Diplomarbeit

XML-basierte generische Zwischendarstellung für Compiler

Markus Fiesel

Diplomarbeit
am Lehrstuhl XII
des Fachbereichs Informatik
der Universität Dortmund

20. Dezember 2001

Betreuer:

Dipl.–Inform. Markus Lorenz
Prof. Dr. Peter Marwedel

Inhaltsverzeichnis

Kapitel 1

Einleitung

Digitalverstärker, PDAs und Handys sind nur einige Beispiele für mikroelektronische Systeme, die uns in immer stärkererem Umfang im Alltag begleiten. Eine wichtige und stark wachsende Gruppe bilden dabei die *eingebetteten Systeme*, die in vielen Bereichen, u.a. der Telekommunikation und der Steuerungstechnik, eingesetzt werden. Um die hohen Entwicklungskosten für spezielle ASICs (*Application Specific Integrated Circuits*) zu vermeiden, werden dafür vielfach programmierbare *eingebettete Prozessoren* eingesetzt. Die Entwicklung der anwendungsspezifischen Software ist schneller und damit preiswerter, hinzu kommt eine Upgrade-Möglichkeit für bereits eingesetzte Systeme. Ein weiterer Vorteil ist, dass diese Prozessoren für viele verschiedene Systeme verwendet werden können, so dass daraus resultierende höhere Stückzahlen Preisvorteile ergeben. Bei der Softwareentwicklung ist der Einsatz von Hochsprachen wünschenswert. Dabei muss allerdings gewährleistet sein, dass im Vergleich zu einer direkten Assemblerprogrammierung *gleichwertige* Maschinenprogramme erzeugt werden. Gleichwertig bezieht sich dabei auf die Korrektheit des Zielprogramms und auf die Effizienz. Diese kann durch unterschiedliche und zumeist konkurrierende Ziele beschrieben sein: Allgemein wird ein besonders schneller Code angestrebt. Bei Programmen, die auf akkubetriebenen eingebetteten Systemen laufen, ist ein kompakter oder energiesparender Code ein zusätzliches wichtiges Ziel.

Es ist die Aufgabe eines Compilers, die Hochsprache in ein gleichwertiges Zielprogramm zu überführen (→Abb. 1.1). Um die oben beschriebenen Ziele zu erreichen muss dabei die Zielarchitektur effizient genutzt werden. Dies ist aber gerade mit herkömmlichen Compilern für eingebettete Prozessoren nicht möglich, da diese häufig stark irreguläre Architekturen haben. So gibt es in DSPs (*Digital Signal Processor*), die eine wichtige Gruppe eingebetteter Prozessoren bilden, z.B. *komplexe Operationen*, wie die MAC-Operation (*M*ultiply-*AC*cumulate), die in einem Taktzyklus eine Multiplikation mit anschließender Aufaddierung eines Wertes ermöglicht. Darüber hinaus erlauben sie eine parallele Ausführung von Instruktionen, deren Nutzung allerdings durch architekturbedingte Ressourcenkonflikte eingeschränkt sein kann. Um einen effizienten Code erzeugen zu können müssen diese Besonderheiten der Architektur berücksichtigt werden.

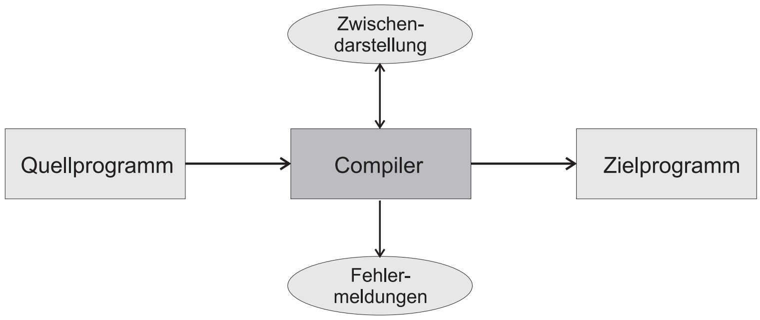

Abb. 1.1: Compiler überführen ein Quellprogramm in ein gleichwertiges Zielprogramm.

Die Entwicklung neuer Compiler ist allerdings ein bedeutender Kostenfaktor, gerade wenn diese speziell auf eine Architektur zugeschnitten sind, für die nur wenige Applikationen entwickelt werden. Daher ist es wünschenswert, ein bestehendes Compilersystem auf einfache Art und Weise auf eine neue Zielarchitektur umstellen zu können (*retargierbarer Compiler*). Viele bislang entwickelte Compiler verfügen zwar über eine standardisierte interne High-Level Zwischendarstellung (*Intermediate Representation*, IR), auf der Optimierungen des Programmcodes durchgeführt werden können, diese sind allerdings architekturunabhängig. Eine entsprechende Zwischendarstellung, die eine Beschreibung der Zielarchitektur beinhaltet, eine sogenannte LIR (*Low-Level Intermediate Representation*), wird dagegen häufig für jede Architektur neu entwickelt. Diese LIR dient als Austauschformat für architekturspezifische Optimierungen und für die Aufgaben der Codegenerierung, der eigentlichen Erzeugung eines gültigen Zielprogramms. Eine LIR, die eine flexible Beschreibung der Zielarchitektur ermöglicht und für die generische Optimierungen implementiert sind, die auf die verschiedenen Architekturen angewendet werden können, vereinfacht die Entwicklung neuer Compiler und Codegeneratoren.

Eine am Lehrstuhl XII des Fachbereichs Informatik an der Universität Dortmund entwickelte Low-Level Zwischendarstellung ist GeLIR (*Generic Low-Level Intermediate Representation*), in der spezielle Eigenschaften von DSP-Architekturen, wie komplexe Operationen und Parallelität, beschrieben werden können, die von generischen Optimierungen genutzt werden.

1.1 Ziele dieser Diplomarbeit

Bei der Entwicklung neuer Optimierungen ist es wichtig, Zwischenzustände einer (L)IR auf einfache Art reproduzieren zu können, um gleiche Voraussetzungen für sich anschließende Testläufe zu haben. Diese Reproduktion ist einerseits zeitaufwendig, da vorhergehende Schritte erneut durchlaufen werden müssen, und andererseits ist es vielfach sogar unmöglich, exakt den gleichen Zustand wiederherzustellen. Arbeiten mehrere Anwender mit GeLIR, so ist es bei Problemen nützlich, anderen Nutzern einen Zwischenzustand für die Analyse bereitzustellen. Außerdem ist es von großem Nutzen, einen lesbaren, manipulierbaren "Schnappschuss" eines Zwischenzustandes zu haben, um bestimmte Optimierungsprozesse nachvollziehen oder Eigenschaften ändern zu können.

Um die genannten Aspekte zu ermöglichen bietet sich ein textuelles Speicherungs- und Austauschformat an. Die Anforderungen an ein solches Speicherformat sind eine einfache übersichtliche Struktur, die das Lesen des Dokuments erleichtert, und ein möglichst einfaches Parsen eines Dokuments, um einen Zustand wiederherstellen zu können. Dafür ist es sinnvoll, auf ein standardisiertes Textformat zurückzugreifen, bei dem auf bereits implementierte Parser zurückgegriffen werden kann und bei dem auch weitere Applikationen, wie z.B. Editoren, entwickelt wurden.

XML (*eXtensible Markup Language*) stellt Lösungen für diese Anforderungen bereit. XML gehört zu den Auszeichnungssprachen (*Markup-Languages*), die entwickelt wurden um *Daten* und *Metadaten* gemeinsam in einem Textdokument unterzubringen. Es ist ein von SGML (*Standard Generalized Markup Language*) abgeleiteter Standard, der die Möglichkeit bietet, eine eigene Menge von Metadaten (ein *Vokabular*) zu definieren und auch später noch zu erweitern. XML ist vom W3C (*World Wide Web Consortium*) im Jahre 1998 als Empfehlung verabschiedet worden [XMLe].

Im Umfeld von XML gibt es weitere Empfehlungen des W3C, die die Arbeit mit XML-basierten Dokumenten unterstützen. Dazu gehört neben einer standardisierten API (*Application Programming Interface*) namens *DOM* u.a. auch die Möglichkeit, mit *Schemata* die Eigenschaften eines Dokuments, wie z.B. das Vokabular, genau zu beschreiben und entsprechende Dokumente gegenüber diesen Regeln zu validieren. Darüber hinaus entstehen fast täglich neue Anwendungen, angefangen mit einfachen Viewern über komplexe Editoren bis hin zu aufwendigen Validierungstools.

XML-Dokumente sind streng hierarchisch aufgebaut. Diese Struktur begünstigt die Suche nach bestimmten Mustern innerhalb des Dokuments um bestimmte Informationen zu erhalten und auszuwerten.

Ausgehend von den dargestellten Anforderungen an eine textbasierte Zwischendarstellung und den zusätzlichen Möglichkeiten, die sich durch die Verwendung von XML ergeben, leiten sich für diese Diplomarbeit die folgenden Ziele ab:

1. die Entwicklung einer XML-basierten Zwischendarstellung (*XeLIR*) als direktes Austauschformat für GeLIR,

2. die Untersuchung der Validierungsmöglichkeiten für ein XeLIR-Dokument mittels XML-Schema,

3. die Untersuchung von weiteren Möglichkeiten der Nutzung einer XML-basierten Zwischendarstellung, insbesondere durch Patternmatching.

Dabei stehen beim Patternmatching die folgenden Aspekte im Vordergrund:

- die Erzeugung von *Assemblercode* aus der textbasierten Darstellung durch die Beschreibung von Ausgaberegeln für einzelne Programmkomponenten und
- die Durchführung bzw. Unterstützung von Peepholeoptimierungen.

Die Schnittstellen der XML-basierten Zwischendarstellung XeLIR (*XML-based gEneric Low-Level Intermediate Representation*) sind in Abb. 1.2 noch einmal dargestellt.

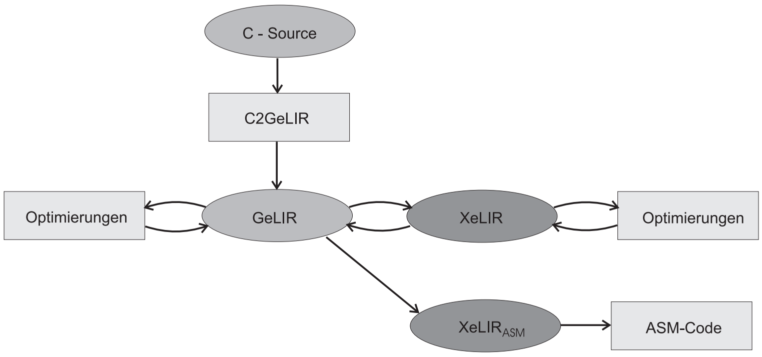

Abb. 1.2: Schnittstellen von XeLIR

1.2 Übersicht über die Kapitel

- Kapitel 2 gibt eine Übersicht über die Teilgebiete des Compilerbaus und eine Einordnung des Begriffes Zwischendarstellung in den Compilierungsprozess.
- Kapitel 3 stellt die am Lehrstuhl XII entwickelte Zwischendarstellung GeLIR vor und zeigt deren Konzepte der Architekturbeschreibung und Programmdarstellung.
- Kapitel 4 gibt eine Einführung in XML und stellt verschiedene Standards vor, die im Umfeld von XML entwickelt wurden. Darüber hinaus werden die Anforderungen an die in dieser Diplomarbeit eingesetzte Software diskutiert.
- Kapitel 5 beschreibt den Aufbau von XeLIR und stellt die in einem XML-Schema definierte Dokumentstruktur und Datentyp-Einschränkungen für XeLIR-Komponenten vor.
- Kapitel 6 behandelt Anwendungsmöglichkeiten von XeLIR. Dazu gehört neben dem Austauschformat für GeLIR auch die eigenständige Generierung von Assemblercode aus einem XeLIR-Dokument und die eigenständige oder kooperative Möglichkeit für Peepholeoptimierungen.
- Kapitel 7 fasst die Ergebnisse der Arbeit nochmals kurz zusammen und gibt einen Ausblick auf weiteren Forschungs- und Entwicklungsbedarf.

Kapitel 2

Grundlagen des Compilerbaus

Dieses Kapitel gibt einen Überblick über die Aufgaben und den Aufbau eines Compilers. Dabei werden zunächst die Phasen, die ein Compiler durchläuft, und deren Zusammenhänge beschrieben und anschließend im Detail besprochen. Für einzelne Aufgaben des Compilerbaus werden Lösungsansätze skizziert, zudem wird ein Überblick über bestehende Compilersysteme gegeben.

2.1 Aufgaben und Aufbau von Compilern

Die Aufgabe von Compilern ist es, ein Quellprogramm in ein äquivalentes Zielprogramm zu überführen. Das Quellprogramm kann eine Hochsprache wie C++, aber auch eine Spezialsprache oder Maschinensprache sein. Genauso groß ist das Spektrum für die Zielsprache. Diese kann neben Assembler- und Maschinenprogrammen auch eine andere Hochsprache sein, für die schon Compiler existieren, um diese in ein Assemblerprogramm zu übersetzen.

Die ersten Compiler wurden parallel und unabhängig voneinander Mitte der 50er Jahre entwickelt. Dabei stellte sich heraus, dass alle Compiler ein bestimmtes Grundmuster besitzen. Die Phasen eines Compilers lassen sich in ein *Frontend* (Analysephase) und ein *Backend* (Synthesephase) aufteilen (vgl. Abb. 2.1). Vielfach wird auch noch das *Middleend* als von dem Frontend unabhängiger Teil unterschieden. Die Aufgabe des Frontends ist es, das Quellprogramm syntaktischen und semantischen Analysen zu unterziehen. Anschließend wird das Programm in eine interne Zwischendarstellung (siehe auch Kap. 2.5) des Middleends überführt. Bis zu diesem Zeitpunkt ist die Betrachtung unabhängig von der Zielarchitektur. Die zentrale Aufgabe des Backends ist die Codegenerierung, d.h. die Überführung des Programms in ein auf einer Zielmaschine ausführbares Maschinenprogramm oder in einen gültigen Assemblercode. Diese Phase hängt nicht mehr von dem Quellprogramm ab, sondern von der Zwischendarstellung, und orientiert sich an der Architektur der Zielmaschine. Die Zweiteilung des Compileraufbaus hat den Vorteil, dass bei Änderung der Hochsprache oder der Zielarchitektur nicht komplett neue Compiler entwickelt werden müssen. So können

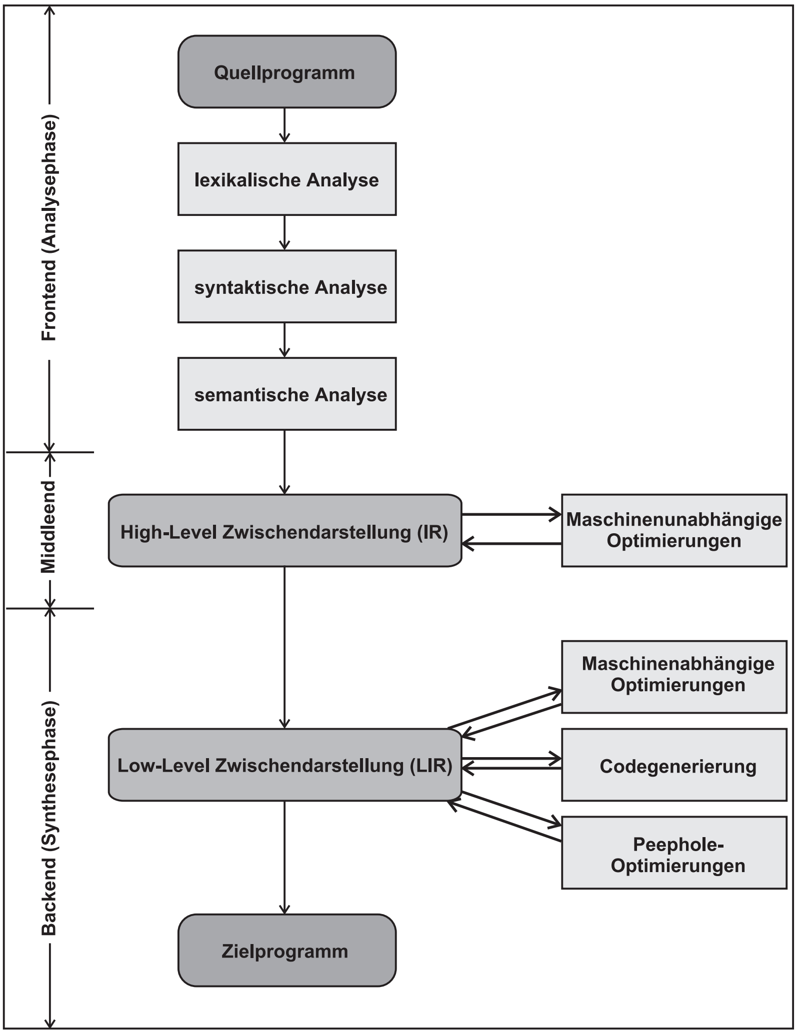

Abb. 2.1: Überblick über die Compilierungsphasen

verschiedene Frontends für verschiedene Hochsprachen entwickelt werden, die in dieselbe Zwischendarstellung münden, aber alle dasselbe Backend benutzen. Umgekehrt muss für eine neue Zielarchitektur nur das Backend angepasst werden, ohne das Frontend zu ändern.

Neben der Erzeugung eines *korrekten* Zielprogramms ist es von Interesse, einen *effizienten* Code zu erzeugen. Die Akzeptanz der ersten Hochsprache Fortran war stark davon abhängig, dass ein ähnlich effizienter Code erzeugt werden konnte wie durch direkte Assemblerprogrammierung (vgl. [ASU87]). Die Effizienz kann durch unterschiedliche Kriterien beschrieben sein. So ist im Allgemeinen das Ziel, einen möglichst kompakten, schnellen und energiesparenden Code zu erzeugen, letzteres ist gerade für akkubetriebene, eingebettete Systeme von Bedeutung. Die einzelnen Ziele konkurrieren häufig miteinander, so dass in der Praxis Kompromisse gefunden werden müssen, die häufig von der Anwendung des Zielprogramms bestimmt sind.

Diese durch *Optimierungen* erreichten Codeverbesserungen werden auf der Zwischendarstellung ausgeführt, die durch eine Struktur gekennzeichnet ist, die die Entwicklung und die Durchführung von Optimierungen vereinfacht. Während die Zwischendarstellung im Middleend noch architekturunabhängig (*abstrakt*) ist und dementsprechend nur sogenannte High-Level Optimierungen durchgeführt werden können, erfolgt anschließend die Überführung in eine Low-Level Zwischendarstellung (LIR), die eine Architekturbeschreibung der Zielmaschine beinhaltet.

Auf dieser LIR werden zusätzlich architekturspezifische Optimierungen möglich, die spezielle Hardware wie z.B. MAC-Operationen oder mögliche parallele Ausführungen ausnutzen können. Außerdem ist es vorteilhaft, die Codegenerierung (siehe Kap.2.4.1) auf der Zwischendarstellung auszuführen, da generische Optimierungen und Techniken für verschiedene Zielarchitekturen wieder verwendet werden können.

Als letzter Schritt der Compilierung können auf einem generierten Assembler- oder Maschinencode noch Peepholeoptimierungen ausgeführt werden, die nur einen kleinen Ausschnitt des Codes betrachten und verändern (siehe Kap.2.4.2).

Die folgenden Abschnitte geben einen genaueren Überblick über die Aufgaben und den Aufbau der einzelnen Komponenten der Compilierung.

2.2 Frontend

Im Frontend durchläuft das Quellprogramm verschiedene Analysen, die im Folgendem genauer betrachtet werden:

Lexikalische Analyse

Beim ersten Schritt wird das Quellprogramm von einem Scanner linear eingelesen und in lexikalische Gruppen eingeteilt (sogenannte Tokens). Dies sind u.a. reservierte Wörter (in C z.B. *for* oder *while*), Kommentare, Sonderzeichen (z.B. {,}, *, +) oder Bezeichner. In dem Ausdruck

```
x = 5 + y * 10;
```

eines C-Programms werden die folgenden Tokens erkannt: Der Bezeichner x, das Zeichen =, die Konstante *5*, das Zeichen +, der Bezeichner y, das Zeichen *, die Konstante *10* und das Sonderzeichen *;*.

Syntaktische Analyse

In einem nächsten Schritt wird von einem sogenannten Parser überprüft, ob die erkannten Tokens einer bekannten kontextfreien Grammatik folgen, die für jede Programmiersprache definiert ist. Der Parser versucht Tokens zu Sätzen zusammenzufassen und baut einen Syntaxbaum auf. Dieser enthält alle Bezeichner, Konstanten und Sonderzeichen. Aus diesem kann ein abstrakter Syntakbaum konstruiert werden, der nur noch die wesentlichen Elemente wie Bezeichner, Zeichen und Konstanten enthält. Aus der obigen Programmzeile wird der in Abb.2.2 gezeigte Syntaxbaum aufgebaut.

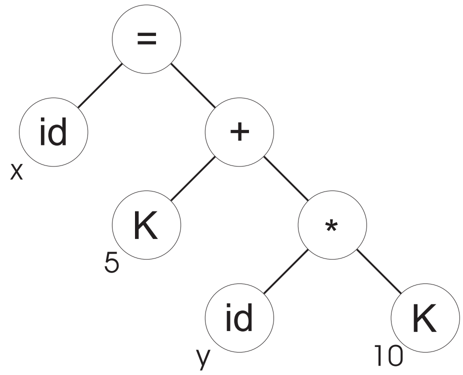

Abb. 2.2: Abstrakter Syntaxbaum

Semantische Analyse

Viele Fehler können von der syntaktischen Analyse nicht gefunden werden. Deshalb wird noch eine semantische Analysephase durchgeführt, die u.a. doppelte Deklarationen entdeckt wie

```
int i;
int i;
```

Für die Analyse werden Informationen aus dem bereits aufgebauten Syntaxbaum gesammelt und ausgewertet. Zudem werden werden in dieser Phase Typüberprüfungen durchgeführt und Typumwandlungen, so weit es möglich ist, automatisch eingefügt.

2.3 Middleend

Nach den Analysen des Frontends wird der Code in eine abstrakte Zwischendarstellung des Middleends überführt, auf der High-Level Optimierungen ausgeführt werden können. Die Aktionen hängen nicht mehr vom Quellprogramm, sondern von der Zwischendarstellung ab. Die gemeinsame Nutzung einer IR vereinfacht die Entwicklung neuer oder die Erweiterung bestehender Optimierungen. Beispiele für Optimierungsansätze sind:

- **Constant Folding:** Sind alle Operanden einer Anweisung Konstanten, so wird im weiteren Programm die Benutzung des Ergebnisses direkt durch eine Konstante ersetzt.
- **Copy Propagation:** Zuweisungen zwischen Variablen werden durch die Benutzung der Originalvariable ersetzt, soweit dies möglich ist.
- **Dead Code Elimination:** Variablen oder Instruktionen, deren Werte bzw. Ergebnisse nicht benutzt werden, werden entfernt.
- **Inlining:** Funktionsaufrufe werden direkt durch den Funktionsbody ersetzt, was die Bearbeitung der Parameterübergabe und -rückgabe überflüssig macht.

Häufig konterkarieren Optimierungen die bereits durch andere Optimierungen erzielten Verbesserungen, oder es werden neue Optimierungsmöglichkeiten geschaffen, die von bereits ausgeführten Optimierungen genutzt werden können. Eine mehrfache und abwechselnde Ausführung mehrerer Optimierungen ist deshalb sehr sinnvoll, bedarf allerdings einer genauen Planung, um optimale Ergebnisse zu erzielen. Optimierungen des Middleends werden häufig auch zusätzlich in der LIR des Backends ausgeführt. Eine ausführliche Übersicht über Optimierungsansätze und deren Ablaufplanung gibt [Muc97].

2.4 Backend

Im Backend (Synthesephase) verfügt die Zwischendarstellung im Gegensatz zum Middleend zusätzlich über Informationen über die Zielarchitektur, an der sich die Aktionen orientieren. Die wesentlichen Aufgaben des Backends sind abstrakte und architekturspezifische Optimierungen sowie die Codegenerierung. Darüber hinaus sind auch die bereits oben erwähnten Peepholeoptimierungen Teil dieser Phase. Diese Teilgebiete werden in den folgenden Abschnitten genauer betrachtet.

2.4.1 Codegenerierung

Die Aufgabe der Codegenerierung ist es, einen korrekten, ausführbaren Code zu erzeugen. Außerdem sollen die Ressourcen der Zielarchitektur effizient genutzt werden. Einige Probleme dieser Phase sind NP-vollständig, so dass die effiziente Durchführung der Codegenerierung eine weitere Anforderung ist. Die Aufgaben gliedern sich in in die drei Bereiche Instruktionsauswahl, Registerallokation und Instruktionsanordnung, die im Folgenden skizziert werden. Häufig sind auch die Adresscodegenerierung und Adresszuweisung Teil dieser Phase. Einen Überblick über bestehende Ansätze für die einzelnen Bereiche findet sich in [Bas95].

- **Instruktionsauswahl:** Die Instruktionsauswahl (Code Selection) hat die Aufgabe, die abstrakten Operationen in der Zwischendarstellung möglichst kostenoptimal mit

Instruktionen der Zielarchitektur zu überdecken. Ein einfaches Verfahren wäre es, für jede abstrakte Operation ein Muster zu erzeugen, das eine passende Instruktion der Zielarchitektur enthält, und die komplette Programmdarstellung mit Hilfe dieser Muster abzubilden. Dies ist allerdings ineffizient, da dabei häufig redundanter Code erzeugt wird. Die Schwierigkeit der Code Selection ist stark vom Aufbau der Zielarchitektur abhängig. Für CISC-Architekturen gibt es häufig mehrere Möglichkeiten der Überdeckung. Je irregulärer die Architektur, desto mehr Spezialbehandlungen sind für die Befehlsauswahl nötig., Z.B. lässt sich eine Addition mit 1 auf vielen Maschinen mit einem speziellen Inkrementbefehl (INC) ausführen, eine einfache Musterüberdeckung der Addition resultierte jedoch in ineffizienteren Code.

Der meist verbreitete Ansatz ist die Baummustererkennung, die in [WM97] beschrieben ist. Dabei wird der Graph an sog. *Common-subexpressions* in Bäume aufgesplittet. Für jeden Teilbaum können in linearer Zeit optimale Lösungen gefunden werden, allerdings kann bei diesem Verfahren keine Parallelität berücksichtigt werden. Dies ist bei graphbasierten Verfahren möglich, die allerdings NP-vollständig sind.

- **Registerallokation:** Zugriffe auf Register sind schneller und energieeffizienter als Speicherzugriffe und der erforderliche Code ist i.d.R. kompakter. Daher ist es anzustreben möglichst viele Werte in Registern zu halten. Insbesondere wenn häufiger auf dieselbe Variable zugegriffen wird, wie z.B. auf Laufvariablen in Schleifen, schafft die Speicherung dieser Werte in Registern einen effizienteren Code. Zunächst wird auf virtuellen Registern gearbeitet, die in dieser Phase durch reale Register ersetzt werden. Einerseits muss entschieden werden, welche Werte in Registern gehalten werden (*Registervergabe*), andererseits müssen die ausgewählten Werte an bestimmte Register gebunden werden (*Registerbindung*). Eine wesentliche Analyse besteht in der Bestimmung der Lebensspannen der einzelnen Werte, die durch DU-Wege (define-use) (siehe auch Instruktionsanordnung) im Datenfluss festgelegt sind. Ein Wert muss nur im Register gehalten werden, falls dieser noch benutzt werden wird, ansonsten kann das Register für einen anderen Wert freigegeben werden. Die existierenden Analysen betrachten unterschiedliche Programmausschnitte, von der Basisblockebene über eine sogenannte globale Betrachtung auf der Funktionsebene bis hin zu einer interprozeduralen Sichtweise des Datenflusses. Die Analysen erzeugen zwar bei der Betrachtung größerer Ausschnitte einen besseren Code, werden gleichzeitig aber auch bedeutend aufwändiger.

  Ein systematischer Ansatz für die Registerallokation ist die so genannte *Graphfärbung*. Hierbei wird jeder Wert durch einen Knoten dargestellt und die Lebensspannen der Werte werden untersucht. Überschneiden sich zwei Lebensspannen, so werden die zugehörigen Knoten verbunden und es entsteht ein *Registerkonfliktgraph*. Es wird nun versucht, diesen Graphen mit k Farben (für k physikalische Register) einzufärben, so dass keine benachbarten Knoten dieselbe Farbe haben. Dieses Problem ist für $k>2$ NP-vollständig. Sollte es nicht möglich sein, den Graphen komplett einzufärben, so gibt es einen Registerkonflikt, der durch einen *Registerabwurf* (der Speicherung des Wertes im Hauptspeicher und eventuell späterer Wiederherstellung in einem Register) aufgehoben wird (das so genannte *Spilling*). Die Entscheidung, welches Register abgeworfen wird, damit möglichst wenig Spillcode produziert wird, ist ein weiteres Problem der Registerallokation. Eine ausführliche Beschreibung erfolgt in [Muc97]. Eine Ansatz für eine optimale Registerbindung in eindimensionalen Schleifen mittels linearer Programmierung wird in [Tei97] gezeigt.

- **Instruktionsanordnung:** Die Veränderung der Instruktionsanordnung (*Instruction Scheduling*) kann sich positiv auf die Codeeffizienz auswirken, weil dadurch z.B. das Abwerfen eines Registers und der damit verbundene Spillcode verhindert werden können. Eine weitere Effizienzverbesserung kann durch eine bessere Ausnutzung der Hardware entstehen, z.B. bei VLIW- oder Pipeline-Architekturen.

  Große Beachtung muss dabei der Bewahrung der Korrektheit des Programmes geschenkt werden. Zwischen den folgenden Befehlen bestehen unterschiedliche Abhängigkeiten, die ein Vertauschen verhindern (nach [WM97]):

```
(a)  X = a + b
(b)  X = c + ?
(c)  y = X + d
(d)  X = e + f
```

  Die Abhängigkeiten werden danach unterschieden, ob ein Wert gesetzt (define) oder benutzt (use) wird. Das Setzen eines Wertes kann z.B. das Ablegen eines Ergebnisses in einem Register, ein Autoinkrement oder das Setzen/Löschen eines Flags sein. Beispiele für eine Benutzung sind die Verwendung eines Registers in Operationen oder das Abfragen von Flags in einem Sprungbefehl. Je nachdem, welche Folge von Setzen/Benutzen auftritt, unterscheidet man zwischen folgenden Abhängigkeiten:

  - Eine Folge von **definition-definition (dd)** heißt *Ausgabeabhängigkeit (output dependence)*. Diese besteht im Beispiel zwischen (a) und (b). Die Berechnung in (a) scheint überflüssig zu sein, da das Ergebnis in (b) direkt neu berechnet wird. (a) ist allerdings nur dann überflüssig, wenn das dort errechnete Ergebnis nicht in (b) verwendet wird (im Beispiel ausgedrückt durch "?")
  - Eine Folge von **definition-use (du)** nennt sich *Datenabhängigkeit (true dependence)* und ist im Beispiel zwischen (b) und (c) gegeben. In (b) wird ein Wert definiert und in (c) verwendet.
  - Eine Folge von **use-definition (ud)** heißt *Antiabhängigkeit (anti dependence)*. Im Beispiel gilt dies für die Zeilen (c) und (d). Im Falle einer Vertauschung der beiden Instruktionen würde bei der Benutzung der Variablen "X" ein falscher Wert vorliegen, nämlich der, der ursprünglich erst in (d) gesetzt werden sollte.

  Schwierigkeiten bei der Feststellung dieser Abhängigkeiten können sich bei dynamischer Adressierung ergeben. Zwei Befehle können z.B. auf dieselbe Speicherzelle zugreifen, ohne dass dieses erkannt wird. Ein einfacher Lösungsansatz wäre, den gesamten Hauptspeicher als ein Objekt für die Datenflussanalyse zu betrachten.

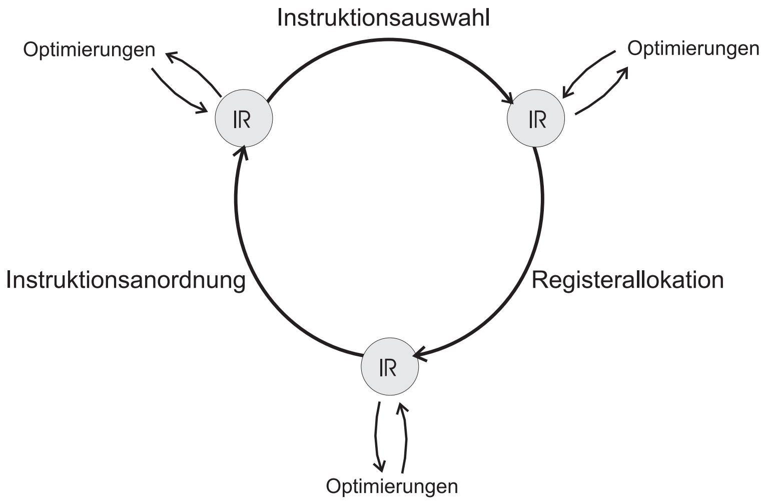

Abb. 2.3: Phasenkopplung in der Codegenerierung

Instruktionskompaktierung ist ein Unterbereich der Instruktionsanordnung, deren Aufgabe es ist, mehrere parallel ausführbare Maschinenoperationen zu einer Maschineninstruktion zusammenzufassen.

