passiven Bauelementen. Eine adäquate Beschreibungsform für den Transistor ist ein Differentialgleichungssystem, die physikalisch-geometrische Implementierung kennt hier vor allem die Rechtecke und Polygonzüge der einzelnen Schichten.

Die Übergänge zwischen den verschiedenen Abstraktionsebenen in den drei Bereichen werden wie folgt benannt:

Synthesis: Als Synthese bezeichnet man den Übergang vom Verhaltens-Bereich in den strukturellen Bereich auf derselben Abstraktionsebene. Dabei wird die jeweilige Spezifikation durch strukturelle Information ergänzt.

Analysis: Die Analyse entspricht umgekehrt einem Übergang vom Strukturbereich zum Verhaltensbereich, wobei aus der Struktur das Verhalten abgeleitet wird.

Refinement: Die Verfeinerung ist ein Abstieg in die nächstniedrigere Abstraktionsebene im selben Bereich. Dabei wird Detailinformation ergänzt.

Abstraction: Die Abstraktion ist im Gegensatz dazu ein Übergang von einer detaillierteren zu einer allgemeineren Abstraktionsebene im selben Bereich.

Generation: Als Generierung im Sinne des Y-Chart Diagramms wird ein Übergang vom strukturellen Bereich in den physikalischen Bereich auf derselben Abstraktionsebene bezeichnet. Dabei wird physikalische Information hinzugefügt. Wenn man hingegen, wie in der Praxis üblich, die physikalische Ausprägung nicht generiert, sondern auf eine Bibliothek abbildet, so bezeichnet man den Vorgang als *Technology Mapping*. Die zugehörige Abstraktionsebene ist zumeist die Logische oder die Register-Transfer-Ebene.

Extraction: Als Extrahierung benennt man den zur Generierung inversen Vorgang, wo aus der physikalisch-geometrischen Ausprägung die Struktur gewonnen wird.

Das Top-Down-Design einer Hardware beginnt nun mit der Spezifikation auf Systemebene. Jeder weitere Schritt besteht aus einem Syntheseschritt und gleichzeitig einem Verfeinerungsschritt. So wird zum Beispiel die Synthese auf Register-Transfer-Ebene (RT-Ebene) begleitet von einer Verfeinerung im Verhaltens-Bereich hin zu einer Beschreibung der einzelnen Elemente in Form Boolescher Gleichungen. Dies ist auch zur Zeit die abstrakteste in der Praxis vorkommende Synthese: Aus der RT-Beschreibung wird der Datenpfad verfeinert zu Booleschen Gleichungen, während der Steuerpfad synthetisiert wird. Und genau hier könnte die Generierung der Firmware einsetzen, indem als Steuerpfad kein PLA[5] synthetisiert wird, sondern ein Mikroprogramm.

Bei dieser Synthese müßte die Prüfung auf Hardwarekonflikte *implizit* erfolgen, da ja kein Compiler mehr generiert wird. Die *Methoden* der Konfliktprüfung selbst bleiben jedoch die gleichen, sodaß es sinnvoll erscheint, die der MDL innewohnenden Grundgedanken in diesem Bereich anzuwenden. Von der Struktur auf RT-Ebene kann dann direkt das Technology-Mapping des Datenpfades auf die Macro-Zellen erfolgen, während der Steuerpfad als Mikroprogramm direkt in einem ROM[6] implementiert wird. Es wäre nur noch praktisch zu prüfen, ob die auf diese Weise generierte Hardware auch den Anforderungen an Geschwindigkeit und Kosten gerecht wird.

[5] Programable Logic Array.
[6] Read Only Memory.

3.4 Ausblick und Schluß

Zur Zeit hat die Mikroprogrammierung ihre Daseinsberechtigung noch nicht verloren, und die vorgestellten Methoden zur Generierung des Firmwarecompilers können den Entwicklungszyklus mikroprogrammierter Hardware drastisch beschleunigen. Darüberhinaus erlaubt eine Verallgemeinerung der dargelegten Methoden die direkte Generierung der Firmware selbst, wodurch ein neuer Aspekt in der High-Level Synthese aufgezeigt wird.

Andererseits hat sich mit der Einführung der RISC–Architektur die Problematik der Firmware in das Backend des Hochsprachen–Compilers verlagert: Durch die starke Hardwareabhängigkeit eines RISC–Compilers muß jetzt der Compilerentwickler Probleme bewältigen, die früher im Bereich der Firmware aufgelöst wurden und für die Software unsichtbar blieben. Auch hier können die Methoden der MDL in angepaßter Form sinnvoll eingesetzt werden. Den größten Nutzen bringt dabei die Möglichkeit, den in Entwicklung befindlichen Prozessor in allen Stadien an bestehender Software zu messen, was letztendlich dazu beiträgt, daß die Entwickler der Hardware das in [14] so treffend formulierte eigentliche Ziel nie aus den Augen lassen: "A computer architect designes machines to run programs" □

Anhang A

Lexikalische Regeln und Syntax

Die Lexikalischen Regeln und die Syntax der Sprache MDL/HD sind in [4] ausführlich beschrieben, daher wird in diesem Anhang nur die Erweiterung der MDL durch den BED-Teil dargestellt.

