Ein Ada-Compiler

R.P. Wehrum, G. Dießl, W. Hoyer, J. Winkler

0. Einführung

In diesem Beitrag soll über einige Aspekte der Entwicklung eines Compilers für die Programmiersprache Ada berichtet werden. Die Compiler-Entwicklung wurde im Rahmen des Siemens-Ada-Compiler-Projektes (SACP) durchgeführt. Das SACP hat seinen Ursprung im European Ada Compiler Project, welches von 1981 bis 1984 abgewickelt und während dieser Zeit von der Kommission der Europäischen Gemeinschaft finanziell gefördert wurde. Unsere Kooperationspartner in diesem Projekt waren die beiden Firmen Bull S.A. und Alsys S.A. aus Frankreich.

Das Hauptziel des ursprünglichen europäischen Projektes und des SACP war, einen übertragbaren und anpaßbaren Compiler für ANSI-Ada zu entwickeln. Gast- und Zielrechner für den Ada-Compiler sind im SACP zunächst die Maschinen der Siemens-Serie 7000 unter dem Betriebssystem BS2000.

In diesem Beitrag werden drei Themen mit einiger Ausführlichkeit behandelt: (1) die im Compiler verwendeten Zwischensprachen, (2) die Struktur des Compilers und (3) der Bootstrapping-Prozeß.

Zwischensprachen stellen wichtige Architekturmerkmale für einen Compiler dar. Sie bestimmen die internen und externen Schnittstellen des Compilers und haben somit Einfluß auf die Compiler-Struktur. Es ist vorteilhaft, einen Compiler in der gleichen Sprache zu schreiben, für die er gebaut wird. Aus diesem Grunde ist der Ada-Compiler im SACP in Ada geschrieben worden. Der in diesem Fall notwendige Bootstrapping-Prozeß ist eine Methode, durch Selbstanwendung eine ablauffähige Version des Compilers zu erzeugen.

1. Zwischensprachen

1.1 Der Einfluß der Übertragbarkeit auf die Compilerstruktur

Die Forderung nach *Übertragbarkeit* und *Anpaßbarkeit* beeinflußt den Entwurf des Compilers wesentlich. Übertragbarkeit heißt, daß der Compiler leicht auf einer Reihe verschiedener Gastrechner mit unterschiedlichen Architekturen und Betriebssystemen zum Ablauf gebracht werden kann. Anpaßbarkeit heißt, daß der Compiler leicht an eine entsprechende Reihe unterschiedlicher Zielrechner angepaßt werden kann. Um weitgehende Übertragbarkeit und Anpaßbarkeit zu erreichen, müssen die Teile des Compilers, die

vom Gastrechner oder Zielrechner abhängen, in möglichst wenig Bausteinen konzentriert werden.

Abhängigkeiten von diesen Rechnern sind bedingt durch:

- Attribute von Ada, die von der Zielmaschine abhängen,
- Schnittstellen des Compilers zu dem Rechnersystem, auf welchem der Compiler abläuft,
- Schnittstellen des Compilers zum Zielsystem, für das Code erzeugt werden soll.

Im sog. *Root-Compiler*, der den Analyseteil und den Transformationsteil des Compilers umfaßt, wird die Konzentration der Systemabhängigkeiten folgendermaßen realisiert:

- die Schnittstellen aller beteiligten Algorithmen sind als systemunabhängige Spezifikationen von Ada-Programmeinheiten formuliert;
- die Implementierung der meisten Algorithmen ist realisiert in Form *systemunabhängiger Rümpfe*. Diese Rümpfe benutzen *systemabhängige Parameter*, die ihrerseits in speziellen Ada-Paketen zusammengefaßt sind;
- einige wenige Rümpfe sind systemabhängig formuliert.

Die Übertragung des Compilers auf ein anderes Rechnersystem erfordert die Anpassung der Parameter-Pakete und der systemabhängigen Rümpfe. Im Idealfall müssen weder die Schnittstellen noch die systemunabhängigen Algorithmen modifiziert werden. Dadurch wird die Übertragung des Compilers zu einer klar umrissenen Aufgabe von begrenztem Umfang.

Um den Compiler auch auf Rechnersystemen mit kleinen Speichern bzw. Adreßräumen ablaufen lassen zu können, wurde im Compiler eine eigene virtuelle Speicherverwaltung auf Seitenbasis realisiert. Dieser virtuelle Speicher wird dazu benutzt, die Interndarstellungen des zu übersetzenden Programmes abzuspeichern.