Eine ausführliche Behandlung des Themas Instruktionsanordnung findet sich in [WM97] oder auch in [Muc97].

Die vorgestellten Teilaufgaben der Codegenerierung wurden zwar getrennt betrachtet, von einer gemeinsamen Behandlung der Teilaufgaben in abwechselnden Phasen, wie in Abb.2.3 dargestellt, können die Bereiche jedoch wechselseitig profitieren. Die Anordnung unterliegt allerdings bestimmten Einschränkungen, da durch die Instruktionsauswahl z.B. Register gebunden werden, was dann in der Registerallokation berücksichtigt werden muss. Ein Überblick über Ansätze der *Phasenkopplung* (Phase coupling) findet sich in [Bas95].

2.4.2 Peepholeoptimierungen

Bei Peepholeoptimierungen wird nur ein kleiner Aussschnitt des Programms betrachtet um lokal begrenzte Codeverbesserungen durchzuführen. Diese können auch auf einem bereits geschriebenen Assemblercode ausgeführt werden. Ansätze für diese Optimierungen sind u.a.:

- **Überflüssiges Laden/Speichern:** Die folgende Befehlsfolge zeigt einen überflüssigen Datentransfer:

```
MOV R0, a
MOV a, R0
```

Der zweite Befehl kann gelöscht werden, falls sichergestellt werden kann, dass beide Befehle immer direkt hintereinander ausgeführt werden, d.h. dass der zweite Befehl nicht Ziel einer Sprungoperation ist.

- **Unerreichbarer Code:** Dazu gehört z.B. Code, der direkt auf eine unbedingte Sprunganweisung folgt, ohne selbst Ziel eines Sprunges zu sein.
- **Kontrollflussoptimierungen:** Erfolgen im Kontrollfluss zwei Sprünge direkt hintereinander, so kann der erste Sprung direkt zum Ziel des zweiten Sprunges erfolgen.
- **Algebraische Vereinfachungen:** Diese bestehen in dem Auffinden und Eliminieren überflüssiger Operationen, wie z.B.:

  ```
  x:= x + 0
  x:= x * 1
  ```

- **Algebraische Transformationen:** Diese verändern einzelne Operationen um z.B. eine Verringerung der Operationskosten auf der Zielmaschine zu erreichen. Ein Beispiel ist die Wandlung einer Multiplikation mit 2 in eine Shiftleft-Operation.

  ```
  MUL x, 2  ->  SHL x, 1
  ```

- **Ausnutzung spezieller Eigenschaften der Zielmaschine:** Dies kann z.B. die in vielen Architekturen mögliche Nutzung der Autoinkrementfunktion bei Adressierungen sein.

2.5 Zwischendarstellungen

Zwischendarstellungen, die das Austauschformat für Optimierungen und im Backend für die Codegenerierung sind, können in zwei Gruppen eingeteilt werden:

- **Maschinenunabhängige Zwischendarstellungen** (*Intermediate Representations*), die das Programm in abstrakten Maschineninstruktionen beschreiben, die unabhängig von einer Zielarchitektur sind.
- **Maschinenabhängige Zwischendarstellungen** (*Low-Level Intermediate Representations*), bei denen das Programm durch Vorstufen von realen Maschineninstruktionen dargestellt wird.

Die für imperative Sprachen meist verbreitete Form einer Zwischendarstellung ist wohl der **Drei-Adress-Code**, bei dem alle Anweisungen in eine ähnliche Grundform mit einem Ergebnis, einem Operator und zwei Operanden nach dem Muster

```
x := y op z
```

überführt werden. Der Operator kann eine arithmetische oder logische Operation sein. Komplexere arithmetische Ausdrücke werden aufgelöst:

```
x * y + z    --->   1: t1 = x * y
                    2: t2 = z + t1
```

Im Drei-Adress-Code werden verschiedene Standardanweisungen unterstützt, mit deren Hilfe Konstrukte imperativer Sprachen dargestellt werden können:

- Zuweisungen für binäre arithmetische oder logische Operationen: x := y op z
- Zuweisungen für unäre arithmetische oder logische Operationen: x := op y
- Einfache Datentransfers: x := y
- Unbedingte Sprünge: goto l
- Bedingte Sprünge: if x goto l
- Funktionsaufrufe und -rücksprünge: call f; return x
- Indizierte Zuweisungen: x := y[i] (die i-te Speicherzelle hinter y)
- Adressoperationen: x = &y

Diese einfachere Struktur der Programmdarstellung erleichtert die Durchführung von Optimierungen, und ermöglicht eine schnelle Umordnung von Anweisungen. Da der Drei-Adress-Code bereits eine starke Ähnlichkeit zu Assemblerbefehlen hat, ergeben sich auch Vorteile für die Codegenerierung. Die Implementierung einer Drei-Adress-Darstellung wird in [ASU87] besprochen.

Eine weitere Klasse von Zwischendarstellungen bilden **Graphbasierte Zwischendarstellungen**, die bestimmte Strukturen im Programm, wie z.B. Kontroll- und Datenfluss, sowie Datenabhängigkeiten beschreiben, die u.a. in der Codegenerierung wichtig und dieser damit förderlich sind.

Mehrfachzuweisungen (**Multi-Assignments**) an dieselbe Programmvariable sind in fast allen imperativen Sprachen erlaubt. Diese schaffen zusätzliche Datenabhängigkeiten, die eine Änderung der Instruktionsanordnung erschweren (vgl. Kap.2.4.1). Dies ist die Motivation für einen weiteren Ansatz für Zwischendarstellungen, dem **Static-Single-Assignment (SSA)**, bei dem jeder Variable nur genau einmal im gesamten Programm ein Wert zugewiesen wird. Existierende Mehrfachzuweisungen werden durch Hinzufügen weiterer Variablen aufgelöst. Das SSA erfordert an Punkten, an dem Kontrollflusspfade verschmelzen, eventuell spezielle ϕ-Knoten, falls für eine Variable des Quellcodes in verschiedenen Pfaden mehrere Werte zugewiesen werden, die allerdings in der SSA-Darstellung durch verschiedene Variablen repräsentiert werden. Neben einer vereinfachten Umsortierung bietet SSA auch vielfältige Vereinfachungen für Optimierungen.

Viele Compilersysteme mit verschiedenen Zwischendarstellungen sind entwickelt worden, die jeweils unterschiedliche Schwerpunkte setzen. Dazu gehören u.a.:

- **Trimaran:** Das Hauptaugenmerk von Trimaran [Tri] richtet sich auf die Ausnutzung von Parallelität auf Instruktionsebene[1]. Dabei stehen Optimierungen im Backend im Vordergrund. Trimaran erlaubt es mit der Sprache MDES (*Machine Description Language*) Zielarchitekturen zu beschreiben. Diese ist allerdings in erster Linie auf homogene Architekturen ausgerichtet. Für die IR existiert ein ASCII-basiertes für den Benutzer lesbares Pendant, das die Auslagerung und das Wiedereinlesen von Zwischenzuständen erlaubt.

- **SUIF:** SUIF [SUI] ist eine High-Level Zwischendarstellung, die zwei Darstellungen unterstützt. In High-SUIF bleibt die Beschreibung von Hochsprach-Konstrukten wie Schleifen komplett erhalten. Low-SUIF bietet eine Darstellung, bei der diese Konstrukte auf einfachere Strukturen abgebildet werden. Zudem ist es möglich, im Frontend Eigenschaften von Typen wie z.B. Bitbreiten und Little-/Big-Endian für die Zielarchitektur detailliert zu beschreiben und die Darstellung damit für das Backend vorzubereiten.

- **Lance:** Lance [Lan] bietet ein ANSI-C Frontend und eine Bibliothek von maschinenunabhängigen Standard-Optimierungen zur Manipulation einer internen Drei-Address-Code Zwischendarstellung. Aus der IR lassen sich alle Konstrukte auf einfache Weise in C-Syntax herausschreiben, was eine einfache Validierung von Zwischenzuständen erlaubt. Lance bietet eine Schnittstelle zu dem Compiler Generator Generator Olive [FHP00], wobei aus dem Drei-Adress-Code Datenflussbäume generiert werden, für die Olive jeweils eine optimale Überdeckung mit Assemblerinstruktionen findet. Für die Aufgaben des Backends steht eine generische Klassenbibliothek zur Verfügung, die durch Spezialisierungen an die jeweilige Zielarchitektur angepasst wird.

- **CoLIR:** CoLIR [CoL] zeichnet sich durch eine generische Struktur aus, die einfach durchzuführende Anpassungen an neue Prozessoren erlaubt. Sie ermöglicht eine Darstellung alternativer Maschinenprogramme und dient als Basis neuer phasengekoppelter Techniken, bei denen die einzelnen Phasen über *Constraints* Abhängigkeiten an andere Phasen der Codegenerierung weiterreichen können.

- **GeLIR:** [GeL] Die am Lehrstuhl XII entwickelte Low-Level Zwischendarstellung erlaubt die Beschreibung irregulärer Architekturen und konzentriert sich dabei auf spezifische Eigenschaften von DSP-Architekturen. Ein weiteres Ziel ist es generische Optimierungen für verschiedene Architekturen wieder zu verwenden. GeLIR wird in Kap.3 vorgestellt.

[1] Allgemein wird zwischen Parallelität auf Instruktionsebene (Instruction Level Parallelism, ILP) und auf Prozessebene unterschieden. In dieser Diplomarbeit bezieht sich der Begriff Parallelität immer auf die Instruktionsebene

Kapitel 3

Generische Low-Level Zwischendarstellung – GeLIR

GeLIR ist eine Low-Level Zwischendarstellung für Compiler, die am Lehrstuhl XII des Fachbereichs Informatik an der Universität Dortmund entwickelt wurde. Die in C++ programmierte Datenstruktur erlaubt neben der Darstellung des Programms auch die Beschreibung der Zielarchitektur, bestehend aus Operationen und Ressourcen wie Registerfiles und Funktionseinheiten. Weiter bietet GeLIR die Möglichkeit, Parallelität von Maschinenoperationen (MO) auszudrücken und komplexe Maschinenoperationen zu beschreiben, wie z.B. die für DSPs typischen MAC-Operationen (*M*ultiply-*AC*cumulate). GeLIR dient als Austauschformat für abstrakte und architekturspezifische Optimierungen sowie für die Aufgaben der Codegenerierung. Dabei ist es möglich, irreguläre Architekturen zu berücksichtigen.

Während des Prozesses der Optimierung und Codegenerierung wird ein Programm in der Regel zunächst in einer abstrakten (d.h. maschinenunabhängigen) Form abgelegt. Die Überführung in ein äquivalentes Maschinenprogramm geschieht in einem mehrstufigen Prozess, bei dem zunächst für jede abstrakte Operation alle Maschinenoperationen bestimmt werden, die eine gleichwertige Darstellung ermöglichen. Eine abstrakte Operation wird von diesen Zielmaschinenoperationen *überdeckt*. Die Definition und die Argumente einer Maschinenoperation werden dann wiederum mit allen Registerfiles überdeckt, die für die überdeckenden Operationen gültige Kombinationen erlauben. Danach werden diese Ressourcen nach und nach eingeschränkt, bis eine eindeutige, gültige Überdeckung für jede MO gefunden ist (vgl. Kap. 3.2.5).

Zunächst wird jedoch der generelle Aufbau von GeLIR betrachtet. Abb. 3.1 gibt einen Überblick über die in GeLIR verwendeten Klassen. Die Pfeile repräsentieren eine "benutzt"-Beziehung. Deutlich wird die Zweiteilung von Programm- und Zielarchitekturdarstellung, deren Aufbau in den beiden folgenden Abschnitten 3.1 und 3.2 im Detail besprochen wird.

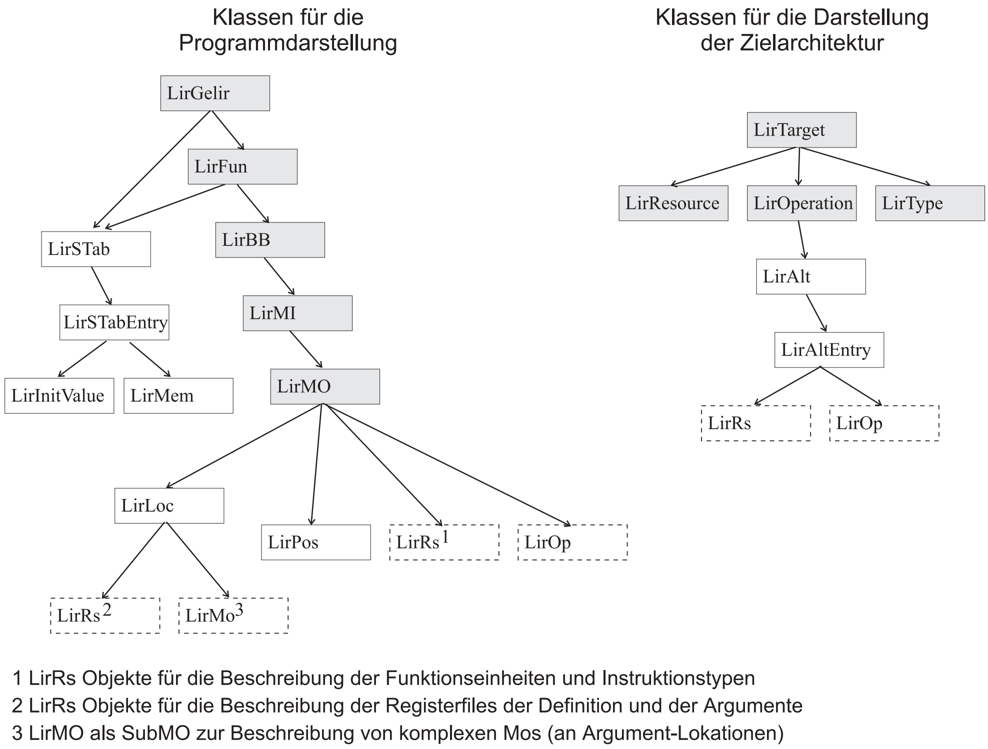

Abb. 3.1: Überblick über die GeLIR-Klassen

3.1 Zielarchitektur-Beschreibung

Zur Beschreibung der Zielarchitektur gehören *Ressourcen*, wie Registerfiles und Funktionseinheiten, sowie *Typen* und *Operationen*. Diese werden alle in einem *LirTarget*-Objekt verwaltet. Als zusätzliche Ressource werden in GeLIR *Instruktionstypen* definiert, die die Möglichkeiten der parallelen Ausführung verschiedener Operationen beschreiben. Die verschiedenen Komponenten werden im Folgenden im Einzelnen vorgestellt:

3.1.1 Typen

GeLIR unterstützt alle Standardtypen von C, wie z.B. CHAR, INT, FLOAT, PTR oder VOID, und erlaubt darüber hinaus die Definition von BOOL- und STRING-Typen. Jeder dieser Typen wird mit weiteren Informationen beschrieben, wie Größe in Bytes oder Bits, signed/unsigned und für floats eine exakte Angabe für Mantisse und Exponent. Komplexe Typen wie Arrays, Structs und Pointer werden durch Hierarchien von *LirType*-Objekten beschrieben, indem in einem Basisobjekt die Klasse des Typen (ARRAY/STRUCT/PTR) definiert wird und von dort als Subtyp(en) die weiteren Informationen angelegt werden. In Abb.3.2 werden z.B. die Typen *int*** und ein Array vom Typ *int* definiert. Die Dimension des Arrays wird im Basisobjekt abgelegt.

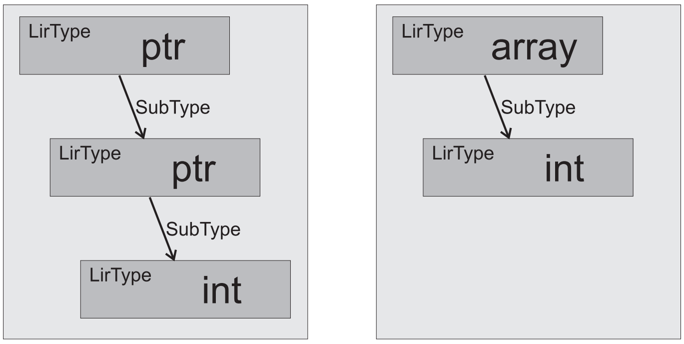

Abb. 3.2: Aufbau von komplexen Typen: *int**** (links), *array von integern* (rechts)

Als weiterer Typ wird der Funktionstyp unterstützt, der den Rückgabewert (in einem Subtyp) und die Parameter (in einem Vektor) einer Funktion beschreibt. Typen können durch ein Flag als Maschinentypen gekennzeichnet werden, aber unabhängig davon werden alle Typen im *LirTarget*-Objekt abgelegt. Jedes *LirType*-Objekt verfügt darüber hinaus über weitere Informationen, wie den Namen des Typen und einen eindeutigen Identifier (ID).

3.1.2 Ressourcen

Die Ressourcen der Zielarchitektur werden alle im *LirTarget*-Objekt abgelegt. Es wird zwischen Registerfiles, Funktionseinheiten und Instruktionstypen unterschieden, die durch jeweils eine eigene Instanz der *LirResource* Klasse beschrieben werden. In jeder Ressource werden Informationen wie z.B. der Name und eine ID abgelegt, andere Informationen hängen von der Art der Ressource ab.

Registerfiles

Alle Speicherressourcen, wie Register, Hauptspeicher und flüchtige Ressourcen[1] werden jeweils durch ein *LirResource*-Objekt beschrieben. In dem Objekt wird neben dieser Klassifizierung auch die Größe jeder Ressource angegeben, die die Zahl der Speichereinheiten für Hauptspeicher oder die Zahl der Instanzen eines Registers beschreibt. Jede dieser Speichereinheiten bzw. Instanzen wird mit einem global eindeutigen Index versehen. Darüber hinaus wird der Typ spezifiziert, der von einer Instanz/Speichereinheit der Ressource aufgenommen werden kann. Außerdem können weitere Informationen abgelegt werden, wie z.B. ein Assemblermnemonik.

GeLIR unterstützt die Beschreibung von Sub-Registerfiles, mit denen z.B. eine Modellierung von parallelen Datenpfaden in SIMD-Architekturen möglich ist, oder irreguläre Registerfiles

[1]Flüchtige Ressourcen sind nur innerhalb eines Taktzyklus gültig und werden zur Darstellung von komplexen Maschinenoperationen benutzt (siehe Kap.3.2.4)

unterschieden werden können. Im folgenden Beispiel wird die Registerfile-Ressource *RegA* für die M3-DSP-Architektur [FWD+98] beschrieben. Diese wird für die verschiedenen Pfade in einzelne Sub-Registerfiles aufgeteilt (siehe Abb. 3.3).

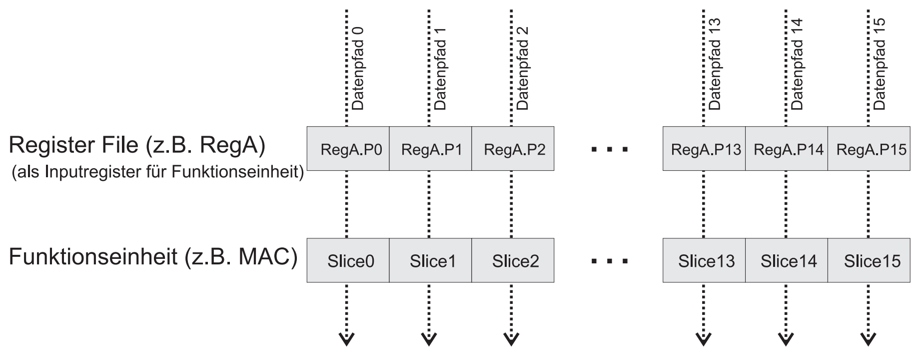

Abb. 3.3: Ausschnitt einer SIMD-Architektur mit 16 Datenpfaden

In GeLIR können diese Register durch eine Registerhierarchie ausgedrückt werden (siehe Abb. 3.4).

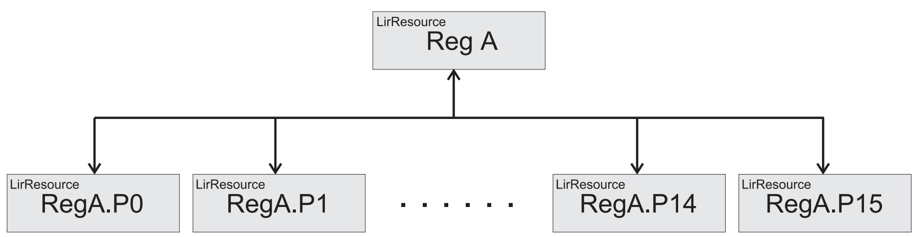

Abb. 3.4: Registerfilebeschreibung (oben) und einzelne Sub-Register

Funktionseinheiten

Funktionseinheiten werden zwar auch in einem *LirResource*-Objekt abgelegt, allerdings reichen für ihre Beschreibung weniger Informationen aus. Funktionseinheiten benötigen keinen Typ (in GeLIR wird standardmäßig auf ein *LirType*-Objekt vom Typ *none_type* verwiesen) und werden im Gegensatz zu Speicherressourcen nicht weiter klassifiziert. Mehrere Instanzen einer Funktionseinheit können in einem Objekt durch die Größe definiert werden. Dabei wird jeder Instanz ein eindeutiger Index zugewiesen.

Instruktionstypen

Mit Hilfe von Instruktionstypen können Möglichkeiten der parallelen Nutzung mehrerer Funktionseinheiten in einem Maschinenzyklus beschrieben werden. Eine einfache Überprüfung, ob in einem Zyklus jede Funktionseinheit nur einmal belegt wird, ist nicht ausreichend, da bestimmte Kombinationen von Funktionseinheiten architekturbedingt nicht erlaubt sein können. Um diese Randbedingungen zu beschreiben, werden Instruktionstypen definiert. Dabei werden Operationen, die potentiell parallel zueinander ausgeführt werden können, demselben Instruktionstypen zugeordnet; hierbei ist es möglich eine Operation mehreren Instruktionstypen zuzuordnen. Die Möglichkeit einer parallelen Ausführung kann sich dennoch durch andere Ressourcekonflikte, wie z.B. bei Registerfiles, als nicht machbar erweisen. Die Beschreibung eines Instruktionstypen in GeLIR ähnelt dem Aufbau von Funktionseinheiten. Diese sind ebenfalls in einem *LirResource*-Objekt abgelegt, die Größe ist allerdings auf eins begrenzt, so dass jeder Instruktionstyp genau einen Index hat. Der Typ wird ebenfalls standardmäßig als *none_type* definiert.

3.1.3 Operationen

In GeLIR werden sowohl abstrakte Operationen als auch Zielmaschinenoperationen durch jeweils ein *LirOperation*-Objekt beschrieben. Diese werden alle im *LirTarget*-Objekt abgelegt. In jeder Operation werden ein Name und optional ein Simulationsname definiert und ein Typ referenziert. Im undefinierten Zustand ist dies *none_type*, sonst ein Funktionstyp, der die Definition und die Argumente beschreibt.

*			
1. Alternative		2. Alternative (swapped_args)	
OP	MUL	OP	MUL
FU	DMU	FU	DMU
IT	it_1	IT	it_1
Def	Accu.Lo, *	Def	Accu.Lo, *
Arg1	RegA, RegB, const_0, const_1	Arg1	RegA, RegB, RegC, Accu.Lo
Arg2	RegA, RegB, RegC, Accu.Lo	Arg2	RegA, RegB, const_0, const_1

Abb. 3.5: Zwei mögliche Alternativen für eine abstrakte Operation "*"

Für jede Operation werden nun in mehreren *Alternativen* die gültigen Ressourcekombinationen beschrieben. Eine Alternative wird in jeweils einem separaten *LirAltEntry*-Objekt ausgedrückt. Alle *LirAltEntry*-Objekte einer Operation werden zu einem *LirAlt*-Objekt zusammengefasst. Jede Alternative setzt sich aus einer Operation, aus Funktionseinheiten, Instruktionstypen sowie Registerfile-Ressourcen für die Definition und die Argumente zusammen (→Abb.3.5).

Sowohl abstrakte als auch Zielmaschinenoperationen werden mit Alternativen überdeckt. Dabei beschreiben die Alternativen einer Zielmaschinenoperation ausschließlich die Kombinationsmöglichkeiten der Hardwareressourcen und haben deshalb als feste Operationsalternative nur die überdeckte Operation selbst. Abstrakte Operationen hingegen werden mit Alternativen überdeckt, die verschiedene Operationen der Zielarchitektur enthalten, und zwar solche, die es ermöglichen, die abstrakte Operation durch gleichwertige zu ersetzen. Im Beispiel in Abb. 3.6 wird die abstrakte Operation "+" von den Zielmaschinenoperationen ADD und INC überdeckt. Die Zielmaschinenoperation ADD hingegen enthält nur Überdeckungen mit der eigenen Operation, die dann die gültigen Ressourcekombinationen beschreiben.

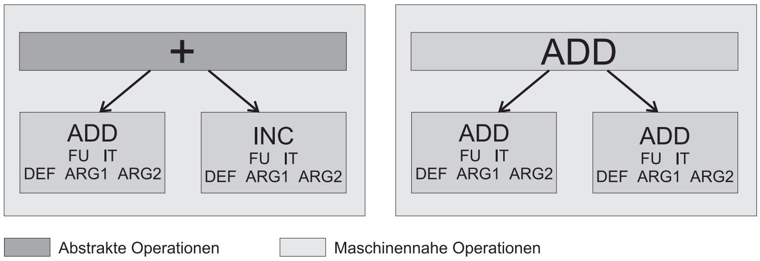

Abb. 3.6: Überdeckungsprinzip für abstrakte und Zielenmaschinenoperationen

Um Kommutativität in binären Operationen auszudrücken werden "gespiegelte" Alternativen angelegt, indem die Ressourcemengen der Argumente bei der zweiten Alternative gegenüber der ersten getauscht werden (vgl. Abb.3.5). In der Alternative, in der die Argumentressourcen vertauscht sind, wird das Attribut *swapped_args* gesetzt, wodurch beim Herausschreiben von Assemblercode die Korrektheit gewährleistet werden kann. GeLIR bietet die Möglichkeit, ein in der Datenstruktur repräsentiertes Programm zu simulieren. Dazu werden für nicht abstrakte Operationen weitere Informationen benötigt, die eine genaue Berechnungsvorschrift für diese Operationen vorgeben. Diese können in jedem *LirOperation*-Objekt abgelegt werden.

3.2 Programmdarstellung

3.2.1 Programmhierarchie

Die Anweisungen der Programmdarstellung sind, wie in Abb.3.7 dargestellt, in einer hierarchischen Struktur (LirGeLIR→LirFun→LirBB→LirMI→LirMO) abgelegt , wobei jede Ebene besondere semantische Eigenschaften repräsentiert. Jedes Objekt dieser Hierarchie besitzt eine eindeutige ID.

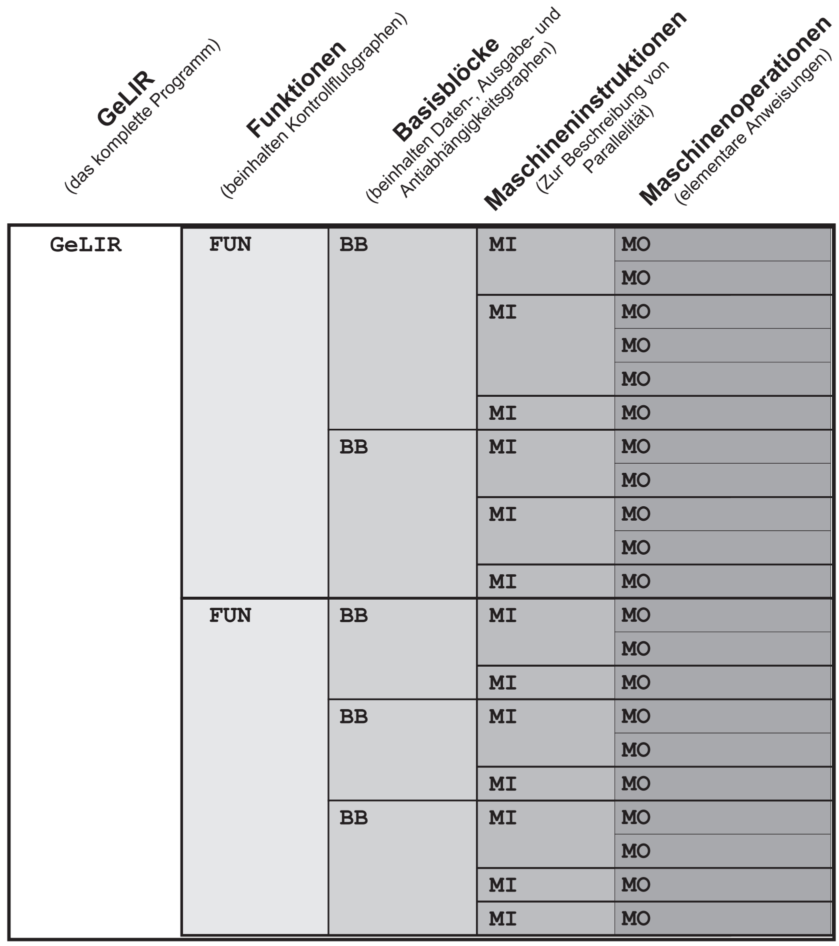

Abb. 3.7: Hierarchie der GeLIR-Programmdarstellung

- **LirGeLIR:** Die oberste Stufe verwaltet eine Liste von Funktionen und repräsentiert damit das komplette Programm. Außerdem ist die globale Symboltabelle (*LirSTab*) hier abgelegt, die Informationen zu den im Programm benutzten globalen Variablen und Konstanten beinhaltet.
- **LirFun:** Jedes Objekt dieser Klasse repräsentiert eine Funktion des Sourcecodes. Zu jeder Funktion gibt es eine lokale Symboltabelle (*LirSTab*), die wiederum einen Verweis auf die globale Symboltabelle enthält. Diese Struktur ermöglicht, dass nicht gefundene lokale Symbole automatisch in der globalen Symboltabelle gesucht werden. Eine GeLIR-Programmdarstellung enthält zumindest die "main"-Funktion. In jeder Funktion sind Informationen wie der Name der Funktion und der Ursprungsdatei abgelegt. Zusätzlich werden in *LirFun*-Objekten Daten von Kontrollflussanalysen verwaltet.
- **LirBB:** Die nächste Stufe in der Hierarchie der Programmdarstellung sind die Basisblöcke.

Definition 3.2.1 *Ein Basisblock bezeichnet eine maximale Folge fortlaufender Anweisungen, in die der Kontrollfluss am Anfang eintritt und die er am Ende verlässt, ohne dass er - außer am Ende - verzweigt.*

Abb.3.8 zeigt die Aufteilung einer Folge von Instruktionen in Basisblöcke. In MO3 teilt sich der Kontrollfluss zum ersten Mal, so dass die ersten drei MOs zu einem Basisblock zusammengefasst werden. In MO7 läuft der Kontrollfluss zusammen, folglich bildet auch diese einen eigenen Basisblock. Der unbedingte Sprung in MO5 unterbricht den sequentiellen Kontrollfluss. also liegt auch hier eine Basisblockgrenze. Die noch verbleibende MO6 wird in einem eigenen Basisblock eingebettet.

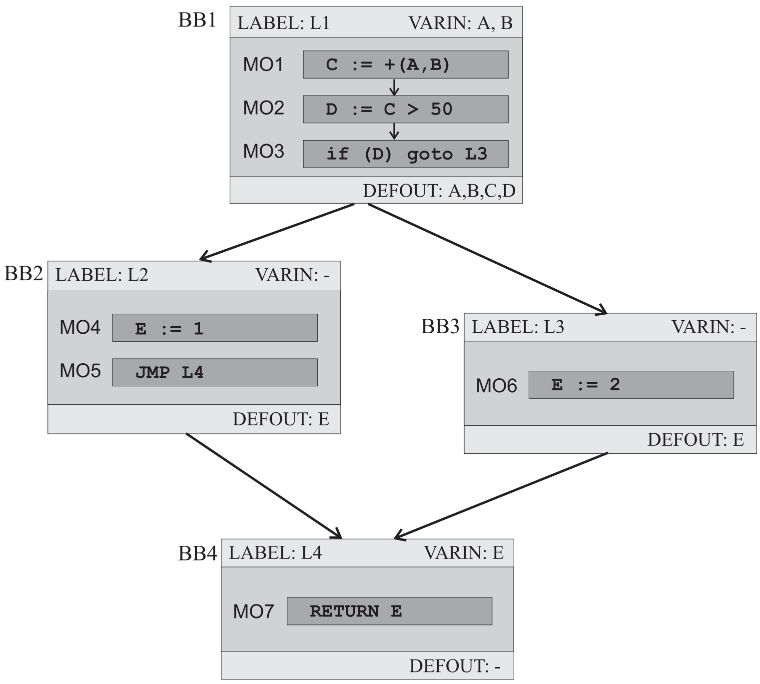

Abb. 3.8: Aufteilung eines Programmteils in Basisblöcke

Jeder Basisblock kann als Knoten im Kontrollflussgraphen dargestellt werden. In Basisblöcken werden Sprunglabel verwaltet und zusätzliche Informationen zum Datenfluss abgelegt. Jede Variable, die in einem Basisblock benutzt wird (use), ohne vorher in diesem definiert zu werden (define), wird als *VARIN* in eine Liste des Basisblocks eingetragen. Jede innerhalb eines Basisblockes definierte Variable wird ihrerseits in die *DEFOUT*-Liste eingetragen. Diese Eintragungen werden von GeLIR automatisch durchgeführt und dienen als Basis für die Analyse von Daten- Ausgabe- und Antiabhängigkeiten.