A.1 Lexikalische Regeln

MDL besteht, wie jede Sprache, aus einer Menge von Sätzen, die ihrerseits Folgen von *Eingabesymbolen* sind. Normalerweise besitzt eine Programmiersprache drei Arten von Eingabesymbolen: *Operatoren und Begrenzer* werden als kurze Folge von Sonderzeichen dargestellt, *reservierte Schlüsselwörter* werden als vordefinierte Folge von Buchstaben dargestellt, deren Bedeutung nicht verändert werden kann, und schließlich vom Benutzer eingeführte Eingabesymbole, als da sind: *Konstanten* und *Identifier oder Bezeichner*. Zusätzlich gibt es noch Zwischenraumzeichen[1], die zwar die Eingabesymbole gegeneinander abgrenzen, sonst aber unwesentlich sind.

Die lexikalischen Regeln beschreiben jenen Teil des Generators, der aus der unstrukturierten Folge der Eingabezeichen jene Eingabesymbole konstruiert, die schließlich das *Alphabet* der Grammatik darstellen.

Die lexikalische Struktur von MDL hat sich durch die Ergänzung durch den BED-Teil kaum geändert, lediglich zwei Veränderungen sind zu berücksichtigen:

1. Tabelle A.1 zeigt die neuen MDL–Schlüsselwörter.
2. Im Bereich der Backend-Beschreibung hat der Tabulator eine zusätzliche Bedeutung erhalten.

Die Anzahl der unmittelbar aufeinanderfolgenden Tabulatoren vor einer Regelzeile der Baumgrammatik bestimmt die Tiefe im IR-Baum: Dabei kann nach einer solchen Tabulatorfolge nur ein `Identifier` oder eine eckige Klammer, gefolgt von einem `Identifier`,

[1] z. B. Leerzeichen, Tabulatorzeichen und Zeilentrenner.

END	ENDGRAMMAR	GRAMMAR
IS	LEFT	RIGHT
SUBTREE	TO	TYPE
UNION	VALUE	

Tabelle A.1: Die neuen MDL/BED–Schlüsselwörter.

erwartet werden. Diese beiden Formen werden nun bereits lexikalisch umgesetzt, und zwar werden z.B. 3 Tabulatoren[2] durch `[3]` ersetzt. Daher erwartet die Grammatik an dieser Stelle *immer* einen Ausdruck der Form `[i]`, obwohl die Eingabe dort nur Tabulatoren erhält.

A.2 Syntax

Nun wird die erweiterte Syntax von MDL und ihre Implementierung beschrieben, wobei die Darstellung mittels EBNF[3] verwendet wird.

Die BNF–Definition einer Sprache, in weiterer Folge auch *Grammatik* genannt, besteht aus einer Folge von *Regeln*. Eine Regel besteht aus einer linken Seite und einer rechten Seite, die durch einen Doppelpunkt getrennt sind. Die linke Seite besteht aus einem einzigen, eindeutigen *Grammatikbegriff* (non-terminal-symbol). Die rechte Seite besteht aus einer Folge von einer oder mehreren *Formulierungen*, die voneinander durch einen `|` getrennt sind. Eine Formulierung in einer Regel kann leer sein; sonst besteht eine Formulierung aus einer Folge von Grammatikbegriffen oder *Eingabesymbolen* (terminal symbols). Grammatikbegriffe und Eingabesymbole sollen hier unter dem gemeinsamen Begriff *Symbole* zusammengefaßt werden.

Eine Grammatik beschreibt eine Sprache dadurch, daß sie erklärt, welche Sätze gebildet werden können. Der Grammatikbegriff auf der linken Seite der ersten Regel ist dabei das *Startsymbol*. Für jeden Grammatikbegriff muß eine Regel existieren und jede Formulierung aus dieser Regel kann an Stelle des Grammatikbegriffes eingefügt werden. Diese Ersetzung geschieht solange, bis eine Folge von Eingabesymbolen, ein *Satz*, aus dem Startsymbol der Grammatik *produziert* wurde. Die Eingabesymbole wiederum bilden das Alphabet der Grammatik. In unserem Fall werden diese Eingabesymbole aus der lexikalischen Analyse gewonnen. Dabei ist besonders die weiter oben beschriebene Umsetzung der Tabulatoren vor einer Regelzeile der Baumgrammatik in die Form `[i]` zu beachten. Weitere Details sind in [20] nachzulesen.

Die hier verwendete EBNF benutzt noch einige *spezielle Symbole* innerhalb der Formulierungen: Die eckigen Klammern `[` und `]` beschreiben optionale Teile einer Formulierung, die entweder einmal oder gar nicht vorkommen können. In den geschweiften Klammern { und } werden Formulierungen eingeschlossen, die beliebig oft (und auch gar nicht) auftreten können. Werden die geschweiften Klammern aber mit `}+` abgeschlossen, dann muß die

[2] Aber nur unmittelbar vor einer Regelzeile!