1.2 Interne und externe Schnittstellen

Für eine komplexe Sprache wie Ada erscheint es schwierig, die Übersetzung in einem Schritt, d.h. in einem Paß, durchzuführen. Es sind mehrere Pässe erforderlich, und der Compiler besteht daher aus einer Reihe von Phasen. Als Folge davon gibt es mehrere *interne Darstellungen* des zu übersetzenden Programmes, die in verschiedenen *Zwischensprachen* formuliert sind. Die Eigenschaften dieser Zwischensprachen werden durch interne und externe Anforderungen beeinflußt. Die getrennte Übersetzbarkeit und die sprachorientierten Werkzeuge der Programmierumgebung erfordern eine Zwischensprache, die noch relativ nahe zum Quellprogramm ist und die im *Bibliothekssystem* gespeichert werden kann. Eine weitere Zwischensprache, die der Maschinensprache des Zielrechners nähersteht, dient als Schnittstelle zu den Codegeneratoren.

Im Siemens-Ada-Compiler werden derzeit zwei verschiedene *baumstrukturierte* Zwischensprachen benutzt: die Höhere Zwischensprache (HLIL = High Level Intermediate Language) und die Niedere Zwischensprache (LLIL = Low Level Intermediate Language). Die HLIL ist eine Modifikation von DIANA (= Descriptive Intermediate Attributed Notation for Ada), welche von Goos, Wulf und Mitarbeitern entwickelt worden ist [2]. Ein HLIL-Programm ist die interne Repräsentation eines lexikalisch, syntaktisch und semantisch analysierten Ada-Programmes.

Diese Interndarstellung ist ein attributierter abstrakter Syntaxbaum. (Genau genommen ist die Interndarstellung ein allgemeiner, gerichteter Graph, welcher aus dem abstrakten Syntaxbaum durch Hinzunahme der semantischen Attribute und ihrer Beziehungen entsteht.) Die HLIL-Darstellung kann als eine Schnittstelle für alle sprachorientierten Werkzeuge einer Programmierumgebung dienen, z.B. für einen syntaxorientierten Editor. Für die getrennte Übersetzung ist es nicht erforderlich und aus Speicherplatzgründen auch nicht erwünscht, die HLIL-Darstellung einer Ada-Übersetzungseinheit vollständig abzuspeichern. Es genügen vielmehr die Teile davon, welche die Schnittstelle der betreffenden Übersetzungseinheit zu anderen Übersetzungseinheiten beschreiben. In der HLIL sind diese Teile durch wohl abgegrenzte Teilstrukturen dargestellt und können daher zum Zwecke der längerfristigen Speicherung in der Ada-Bibliothek leicht abgespalten werden. Funktionell spielen diese Teile die Rolle von *Symboltabellen* oder *Größenverzeichnissen*. Um den Kontext für eine Übersetzungseinheit U herzustellen, ist es dann nur erforderlich, die Größenverzeichnisse der Übersetzungseinheiten zu konsultieren, die von U direkt oder indirekt referiert werden können. Die Größenverzeichnisse dienen auch als Schnittstelle zur symbolischen Testhilfe, einem wichtigen Werkzeug einer Ada-Programmierumgebung.

Da die HLIL die Quellform unmittelbar widerspiegelt, wäre es ziemlich kompliziert, daraus direkt Maschinencode zu erzeugen. Im Siemens Ada-Übersetzer wird dieser Schritt daher in mehrere Teilschritte unterteilt. Zu diesem Zweck wird eine weitere Zwischensprache, die oben bereits erwähnte LLIL, eingeführt. LLIL ist näher an der Maschine und dient daher als Eingangsschnittstelle für die Codegeneratoren.

LLIL ist aber dennoch, ebenso wie HLIL, eine *maschinenunabhängige* Sprache und ist ebenfalls baumstrukturiert. Im Gegensatz zur HLIL ist die LLIL eine flexiblere Notation, in welcher Entwurfsentscheidungen der Codeerzeugung leicht berücksichtigt werden können. Die folgenden Beispiele geben einen Eindruck von den Unterschieden zwischen HLIL und LLIL :

- *Zugriffspfade* zu Objekten und Werten sind in der LLIL im Gegensatz zur HLIL voll expandiert, d.h. die Adreßberechnungen und die Dereferenzierungen werden durch LLIL-*Operationen* ausgedrückt.
- *Laufzeitüberprüfungen*, die in der HLIL nur durch Attribute repräsentiert werden, sind in der LLIL durch entsprechende Operationen dargestellt.


```
procedure DEMO is

    type ARR is array (1 .. 10) of NATURAL;

    Y    :  ARR      :=  (1, 2, others => 100);

    I    :  INTEGER := 10;

    J    :  INTEGER := 1000;

    subtype S is POSITIVE range 1 .. J;

    X: S;

begin

    X := Y(I);

end DEMO;
```

Abb. 1a Ada-Quellcode der Prozedur DEMO

Betrachten wir dazu die rechte Seite der Zuweisung „X := Y(I);" in Abb. 1a. In Abb. 1b ist der zugehörige HLIL-Teilbaum dargestellt. Der Baum für die rechte Seite „Y(I)" ist ein Teilbaum davon, und zwar ist es der rechte, untere Teilbaum mit einem *Indexed*-Knoten als Wurzel.