- **LirMI:** Maschinenoperationen (MO), die parallel ausgeführt werden können, werden zu einer Maschineninstruktion (MI) zusammengefasst. Die Möglichkeit der parallelen

Ausführung ist allerdings nur gegeben, wenn keine Ressourcekonflikte vorliegen und Datenabhängigkeiten berücksichtigt werden (vgl. 3.1.3).

- **LirMO** Die unterste Stufe, die Maschinenoperation (MO), repräsentiert genau eine elementare Anweisung. In dieser können einfache Konstanten, Adressen, arithmethische und logische Operationen, Loads and Stores, bedingte und unbedingte Sprünge, sowie Funktions-Aufrufe und Rücksprünge (Returns) beschrieben werden. Spezielle MOs, die Variablen repräsentieren, sind die oben erwähnten VARINs und DEFOUTs. Diese werden auch in einem *LirMO*-Objekt instanziiert, sind allerdings nur in speziellen Listen innerhalb eines *LirBB*-Objekts verankert und kommen nicht als Teil einer Maschineninstruktion der Programmdarstellung vor. Eine detaillierte Übersicht über die verschiedenen MOs, die von GeLIR unterstützt werden, und die dort abgelegten Informationen findet sich in Kap.3.2.3.

GeLIR unterstützt die Programmanalyse, indem Kontrollflussgraphen und verschiedene Datenflussgraphen automatisch generiert werden. Der Kontrollfluss kann anhand der Verknüpfungen der Basisblöcke einer Funktion oder von Funktionsaufrufen bestimmt werden. Die Informationen werden auf der jeweils nächst höheren Hierarchiestufe in einem *LirFun* bzw. im *LirGeLIR* generiert und abgelegt. Datenflussgraphen hingegen werden aus Informationen auf MI-Ebene generiert, die in Basisblöcken abgelegt werden. Änderungen auf der unteren Hierarchieebene (MOs/MIs), die Auswirkungen auf die Graphen haben, werden über interne Events an die oberen Ebenen weitergereicht.

3.2.2 Symboltabellen

Sowohl im globalen *LirGeLIR*-Objekt als auch in jedem *LirFun*-Objekt ist eine globale bzw. lokale Symboltabelle abgelegt. Diese bestehen aus einer Liste von Symboltabelleneinträgen, die neben den Basisinformationen Initialisierungswerte (*LirInitValue*) oder ein Mapping des abstrakten Symboltabelleneintrags auf eine oder mehrere Speicherressourcen (*LirMem*) enthalten können. Jeder Eintrag wird auf zwei Arten klassifiziert:

- Der Typ des Eintrags: VAR / FUN / LABEL / CONST
- Der Geltungsbereich (Scope): GLOBAL / LOCAL / STATIC

Zu den weiteren Daten, die abgelegt werden, gehören der Name und die Referenzierung eines Typen. Darüber hinaus wird über verschiedene Attribute beschrieben, ob ein Eintrag temporär ist oder in einem Register gehalten werden soll.

In einem *LirInitValue*-Objekt wird ein Initialisierungswert eingetragen. Sollte der Typ des Symboltabelleneintrags komplex sein (z.B. ein Array), so wird eine *LirInitValue*-Hierarchie aufgebaut.

LirMem-Objekte werden als relative und absolute Speicherbereiche klassifiziert. Als physikalische Speicherressource wird eine Registerfile- Ressource referenziert und der Speicherbereich (Start- und Zieladresse) definiert.

3.2.3 Darstellung verschiedener MOs

In folgenden Abschnitt werden die von GeLIR unterstützten Typen von Maschinenoperationen beschrieben und die dafür nötigen Datenstrukturen aufgezeigt. Dabei werden die für die abstrakte Darstellung und für die Überdeckung nötigen Objekte vorgestellt. Die von GeLIR unterstützten Typen von MOs lassen sich in drei Gruppen aufteilen:

- Zur ersten Gruppe gehören alle MOs, die eine abstrakte Operation repräsentieren. Diese können von einer Zielmaschinenoperation überdeckt werden. Dazu gehören einfache n-äre Operationen, Load und Stores, Sprungoperationen, Funktionsaufrufe und -rücksprünge. Ein *LirMO*-Objekt dieser Gruppe enthält immer eine Referenz zu einem *LirOp*-Objekt, welches Informationen zu der abstrakten Operation enthält. Des Weiteren gibt es je nach MO-Typ eine Definition und eine bestimmte Zahl von Argumenten. Jede dieser Lokationen ist in einem *LirLoc*-Objekt gekapselt. Einige MOs benötigen weitere Informationen, wie z.B. eine Sprungadresse, die ebenfalls in der MO abgelegt werden.

- Die zweite Gruppe bilden unterstützende MOs, etwa die Darstellung einer Konstanten oder Adresse. Diese *LirMO*-Objekte verweisen nicht auf ein *LirOp*-Objekt, sondern in der Regel nur auf ein Objekt einer spezifischen Klasse, z.B. *LirConst* oder *LirAddr*.

- Einen Sonderfall bildet die dritte Gruppe von MOs, die eine Variable beschreiben (VAR-MOs). Diese werden von GeLIR automatisch innerhalb eines Basisblocks angelegt und repräsentieren die eingehenden und ausgehenden Variablen des jeweiligen Blocks (vgl. Kap.3.2). VAR-MOs bestehen entweder aus einer Definition (bei VARIN) oder genau aus einem Argument (DEFOUT).

Im Einzelnen lassen sich folgende von GeLIR unterstützte Typen von Maschinenoperationen benennen:

- **Arithmetische und logische Operationen (OPR)** (→ Abb.3.9) haben immer eine Definition und eine bestimmte Zahl von Argumenten, abhängig davon, ob sie eine unäre, binäre oder n-äre Operation repräsentieren. Die Informationen über die Operation werden in einem *LirOp*-Objekt abgelegt, dieses enhält im Wesentlichen eine Referenz auf das entsprechende abstrakte *LirOperation*-Objekt in *LirTarget* (vgl. 3.1.3). Im MO-Objekt werden auch die Keys der Operationen abgelegt, die diese Operation überdecken (*KeySet*). Die Informationen zur Definition und zu den Argumenten sind in jeweils einem *LirLoc*-Objekt abgelegt. Neben abstrakten Informationen, wie z.B. einer Referenz zum zugehörigen Symboltabelleneintrag, wird in diesen Objekten jeweils auf ein *LirRs*-Objekt verwiesen, in dem alle Registerfile-Ressourcen enthalten sind, die die Definition bzw. das Argument überdecken können. Argument-Lokationen können darüber hinaus auf eine SubMO verweisen. In diesem Fall ist *diese* MO Basis einer komplexen MO (siehe auch 3.2.4). Schließlich sind in der MO noch zwei *LirRs*-Objekte abgelegt. Diese enthalten Informationen, auf welchen Funktionseinheiten die betreffende MO ausgeführt werden kann, bzw. welche Instruktionstypen dieser MO zugewiesen werden können.

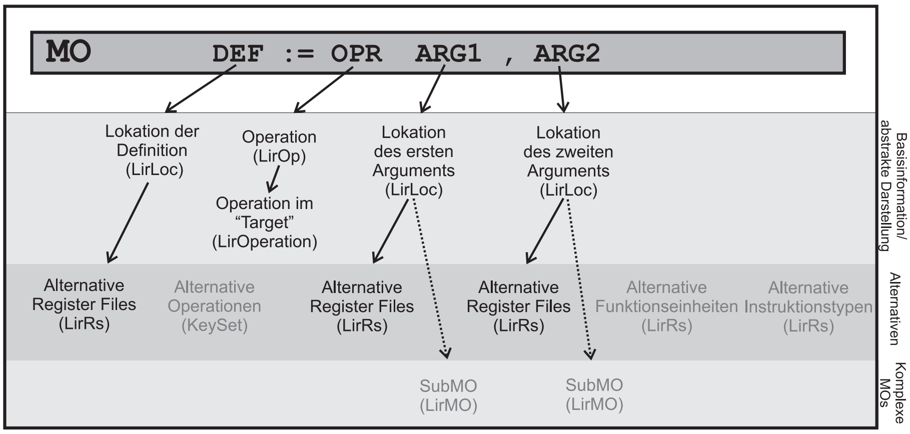

Abb. 3.9: Aufbau einer MO, die eine binäre Operation repräsentiert

Folgende MOs haben den gleichen Aufbau wie arithmetische Operationen:

- **Virtuelle Operationen (VOPR)** sind n-äre Operationen und können z.B. zur Darstellung eines ϕ-Knoten in einer SSA Struktur benutzt werden.
- **Loads (LD)** werden als unäre oder binäre Operation dargestellt. In der binären Darstellung beschreibt die Definition das zu ladene Registerfile, das erste Argument die Adresse, auf die zugegriffen wird, und das zweite Argument die Speicherressource.
- **Stores (ST)** werden als binäre Operation repräsentiert. Die Definition beschreibt die Speicherressource, das erste Argument beschreibt die Adresse und das zweite Argument gibt das zu speichernde Registerfile an.
- Ein **Move (MV)** dient zur Darstellung eines Datentransfers auf Maschinenebene und hat den Aufbau einer unären Maschinenoperation.
- Ein **Copy (COPY)** kann im Gegensatz zu einem Move auch mehrere Datentransfers umfassen. Um diese Operation auf einfache Weise durch eine LD/ST-Operation ersetzen zu können, ist eine COPY-MO als binäre Operation definiert.
- Ein **Casting (CAST)** zur Typumwandlung ist eine unäre Operation.

Darüber hinaus gibt es folgende Maschinenoperationen, die vom Aufbau arithmetischen Maschinenoperationen gleichen, aber zusätzliche Informationen enthalten:

- **Unbedingte Sprünge (JMP)** (→ Abb.3.10) haben kein Argument, verweisen aber auf ein *LirAddr*-Objekt, in dem das Sprungziel definiert wird.

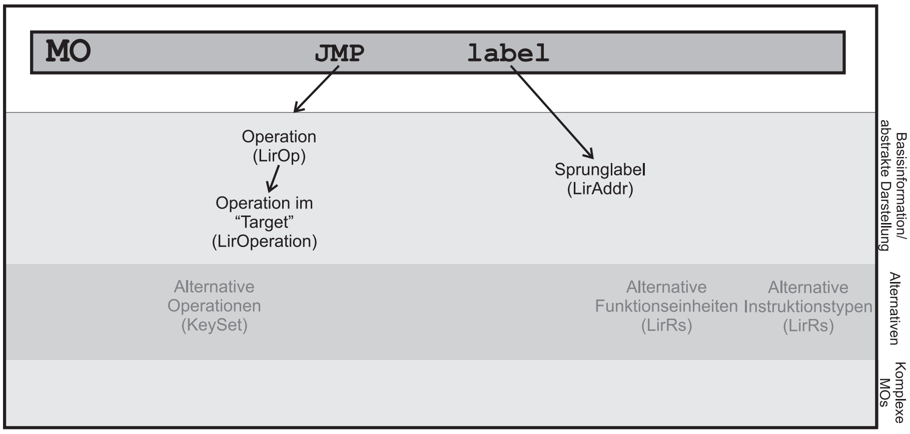

Abb. 3.10: Aufbau einer MO für einen unbedingten Sprung

- Bei **bedingten Sprüngen (CJMP)** (→ Abb.3.11) wird gegenüber unbedingten Sprüngen zusätzlich ein Argument verwaltet, das die Sprungbedingung enthält. Für dieses Argument können auch alternative Registerfile-Ressourcen abgelegt werden.

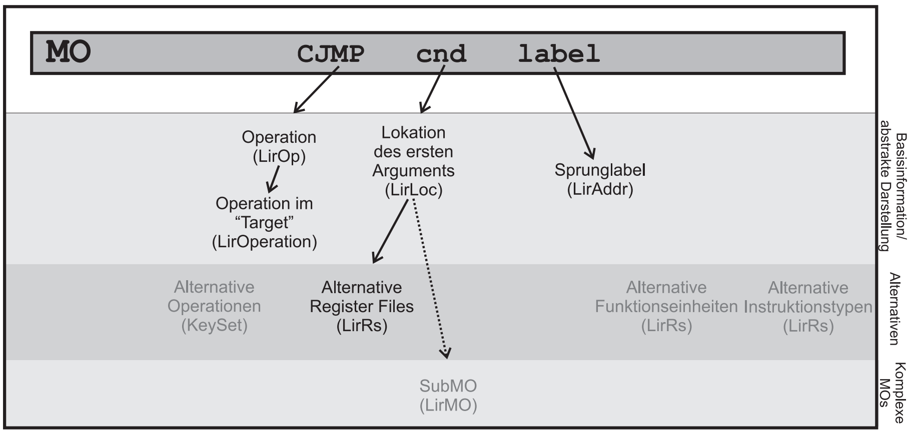

Abb. 3.11: Aufbau einer MO für einen bedingten Sprung

- In **Funktionsaufrufen (CALL)** (→ Abb.3.12) werden in der Definition der Rückgabewert und in n Argumenten die einzelnen Funktionsparameter abgelegt. Zusätzlich wird ein Symboltabelleneintrag referenziert, der die Information für die aufzurufene Funktion enthält.

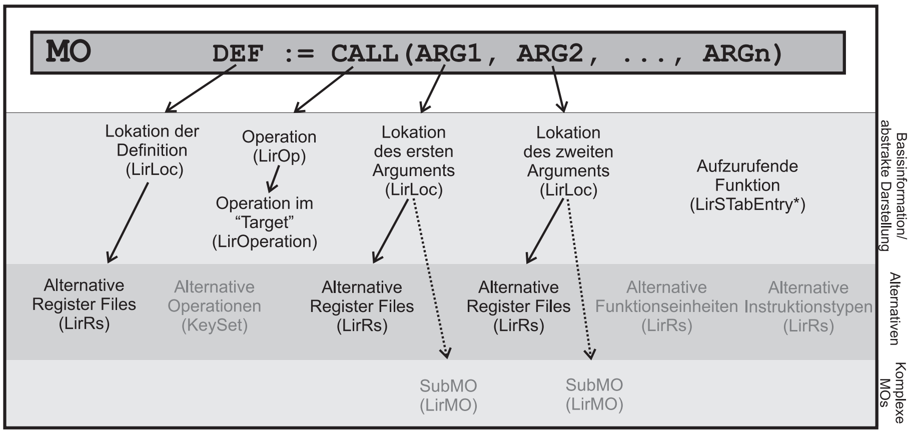

Abb. 3.12: Aufbau einer MO für einen Funktionsaufruf

- Ein **einfacher Funktionsrücksprung (RET)** (→ Abb.3.13) enthält keine Definition und keine Argumente.

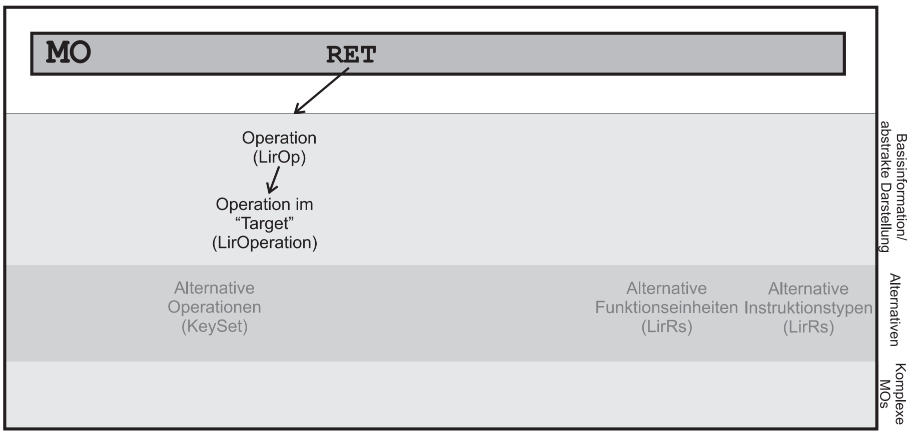

Abb. 3.13: Aufbau einer MO für einen Funktionsrücksprung

- Ein **Funktionsrücksprung mit Rückgabewert** (→ Abb.3.14)wird gegenüber einem einfachen Funktionsrücksprung um ein Argument erweitert, das die Rückgabevariable enthält.

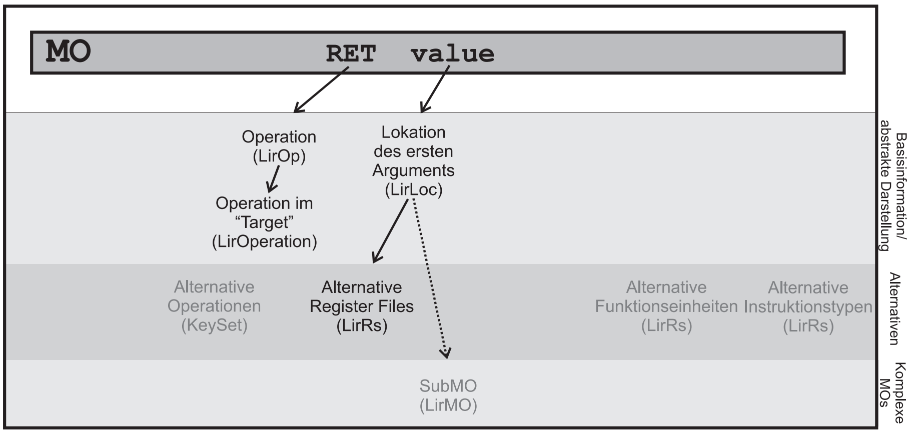

Abb. 3.14: Aufbau einer MO für einen Funktionsrücksprung mit Rückgabewert

- Eine spezielle, stark maschinenorientierte Klasse bilden die MOs zur Modellierung des **Zero-Overhead-Hardwareloops (ZLOOP/ZJMP)**, der mit einer einleitenden ZLOOP-MO und einer abschliessenden ZJMP-MO beschrieben wird. Eine **ZLOOP-MO** (→ Abb.3.15) hat als zusätzliche Information zum einen ein *LirAddr*-Objekt, das das Sprunglabel für die im Programm nachfolgende **ZJMP-MO** repräsentiert, zum anderen die Zahl der Iterationen, die durchgeführt werden sollen. Um verschachtelte Hardwareloops zu ermöglichen, ist ein zusätzliches Attribut definiert, das Auskunft darüber gibt, ob diese Schleife innerhalb einer anderen liegt. Der Aufbau einer **ZJMP-MO** entspricht der MO für einen unbedingten Sprung. Allerdings handelt es sich hierbei semantisch um einen bedingten Sprung, der ein spezielles Hardware-Register auf ungleich 0 testet und dekrementiert. Zusätzlich wird die zugehörige ZLOOP-MO referenziert.

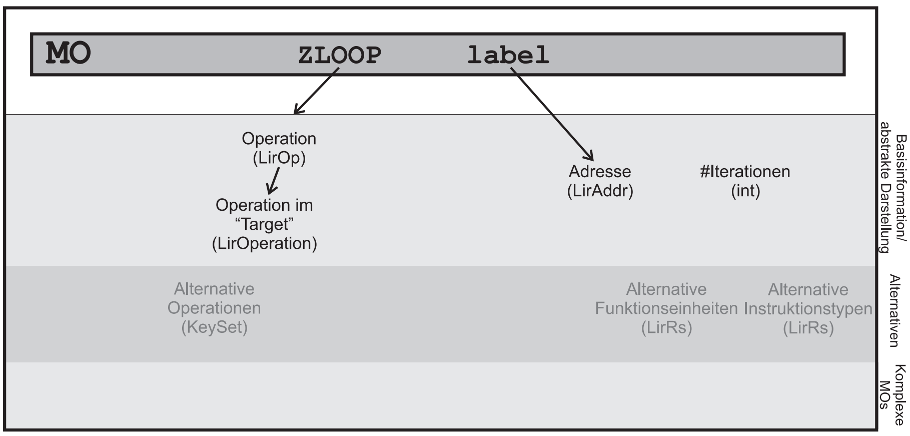

Abb. 3.15: Aufbau einer MO für einen Zero-Overhead Hardwareloop

Die folgenden MOs stellen keine abstrakte Operation dar, sondern dienen zur Modellierung von Konstanten bzw. Adressen:

- Eine MO für eine **Konstante (CONST)** (→ Abb.3.16) besteht aus einer Definition, die einen konstanten Wert darstellt, und einem referenzierten *LirConst*-Objekt, das Informationen über den Typ und den Wert der Konstanten enthält.

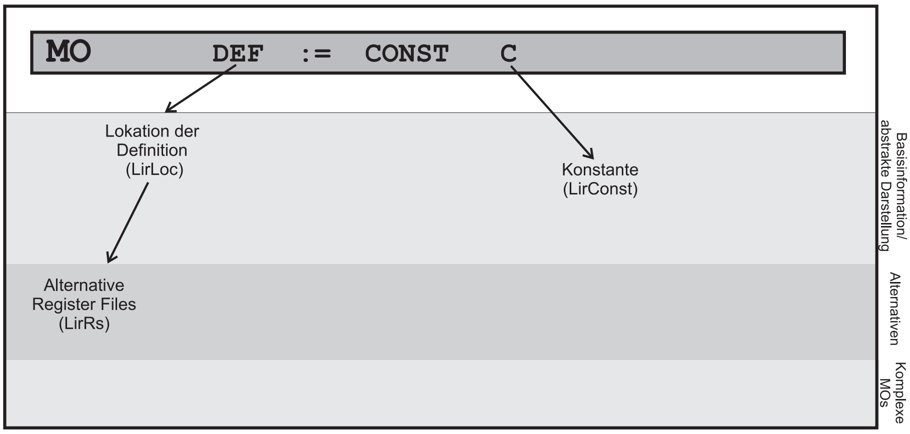

Abb. 3.16: Aufbau einer MO für eine Konstante

- Eine **Adresse (ADDR)** (→ Abb.3.17) besteht aus einer Definition, die eine Speicheradresse darstellt. Informationen zu dieser Speicheradresse sind in einem *LirAddr*-Objekt abgelegt.

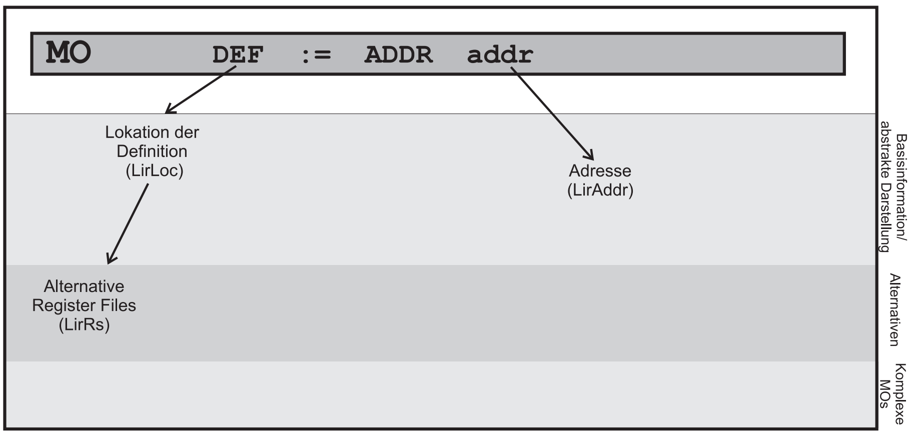

Abb. 3.17: Aufbau einer MO für eine Adresse

Schließlich gibt es in GeLIR noch zwei Typen von Var-MOs, die u.a. die Basis für Datenflussanalysen bilden.

- **VARIN** (→ Abb.3.18) und **DEFOUT** (→ Abb.3.19) bestehen entweder aus einer Definition oder einem Argument. Diese Informationen sind ausreichend für die verschiedenen Datenflussanalysen.

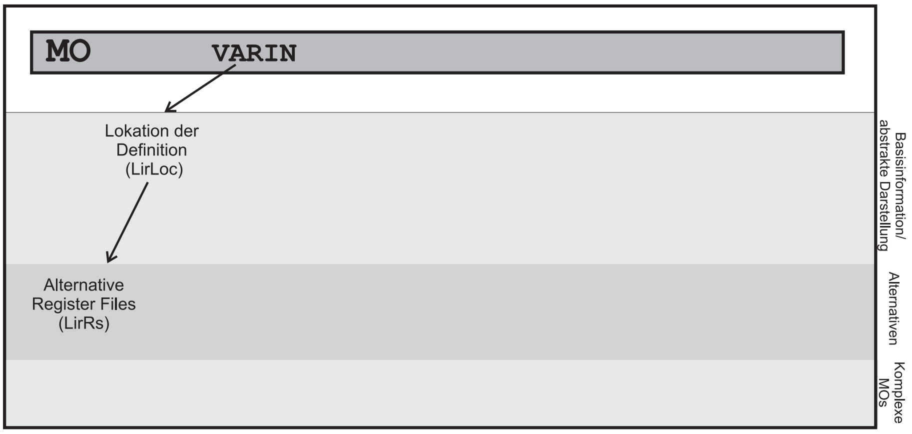

Abb. 3.18: Aufbau einer VARIN-MO

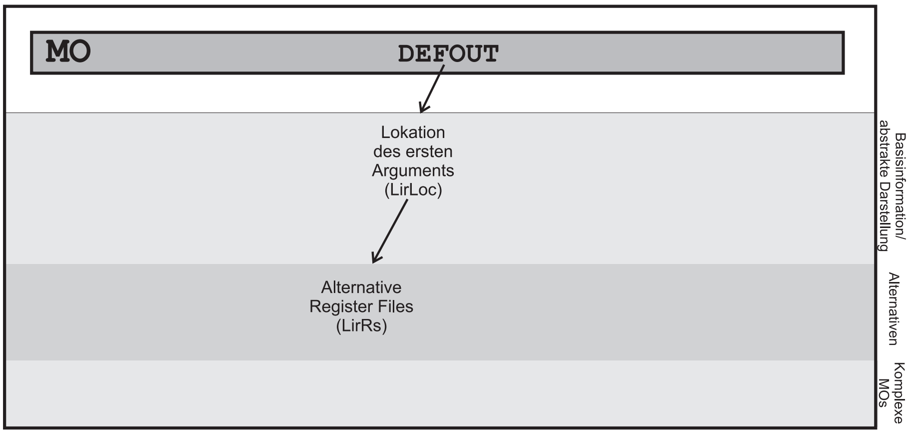

Abb. 3.19: Aufbau einer DEFOUT-MO

3.2.4 Darstellung komplexer Operationen

GeLIR unterstützt die Darstellung komplexer Operationen, wie z.B. die MAC-Operation, und beschreibt diese durch Hierarchien von LirMO-Objekten. Das Beispiel in Abb. 3.20 zeigt links den Aufbau einer MAC-Operation mit einer Multiplikation und einer anschließenden Addition des Ergebnisses mit einem zusätzlichen Eingabewert. Rechts wird die Modellierung in GeLIR gezeigt. Die beiden Berechnungsschritte werden durch je ein *LirMO*-Objekt repräsentiert. Die Berechnung des ersten Argumentes der Addition geschieht in einem vorgelagerten Schritt. Diese wird als so genannte SubMO an das Argument angehängt. Nur die Basis-MO, nicht aber die SubMO ist Teil der Maschineninstruktion (MI).

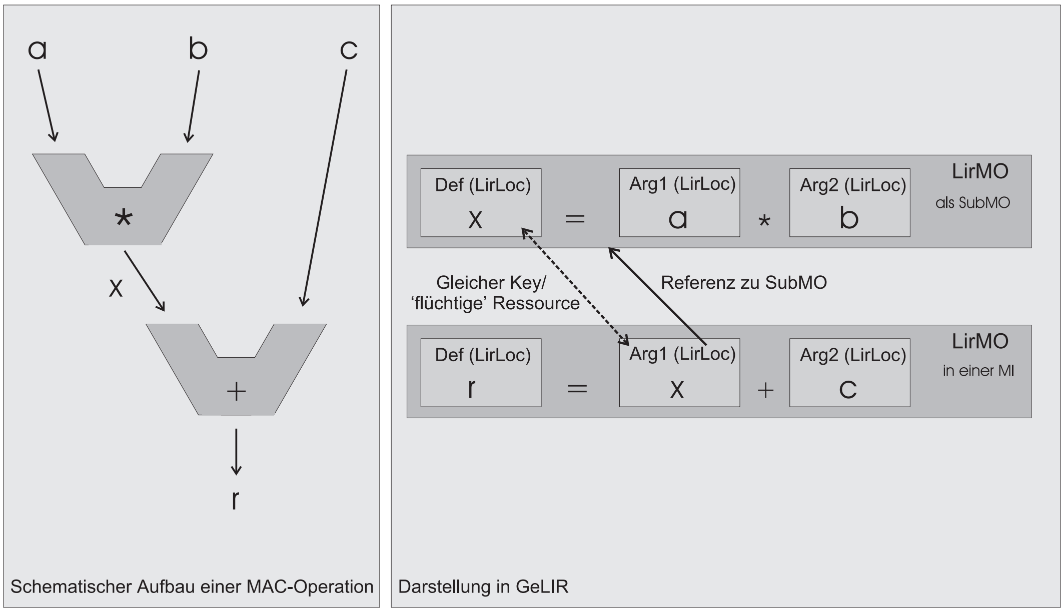

Abb. 3.20: Schematische Darstellung einer MAC-Operation (links) und die Darstellung in GeLIR

Die Symboltabelleneinträge des Argumentes der Basis-MO und der Definition der SubMO müssen übereinstimmen. Um bei Überdeckungsalternativen komplexe Kombinationsmöglichkeiten zu beschreiben, werden die Lokationen mit so genannten flüchtigen Ressourcen überdeckt, deren Gültigkeit auf einen Taktzyklus beschränkt ist. Zum Beispiel werden für die Beschreibung der MAC-Operation die Definition der Multiplikation (vgl. Abb.3.5) und das erste Argument der Addition mit einer flüchtigen Ressource überdeckt.

Die Hierarchien komplexer MOs können beliebig erweitert werden. Da Konstanten in GeLIR als eigene *LirMO* dargestellt werden, könnte z.B. ein konstanter Parameter der MAC-Operation als weitere SubMO an eines der Argumente der ersten SubMO angehängt werden.

3.2.5 Darstellung alternativer Ausführungsmöglichkeiten

Zum Abschluss von Kapitel 3 soll noch einmal das dynamische Vorgehen während der Codegenerierung verdeutlicht werden. Abb. 3.21 zeigt die Entwicklung der Überdeckungen einer einzelnen Maschinenoperation(MO) während der Durchführung einer Codegenerierung.

