RUHR - UNIVERSITÄT BOCHUM

Arbeitsberichte

des

Rechenzentrums

Direktor: Prof. Dr. H. Ehlich

Nr. 7501

Zur Theorie der Montage von Programmoduln

von

Rainard Buchmann

Bochum, März 1975
Universitätsstr. 150, Gebäude NA

INHALTSVERZEICHNIS

1. EINFÜHRUNG

Die vorliegende Arbeit beschäftigt sich mit einem Teil der Computer-Software, dem in der Literatur relativ wenig Beachtung geschenkt wird, dem Montierer. Dabei werden Hardware- und Software-Gegebenheiten folgender Computertypen berücksichtigt:

TELEFUNKEN TR44o,
IBM Systeme /36o und /37o,
CDC 66oo und Serie CYBER 17o (insbes. Modell 175),
UNIVAC 11o8 und 94oo,
HONEYWELL BULL Serie GE6oo (insbes. GE645),
RANK XEROX SIGMA-Serie (insbes. SIGMA 9)
sowie
BURROUGHS B67oo.

Die Reihenfolge ist insofern willkürlich, als sie abhängig vom Informationsniveau des Verfassers gewählt wurde.
Alle Aussagen sind natürlich nur im Rahmen dieser Beschränkung auf einige Rechnertypen zu werten.
Die Terminologie lehnt sich in der Hauptsache an die der Firma Telefunken an, da sie die beste dem Verfasser bekannte deutsche Begriffsbildung darstellt.
In einigen Fällen werden zur Erläuterung die entsprechenden Bezeichnungen der anderen Hersteller genannt.

2. VORAUSSETZUNGEN

2.1 Zweistufige Erzeugung eines Operators

Zum besseren Verständnis der Aufgaben eines Montierers (auch linker, binder, collector, consolidator etc.) soll zuerst der Weg vom Quellprogramm (z. B. ALGOL oder FORTRAN) bis zum lauffähigen Programm veranschaulicht werden (siehe Abb. 1).

Das Quellprogramm wird von einem Compiler oder Assembler auf syntaktische und semantische Richtigkeit überprüft und in eine Zwischensprache übersetzt, die bereits sehr nahe am Maschinencode liegt und für alle Compiler/Assembler einer Maschine identisch ist. Das erzeugte Objekt in der Zwischensprache nennen wir Montageobjekt (auch segment, objekt module etc.).

Das Montageobjekt kann zur späteren Verwendung auch in externen Bibliotheken gespeichert werden. Der Montierer verbindet - montiert - mehrere Montageobjekte, evtl. unter Berücksichtigung von externen Bibliotheken, zu einem einzigen Objekt, dem Operatorkörper (auch load objekt, load module etc.), der wiederum in Operatorkörper-Bibliotheken gelagert werden kann. Der Lader (loader) schließlich transportiert einen Operatorkörper, evtl. aus einer Bibliothek, in den Hauptspeicher, so daß er ein lauffähiger Operator wird.

Abhängig vom jeweiligen Rechner gibt es unter Umständen innerhalb der verschiedenen Stationen weitere Abstufungen oder auch Verbindungen:

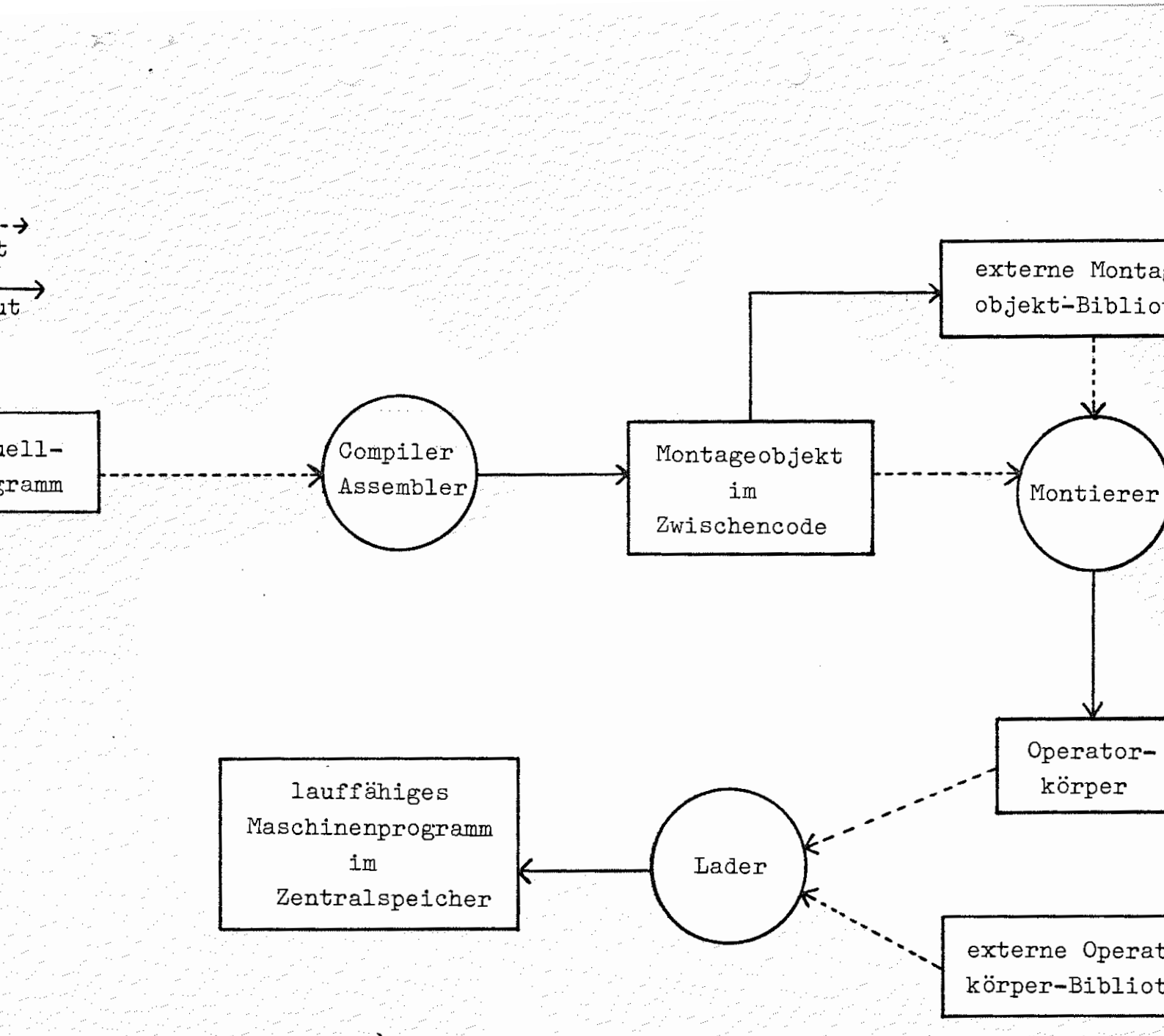

uell-
gramm
Compiler
Assembler
Montageobjekt
im
Zwischencode
externe Monta
objekt-Biblio
Montierer
Operator-
körper
externe Operat
körper-Bibliot
Lader
lauffähiges
Maschinenprogramm
im
Zentralspeicher

Abb. 1

a) Es gibt Compiler-Formen, die ganz auf einen Montierer (und Lader) verzichten und entweder das gesamte Quellprogramm interpretieren oder es direkt unter Fortlassung der Montageobjekt-Zwischensprache in absolut adressierten Maschinencode wandeln und (evtl. auf ihren eigenen Speicherplatz) laden und dann starten.
Diese Art wird uns in diesem Rahmen gar nicht interessieren.

b) Es gibt die Möglichkeit, über verschiedene Zwischensprachen erst nach mehreren Compilerläufen zum eigentlichen Montageobjekt-Zwischencode zu gelangen (dazu zählen z. B. der O-Code der Programmiersprache BCPL [27] und die Makrosprachen). Auch darauf werden wir nicht weiter eingehen.

c) Relativ häufig kommen Kombinationen von Montierer und Lader vor, unter Bezeichnungen wie linking loader etc. (IBM [13] , RANK XEROX [25] u.a.).

Einige der Vorteile des zweistufigen Vorgehens - compilieren in Montageobjekt-Zwischencode und danach montieren - sollen an dieser Stelle aufgezählt werden:

a) Durch langfristige Speicherung von solchen Zwischencodeobjekten in Bibliotheken ist es möglich, ganze Programmsysteme modular und schrittweise zu entwickeln.

b) Durch Kombinationen von Objekten aus verschiedenen Programmiersprachen lassen sich die Vorteile unterschiedlicher Sprachen miteinander verbinden, beispielsweise die effektiven Programmiermöglichkeiten eines maschinennahen Assemblers mit der Übersichtlichkeit und Anwendungsbequemlichkeit einer höheren Programmiersprache wie ALGOL6o.

c) Durch das Prozedurenkonzept lassen sich einfach Spracherweiterungen ohne Compileränderungen durchführen - man denke z. B. an Stringhandling- oder E/A-Prozedurensätze [30], [3].

d) Bereits auf Compilerebene können gewisse Sprachelemente durch sogenannte offene Unterprogramme, Intrinsics oder Assembler-Prozeduren realisiert werden, z. B. E/A-Anweisungen, mathematische Funktionen u.v.a.

2.2 Adressierung im Operator

Zur Abgrenzung der Funktionen des Laders von denen des Montierers soll eine kurze Darstellung der verschiedenen Adressierungsmöglichkeiten gegeben werden.
Wir kennen vier grundsätzlich verschiedene Arten des hardwaremäßigen Zentralspeicherzugriffs per Adresse:

2.2.1 Absolute Adressierung

Der gesamte Zentralspeicher besteht aus einer Folge von Worten oder Bytes, denen eine sequentielle Adreßzuordnung entspricht, gewissermaßen eine Durchnumerierung der Speicherzellen, beginnend bei der Adresse o bis zum maximalen Speicherausbau.
Diese Form der Adressierung die wir z. B. bei allen IBM-Rechnern antreffen, die nicht unter VS (virtueller Speicher [15]) laufen, verlangt daher vom Montierer/Lader eine Ablage des Maschienencodes in Speicherzellen mit den echten absoluten Speicheradressen und ebenso eine entsprechende Wandlung der Adreßbezüge.
Soll die Möglichkeit bestehen, einen Operator zwischenzeitlich aus dem Zentralspeicher zu verdrängen, wie dies bei Timesharing-Systemen (z. B. dem TSO) nötig ist, so muß der Montierer zusätzlich ein RLD (relocation dictionary)[17] erzeugen, anhand dessen der Lader beim erneuten

Laden, evtl. an eine andere Stelle im Zentralspeicher, gewisse Adreßänderungen vornehmen kann, die durch die Verschiebung des Operators notwendig werden.
Die Ablage des Operatorkörpers erfolgt i.a. gewissermaßen "am Stück" ab der ersten freien Zentralspeicherzelle sequentiell in seiner jeweiligen Länge.

2.2.2 Relative Adressierung

Die Adressen eines Operators beginnen grundsätzlich ab der Adresse o, werden also operatorrelativ hochgezählt. Der Operatorkörper wird wiederum in einem Block im Hauptspeicher abgelegt; zusätzlich wird ein 2-Tupel (Anfangsadresse, Länge) generiert, durch das gekennzeichnet wird, ab welcher echten absoluten Speicheradresse das Programm beginnt und wieviel Speicherzellen es lang ist. Diese Zusatzinformation wird dann bei jedem Ansprechen einer Speicheradresse hardwaremäßig interpretiert, die Anfangsadresse also von der Hardware während des Ansprechens als Basisadresse zu der jeweiligen Operatoradresse addiert.