Im obigen Kontext bedeutet „Y(I)" den Wert eines Objektes, dessen Adresse bestimmt wird durch die Anfangsadresse der Reihungsvariablen Y und der Relativadresse des I-ten Elementes von Y. In Abb. 1c ist dies explizit im linken Unterbaum des *binary_assign*-Knotens ausgedrückt: der Baum unter dem *array_offset*-Knoten stellt die Berechnung der Adresse für Y(I) dar und der *value*-Knoten direkt darüber repräsentiert die Dereferenzierung, um den Wert von Y(I) zu beschaffen.

Der HLIL-Knoten *indexed* in Abb. 1b enthält ein Attribut, welches besagt, daß eine Indexüberprüfung durchgeführt werden muß. D.h., es muß überprüft werden, ob der Wert des Ausdrucks, der durch den rechten Sohn von *indexed* dargestellt wird (hier der Wert von I), zum Wertebereich des Index-Untertyps des Typs der Variablen, welche durch den linken Sohn von *indexed* dargestellt wird (hier das Reihungsobjekt Y). Diese Semantik, die sich in der HLIL implizit durch Attribute ausdrückt, wird in der LLIL explizit dargestellt durch den Knoten *binary_range_check*. Dadurch wird der Wert von I mit unterer und oberer Grenze des Indexbereiches verglichen und der Wert von I nur dann an den Vaterknoten

binary_minus weitergegeben, wenn die Indexbereichsgrenzen nicht verletzt werden. Andernfalls liefert die Überprüfung einen Fehler, und dadurch wird dann die Ausnahme CONSTRAINT_ERROR ausgelöst. Die Zuweisung „X = Y(I);" enthält noch eine weitere Überprüfung. Diese wird im HLIL-Knoten *assignment* (Abb. 1b) durch das Attribut *range_check* ausgedrückt und in der LLIL durch den oberen *binary_range_check*-Knoten (Abb. 1c).

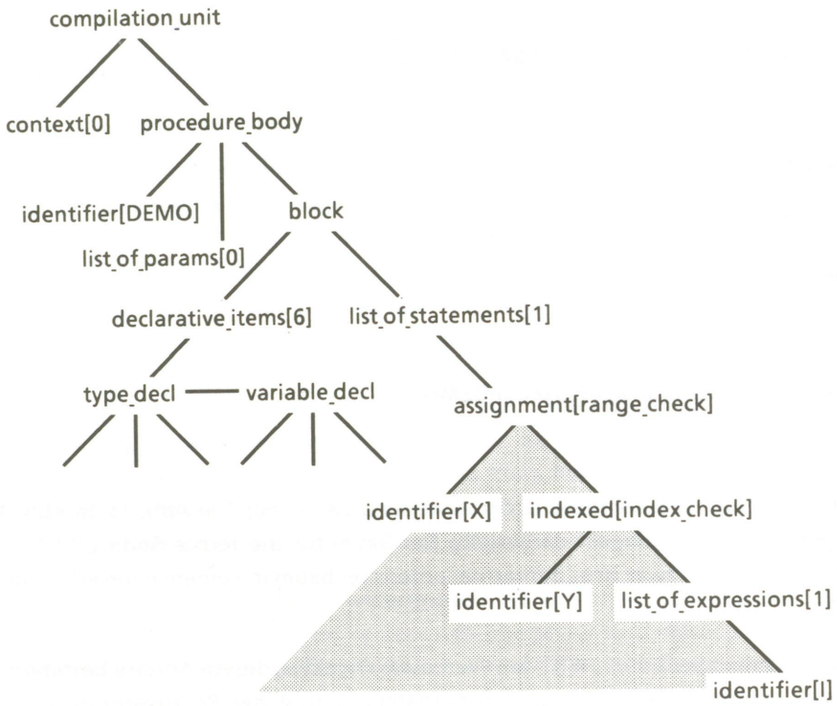

Abb. 1b **Die HLIL-Form der Prozedur DEMO. Das schattierte Dreieck repräsentiert die Zuweisung „X : = Y(I); " von Abb. 1a.**