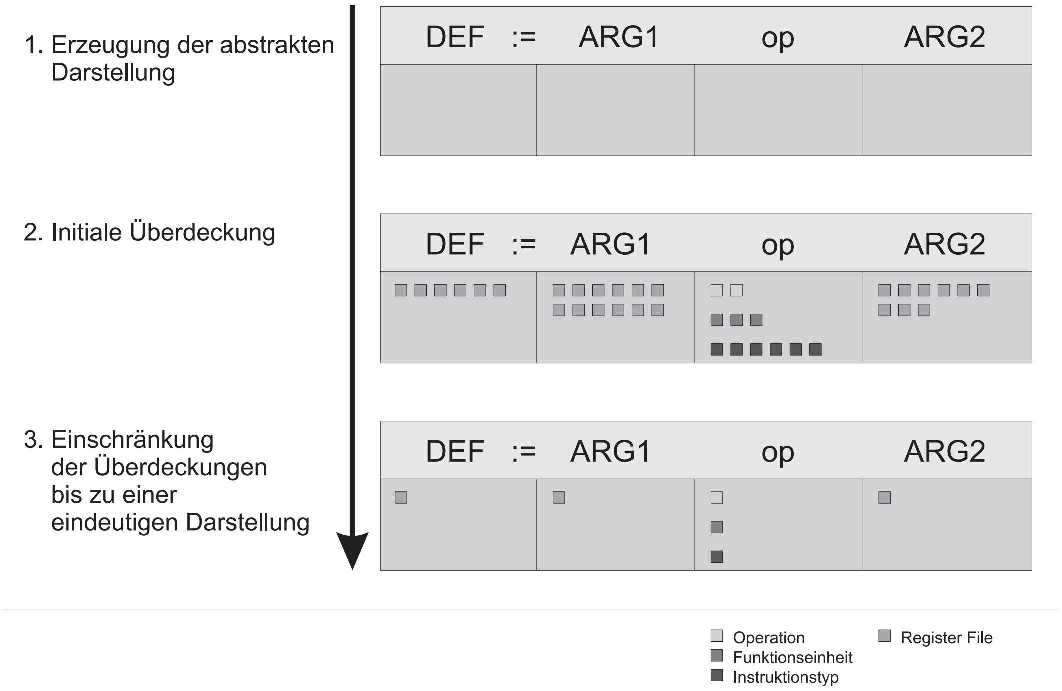

Abb. 3.21: Entwicklung der Überdeckung einer einzelnen MO während der Codegenerierung

1. Zuerst wird die rein abstrakte Darstellung erzeugt. Dazu werden die für die Programmdarstellung benötigten *LirMO*-Objekte und die zugehörige Hierarchie aufgebaut. Diese ist zunächst ohne jede Überdeckung. Es folgt eine Beschreibung der Zielarchitektur, insbesondere eine Überdeckung der abstrakten Operationen mit Alternativen.

2. Im zweiten Schritt wird in jeder MO die abstrakte Operation mit allen Alternativen überdeckt, die dafür in der Beschreibung der Zielarchitektur abgelegt wurden. Dazu werden die alternativen Operationen und die zugehörigen Funktionseinheiten und Instruktionstypen in jeder MO eingetragen, und für die Definition und die Argumente alle gültigen Registerfile-Ressourcen abgelegt.

3. Nach diesen beiden Schritten der Initialisierung werden nun die Ressourcenmengen nach und nach eingeschränkt. Dies ist Teil der eigentlichen Codegenerierung, deren Ziel es ist, für jede MO eine eindeutige Überdeckung zu haben ohne die Semantik der Programmdarstellung zu verändern.

Kapitel 4

eXtensible Markup Language – XML

XML gehört zu den Markup-Languages (ML), deren Idee es ist, in einem Dokument neben den eigentlichen Daten auch Informationen über diese Daten (Metadaten) zu speichern. Die Metadaten bilden das so genannte Markup. Eine erste Form war die Generalized Markup Language (GML) aus dem Jahre 1969, die in einer Forschungsabteilung von IBM entwickelt wurde. Daraus entwickelte sich die Standard Generalized Markup Language (SGML), die 1986 von der ISO (International Organization for Standardization) als internationaler Standard für den Datenaustausch festgelegt wurde. Mit Hilfe dieser Sprache wurde die Möglichkeit gegeben, Daten und Metadaten nach einem vorgegebenen Format in einem Textdokument zu speichern und durch die Bildung von Hierarchien eine Beziehung zwischen einzelnen Daten herzustellen. Die Namen und Anordnung der Daten und Metadaten sind für verschiedene Anwendungsbereiche jeweils frei wählbar und erweiterbar. Die wohl bekannteste Markup-Language, die aus SGML hervorgegangen ist, ist HTML (HyperText Markup Language), für die eine Dokumentbeschreibung existiert, die die Namen der Metadaten (das sogenannte *Vokabular*) und den Aufbau der Dokumente festlegt.

SGML ist zwar durch die Erweiterbarkeit sehr flexibel, hat aber eine hohe Komplexität, die eine Anwendung erschwert. HTML hat dagegen zwar einen einfachen Aufbau, aber auch ein vorgegebenes Vokabular und ist deswegen nicht erweiterbar. Das World Wide Web Consortium (W3C) hat versucht die Einfachheit von HTML mit der Erweiterbarkeit von SGML zu kombinieren und im Februar 1998 XML, die *eXtensible Markup Language* (Erweiterbare Auszeichnungssprache) als Empfehlung (*recommendation*) verabschiedet. XML ist eine stark vereinfachte, echte Teilmenge der SGML-Grammatik, bei der die Erweiterbarkeit bewahrt wurde. Auch diese ML bietet die Möglichkeit ein eigenes Vokabular zu entwickeln, und den Aufbau von Daten und Metadaten in einem vorgegebenen Rahmen frei zu bestimmen. Ein Beispiel für ein von XML abgeleitetes Vokabular (siehe Abb.4.1) ist z.B. die Chemical Markup Language [CML] oder die in dieser Diplomarbeit entwickelte Compiler-Zwischendarstellung XeLIR.

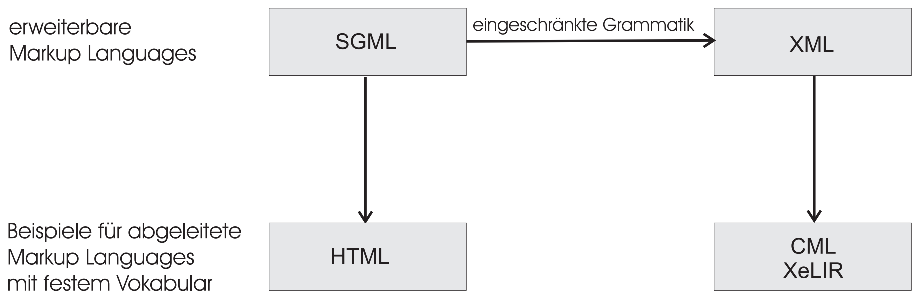

Abb. 4.1: Zusammenhänge zwischen Markup-Languages

Im nächsten Abschnitt (Kap.4.1) wird eine kurze Einführung in die XML-Syntax gegeben, die darauf folgenden Kapitel bieten eine Übersicht über Standards und Applikationen, die rund um XML entwickelt und eingesetzt werden. Dazu gehören die Beschreibung und Festlegung von einem Vokabular und einer Dokumentstruktur mit Hilfe von Document Type Definitions (DTDs) und XML-Schema (→Kap. 4.2), die Notwendigkeit und Durchführung der Definition von Namensräumen (→Kap. 4.3) und der Zugriff auf XML-Dateien mit standardisierten APIs wie DOM (*Document Object Model*) oder SAX (*Simple Api for XML*) (→Kap. 4.4). Ferner werden das Linking zwischen oder innerhalb von Dokumenten mit XLink bzw. XPath/XPointer dargestellt (→Kap. 4.5) und die Möglichkeit von Dokumenttransformationen beschrieben, die mittels XSLT ausgeführt werden können (→Kap. 4.6). Abschließend wird eine Auswahl von Applikationen für XML vorgestellt (→Kap. 4.7) und auf die im Rahmen dieser Diplomarbeit notwendige Software eingegangen.

Die folgenden Abschnitte geben einen Überblick über die Themenbereiche, für eine detailliertere Besprechung sei an dieser Stelle auf [Hun00] und [And00] verwiesen. Da viele Standards erst nach Drucklegung dieser Bücher als endgültige Empfehlung des W3C vorlagen, gibt es teilweise Abweichungen, die in erster Linie die Syntax betreffen. Die folgenden Kapitel beziehen sich auf die offiziellen Empfehlungen, auf die jeweils verwiesen wird.

4.1 Wohlgeformte XML-Dokumente: Die Syntax

In diesem Kapitel werden die Komponenten eines XML-Dokuments sowie die wesentlichen grammatischen Regeln dargestellt. Grundlage hierfür ist die Empfehlung des W3C [XMLe], für die eine deutsche Übersetzung unter [XMLf] zu finden ist.

Allgemein wird zwischen *wohlgeformten* und *gültigen* XML-Dokumenten unterschieden. Letztere müssen strengere Anforderungen erfüllen.

Definition *Wohlgeformte* XML-Dokumente entsprechen der allgemeinen XML-Syntax und den Strukturregeln der XML 1.0 Spezifikation. *Gültige* XML-Dokumente sind wohlgeformte

Dokumente, die zusätzlich den Syntax-, Struktur- und weiteren Regeln, die in einer DTD (*Document Type Definition*) definiert sind, entsprechen.

Bevor die einzelnen Komponenten eines XML-Dokuments besprochen werden, gibt der folgende Text ein Beispiel für ein XML-Dokument, das Personen bescheibt:

```
<?xml version="1.0" encoding="UTF-8"?>
<Personen>
  <!-- Einzelne Personen -->
  <Person alter="35">
    <Anrede basis="Herr" titel="Dr."/>
    <Vorname>Richard</Vorname>
    <Nachname>Müller</Nachname>
  </Person>
  <Person alter="21">
    <Anrede basis="Frau"/>
    <Vorname>Christina</Vorname>
    <Nachname>Kreuz</Nachname>
  </Person>
</Personen>
```

Die wichtigste Einheit innerhalb eines XML-Dokuments ist das sogenannte Element, zu dem ein Starttag und ein Endtag gehören, die jeweils mit einem $<$ begonnen und einem $>$ beendet werden. Die Tags schließen die eigentlichen Daten (den sogenannten Inhalt des Elements) ein und bilden die Metadaten. Das Endtag muss den gleichen Namen haben wie das Starttag, allerdings wird dem Elementnamen ein Slash ('/') vorangestellt (→Abb.4.2).

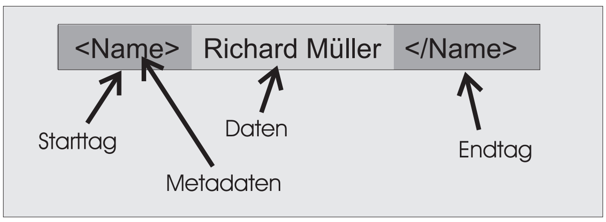

Abb. 4.2: Einfaches XML-Element

Der Inhalt eines Elements kann Text oder können weitere Kindelemente sein (→Abb.4.3). Durch diesen Aufbau einer Baumstruktur können Beziehungen zwischen Textelementen ausgedrückt werden. Kindelemente müssen im Gegensatz zu HTML streng hierarchisch sein, d.h. es darf keine Überlappung von Tags geben, wie sie von SGML erlaubt ist. Ein wohlgeformtes XML-Dokument darf darüber hinaus nur genau ein Wurzelelement (Root-Element) enthalten. Die Tag-Namen sind case-sensitiv und unterliegen bestimmten Einschränkungen.

So dürfen z.B. keine Elementnamen mit *XML* beginnen. Die Namenskonventionen sind in der Empfehlung des W3C genau definiert.

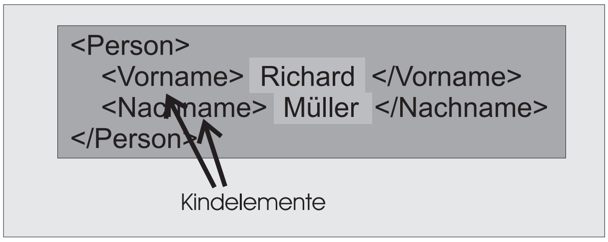

Abb. 4.3: Verschachtelte Elemente

Zwischen dem Starttag und Endtag befindet sich der *Inhalt des Elementes (element content)*. Ist dieser Inhalt reiner Text, so spricht man von **PCDATA** (Parsed Character Data). In dem obigen Beispiel sind dies die eigentlichen Daten.

Alle Leerzeichen, Zeilenvorschübe, Wagenrückläufe und Tabulatoren bilden den **White-Space**. Überflüssiger White-Space, wie z.B. mehrere aufeinanderfolgende Leerzeichen, wird im Gegensatz zu HTML nicht aus dem Element-Inhalt entfernt. Eine Ausnahme bildet der Zeilenumbruch, der unabhängig vom Betriebssystem generell aus genau einem Linefeed besteht - zusätzliche Zeichen wie ein Carriage Return werden von XML-Parsern entfernt. Darüber hinaus kann es so genannten **extraneos White-Space** geben, der nicht direkt zum Dokument gehört, sondern nur die Lesbarkeit erhöht.

Innerhalb eines Tags können **Attribute** definiert werden, die jeweils aus einem Namen und einem Wert bestehen (→Abb. 4.4). Innerhalb eines Elementes muss der Name des Attributes eindeutig sein. Der Wert ist obligatorisch und wird entweder von einfachen oder doppelten Anführungsstrichen eingeschlossen.

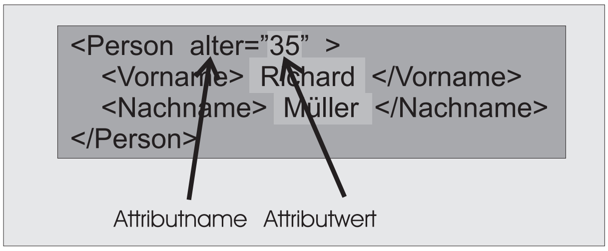

Abb. 4.4: Elemente mit Attribut

Alle Informationen, die in Attributen stehen, könnten genauso mit Hilfe von Kindelementen ausgedrückt werden, es stellt sich also die Frage nach ihrem Sinn. Attribute haben gegenüber

Kindelementen den Vorteil Informationen kompakter darzustellen. Zusätzliche Kindelemente können dagegen die Lesbarkeit des Dokuments erhöhen (→ Abb.4.5) sowie mehr Spielraum für spätere Erweiterungen geben, da z.B. die Kardinalität eines Elementes zu ändern ist, die eines Attributes innerhalb eines Elementes jedoch nicht. Des Weiteren kann man durch Attribute eine semantische Trennung von Informationen vornehmen, indem man die wesentlichen Daten in Elementen und zusätzliche Informationen mit Attributen beschreibt. In XeLIR wird sich diese Möglichkeit der Trennung als nützlich erweisen. Objekte und deren Verschachtelung wird durch Elemente und Elementhierarchien ausgedrückt, während in Attribute einzelne Objektinformationen abgelegt werden (Kap. 5).

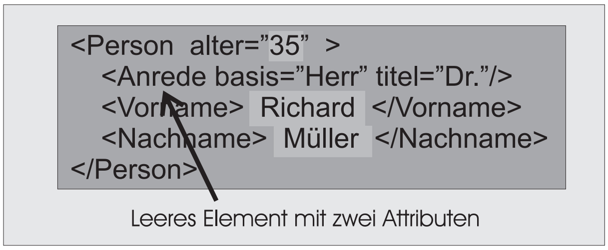

Abb. 4.5: Beispiel erweitert durch ein leeres Element mit zwei Attributen

Kommentare werden mit <! − − eingeleitet und mit −− > beendet (→Abb.4.6). Kommentare dürfen nicht innerhalb eines Tags stehen und innerhalb des Kommentars darf die Zeichenkette −− nicht vorkommen. Die XML-Empfehlung besagt, dass Parser *nicht* verpflichtet sind Kommentare an eine Applikation weiterzureichen. Dies verhindert eine nicht gewollte zusätzliche Nutzung von Kommentaren wie in HTML. Dort wird in Kommentaren z.B. der Code für JavaScript abgelegt, der dann von älteren Browsern, die neue Tags wie *<SCRIPT>* nicht unterstützen, als Kommentar einfach ignoriert wird.

Abb. 4.6: XML-Kommentar

Jedes XML-Dokument kann eine **Deklaration** enthalten (→Abb.4.7). Diese enthält allgemeine Informationen des Dokuments und hat folgende Form:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
```

Abb. 4.7: XML Deklaration

Falls eine Deklaration hinzugefügt wird, so muss diese das Versions-Attribut enthalten, während die beiden anderen Attribute optional sind. Das Versionsattribut enthält die XML-Version, auf der das Dokument beruht. Zurzeit ist dies immer die Version 1.0. Das Attribut *encoding* enthält die in dem Dokument verwendete Zeichencodierung. Der Wert des Attributes muss der Name einer gültigen Zeichencodierung sein, wie z.B. *UTF-8* oder *UTF-16* für Unicode. UTF-8 und UTF-16 müssen laut Spezifikation von jedem Parser unterstützt werden und sind gleichzeitig die Voreinstellung. Das Attribut *standalone* hat entweder den Wert *yes* oder *no* und legt fest, ob dieses Dokument unabhängig von weiteren Definitionen ist oder ob im Dokument auf eine DTD (siehe Kap.4.2) verwiesen wird. Dieses Attribut ist primär als Hinweis für einen Parser gedacht.

Mit Hilfe von **Processing Instructions (PI)** (Verarbeitungsanweisungen) können beim Parsen eines XML-Dokuments Instruktionen an eine Applikation weitergereicht werden. Eine PI besteht aus einem einleitenen <?, dem Namen der Applikation, an die die Daten weitergereicht werden sollen, den eigentlichen Anweisungen und einem abschließenden ?> (→Abb. 4.8).

```
<? Applikationsname    Anweisungen  ?>
```

Abb. 4.8: XML Processing Instruction

Einige Zeichen, die zur XML-Syntax gehören, wie < oder &, können nicht direkt in PCDATA-Blöcken benutzt werden. Es gibt zwei Möglichkeiten, diese Zeichen zu verwenden: Entweder durch das einfache Ersetzen dieser Zeichen mit Escape-Sequenzen oder durch die Einbettung in einen sogenannten CDATA-Block. In XML sind folgende Escape-Sequenzen vordefiniert:

- Das Ampersand (&): **&**
- Das 'kleiner' Zeichen (<): **<**
- Das 'größer' Zeichen (>): **>**
- Einfaches Anführungszeichen ('): **'**
- Doppeltes Anführungszeichen ("): **"**

In einem XML-Dokument können darüber hinaus auch eigene Escape-Sequenzen definiert werden.

Ein CDATA-Block wird mit <![$CDATA$[eingeleitet und mit]] > abgeschlossen. Innerhalb dieses Blocks ist jede Zeichenkette außer]] erlaubt (→Abb.4.9).

```
<![CDATA[ daten ]]>
```

Abb. 4.9: Definition einer CDATA-Section

4.2 Gültige Dokumente: DTDs/XML-Schema

Wohlgeformte Dokumente erlauben eine beliebige Struktur und einen beliebigen Inhalt, sofern diese der allgemeinen Syntax von XML folgen. Die Vorgabe einer Grammatik, die weitere Dokumenteigenschaften vorgibt, kann aber aus verschiedenen Gründen sinnvoll sein. Diese Einschränkungen können sich z.B. auf die gültigen Elementnamen (das sogenannte *Vokabular*) und die hierarchischen Beziehungen zwischen Elementen beziehen. Einschränkungen können aber auch auf die Daten selbst definiert werden, indem der Datentyp oder die Syntax eingeschränkt werden.

Eine formale Beschreibung des Inhalts und der Struktur ermöglicht es, den Aufbau des Dokuments anderen Anwendern detailliert mitzuteilen, die dann in der Lage sind, selbst XML-Dokumente gleicher Grammatik zu erstellen. Zusätzlich erlaubt es in Anwendungen, die diese strukturierten XML-Dokumente benutzen, ein bestimmtes Format des Dokuments vorauszusetzen. Denn mit Hilfe solcher Definitionen kann automatisch verifiziert werden, ob alle zwingend notwendigen Elemente vorhanden sind und keine unerwarteten Elemente im Dokument verwendet werden. Darüber hinaus sind die Baumstruktur und die Element-Attribute bekannt.

4.2.1 DTDs

Der schon mit SGML entwickelte und auch von XML berücksichtigte Ansatz heißt DTD (Document Type Defintion). DTDs erlauben es einen *Dokumenttypen* zu definieren, indem das Vokabular und die Struktur definiert werden. Ein Dokument, das den im Dokumenttypen definierten Anforderungen entspricht, heißt *Dokumentinstanz.*

DTDs verhindern, dass Anwender fehlerhafte XML-Dokumente produzieren. Mit DTDs ist es außerdem möglich für im XML-Dokument fehlende Attribute vordefinierte Werte zu setzen.

Die wesentlichen Deklarationen einer DTD sind *DOCTYPE* für die Verbindung eines XML-Dokuments zur zugehörigen DTD, *ELEMENT*, um den Aufbau eines Elementes, und *ATTLIST*, um die Attribute eines bestimmten Elementes zu beschreiben. Diese werden nun genauer betrachtet.

- Mit einer DOCTYPE-Deklaration wird aus einem XML-Dokument auf die zugehörige DTD verwiesen.
- In einer ELEMENT-Deklaration wird der Elementname angegeben und der Inhalt dieses Elements beschrieben. Die Beschreibung ist entweder eine allgemeine Aussage, in der eine vordefinierte Kategorie angegeben wird, z.B. beliebiger Inhalt, nur Text, nur Elemente, Text+Elemente oder ein leeres Element (außer Attribute), oder es wird ein Inhaltsmodell (*content model*) angegeben. In diesem wird in Klammern eine Liste von Kindelementnamen und/oder "#PCDATA" für Text angegeben. Für die Kindelemente kann bestimmt werden, ob sie nur in einer bestimmten Reihenfolge (sequence) gültig sind oder ein beliebiges Element einer Liste (choice) gültig ist. Diese Angaben können auch zu komplexeren Ausdrücken verschachtelt werden. Zusätzlich lassen sich grobe Angaben über die erlaubte Kardinalität einzelner Elemente machen, diese sind allerdings auf folgende Kategorien beschränkt: genau einmal, 0- bis 1-mal, beliebig häufig oder mindestens einmal.
- Eine ATTLIST besteht aus dem Namen des Elements, für das die Attribute festgelegt werden, einer Auflistung der Namen der erlaubten Attribute, einer Angabe über den Typ des Attributwertes und die Festlegung, ob dieses Attribut obligatorisch, optional oder auf einen Wert festgelegt ist. Für die Typangabe können nur vordefinierte Typen benutzt werden, zu denen u.a. die folgenden gehören: CDATA für beliebigen Text, ID und IDREF [1] oder Aufzählungen (enumerations), in denen eine Menge möglicher Werte definiert sind.

DTDs haben Einschränkungen, die das W3C bewogen haben einen weiteren Standard zur Dokumentbeschreibung einzuführen: *XML-Schema* (nachfolgend auch einfach Schemata oder (engl.) Schema genannt), die am 2. Mai 2001 als Empfehlung des W3C verabschiedet wurden [XMLb] [XMLc]. Die Tabelle 4.1 zeigt die Einschränkungen von DTDs und die Möglichkeiten, die Schemata bieten.

Ein fundamentaler Unterschied zu DTDs ist, dass Schemata XML-Dokumente sind und somit mit den gleichen Tools bearbeitet werden können wie die XML-Dokumentinstanzen selbst. Mit DTDs kann nur begrenzt die Kardinalität von Elementen ausgedrückt werden, außerdem schließen sich die Definition einer Auswahl aus mehreren Elementen und von gemischten Inhalten gegenseitig aus. Diese Einschränkungen sind mit Schemata behoben. DTDs

[1] ID und IDREF sind XML-Standardtypen, die die Überprüfung von Eindeutigkeiten und Referenzierungen ermöglichen, falls diese von dem benutzten Parser unterstützt werden. Ein Attributwert vom Typ ID muss eindeutig im ganzen Dokument sein, zu einem Attributwert vom Typ IDREF muss ein Element mit einem entsprechendem ID-Attribut existieren.

Eigenschaft	DTD	XML Schema
Syntax	EBNF+pseudo XML	XML 1.0
Tools	existierende (komplexe) SGML-Tools	so gut wie alle XML-Tools
DOM Support	nein	ja, da XML
Strukturbeschreibung	eingeschränkt	detailliert
Datentypbeschreibung	eingeschränkt	detailliert
Scope	nur global	global und lokal
Vererbung	nein	ja
Erweiterbarkeit	nein	unbegrenzt auf XML-Basis
Historische Einschränkungen	ja, da von SGML übernommener Standard	keine
Zahl der Vokabulare pro Dokument	maximal 1	beliebig - Basierend auf Namespaces
Dynamische Schemata	nein	ja, können zur Laufzeit verändert werden

Tabelle 4.1: Vergleich zwischen DTDs und Schemata (entnommen aus [Hun00])

unterstützen nur wenige Standardtypen (String und ein paar wenige mehr), während Schemata alle gängigen modernen Datentypen unterstützen und die Definition eigener Typen, inklusive Vererbung und Beschreibung des Gültigkeitsbereichs, erlauben.

Der Begriff *gültiges Dokument* bezieht sich im Allgemeinen nur auf die Validierung durch eine DTD. Im Rahmen dieser Diplomarbeit ist damit auch die Validierung durch Schemata gemeint.

Schemata haben zwei zentrale Bereiche, die in getrennten Empfehlungen vom W3C verabschiedet wurden: *Strukturen* und *Datentypen*. In den folgenden Abschnitten wird ein kurzer Überblick über die grundlegenden Elemente und Beschreibungsmöglichkeiten von Schemata gegeben. Eine ausführliche Besprechung findet man z.B. in [Hun00], wobei dort noch einige Syntax-Abweichungen gegenüber der endgültigen Empfehlung des W3C [XMLb]/[XMLc] zu beachten sind.

4.2.2 XML Schema: Datentypen

In XML-Schema definierte Datentypen werden, wie in Abb.4.10 dargestellt, klassifiziert. Primitive Datentypen bilden die Basis für alle weiteren Typen. Zu den in XML-Schema definierten primitiven Datentypen gehören z.B. *string, boolean, float, binary, ID, IDREF*. Abgeleitete Typen sind entweder schon in Schema vordefiniert (built-in) - dazu gehören z.B. *Name* (gültiger XML-Name), *integer*, *negativeInteger*, sowie *date* und *time* - oder sie sind vom Benutzer definiert (user-derived). Eine weitere Unterscheidung wird zwischen atomaren

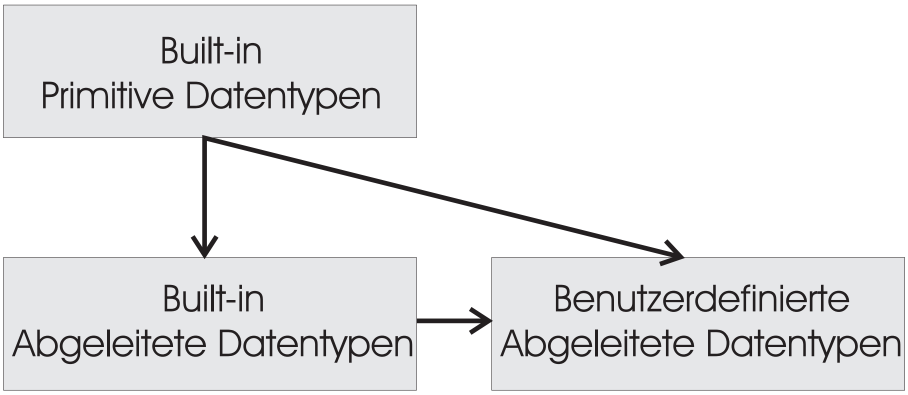

Abb. 4.10: Klassifizierung von Schema Datentypen

Datentypen (atomic datatypes), die einen Wert haben und unteilbar sind, und Listendatentypen (list datatypes), die eine Liste eines atomaren Datentypen darstellen, getroffen.

Die Eigenschaften von Datentypen werden durch Facetten beschrieben. Die fünf **fundamentalen Facetten** beschreiben die semantischen Eigenschaften eines Datentyps. Dies sind im Einzelnen:

- Gleichheit (equal) legt grundlegende Eigenschaften wie Identität oder Kommutativität fest.
- Ordnung (ordered) legt die Transitivität des Datentypen fest.
- Grenzen (bounded) definieren obere und untere Grenzen für Datentypen
- Kardinalität (cardinality) legt die Eigenschaften des Wertebereichs fest: endlich, abzählbar unendlich oder nicht-abzählbar unendlich.
- Numerisch (numeric) legt fest, ob es ein numerischer Datentyp ist, und kann wahr oder falsch sein

Einschränkende Facetten (Constraining Facets) begrenzen die Wertemenge eines abgeleiteten Datentyps. Dies kann entweder eine lexikalische Eingrenzung oder eine Einschränkung des Wertebereichs sein.

- *length, minLength, maxLength* schränken die Länge von strings oder davon abgeleiteter Typen ein.
- Mit *pattern* lässt sich die Menge regulärer Ausdrücke definieren, die die Wertemenge eines Datentyps bilden.
- Mit *enumeration* kann eine Wertemenge auf bestimmte Werte eingeschränkt werden.

- *whiteSpace* kann den Wert preserve, collapse oder replace haben und beschreibt den Umgang mit Zeichen, die zum Whitespace gehören.
- *minExclusive, maxExclusive, minInclusive, maxInclusive* beschränken den Wertebereich nach oben oder unten.
- *totalDigits, fractionDigits* geben die erlaubte Gesamtzahl der Ziffern bzw. die erlaubten Nachkommastellen einer Zahl an.

Neue Typen können auf drei verschiedene Arten von Basistypen oder bereits abgeleiteten Typen definiert werden:

- Der neue Datentyp kann mit Facets einen definierten Datentyp einschränken (*derivation by restriction*).
- Der neue Datentyp kann eine Liste mit Elementen eines Datentypen sein (*derivation by list*).
- Der neue Typ kann die Vereinigungsmenge mehrerer anderer Datentypen sein (*derivation by union*).

4.2.3 XML Schema: Strukturen

Mit Hilfe der folgenden Deklarationen lassen sich Elemente, Attribute und die Dokumenthierarchie beschreiben.

Mit **einfachen Typen** werden die Inhalte von Attributen und von PCDATA beschrieben. Es sind entweder eingebaute Typen oder werden von diesen, wie im Kap.4.2.2 gezeigt, durch Einschränkung, Listengenerierung oder Vereinigung abgeleitet.

Komplexe Typen erlauben eine Beschreibung des *content models* von Elementen, im Einzelnen die erlaubten Kindelemente, deren Kardinalität und Kombinationsmöglichkeiten. Darüber hinaus beschreiben sie erlaubte Attribute und deren Typ. Außerdem kann festgelegt werden ob diese obligatorisch oder optional sind oder ob sie einen vordefinierten oder festen Wert haben. Das folgende Beispiel definiert einen komplexen Typ, der entweder ein Element *Name* enthält oder die Folge der folgenden Elemente: ein Element *Vorname*, ein optionales Element *Beiname* und genau ein Element *Nachname*. Wie aus dem Beispiel zu ersehen ist, lässt sich mit dem Element *choice* eine Auswahl von Elementen definieren, die jeweils in einem Kindelement stehen. Zudem kann mit *sequence* eine verbindliche Folge von Elementen bestimmt werden. Diese Elemente lassen sich beliebig verschachteln. Die Attribute *minOccurs* und *maxOccurs* definieren die Ober- und Untergrenze der Anzahl, mit der das jeweilige Element im XML-Dokument vorhanden sein darf.

```
<complexType>
  <choice>
   <element name="Name" type="Text" minOccurs="1" maxOccurs="1"/>
   <sequence>
     <element name="Vorname" type="Text" minOccurs="1" maxOccurs="1"/>
     <element name="Beiname" type="Text" minOccurs="0" maxOccurs="1"/>
     <element name="Nachname" type="Text" minOccurs="1" maxOccurs="1"/>
   </sequence>
  </choice>
</complexType>
```

Ier Inhalt von <element> beschreibt den eigentlichen Aufbau. Für ein Element, das keine Attribute oder Kindelemente enthält, kann der Typ entweder direkt referenziert werden

```
<element name="name" type="type"/>
```

oder in einem *simpleType*-Kindelement definiert werden:

```
<element name="name">
  <simpleType>
     ...
  </simpleType>
</element>
```

Elemente, die Attribute und/oder Kindelemente haben, werden durch einen komplexen Typen beschrieben

```
<element name="name" minOccurs="int" maxOccurs="int">
  <complexType>
     ...
  </complexType>
</element>
```

In einem **Attribut**-Element wird ein einzelnes Attribut beschrieben, das Teil eines Elements sein kann. Dies kann als Kindelement eine Anmerkung (siehe unten) sowie einen einfachen Typen haben. Fehlt das *simpleType*-Kindelement, so unterliegt der Wert des Attributs keinerlei Einschränkung. Das Beispiel zeigt die Definition eines Attributes *wert*, das einen Wert zwischen 1 und 50 annehmen kann. Zusätzlich wird in diesem Beispiel ein vordefinierter Wert (30) deklariert, der von einem validierenden Parser gesetzt wird, falls das Attribut im XML-Dokument fehlt.