Dieses Vorgehen finden wir z. B. bei CDC [8].
Durch die Längeninformation ist gleichzeitig ein simpler Hardwareschutz gegen Zugriff auf Speicherzellen fremder Programme gegeben, der bei CDC auch den einzigen implementierten Speicherschutz darstellt.
Verschiedene Modifikationen dieses Verfahrens sind anzutreffen, wie z. B. das von UNIVAC [34], wo zwei 2-Tupel (untere Grenze, obere Grenze) vorhanden sind, die für Befehlsbereich und Datenbereich getrennt verantwortlich sind.
Das Vorgehen aber ist identisch:
Die Beschränkungsangaben sind jeweils operatorlaufspezifisch in einem oder mehreren Hardware-Registern untergebracht und werden von dort als additive Basisadressen geholt.

Das bedeutet für den Montierer, daß er grundsätzlich alle Operatorkörper-Adressen bei o beginnen lassen kann.

2.2.3 Seitenadressierung (paging)

Ein etwas anderes Vorgehen finden wir bei Rechnern mit paging-Algorithmus wie z. B. dem TR440 [29], dem GE645 11 und dem SIGMA 9 [23] sowie den IBM-Modellen mit VS-System [15].
Die Seitenadressierung wird nur soweit erwähnt, wie es im Rahmen dieser Arbeit von Interesse ist (siehe Abb. 2), d.h. es wird nur grob der Mechanismus angedeutet.

Das Grundelement dieses Algorithmus' ist für die Software die Seite (page), der hardwaremäßig die Kachel entspricht.
Von einer Operatoradresse wird der linke Teil zur Bezeichnung einer Seitennummer und die restlichen Bits als seitenrelative Adresse benutzt. Mit Hilfe dieser Seitennummer und einer operatorspezifischen Leitblockadresse wird in einer Seiten-Kachel-Zuordnungstabelle das richtige Paar (Seitennummer, Kachelnummer) gesucht; die so ermittelte Kachelnummer ergibt zusammen mit der seitenrelativen (und damit auch kachelrelativen) Adresse im rechten Teil der Operatoradresse die echte Kernspeicheradresse.

Auf die Organisation der Seiten in Gebiete (segments) und die evtl. damit verbundene mehrstufige Suche der Kachelnummer (z. B. bei GE645 [11] sowie VS [15]) oder andere Feinheiten wie die Verwendung von assoziativen Speicherregistern zur schnelleren Auffindung der absoluten Zentralspeicheradresse soll hier nicht näher eingegangen werden.

Für den Montierer bedeutet diese Art der Seitenadressierung, daß er Adressen grundsätzlich operatorrelativ, bei o beginnend, erzeugen kann, aber genauso, daß adressenmäßig

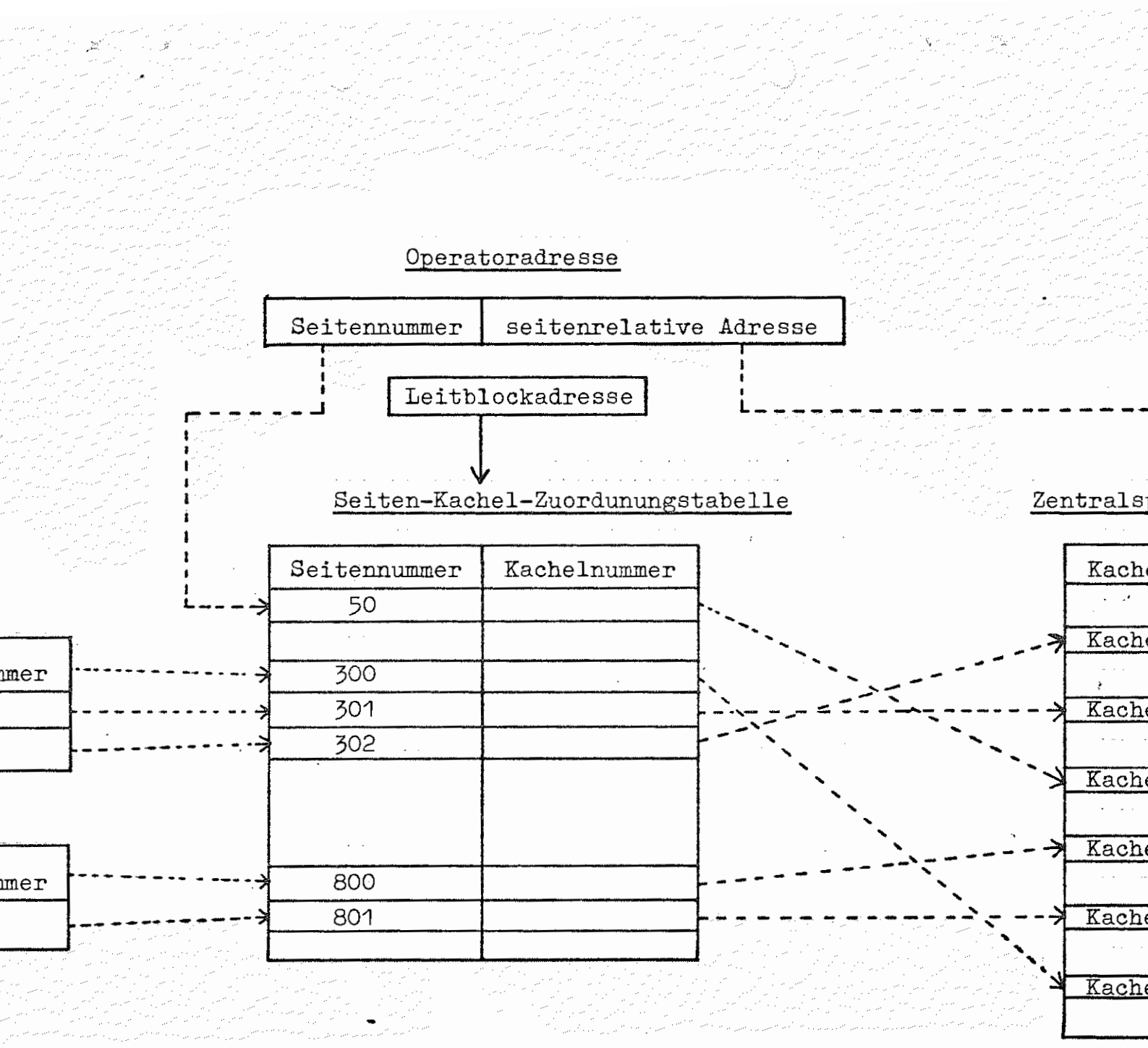

Operatoradresse
Seitennummer
seitenrelative Adresse
Leitblockadresse
Seiten-Kachel-Zuordunungstabelle
Zentrals
Seitennummer
Kachelnummer
50
300
301
302
800
801
Kache
Kache
Kache
Kache
Kache
Kache

Abb. 2

Lücken vorhanden sein dürfen, die entweder sich aus der Montiererstrategie zweckmäßig ergeben oder explizit vom Benutzer gewünscht werden können, ohne daß er einen Speicherverschnitt in Kauf nehmen muß.
Die Seitenadressierung bedeutet aber ebenfalls, daß alle Speicherschutzarten in der Regel nur in Einheiten von Seiten bzw. Kacheln hardwaremäßig realisierbar sind, u.U. sogar nur in Einheiten von Gebieten, also ganzzahligen Vielfachen von Seiten [29].
Da eine Seite bei den heutigen Rechnern im allgemeinen in der Größenordnung von 1K (= 1o24) Worten liegt, kann nur durch einen gut durchdachten Montiereralgorithmus ein Verschnitt in der Speicheraufteilung weitgehend vermieden werden.

2.2.4 Adressierung bei Stackrechnern

Einen gänzlich anderen Mechanismus finden wir in Stackrechnern wie BURROUGHS B67oo [5].
Dort wird der gesamte Zentralspeicher stackmäßig verwaltet, d.h. als Speicherbereich-Keller.
Der Zentralspeicher ist in mehrere Stacks aufgeteilt, gesteuert vom Systemkern (master control program) mittels einer Stacktabelle.
Für jedes Programm, das sich im Speicher befindet, existiert ein Stack. Wird nun eine Speicherzelle angesprochen, so wird zunächst in der Stacktabelle der dem Programm zugeordnete Stack und in diesem die Segmentbeschreibungstabelle gesucht, anhand derer wiederum die stackrelative Adresse gefunden wird.
Dabei werden noch mit Hilfe einer Reihe von Hardware-Registern, die die dynamische Blockstruktur des Programmes beschreiben, gewisse Elemente des Stacks hardwaremäßig als Programmkontrollwörter, Rücksprungkontrollwörter und Blockbasisregister interpretiert.
Das bedeutet also, daß die für den Programmablauf benötigten Register zur Befehlsadressfortschaltung etc. selbst

als Teile des jeweiligen Programmstacks hardwaremäßig gekellert werden. Die Adressen innerhalb eines Stacks werden zusätzlich zu den blockspezifischen Programmkontrollwörtern noch mit den stackspezifischen Stackbasisregistern translatiert.

Für den Montierer bedeutet dies, daß er keinen zusammenhängenden Operatorkörper erzeugen muß, sondern lediglich eine Beschreibung der zu dem Operator gehörenden Segmente sowie Listen für die symbolischen Bezüge anzulegen hat. Die Segmente werden dann vom Lader (ein Teil des MCP) beim ersten Ansprechen in den jeweiligen Stack gebracht.

Durch diese Hardwarekonzeption wird ein Teil der Probleme beim Montierer überflüssig, und ein anderer Teil wird so reduziert, daß im Grunde beim Stackrechner nur die Forderungen nach einer Vormontage (siehe 4.2) zur Vermeidung von Namenskollisionen relevant bleiben.
Aus diesem Grunde werden Stackrechner auch nur noch am Rande erwähnt werden.

2.2.5 Allgemeines zur Adressierung

Bei all diesen Adressierungsformen wurden die Darstellungen wesentlich vereinfacht, indem davon abgesehen wurde, daß selbstverständlich der Systemkern grundsätzlich die Möglichkeit haben muß, den gesamten zur Verfügung stehenden Zentralspeicher absolut zu adressieren.

Dazu kommt noch, daß bei allen Multiprogramming-Systemen die eigentlichen Akteure Prozesse sind, die die verschiedenen Benutzerprogramme verwalten und koordinieren.
Das wird dadurch erreicht, daß Programme in verschiedenen hardwaregesteuerten Modi arbeiten können, wobei bestimmte Befehle im einen Modus verboten sind oder eine andere Bedeutung haben als in anderen.

In dieser Arbeit sollen nur Programme betrachtet werden, die im "unterprivilegiertesten", dem Benutzermodus, arbeiten und außerdem nicht Bestandteil des Betriebssystems sind, da sonst noch Probleme des Code-sharings hinzukommen, die unberücksichtigt bleiben sollen.

Wenn im folgenden von absoluten Adressen die Rede sein wird, dann ist immer, falls nicht ausdrücklich anders gesagt, die Adresse gemeint, die der startbare Operatorkörper erhalten muß, um lauffähig zu sein - vollkommen unabhängig davon, wie diese Adresse aussieht, ob sie tatsächlich eine absolute Zentralspeicheradresse oder eine operatorrelative Adresse ist.