[3] Extended Backus Naur Form.

eingeschlossene Formulierung mindestens einmal auftreten. Die runden Klammern (und) schließlich führen einen Vorrang ein, um eine Auswahl mittels | von der Verkettung der Symbole trennen zu können. Außerdem besitzt jede Regel eine Nummer, mittels derer eventuelle Erläuterungen auf die jeweilige Regel verweisen können.

Das Alphabet hat sich, abgesehen von den neuen Schlüsselwörtern, nicht verändert, sodaß nun die neuen Regeln der Grammatik aufgelistet werden. „Startpunkt“ für die Grammatik-*Erweiterung* ist `syntax` in Regel S36, welches auf der rechten Seite der ursprünglichen Regel S3 auftritt. Die Regeln S1 bis S35 sind in [4] aufgezählt und bleiben bis auf die Regel S3 unverändert. Nur die veränderte Regel S3 wird hier angeführt.

In Regel S37 wird der Grammatikbegriff `c_declaration` verwendet, der nirgends auf der linken Seite auftritt. Dieser Begriff beschreibt die Deklaration von Elementen einer `union` in C und wurde nicht weiter ausgeführt, um die Beschreibung nicht unübersichtlich werden zu lassen. Exakte Syntaxdiagramme sind in [23] enthalten.

Rein optisch kann man Grammatikbegriffe (non-terminal symbols) und Eingabesymbole (terminal symbols) dadurch voneinander unterscheiden, daß die Grammatikbegriffe alle durchwegs klein geschrieben sind. Eingabesymbole hingegen können entweder als Schlüsselwörter ganz groß geschrieben sein, als Schlüsselwörter mit Inhalt zumindest mit einem Großbuchstaben beginnen (Bezeichner und Konstanten) oder es handelt sich überhaupt um einzelne Eingabezeichen, die dann in doppelten Anführungszeichen eingeschlossen sind.

S3

```
definition
  : map_definition
  | alias_definition
  | resource_definition
  | connection_definition
  | microp
  | syntax
```

S36

```
syntax
  : GRAMMAR [ gram_head ] { rule } ENDGRAMMAR
```

S37

```
gram_head
  : UNION c_declaration END { TYPE "<" Identifier ">" Identifier }
```

S38

```
rule
  : ( ROOT | SUBTREE Identifier )
    ":" formulation { "|" formulation } END
```

S39

```
formulation
  : { gram_line }+
```

S40

```
gram_line
  : "[" integer "]"
    [ LEFT | RIGHT ] ( Identifier | "[" Identifier "]" )
    [ IS Identifier ]
    [ VALUE String_constant ]
    [ TO  ( integer { "," integer } )
        | ( "(" integer "," integer ")" ]
```

Anhang B

Baumgrammatik und semantische Aktionen

B.1 Baumgrammatik

Das erste Beispiel auf den folgenden Seiten stellt die Baumgrammatik für das Backend eines typischen Prozessors dar. Dabei handelt es sich um jenen Teil des Prozessors, der die arithmetisch-logischen Operationen durchführt. Das Beispiel ist zwar richtig und bearbeitbar, dennoch ist es nicht vollständig: Nur einige IR-Referenzen sind vollständig ausgeführt, die übrigen sind nur syntaktisch beschrieben, um einen Einblick in das Aussehen einer solchen Backend-Beschreibung anhand eines etwas längeren Beispieles zu geben. Die in den semantischen Aktionen verwendeten Unterprogramme und C-Definitionen werden im folgenden Abschnitt vorgestellt.

```
/*

SAMPLE PROCESSOR P95

*/

UNION
    int                t_int;
    struct range *     t_ran;
    struct variable *  t_var;
END

TYPE <t_var> Operand
TYPE <t_var> Simple_Operand
TYPE <t_ran> Index
TYPE <t_ran> Range
TYPE <t_int> Pointer

ROOT
/*
<AC> = AND (<AC>,<B>);;
<AC> = OR (<AC>,<B>);;
<AC> = XOR (<AC>,<B>);;
<AC> = NXOR (<AC>,<B>);;
*/
:   [Assignment]
       Operand IS AC_Op
       [Builtin] IS opcode VALUE "AND","OR","XOR","NXOR"
                           TO 0b10111, 0b10101, 0b10011, 0b10001
           [List]
              Operand IS AC_Op,
              Operand IS B_Op
    { /* conflict check */
      alu_conflict($AC_Op,$B_Op);
      /* encoding */
      encode("fgroup",$0b00);
      encode("ffunction",$opcode);
      adr_encode($AC_Op,$B_Op);
    }
/*
<AC> = <AC> & <B>;;
<AC> = <AC> | <B>;;
<AC> = <AC> # <B>;;
<AC> = <AC> + <B>;;
<AC> = <AC> - <B>;;
<AC> = <AC> $+ <B>;;
<AC> = <AC> $- <B>;;
*/
|   [Assignment]
       Operand IS AC_Op
       [Binop] VALUE "&","|","#","+","-","$+","$-"
           Operand IS AC_Op
           Operand IS B_Op
/*
<AC> = NOT <B>;;
<AC> = - <B>;;
<AC> = $- <B>;;
*/
|   [Assignment]
       Operand IS AC_Op
```