```
<attribute name="wert" default="30">
  <simpleType base="positiveInteger">
    <minInclusive value="1"/>
    <maxInclusive value="50"/>
  </simpleType>
</attribute>
```

Da Kommentare nicht unbedingt von einem Parser weitergegeben werden müssen, sind in XML-Schema sogenannte **Anmerkungen (annotations)** vorgesehen. Diese können erklärende Zusätze enthalten, die entweder für den Anwender oder eine Applikation gedacht sein können und in einem <documentation> bzw. <appinfo> Kindelement untergebracht sind:

```
<simpleType>
  <annotation>
   <documentation>Textuelle Beschreibung </documentation>
   <appinfo>
    <speziellesElement>Spezielle Anweisungen</speziellesElement>
   </appinfo>
  </annotation>
</simpleType>
```

4.3 Namensräume

Jedes XML-Dokument kann seine eigene Menge von gültigen Tagnamen, also ein eigenes Vokabular definieren. Falls es nötig ist, in einem Dokument mehrere verschiedene Vokabulare zu benutzen, kann es Konflikte bei den Tagnamen geben. Dieses Problem wird behoben, indem jedes Vokabular einen eigenen Namespace zugewiesen bekommt. Namespaces sind am 14.01.1999 als Empfehlung vom W3C verabschiedet worden [XMLa].

Ein Namespace wird durch das zusätzliche Attribut **xmlns** in einem Element definiert. Der Wert des Attributs gibt den Namen des Namespaces an. Dieser sollte möglichst eindeutig sein, weshalb üblicherweise eine URL angegeben wird. Da der Name dadurch in der Regel recht lang und damit unhandlich wird, kann eine Abkürzung des Namespaces definiert werden, indem dem Attributnamen, getrennt durch einen Doppelpunkt, eine Abkürzung hinzugefügt wird, die dann in allen Kindelementen benutzt werden kann. Fehlt diese Abkürzung, ist dieser Namespace der vordefinierte für dieses Element und alle Elemente dieses Teilbaums (siehe unten). Im folgenden Beispiel werden zwei Namespaces definiert. Der erste hat den Namen *http://www.cs.uni-dortmund.de/liste*, der innerhalb des Teilbaums des XML-Dokuments mit *liste* angesprochen wird, der zweite entsprechend.

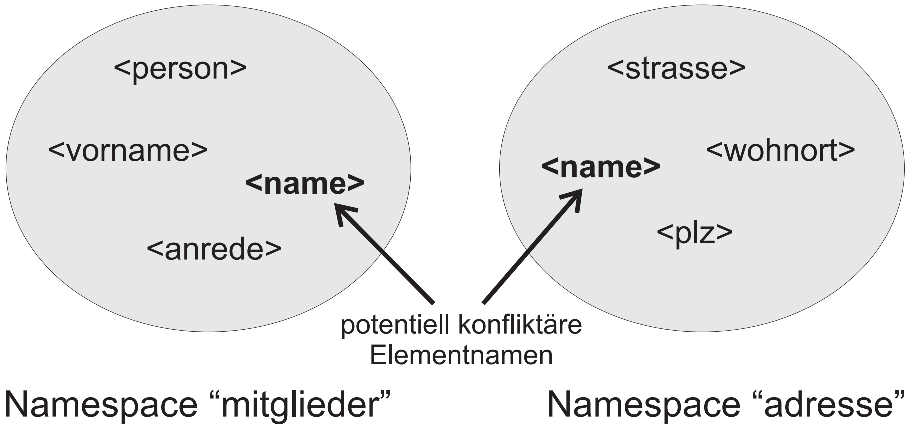

Abb. 4.11: Zwei Namespaces mit je einem getrennt definiertem Vokabular

```
<liste:hauptelement xmlns:liste="http://www.cs.uni-dortmund.de/liste"
                    xmlns:adresse="http://www.cs.uni-dortmund.de/adressen">
  <adresse:person/>
  <liste:person/>
</hauptelement>
```

Um nun bei einem Element den Namespace festzulegen, zu dem dieses gehört, wird dem Elementnamen, getrennt durch einen Doppelpunkt, der Name des Namespaces oder dessen Abkürzung vorangestellt. Im Beispiel gehört das erste *person*-Element zum Namespace
http://www.cs.uni-dortmund.de/adressen
und das zweite Element zum Namespace
http://www.cs.uni-dortmund.de/liste.
Ein Elementname, der den Namespace mit einschließt, wird *qualifizierter Name (qualified name)* genannt. Um nicht bei jedem Element den Namespace explizit angeben zu müssen, kann ein Namespace vordefiniert werden, indem die Abkürzung bei der Namespace-Definition weggelassen wird. Die folgende Definition ist also gleichbedeutend:

```
<hauptelement xmlns="http://www.cs.uni-dortmund.de/liste"
              xmlns:adresse="http://www.cs.uni-dortmund.de/adressen">
  <adresse:person/>
  <person/>
</hauptelement>
```

Der vordefinierte Name des Namespaces ist in dem Element und dem anhängenden Teilbaum gültig, es sei denn, in einem untergeordneten Teilbaum wird auf gleiche Weise ein neuer Namespace vordefiniert:

```
<hauptelement xmlns="http://www.cs.uni-dortmund.de/liste"
              xmlns:adresse="http://www.cs.uni-dortmund.de/adressen">
  <adresse:person/>
  <person/>
  <teilbaum xmlns="http://www.cs.uni-dortmund.de/subliste">
    <element1/>
    <element2/>
  </teilbaum>
</hauptelement>
```

Der vordefinierte Namespace für das Elemente *teilbaum* und dem anhängendem Teilbaum ist *http://www.cs.uni-dortmund.de/subliste*, während für alle anderen Elemente der vordefinierte Wert *http://www.cs.uni-dortmund.de/liste* ist.

Möchte man den Namespace für einzelne Elemente oder Teilbäume ganz aufheben, so kann dies durch die Definition eines leeren Namespaces geschehen:

```
<teilbaum xmlns="">    </teilbaum>
```

4.4 Standard-APIs für XML: SAX/DOM

Um Applikationen den Zugriff auf ein XML-Dokument zu ermöglichen, um diese zu manipulieren oder zu erweitern, haben sich im Umfeld von XML zwei Standards durchgesetzt. Das DOM (*Document Object Model*) ist eine Interfacebeschreibung des W3C [DOM]. DOM bietet sich an, wenn ein XML-Dokument eingelesen werden soll, und dieses verändert oder erweitert wird. Will man allerdings ein XML-Dokument nur einlesen und spezifische Daten extrahieren, gibt es eine Alternative zu DOM, die Simple API for XML (SAX). Während DOM ein baum-basierter (engl. *tree-based*) Ansatz ist, das XML-Dokument komplett einliest und eine interne Baumstruktur aufbaut, ist SAX ein ereignis-basierter (engl. *event-based*) Ansatz, der das XML-Dokument sequentiell durchläuft und die einzelnen Bestandteile (Elemente, Whitespace, Header) an die Applikation weiterreicht. Falls der Aufbau einer komplexeren Datenstruktur nötig sein sollte, bleibt dies der Applikation überlassen.

4.4.1 Das Document Object Model (DOM)

DOM ist eine von der Programmiersprache und der Plattform unabhängige Interface-Definition. In einer Implementierung dieses Interfaces wird ein XML-Dokument komplett eingelesen und in eine interne Baumstruktur überführt. Dazu ein Beispiel:

```
<Person alter="35">
  <Vorname>Richard</Vorname>
  <Nachname>Müller</Nachname>
</Person>
```

Dieses XML-Dokument führt zu einer in Abb. 4.12 dargestellten internen Baumstruktur.

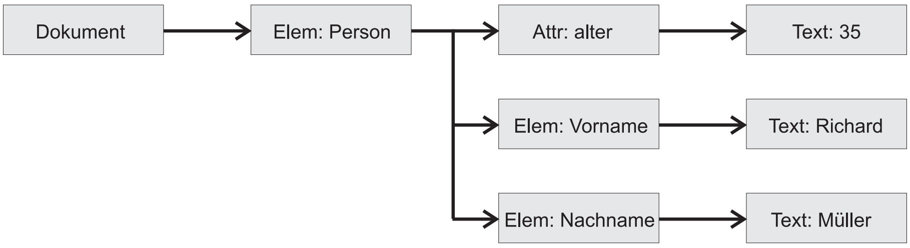

Abb. 4.12: Schematische Darstellung der internen DOM-Baumstruktur

Durch diese Baumstruktur, kann nun über ein standardisiertes Interface navigiert und Teile können manipuliert werden. So können z.B. Informationen herausgefiltert, Teile des Dokuments kopiert oder gelöscht, Attribute hinzugefügt oder geändert werden. Neben diesen Vereinfachungen der Bearbeitung von Dokumenten garantiert DOM die Wohlgeformtheit des aktuellen Dokuments, da unerlaubte offene oder ineinander verschachtelte Elemente nicht generiert werden können. Der Ablauf bei DOM ist in Abb. 4.13 dargestellt und erfolgt folgendermaßen: Die Applikation konfiguriert im DOM-Interface den Parser und stößt anschließend das Einlesen (parsen) des Dokuments an. Der Parser liest das Dokument komplett ein und baut die interne Baumstruktur auf. Auf diese kann dann über das DOM-Interface zugegriffen werden.

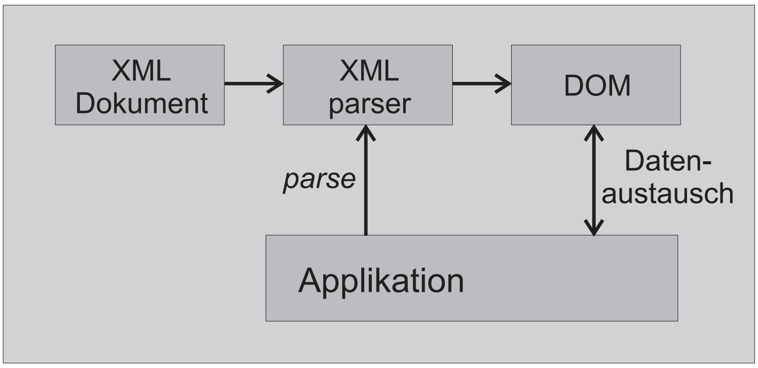

Abb. 4.13: Schematischer Aufbau von DOM (aus [Hun00])

Das DOM-Interface teilt sich in das grundlegende (*fundamental*) und das erweiterte (*extended*) Interface. Die zentralen Komponenten des grundlegenden Interfaces sind *Node*, *Document* und *Element*. Das Basis-Interface Node stellt alle Methoden bereit, die auf die verschiedenen Komponenten eines XML-Dokuments, wie z.B. Element, Attribut, Kommentar oder Textknoten, gemeinsam anwendbar sind. Es ermöglicht auch den Zugriff auf die Kindknoten, inklusive des Hinzufügens und Löschens einzelner Knoten. Fast alle anderen Interfaces erben von dem Node-Interface. Das Document-Interface repräsentiert das komplette Dokument und bietet die Möglichkeit, zur Laufzeit einzelne Nodes hinzuzufügen. Das Element-Interface erlaubt es, auf die Komponenten eines XML-Elementes, wie z.B. Attribute, zuzugreifen. Außerdem bietet DOM spezielle Interfaces für spezielle Komponenten, wie z.B. Attribut-, CDATA-,Text- und Kommentarknoten (*Attr*,*CharacterData*,*Text* und *Comment*). Ein weiterer DOM-Teil vereinfacht die Arbeit mit Knotenmengen: *NodeList* und *NamedNodeMap*. Knotenmengen können z.B. durch die Zusammenfassung sämtlicher Kindknoten, oder (was auch vom DOM-Interface direkt vorgesehen ist) eine Zusammenstellung aller Elemente des gleichen Namens entstehen. Für Fehler während des Parsens oder der Bearbeitung ist ein Interface zur Ausnahmebehandlung definiert (*DomException*).

Im erweiterten Interface werden XML-spezifische Komponenten berücksichtigt. Beispiele hierfür sind *CDATASection*, *ProcessingInstruction*, *DocumentType*, *Notation*, *Entity* und *EntityReference* berücksichtigt.

Zu DOM gibt es mittlerweile zwei Empfehlungen (DOM Level-1 und Level-2), eine dritte ist in Vorbereitung. Mit DOM Level-2 werden z.B. Namespaces unterstützt. DOM Level-3 sieht eine Unterstützung für XPath vor (siehe 4.5.2).

4.4.2 Simple API for XML: SAX

Falls nur einzelne Daten aus einem Dokument zu extrahieren sind und das Dokument nicht weiter manipuliert werden muss, so bietet sich die Verwendung von SAX an.

Bei SAX wird das Parsen von der Applikation in einem *Parser*-Objekt angestoßen, und diese bekommt für die einzelnen Bestandteile, wie Start- und Endtag, Dokumentenheader, PCDATA usw., jeweils ein Event, das von einem vorher eingerichtetem *DocumentHandler* bearbeitet wird. Dieser kann von der Basisklasse *HandlerBase* geerbt und an die Anforderungen der Anwendung angepasst werden oder er wird vollständig selbst implementiert.

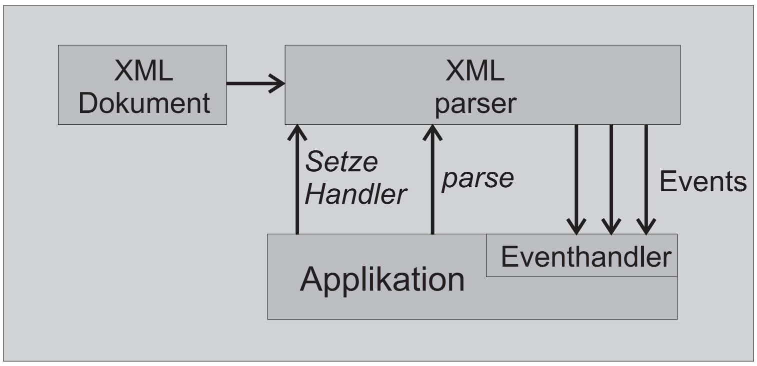

Abb. 4.14: Schematischer Aufbau von SAX (aus [Hun00])

Der Documenthandler hat standardisierte Methoden. Die wichtigsten davon sind:

- **startDocument:** wird zu Beginn des Dokuments aufgerufen.
- **endDocument:** wird am Ende des Dokuments aufgerufen.
- **startElement:** wird bei einem geparstem Starttag aufgerufen und übermittelt den Namen und die Attribute.
- **endElement:** wird bei einem geparstem Endtag aufgerufen.
- **characters:** wird bei gefundenen PCDATA aufgerufen.

Innerhalb der Implementierung der *startElement*-Behandlungsmethode wird für die verschiedenen Elementnamen über einen switch die entsprechende Behandlungsroutine implementiert. Auch die zu dem jeweiligen Element gehörenden Attribute werden der Methode als *AttributeList*-Objekt direkt übergeben und können dort ausgewertet werden.

```
void startElement(string name,  AttributeList atts)
{
  switch(name)
  {
    case "Person":
      /*Handling für Person*/
      /*inkl. Attributbehandlung*/
      break;

    case "Name":
      /*Handling für Name*/
      /*inkl. Attributbehandlung*/
      break;

    default:
      break;
  }
}
```

Die verkürzte Schreibweise für leere Elemente wird von SAX genauso behandelt, als ob es sich um getrennte Start- und Endtags handelte. Die folgende Ausdrücke sind seitens der Applikation nicht zu unterscheiden:

```
<Person/>
<Person></Person>
```

Wie DOM hat auch SAX definierte Exceptions, die z.B. bei einem Syntaxfehler während des Parsens ausgelöst (*SaxParseExeception*), oder vom Anwender bei semantischen Fehlern erzeugt werden können (*SaxException*). Ein sogenanntes *Locator*-Objekt ist ebenfalls vom Interface vorgesehen, das Informationen über die aktuelle Position im XML-Dokument, also z.B. Zeilen und Spaltennummer, liefert.

4.4.3 Vor- und Nachteile von SAX und DOM

DOM bietet alle Möglichkeiten zur Manipulation eines XML-Dokuments, da dessen Struktur direkt bearbeitet werden kann. Ein erheblicher Nachteil von DOM ist, dass durch das komplette Einlesen der Ressourcenaufwand sehr groß ist. Geht es nur um das Extrahieren von spezifischen Informationen aus einem XML-Dokument, so bietet sich dafür der Einsatz von SAX an: Da Elemente hier einzeln eingelesen werden, ist der Ressourcenaufwand gering. Zusätzlich ist das Parsen deutlich schneller, da der Aufbau der Baumstruktur entfällt. Gerade in den Fällen, wo nur ein kleiner Teil des Dokuments benötigt wird, ist der Einsatz von SAX zu empfehlen.

Zusammenfassend lässt sich festhalten, dass sich DOM und SAX wegen der jeweiligen Stärken und Schwächen gut ergänzen. Die Entscheidung für die eine oder andere API hängt im Wesentlichen von der zu bearbeitenden Problemstellung ab.

4.5 Linking in XML

Ein elementarer Teil von HTML ist die Möglichkeit, Dokumente mit Hyperlinks zu verknüpfen. Das W3C hat zwei wichtige Ansätze geschaffen, um das Instrument des Hyperlinks in XML-Dokumenten zu benutzen. Mit **XLink** (Empfehlung vom 27.06.2001 [XLi]) können mehrere XML-Dokumente miteinander verknüpft werden. **XPath** (Empfehlung vom 16.11.1999 [XPa]) und **XPointer** (Kandidat für eine Empfehlung vom 11.09.2001 [XPo]) bieten die Möglichkeit, Teile eines XML-Dokuments zu referenzieren.

4.5.1 XLink

XLinks sind das Pendant zu HTML-Hyperlinks, bieten aber darüber hinaus erweiterte Möglichkeiten wie die Beschreibung bidirektionaler Links und die Verknüpfung ganzer Gruppen von Ressourcen. Ein einzelner Link wird durch eine Gruppe von Attributen aus dem XLink-Namespace innerhalb eines Elementes definiert. Dabei legt das Attribut *xlink:type* den Elementtypen fest.

```
<EinfacherLink xlink:type="simple"
               xlink:href="ls12.cs.uni-dortmund.de/address">
  Text des Links
</EinfacherLink>
```

Bei XLink wird zwischen **einfachen** und **erweiterten** Links unterschieden, die mit dem *xlink:type*-Attribut durch den Attributwert *simple* bzw. *extended* deklariert werden. Während einfache Links den aus HTML bekannten <HREF> gleichen, können mit erweiterten Links deutlich stärkere Verknüpfungsaussagen, wie mehrfach- und bidirektionale Vernüpfung, definiert werden. Dies geschieht, indem eine Menge von Ressourcen deklariert wird, die jeweils einer Klasse zugeordnet werden. Zwischen diesen Klassen wird dann eine gerichtete Verknüpfung definiert. Dazu werden XLink-Kindelemente von folgenden Typen hinzugefügt:

- **locator:** zur Beschreibung einer externen Ressource, die per *href*-Attribut spezifiziert wird.
- **resource:** zur Beschreibung einer lokalen Ressource, die direkt angegeben wird.
- **arc** beschreibt mittels den Attributen *from* und *to* die Verknüpfung zweier Ressource-Klassen.

Das folgende Beispiel definiert mehrere externe Ressourcen und verbindet schließlich jedes Element mit dem label *group1* mit jedem Element mit dem label *group2*, also im Beispiel insgesamt vier Verknüpfungen. Diese sind zunächst unidirektional und können durch Hinzufügen eines weiteren *arc*-Elementes mit umgekehrter Richtung als bidirektionale Verknüpfung definiert werden. Zusätzlich können auch noch andere Gruppen miteinander verknüpft werden.

```
<ErweiterterLink xlink:type="extended">
  <elem xlink:type="locator" xlink:label="group1" xlink:href="target1"/>
  <elem xlink:type="locator" xlink:label="group1" xlink:href="target2"/>
  <elem xlink:type="locator" xlink:label="group2" xlink:href="target3"/>
  <elem xlink:type="locator" xlink:label="group2" xlink:href="target4"/>
  <elem xlink:type="locator" xlink:label="group3" xlink:href="target5"/>
  <elem xlink:type="locator" xlink:label="group3" xlink:href="target6"/>
  <elem xlink:type="arc" xlink:from="group1" xlink:to="group2"/>
</ErweiterterLink>
```

Die einzelnen XLink-Elementtypen lassen sich durch Attribute genauer beschreiben, die je nach Elementtyp obligatorisch, optional und nicht erlaubt sind (siehe dazu Tabelle unten). Ein einfacher Verweis oder eine externe Ressource werden mit dem *xlink:href*-Attribut beschrieben. Um die Semantik eines Elementtyps zu beschreiben, kann *xlink:role* für die maschinelle Bearbeitung und *xlink:title* für eine vom Benutzer lesbare Form benutzt werden. Für die Beschreibung des Verhaltens des Links gibt es *xlink:actuate*, das den Zeitpunkt festlegt, wann ein Link aktiviert werden soll: automatisch (*onLoad*), durch den Benutzer (*onRequest*) oder nicht definiert (*undefined*). Das Attribut *xlink:show* legt die Art und Weise der Darstellung fest: Das Zieldokument wird in einem separatem Fenster dargestellt (*new*), das aktuelle Dokument wird komplett ersetzt (*replace*), nur der Link wird ersetzt (*embed*) oder das Verhalten ist nicht weiter definiert (*undefined*). Im letzten Fall bleibt die Darstellung der Applikation überlassen.

Neben den oben dargestellten XLink-Elementtypen gibt es noch das Element **title**, das als Alternative zum *title*-Attribut zur Verfügung steht, falls weitere Kindelemente hinzugefügt werden müssen.

Die folgende Tabelle gibt noch einmal einen Überblick über XLink-Elementtypen und die erlaubten Attribute (X=obligatorisch O=optional):

	simple	extended	locator	arc	resource	title
type	X	X	X	X	X	X
href	O		X			
role	O	O	O		O	
arcrole	O			O		
title	O	O	O	O	O	
show	O			O		
actuate	O			O		
label			O		O	
from				O		
to				O		

4.5.2 XPath/XPointer

Vielfach ist es nötig, auf einzelne Teile eines Dokuments zu verweisen. Für diesen Zweck hat das W3C am 16.11.1999 XPath als Empfehlung verabschiedet [XPa]. Ähnlich wie DOM, wird das XML-Dokument als Baumstruktur gesehen, dessen Knoten in sieben Kategorien aufgeteilt sind: Root-, Element- und Attributknoten sowie spezielle Komponenten wie Text-, Namensraum-, PI- und Kommentarknoten. XPath beschreibt einen Punkt im Dokument durch *Lokationspfade*. Diese können absolut sein oder relativ zu einem *Kontextknoten*. Ein Lokationspfad besteht aus *Lokationsschritten*, die jeweils durch ein Slash (/) voneinander getrennt werden. Ein absoluter Pfad beginnt im Gegensatz zu einem relativen Pfad mit einem Slash:

```
/schritt1/schritt2/...                  // Absoluter Lokationspfad
schritt1/schritt2                       // Relativer Lokationspfad
```

Jeder Schritt besteht aus der *Achse*, auf der man sich bewegt und einem *Knotentest* und liefert alle Knoten zurück, die den Knotentest auf der angegebenen Achse bestehen. In XPath gibt es insgesamt 13 verschiedene Achsen, u.a. folgende: alle Nachkommen (descendant), alle direkten Kindknoten (child), alle Vorgängerknoten (ancestor), alle Geschwister (sibling), der Elternknoten (parent), alle Attribute (attribute) oder der Knoten selbst (self). Der Knotentest kann entweder der Test auf einen bestimmten Knotennamen oder auf den Knotentyp

(Text/Comment/PI/Node) sein. Die Achsenbeschreibung und der Knotentest werden durch zwei Doppelpunkte (::) voneinander getrennt. Beispiele für einen Lokationsschritt:

- *child::Vorname* liefert alle Kindknoten des Kontextknotens mit dem Namen "Vorname" zurück.
- *descendant::Name* liefert alle Knoten mit dem Namen "Name" zurück, die gleichzeitig Nachfolger des Kontextknotens sind.
- *attribute::** liefert alle Attribute des Kontextknotens zurück.

Alle in einem Lokationsschritt ermittelten Knoten sind Kontextknoten für den nächsten Lokationsschritt. Die Lösung des gesamten XPath-Ausdrucks ist die Vereinigungsmenge aller Knoten, die in allen letzten Lokationsschritten (auch über verschiedene Zwischen-Kontextknoten) ermittelt werden.

Optional kann jeder Knotentest noch mit einem Prädikat versehen werden, um die Zahl der Zwischenkontextknoten einzuschränken. Als Prädikate können verschiedene Funktionen einer vorgegebenen Bibliothek benutzt werden, dazu gehören z.B.

- *position* um die Position eines Knotens in einer Knotenmenge zu bestimmen,
- *count* um die Gesamtzahl der Knoten zu testen, oder
- *last* um nur den letzten Knoten zu nehmen.

Darüber hinaus gibt es String- und Boolean-Funktionen. Jedes Prädikat wird in eckigen Klammern an den Knotentest angehängt. Innerhalb eines Lokationsschrittes können mehrere Prädikate definiert werden. Beispiele hierfür sind

- *child::Arg*[*position()=2*] liefert das zweite Kindelement mit dem Namen *Arg* zurück
- *child::Arg*[*position()=2*][*attribute::key*] liefert das zweite Kindelement mit dem Namen *Arg* zurück, falls dies auch ein Attribut mit dem Namen "key" hat.
- *child::Arg*[*position()=2*][*attribute::key="15"*] liefert das zweite Kindelement mit dem Namen *Arg* zurück, falls dies auch ein Attribut mit Namen *key* und dem Wert *15* hat.

XPath bietet verkürzte Schreibweisen an, wie z.B.:

- Ist keine Achse angegeben, so ist *child* die Voreinstellung.
- Die Attributachse kann mit @ abgekürzt werden.
- Statt [*position()=x*] kann einfach [*x*] geschrieben werden.

- Um alle Knoten eines bestimmten Namens von allen Nachfolgern inkl. Kontextknoten unabhängig von der Pfadtiefe zu bekommen, kann ein Doppelslash (//) benutzt werden.

Folgende Ausdrücke sind also gleichbedeutend:

```
child::Personen/child::Namen[position()=2]/child::Vorname/attribute::alter
Personen/Namen[2]/Vorname/@alter
```

Beide wählen von allen Kindelementen des Kontextknotens mit dem Namen *Personen* jeweils das zweite Kindelement mit dem Namen *Namen* aus. Von allen dadurch ermittelten Zwischenknoten werden alle Kindelemente mit dem Namen *Vorname* und von diesen wiederum jeweils das Attribut *alter* ermittelt.

Eine Erweiterung, die auf XPath aufbaut, heißt XPointer. Hiermit sollen Teile eines XML-Dokuments selektiert werden. Dies geschieht z.B. über die Auswahl eines echten Knoten, der die Wurzel eines Teilbaumes darstellt. Allerdings soll es mit XPointern z.B. auch möglich sein, nur Teile einer Zeichenkette zu selektieren, wozu XPath-Ausdrücke allein nicht ausreichend sind. Deshalb können mit XPointer Punkte (points) im Dokument bestimmt und mit zwei Punkten Bereiche (ranges) definiert werden. XPointer hat leider noch den Status eines Kandidaten zur Empfehlung, so dass an dieser Stelle auf eine detailliertere Darstellung verzichtet wird.

4.6 XML-Transformation: XSL

Mit XSL (eXtensible Stylesheet Language [XSL]) werden Stylesheets generiert, die von einer XML-Engine dazu benutzt werden, ein XML-Dokument in ein anderes Format zu transformieren (→Abb.4.15).

Das Quelldokument heißt *Quellbaum (source tree)*, das Zieldokument *Zielbaum (target tree)*. Zwei Sprachen auf Basis von XSL sind bislang vollständig definiert worden:

- XSLT (eXtensible Stylesheet Language for Transformations) für die Transformation von Dokumenten

- XSL-FO (XSL Formatting Objects), um die Ausgabe für XML Dokumente zu beschreiben

Die zentrale Komponente von XSL sind Templates.