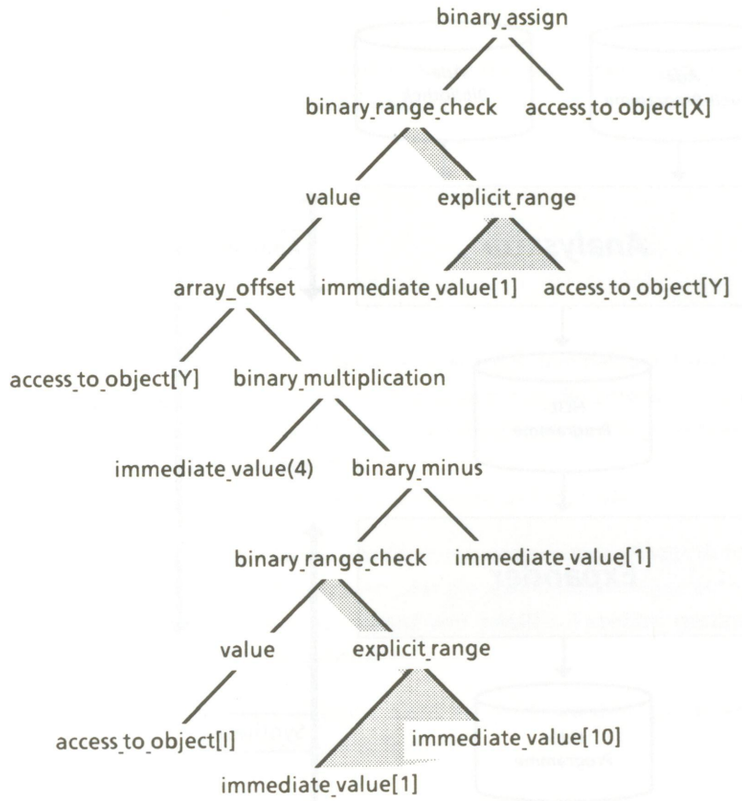

Abb. 1c **Die LLIL-Form des Zuweisungsunterbaums von Abb. 1b (schattiertes Dreieck in Abb. 1b). Die schattierten Bereiche der obigen Abb. zeigen jene Knoten, die für Laufzeitüberprüfungen eingeführt wurden.**

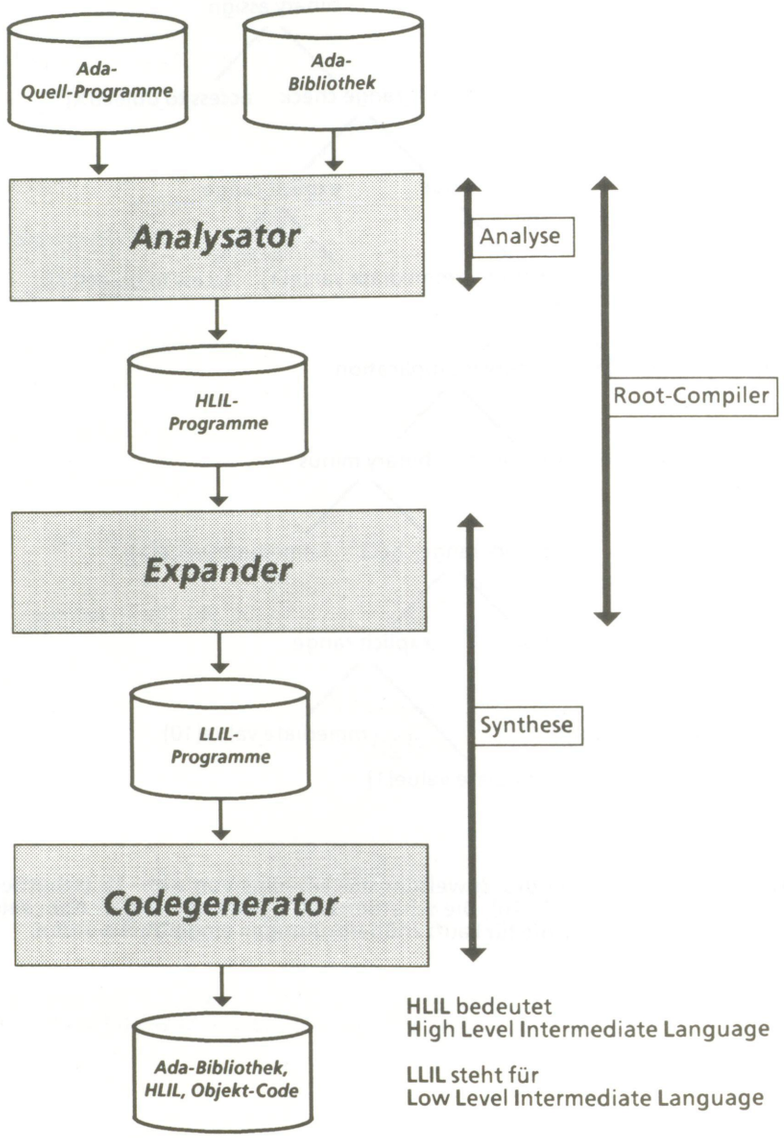

Abb. 2 Struktur des Siemens-Ada-Compilers

2. Struktur des Compilers

Der Compiler setzt sich aus drei verschiedenen Teilen zusammen: dem Analysator, dem Expander und dem Codegenerator (siehe Abb. 2).