Es wird also im weiteren völlig davon abgesehen, was beim Laden und Starten des Operatorkörpers geschehen muß, und wie der hardwaremäßige Speicherzugriff organisiert ist.

3. DER KONVENTIONELLE MONTIERER

3.1 Aufgaben des Montierers

Wir wollen nun die Funktionen des Montierers etwas näher untersuchen.
Dazu ist die Kenntnis des groben Aufbaues des Montageobjekt-Zwischencodes nötig, wie er bei allen erwähnten Rechnern anzutreffen ist.

3.1.1 Adreßgenerierung für lokale Größen

Da der Übersetzer keine oder nur begrenzte Informationen darüber besitzt, welche Objekte mit dem jeweiligen Programm verbunden werden müssen, bis ein Operatorkörper entsteht, wieviel Speicherbedarf diese anderen Montageobjekte haben und welche Zusatzforderungen bezüglich der Adreßzuteilung bestehen, zerlegt der Übersetzer ein Programm in mehrere Programmabschnitte, sogenannte Zonen (auch Segmente, linkage sections etc.).

Der generierte Zwischensprachencode besteht im allgemeinen bereits aus den echten Maschinenbefehlen, jedoch werden alle Adressen nur relativ zu dem jeweiligen erzeugten Montageobjekt bzw. relativ zu seinen diversen Zonen angegeben.
Der Montierer hat die Aufgabe, die Montageobjekte und ihre Zonen zu einem Operatorkörper zusammenzufügen, und zwar so, daß keine ungenutzten Lücken in dem Operatorkörper entstehen.
Weiterhin erhält der Montierer Zusatzinformationen über die Zonen, die er zu berücksichtigen hat, die zum Teil abhängig von der jeweiligen Maschine sind. Einige Typen, die bei den meisten Großrechnern auftreten können, sollen genannt werden:

a) Speicherschutz für eine Zone. Dabei muß unterschieden werden zwischen folgenden Speicherschutzformen:

1) Schutz vor schreibendem Zugriff
2) Schutz vor lesendem Zugriff
3) Schutz vor Ausführung als Programm

Daneben können natürlich noch Kombinationen dieser einzelnen Arten auftreten.

b) Der Beginn einer Zone muß einer gewissen Adressierungsbedingung genügen, er muß z. B. auf einer Wortgrenze beginnen.

c) Bestimmte Zonen sollen genau hintereinandergelegt werden, in spezifizierter Reihenfolge und ohne Lücke.

d) Einer Zone soll nur Adreßraum, aber kein echter Speicherbereich zugewiesen werden - diese Forderung gibt es allerdings nur bei Rechnern mit paging.

e) Eine Zone soll in einem bestimmten Adreßbereich angeordnet werden, soll z. B. mit maximal 16 Bits adressierbar sein

Dergleichen Forderungen gibt es noch mehr; sie beziehen sich auf bestimmte Zonen eines einzelnen Montageobjektes.

Der Montierer hat die Aufgabe, den Zonen Adressen zuzuordnen und alle zonenrelativen Adreßbezüge entsprechend mit einer Translation zu versehen (siehe Abb. 3).
Beim Zusammenfügen der Zonen aus den verschiedenen Montageobjekten zu einem Operatorkörper muß eine Strategie gefunden werden, unter Beachtung der Zonen-Zusatzforderungen wie Speicherschutz, Adressierungsbedingung etc. eine möglichst lückenlose Abfolge der Zonen zu erzielen, um den Hauptspeicherbedarf möglichst niedrig zu halten. Lücken könnten z. B. dadurch entstehen, daß gewisse Zusatzforderungen wie der Speicherschutz nur in größeren Speichereinheiten hardwaremäßig realisierbar sind (siehe 2.2).

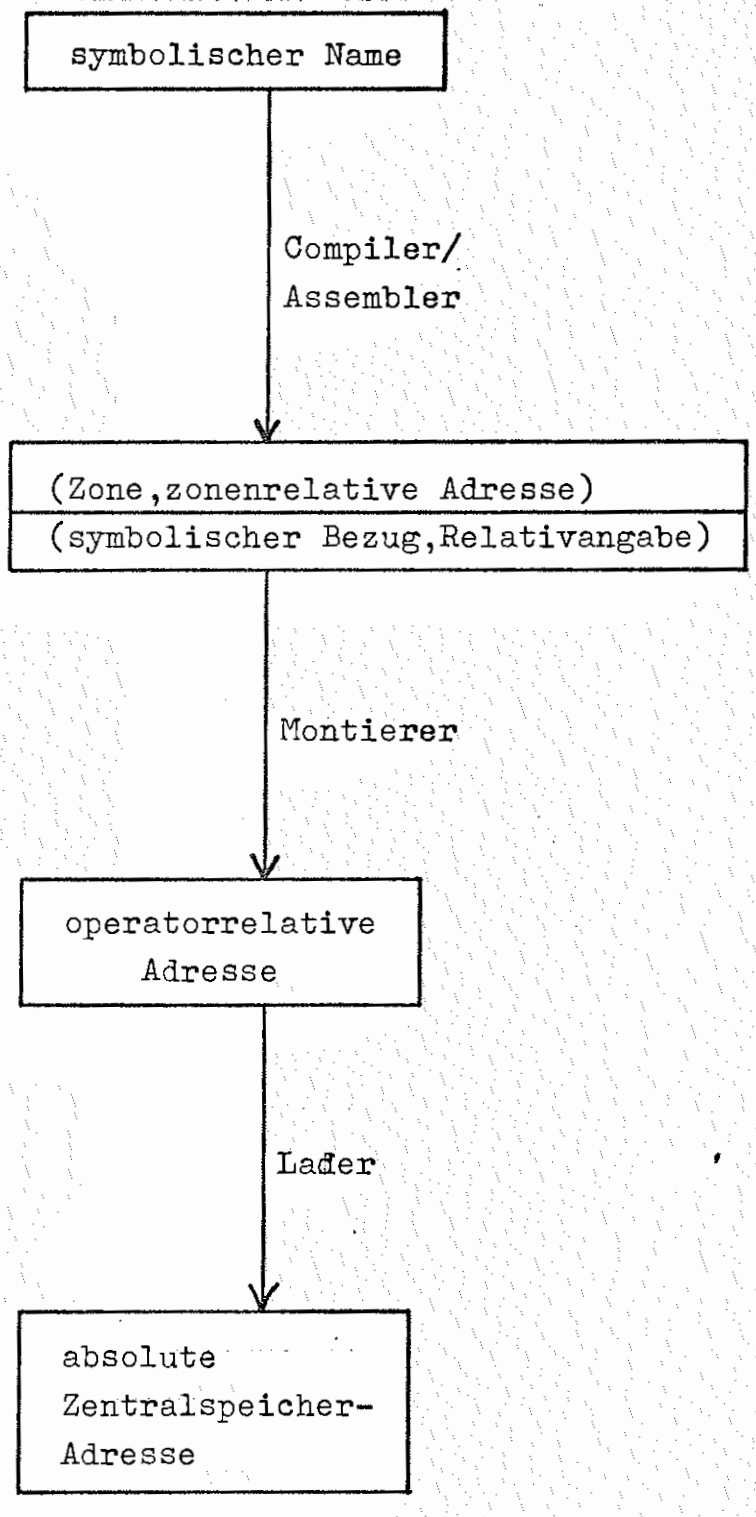

symbolischer Name
Compiler/
Assembler
(Zone,zonenrelative Adresse)
(symbolischer Bezug,Relativangabe)
Montierer
operatorrelative
Adresse
Lader
absolute
Zentralspeicher-
Adresse

Abb. 3

3.1.2 Adreßgenerierung für globale Größen

Bisher war nur die Rede von Adreßgenerierung irgendwelcher Bezüge innerhalb der Montageobjekte durch den Montierer, wobei hauptsächlich die Bezüge auf lokale Größen angesprochen wurden.
Den Unterschied zwischen lokalen und globalen Größen kann man sich am einfachsten veranschaulichen, wenn man die Analogie zu einer strukturierten Sprache wie ALGOL6o betrachtet und ein Montageobjekt mit einem Block und den Operatorkörper mit dem ganzen Programm vergleicht:
Die innerhalb eines Blockes definierten Größen sind für diesen Block die lokalen und die Bezüge auf Variable in umgebenden Blöcken bzw. dem äußersten Block die globalen.

Eine Hauptaufgabe des Montierers außer der Generierung von absoluten Adressen für lokale Bezüge, und mit dieser Aufgabe beschäftigt sich die vorliegende Arbeit im wesentlichen, ist es, die symbolischen Bezüge zwischen verschiedenen Montageobjekten aufzulösen und umzuformen.

Folgende Arten von Querbezügen zwischen Montageobjekten sind bei allen Großrechnern anzutreffen:

a) Der Name eines Montageobjektes ist in einem anderen bekannt und wird dort als Sprungadresse zum Unterprogrammaufruf benutzt.

b) Zusätzlich als Eingänge (Entry) deklarierte Speicherzellen des Montageobjektes können ebenfalls als Unterprogrammnamen verwendet werden. Dabei muß ein solcher Eingang genau so behandelt werden wie einer, der gleichzeitig der Name des Montageobjektes ist.

c) Zonen eines Montageobjektes können mit Namen versehen werden. Eine bestimmte Zonenkennzeichnung teilt dem

Montierer mit, daß alle Zonen aus allen Montageobjekten, die den gleichen Namen haben, im Operatorkörper übereinandergelegt werden sollen, so daß dadurch ein mehreren Montageobjekten gemeinsamer Bereich entsteht zur Datenübergabe.
Ein solcher, Commonzone genannter Bereich wird dann so lang wie die längste der übereinandergelegten Zonen.
Bekannt ist dieser Typ vor allem von dem gleichnamigen FORTRAN-Sprachelement her.

Während diese drei Arten von symbolischen Bezügen auch in höheren Programmiersprachen wie FORTRAN, ALGOL6o etc. benutzbar sind, gibt es darüber hinaus auch Formen von Querbezügen, die nur auf Assemblerebene bzw. bei einigen Rechnern gar nicht möglich sind:

d) Datenbereiche eines Montageobjektes können einem anderen durch einen symbolischen Bezug in der Form eines Einganges bekanntgemacht und auf diese Art zur Bearbeitung zur Verfügung gestellt werden.

e) Es gibt Bezüge, sogenannte Kontakte, die wie Eingänge gehandhabt werden, aber an einen bestimmten Montageobjektnamen gebunden und somit z. B. höheren Programmiersprachen i.a. nicht zugänglich sind.
Dies ist die einfachste Form einer Struktur von globalen Namen, die schon auf Assemblerebene festgelegt wird (z. B. TR44o [29]).

3.2 Probleme bei der Montage

Das augenfälligste Problem beim Montieren ist das der großen anfallenden Rechenzeit (CPU-Time), denn je größer die zur Montage heranzuziehenden Bibliotheken sind bzw. aus je mehr Montageobjekten ein Operatorkörper besteht, um so rechenzeitintensiver wird das Montieren des Operatorkörpers, da sehr große Mengen von Zwischencode jedesmal neu in Maschinencode übersetzt werden müssen.