Beispiel B.1: Die Backendbeschreibung `processor.mdl`.

```
        [Unop] VALUE "NOT", "-", "$-"
           Operand IS B_Op
/*
<AC> = NOT (<AC> # <B>);;
*/
|   [Assignment]
       Operand IS AC_Op
       [Unop] VALUE "NOT"
           [Binop] VALUE "#"
              Operand IS AC_Op
              Operand IS B_Op
/*
<AC> = <AC> + <B> + CARRY;;
<AC> = <AC> $+ <B> + CARRY;;
*/
|   [Assignment]
       Operand IS AC_Op
       [Binop] VALUE "+"
           [Binary_Operation] VALUE "+", "$+"
              Operand IS AC_Op
              Operand IS B_Op
           [Variable] VALUE "CARRY"
/*
<AC> = <AC> + NOT <B> + CARRY;;
<AC> = <AC> $+ NOT <B> + CARRY;;
*/
|   [Assignment]
       Operand IS AC_Op
       [Binop] VALUE "+"
           [Binop] VALUE "+", "$+"
              Operand IS AC_Op
              [Unop] VALUE "NOT"
                 Operand IS B_Op
           [Variable] VALUE "CARRY"
/*
<AC> = NOT <B> + CARRY;;
<AC> = $NOT <B> + CARRY;;
*/
|   [Assignment]
       Operand IS AC_Op
       [Binop] VALUE "+"
           [Unop] VALUE "NOT", "$NOT"
              Operand IS B_Op
           [Variable] VALUE "CARRY"
/*
<AC> = <B>;;
*/
|   [Assignment]
       Operand IS AC_Op
       Operand IS B_Op
/*
<AC> = TRI(<B>);;
<AC> = TREXA(<B>);;
<AC> = TREXB(<B>);;
<AC> = INS(<B>);;
<AC> = EXT(<B>);;
*/
|   [Assignment]
       Operand IS AC_Op
       [Builtin] VALUE "TRI", "TREXA", "TREXB", "INS", "EXT"
```

Beispiel B.1 (Fortsetzung): Die Backendbeschreibung `processor.mdl`.

```
            Operand IS B_Op
/*
<AC> = TR(SIGN || <B>);;
*/
|   [Assignment]
        Operand IS AC_Op
        [Builtin] VALUE "TRI"
            [Binop] VALUE "||"
                [Variable] VALUE "SIGN"
                Operand IS B_Op
/*
<AC> = SL(<B>,<X>);;
<AC> = SL(<B> || 0, <X>);;
<AC> = SL(<B> || 1, <X>);;
<AC> = SL(<B> || SIGN, <X>);;
<AC> = SL(<B> || SC, <X>);;
*/
|   [Assignment]
        Operand IS AC_Op
        [Builtin] VALUE "SL"
            [List]
                Shift_Left_Expression
                [Constant] VALUE "1","2","3","4"
/*
<AC> = SR(<B>,<X>);;
<AC> = SR(0 || <B>, <X>);;
<AC> = SR(1 || <B>, <X>);;
<AC> = SR(SIGN || <B>, <X>);;
<AC> = SR(SC || <B>, <X>);;
*/
|   [Assignment]
        Operand IS AC_Op
        [Builtin] VALUE "SR"
            [List]
                Shift_Right_Expression
                [Constant] VALUE "1","2","3","4"
/*
<AC> = INS(SL(<B>,<X>));;
<AC> = INS(SL(<B> || 0,<X>));;
<AC> = INS(SL(<B> || 1,<X>));;
<AC> = EXT(SL(<B>,<X>));;
<AC> = EXT(SL(<B> || 0,<X>));;
<AC> = EXT(SL(<B> || 1,<X>));;
*/
|   [Assignment]
        Operand IS AC_Op
        [Builtin] VALUE "INS", "EXT"
            [Builtin] VALUE "SL"
                [List]
                    Shift_Left_Expression_1
                    [Constant] VALUE "1","2","3","4"
/*
<AC> = INS(SR(<B>,<X>));;
<AC> = INS(SR(0 || <B>,<X>));;
<AC> = INS(SR(1 || <B>,<X>));;
<AC> = EXT(SR(<B>,<X>));;
<AC> = EXT(SR(0 || <B>,<X>));;
<AC> = EXT(SR(1 || <B>,<X>));;
*/
|   [Assignment]
```