2.1 Der Analysator

Der *Analysator*

- liest das Quellprogramm;
- baut den Kontext für die Übersetzung auf, d.h. sammelt alle Tabellen und Einträge von Übersetzungseinheiten zusammen, die bereits erfolgreich übersetzt wurden und die vom zu übersetzenden Programm benötigt werden;
- analysiert das Quellprogramm;
- generiert einen attributierten Syntaxbaum, den HLIL-Baum;

Lexikalische und syntaktische Analyse werden zusammen in einer Phase durchgeführt. Die Syntaxanalyse fungiert als Hauptprogramm, das die lexikalische Analyse aufruft. Der Parser arbeitet im Bottom-up-Verfahren und wird von LALR(1)-Tabellen gesteuert, die vom Parsergenerator SYNTAX [3] erzeugt wurden.

Um eine einzelne Phase des Compilers nicht zu groß werden zu lassen, wurde die semantische Analyse in zwei Teilphasen aufgeteilt.

In diesen zwei Phasen

- wird die statische Semantik allgemein überprüft;
- findet eine umfangreiche Überprüfung der Namen statt;
- werden insbesondere die überladenen Namen von Unterprogrammen richtig zugeordnet;
- werden die statischen „universellen" Ausdrücke mit Hilfe einer Rationalzahlarithmetik errechnet (diese arbeitet mit geordneten Paaren von Superzahlen, die durch ganzzahlige Werte beliebiger Größe dargestellt werden);
- werden generische Ausprägungen (ähnlich einer Makro-Expansion) abgearbeitet;
- werden die Ada-Typen implementiert, d.h. das Layout von Objekten wird festgelegt unter Berücksichtigung der Darstellungsklauseln;
- wird bestimmt, bei welchen Konstrukten des aktuellen Programms dynamische Überprüfungen zur Laufzeit durchgeführt werden müssen. Die entsprechenden Konstrukte werden im HLIL-Baum durch zusätzliche Attribute gekennzeichnet.

Der Analysator ist nur insoweit maschinenabhängig, als er Eigenschaften der Maschine zu berücksichtigen hat, die durch die Sprache selbst eingeführt werden. Das folgende Beispiel illustriert diesen Sachverhalt:

type INT **is range** 0 .. 1E7 * T'SIZE;

wobei T einen beliebigen statischen Typ bezeichnet.

Die semantische Analyse muß die Korrektheit oder Inkorrektheit dieser Typ-Deklaration feststellen. Zu diesem Zweck muß sie prüfen, ob es einen vordefinierten ganzzahligen Typ gibt, auf den INT abgebildet werden kann. In dem angeführten Beispiel muß sie den Wert des statischen Ausdrucks T'SIZE kennen, der von der Zielmaschine und von einer speziellen Implementierungsentscheidung abhängt.

Der Ada-Compiler ist in Ada geschrieben. Dies hat unter anderem den Vorteil, daß die Implementierung direkt von Sprachkonzepten profitiert, die die Prinzipien des Software-Engineering unterstützen. Ada bietet beispielsweise ausgefeilte Konstrukte für die Modularisierung an.

Abb. 3 offenbart die modulare Struktur der Phase 3 des Compilers, die sich mit der semantischen Analyse befaßt. Der Graph von Abb. 3 zeigt die Übersetzungseinheiten und ihre Abhängigkeiten. Die Knoten im Graphen stehen für Übersetzungseinheiten, die Kanten repräsentieren die Abhängigkeiten. Die Richtung der Kanten veranschaulicht die Sichtbarkeit: eine gerichtete Kante verbindet die benutzende mit der deklarierenden Einheit. Das Hauptprogramm dieser Compilerphase heißt Semantic_Checks_Part_2. Es wurde mit Hilfe mehrer *Untereinheiten* realisiert (top-down-Verfahren). Im Bild wird dies durch die baumartigen Muster unter dem Knoten, der das Hauptprogramm repräsentiert, verdeutlicht. Andererseits werden gemeinsame Ressourcen (z.B. HLIL_Description und Compiler_IO), die vom Hauptprogramm der Phase 3 und von anderen Teilen des Compilers verwendet werden, als *Bibliothekspakete* realisiert (bottom-up-Verfahren).