Das wirkt sich insbesondere in der Testphase eines Programmes stark aus, wenn z. B. ein neuer Modul hinzugefügt werden soll, denn die Compile-Zeit dieses Moduls ist unter Umständen gering, aber nach jeder Änderung eines Montageobjektes muß der gesamte Operatorkörper neu generiert werden.

Dabei wird eine Menge Rechenzeit für immer die gleichen Vorgänge verbraucht, denn die Montageobjekte, an denen nichts geändert wurde, müssen trotzdem jedesmal neu in Maschinencode übertragen werden. Am unangenehmsten ist dieser Effekt in einem Timesharing-Dialogsystem, denn ein Teil der Vorteile des Austestens von Programmen im ständigen Dialog mit dem Rechner geht durch die anfallende große Rechen- und Wartezeit wieder verloren. Daran vermag auch eine noch so optimierte Zwischensprache nichts zu ändern, sondern nur eine geänderte Strategie des Montierers.

Eine weitere Schwierigkeit ergibt sich, wenn mehrere Prozedurensätze aus getrennten Bibliotheken gekoppelt werden sollen:
Innerhalb eines Teams, das einen solchen Prozedurensatz entwickelt, ist es möglich, durch Koordination gewisse Namens- und Aufbau-Konventionen für das System einzuhalten, aber bei der Verknüpfung unabhängig voneinander entwickelter Systeme kommt es häufig zu Namenskollisionen in den internen Querverweisen der Prozedur-Komplexe.

Dadurch wird ebenso die Modularität solcher Montageobjekt-Bibliotheken eingeschränkt, denn der Benutzer der Prozedurensätze muß sich u.U. an Namenskonventionen halten, die ihn im Grunde gar nicht interessieren, da etwa intern in dem Prozedurensatz gewisse Standard-Namen benutzt werden, die dem Benutzer nicht zugänglich sein sollen, die aber durch die internen Querbezüge zu reservierten, folglich dem Benutzer verbotenen Namen werden.

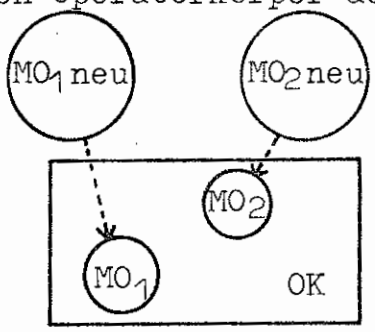
Eine Abhilfe wäre hier möglich, wenn man dem Montierer als Zusatzinformation zu den Montageobjekten eine hierarchische Struktur für die Gültigkeit von globalen und lokalen Namen liefern könnte, wobei sich etwa eine Blockstruktur ähnlich der in ALGOL6o oder anderen Programmiersprachen anböte.

3.3 Forderungen an einen optimalen Montierer

Aus diesen Überlegungen ergeben sich einige grundlegende Forderungen an einen Montierer:

3.3.1 Ersetzung von Montageobjekten

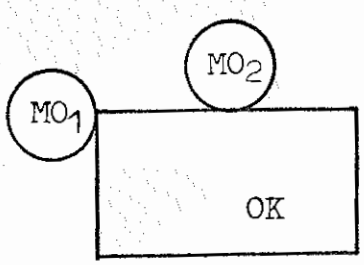
Es sollte die Möglichkeit geben, in einem fertig montierten Operatorkörper auf Wunsch bestimmte Montageobjekte



ersetzen zu lassen. Das brächte den Vorteil, beim Austesten von Programmen immer nur diejenigen Teile (Prozeduren, Funktionsprozeduren etc.) neu in Zwischensprache compilieren und anschließend in Maschinencode übersetzen zu müssen, die tatsächlich geändert wurden. Dadurch könnte man den größten Teil der Rechen- und Wartezeit, die beim Montieren benötigt wird, einsparen.

3.3.2 Anfügen von Montageobjekten

Wünschenswert wäre es, wenn man an einen Operatorkörper nachträglich Montageobjekte anfügen könnte, deren Namen zur Montagezeit noch nicht feststehen. Beispiel dafür wären

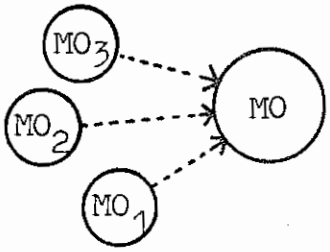


etwa ein Dialoginterpreter wie AIDA [20], an den der Benutzer nachträglich eigene Montageobjekte, z. B. ALGOL-Prozeduren, anfügen möchte oder andere Dialogoperatoren, die mit dynamisch festgelegten, z. B. am Benutzerterminal erfragten

Montageobjektnamen irgendwelche mathematischen Funktionen einfügen könnten, um sie zu plotten etc.
Das würde ein nachträgliches Anmontieren zur Laufzeit des Operators bedingen.

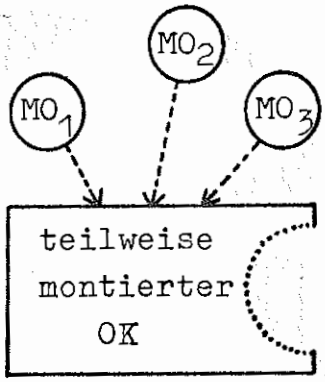
3.3.3 Erzeugung eines Montageobjektes aus mehreren

Es müßte möglich sein, gezielt eine bestimmte Menge von Montageobjekten zusammenzufügen zu einem einzigen, innerhalb dieser Menge alle internen Querbezüge aufzulösen und somit als offene Bezüge zu eliminieren, und nur bestimmte Eigen- oder Fremdbezüge als solche beizubehalten. Dieses neu erzeugte größere Objekt könnte dann ebenfalls wieder in der Montageobjekt-Zwischensprache abgelegt werden.



3.3.4 Teilweise Vormontage

Eine gewisse Menge von Montageobjekten sollte vorab in Maschinencode übersetzbar sein, unter Beibehaltung eines Teils der offenen Querbezüge. Aus dieser Menge entstünde dann noch kein lauffähiger Operatorkörper, aber der Benutzer einer Montageobjekt-Bibliothek, die auf diese Weise schon zum Teil in Maschinencode übersetzt wäre, brauchte nur noch seine eigenen Montageobjekte anmontieren zu lassen, um evtl. mit geringem Zeitaufwand sein Programm als Operatorkörper lauffähig zu erhalten.



Die Forderungen 3.3.1 und 3.3.2 laufen auf einen Nachmontierer, 3.3.3 und 3.3.4 auf einen Vormontierer hinaus.

4. ALLGEMEINER LÖSUNGSANSATZ FÜR DEN OPTIMALEN MONTIERER

4.1 Der Nachmontierer

4.1.1 Vorbemerkungen

Der konventionelle Montierer wandelt bereits zur Montagezeit jedes 2-Tupel (Zone, Relativadresse) bzw. (symbolischer Bezug, Relativadresse) in eine absolute Adresse (siehe 2.2.5) um und fixiert somit jede relative Angabe vor dem Start des Programms.
Das hat den Vorteil, daß zur Laufzeit des Programms nur noch hardwaremäßig mit absoluten Adressen gerechnet werden muß und das Programm ohne erneute Montage mehrmals gestartet werden kann.

Außerdem wird für den eigentlichen Programmlauf keine weitere Information mehr benötigt, aus welchen Montageobjekten der Operatorkörper besteht etc.
Höchstens im Falle eines auftretenden Fehlers kann mit Hilfe solcher Informationen eine Fehlersuche erleichtert werden, indem eine Rückverfolgung der Aufrufverschachtelung ausgedruckt oder auch ein quellsprachenbezogener Dump geliefert wird. Für diesen Fall reicht es jedoch, die dafür nötigen Informationen auf einem Hintergrundspeicher zu lagern und nur bei einem Fehler von dort abzurufen.

Die Änderung eines Montageobjektes verlangt aber eine Änderung der relevanten Umgebung, in der die Querbezüge bei der Objektcodegenerierung fixiert wurden. Deshalb hat jede Änderung eines Montageobjektes die Konsequenz, alle symbolischen Bezüge auf dieses Montageobjekt zu regenerieren, d.h. auch eine Anzahl anderer Montageobjekte neu in Objektcode zu übertragen.

Da es beim konventionellen Montierer jedoch keine Information über die relevante Umgebung gibt und alle Bezüge

vor Operatorlaufbeginn schon fixiert sind, heißt das, daß der gesamte Operatorkörper aus den benötigten Montageobjekten neu erzeugt werden muß.
Der konventionelle Montierer ist also sehr wenig flexibel gegenüber Modifizierung einzelner Montageobjekte und deshalb unbrauchbar zur Erfüllung der Forderung 3.3.1 .

4.1.2 Der Zwischencode-Interpreter

Als Alternative bietet sich der reine Interpreter des Montageobjekt-Zwischencodes an.
Beim Interpretieren könnte man etwa so vorgehen:

Jeder absolute Wert sowie jeder Befehl mit einem absoluten Adreßteil, z. B. Shift-Befehle, Befehle, die nur Register oder Indexregister ansprechen sowie gewisse Systemsprünge zur Erbringung von Leistungen durch den Systemkern (master scheduler, master control program etc.), die es in allen Rechnern gibt, werden jeweils bei der ersten Interpretation in die endgültige Form gebracht, wie dies auch der konventionelle Montierer täte (falls sie nicht schon vom Compiler/Assembler in dieser Form im Montageobjekt-Zwischencode abgelegt würden) und dann ausgeführt ; in allen betrachteten Rechnern gibt es einen Spezialbefehl, der es erlaubt, einen dynamisch in einem Register oder einer Speicherzelle erzeugten Befehl auszuführen, d.h. quasi "dazwischenzuschieben" ohne die Programmlaufkontrolle durch Unterprogrammaufruf abzugeben.

Alle Befehle, die den Zentralspeicher ansprechen, seien es Sprungbefehle oder Befehle, die Daten manipulieren, werden rein interpretiert, indem jedesmal das 2-Tupel (Zone, Relativangabe) bzw. (symbolischer Bezug, Relativangabe)gewandelt wird in einen Speicherbezug auf einen Datenbereich des Interpreters.

Dadurch entstünde keinerlei Problem beim Ersetzen oder Anfügen eines Montageobjektes, da kein symbolischer Bezug

fixiert wird, sondern alle Relativbezüge interpretativ verarbeitet werden.

Dieses Vorgehen bietet ein Maximum an Flexibilität, und es wäre sicherlich möglich, damit alle gestellten Forderungen zu erfüllen, aber in der Praxis wäre es zu rechenzeitaufwendig, da ja alle Bezüge, auch die der nicht modifizierten Montageobjekte, jedesmal wieder neu interpretiert werden müssen, und zwar nicht nur bei jedem neuen Operatorlauf, sondern fatalerweise auch bei jedem Durchlauf etwa einer Schleife, so daß die Methode des reinen Interpretierens des Montageobjekt-Zwischencodes sich als unbrauchbar erweist - selbst bei einer denkbaren Hardwareunterstützung, z. B. in einem Stackrechner.

Lediglich zu einem Zeitpunkt, zu dem die Großzahl der Montageobjekte sich noch in der Testphase befindet, könnte ein interpretatives Abarbeiten wegen seiner Flexibilität Vorteile haben.

Auch wäre es in diesem Stadium wahrscheinlich möglich, während der Interpretation gute Testhilfen in Form von dynamischen Kontrollen und Trace-Ausdrucken zu bieten.

Sobald aber größere Bibliotheken mit zum Erzeugen des Operatorkörpers herangezogen werden müssen, wird die für das Interpretieren benötigte Rechenzeit zu groß.