Beispiel B.1 (Fortsetzung): Die Backendbeschreibung `processor.mdl`.

```
        Operand IS AC_Op
        [Builtin] VALUE "INS", "EXT"
            [Builtin] VALUE "SR"
                [List]
                    Shift_Right_Expression_1
                    [Constant] VALUE "1","2","3","4"
/*
STATUS = <AC> - <B>;;
STATUS = <AC> $- <B>;;
*/
|   [Assignment]
        [Variable] VALUE "STATUS"
        [Binop] VALUE "-", "$-"
            Operand IS AC_Op
            Operand IS B_Op
/*
STATUS = <AC> + NOT <B> + CARRY;;
STATUS = <AC> $+ NOT <B> + CARRY;;
*/
|   [Assignment]
        [Variable] VALUE "STATUS"
        [Binop] VALUE "+"
            [Binop] VALUE "+", "$+"
                Operand IS AC_Op
                [Unop] VALUE "NOT"
                    Operand IS B_Op
            [Variable] VALUE "CARRY"
END /* ROOT */

SUBTREE Shift_Left_Expression
:   Shift_Left_Expression_1
|   [Binop] VALUE "||"
        Operand IS B_Op
        [Variable] VALUE "SIGN","SC"
END /* SUBTREE Shift_Left_Expression */

SUBTREE Shift_Left_Expression_1
:   Operand IS B_Op
|   [Binop] VALUE "||"
        Operand IS B_Op
        [Constant] VALUE "0","1"
END /* SUBTREE Shift_Left_Expression_1 */

SUBTREE Shift_Right_Expression
:   Shift_Right_Expression_1
|   [Binop] VALUE "||"
        [Variable] VALUE "SIGN","SC"
        Operand IS B_Op
END /* Shift_Right_Expression */

SUBTREE Shift_Right_Expression_1
:   Operand IS B_Op
|   [Binop] VALUE "||"
        [Constant] VALUE "0","1"
        Operand IS B_Op
END /* Shift_Right_Expression_1 */

SUBTREE Operand
:   Simple_Operand IS Operand
    { $Operand->address = make_address($Operand,0);
```

Beispiel B.1 (Fortsetzung): Die Backendbeschreibung `processor.mdl`.

```
      $$ = $Operand; }
|   [Builtin] VALUE "HSTR"
        [List]
            Simple_Operand IS Operand
            [Constant] IS digits VALUE "1","2","3","4","5","6","7","8" TO (1,1)
    { $Operand->bits->length = $digits * 4; /* explizite Verarbeitungslaenge */
      $Operand->address = make_address($Operand,1);
      $$ = $Operand; }
|   [Builtin] VALUE "BSTR"
        [List]
            Simple_Operand IS Operand
            [Constant] IS bytes VALUE "1","2","3","4" TO (1,1)
    { $Operand->bits->length = $digits * 8; /* explizite Verarbeitungslaenge */
      $Operand->address = make_address($Operand,0);
      $$ = $Operand; }
|   [Builtin] VALUE "BSTR"
        [List]
            Simple_Operand IS Operand
            [Builtin] IS direction VALUE "L","R" TO 1,0
    { $Operand->direction = $direction;
      $Operand->address = make_address($Operand,3);
      $$ = $Operand; }
END /* SUBTREE Operand */

SUBTREE Simple_Operand
:   [Variable] IS reg
    { $$ = make_operand($reg,implicit_index($reg),implicit_bits($reg),0); }
|   [Variable] IS reg
        LEFT Index IS field
    { $$ = make_operand($reg,$field,implicit_bits($reg),0); }
|   [Variable] IS reg
        LEFT Index IS field
        RIGHT [Area]
            [Constant] IS offset
            [Constant] IS length
    { $$ = make_operand($reg,$field,make_range(0,0,0,$offset,$length,0)); }
|   [Variable] IS reg
        RIGHT Range IS bits
    { $$ = make_operand($reg,implicit_index($reg),$bits); }
END /* SUBTREE Simple_Operand */

SUBTREE Index
:   Range
|   [Binop] VALUE "#"
        Pointer IS pointer
        [Constant] IS xor VALUE "0", "1" TO 0, 1
    { $$ = make_range(1,pointer,xor,0,0); }
END /* SUBTREE Index */

SUBTREE Range
:   [Area]
            [Constant] IS offset
            [Constant] IS length
    { $$ = make_range(0,0,0,offset,length); }
|   Pointer IS pointer
    { $$ = make_range(1,pointer,0,0,0); }
END /* SUBTREE Range */

SUBTREE Pointer
:   [Variable] IS reg VALUE "P0","P1","P2","P3","P4","P5","P6","P7" TO (0,1)
```

Beispiel B.1 (Fortsetzung): Die Backendbeschreibung `processor.mdl`.

```
                          /* P0->0, P1->1, etc.. (offset,step) */
    { $$ = $reg; }
END /* SUBTREE Pointer */
```

Beispiel B.1 (Fortsetzung): Die Backendbeschreibung `processor.mdl`.