```
<xsl:template match="Vorname">Das Element Vorname gefunden!</xsl:template>
```

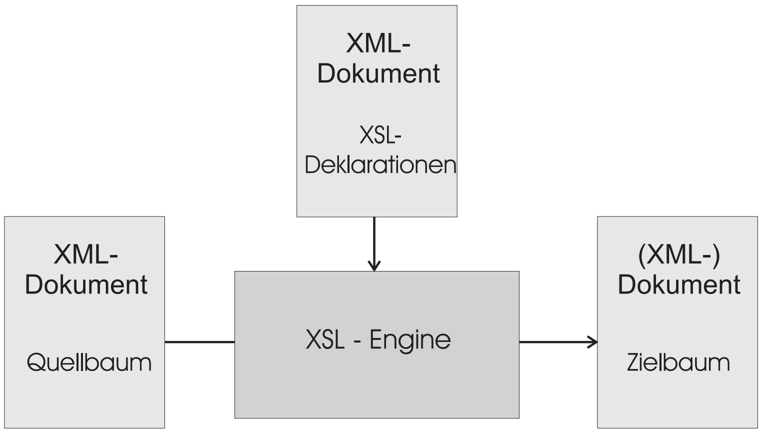

Abb. 4.15: XSL Transformationen

Die wesentlichen Bestandteile für die Deklaration eines Templates sind das XSL-Element *xsl:template*, innerhalb dessen das Template definiert wird, das *match*-Attribut, das das Pattern definiert, nach dem im Quellbaum gesucht wird, und der Inhalt des Elementes, in dem angegeben wird, was auf dem Zielbaum ausgegeben wird. Im obigen Beispiel ist dies einfacher Text, i.d.R. sind dies allerdings komplexe Anweisungen. Der Wert des *match*-Attributs ist ein XPath-Ausdruck. In einem XSLT-Dokument können beliebig viele Templates definiert werden. Die Ausführung beginnt mit dem Template, das den Rootknoten als Patterns definiert hat. Fehlt ein solches, so gibt es zwei vordefinierte Templates, die für jeden Knoten ein Template suchen und den Inhalt aller Attribut- und Textknoten ausgeben. Neben dem <xsl:template>-Element sieht XSLT weitere Elemente für verschiedene Aufgaben vor. So können aus einem Template andere Templates aufgerufen werden, Werte von mit XPath spezifizierten Knoten ausgegeben werden, Elemente und Attribute dynamisch in den Zielbaum eingefügt werden oder bedingte Ausführungen definiert werden. Auch die Ausführung für jedes Element einer mit XPath beschriebenen Knotenmenge und das einfache Kopieren einer Knotenmenge sind durch spezielle Elemente möglich. Darüber hinaus wird noch die Möglichkeit geboten, eine Gruppe von Elementen automatisch zu sortieren.

Mit Hilfe dieser Elemente ist es nun möglich, für beliebige Komponenten des Quellbaums Templates zu definieren und die Regeln zu bestimmen, wie deren Inhalt in dem Zielbaum ausgegeben wird. XSLT eignet sich daher besonders in den Fällen, in denen die Elementstruktur modifiziert werden muss. Zwei Anwendungsbeispiele sind die Transformation eines XML-Dokuments in ein HTML-Dokument und die wechselseitige Überführung zweier XML-Vokabulare, die einen ähnlichen Inhalt, aber verschiedene Elementstrukturen haben.

4.7 Tools für XML

Für XML wurden und werden viele Tools für verschiedene Anwendungsbereiche entwickelt[2]. Einige von vielen Beispielen sind:

- Konverter, z.B. XML nach PDF, XML nach RTF oder Text nach XML,
- Browser,
- Editoren, z.B. emacs-Plugins oder Java-Applikationen,
- Editor-Generatoren,
- Diff & Mergetools,
- an XML-Dokumenten orientierte Kompressions-Programme oder
- spezifische Utilities für XSLT oder Schema.

Für diese Diplomarbeit ist die Ein- und Ausgabe von XML-Dokumenten ein zentraler Punkt. Die Ausgabe erfolgt direkt über einen Ausgabestrom. Für die Eingabe sind der Einsatz eines Standardparsers und der Zugriff über ein standardisiertes Interface sinnvoll. Eine Bibliothek sollte nach Möglichkeit SAX, DOM Level-2 und XML Schema unterstützen. Da XML stark am Internet orientiert ist, basieren viele APIs auf Java. Für GeLIR sollte die verwendete Bibliothek aber C++ unterstützen. Zudem sollte die Bibliothek plattform-übergreifend, zumindest aber SunOS und Linux unterstützen, und kostenlos sein.

Eine Bibliothek, die diese Anforderungen erfüllt, ist Xerces-C++ [Xer], das vom Apache XML-Projekt entwickelt wurde. Die Unterstützung für XML Schema ist noch nicht vollständig, die wesentlichen Komponenten sind aber schon implementiert. Darüber hinaus ist Xerces auf vielen verschiedenen Plattformen einsetzbar.

[2] Eine Übersicht gibt es unter [XMLd]

Kapitel 5

XeLIR

5.1 Anforderungen und Schnittstellen

Bei der Entwicklung der XeLIR-Datenstruktur stehen zwei Dinge im Vordergrund: Zum einen muss das Vokabular, d.h. die Elementehierarchie und die benötigten Attribute zur Informationsdarstellung, definiert werden, zum anderen müssen für die Daten und Strukturen mit XML-Schema nötige und vernünftige Einschränkungen gefunden werden, um ungültige XeLIR-Darstellungen zu verhindern oder zu erkennen. Bevor diese im Detail besprochen werden, gibt die Abb. 5.1 noch einmal einen Überblick über die Schnittstellen von XeLIR zu anderen Komponenten:

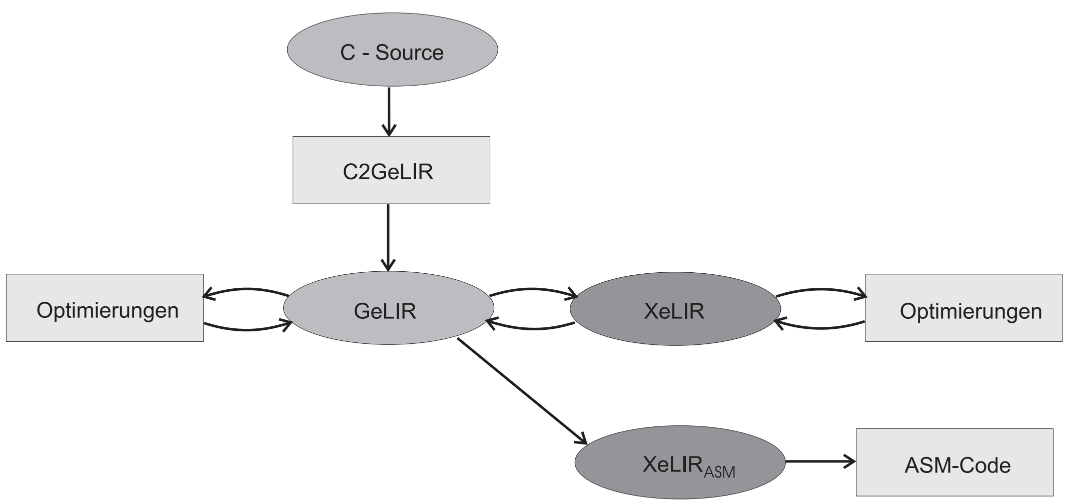

Abb. 5.1: Schnittstellen von XeLIR

- Die wichtigste Anforderung an XeLIR ist, Zustände aus GeLIR zu speichern und wieder einzulesen. Dies setzt für XeLIR voraus, dass spezielle Eigenschaften von GeLIR, wie eine Zielarchitekturbeschreibung, die Beschreibung von Parallelität oder komplexer Maschinenoperationen unterstützt werden. Ein Ziel ist außerdem, dass ein in einem XeLIR-Dokument gespeicherter Zustand deckungsgleich in GeLIR reproduziert werden kann. Darüber hinaus sollte das Textformat einen gut lesbaren Aufbau haben, um leicht manuelle Änderungen durchführen zu können.

XeLIR als GeLIR-Austauschformat erlaubt u.a. folgende Einsatzmöglichkeiten:

- In der Entwicklungsphase neuer Optimierungen muss für Tests häufig erst ein bestimmter immer wiederkehrender Zustand in GeLIR erzeugt werden. Dies umfasst z.B. das Durchlaufen des kompletten Frontend und High-Level Standardoptimierungen und kann sich bis zu Low-Level Aufgaben wie Adresscodegenerierung hinziehen. Dieser Vorgang bringt immer wieder Verzögerungen während der Entwicklung mit sich, die sich durch das direkte Einlesen eines vorher gespeicherten Zustands verhindern lassen.
- Änderungen an der Architekturbeschreibung (für schnelles Prototyping) und der Programmdarstellung (für Handoptimierungen) können textuell durchgeführt werden.
- Falls mehrere Benutzer gemeinsam an verschiedenen Orten mit GeLIR arbeiten, so können Zwischenzustände einfach übermittelt werden. Dies erübrigt das häufig aufwendige oder sogar unmögliche Reproduzieren von Zwischenzuständen für den Empfänger.
- Zur Überprüfung von entwickelten Optimierungen können z.B. zwei XeLIR-Dokumente erzeugt werden, eins direkt vor und eins direkt nach einer Optimierung. Ein Vergleich der beiden Dokumente macht die Veränderungen transparenter und überprüfbarer, da schon durch einen einfachen Vergleich der Dokumente die signifikanten Stellen ermitteln lassen.

Einzelheiten zu den Konvertierungen GeLIR $\leftrightarrow$ XeLIR sind in Kap. 6.1 beschrieben.

- Da XML-Dokumente hierarchisch aufgebaut sind, bietet es sich an, diese Struktur für weitere Anwendungsbereiche zu nutzen. Mit Hilfe von Mustern (engl. patterns) kann in einem XeLIR-Dokument nach bestimmten Strukturen gesucht werden, um dann spezifische Transformationen vorzunehmen. Hier sind z.B. die folgenden Einsatzmöglichkeiten denkbar:

  - Peepholeoptimierungen: Es werden Paare von Mustern gebildet, die nach dem Verfahren *Suchen und Ersetzen* funktionieren. Das heißt eine mit einem definierten Suchmuster erkannte Struktur wird durch eine in einem zweiten Muster definierte Struktur ersetzt. Dies ist z.B. für algebraische Transformationen denkbar.
  - Schreiben von Assemblercode: Für jede in einem XeLIR-Dokument vorkommende (komplexe) Maschinenoperation wird ein Muster definiert und es werden zusätzlich Ausgabevorschriften angegeben, die aus den Informationen, die in dieser MO stehen, einen gültigen Assemblercode produzieren. Ein XeLIR Dokument wird

nun sequentiell durchlaufen. Dabei wird für jede Maschinenoperation nach einem passenden Muster gesucht und die dazugehörigen Ausgabevorschriften werden ausgeführt.

Diese Erweiterungen erlauben z.B. folgendes Szenario:

1. Erzeugung einer initialen GeLIR-Darstellung
2. Durchführung der Codegenerierung mit Registerallokation, Instruktionsauswahl, Instruktionsanordnung
3. Generierung eines XeLIR-Dokuments aus GeLIR
4. Durchführung von Peepholeoptimierungen auf dem XeLIR-Dokument mit Hilfe von definierten Mustern
5. Wiedereinlesen des XeLIR-Dokuments in GeLIR und Durchführung weiterer Optimierungen
6. Erzeugung eines neuen XeLIR-Dokuments
7. Schreiben eines gültigen Assemblerprogramms aus XeLIR mit Hilfe von Mustern

- Durch die Definition eines XML-Schemas lässt sich ein XeLIR-Dokument in begrenztem Umfang validieren. So wird es möglich, u.a. die Elementehierarchie und Kardinalität zu bestimmen, Typüberprüfungen durchzuführen und die Eindeutigkeit von Identifiern sicherzustellen. Der Einsatz von Schemata vereinfacht das Einlesen in GeLIR u.a. dadurch, indem sie gewährleisten, dass

  - die Datenstruktur alle nötigen Komponenten enthält,
  - bestimmte Komponenten eindeutig definiert sind und
  - das Dokument Mindestvoraussetzungen erfüllt, die die Implementierung des *XeLIR2GeLIR*-Konverters vereinfachen.

  In den folgenden Kapiteln wird man sehen, dass es für andere Schnittstellen, wie z.B. *XeLIR2ASM* sinnvoll sein kann, strengere Anforderungen an die Dokumentstruktur zu stellen. Dies kann mit einem separaten Schema garantiert werden (vgl. Kap. 6.2).

  Die genauen Möglichkeiten und Grenzen von Validierungen mit Schemata finden sich in Kap. 5.3.

- XeLIR kann als bindendes Glied zu anderen Zwischendarstellungen fungieren. Dies ist auf zweierlei Arten denkbar:

  1. Eine andere Zwischendarstellung schafft eine direkte Schnittstelle zu XeLIR
  2. Eine andere Zwischendarstellung besitzt ebenfalls ein XML-basiertes Austauschformat. Die XML-Technologie bietet dann mit XSL ein mächtiges Instrument, diese beiden Formate in das jeweils andere zu überführen.

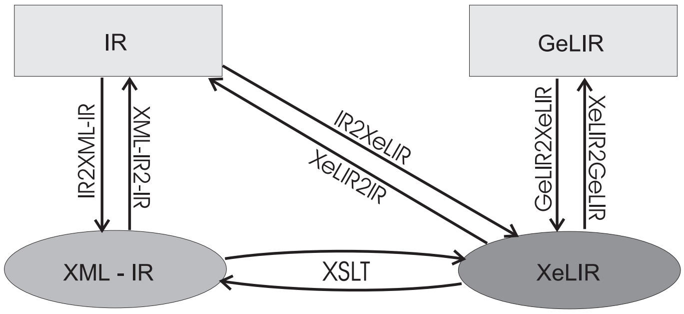

Abb. 5.2: Mögliche Schnittstellen zu GeLIR und anderen Zwischendarstellungen

- Da XML ein standardisiertes Format für den Dokumentenaustausch entwickelt hat und dementsprechende Unterstützung findet, bieten sich viele erweiterte Möglichkeiten. So ist es z.B. möglich,
  - mit Hilfe eines XML-Viewers ein XeLIR-Dokument anzuzeigen.
  - mit XSLT ein XeLIR-Dokument in ein HTML-Dokument zu überführen und mit einem HTML-Browser darzustellen.
  - mit einem XML-Editor ein XeLIR-Dokument zu bearbeiten (bei Editoren, die XML-Schema unterstützen, kann das sogar validierend geschehen). Dies erleichtert Handoptimierungen und Änderungen an der Architekturbeschreibung.

5.2 Der Aufbau von XeLIR

Im Folgenden wird nun der Grundaufbau von XeLIR beschrieben, der sich stark an GeLIR orientiert, weil XeLIR direkt zur Speicherung von Zwischenzuständen von GeLIR dienen soll.

Das Rootelement eines XeLIR-Dokuments ist <XeLIR>. Genau wie GeLIR teilt sich XeLIR in die Bereiche Zielarchitektur (als <Target>-Kindelement) und Programmdarstellung (als <Program>-Kindelement). Die Zielarchitekturbeschreibung gliedert sich wiederum in Typen, Registerfiles, Funktionseinheiten, Instruktionstypen und Operationen, die jeweils als Kindelemente unter dem <*Target*>-Element angeordnet werden, wobei für jede einzelne Instanz ein eigenes Kindelement erzeugt wird. Das Programm besteht aus genau einer globalen Symboltabelle und den Funktionen des Programms. Jedes XeLIR-Dokument muss ein <Target>-Element enthalten, da jede Programmdarstellung auf darin enthaltene Komponenten verweist (z.B. Typen). Die Programmdarstellung ist dagegeben optional, wenn z.B. nur eine Architekturbeschreibung in einem XeLIR-Dokument ablegt werden soll.

Diese Eigenschaften ergeben das folgende Grundgerüst von XeLIR, wobei im XeLIR-Wurzelknoten schon der Namespace und alle Informationen zur Benutzung des XeLIR-Schemas eingetragen sind:

```
<XeLIR xmlns="http://ls12.cs.uni-dortmund.de/xelir"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://ls12.cs.uni-dortmund.de/xelir xelir.xsd">

  <Target>                                     <!-- genau 1 Instanz -->
    <!-- Beschreibung von Typen, Registerfiles, Funktionseinheiten,
         Instruktionstypen und Operationen -->
    <Type>            ...             </Type>  <!-- n Instanzen -->
    <RegisterFile>    ...     </RegisterFile>  <!-- n Instanzen -->
    <FunctionUnit>    ...     </FunctionUnit>  <!-- n Instanzen -->
    <InstructionType> ... </InstructionType>   <!-- n Instanzen -->
    <Operation>       ...        </Operation>  <!-- n Instanzen -->
  </Target>

  <Program>                                    <!-- max. 1 Instanz -->
    <!-- Beschreibung des Programms -->
    <Symboltable>     ...      </Symboltable>  <!-- genau 1 Instanz -->
    <Function>        ...         </Function>  <!-- n Instanzen -->
  </Program>

</XeLIR>
```

5.2.1 Zielarchitektur

Die einzelnen Komponenten der Zielarchitekturbeschreibung sind nachfolgend dargestellt:

- Jeder **Typ** hat eine im ganzen Dokument eindeutige ID der Form *Type-xxx*. Die IDs aller anderen Komponenten der Zielarchitektur haben eine entsprechende Form mit jeweils einem anderen Prefix. Darüber hinaus besteht jeder Typ aus einer Klassifizierung (z.B. fun, integer, ptr) und einem Namen. Diese Informationen werden jeweils in einem entsprechenden Attribut des *Type*-Elements angegeben. Die Größe wird je nachdem, ob sie in Bits oder Bytes definiert wird, für die erforderlichen Datentypen in einem entsprechendem Attribut (*bits, bytes*) angegeben. Für verschiedene Typklassen können weitere Attribute definiert werden, z.B. das leere Attribut *unsigned* sowie *mantissa* und *exponent* für die exakte Beschreibung von float-Datentypen.

  ```
  <Type id="Type-1" class="int" name="type_int" bytes="2"/>
  ```

Um einen definierten Typen zu referenzieren gibt es das eigenständige Element *Typeref*. Dieses enthält das Attribut *ref*, das als Wert die ID des referenzierten Typen enthält.

```
<Typeref ref="Type-1"/>
```

Die Beschreibung komplexer Typen wie z.B. *ptr* auf *char* kann auf zwei verschiedene Arten geschehen: Entweder werden zunächst beide Typen einzeln definiert und dem *ptr*-Element wird zusätzlich eine Referenzierung auf den definierten *char*-Typen hinzugefügt

```
<Type id="Type-3" class="char" name="Char" bytes="1"/>
<Type id="Type-2" class="ptr" name="CharPtr" bytes="4">
  <Typeref ref="Type-3"/>
</Type>
```

oder der Subtyp wird direkt als Kindelement angehängt:

```
<Type id="Type-2" class="ptr" name="type_CharPtr" bytes="4">
  <Type id="Type-3" class="char" name="type_Char" bytes="1"/>
</Type>
```

Die zweite Darstellung ist intuitiver, um Typen manuell hinzuzufügen. Bei der Konvertierung von GeLIR zu XeLIR werden allerdings solche Hierarchien vermieden, weil es dadurch einfacher ist, Mehrfachdefinitionen eines Typen (einmal als Typ und einmal als Subtyp eines anderen Typen) in einem Dokument zu verhindern. Dies kann deshalb passieren, weil in GeLIR jeder Subtyp auch als selbstständiger Typ definiert ist. Arrays sind entsprechend wie *Ptr*-Typen aufgebaut. *Typeref*-Elemente können keine Kindelemente haben.

Typen der Klasse *fun* definieren den Ergebnistyp und die Typen der Argumente einer Funktion. Diese sind als Kindelemente dieses Elementes abgelegt, wobei das erste Element den Ergebnistypen spezifiziert und alle weiteren die Argumente. Für eine binäre Funktion kann dies z.B. folgendes Aussehen haben:

```
<Type id="Type-3" class="int" name="type_int" bytes="2"/>
<Type id="Type-101" class="fun" name="type-binary-function">
  <Typeref ref="Type-3" info="int"/>    <!-- Ergebnistyp (int) -->
  <Typeref ref="Type-2" info="char*"/>  <!-- 1. Argument (char*) -->
  <Typeref ref="Type-3" info="int"/>    <!-- 2. Argument (int) -->
</Type>
```

- In **Registerfiles (RF)** können verschiedene Speichertypen wie Hauptspeicher (*mem*), Register (*reg*), oder flüchtiger Speicher (*transitory*) definiert werden. Dies wird durch das Attribut *class* festgelegt. Die weiteren Informationen, die eine Ressource beschreiben, sind in folgenden Kindelementen abgelegt:

– Das Element *Range* gibt den Indexbereich der Ressource an (vgl. Kap. 3.1.2), der durch die Attribute *idxLow* und *idxHigh* definiert wird. Aus diesem Bereich wird auch die Größe der Ressource ermittelt.
– Der Typ wird durch eine Referenzierung im Element *Typeref* festgelegt.
– Das Element *Name* ist obligatorisch.
– Der *AsmName* ist optional wird aber z.B. für das Herausschreiben von Assemblercode genutzt.
– Der *SimName* ist optional zur Simulationsunterstützung definierbar.

```
<RegisterFile id="RF-1024" class="reg">
  <Range idxLow="769" idxHigh="769" size="1"/>
  <Typeref  ref="Type-5" info="int"/>
  <Name>Reg_A</Name>
  <AsmName>ax</AsmName>
  <SimName>Reg_A</SimName>
</RegisterFile>
```

Bei Registerhierarchien hat das <RegisterFile>-Element weitere Kindelemente. Um anzuzeigen, dass dieses Register ein Teil eines anderen Registerfiles ist, wird in einem <RegisterFileSet>-Element auf das andere Registerfile verwiesen. Umgekehrt verweist ein <RegisterFileElement> auf eventuelle Teilregister. Die Registerfiles, auf die verwiesen wird, sind eigenständig definiert.

• **Funktionseinheiten (FU)** haben einen ähnlichen Aufbau wie Registerfile-Ressourcen, das Grundelement hat den Namen *FunctionUnit*. FUs werden nicht weiter klassifiziert, so dass das Attribut *class* entfällt. Der Indexbereich ist wie bei Registerfiles mit den Attributen *idxLow* und *idxHigh* im Kindelement *Range* definiert. Jeder Index beschreibt dabei eine Instanz der Funktionseinheit.

```
<FunctionUnit id="FU-8193">
  <Range idxLow="2857" idxHigh="2857"/>
  <Typeref ref="Type-1"/>
  <Name>AGU</Name>
  <SimName>AGU</SimName>
</FunctionUnit>
```

• **Instruktionstypen (IT)** haben das Grundelement *InstructionType*. Der Aufbau entspricht ansonsten dem von Funktionseinheiten, der Indexbereich hat aber i.d.R. die Größe eins.

```
<InstructionType id="IT-8199">
  <Range idxLow="2863" idxHigh="2863"/>
```

```
    <Typeref ref="Type-1"/>
    <Name>it_1</Name>
    <SimName>it_1</SimName>
  </InstructionType>
```

- **Operationen** Der Grundaufbau einer Operation ist nachfolgend dargestellt:

```
  <Operation id="OP-6" covered_arg="0">
    <Name>+</Name>
    <Typeref ref="Type-2" info="fun(none, none)"/>
    <DefaultInit>
      <AltEntry> ... </AltEntry>
    </DefaultInit>
    <AltEntry> ... </AltEntry>
       ...
    <AltEntry> ... </AltEntry>
    <SimExecRule>...</SimExecRule>
       ...
    <SimExecRule>...</SimExecRule>
    <SimCondString> ... </SimCondString>
  </Operation>
```

In Kindelementen werden der Name der Operation sowie der Typ (Referenz auf einen Funktionstypen) deklariert. In jeweils einem Kindelement <AltEntry> werden die kombinierbaren Ressourcen beschrieben, der genaue Aufbau ist unten aufgeführt. <DefaultInit> definiert optional eine Standardüberdeckung einer Operation. Für nicht abstrakte Operationen können Strings zur Simulationsunterstützung in den Kindelementen <SimExecRule> und <SimCondString> angegeben werden (vgl. Kap. 3).

Jede Alternative hat folgenden Aufbau:

```
<AltEntry exec_time="1" latency="1">
  <OperationAlt ref="OP-129"/>  <!-- alt. Operation           -->
  <FUAlt ref="FU-8197"/>        <!-- alt. Funktionseinheit    -->
  <ITAlt ref="IT-8199"/>        <!-- alt. Instruktionstyp     -->
  <DefAlts>                     <!-- Menge der Alt. für Def.  -->
    <DefAlt ref="RF-8192"/>         <!-- eine Definitionsalt. -->
        ...                         <!-- weitere Alternativen -->
  </DefAlts>
  <ArgAlts>                     <!-- Alt. für 1. Argument     -->
    <ArgAlt ref="RF-8193"/>         <!-- eine Alternative     -->
        ...                         <!-- weitere Alternativen -->
  </ArgAlts>
  <ArgAlts>                     <!-- Alt. für 2. Argument     -->
```

```
      <ArgAlt ref="RF-8194"/>          <!-- eine Alternative     -->
        ...                            <!-- weitere Alternativen -->
    </ArgAlts>
  </AltEntry>
```

Für jede Alternative können die Ausführungszeit (exec_time) und die Latenzzeit (latency) in den entsprechenden Attributen angegeben werden. Diese Werte können damit für verschiedene Ressourcekombinationen individuell gesetzt werden. In den Kindelementen sind alle Komponenten der Alternativen referenziert:

– Jedes <**OperationAlt**>-Element referenziert eine Zielmaschinenoperation.
– Jede alternative Funktionseinheit ist in einem <**FUAlt**>-Element abgelegt.
– Jedes <**ITAlt**>-Element repräsentiert einen Instruktionstypen.

Die Elemente <**DefAlts**> und <**ArgAlts**> kapseln die Registerfile-Ressourcen für die Definition bzw. für die Argumente. Die einzelnen <ArgAlts>-Elemente beschreiben nacheinander die Alternativen für jedes Argument, beginnend mit dem ersten. Jeder dieser Elementtypen kann beliebig viele Kindelemente genau eines Typs haben, nämlich <**DefAlt**> bzw. <**ArgAlt**>. In jedem dieser Elemente wird jeweils eine Registerfile-Ressource referenziert, die für die Definition bzw. das Argument benutzt werden kann.

5.2.2 Programmdarstellung

Die Programmdarstellung besteht aus dem hierarchisch aufgebauten Programm und aus den globalen und lokalen Symboltabellen, die jeweils eine beliebige Zahl von Einträgen haben können.

Symboltabelle

Eine **Symboltabelle** wird durch das Element *Symboltable* beschrieben, das keinerlei Attribute, aber für jeden Symboltabelleneintrag ein Kindelement mit dem Namen *Entry* besitzt. Ein *Entry*-Element selbst besteht aus Attributen, die allgemeine Informationen enthalten wie *key* des Eintrags, die Klassifizierung *class* und den Gültigkeitsbereich *scope* (z.B. GLOBAL/LOCAL). In zusätzlichen Kindelementen mit folgenden Namen sind die weiteren Informationen eines Symboltabelleneintrags abgelegt:

- *Name* beinhaltet den symbolischen Namen.
- *Typeref* referenziert den Typen der Variablen.
- In *InitValue* sind Informationen für die Initialisierung der Variablen abgelegt. Dieses Element enthält das Attribut *class* in dem der Typ, den dieses Element repräsentiert, abgelegt wird. Der eigentliche Wert ist in dem Attribut *value* abgelegt. InitValues können Hierarchien bilden um Variablen eines komplexen Typs beschreiben zu können.

- *Mem* beinhaltet Informationen über den Speicherbereich, dem dieser Eintrag zugewiesen ist. Einem Eintrag können mehrere Speicherbereiche zugewiesen werden, was durch mehrere Mem-Kindelemente ausgedrückt wird.

Das folgende Beispiel zeigt die Definition eines Symboltabelleneintrags für eine globale Variable i vom Typ int, die mit dem Wert 3 initialisiert wird und bereits einen Speicherbereich zugewiesen bekommen hat:

```
<Entry key="5" class="VAR" scope="GLOBAL">
   <Name>i</Name>
   <Typeref ref="Type-4105" info="int"/>
   <InitValue class="int" value="3"/>
   <Mem class="REL" memkey="RF-3072">
      <Name>i_7</Name>
      <Range from="2" to="2"/>
      <Typeref ref="Type-5"/>
   </Mem>
</Entry>
```

Programmhierarchie

Da die **Programmstruktur** hierarchisch angelegt ist (Program→Fun→BB→MI→MO), leitet sich die Repräsentation in XeLIR fast automatisch ab:

```
<Program>
  <Function>
    <BasicBlock>
      <MI>
        <!--MO-->
      </MI>
      <MI>
        <!--MO-->
      </MI>
    </BasicBlock>
  </Function>
</Program>
```

Jede Ebene beinhaltet aber noch zusätzliche Informationen:

- Die oberste Ebene, die das komplette **Programm** repräsentiert, enthält die globale Symboltabelle und alle Funktionen:

```
<Program>
  <Symboltable>  ... </Symboltable>
  <Function> ...        </Function>
    ...
  <Function> ...        </Function>
</Program>
```

- Eine einzelne **Funktion** hat folgenden Aufbau:

```
<Function id="Fun-0" key="2" name="main" filename="">
  <FunArg key="50">argc_4</FunArg>
  <FunArg key="51">argv_5</FunArg>
  <Symboltable> ...  </Symboltable>
  <BasicBlock>  ...   </BasicBlock>
    ...
  <BasicBlock>  ...   </BasicBlock>
</Function>
```

  Jede Funktion hat also eine global eindeutige ID, einen Key, der auf einen Eintrag in der globalen Symboltabelle verweist, und einen Namen. IDs können bei verschiedenen Komponenten (Funktionen, Basisblöcke, MIs, MOs) den gleichen Wert haben. Um eine dokumentweite Eindeutigkeit zu erreichen, bekommt jede ID wie in der Beschreibung der Zielarchitektur ein von dem Typ der Komponente abhängigen Präfix (hier: *Fun-*). In jeder Funktion kann zusätzlich der Dateiname hinterlegt sein, in der diese Funktion definiert wurde. Die Parameter der Funktion sind in Kindelementen angeordnet, die per Attribut *key* auf den zugehörigen Symboltabelleneintrag verweisen sowie den Namen der Variable beinhalten. Darüber hinaus sind genau eine lokale Symboltabelle (Aufbau siehe oben) und mindestens ein Basisblock in Kindelementen abgelegt.

- **Basisblöcke** haben eine Folge von <VARIN>- und <DEFOUT>-Elementen, die die VAR-MO repräsentieren, die u.a. für die Datenflussanalyse verwendet werden. Zwischen diesen Elementen sind die Elemente für die Darstellung der Maschineninstruktionen:

```
<BasicBlock id="BB-0">
    <VARIN>  ...  </VARIN>
      ...
    <VARIN>  ...  </VARIN>
    <MI>     ...     </MI>
      ...
    <MI>     ...     </MI>
    <DEFOUT> ... </DEFOUT>
      ...
    <DEFOUT> ... </DEFOUT>
</BasicBlock>
```

- **Maschineninstruktionen (MIs)** beinhalten als Kindelemente alle Maschinenoperationen, die parallel ausführbar sind. Im folgenden Beispiel sind dies eine arithmetische/logische Operation und eine Loadoperation.

```
<MI id="MI-57">
   <OPR> ... </OPR>
   <LD>  ...  </LD>
</MI>
```

- **Maschinenoperationen (MOs)** haben je nach Typ einen anderen Elementnamen. Dies hat den Vorteil, dass für jeden Typ in einem Schema unterschiedliche Regeln für den Aufbau definiert werden können. Folgende Tabelle gibt einen Überblick über die einzelnen Typen und die zugehörigen Elementnamen:

MO-Typ	Elementname
Arithmetische oder logische Operation	<OPR>
Virtuelle Operationen	<VOPR>
Load-Operationen	<LD>
Store Operationen	<ST>
Datentransfer auf Maschinenebene	<MV>
Datentransfer	<COPY>
Casting	<CAST>
Unbedingter Sprung	<JMP>
Bedingter Sprung	<CJMP>
Funktionsaufruf	<CALL>
Funktionsrücksprung	<RET>
Zerooverhead-Hardware-Loop	<ZLOOP>/<ZJMP>
Konstante	<CONST>
Adresse	<ADDR>
Varsin/Defsout	<VARIN>/<DEFOUT>

Der Grundaufbau einer Maschinenoperationen sei anhand einer arithmetischen Operation verdeutlicht, zunächst in einer abstrakten Darstellung:

```
<OPR  id="MO-290" class2="ADDR_OPR"
      op_name="+" op_sym="2" value_no="290">
   <Def name="lir_140" stab_id="140"/>
   <Arg name="lir_130" stab_id="130"/>
   <Arg name="lir_139" stab_id="139"/>
</OPR>
```

Das Hauptelement enthält in den Attributen alle elementaren Informationen wie ID, Klassifizierung, Name der Operation und eine Referenz zu der entsprechenden Operation in der Zielarchitekturbeschreibung (*op_sym*). Die Klassifizierung im *class2*-Attribut gibt einen Hinweis, in welchem Zusammenhang diese MO genutzt wird, in diesem Fall ist es eine Adressoperation (*ADDR_OPR*). Das Attribut *value_no* erlaubt es, MOs zu erkennen, die den gleichen Wert definieren. Z.B. wird bei einem Datentransfer über mehrere MOs derselbe Wert in allen beteiligten MOs gesetzt. Die Lokationen sind in Kindelementen untergebracht. Abhängig von dem Typen der Operation gibt es eine Definition und eine bestimmte Anzahl von Argumenten, in denen jeweils der Name und der zugehörige Symboltabelleneintrag angegeben sind. Dieser Aufbau schafft eine klare Trennung zwischen der Operation im Kern und den zugehörigen Lokationen. Als einziger Maschinenoperationstyp hat die Darstellung einer Konstanten ein zusätzliches Kindelement. Neben einer Definition werden dort weitere Informationen in einem <ConstData>-Kindelement abgelegt:

```
<CONST  class2="ADDR_OPR" id="MO-340" value_no="338">
   <Def name="lir_185" stab_id="185"/>
   <ConstData type="INT" value="-1" stab_id="184"/>
</CONST>
```

Im Laufe der Codegenerierung in GeLIR wird jede einzelne MO von alternativen Ressourcen überdeckt (vgl. Kap. 3). Diese **Überdeckungen** können den einzelnen Komponenten der MO zugeteilt werden: alternative Operationen, Funktionseinheiten und Instruktionstypen werden als Kindelemente des Hauptelements, alternative Registerfile-Ressourcen als Kindelemente der einzelnen Lokationen (Definition bzw. Argument) hinzugefügt:

```
<OPR  id="MO-290" class2="ADDR_OPR"
      op_name="+" op_sym="2" value_no="290">
    <OperationAlt ref="OP-105" op_alt_name="RW_P_M"/>
    <FUAlt ref="FU-8193"/>
    <ITAlt ref="IT-8199"/>

    <Def name="lir_140" stab_id="140">
       <DefAlt ref="RF-5376"/>
       <DefIdx index="2845" asmname=""/>
    </Def>
    <Arg name="lir_130" stab_id="130">
       <ArgAlt ref="RF-3584"/>
       <ArgIdx index="784" asmname=""/>
    </Arg>
    <Arg name="lir_139" stab_id="139">
       <ArgAlt ref="RF-7936"/>
```

```
        <ArgIdx index="2855" asmname=""/>
      </Arg>
   </OPR>
```

Die Maschinenoperationen im Beispiel ist eindeutig überdeckt, da genau eine alternative Operation, Funktion und Instruktion für die abstrakte Operation und je eine Registerfile-Ressource für die Definition und jedes Argument angegeben ist. Um für eine der genannten Komponenten eine zusätzliche alternative Überdeckung anzugeben, wird ein weiteres entsprechendes Element (OperationAlt, FUAlt, usw.) hinzugefügt. Die Alternativen von Lokationen werden durch die Angabe der Registerfile-Ressource in einem *<DefAlt>*/*<ArgAlt>*-Element und zusätzlich durch die Angabe eines eindeutigen Indizes in einem *<DefIdx>*/*<ArgIdx>*-Element dargestellt.

Die Darstellung von **komplexen Maschinenoperationen** erfolgt, indem eine komplette Maschinenoperation das Kindelement eines Arguments bildet. Folgendes Beispiel zeigt eine algebraische Operation, deren zweites Argument eine CONST-MO als SubMO hat. Der Symboltabelleneintrag der Definition dieser MO muss mit dem des zweiten Arguments der umgebenen MO übereinstimmen.

```
<OPR  id="MO-290" class2="ADDR_OPR"
      op_name="+" op_sym="2" value_no="290">
   <Def name="lir_140" stab_id="140"/>
   <Arg name="lir_130" stab_id="130"/>
   <Arg name="lir_139" stab_id="139"/>
      <CONST  class2="ADDR_OPR" id="MO-289" value_no="7">
         <Def name="lir_139" stab_id="139"/>
         <ConstData type="INT" value="1" stab_id="138"/>
      </CONST>
   </Arg>
</OPR>
```

5.3 Validierung eines XeLIR-Dokuments

Schemata bieten eine einfache Möglichkeit, die Struktur eines XML-Dokuments (und damit eines XeLIR-Dokuments) zu definieren, die dann mit Hilfe eines validierenden Parsers überprüft werden kann. In diesem Kapitel werden die Möglichkeiten und Grenzen der Validierung einer XeLIR-Dokumentinstanz mittels Schemata besprochen, die sowohl durch syntaktische als auch in begrenztem Umfang semantische Festlegungen charakterisiert sein können.

- Hinsichtlich der Syntax lässt sich die Struktur des Dokuments festlegen. Im Einzelnen betrifft dies:

  - die Elemente-Hierarchie, bei der z.B. der Aufbau der Programmhierarchie und der Aufbau der Maschinenoperationen exakt beschrieben werden.
  - die benutzten Attribute in einzelnen Elementen, für die alle erlaubten Attribute bestimmt werden und zusätzlich das Vorhandensein einzelner Attribute vorgeschrieben wird.

- Semantisch werden u.a. folgende Eigenschaften des Dokuments vorgeschrieben:
  - die Eindeutigkeit der Identifier einzelner Elemente wie z.B. die Komponenten der Programmdarstellung: Die IDs von MOs, MIs, usw. bekommen den von XML-Schema unterstützten Datentypen ID zugewiesen, der die dokumentweite Eindeutigkeit garantiert.
  - das Vorhandensein referenzierter Objekte: Alle Referenzen bekommen den Datentyp IDREF zugewiesen, der garantiert, dass im Dokument ein Element mit entsprechender ID existiert.
  - Typüberprüfungen einzelner Element- und Attributwerte. Für PCDATA und Attributwerte wird der Wertebereich exakt definiert.

Ein Schema legt die Eigenschaften für alle Dokumentinstanzen fest, was bedeutet, dass jede Definition eine grundsätzliche Einschränkung der Beschreibungsmöglichkeiten für alle XeLIR-Dokumente beinhaltet. Das Schema sollte somit eine Mindestanforderung an den Aufbau eines XeLIR-Dokuments definieren.

Mit einem Schema für XeLIR lässt sich die Element-Hierarchie exakt bestimmen. Z.B. wird durch den folgenden Ausdruck garantiert, dass das <*XeLIR*>-Rootelement genau ein <*Target*>-Kindelement haben muss, optional gefolgt von einem <*Program*>-Kindelement.