4.1.3 Die optimale Lösung

Ein Kompromiß zwischen diesen beiden Arten, dem konventionellen Montierer und dem Montageobjekt-Zwischencode-Interpreter scheint deshalb sinnvoll zu sein.

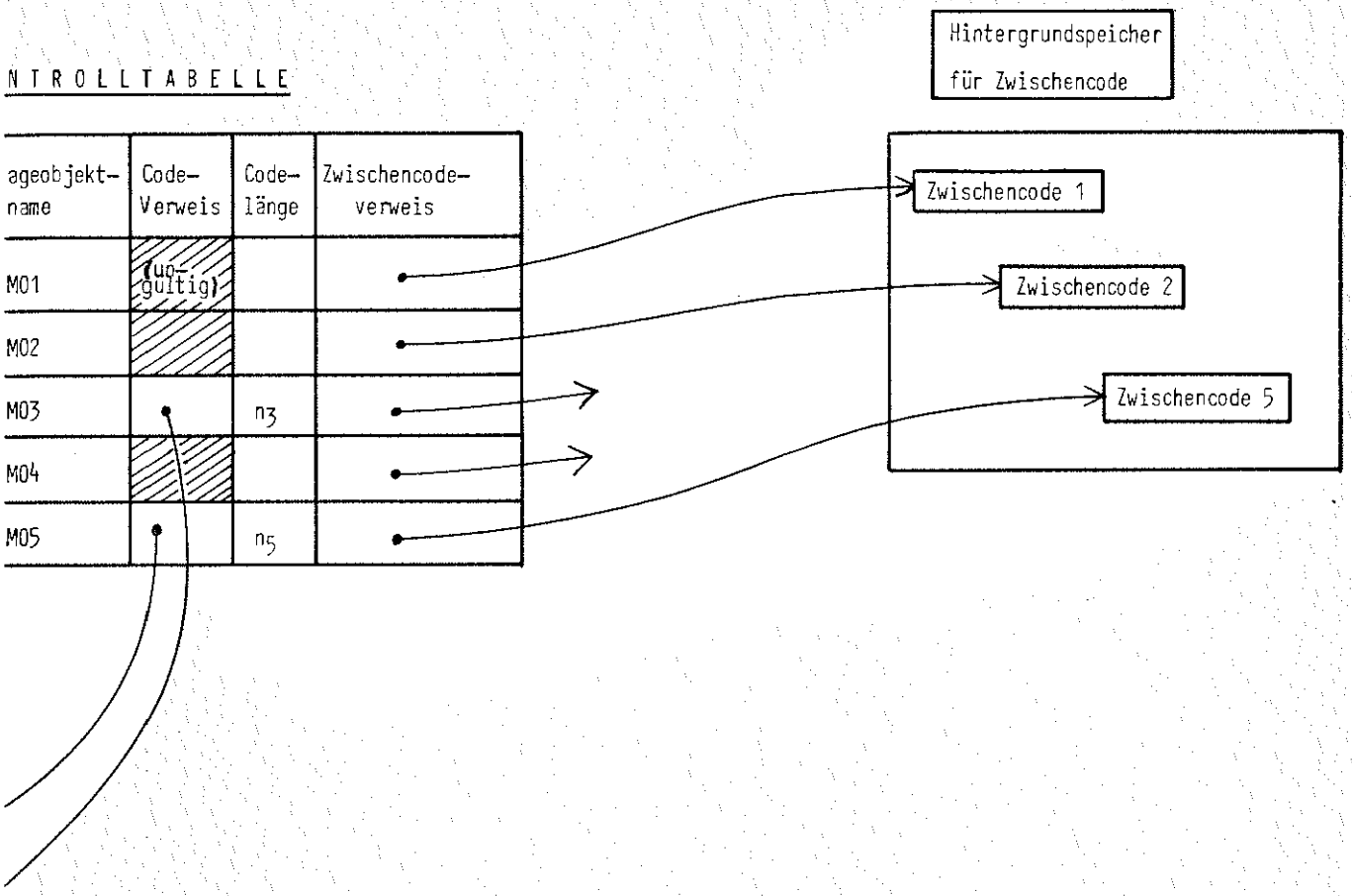
Bei der ersten Montage, also vor Beginn des ersten Programmlaufes, wird vom Montierer nur ein grobes Gerüst des Operatorkörpers erzeugt, das in der Hauptsache aus einer Kontroll- und Steuer-Prozedur sowie aus der Rahmen-Prozedur, auch Hauptprogramm genannt, besteht. Dazu kommen verschiedene Listen und Tabellen, die die statische und dynamische Struktur des - zu Beginn noch nicht fest montierten - Operatorkörpers beschreiben.

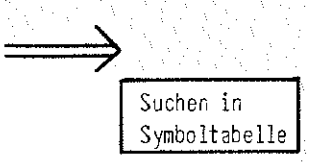
4.1.3.1 Konstruktion des Nachmontierers

Alle Montageobjekte außer dieser Kontrollprozedur und dem Hauptprogramm bleiben in der Montageobjekt-Zwischensprache erhalten und liegen nur auf einem Hintergrundspeicher mit direktem Zugriff.

An Listen werden angelegt (siehe Abb. 4):

a) Eine Tabelle aller vorkommenden symbolischen Bezüge mit den nötigen Einschränkungen hinsichtlich ihrer statischen Gültigkeitskriterien und ihren eventuellen Bindungen an bestimmte andere symbolische Bezüge im Falle von Kontakten.
   Diese Tabelle nennen wir die SYMBOLTABELLE. Sie enthält für jeden symbolischen Bezug einen Verweis auf das Montageobjekt, in dem er eine lokale Größe ist, bzw. im Falle einer Commonzone einen Verweis auf eine COMMON-TABELLE, eine Zusatztabelle aller Montageobjekte, in denen sie benutzt wird sowie eine Angabe über ihre maximal vorkommende Länge.
   Ein Verweis auf ein Montageobjekt ist in der Symboltabelle ein Zeiger auf ein Element der zweiten, der Kontrolltabelle.


Hintergrundspeicher
für Zwischencode
N T R O L L T A B E L L E
ageobjekt-
name
Code-
Verweis
Code-
länge
Zwischencode-
verweis
M01
(un-
gültig)
M02
M03
n3
M04
M05
n5
Zwischencode 1
Zwischencode 2
Zwischencode 5


Suchen in
Symboltabelle

dur

Abb. 4

b) Die KONTROLLTABELLE besteht aus einer Liste aller für den Operatorkörper benötigten Montageobjekte. Sie enthält für jedes Montageobjekt einen Zeiger auf den zugehörigen Montageobjekt-Zwischencode sowie einen Zeiger auf den montierten Maschinencode im Operatorkörper, der zu Beginn des Programmlaufes auf ungültig gesetzt ist.

Dazu kommen verschiedene dynamisch erzeugte Tabellen, die noch beschrieben werden.

Wir wollen nun die Vorgänge während des Programmlaufes betrachten (siehe Abb. 5).

4.1.3.2 Der dynamische Ablauf

Ein bereits montiertes Montageobjekt werde ausgeführt (zu Beginn das Hauptprogramm).
Dieses Montageobjekt ist aber nur montiert mit seinen lokalen Größen, also seinen eigenen Zonen; jeder symbolische Bezug ist ersetzt durch einen Sprung in die Kontrollprozedur.

Wird durch Ansprechen eines symbolischen Bezuges in die Kontrollprozedur gesprungen (mit Spezifizierung des Namens des symbolischen Bezuges), so wird dadurch der normale Ablauf zunächst unterbrochen. Wie diese Unterbrechung am geeignetsten durchgeführt wird, ist von den Hardwaremöglichkeiten abhängig ; dieser Punkt wird später noch behandelt werden (siehe 5.1).

In der Kontrollprozedur wird durch Suchen in der Symboltabelle das zugehörige Element der Kontrolltabelle festgestellt, und es wird geprüft, ob zu dem Kontrolltabellenelement bereits generierter Maschinencode existiert.
Wenn nicht, wird zunächst ein Montiervorgang angestoßen, der das zugehörige Montageobjekt in Maschinencode übersetzt und den erzeugten Code in den Operatorkörper einfügt, indem wiederum alle symbolischen Bezüge durch Sprünge in die Kontrollprozedur ersetzt werden.