B.2 Backend Programmfragment

Das nun folgende Beispiel zeigt das zur Baumgrammatik gehörende Programmfragment. Zwecks höherer Übersichtlichkeit der Baumgrammatik selbst wurden große Teile der semantischen Aktionen als Unterprogramme formuliert, die in diesem C-Programmfragment näher ausgeführt sind. Gemeinsam mit der MDL/HD-Beschreibung in Beispiel 1.1 ab Seite 12 und der Baumgrammatik in MDL/BED in Beispiel B.1 ab Seite 67 bildet dieses Beispiel einen Ausschnitt aus einer vollständigen Eingabe für die Generierung des Firmwarecompilers.

```
/* C - definitions for semantic actions */

struct range
{
    int flag;
    int pointer;
    int xor;
    int offset;
    int length;
};

struct range *
make_range(flag, pointer, xor, offset, length)
    int flag, pointer, xor, offset, length;
{
    struct range *new;

    new = (struct range *) calloc(1, sizeof(struct range));
    new->flag = flag;
    new->pointer = pointer;
    new->xor = xor;
    new->offset = offset;
    new->length = length;
    return new;
}

struct variable
{
    int number;
    struct range *index;
    struct range *bits;
    int direction;          /* 0 = nothing, -1 = left, +1 = right */
    struct address *address;
};

struct variable *
make_operand(number, index, bits, direction)
    int number;
    struct range *index, *bits;
    int direction;
{
    struct variable *new;

    new = (struct variable *) calloc(1, sizeof(struct variable));

    new->number = number;
    new->index = index;
    new->bits = bits;
    new->direction = direction;
    new->address = (struct address *) 0;
    return new;
}

struct address
{
    short art;
    short DL;
    short D;
    short W;
    short B;
    short BL;
    short R;
    short WI;
    short X;
    short DW;
    short BI;
    short A;
};

struct address *
make_address(operand, adr)
    struct variable *operand;
    int adr;                /* addressing mode */
{
    struct address *new;

    new = (struct address *) calloc(1, sizeof(struct address));

    if ((adr == 0) && (operand->index->flag == 0) &&
        (operand->bits->flag == 0) && (operand->bits->length % 8 == 0))
    {
        /* Byte direct */
        new->art = 2;
        new->BL = operand->bits->length / 8 - 1;
        if (strcmp("UT", number_to_string(operand->number)) == 0)
            new->R = 0;
        else
        if (strcmp("GR", number_to_string(operand->number)) == 0)
            new->R = 1;
        else
        if (strcmp("FR", number_to_string(operand->number)) == 0)
            new->R = 2;
        else
        if (strcmp("UX", number_to_string(operand->number)) == 0)
            new->R = 3;
        new->W = operand->index->offset;
        new->B = operand->bits->offset / 4;
        return new;
    };

    if ((adr == 1) /* von HSTR -> sicher Digit direkt */ ||
        ((adr == 0) && (operand->index->flag == 0) &&
        (operand->bits->flag == 0) && (operand->bits->length % 4 == 0)))
    {
        /* Digit direct */
        new->art = 1;
        new->DL = operand->bits->length / 4 - 1;
        new->B = operand->bits->offset / 8;
        new->D = (operand->bits->offset - 8 * new->B) / 4;
        new->W = operand->index->offset;
```

Beispiel B.2: Backend Programmfragment `processor.c`.

```
    return new;
  }

  if ((adr == 0) && (operand->index->flag == 1) &&
      (operand->bits->flag == 0))
    {
      /* Byte indirect (word) */
      new->art = 3;
      new->BL = operand->bits->length / 8 - 1;
      if (strcmp("UT", number_to_string(operand->number)) == 0)
          new->R = 0;
        else
        if (strcmp("GR", number_to_string(operand->number)) == 0)
            new->R = 1;
        else
        if (strcmp("FR", number_to_string(operand->number)) == 0)
            new->R = 2;
        else
        if (strcmp("UX", number_to_string(operand->number)) == 0)
            new->R = 3;
      new->WI = operand->index->pointer;
      new->X = operand->index->xor;
      new->B = operand->bits->offset / 8;
      return new;
    };

  if ((adr == 0) && (operand->index->flag == 0) &&
      (operand->bits->flag == 1))
    {
      /* Byte indirect (Byte, direct length) */
      new->art = 4;
      new->BL = operand->bits->length / 8 - 1;
      if (strcmp("UT", number_to_string(operand->number)) == 0)
          new->R = 0;
        else
        if (strcmp("GR", number_to_string(operand->number)) == 0)
            new->R = 1;
        else
        if (strcmp("FR", number_to_string(operand->number)) == 0)
            new->R = 2;
        else
        if (strcmp("UX", number_to_string(operand->number)) == 0)
            new->R = 3;
      new->DW = operand->index->offset / 2;
      new->BI = operand->bits->pointer;
      return new;
    };

  if (adr == 3)
  {
      /* Byte indirekt (Byte, indirect length) */
      new->art = 5;
      new->A = operand->direction;
      if (strcmp("UT", number_to_string(operand->number)) == 0)
          new->R = 0;
        else
        if (strcmp("GR", number_to_string(operand->number)) == 0)
            new->R = 1;
        else
        if (strcmp("FR", number_to_string(operand->number)) == 0)
            new->R = 2;
        else
        if (strcmp("UX", number_to_string(operand->number)) == 0)
            new->R = 3;
      new->DW = operand->index->offset / 2;
      new->BI = operand->bits->pointer;
      return new;
    };

  }

struct range *
implicit_index(operand)
  struct variable *operand;
{
  /* returns the implicit index of a register */
}

struct range *
implicit_bits(operand)
  struct variable *operand;
{
  /* returns the implicit bitrange of a register */
}

char *
number_to_string(number, type)
  int number, type;
{
  /* liefert die textuelle Repraesentation zur "number" beim Typ "type" */
  /* returns the string of "number" with "type" */
}

int
string_to_number(string, type)
  char *string;
  int type;
{
  /* returns the number of "string" with "type" */
}

void
adr_encode(ac_op, b_op)
  struct variable *ac_op, *b_op;
{
  switch (ac_op->address->art)
    {
    case 1:
```