```
<xs:element name="XeLIR">
  <xs:complexType>
    <xs:sequence>
     <xs:element ref="Target" minOccurs="1" maxOccurs="1"/>
     <xs:element ref="Program" minOccurs="0" maxOccurs="1"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Nach diesem Muster ist der Aufbau der kompletten Elemente-Hierarchie für ein XeLIR-Dokument festgelegt, dazu gehören z.B.

- der Aufbau der Architekturbeschreibung deren Komponenten, wie Typen, Registerfiles, Funktionseinheiten und Instruktionstypen.
- die komplette Programmhierarchie (Program $\rightarrow$ Function $\rightarrow$ BasicBlock $\rightarrow$ MI $\rightarrow$ MO).

- die erlaubten Überdeckungen in einer Maschinenoperation.
- der Aufbau komplexer Maschinenoperationen durch Anhängen einer MO an ein Argument.

Für Attribute kann der Wertebereich festgelegt und das Vorhandensein garantiert werden. Dies wird z.B. für das Element *Typeref* getan, indem vorgeschrieben wird, dass dies genau ein Attribut mit dem Namen *ref* enthalten muss und der Attributwert dem Format einer Typ-ID entspricht:

```
<xs:element name="Typeref">
  <xs:complexType>
    <xs:attribute name="ref" type="Type-IDREF" use="required"/>
  </xs:complexType>
</xs:element>
```

Nach diesem Muster sind z.B. auch folgende Eigenschaften eines XeLIR-Dokuments festgelegt:

- die erlaubten Attribute für jedes einzelne Element
- obligatorische ID für Ressourcen und Programmkomponenten
- obligatorische Referenzattribute in Alternativen

In Schemas sind die Typen *ID* und *IDREF* vordefiniert. Diese garantieren, wie in Kap. 4 beschrieben die Eindeutigkeit von Werten bzw. das Vorhandensein von IDs, wenn sie referenziert werden. Von diesen Basistypen können eigene Typen abgeleitet werden, die noch strengeren Anforderungen genügen. In XeLIR muss die Eindeutigkeit von vielen Komponenten garantiert werden, von Registerfile-Ressourcen über Operationen bis zu Funktionen und MOs. Für diese werden mittels neuer einfacher Typdefinitionen die verschiedenen Formate festgelegt. Folgendes Beispiel zeigt die Definition der Typen *Type-ID* und *Type-IDREF*, die sicherstellen, dass alle Typ-IDs die Form *Type-xxxx* (mit beliebig langer positiver Zahl und ohne führende 0) haben. Die Typen erben (derivation by restriction) von den vordefinierten ID-/IDREF-Typen, indem diese auf bestimmte lexikalische Muster eingeschränkt werden.

```
<xs:simpleType name="Type-ID">
  <xs:restriction base="xs:ID">
    <xs:pattern value="Type-[1-9][0-9]*"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="Type-IDREF">
  <xs:restriction base="xs:IDREF">
```

```
    <xs:pattern value="Type-[1-9][0-9]*"/>
  </xs:restriction>
</xs:simpleType>
```

Die IDs/IDREFs folgender Komponenten werden nach diesem Muster genau festgelegt:

- In der Zielarchitektur: Typen, Registerfiles, Funktionseinheiten, Instruktionstypen, Operationen.
- In der Programmdarstellung: Funktionen, Basisblöcke, Maschineninstruktionen, Maschinenoperationen.
- Im gesamten Dokument: Referenzierungen auf eines der oben genannten Objekte.

Diese Methode ist für Symboltabellen-Einträge nicht anwendbar, da lokale Einträge nur innerhalb einer Funktion eindeutig sind, nicht aber global. *ID*s und *IDREF*s beziehen sich aber immer auf das gesamte Dokument. Eine Überprüfung der Eindeutigkeit muss an dieser Stelle von dem Konverter (*XeLIR2GeLIR*) übernommen werden.

Bei Klassifizierungen bestimmter Komponenten können alle Möglichkeiten exakt festgelegt werden. Die möglichen Klassen eines Symboltabelleneintrags werden folgendermaßen beschrieben:

```
<xs:simpleType name="STabEntryClasses">
  <xs:restriction base="xs:string">
    <xs:enumeration value="NONE"/>
    <xs:enumeration value="VAR"/>
    <xs:enumeration value="FUN"/>
    <xs:enumeration value="LABEL"/>
    <xs:enumeration value="CONST"/>
  </xs:restriction>
</xs:simpleType>
```

Diese Festlegungen werden auch für folgende Komponenten gemacht:

- Typklassen (void, bool, int, ...)
- Klassen von Registerfile-Ressourcen (REG, MEM, ...)
- Gültigkeitsbereich (Scope) von Einträgen (GLOBAL, LOCAL, ...)
- Speicherform (ABS, REL, ...)
- MO-Klassifizierung (ADDR_OPR, PHI_OPR, ...)

Für viele weitere Attribute kann der Wertebereich eingeschränkt werden, z.B.:

- Dimension eines Arrays auf positive Interger
- Index einer Ressource ist größer gleich -1

Zum Abschluss dieses Kapitels folgen noch ein paar Anmerkungen zu weiteren vorgenommenen oder nicht durchführbaren Einschränkungen. Beim Aufbau von komplexen MOs wird die Menge der als Sub-MO erlaubten MOs sinnvoll eingeschränkt. Dies betrifft z.B. JMP-MOs. Im XeLIR-Schema kann nicht festgelegt werden, ob die gleichzeitige Benutzung bestimmter Attribute in einem Element sich ausschließt (z.B. die von bits und bytes in einem Typen). Überprüfungen dieser Art sind Aufgabe des *XeLIR2GeLIR*-Konverters.

Das XeLIR-Schema unterstützt den GeLIR-Anwender in der Überprüfung der Vollständigkeit der Beschreibung der GeLIR-Komponenten. So können fehlende Typreferenzierungen oder fehlende Ressourcen schnell erkannt werden. Da ein Schema *alle* Dokumentinstanzen beschreibt ist jede festgelegte Einschränkung allgemeingültig. Es ist zu überlegen über Schema-Diversifikation für bestimmte Phasen der Codegenerierung separate Schemata mit stärkeren Aussagen zu definieren. So können z.B. für die abstrakte Darstellung das Vorkommen von alternativen Überdeckungen von MOs ausgeschlossen oder für einen generierten Code eine eindeutige Überdeckung mit einem Schema garantiert werden. Der letzte Fall ist z.B. für das Herausschreiben von Assemblercode interessant, das in Kap. 6.2 besprochen wird.

Kapitel 6

Anwendungen von XeLIR

In diesem Kapitel werden Anwendungsfelder von XeLIR besprochen. Der Hauptanwendungsbereich ist die Funktion als Schnittstelle von XeLIR zu GeLIR (Kap. 6.1). Dafür ist es notwendig alle Datenstrukturen von GeLIR auf XeLIR abzubilden und einen in einem XeLIR-Dokument abgelegten Zustand deckungsgleich in GeLIR wiederherstellen zu können. In den darauf folgenden Kapiteln werden weitere Anwendungsmöglichkeit erörtert, die die Baumstruktur eines XML-Dokuments dazu nutzen nach bestimmten Mustern zu suchen. Dies wird für das Herausschreiben von Assemblercode (→ Kap. 6.2) und für Peepholeoptimierungen verwendet (→ Kap. 6.3).

6.1 GeLIR-Austauschformat

Im Mittelpunkt von XeLIR steht die Rolle als GeLIR-Austauschformat. In diesem Abschnitt wird die Konvertierung von GeLIR nach XeLIR und in die umgekehrte Richtung besprochen (→ Abb. 6.1).

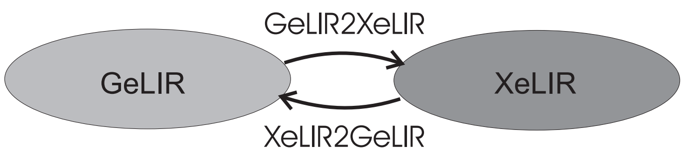

Abb. 6.1: XeLIR als GeLIR-Austauschformat

GeLIR2XeLIR

Die Erstellung eines XeLIR-Dokuments aus GeLIR kann direkt über einen Ausgabestrom erfolgen. Bei dem Durchlaufen der GeLIR-Datenstrukturen muss beachtet werden, dass alle

Daten im XeLIR-Dokument abgelegt werden, die nötig sind, um den aktuellen Zustand zu rekonstruieren. Um die Größe des Dokuments nicht unnötig zu erhöhen, muss auf der anderen Seite bei der Dokumenterstellung darauf geachtet werden, dass redundante Informationen vermieden werden. Dies heißt im Besonderen, dass jedes Objekt von GeLIR nur einmal geschrieben wird. Dies ist auch deshalb von Bedeutung, weil nach einer Wiederherstellung gleich viele Objekte rekonstruiert sein müssen. Mehrfachreferenzierungen auf dasselbe Objekt müssen später exakt rekonstruiert werden, um Inkonsistenzen zu verhindern.

In der Architekturbeschreibung sind im *LirTarget*-Objekt alle Komponenten wie Typen, Ressourcen und Operationen registriert. Beim Schreiben wird über all diese Objekte iteriert und jeweils in das XeLIR-Dokument geschrieben. Alle weiteren Referenzen auf diese Objekte werden in XeLIR auch als Referenzierung umgesetzt. Dies garantiert die Eindeutigkeit im XeLIR-Dokument. Die Referenzierung geschieht über die Angabe der ID der einzelnen Komponenten. Die verschiedenen Objekte der Zielarchitekturbeschreibung werden in einer festen Reihenfolge geschrieben. Da Typen von allen Ressourcen und Operationen referenziert werden und das Wiedereinlesen sequentiell geschieht, ist es vorteilhaft, Typen schon zu kennen, wenn die anderen Ressourcen aufgebaut werden. Deshalb wird beim Erstellen eines XeLIR-Dokuments mit Typen begonnen. Eine ähnliche Abhängigkeit besteht zwischen Ressourcen und Operationen. Da die Alternativen von Operationen auf Ressourcen zurückgreifen, werden erst die Ressourcen und dann die Operationen geschrieben.

Im Programmteil wird über alle Ebenen der Programmdarstellung iteriert, dabei ist es wichtig, die Besonderheiten einzelner MO-Typen zu beachten (z.B. Konstante) und komplexe MOs zu berücksichtigen.

Um die Lesbarkeit des XeLIR-Dokuments zu erhöhen werden zusätzliche Informationen abgelegt. Ein Beispiel sind Alternativen, bei denen neben der Referenzierung auf die Ressource zusätzlich der Name in ein *info*-Attribut geschrieben wird. Da der Name allein keine Eindeutigkeit garantiert, bleibt die Angabe der ID notwendig.

```
<ArgAlt ref="RF-1024" info="Reg_A"/>
<ArgAlt ref="RF-1280" info="Reg_B"/>
```

XeLIR2GeLIR

Die grundsätzliche Frage ist, wie das XeLIR-Dokument wieder in GeLIR eingelesen werden soll. Da XeLIR XML-basiert ist, kann eine bestehende Parserimplementierung verwendet werden. In der benutzten Xerces-Bibliothek des Apache XML-Projektes sind beide XML-Standard-APIs implementiert, also sowohl SAX als auch DOM. DOM bietet zu jeder Zeit den Zugriff auf alle Teile des XeLIR-Dokuments, während bei einem event-gesteuertem Parsen mit SAX einige Datenstrukturen selbst zwischengespeichert werden müssen, falls weitere Informationen benötigt werden, die zu einem späteren Zeitpunkt geparst werden. Trotzdem ist der *XeLIR2GeLIR*-Konverter mit SAX implementiert worden, da dieses ein einfacheres Interface zur Verfügung stellt, das es erleichtert, Änderungen umzusetzen. Zudem ist das

Parsen mit SAX schneller als mit DOM. Da das Dokument nur eingelesen und nicht manipuliert werden muss, ist SAX ausreichend. Eine Überprüfung der XeLIR-Struktur durch ein Schema wird bei Xerces für SAX genauso unterstützt wie für DOM.

Um das Parsen zu vereinfachen und die Zwischenspeicherung von Daten zu minimieren, wurde das Interface von GeLIR um Methoden zum Setzen von Attributen erweitert. Dies betrifft z.B. Maschinenoperationen, die nicht mehr komplett über einen Konstruktor initialisiert, sondern schrittweise erweitert und aufgebaut werden; dies verhindert u.a. das Zwischenspeichern von mehreren MOs einer komplexen MO. Außerdem enthält GeLIR Automatismen, die überbrückt werden müssen. Dazu gehört z.B. die Vergabe von IDs verschiedener Komponenten. Um den Ursprungszustand kongruent wiederherstellen zu können war es erforderlich, durch zusätzliche Methoden Einfluss auf die Vergabe von IDs nehmen zu können. Dies betrifft insbesondere auch die automatisch generierten VAR-MOs in den Basisblöcken.

Testmethoden

Um die korrekte Wiederherstellung der Zustände zu überprüfen, wurden in der Entwicklungsphase verschiedene Methoden angewandt. Ein Ansatz bestand darin, einen Zwischenzustand zu speichern, ihn anschließend direkt wieder einzulesen und erneut unter einem anderen Dateinamen zu speichern. Die beiden erzeugten XeLIR-Dokumente mussten übereinstimmen. Dieses Verfahren entdeckt allerdings nicht das Fehlen spezieller Informationen einzelner Komponenten, da diese in keinem der beiden Dokumente gespeichert werden. Es garantiert nur, dass alle geschriebenen Informationen korrekt wiederhergestellt werden. Der nächste Schritt bestand darin, andere Textformate einzubeziehen, die von GeLIR z.B. zur graphischen Darstellung oder zum Debuggen genutzt werden. Diese anderen Formate wurden einmal vor der Erstellung eines XeLIR-Dokuments geschrieben. Darauf folgte ein direktes Wiedereinlesen von XeLIR sowie ein nochmaliges Schreiben des anderen Textformates in eine zweite Datei. Auch diese beiden Textdateien mussten wie vorher die XeLIR-Dokumente deckungsgleich sein.

Da das Format der XeLIR-Dokumente von der Phase der Codegenerierung abhängt (z.B. keine Überdeckung in der initialen Darstellung), wurden die oben dargestellten Überprüfungen für verschiedene Phasen angewendet. Als letztes wurde die Codegenerierung mit verschiedenen Testprogrammen komplett durchlaufen. Dabei wurde in verschiedenen Phasen eine Zwischenspeicherung in ein XeLIR-Dokument durchgeführt und direkt wieder eingelesen. Die dabei erreichten Ergebnisse wurden mit den entsprechenden Durchläufen ohne Zwischenspeicherung verglichen.

6.2 Assemblercode-Generierung mit Pattern

In diesem und den folgenden Abschnitten werden verschiedene Anwendungsmöglichkeiten aufgezeigt, die sich durch den Einsatz von Mustererkennung auf einem XeLIR-Dokument ergeben. Ein erstes Beispiel ist das Herausschreiben von Assemblercode, das nachfolgend auch Assemblercode-Generierung[1] genannt wird. Bei diesem Prozess muss über alle Maschinenoperation iteriert und der äquivalente Assemblercode herausgeschrieben werden. Dabei verläuft die Generierung für verschiedene MOs desselben Typen i.d.R. nach dem gleichen Muster. Da die verschieden MO-Typen in XeLIR leicht durch verschiedene Muster erkannt werden können und die Zahl der MO-Typen begrenzt ist, liegt es nahe, für jeden Typ von Maschinenoperation ein Muster zu definieren und Ausgabevorschriften zu definieren. Dies erspart eine C++-Implementierung. Für die Assemblercode-Generierung muss ein XeLIR-Dokument nur sequentiell durchlaufen werden und für jede Maschinenoperation ein passendes Muster gefunden werden. Die zu dem Muster gehörenden Ausgabevorschriften werden anschließend umgesetzt. Da der Aufbau von Maschinenoperationen in XeLIR immer eine ähnliche Elementhierarchie hat, gibt es darüber hinaus die Möglichkeit, die verschiedenen Muster automatisch generieren zu lassen oder neue zu einer vorhandenen Menge hinzuzufügen.

Abb. 6.2 gibt einen Überblick über die Schritte, die für die Assemblercode-Generierung mit XeLIR-Pattern nötig sind. Alle nötigen Muster werden automatisch erkannt (1.). Dabei wird für jeden MO-Typ und auch für verschieden aufgebaute komplexe Maschinenoperationen ein neues Muster erzeugt, da für jede dieser Strukturen andere Ausgabevorschriften notwendig sind. Der Benutzer hat nur die Aufgabe, für jedes Muster die Ausgabevorschriften anzugeben (2.). Das anschließende sequentielle Durchlaufen des XeLIR-Dokuments (3.) und die Ausführung der Vorschriften (4.) können wiederum automatisiert werden.

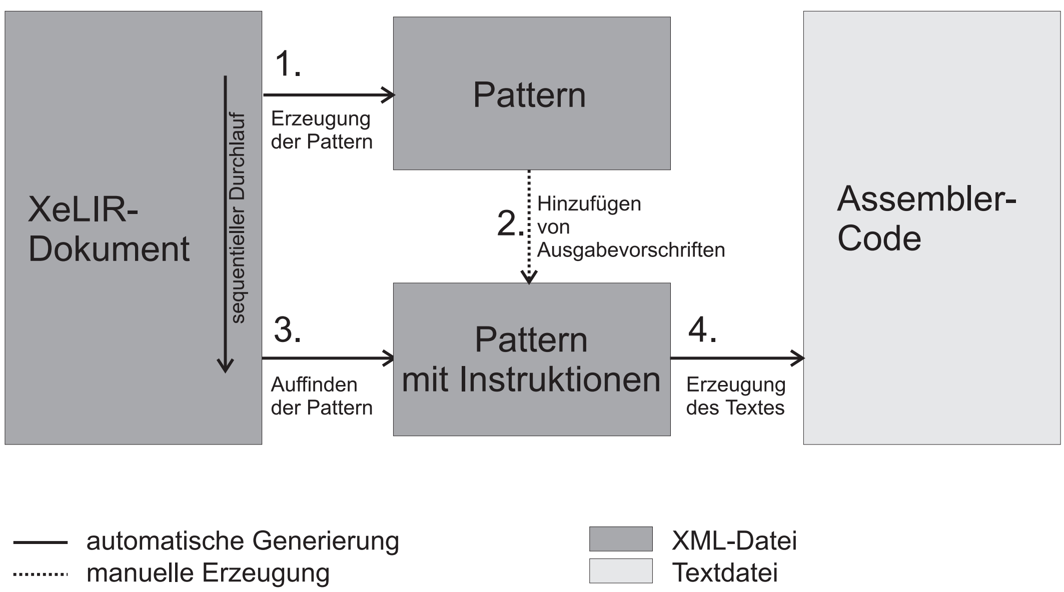

Abb. 6.2: Aufbau des Schreibens von Assemblercode mittels Pattern

[1] Unter dem Begriff Assemblercodegenerierung versteht man i.A. die Erzeugung einer zum Maschinencode äquivalenten Programmdarstellung in der Zwischendarstellung und nicht das Herausschreiben. Da es aber für den Vorgang des Herausschreibens keinen adäquaten Ausdruck gibt, wird auf diesen zurückgegriffen.

Die einzelnen Schritte werden nun noch einmal detailliert betrachtet.

1. Der erste Schritt ist die Erzeugung oder die Erweiterung eines bereits vorhandenen XML-Pattern-Dokuments aus einem bestehenden XeLIR-Dokument. Dazu muss der Grundaufbau von Pattern festgelegt werden, der die Unterscheidungskriterien der Pattern angibt. Zur Veranschaulichung dient folgende Maschinenoperation für einen Datentransfer:

```
<COPY  id="MO-313" class2="ADDR_OPR"
       op_name="cp" op_sym="24" value_no="438">
   <OperationAlt ref="OP-137" op_alt_name="Move"/>
   <FUAlt ref="FU-8198" info="LMU"/>
   <ITAlt ref="IT-8200" info="it_2"/>

   <Def name="lir_151" stab_id="151">
      <DefAlt ref="RF-1280" info="Reg_B"/>
      <DefIdx index="770" asmname="RB"/>
   </Def>
   <Arg name="none" stab_id="0">
      <ArgAlt ref="RF-256" info="rf_none"/>
      <ArgIdx index="-1"/>
   </Arg>
   <Arg name="lir_128" stab_id="128">
      <ArgAlt ref="RF-1024" info="Reg_A"/>
      <ArgIdx index="769" asmname="RA"/>
   </Arg>
</COPY>
```

   Eine andere MO hat nur dann eine von dieser Struktur abweichende Elementehierarchie, wenn

   - der Typ der MO ein anderer ist, d.h. das <COPY>-Element ein anderes Element ist, oder
   - die MO eine SubMO besitzt, die Kindelement eines Arguments ist.

   In beiden Fällen ist ein separates Muster sinnvoll, um andere Ausgabevorschriften zu definieren. Die Differenzierung von Mustern anhand unterschiedlicher Elementehierarchien in Maschinenoperationen ist also ein wichtiges Kriterium. Dies ist allerdings nicht hinreichend, da eine *COPY*-MO von verschiedenen Maschinenoperationen überdeckt werden kann, die verschiedene Ausgabevorschriften zur Generierung von Assemblercode verlangen können. Dies muss berücksichtigt werden. Zwei MOs sind also im Sinne des Patternmatching als gleich anzusehen und benötigen keine unterschiedlichen Muster, falls sie

1. die gleiche Elementehierarchie haben.
2. den gleichen Wert im *ref*-Attribut des <OperationAlt>-Element haben.

Mit Hilfe dieser Kriterien ist es möglich, für jede Maschinenoperation in einem XeLIR-Dokument festzustellen, ob ein entsprechendes Pattern schon definiert wurde. Falls nicht, so wird dieses automatisch erzeugt, indem die Elemente-Hierarchie abgebildet und das *ref*-Attribut inklusive Wert in das Pattern übertragen wird. Das obige Beispiel führt demnach zu folgendem Pattern:

```
<COPY>
   <OperationAlt ref="OP-137" op_alt_name="Move"/>
   <FUAlt/>
   <ITAlt/>

   <Def>
      <DefAlt/>
      <DefIdx asmname=""/>
   </Def>
   <Arg>
      <ArgAlt/>
      <ArgIdx asmname=""/>
   </Arg>
   <Arg>
      <ArgAlt/>
      <ArgIdx asmname=""/>
   </Arg>
</COPY>
```

Das Attribut *asmname* dient als zusätzliche Information, an welcher Stelle in der Hierarchie Daten abgelegt sind, die für die Assemblerausgabe eine Bedeutung haben. Dies ist bei der Definition der Ausgabevorschriften hilfreich (siehe unten). Leere Attribute (dargestellt durch "") im Pattern werden wie nicht vorhandene Attribute bei einem Mustervergleich ignoriert. Das Attribut *op_alt_name* hängt direkt mit dem *ref*-Attribut zusammen, so dass dies keine weitere Patternunterscheidung zur Folge hat.

2. Als nächstes wird für jedes Pattern eine Folge von Ausgabevorschriften hinzugefügt. Nachdem oben bereits das Suchmuster definiert wurde, ist nachfolgend noch einmal der Gesamtaufbau eines einzelnen Pattern angegeben, bestehend aus dem Suchmuster und den Instruktionen:

```
<Pattern>
  <!-- Suchmuster -->
  <Instruction>
```

```
    <Print direct=""/>
    <Print ref=""/>
  </Instruction>
</Pattern>
```

Alle Instruktionen werden in beliebig vielen Kindelementen im <Instruction>-Element abgelegt. Jedes Kindelement ist ein <Print>-Element, das eins von zwei verschiedenen Attributen haben kann, die unterschiedliche Aufgaben haben:

- In einem *direct*-Attribut wird fest verdrahteter Text angegeben, der direkt ausgegeben wird. Dieser Text kann auch Tabulatoren (\t) oder Zeilenumbrüche (\n) enthalten. Die Folge der Ausdrücke

  ```
  <Print direct="CP\n"/>
  <Print direct="MV\n"/>
  ```

  erzeugt z.B. die Ausgabe:

  ```
  CP
  MV
  ```

- In einem *ref*-Attribut wird mit einem eingeschränktem XPath-Ausdruck auf eine Komponente der MO verwiesen und deren Inhalt ausgegeben. Ausgehend von dem Basisknoten der MO wird der Weg zu dem gesuchten Element beschrieben und mit einem vorangestelltem @ wird auf ein bestimmtes Attribut des Zielelementes verwiesen. Möchte man in dem oben angegebenen Beispiel einer MO im XeLIR-Dokument den Inhalt des *asmname*-Attributes im <DefIdx>-Element auswählen, so wird mit Hilfe von:

  ```
  <Print ref="Def/DefIdx/@asmname"/>
  ```

  folgende Ausgabe erreicht:

  ```
  RB
  ```

Eine vollständige Beschreibung einer Instruktion könnte folgendermaßen aussehen:

```
<Print direct="CP "/>
<Print ref="Def/DefIdx/@asmname"/>
<Print direct=", "/>
<Print ref="Arg[2]/ArgIdx/@asmname"/>
<Print direct="\n"/>
```

Diese würde für das oben angegebene Beispiel folgenden Text erzeugen:

```
CP RB, RA
```

3. Nachdem alle Pattern vollständig beschrieben sind, wird ein XeLIR-Dokument komplett sequentiell durchlaufen und für jede MO nach einem passenden Pattern gesucht. In dem jeweils gefundenen Pattern werden dann die Ausgabevorschriften umgesetzt. Allerdings reicht das Durchlaufen alleine nicht aus, um einen vollständigen Assemblercode zu erzeugen. Für verschiedene Aufgaben ist die Definition von weiteren Pattern notwendig:

   - Neben Maschinenoperationen sind Sprungadressen eine wichtige Komponente, die unterstützt werden muss. Die Instruktionen werden in einem speziellen *LabelPattern* definiert. Zu Beginn jedes Basisblocks wird für jedes definierte Label dieses Pattern ausgeführt. Anhand des folgenden Beispiels soll dies verdeutlicht werden. Das folgende *Label*-Element aus einem XeLIR-Dokument

     ```
     <Label name="LL2" stab_id="56"/>
     ```

     und das Pattern

     ```
       <LabelPattern>
         <Instruction>
           <Print ref="@name"/>
           <Print direct=":\n"/>
         </Instruction>
       </LabelPattern>
     ```

     erzeugen

     ```
     LL2:
     ```

     Der Kontextknoten des XPath-Ausdrucks ist also das <Label>-Element.

   - Einzelne Maschineninstruktionen werden auf verschiedene Weise voneinander getrennt. Die Trennung kann über das <InstructionSeparation>-Element festgelegt werden:

     ```
       <InstructionSeparation>
         <Instruction>
           <Print direct=";\n"/>
         </Instruction>
       </InstructionSeparation>
     ```

     Dieses Beispiel trennt einzelne Maschineninstruktionen mit einem Semikolon und fügt zusätzlich einen Zeilenumbruch ein.

   - Neben ganzen Maschineninstruktionen wird Parallelität von Maschinenoperationen meistens durch ein spezielles Zeichen getrennt. Dieses wird innerhalb eines *ParallelSeparation*-Elementes definiert.

     ```
       <ParallelSeparation>
         <Instruction>
     ```

```
      <Print direct=","/>
    </Instruction>
  </ParallelSeparation>
```

- Um einen Standardheader in einem Assemblerprogramm zu definieren, werden entsprechende Ausgaben innerhalb eines *ProloguePattern*-Elementes definiert:

  ```
  <ProloguePattern>
    <Instruction>
      <Print direct="*** begin ***\n"/>
    </Instruction>
  </ProloguePattern>
  ```

- Das Gegenstück zum ProloguePattern für das Ende des Assemblerprogramms ist das *EpiloguePattern*-Element:

  ```
  <EpiloguePattern>
    <Instruction>
      <Print direct="*** end ***\n"/>
    </Instruction>
  </EpiloguePattern>
  ```

Von diesen speziellen Pattern ist nur im LabelPattern eine indirekte Anweisung erlaubt. Alle anderen gestatten nur die Angabe von fest verdrahtetem Text. Alle Pattern, auch die Standard-Pattern, sind direkte Kindelemente eines <Patterns>-Elementes, das gleichzeitig das Root-Element des XML-Pattern-Dokuments ist.

4. Für einige Architekturen kann es notwendig sein, einige Zusatzinformationen in einem Postpass-Prozess hinzuzufügen. Dies können z.B. spezielle Instruktionen für ein Datensegment und einige Variablendefinitionen sein. Da diese aber häufig ein ähnliches Schema haben, kann diese Nachbearbeitung in vielen Fällen automatisiert werden und bedeutet nur einen geringfügigen Mehraufwand.

Um alle Informationen, die für das Schreiben des Assemblercodes nötig sind nicht aufwändig über die Referenz aus der Zielarchitekturbeschreibung ermitteln zu müssen, bietet es sich an, zusätzliche Informationen in jede MO zu schreiben. Ein Beispiel hierfür ist das *asmname*-Attribut, das den Assemblernamen der Registerfile-Ressourcen in den Lokationen angibt. Die Zahl der zusätzlich benötigten Informationen ist gering, so dass diese weder große Änderungen an der GeLIR2XeLIR-Schnittstelle verlangen noch die Struktur des XeLIR-Dokuments verändern.

Die Assemblercode-Generierung ist für die ARM-Architektur beispielhaft umgesetzt und im Rahmen einer anderen Diplomarbeit eingesetzt worden. Dabei wurden mehrere Sortieralgorithmen mit GeLIR dargestellt und nach XeLIR übertragen. Insgesamt wurden in allen XeLIR-Dokumenten 37 verschiedene Pattern gefunden, für die Ausgabevorschriften definiert werden mussten. Für jedes Pattern sind dabei zwischen zwei bis neun Instruktionen nötig gewesen, um die Generierung ausreichend zu beschreiben. Zusätzlich wurde durch das Patternmatching garantiert, dass alle nötigen Fälle für die Assemblercode-Generierung für diese Beispiele abgedeckt wurden.

Eine Überlegung, die in diesem Zusammenhang eine Rolle spielt, ist eine Anpassung des Schemas an die strengeren Anforderungen an ein XeLIR-Dokument für die Assemblercode-Generierung (vgl. Abb.1.2). Die Generierung verlangt eine eindeutige Überdeckung der Maschinenoperationen. Dies kann im Schema durch eine kleine Änderung sichergestellt werden. Die ursprüngliche Definition erlaubt beliebig viele Alternativen:

```
<xs:element name="Def">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="DefAlt" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element ref="DefIdx" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
    ...
  </xs:complexType>
</xs:element>
```

Folgende Deklaration erzwingt genau eine Alternative und optional einen speziellen Index einer Registerfile-Ressource:

```
      <xs:element ref="DefAlt" minOccurs="1" maxOccurs="1"/>
      <xs:element ref="DefIdx" minOccurs="0" maxOccurs="1"/>
```

Diese Änderungen betreffen neben der Definition auch die Argumente und die alternativen Operationen, Funktionseinheiten und Instruktionstypen einer MO.

6.3 Peepholeoptimierungen mit Pattern

Eine weitere Anwendungsmöglichkeit sind die schon in Kap.2.4.2 besprochenen Peepholeoptimierungen. Auch hier steht im Vordergrund, durch die Definition eines Pattern eine möglichst einfache Implementierung einer speziellen Optimierung zu ermöglichen. Beispielhaft werden in diesem Kapitel Möglichkeiten des Findens und Eliminierens überflüssiger algebraischer Operationen, einfache algebraische Transformationen und die Eliminierung überflüssigen Ladens/Speicherns vorgestellt. Unerreichbarer Code oder Kontrollflussoptimierungen sind schneller und einfacher direkt in GeLIR zu realisieren, da die dort vorhandenen Informationen zum Kontrollflussgraph genutzt werden können. Die Ausnutzung der Eigenheiten einer Zielmaschine bieteteine weitere Einsatzmöglichkeit für Pattern.