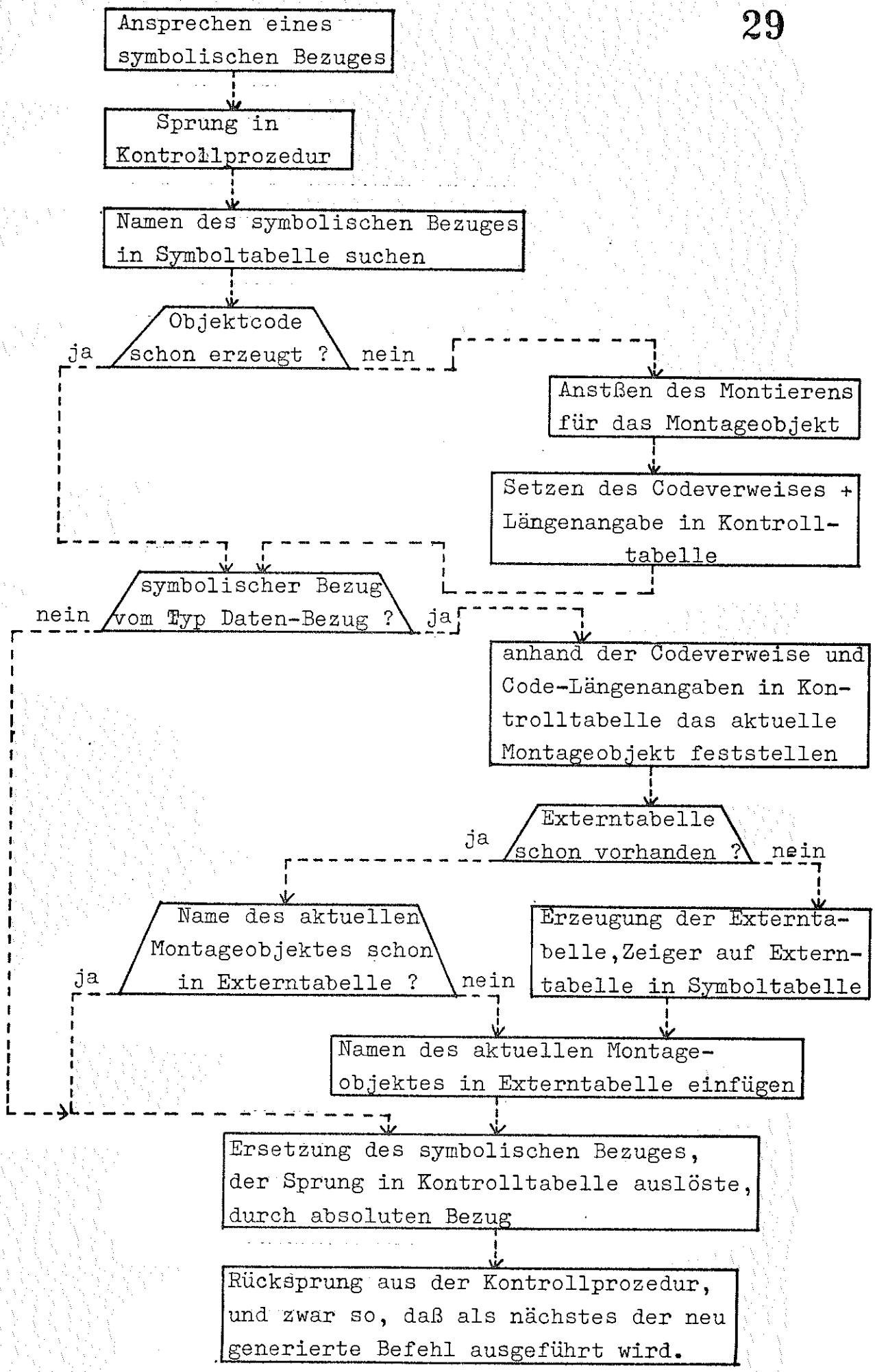

Ansprechen eines symbolischen Bezuges
Sprung in Kontrollprozedur
Namen des symbolischen Bezuges in Symboltabelle suchen
Objektcode schon erzeugt ?
ja
nein
Anstßen des Montierens für das Montageobjekt
Setzen des Codeverweises + Längenangabe in Kontroll-tabelle
symbolischer Bezug vom Typ Daten-Bezug ?
nein
ja
anhand der Codeverweise und Code-Längenangaben in Kontrolltabelle das aktuelle Montageobjekt feststellen
Externtabelle schon vorhanden ?
ja
nein
Name des aktuellen Montageobjektes schon in Externtabelle ?
ja
nein
Erzeugung der Externtabelle, Zeiger auf Externtabelle in Symboltabelle
Namen des aktuellen Montage-objektes in Externtabelle einfügen
Ersetzung des symbolischen Bezuges, der Sprung in Kontrolltabelle auslöste, durch absoluten Bezug
Rücksprung aus der Kontrollprozedur, und zwar so, daß als nächstes der neu generierte Befehl ausgeführt wird.

Abb. 5

Der Zeiger auf den Maschinencode in der Kontrolltabelle wird auf gültig gesetzt und bekommt den entsprechenden Verweis auf die Stelle im Operatorkörper. War schon gültiger Maschinencode vorhanden, so entfällt dieser Vorgang.

Dann wird der Sprung in die Kontrollprozedur, der die Überprüfung auf gültigen Code bzw. evtl. den Montagevorgang ausgelöst hat, durch den nunmehr gültigen absoluten Bezug ersetzt.
Ist dieser Bezug ein Sprung in das durch den symbolischen Bezug gekennzeichnete Montageobjekt, so wird die Kontrolle an dieses Montageobjekt übergeben, das ja jetzt in gültiger Maschienencode-Form vorliegt, anderenfalls wird die Kontrolle an das Montageobjekt zurückgegeben, aus dem heraus der Sprung in die Kontrollprozedur erfolgte.

Auf diese Weise wird der Maschinencode erst sukzessive während des Programmlaufes erzeugt, und zwar nur für diejenigen Montageobjekte, die tatsächlich benötigt werden.

Tritt irgendwann ein Fehler auf, so ist die Rechenzeit gespart, die beim konventionellen Montierer dafür gebraucht worden wäre, die bis zu diesem Zeitpunkt noch nicht benötigten Montageobjekte überflüssigerweise in Maschinencode zu übersetzen.

Was muß nun getan werden, wenn ein Montageobjekt bei einer erneuten Initialisierungsmontage ersetzt werden muß, da es geändert, d.h. neu in Montageobjekt-Zwischencode compiliert wurde?
Wenn es noch nicht benutzt, also auch noch nicht in Maschinencode übersetzt wurde, so ist höchstens der Zeiger auf den Zwischencode in der Kontrolltabelle zu korrigieren.

Außerdem müssen u.U. einige Elemente in der Symboltabelle hinzugefügt, gelöscht oder geändert werden, je nachdem, was in dem Montageobjekt geändert wurde.

Ansonsten braucht in den anderen, vielleicht schon montierten Montageobjekten nicht geändert zu werden, da sie von dieser Modifizierung nicht betroffen werden.

Wenn bereits Maschinencode für das modifizierte Montageobjekt generiert ist, so muß zunächst der entsprechende Zeiger in der Kontrolltabelle auf ungültig gesetzt werden, da bei einem neuerlichen Ansprechen eines symbolischen Bezuges auf das Montageobjekt im nächsten Programmlauf erst neu Code generiert werden muß.

Weiter muß - und das ist die Hauptarbeit und Schwierigkeit beim Ersetzen eines modifizierten Montageobjektes, das bereits montiert war - untersucht werden, welche Umgebung durch die Modifizierung betroffen wird, d.h. welche Folgen die Modifizierung für andere Montageobjekte hat, welche Montageobjekte evtl. ebenfalls neu in Maschinencode montiert werden müssen etc.

Im einfachsten und zum Glück häufigsten Fall tritt lediglich die Anfangsadresse des modifizierten Montageobjektes als symbolischer Bezug in anderen Montageobjekten auf, wenn dieses Montageobjekt nämlich dort als Unterprogramm aufgerufen wird.
Für diesen Fall genügt es, den ersten Befehl des vorher gültigen, inzwischen in der Kontrolltabelle als ungültig gekennzeichneten Maschinencodes durch einen speziellen Sprung in die Kontrollprozedur zu ersetzen.

Die Kontrollprozedur kann dann bei einem solchen Sprung in einen ungültigen Code den Befehl, der zu diesem Sprung führte, durch einen geänderten Sprung in den (evtl. vorher erzeugten) neuen aktuell gültigen Maschinencode ersetzen.

Der Rest des als ungültig gekennzeichneten Codes wird als "frei" markiert, um bei der Wandlung von Montageobjekten in Maschinencode zur Aufnahme neuen Codes zur Verfügung zu stehen.

Um dieses zu erleichtern, wird jeder Zeiger auf Maschinencode in der Kontrolltabelle um eine Zusatzinformation über die Länge des generierten Codes ergänzt. Die "frei"-Markierung von Operatorkörper-Teilen kann entweder durch ein bestimmtes Bit-Muster in diesen Teilen selbst oder besser noch durch eine zusätzliche Tabelle geschehen.

Schwieriger wird die Erkennung der relevanten, mitzuändernden Umgebung, wenn entweder symbolische Bezüge auf Befehle irgendwo in dem Montageobjekt - also nicht auf den ersten Befehl - oder auf Variablenbereiche des Montageobjektes vorhanden sind oder zu Montagezeit (gemeint ist der Zeitpunkt der Codegenerierung beim ersten Ansprechen des Montageobjektes) noch nicht eindeutig klärbar ist, ob der symbolische Bezug ein Bezug auf Befehls- oder Variablenbereiche ist.

In diesen Fällen muß das jeweilige Element der Symboltabelle einen Verweis auf eine zusätzliche spezifische EXTERNTABELLE enthalten, in der alle Montageobjekt-Namen aufgelistet werden, die auf den symbolischen Bezug zugegriffen haben; diese Eintragungen werden ebenfalls dynamisch vorgenommen, und zwar jedesmal, wenn ein symbolischer Bezug von der Kontrollprozedur durch den absoluten Bezug ersetzt wird.

Auf diese Art und Weise ist es bei der Behandlung von modifizierten Montageobjekten möglich, genau die Umgebung, d.h. die Menge von Montageobjekten zu charakterisieren, in der der Code gültig war, und in der deswegen nach der Modifizierung die symbolischen Bezüge auf das geänderte Montageobjekt regeneriert werden müssen.

Dabei ist es jedoch effizienter, nicht wiederum den gesamten Code dieser "Umgebungs-Montageobjekte" auf ungültig zu setzen, sondern nur den Montageobjekt-Zwischencode mit dem bisherigen Maschinencode zu vergleichen, und überall dort, wo im Zwischencode ein symbolischer Bezug auf das geänderte Montageobjekt auftaucht,

statt des entsprechenden Befehles im Maschinencode wieder einen Sprung in die Kontrollprozedur einzusetzen, also nur die Auflösung der symbolischen Bezüge in absolute vom letzten Programmlauf wieder rückgängig zu machen.

Dieses Vorgehen wird zwar nicht weniger Rechenzeit in Anspruch nehmen als eine völlige Neumontage der von der Modifizierung betroffenen Montageobjekte, aber erstens muß der freie Operatorkörper-Platz nicht neu kalkuliert werden, denn die Länge und Anordnung wird durch eine Rückwandlung der absoluten in symbolische Bezüge nicht verändert, und zweitens würden ja durch ein Ungültigsetzen des Codes wieder weitere Montageobjekte betroffen, die dann ebenfalls neu montiert werden müßten - das würde sich u.U. zu einer völligen Neumontage des gesamten Operatorkörpers ausweiten.

Eine Sonderbehandlung unter den symbolischen Bezügen müssen die Commonzonen erfahren:

Da sie mehreren Montageobjekten als gemeinsame Speicher dienen, ist es notwendig, für jede eine gesonderte statische COMMONTABELLE anzulegen, in der verzeichnet ist, zu welchen Montageobjekten sie gehört, statt eines Verweises auf ein einzelnes Montageobjekt. Das ist deswegen nötig, weil die Angabe über die Länge einer Commonzone aus der maximal vorkommenden Länge in allen diesen Montageobjekten gebildet werden muß.

Ebenso besteht bei einigen Rechnern (z. B. TR440 [29]) die Möglichkeit, daß die Vorbesetzung von Commonzonen aus allen sie ansprechenden Montageobjekten zusammengesucht werden muß. Das ist allerdings nur auf Assemblerebene möglich, denn in höheren Programmiersprachen wie FORTRAN ist dazu i.a. ein spezielles Montageobjekt, ein Blockdata-Unterprogramm nötig (außer beim TR440 [30]).

Eine zusätzliche Schwierigkeit ergibt sich, wenn ein

Element einer Commonzone in einem Montageobjekt außerdem als Externdeklaration auftritt, was außer bei IBM [14] bei allen Rechnern möglich ist.

Dann reicht die Commontabelle nicht aus zur Beschreibung der Bezüge auf die Commonzone, und es muß zusätzlich wie bei anderen Datenbezügen auch eine Externtabelle angelegt werden.

Aus diesen Gründen empfiehlt es sich, einen symbolischen Bezug auf eine Commonzone grundsätzlich als Datenbezug zu behandeln, auch wenn es z. B. beim TR440 die Möglichkeit gibt, Befehle in einer Commonzone abzulegen und sie nur per Sprungbefehl anzusprechen.

Wenn in einem modifizierten Montageobjekt eine Commonzone vorkommt, so müssen außer in den Montageobjekten, die in der zugehörigen Externtabelle aufgeführt sind, die die Commonzone also schon dynamisch angesprochen haben, auch in allen, die überhaupt einen Bezug auf sie enthalten und schon montiert wurden, die symbolischen Bezüge regeneriert werden, wofür dann die Commontabelle herangezogen wird (in einer Externtabelle sind ja sowieso nur Montageobjekte aufgeführt, die bereits montiert wurden).

4.1.3.3 Zusammenfassung

Die Aktionen des Nachmontierers gliedern sich demnach in folgende Teile:

a) Der Initialisierungslauf

Er wird immer dann angestoßen, wenn ein Programm das erstemal montiert werden soll, d.h. wenn noch kein Operatorkörper vorhanden ist, oder wenn durch vielfache Ersetzungen von Montageobjekten der Operatorkörper oder auch ein Teil seiner Listen so zerstückelt ist, daß es wegen erheblichen Speicherverschnittes rentabler

ist, den Operatorkörper ganz neu zu initialisieren. Auch auf Wunsch des Benutzers kann der Initialisierungslauf angestoßen werden.