Beispiel B.2 (Fortsetzung): Backend Programmfragment `processor.c`.

```
        break;
      case 3:
        encode("fb.art", 2);
        encode("fb.bl", b_op->address->BL);
        encode("fb.r", b_op->address->R);
        encode("fb.wi", b_op->address->WI);
        encode("fb.x", b_op->address->X);
        encode("fb.b", b_op->address->B);
        break;
      case 4:
        encode("fb.art", 3);
        encode("fb.bl", b_op->address->BL);
        encode("fb.r", b_op->address->R);
        encode("fb.dw", b_op->address->DW);
        encode("fb.x", 0);
        encode("fb.bi", b_op->address->BI);
        break;
      case 5:
        encode("fb.art", 3);
        encode("fb.a", b_op->address->A);
        encode("fb.r", b_op->address->R);
        encode("fb.dw", b_op->address->DW);
        encode("fb.x", 1);
        encode("fb.bi", b_op->address->BI);
        break;
      default:
        error("incorrect address!");
      };
    }

    void
    alu_conflict(ac_op, b_op)
      struct operand *ac_op, *b_op;
    {
      fprintf(query, "MICROP alu_op VAR %s[%d], %s[%d];",
              number_to_string(ac_op->number, N_VAR), ac_op->index->offset,
              number_to_string(b_op->number, N_VAR), b_op->index->offset);
    }

    void
    backend()
    {
      FILE * query;
      query = popen("mdl -a processor",w);

    /*  processor.mdl contains the MDL-description, with option -a the
        analyzer reads the input for the top-down-analysis from stdin,
        which is connected to the pipe.
        Error messages are sent directly to stdout.
    */

    }
      encode("fac.art", 0);
      encode("fac.dl", ac_op->address->DL);
      encode("fac.d", ac_op->address->D);
      encode("fac.w", ac_op->address->W);
      encode("fac.b", ac_op->address->B);
      break;
    case 2:
      encode("fac.art", 1);
      encode("fac.bl", ac_op->address->BL);
      encode("fac.r", ac_op->address->R);
      encode("fac.w", ac_op->address->W);
      encode("fac.b", ac_op->address->B);
      break;
    case 3:
      encode("fac.art", 2);
      encode("fac.bl", ac_op->address->BL);
      encode("fac.r", ac_op->address->R);
      encode("fac.wi", ac_op->address->WI);
      encode("fac.x", ac_op->address->X);
      encode("fac.b", ac_op->address->B);
      break;
    case 4:
      encode("fac.art", 3);
      encode("fac.bl", ac_op->address->BL);
      encode("fac.r", ac_op->address->R);
      encode("fac.dw", ac_op->address->DW);
      encode("fac.x", 0);
      encode("fac.bi", ac_op->address->BI);
      break;
    case 5:
      encode("fac.art", 3);
      encode("fac.a", ac_op->address->A);
      encode("fac.r", ac_op->address->R);
      encode("fac.dw", ac_op->address->DW);
      encode("fac.x", 1);
      encode("fac.bi", ac_op->address->BI);
      break;
    default:
      error("incorrect address!");
    };
    switch (b_op->address->art)
    {
    case 1:
      encode("fb.art", 0);
      encode("fb.dl", b_op->address->DL);
      encode("fb.d", b_op->address->D);
      encode("fb.w", b_op->address->W);
      encode("fb.b", b_op->address->B);
      break;
    case 2:
      encode("fb.art", 1);
      encode("fb.bl", b_op->address->BL);
      encode("fb.r", b_op->address->R);
      encode("fb.w", b_op->address->W);
      encode("fb.b", b_op->address->B);
```

Beispiel B.2 (Fortsetzung): Backend Programmfragment `processor.c`.

Abbildungsverzeichnis

1.1 Der ursprüngliche Entwicklungszyklus 2
1.2 Der Entwicklungszyklus nach Einführung des Hardwaresimulators 4
1.3 Der bisherige Aufbau des Firmwarecompilers 6
1.4 Der Entwicklungszyklus mit MDL 9
1.5 Der Entwicklungszyklus mit erweiterter MDL und Compilergenerator . . 17
1.6 Der Entwicklungszyklus mit teilweise generierter MDL 19

2.1 Die IR als stabile Basis 25
2.2 Der Knotentyp `[List]` 28
2.3 Syntaxdiagramm der Regel `syntax` 32
2.4 Syntaxdiagramm der Regel `rule` 33
2.5 Syntaxdiagramm der Regel `formulation` 33
2.6 Syntaxdiagramm der Regel `gram_line` 34
2.7 Der Gesamtaufbau des generierten Firmwarecompilers 47