Bei Peepholeoptimierungen mit Pattern werden die Muster und Ausgabevorschriften, die im Folgenden Instruktionen genannt werden, manuell erstellt. Anschließend wird mit einer Applikation in einem XeLIR-Dokument nach Komponenten gesucht, bei denen diese

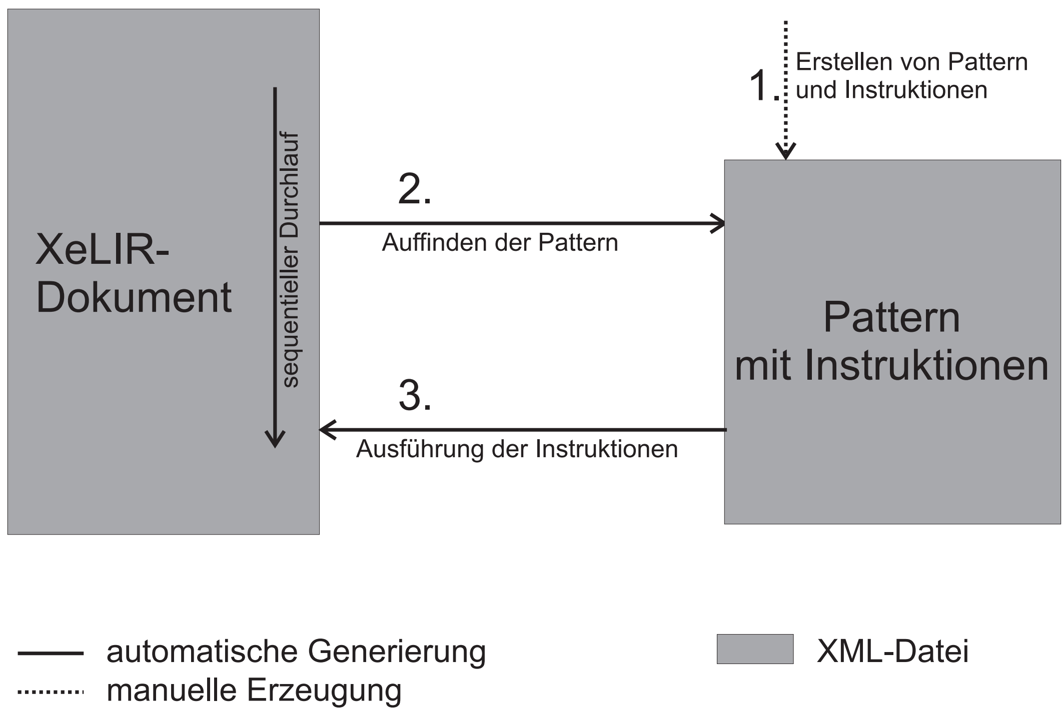

Abb. 6.3: Ablauf von Peepholeoptimierungen mit Pattern

Muster passen, und die entsprechenden Instruktionen ausgeführt (vgl. Abb. 6.3). Peepholeoptimierungen stellen stärkere Anforderungen an die Definition von Mustern und an die Instruktionen. Für das Erkennen überflüssiger algebraischer Operationen wie

```
A = A + 0
```

ist es nötig, die Übereinstimmung der Definition mit dem ersten Argument zu erkennen. Dies kann über einen Vergleich der zugehörigen Symboltabelleneinträge geschehen. Die im vorigen Kapitel vorgestellten Pattern erlauben jedoch nur statische Vergleiche und keine Überprüfung variabler Inhalte wie Referenzen. Deshalb ist es nötig, in einem Pattern Platzhalter zu definieren, die beim ersten Auftreten mit dem Wert gesetzt werden, der im XeLIR-Dokument in der gerade betrachteten MO steht, und beim weiteren Patternvergleich den identischen Wert haben müssen. Dazu ein Beispiel, in dem die Variablen durch ein $-Zeichen eingeleitet werden.

```
<OPR>
   <Def stab_id="$NAME"/>
   <Arg stab_id="$NAME"/>
   <Arg>
      <CONST>
         <Def/>
```

```
        <ConstData value="1"/>
      </CONST>
   </Arg>
</OPR>
```

Wird nun die folgende abstrakte MO in einem XeLIR-Dokument mit dem Pattern verglichen,

```
<OPR  id="MO-290" class2="ADDR_OPR" op_name="+" op_sym="2" value_no="290">
  <Def name="lir_140" stab_id="140"/>
  <Arg name="lir_130" stab_id="130"/>
  <Arg name="lir_139" stab_id="139"  volatile="true">
     <CONST  class2="ADDR_OPR" id="MO-289" value_no="7">
        <Def name="lir_139" stab_id="139"/>
        <ConstData type="INT" value="1" stab_id="138"/>
     </CONST>
  </Arg>
</OPR>
```

dann wird die Patternvariable *\$NAME* in der Definition mit dem Wert *140* initialisiert. Beim weiteren Vergleich muss nun an den Stellen, an denen diese Variable im Pattern gesetzt ist, an der entsprechenden Stelle in der MO der Wert mit dem initialisierten Variablenwert übereinstimmen. Im obigen Beispiel passt das Muster also nicht, da der Symboltabelleneintrag für das Argument sich von dem der Definition unterscheidet, obwohl im Pattern durch die Variable vorgegeben ist, dass sie den gleichen Wert haben müssen.

Neben dieser Erweiterung in der Patterndefinition sind auch wesentliche Erweiterungen bei den Instruktionen nötig. Für die Ausführung algebraischer Transformationen sind die folgenden Instruktionen notwendig. Für deren Umsetzung wurden entsprechende Implementierungen in der Applikation, die die Peepholeoptimierungen ausführt, notwendig.

- Nachdem überflüssige algebraische Operationen gefunden werden, müssen sie per Instruktion gelöscht werden. Dies geschieht durch die Instruktion <**Remove**>.
- Für algebraische Transformationen ist es nötig, bestimmte Elemente und Attributwerte zu überschreiben. Bleibt die Elementstruktur erhalten und müssen nur einige Elementnamen und Attributwerte geändert werden, so geschieht dies mit der Instruktion <**Modify**>, an der eine vollständige MO-Struktur hängt, die nur die zu modifizierenden Attribute enthält. Folgendes Beispiel würde die gesamte Struktur des obigen Beispiels beibehalten und nur den Wert der Konstanten ändern:

  ```
  <Modify>
    <OPR>
       <Def/>
       <Arg/>
  ```

```
      <Arg>
         <CONST>
            <Def/>
            <ConstData value="2"/>
         </CONST>
      </Arg>
   </OPR>
</Modify>
```

- Während <Modify> nur einzelne Komponenten ändert, ist es häufig notwendig, eine komplett neue Struktur zu beschreiben. Für diese Ersetzung ist die Instruktion <**Replace**> vorgesehen, die als Kindelemente die neue Struktur enthält.

- Für viele Transformationen ist es nötig, neue Symboltabelleneinträge zu definieren. Dies geschieht durch die Instruktion <**InsertSTabEntry**>, bei der der Key und der Name des Eintrages in eine Variable geschrieben wird und bei anschließenden Modifikationen und Ersetzungen benutzt werden kann:

```
<InsertSTabEntry key="$KEY" name="$NAME" class="CONST"
                 scope="LOCAL" type="TYPE"/>
```

- Algebraische Transformation sind i.d.R. mit einer Änderung der Operation verbunden. Da die ID nicht feststeht, muss es möglich sein, diese zur Laufzeit zu ermitteln. Dies geschieht durch die Instruktion <**GetOperationID**>, bei der der Name der Operation übergeben wird und die ID in einer Variablen zurückgeliefert wird. Der folgende Ausdruck ermittelt die ID der Operation MUL und speichert diese in der Variablen OPID.

```
<GetOperationID name="MUL" op_id="$OPID"/>
```

Abschließend werden beispielhaft Muster für verschiedene Optimierungen vorgestellt, zunächst für die Eliminierung überflüssiger algebraischer Operationen.

1. **ADD x,x,0**
   Das folgende Pattern "matcht" alle MOs, die die abstrakte Operation + repräsentieren, deren Definition mit dem ersten Argument übereinstimmt und deren zweites Argument die Konstante 0 ist. Diese Addition mit 0 ist eine überflüssige Operation, deshalb wird in den Instruktionen angegeben eine gefundene MO zu löschen. Das Suchmuster ist in einem neuen speziellen Kindelement *Find*, dies erweitert die Möglichkeiten der Definition von Suchmustern, da diese nicht mehr genau einen RootKnoten haben müssen.

```
<Pattern>
  <Find>
    <OPR op_name="+">
      <Def stab_id="$NAME"/>
      <Arg stab_id="$NAME"/>
      <Arg>
        <CONST>
          <Def/>
          <ConstData value="0"/>
        </CONST>
      </Arg>
    </OPR>
  </Find>
  <Instruction>
    <Remove/>
  </Instruction>
</Pattern>
```

2. **MUL x,x,1**
   Ein weiteres Beispiel nach diesem Muster ist eine Multiplikation mit 1, die in folgendem Pattern beschrieben ist:

```
<Pattern>
  <Find>
    <OPR op_name="*">
      <Def stab_id="$NAME"/>
      <Arg stab_id="$NAME"/>
      <Arg>
        <CONST>
          <Def/>
          <ConstData value="1"/>
        </CONST>
      </Arg>
    </OPR>
  </Find>
  <Instruction>
    <Remove/>
  </Instruction>
</Pattern>
```

Neben dem Finden und Eliminieren überflüssiger algebraischer Operationen sind algebraische Transformationen eine Möglichkeit für Peepholeoptimierungen:

- **MUL y,x,2 → SHL y,x,1**
  Das folgende Pattern findet alle MOs, die eine Multiplikation mit 2 repräsentieren. Diese sollen in ein Shiftleft 1 umgewandelt werden. Für diese Transformation wird zunächst die Konstante 1 zu der Symboltabelle hinzugefügt. Im nächsten Schritt wird die ID der SHL-Operation (<< ist eine Escape-Sequenz für <<, da das kleiner Zeichen zur XML-Syntax gehört) ermittelt, und alle nötigen Komponenten in der gefundenen MO werden ersetzt. Dies sind die Attribute *op_name* und *op_sym* im *OPR*-Element und die Attribute *value* und *stab_id* im *ConstData*-Element.

```
<Pattern>
  <Find>
    <OPR op_name="*">
       <Def/>
       <Arg/>
       <Arg>
          <CONST>
             <Def/>
             <ConstData value="2" stab_type="$CONSTTYPE"/>
          </CONST>
       </Arg>
    </OPR>
  </Find>
  <Instruction>
     <InsertSTabEntry key="$STABKEY" name="$STABNAME" class="CONST"
                      scope="LOCAL" type="$CONSTTYPE"/>
     <GetOperationID name="&lt;&lt;" op_id="$OPID"/>
     <Modify>
        <OPR op_name="&lt;&lt;" op_sym="$OPID">
           <Def/>
           <Arg/>
           <Arg>
              <CONST>
                 <Def/>
                 <ConstData value="1" stab_id="$STABKEY"/>
              </CONST>
           </Arg>
        </OPR>
     </Modify>
  </Instruction>
</Pattern>
```

Ein weiteres Beispiel für Peepholeoptimierungen ist das Entfernen überflüssiger Datentransfers, für das im Folgenden ein Beispiel gegeben wird, wie dies mit Pattern umgesetzt werden kann.

- **Entfernen überflüssiger Datentransfers**
  Als letztes Beispiel dieses Abschnitts soll ein überflüssiger Datentransfer entdeckt und entfernt werden. Bei zwei aufeinanderfolgenden Datentransfers nach folgendem Muster kann der letzte entfernt werden.

```
A := B
B := A
```

  Da sich die Struktur bei der Optimierung grundlegend ändert, muss der Ausdruck komplett ersetzt werden. Dies erfordert allerdings die vollständige Beschreibung des ersten Datentransfers innerhalb des <Replace>-Elementes. Dies wird durch Variablen realisiert, die für alle elementaren Attribute des Suchmusters definiert werden und im Ersetzungsbaum angewendet werden. Für die eigentliche Mustererkennung werden die Variablen $OPSYM, $ARG1 und $ARG2 benötigt. Die Variable $OPSYM garantiert, dass beide Datentransfers auf derselben abstrakten Operation aufbauen. $ARG1 und $ARG2 garantieren, dass die Transfers jeweils auf dieselben Variablen zugreifen und der Transfer in der zweiten MO in umgekehrter Richtung verläuft. Das vollständige Pattern hat also für die abstrakte Darstellung die folgende Struktur:

```
<Pattern>
  <Find>
    <MI id="$MIID">
      <COPY  id="$MOID1" class2="$MOCLASS"
             op_name="$MOOPNAME" op_sym="$OPSYM" value_no="$MOVALUENO">
        <Def stab_id="$ARG1"/>
        <Arg stab_id="0"/>
        <Arg stab_id="$ARG2"/>
      </COPY>
    </MI>
    <MI id="$MIID2">
      <COPY  id="$MOID2" op_sym="$OPSYM">
         <Def stab_id="$ARG2"/>
         <Arg stab_id="0"/>
         <Arg stab_id="$ARG1"/>
      </COPY>
    </MI>
  </Find>
  <Instruction>
    <Replace>
```

```
          <MI id="$MIID">
            <COPY  id="$MOID1" class2="$MOCLASS"
                   op_name="$MOOPNAME" op_sym="$OPSYM" value_no="$MOVALUENO">
              <Def stab_id="$ARG1"/>
              <Arg stab_id="0"/>
              <Arg stab_id="$ARG2"/>
            </COPY>
          </MI>
        </Replace>
      </Instruction>
    </Pattern>
```

Es zeigt sich, dass XeLIR als Basis für Optimierungen mit Pattern Grenzen erreicht. Die Suchmuster selbst müssen über die eigentliche Musterbeschreibung hinausreichende Informationen enthalten, um die anschließenden Ersetzungen exakt definieren zu können. Zudem werden für die Instruktionen Erweiterungen benötigt, die eine leichte Umsetzung vielfach erschweren. Spezielle Instruktionen, wie etwa das Hinzufügen von Symboltabelleneinträgen, wären einfacher in GeLIR durchzuführen als in einem XML-Dokument. Die Muster ermöglichen für einfache Strukturen aber eine schnelle Umsetzung, die bei einer vollständigen Implementierung in einer Hochsprache entschieden mehr Zeit beanspruchen würde, so dass eventuell die gesamte Implementierung in Frage zu stellen wäre. Da die größte Schwierigkeit in der Umsetzung der Instruktionen liegt, aber die Muster selbst eine gute Möglichkeit bieten, Strukturen in einem XeLIR-Dokument zu finden, bietet sich ein kooperativer Einsatz von GeLIR und XeLIR an, der im nächsten Abschnittbesprochen wird.

6.4 Kooperativer Ansatz von XeLIR und GeLIR

Der kooperative Ansatz besteht darin, GeLIR oder auch anderen Programmen eine Schnittstelle zur Verfügung zu stellen, um mit definierten Pattern nach bestimmten Strukturen in einem XeLIR-Dokument zu suchen. Bei dem Funktionsaufruf, der von der Schnittstelle zur Verfügung gestellt wird, wird das Pattern, das benutzt werden soll, angegeben (1.). Dieses wird im nächsten Schritt (vgl. Abb. 6.4) aus einem Pattern-Dokument herausgesucht (2.) und dann auf einem zur aktuellen GeLIR-Darstellung äquivalenten (3.) XeLIR-Dokument angewendet. Für jede zu diesem Muster passenden Struktur, werden die Informationen an die Applikation zurückgeliefert (4.).

Um auf bestimmte Pattern zuzugreifen, werden diese noch einmal erweitert. Jedes Pattern wird mit einem Namen versehen, der in dem zusätzlichen Attribut *name* im Patternelement abgelegt wird:

```
<Pattern name="PatMul2">
  <!-- Das eigentliche Pattern -->
</Pattern>
```

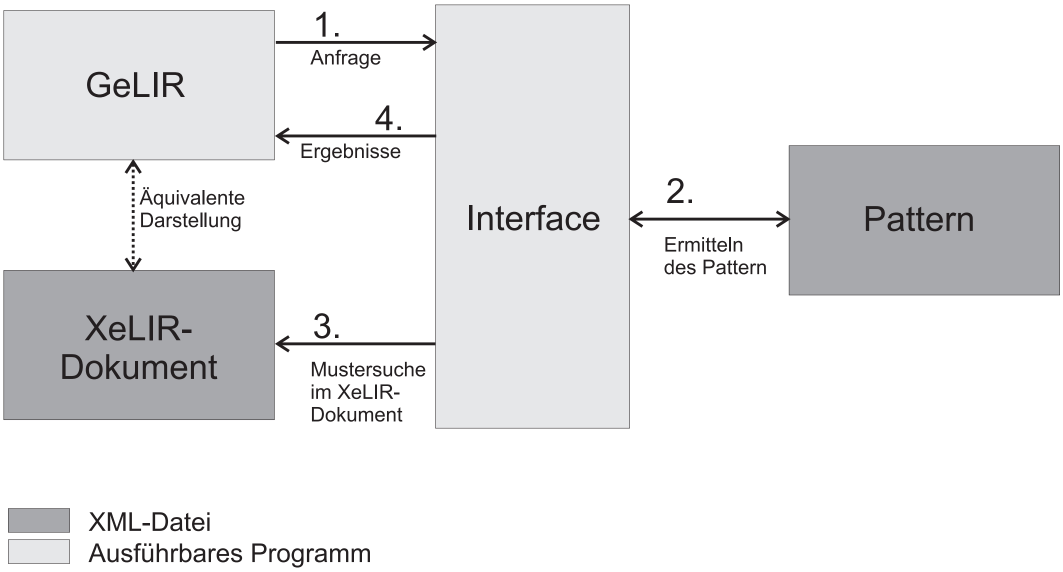

Abb. 6.4: Aufbau des kooperativen Ansatzes von XeLIR und GeLIR

Der eigentliche Datenaustausch geschieht über die Variablen, die im Pattern definiert sind. Alle Variablen und deren Werte werden an die Applikation zurückgeliefert. Um nun bestimmte Attributwerte der zum Pattern passenden Strukturen des XeLIR-Dokuments zu erhalten, muss nur an der entsprechenden Stelle im Pattern eine neue eindeutige Variable hinzugefügt werden. Um z.B. jeweils die ID einer zum im letzten Abschnitt vorgestellten Pattern passenden MO zu erhalten, wird die Variable $MOID hinzugefügt:

```
<Pattern name="PatMul2">
  <Find>
    <OPR id="$MOID" op_name="*">
       <Def/>
       <Arg/>
       <Arg>
          <CONST>
             <Def/>
             <ConstData value="2"/>
          </CONST>
       </Arg>
    </OPR>
  </Find>
</Pattern>
```

Dieses Pattern liefert nun durch den Aufruf[2]

[2]Die Parameter für den Namen der XeLIR- und der Patterndatei sind wegen der Übersichtlichkeit weggelassen worden.

```
ErgebnisListe = FindPattern("PatMul2");
```

eine Liste aller passenden MOs zurück, wobei jeder Eintrag zusätzlich die Variable $MOID und deren Wert enthält. Diese Daten können dann in einem weiteren Schritt dazu benutzt werden, die entsprechenden Maschinenoperationen in GeLIR zu bearbeiten.

Um die Einsatzmöglichkeiten der Pattern zu erhöhen, wird die Schnittstelle dadurch erweitert, dass die Variablen eines Musters im voraus gesetzt werden können. Diese sind dann während der Suche im XeLIR-Dokument konstant. Dadurch wird es z.B. möglich, dass das obige Muster nicht auf die Konstante 2 festgelegt ist, sondern zur Laufzeit beliebig gesetzt werden kann. Dies geschieht durch folgende Umformung:

```
<Pattern name="PatMulX">
  <OPR id="$MOID" op_name="*">
     <Def/>
     <Arg/>
     <Arg>
        <CONST>
           <Def/>
           <ConstData value="$CONSTVALUE"/>
        </CONST>
     </Arg>
  </OPR>
</Pattern>
```

Der Aufruf

```
ErgebnisListe = FindPattern("PatMulX", "CONSTVALUE=2");
```

liefert dann dieselben Ergebnisse wie das obige Muster zurück. Wird die Variable nicht gesetzt, so werden alle MOs zurückgeliefert, die eine Multiplikation repräsentieren und gleichzeitig als zweites Argument eine Konstante mit beliebigem Wert besitzen.

Zum Abschluss dieses Abschnittes werden noch einige Beispiele für einen kooperativen Einsatz vorgestellt. Das folgende Pattern ermöglicht das Suchen nach bestimmten Typdefinition. Dies kann z.B. dazu benutzt werden, Mehrfachdefinitionen zu entdecken.

```
<Pattern name"PatType">
  <Find>
    <Type id="$TYPEID" class="$TYPECLASS"
          name="$TYPENAME" bytes="$TYPESIZE"/>
  </Find>
</Pattern>
```

Werden überflüssige Typen gelöscht, so müssen eventuell noch Referenzierungen umgestellt werden. Dazu ist das folgende Pattern sinnvoll, das zusätzlich auch Instruktionen enthält:

```
<Pattern name"PatTypeRef">
  <Find>
    <Typeref ref="$TYPEID"/>
  </Find>
  <Instruction>
    <Modify>
      <Typeref ref="$NEWTYPEID"/>
    </Modify>
  </Instruction>
</Pattern>
```

Um nun eine Referenzierung von Type-001 nach Type-123 umzustellen, genügt der Aufruf

```
FindPattern("PatTypeRef", "TYPEID=Type-001", "NEWTYPEID=Type-123");
```

Pattern können auch für die Ermittlung der MOs bestimmter Typen genutzt werden. Das folgende Pattern ermittelt z.B. alle VARINs:

```
<Pattern name"PatVarin">
  <Find>
    <VARIN  id="$MOID">
       <Def/>
    </VARIN>
  </Find>
</Pattern>
```

Im letzten Abschnitt wurde schon das Entfernen überflüssiger Datentransfers gezeigt. Als Beispiel wurden zwei aufeinanderfolgende MOs mit folgendem Muster ausgewählt:

```
A := B
B := A
```

Das entsprechende Suchmuster enthielt viele zusätzliche Attribute, um eine vollständige MO im Ersetzungsteil definieren zu können. Eine Alternative ist, die ID der zu löschenden Maschinenoperation an GeLIR zurückzuliefern und diese dort zu löschen. Dadurch werden für die Definition des Musters einerseits keine Instruktionen mehr benötigt und andererseits kann auf Behandlung zusätzlicher Attribute verzichtet werden. Damit hat das Pattern folgende einfachere Struktur:

```
<Pattern>
  <Find>
    <MI id="$MIID1">
      <COPY  id="$MOID1" op_sym="$OPSYM">
        <Def stab_id="$ARG1"/>
        <Arg stab_id="0"/>
        <Arg stab_id="$ARG2"/>
      </COPY>
    </MI>
    <MI id="$MIID2">
      <COPY  id="$MOID2" op_sym="$OPSYM">
         <Def stab_id="$ARG2"/>
         <Arg stab_id="0"/>
         <Arg stab_id="$ARG1"/>
      </COPY>
    </MI>
  </Find>
</Pattern>
```

Bei der Suche nach bestimmten Strukturen mit Pattern wird deutlich, dass sich schon mit einfachen Mustern vielfältige Informationen aus einem XeLIR-Dokument gewinnen lassen. Das letzte Beispiel zeigt zudem, dass mit einem kooperativen Ansatz die Umsetzung von Peepholeoptimierungen vereinfacht werden kann, da die Definition der Muster weniger komplex ist.

Kapitel 7

Zusammenfassung und Ausblick

7.1 Zusammenfassung

In einem Compiler dienen Low-Level Zwischendarstellungen (LIRs) als Austauschformat für die Codegenerierung und für Optimierungen des Backends. Sie besitzen neben der Programmdarstellung auch eine Zielarchitekturbeschreibung. Eine *flexible* Beschreibungsmöglichkeit verbunden mit generischen Optimierungen bietet in großem Umfang Möglichkeiten zur Wiederverwendung für verschiedene Prozessoren. Ein Beispiel für eine LIR, die diese Eigenschaften besitzt, ist die am Lehrstuhl XII der Informatikfakultät Dortmund entwickelte GeLIR. Diese unterstützt im Besonderen die Beschreibung spezieller Eigenschaften von DSPs, wie beispielsweise komplexe Operationen und Parallelität, und kann dabei auch stark irreguläre Architekturen berücksichtigen.

Im Rahmen dieser Diplomarbeit wurde GeLIR mit XeLIR eine textbasierte Zwischendarstellung zur Seite gestellt. Diese erleichtert die Entwicklung einer Codegenerierung und der einzelnen Optimierungen wesentlich, da Zwischenzustände gespeichert und später auf einfache Weise deckungsgleich reproduziert werden können. Da gespeicherte Zustände an andere Benutzer übermittelt werden können, unterstützt XeLIR zudem die Arbeit im Team. Darüber hinaus erlaubt die leicht verständliche und *lesbare* Darstellung gezielte manuelle Änderungen, die z.B. in der Programmdarstellung für Handoptimierungen genutzt werden können. Auch Änderungen an der Beschreibung der Zielarchitektur sind leicht möglich, was z.B. ein schnelles Prototyping erlaubt.

XML hat sich als geeignete Basis für eine solche textbasierte Zwischendarstellung erwiesen. Als standardisiertes Austauschformat profitiert es von einer vielfältigen Tool-Unterstützung. Zudem sind im Umfeld von XML mehrere Erweiterungen entwickelt worden, die zusätzliche Erleichterungen und Möglichkeiten bieten:

- Mit einem XML Schema wird die Struktur eines XeLIR-Dokuments detailliert beschrieben. Da diese Beschreibung auch Datentypen einschließt, wird eine begrenzte

Validierung ermöglicht. Diese formale Beschreibung kann neuen Nutzern zusätzlich einen umfassenden Überblick geben, wie Architekturen und Programme in XeLIR beschrieben werden.

- Durch die Standardisierung entfällt die Entwicklung eines eigenen Parsers. Für diese Aufgabe kann eine Reihe frei erhältlicher Bibliotheken genutzt werden.

XeLIR bietet als XML-Dokument eine Elemente-Hierarchie. Diese bietet die Möglichkeit, nach bestimmten Strukturen in einem XeLIR-Dokument zu suchen. Im Falle des Herausschreibens des Assemblercodes vereinfacht dieses Patternmatching den Aufwand des Anwenders erheblich, wie am Beispiel des ARM-Prozessors gezeigt wurde. So ist die Beschreibung der Ableitungsregeln für einzelne Muster die einzige Aufgabe, die dem Anwender verbleibt. Das Auffinden verschiedener Strukturen von Maschinenoperationen sowie die spätere Umsetzung eines XeLIR-Dokuments in einen Assemblercode ist dagegen automatisiert worden. Dabei werden verschiedene syntaktische Besonderheiten des einzelnen Assemblercodes, wie beispielsweise die Trennung paralleler Maschinenoperationen und ganzer Maschineninstruktionen, unterstützt.

Mit Hilfe der Mustererkennung können auch Peepholeoptimierungen auf einem XeLIR-Dokument durchgeführt werden. Der Aufwand für die Beschreibung einfacher Suchmuster ist, wie an verschiedenen Beispielen zu sehen war, äußerst gering. Müssen für die Optimierungen jedoch größere Modifikationen am XeLIR-Dokument vorgenommen werden, so wird in einigen Fällen die Implementierung zusätzlicher Instruktionen notwendig. Dies gilt etwa für das Hinzufügen von Symboltabelleneinträgen, die im Vergleich zu einer Implementierung auf GeLIR-Ebene sehr aufwändig sind.

Ein kooperativer Ansatz zwischen GeLIR und XeLIR verbindet die Vorteile der Mustersuche auf einem XeLIR-Dokument mit der einfachen Manipulationsmöglichkeit in GeLIR. Dies erlaubt zusätzlich eine flexiblere Verwendung von Mustern, da viele Eigenschaften des Musters zur Laufzeit verändert werden können. Außerdem können die Ergebnisse eines Suchvorgangs in einen weiteren einfliessen.

7.2 Ausblick

In dieser Arbeit wurde eine textbasierte Zwischendarstellung entwickelt. Darüber hinaus wurden verschiedene Anwendungsmöglichkeiten erörtert und vorgestellt. Die in der Einleitung formulierten Ziele sind damit, soweit es im Rahmen dieser Diplomarbeit möglich war, bearbeitet und umgesetzt worden. Auf der Grundlage der erreichten Ergebnisse ergeben sich für die Zukunft folgende Anknüpfungspunkte und Fragestellungen:

- XPath erlaubt es, in einem XeLIR-Dokument zu navigieren und einzelne Komponenten zu referenzieren. Da XPath erst zukünftig mit DOM Level-3 aus einer Applikation vollständig angewandt werden kann, war dessen Nutzung in dieser Arbeit nur eingeschränkt möglich. Für die Zukunft gilt es, XPath stärker in XeLIR zu integrieren, um dessen spezifische Vorteile zu nutzen.

- Beim Herausschreiben eines Assemblercodes wird die komplette Programmstruktur erstellt. Es kann dennoch nötig sein nachträglich Informationen hinzuzufügen, wie z.B. einen speziellen Code für die Initialisierung globaler Variablen. Es bleibt zu untersuchen, wie diese Modifikationen automatisiert werden können.

- Für Peepholeoptimierungen sind verschiedene Muster beispielhaft entwickelt worden. Es bietet sich an durch zusätzliche Pattern weitere Nutzungsmöglichkeiten zu erschließen.

Literaturverzeichnis

[And00] Richard Anderson. *XML Professionell.* MITP-Verlag, 2000.

[ASU87] A. Aho, R. Sethi, and J.D. Ullman. *COMPILERS Principles, Techniques and Tools.* Addison-Wesley, 1987.

[Bas95] Steven Bashford. *Code Generation Techniques for Irregular Architectures.* Forschungsbericht Nr.596, LS Informatik XII, Universität Dortmund, 1995.

[CML] Chemical Markup Language (CML). Homepage: http://www.xml-cml.org/.

[CoL] Bashford, Steven: Constraintbasierte Codegenerierung für eingebettete Prozessoren. Website
http://eldorado.uni-dortmund.de:8080/FB4/ls12/forschung/2001/Bashford.

[DOM] Document Object Model (DOM). Website des W3C:
http://www.w3c.org/DOM/.

[FHP00] C.W. Fraser, D.R. Hanson, and T.A. Proebsting. Engineering a Simple, Efficient Code Generator Generator. In *ACM Letters on Programming Languages and Systems, vol. 1, no. 3*, pages 213–226, June 2000.

[FWD+98] G. Fettweis, M. Weiss, W. Drescher, U. Walther, F. Engel, and S. Kobayashi. Breaking new grounds over 3000 MOPS: A broadband mobile multimedia modem DSP. In *Proc. of ICSPAT'98*, pages 1547–1551, Toronto, Canada, 1998.

[GeL] GeLIR - Generic Intermediate Representation. Website
http://ls12-www.cs.uni-dortmund.de/research/gelir/.

[Hun00] David Hunter. *Beginning XML.* Wrox Press Ltd., 2000.

[Lan] LANCE Retargetable C Compiler. Homepage:
http://ls12.cs.uni-dortmund.de/lance.

[Muc97] S. Muchnick. *Advanced Compiler Design and Implementation.* Academic Press, 1997.

[SUI] The Stanford SUIF Compiler Group. SUIF Compiler System Homepage: http://www-suif.stanford.edu/.

[Tei97] Dr. J. Teich. *Digitale Hardware/Software-Systeme*. Springer, 1997.

[Tri] Trimaran Homepage. http://www.trimaran.org/.

[WM97] R. Wilhelm and D. Maurer. *Übersetzerbau*. Springer, 1997.

[Xer] The Apache XML Project. Website unter http://xml.apache.org/.

[XLi] XML Linking Language (XLink). Website des W3C: http://www.w3.org/TR/xlink/.

[XMLa] Namespaces in XML. Website des W3C: http://www.w3.org/TR/REC-xml-names/.

[XMLb] XML-Schema Part 1: Structures. Website des W3C: http://www.w3.org/TR/xmlschema-1/.

[XMLc] XML-Schema Part 2: Datatypes. Website des W3C: http://www.w3.org/TR/xmlschema-2/.

[XMLd] XMLSOFTWARE: The XML Software Site. Homepage: http://www.xmlsoftware.com/.

[XMLe] Extensible Markup Language (XML) 1.0 (Second Edition). Website des W3C: http://www.w3.org/TR/REC-xml.

[XMLf] Extensible Markup Language (XML) 1.0 - Deutsche Übersetzung. http://www.mintert.com/xml/trans/REC-xml-19980210-de.html.

[XPa] XML XPath Language (XPath). Website des W3C: http://www.w3.org/TR/xpath/.

[XPo] XML Pointer Language (XPointer) Version 1.0. Website des W3C: http://www.w3.org/TR/xptr/.

[XSL] Extensible Stylesheet Langauge (XSL). Website des W3C: http://www.w3.org/Style/XSL.