Dabei wird das Gerüst des Operatorkörpers angelegt mit der Kontrollprozedur, dem Hauptprogramm, der Kontrolltabelle und der Symboltabelle mit den dazugehörigen Commontabellen.

Alle diese Tabellen müssen ebenfalls im Operatorkörper selbst liegen, da es wegen der häufigen Zugriffe auf sie unerläßlich ist, sie in den Zentralspeicher zu legen, um nicht jedesmal einen zeitaufwendigen Hintergrundtransport durchführen zu müssen.

Da im Initialisierungslauf ohnehin zur Erzeugung von Symbol- und Kontrolltabelle alle globalen Deklarationen und Bezüge aller beteiligten Montageobjekte angesehen werden müssen, empfiehlt es sich, zu diesem Zeitpunkt bereits eine weitere Gruppe von symbolischen Bezügen abzuhandeln, von denen bisher noch nicht die Rede war, da sie maschinenabhängig, und zwar nur bei IBM [12] und HONEYWELL BULL [11] möglich sind.

Damit sind symbolische Bezüge gemeint, die schon in dem Montageobjekt, in dem sie deklariert sind, einen absoluten Wert und keine relative Zonenadresse darstellen. Sie können in Form von absoluten Zahlen, Summen- und Differenzbezügen oder bei IBM sogar Produktbezügen auftreten oder aus Kombinationen dieser Möglichkeiten bestehen.

Sie werden unter anderem dazu benutzt, Freihalteangaben für Variablenbereiche zu spezifizieren, d.h. um das Analogon in den höheren Programmiersprachen zur Verdeutlichung zu nennen, zur Festlegung der Größe von Feldern, aber auch zur Spezifizierung von bestimmten absoluten Adreßteilen, die bei einigen

Befehlen vorkommen können (siehe auch 4.1.2).
Im TR44o würde das z. B. bedeuten, daß ein Eingangsname, dem mittels eines GLCH-Befehls ein absoluter Adreßteil zugeordnet wäre, in einem anderen Montageobjekt als Adreßteil eines ASP-Befehls benutzt würde.

Da der Wert dieser globalen Größen unabhängig von der Montage der Montageobjekte ist, in denen sie deklariert sind, ist es nützlich, weil zeitsparend, für sie eine Zusatztabelle anzulegen, in der schon während des Initialisierungslaufes die berechneten, absoluten Werte abgelegt werden.

Im übrigen müssen Bezüge auf sie jedoch wie symbolische Datenbezüge behandelt werden, mit Anlegen einer Externtabelle etc.

b) Der Ersetzungslauf

In diesem Lauf werden Montageobjekt-Zwischencodeverweise in der Kontrolltabelle ersetzt, die Symboltabelle korrigiert, Maschinencodefolgen im Operatorkörper sowie Verweise auf sie in der Kontrolltabelle auf ungültig gesetzt und nach dem Feststellen der relevanten Umgebung der modifizierten Montageobjekte symbolische Bezüge regeneriert - unter Zuhilfenahme der Extern- und Commontabellen.

Danach werden evtl. die Commontabellen korrigiert sowie die Externtabellen, die zu Deklarationen in den ersetzten Montageobjekten gehörten mitsamt den Verweisen auf sie in der Symboltabelle gelöscht.

Eine Komplikation tritt ein, wenn in einem Montageobjekt durch die Änderung eine globale Deklaration eingefügt wurde, deren Name (und damit auch ihre Eintragung in der Symboltabelle) bisher auf ein anderes Montageobjekt verwies.

In diesem Fall muß nicht nur die Eintragung in der Symboltabelle geändert werden, sondern auch das Montageobjekt, auf das der symbolische Verweis bisher zeigte, muß als geändert angesehen werden.

c) Der Einfügungslauf

Dieser Lauf ist notwendig, wenn durch die Änderung in einem Montageobjekt zusätzliche Externbezüge auf Montageobjekte hinzugekommen sind, die bisher noch nicht benötigt wurden. In diesem Fall müssen nämlich

1. die Kontrolltabelle und
2. die Symboltabelle erweitert werden.
3. Evtl. müssen auch Commontabellen verlängert bzw. neue Commontabellen hinzugefügt werden.
4. Tritt der Fall ein, daß in den hinzugekommenen Montageobjekten ebenfalls Bezug auf bereits vorhandene Commonzonen genommen wird, so hat das die gleichen Folgen wie die Ersetzung der Montageobjekte, die diese Commonzonen benutzen.

Es soll nun behandelt werden, wie sich die anderen Forderungen an einen Montierer in das hier entworfene Konzept einfügen lassen.

Die Erfüllung der Forderung 3.3.2, dynamisch während des Programmlaufes Montageobjekte zu ersetzen oder Montageobjekte anzufügen, deren Namen erst im Lauf festgelegt werden, ergibt sich fast von selbst durch einen dazwischengeschalteten Ersetzungs- oder Einfügungslauf.

Allerdings tauchen zusätzliche Probleme dadurch auf, daß die für ein infolge der Ersetzung/Einfügung modifiziertes Montageobjekt relevante Umgebung nicht mehr nur durch die statischen symbolischen Bezüge bestimmt wird, sondern auch durch Register- und dynamisch geänderte Speicherinhalte.

Wenn nämlich Adressen von umgewandelten symbolischen Bezügen an fremde Montageobjekte weitergereicht werden, läßt sich nach einer Änderung von Montageobjekten, in denen diese symbolischen Namen deklariert sind, nicht mehr feststellen, daß die Adressen weitergereicht wurden; folglich läßt sich dann auch nicht mehr sagen, welche Register bzw. Speicherinhalte durch die Modifizierung unbrauchbar geworden sind.

Dieses Problem läßt sich allgemeingültig nur durch ein generelles Verbot solcher Manipulationen (die ohnehin nur auf Assemblerebene möglich sind) lösen, anderenfalls liefert die Ersetzung oder Einfügung von Montageobjekten während des Operatorlaufes u.U. kein definiertes Ergebnis mehr.

Ein weiteres generelles Problem bei diesem Montierer-konzept soll auch an dieser Stelle angeschnitten werden, da es zum gleichen Problemkreis gehört:

Es muß sichergestellt sein, daß der Operator reentrant aufgebaut ist.

Das bedeutet, daß der Operator selbst nichts an seinem eigenen Code, seinen Konstanten oder der Vorbesetzung seiner Variablen ändern darf, da sonst durch einen ähnlichen Effekt wie den gerade beschriebenen des Weiterreichens von umgewandelten symbolischen Bezügen evtl. das Ergebnis der Nachmontierer-Aktionen nicht mehr definiert ist.

Damit sind natürlich nicht die Änderungen der Kontrollprozedur am Code und an Konstanten gemeint oder diejenigen des Montierers.

Der Montierer muß vielmehr selbst dafür sorgen, daß der (sukzessive) erzeugte Operatorkörper reentrant aufgebaut ist, indem nämlich für die Variablenbereiche eine

Kopie der Vorbesetzungen (siehe DATA/BLOCKDATA-Anweisungen in FORTRAN) auf dem Hintergrundspeicher angelegt wird, mit der die Variablenbereiche bei jedem Operatorstart vom Lader überschrieben und somit initialisiert werden.

4.2 Der Vormontierer

4.2.1 Verknüpfung mehrer Montageobjekte zu einem Montageobjekt

Die Wünsche nach Verknüpfbarkeit von Montageobjekten lassen sich unter zwei wesentlichen Gesichtspunkten zusammenfassen:

a) Man möchte die Zahl der globalen Deklarationen und der symbolischen Bezüge innerhalb einer Menge von Montageobjekten vermindern, um Speicherplatz in einer Montageobjekt-Bibliothek und Rechenzeit bei der Montage einzusparen.

b) Man möchte die Struktur der globalen Namen und der Bezüge auf sie manipulieren, um möglichst modular Programmsysteme entwerfen zu können und Namenskonflikte zwischen getrennten Montageobjekt-Paketen zu vermeiden.

Da das Ergebnis wiederum ein Objekt im Montageobjekt-Zwischencode sein soll, sind diese Überlegungen völlig unabhängig vom bisher entworfenen Nachmontierer-Konzept zu sehen.

Die Struktur der Verknüpfung von Montageobjekten mit ihren symbolischen Querbezügen können wir in Form eines gerichteten Graphens darstellen (siehe Abb. 6), wobei die Montageobjekte (MO_n , n= 1,...,8) die Knoten und die symbolischen Bezüge (E_n , n= 1,...,19) die Kanten sind.
Die Probleme, die bei der Verknüpfung auftreten, sind folgende:

a) Gewisse globale Deklarationen sollen zwar intern benutzt werden, aber außerhalb keine Gültigkeit haben.

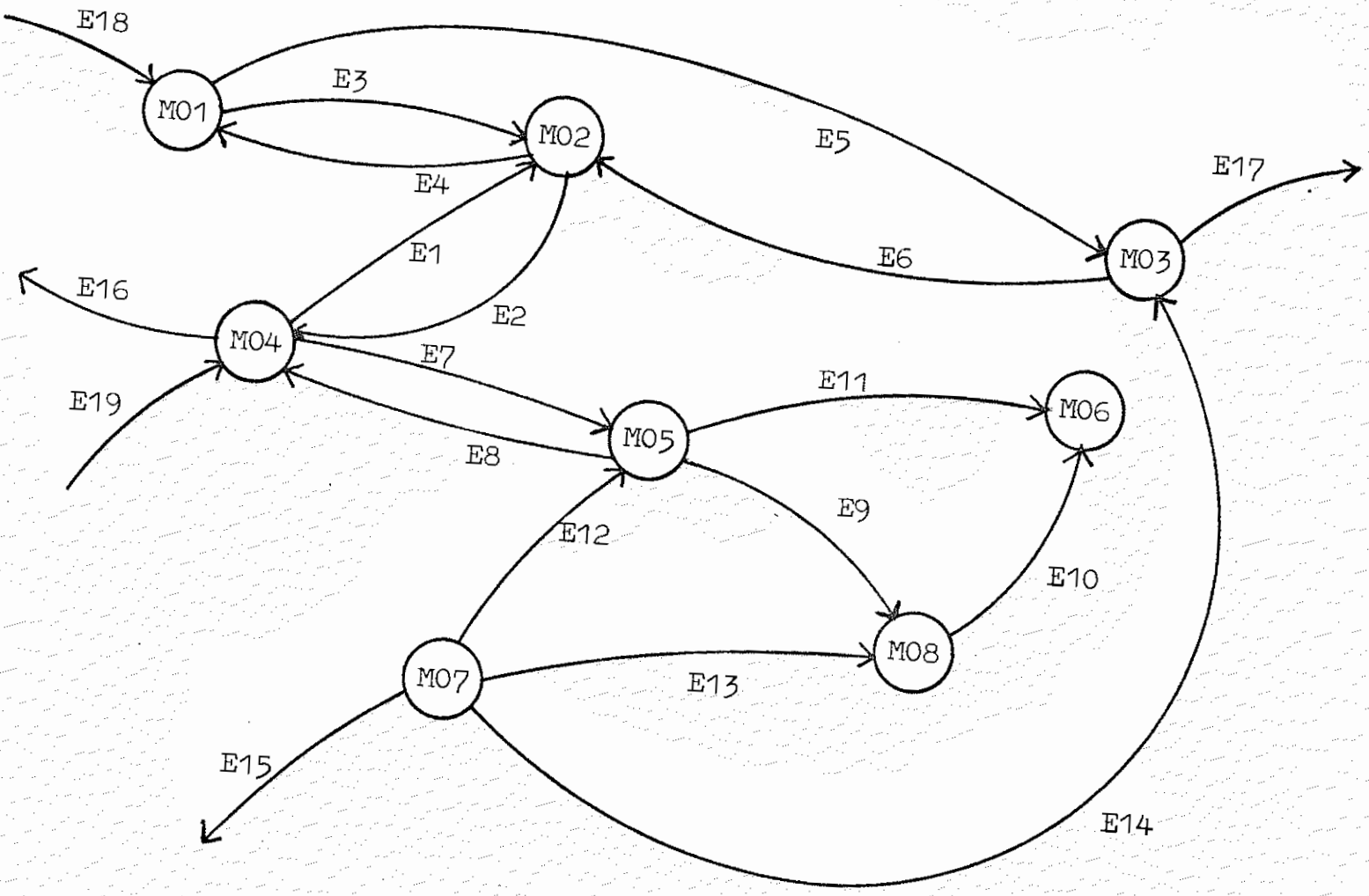

E18
M01
E3
M02
E5
E17
E4
E1
E6
M03
E16
E2
M04
E7
E11
E19
M06
M05
E8
E9
E12
E10
M08
M07
E13
E15
E14

Abb. 6

b) Globale Deklarationen können kollidieren, d.h. den gleichen Namen haben bei unterschiedlicher Bedeutung.