2.2 Der Expander

Wie bereits erwähnt, ist die HLIL-Darstellung dem Quellprogramm sehr nahestehend. Um Maschinenniveau zu erreichen, ist noch ein langer Weg zurückzulegen. Die Umsetzung von einem HLIL-Programm in ein äquivalentes Maschinenprogramm wurde daher im SACP in zwei aufeinanderfolgende Teilaufgaben aufgeteilt: die erstere Teilaufgabe ist weitgehend maschinenunabhängig und wird vom sog. *Expander* wahrgenommen, die letztere ist maschinenabhängig und wird vom Codegenerator im engeren Sinn besorgt. In traditioneller Sprechweise bilden der Expander und der Codegenerator zusammen das Backend

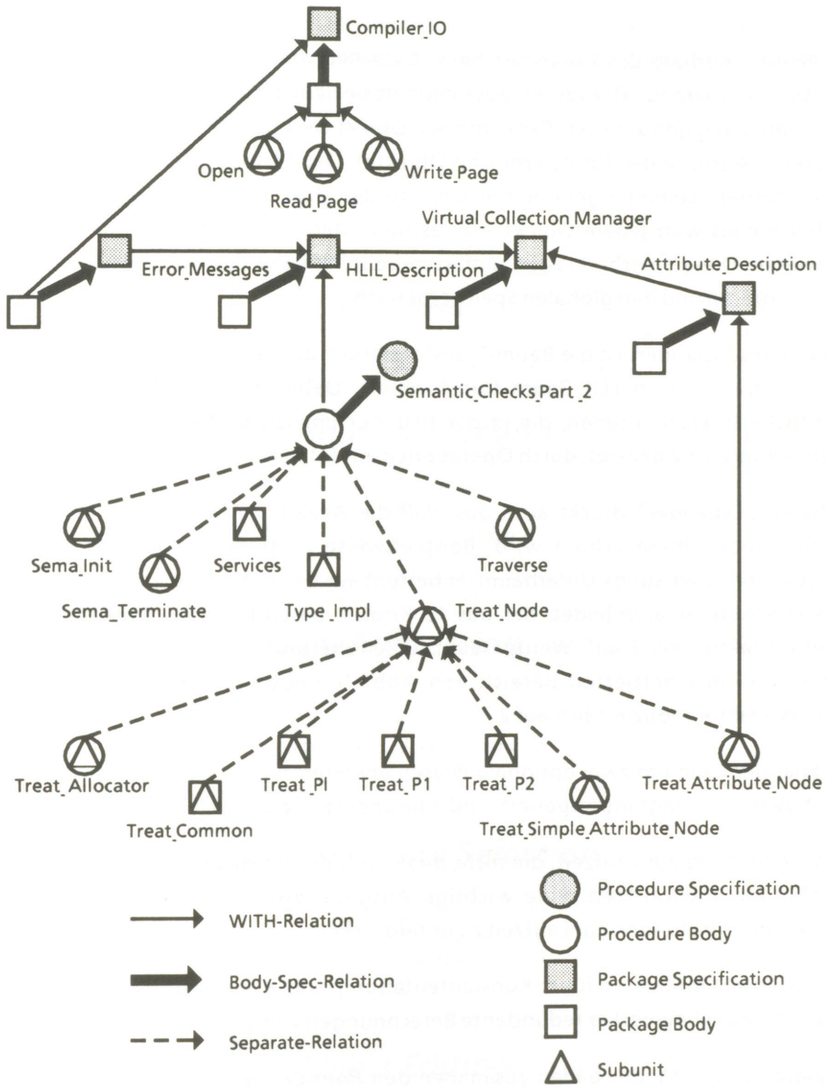

Abb. 3 Ein Ausschnitt aus dem Abhängigkeitsgraphen der dritten Phase des Compilers, die einen Teil der semantischen Analyse abdeckt.

des Compilers, wo die *Synthese* des Maschinenprogramms vorbereitet und durchgeführt wird.

Eine Phase innerhalb des Expanders heißt *Data-Allocation*. Sie weist den Datenobjekten ihren Speicherplatz zu. Darüberhinaus generiert sie neue Objekte, die nicht auf Quellebene erscheinen. Dazu gehören Typ-Deskriptoren, Objekt-Deskriptoren und sogenannte „Dope-Vektoren" (Adressfelder für dynamische Objekte). Data-Allocation fügt die Beschreibung dieser internen, compiler-generierten Objekte der HLIL-Darstellung des Programms an. Darüber hinaus wird jedem Objekt - sei es nun intern oder extern - eine Speicherklasse zugewiesen. Diese Speicherklassen umfassen beispielsweise den Laufzeitkeller, die Halde, den temporären und den globalen Speicherbereich.

Der Kern des Expanders ist die Baum-Transformation, die den HLIL-Baum eines Programms in einen äquivalenten LLIL-Baum transformiert. Dabei wird die Semanik *expandiert*: semantische Interpretationen, die in der HLIL nur implizit vorhanden sind, werden, wie bereits inKapitel 2.2 gezeigt, durch Operationen auf LLIL-Ebene explizit ausgedrückt .