3.1 Der Hardware/Software-Entwicklungsprozeß 51
3.2 Die Software steht im Vordergrund 51
3.3 Die Hardware steht im Vordergrund 52
3.4 Das Y-Chart Diagramm 58

Tabellenverzeichnis

A.1 Die neuen MDL/BED–Schlüsselwörter 63

Beispielverzeichnis

1.1 Beispiel für eine MDL-Beschreibung 12
1.1 Fortsetzung 13
2.1 Die Referenz des gesamten IR-Baumes 37
2.2 Die Typdefinition der Unterbäume 39
2.3 Die Definition der Unterbäume `Operand` und `Simple_Operand` 40
2.4 Konfliktprüfung mittels Top-Down-Analyse 42
2.5 Codierung über die Top-Down-Analyse 44
B.1 Die Backendbeschreibung `processor.mdl` 67
B.1 Fortsetzung 68
B.1 Fortsetzung 69
B.1 Fortsetzung 70
B.1 Fortsetzung 71
B.1 Fortsetzung 72
B.2 Backend Programmfragment `processor.c` 74
B.2 Fortsetzung 75
B.2 Fortsetzung 76

Literaturverzeichnis

[1] Karl M. Göschka: *Frontend für Mikado.* Praktikum an der Technischen Universität Wien, Institut für Computersprachen **1995**

[2] Georg Gottlob u.a.: *Datenbanksysteme.* Skriptum zur gleichnamigen Vorlesung, Technische Universität Wien, Institut für Informationssysteme, Abteilung für Datenbanken und Expertensysteme, **1994**

[3] Stephen L. Diamond, *Architecture Neutral Distribution Format.* IEEE Micro pp. 73 – 76, Dec **1994**

[4] Karl M. Göschka: *Microcompiler Design Language.* Diplomarbeit an der Technischen Universität Wien, Institut für Computertechnik **1993**

[5] T. Staudinger: *SAB 88C166: Flash-EPROM-Controller setzt neue Maßstäbe.* Siemens Components März/93, 95–97 **1993**

[6] Lauther, Michel *The Synthesis Approach to Digital System Design.* Kluwer **1992**

[7] K. Diefendorff, M. Allen *Organization of the Motorola 88110 Superscalar RISC Microprocessor.* IEEE Micro **April 1992**

[8] *Alpha Architecture Handbook.* Digital Equipment Corporation **1992**

[9] Handbuch: *Mikado: MPL–Y Sprachbeschreibung.* Siemens **1991**

[10] B. J. Catanzaro *The SPARC Technical Papers.* Springer Verlag **1991**

[11] C. W. Fraser/R. R. Henry/T. A. Proebsting: *BURG — Fast Optimal Instruction Selection and Tree Parsing.* `ftp: kaese.cs.wisc.edu in pub/burg.shar.Z` **1991**

[12] *R4000 User's Manual.* MIPS Computer Systems, Inc. **1991**

[13] M. Sonntag: *Scannergenerator SG.* Siemens AG **1990**

[14] D. A. Patterson/J. L. Hennessy: *Computer architecture: a quantitative approach.* Morgan Kaufmann Publishers, Inc. **1990**

[15] *MC88100 RISC Microprocessor User's Manual.* Motorola, Inc. **1990**

[16] H. Emmelmann/F. W. Schröer/R. Landwehr: *BEG — a Generator for Efficient Back Ends.* ACM SIGPLAN **1989**

[17] W. Wölfel: *Programmgenerator PG – Benutzerhandbuch und Beispiele.* Siemens AG **1988**

[18] Manual: *IEEE Standard VHDL Language Reference Manual.* IEEE **1988**

[19] J. F. Nixon a. o.: *A microarchitecture description language for retargeting firmware tools.* MICRO–19 **1986**

[20] A. T. Schreiner/G. Friedman: *Compiler bauen mit UNIX – Eine Einführung.* Carl Hanser Verlag München Wien **1985**

[21] M. Banahan/A. Rutter: *UNIX – lernen, verstehen, anwenden.* Carl Hanser Verlag München Wien **1984**

[22] M. J. Eager: *M29 – An Advanced Retargetable Microcode Assembler.* MICRO–16 **1983**

[23] B. W. Kerninghan/D. M. Ritchie: *Programmieren in C.* Carl Hanser Verlag München Wien **1983**

[24] Lutz: *Sprachbeschreibung: Problemsprache für Mikrobefehlsnotation.* Siemens **1981**

[25] A. van Dam a. o.: *Simulation of a Horizontal Bit–Sliced Processor Using the ISPS Architecture Simulation Facility.* IEEE Transactions on Computers C–30 **1981**

[26] Strutynski/Haff: *Funktionsbeschreibung und Mikrobefehlsliste P1.* Siemens DFY 3.3.2.3.27 **1979**

[27] Baumgartner: *Mikroprogrammsprache des Y–Prozessors P1.* Siemens DFY 3.3.2.3.25 **1979**

[28] M. R. Barbacci a. o.: *The ISPS Computer Description Language.* Technical Report, Department of Computer Science, Carnegie–Mellon University **1977**