Der Name „Expander" drückt auch aus, daß die Anzahl der Knoten durch die Baum-Transformations-Phase erhöht wird. Beispielsweise zeigt der schattierte Bereich von Abb. 1b einen Zuweisungs-Unterbaum. Er besteht aus 6 HLIL-Knoten und wird auf den LLIL-Baum von Abb. 1c abgebildet, der aus 19 Knoten besteht. Dieses Beispiel weist einen Expansionsfaktor von 3 auf. Werden die Laufzeitüberprüfungen unterdrückt, dürfen alle Knoten, die im schattierten Bereich von Abb. 1c liegen, entfernt werden, und der Expansionsfaktor reduziert sich auf 2.

Der Expander kann um zwei optionale Phasen erweitert werden, eine Optimierungsphase, die auf dem HLIL-Programm operiert, und eine andere, die den LLIL-Baum manipuliert.

Jede Überprüfung zur Laufzeit, die nicht direkt von der Hardware unterstützt wird, kostet Speicherplatz und Laufzeit. Eine wichtige Aufgabe des High-Level-Optimierers ist es, unnötige Überprüfungen zur Laufzeit zu unterdrücken.

Der Low-Level-Optimierer führt Konstantenfaltung und Konstantenpropagierung durch. Darüber hinaus eliminiert er redundante Berechnungen von gemeinsamen Teilausdrücken.

Analysator und Expander bilden zusammen den *Root-Compiler*, der im wesentlichen den maschinenunabhängigen Teil repräsentiert. Er kann leicht auf eine neue Anlage portiert werden (rehosting), und er kann leicht an eine neue Zielmaschine angepaßt werden (retargeting).

2.3 Der Codegenerator

Der Codegenerator transformiert die LLIL-Konstrukte in die Maschinensprache der Siemens-Serie 7000. Das größtenteils in Ada geschriebene Laufzeitsystem ist in das Betriebssystem BS2000 eingebettet. Die Struktur des Codegenerators ist in Abb. 4 dargestellt.

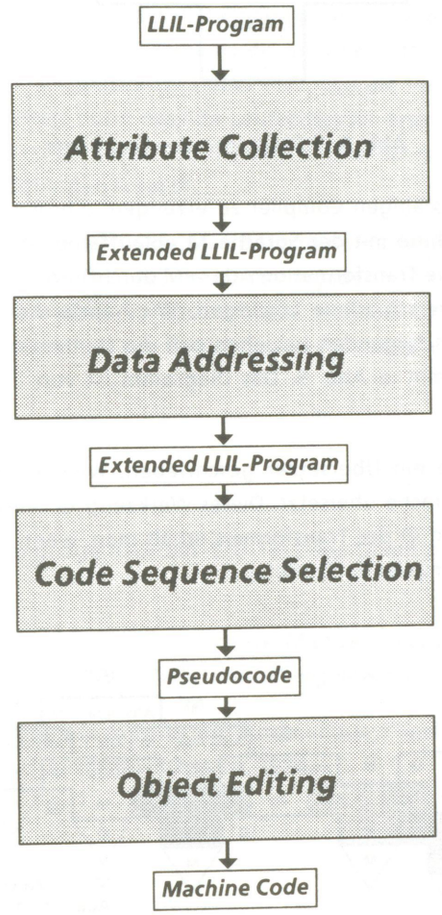

Abb. 4 **Die Struktur des Codegenerators für die Siemensanlagen der Serie 7000**

3. Der Bootstrapping-Prozeß

Der Ada-Compiler ist selbst in Ada geschrieben worden, genauer gesagt in einer Untermenge Ada-1 der vollen Sprache. Dieser Compiler kann durch das nachstehende T-Diagramm dargestellt werden, welches die Quellsprache ANSI-Ada, die Zielsprache M und die Implementierungssprache Ada-1 zeigt. M ist in unserem Fall die Maschinensprache der Siemens-Serie 7000.

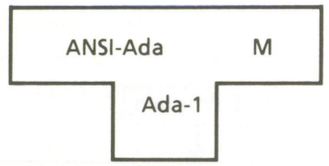

Abb. 5 T-Diagramm des Compilers

Um hieraus einen arbeitsfähigen Compiler zu erzeugen, d.h. einen Compiler in binärem Code, der auf der Maschine mit der Sprache M ablauffähig ist, wird bereits zuvor ein Compiler benötigt, der die Transformation Ada → M durchführt. Dieses scheinbar paradoxe Problem hat mehrere verschiedene Lösungen. Wir haben die in Abb. 6 dargestellte Methode angewandt. Im folgenden beziehen sich die Nummern in den Kreisen auf das entsprechende T-Diagramm in Abb. 6. Das Diagramm ist von links nach rechts, unten beginnend, zu lesen.

Im ersten Schritt wurde ein Übersetzer geschrieben, welcher Ada-1-Quellsprache in äquivalente PL/1-Quellsprache übersetzt. Dieses Werkzeug ② wird Transformer genannt. Eine ablauffähige Version ③ des Transformers erhält man, wenn man seine Quellform ② mit dem Siemens-PL/1-Compiler ① übersetzt.

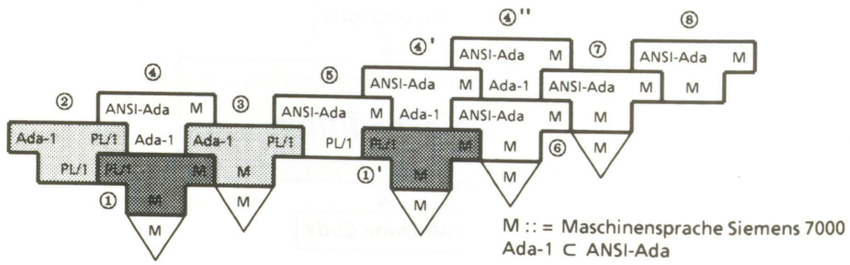

Abb. 6 Bootstrapping-Prozeß des Compilers. Die Dreiecke kennzeichnen den Ablauf der entsprechenden Compiler auf der Maschine M.

Im zweiten Schritt wird die Quellform ④ des ANSI-Ada-Compilers, der in Ada-1 implementiert ist, mit Hilfe von ③ in PL/1-Quellform ⑤ transformiert. Die PL/1-Form ⑤ wird dann im dritten Schritt mit dem PL/1-Compiler ①' übersetzt. Damit erhält man die erste ablauffähige Version ⑥ des Ada-Compilers. (In Abb. 6 bezeichnet eine in einen Kreis eingeschlossene Nummer den Übersetzer eindeutig; Apostrophe zeigen nur wiederholte Anwendungen desselben Programms an. So verweist z.B. ①' auf denselben Compiler wie ①.)

Da der Transformer nur als Bootstrapping-Werkzeug benutzt worden ist, wurde bei seiner Implementierung kein großer Wert auf Effizienz gelegt. Der Ada-Compiler in der Form ⑥ enthält als direktes Ausgabeprodukt des PL/1-Compilers noch viele Ineffizienzen, obgleich er bereits aus Ada endgültigen Maschinencode generiert. Der Grund dafür liegt im Transformer und dem Übersetzungsprozeß über PL/1. Um die Ineffizienzen loszuwerden und um das PL/1-Laufzeitsystem durch das Ada-Laufzeitsystem ersetzen zu können, muß der Ada-Compiler ④' mit ⑥ in einem vierten Schritt noch einmal übersetzt werden. Dies liefert den Compiler in seiner endgültigen Form ⑦.

Zur Überprüfung der Korrektheit des gesamten Bootstrapping-Prozesses kann eine weitere Übersetzung durchgeführt werden: ④'' dient als Eingabe in ⑦, wobei als Ausgabe ⑧ erzeugt wird. Falls sich ⑦ hierbei reproduziert, d.h. wenn ⑦ und ⑧ sich als identisch herausstellen, kann die Korrektheit des Bootstrapping-Prozesses als hinreichend gesichert gelten.

4. Literatur

[1] Ada Programming Language ANSI/MIL-STD-18. Department of Defense. Washington, D.C. 1983.

[2] Goos, G.; Wulf, W.A. eds.: Diana Reference Manual. Lecture Notes in Computer Science, No. 161. Springer Verlag, Berlin Heidelberg New York, 1983.

[3] Boullier, P.: Contribution à la construction automatique d'analyseurs lexicographiques et syntaxiques. Thèse L'Université d'Orléans, 1984.

Informatik in der Praxis

Aspekte ihrer industriellen Nutzanwendung

Herausgegeben von H. Schwärtzel

Mit 177 Abbildungen

Springer-Verlag Berlin Heidelberg New York
London Paris Tokyo 1986

Informatik in der Praxis

Aspekte ihrer industriellen Nutzanwendung

Herausgegeben von H. Schwärtzel

Mit 177 Abbildungen

Springer-Verlag Berlin Heidelberg New York
London Paris Tokyo 1986

Professor Dr. techn. Heinz Schwärtzel
Leiter des Hauptbereichs Zentrale Aufgaben Informationstechnik,
Siemens AG, München

ISBN 3-540-17054-5 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-17054-5 Springer-Verlag New York Heidelberg Berlin

CIP-Kurztitelaufnahme der Deutschen Bibliothek.
Informatik in der Praxis: Aspekte ihrer industriellen Nutzanwendung / hrsg. von H. Schwärtzel.
– Berlin; Heidelberg; New York; London; Paris; Tokyo: Springer, 1986.
ISBN 3-540-17054-5 (Berlin ...)
ISBN 0-387-17054-5 (New York ...)
NE: Schwärtzel, Heinz G. [Hrsg.]

Druck: Color-Druck, Berlin; Bindearbeiten: Lüderitz & Bauer, Berlin
2362/3020 543210

Dem Andenken

von

Karl Heinz Beckurts

gewidmet