c) Symbolische Bezüge können kollidieren, indem sie sich unter dem gleichen Namen auf verschiedene Objekte beziehen.

d) Bei den meisten Rechnern ist es möglich, daß Zonen (z. B. Commonzonen) denselben Namen wie andere globale Deklarationen bzw. Bezüge haben, so daß die Bezüge anhand des Typs gefunden werden müssen.

e) Montageobjektnamen können kollidieren, wenn sie in verschiedenen Bibliotheken lagern.

Ein Teil der Probleme läßt sich dadurch lösen, daß Bibliotheken, in denen Montageobjekte lagern, in einer linearen Hierarchie zur Suche nach verlangten globalen Deklarationen herangezogen werden oder dadurch, daß bestimmte symbolische Bezüge im Montageobjekt an einen bestimmten Montageobjektnamen gebunden werden (z. B. Kontakte im TR44o [29]).

Diese Einschränkungen bzw. Zusatzinformationen sind jedoch ein viel zu grobes Beschreibungsmittel, um die komplizierte Struktur des gerichteten Graphens zu erfassen.

4.2.2 Formale Beschreibung des Vormontierers

Def. 1: a) Sei A ein Alphabet, A^+ die Menge aller Namen über dem Alphabet A.

b) Seien $B,E,C \subset A^+$ endlich.
E heißt Menge der Eingangsnamen,
B heißt Menge der symbolischen Bezugsnamen,
C heißt Menge der Commonzonen-Namen.
Dann ist $N := E \cup B \cup C$ die Menge der globalen Namen.

Bemerkung: A ist maschinenabhängig, z.B.:
$A = \{\langle Bu\rangle\} \cup \{\langle Zi\rangle\} \cup \{\langle Soz\rangle\}$
$\langle Bu\rangle$ = Menge der Buchstaben $\{A,\ldots,Z\}$
$\langle Zi\rangle$ = Menge der Ziffern $\{0,\ldots,9\}$
$\langle Soz\rangle$ = Menge der Sonderzeichen $\{\&,\$,\ldots\}$

Sei im folgenden $N = E \cup B \cup C \subset A^+$ eine Menge von (globalen) Namen.

Def. 2: Ein 4-Tupel
$m = (n_m, E_m, B_m, C_m)$ heißt Montageobjekt, wenn:

(i) $E_m \subset E$
$B_m \subset B$
$C_m \subset C$

(ii) $n_m \in E_m$

(iii) $E_m \cap B_m = \emptyset$

Das heißt, ein Montageobjekt wird gekennzeichnet durch seinen Namen, seine Eingangsnamen, seine symbolischen Bezugsnamen und seine Commonzonen-Namen, wobei sein Name ein ausgezeichneter Eingangsname ist.
Hinzu kommt, daß kein symbolischer Bezugsname gleichzeitig im selben Montageobjekt Eingangsname sein darf.

Def. 3: Sei M eine Menge von Montageobjekten.

$M_i \subset M$ ($1 \leq i \leq n < \infty$) heißt Bibliothek,

wenn gilt:

(i) $\bigcup_{1 \leq i \leq n} M_i = M$

(ii) $\bigwedge_{1 \leq i,j \leq n} M_i \cap M_j = \emptyset$

(iii) $m_1, m_2 \in M_i \Longrightarrow E_{m_1} \cap E_{m_2} = \emptyset$

Die Bedingung (ii) ist nach unserer Definition eines Montageobjektes eine willkürliche Einschränkung. Sie beschränkt jedoch nicht die Allgemeingültigkeit der Aussagen, da sie leicht zu umgehen wäre durch einen zusätzlichen Index, also durch eine Erweiterung der Montageobjekte auf 5-Tupel; diesen Index wollen wir aber zugunsten einer vereinfachten Schreibweise fortlassen.

Die Bedingung (iii) besagt, daß Eingangsnamen innerhalb einer Bibliothek eindeutig sein müssen, also insbesondere Montageobjektnamen. Für Kontaktnamen (siehe [29]) erreichen wir dies leicht durch eine Verlängerung des Kontaktnamens um den Montageobjektnamen, der ja auch bei einem Bezug auf einen Kontaktnamen mit angegeben werden muß.

Def. 4.1: Sei $\varphi := (n_\varphi, E_\varphi, B_\varphi, C_\varphi)$.

Wir nennen φ freies Montageobjekt, wenn:

(i) $E_\varphi = \{e_\varphi\} \subset A^+$, $e_\varphi \notin N$

(ii) $B_\varphi = \{b_\varphi\} \subset A^+$, $b_\varphi \notin N$

(iii) $n_\varphi \in A^+$, $n_\varphi \notin N$

(IV) $C_\varphi = \emptyset$

Def. 4.2: Als Erweiterungen definieren wir:

$$\overline{N} := N \cup \{n_\varphi\}$$

$$\overline{E} := E \cup \{e_\varphi\}$$

$$\overline{B} := B \cup \{b_\varphi\}$$

Def. 5: Seien $M := \{m \mid m \text{ ist Montageobjekt}\}$

$$\overline{M} := M \cup \{\varphi\}$$

$$\overline{M} \subset \overline{N} \times \overline{E} \times \overline{B} \times C$$

Dann definieren wir die Projektionen

$$pre : \overline{M} \to \overline{E}$$

$$prb : \overline{M} \to \overline{B}$$

$$prc : \overline{M} \to C$$

mit :

$$pre(m) = E_m$$
$$prb(m) = B_m \qquad \text{für } m \in \overline{M}$$
$$prc(m) = C_m$$

Def. 6: Sei $\overline{M} \subset \overline{N} \times \overline{E} \times \overline{B} \times C$ eine Menge von Montageobjekten wie in Definition 5 , $\varphi \in \overline{M}$.

Dann nennen wir

$G \subset \overline{M} \times \overline{B} \times \overline{E} \times \overline{M}$ den gerichteten Graphen der symbolischen Bezüge von $\overline{M}$, wenn gilt:

(i) $G \ni g = (\hat{m}, b_{\hat{m}}, e_{\check{m}}, \check{m}) \Longrightarrow \hat{m} \neq \check{m}$

$\hat{m}, \check{m} \in \overline{M}$,

$b_{\hat{m}} \in prb(\hat{m})$,

$e_{\check{m}} \in pre(\check{m})$

(ii) $\bigwedge_{\hat{m} \in M} \left(b \in prb(\hat{m}) \Longrightarrow \bigvee_{\substack{\check{m} \in \overline{M}, \\ e \in pre(\check{m})}} (\hat{m}, b, e, \check{m}) \in G \right)$

Ein Element des gerichteten Graphen der symbolischen Bezüge besteht also jeweils aus einem Montageobjekt, einem seiner symbolischen Bezüge und dem dazugehörigen Eingang eines anderen Montageobjektes (i).
Zur Veranschaulichung:

Mit $G\ni(m,b_m,e_\varphi,\varphi)$ ist ein offener Außenbezug gemeint,

mit $G\ni(\varphi,b_\varphi,e_m,m)$ ein offener Eingang.

Wir kommen nun zur Definition des Vormontierers, der mehrere Montageobjekte zu einem einzigen verknüpft:

Def. 7: Sei M_v eine Menge von Montageobjekten, $M_v \subset \overline{M}$

Eine Abbildung $vm : M_v \longmapsto (m', G_v)$ heißt Vormontierer, wenn gilt:

(i) $m' = (n_{m'}, E_{m'}, B_{m'}, C_{m'})$ ist Montageobjekt

mit: $n_{m'} \in A^+$; $E_{m'}, B_{m'}, C_{m'} \subset A^+$

$\overline{M} \cup \{m'\} =: \overline{M}'$

$\overline{N} \cup \{n_{m'}\} =: \overline{N}'$

$\overline{E} \cup \{E_{m'}\} =: \overline{E}'$

$\overline{B} \cup \{B_{m'}\} =: \overline{B}'$

$C \cup C_{m'} =: \overline{C}'$

(ii) $m' \notin M_v$

(iii) G_v ist der gerichtete Graph der symbolischen Bezüge von M_v.

Die Bedingung (ii) bedeutet: Das neu zu erzeugende Montageobjekt m' kann nicht eines der zu verknüpfenden sein, wohl aber eines der vorhandenen ($m' \in \overline{M}$) - dieses würde dann ersetzt.

Def. 8: Sei $N \subset A^+$ eine Menge von globalen Namen.
Eine Abb. $u : N \longrightarrow A^+ \cup \{\lambda\} = A^*$ nennen wir Umbenennung, wenn gilt:

entweder (i) $u(n_1) = n_2 \in A^+ \wedge n_1 \neq n_2$; dann heißt n_1 umbenannt,

oder (ii) $u(n_1) = \lambda$; dann heißt n_1 als globaler Name gelöscht,

oder (iii) $u = id$ sonst.

Außerdem muß gelten:

(IV) Ist $m = (n_m, E_m, B_m, C_m)$ Montageobjekt, dann muß sein: $u(E_m) \cap u(B_m) = \emptyset$

Diese Bedingung ist notwendeig, damit die Montageobjekt-struktur von m erhalten bleibt (siehe Def.2 (iii)).

Satz: Sei im weiteren $M_v \subset \overline{M}$ eine Menge zu verknüpfender Montageobjekte, $\varphi \in M_v$.
Seien weiter

$$E_v := \bigcup_{m \in M_v} E_m$$

$$B_v := \bigcup_{m \in M_v} B_m$$

$$C_v := \bigcup_{m \in M_v} C_m$$

Dann existieren Abbildungen

$$\beta : M_v \longrightarrow B_1 \subset B_m,$$

$$\varepsilon : M_v \longrightarrow E_m,$$

$$\zeta : M_v \longrightarrow C_m,$$

$$\gamma : M_v \times M_v \longrightarrow 2^{M_v \times B_v \times E_v \times M_v}$$

so daß:

$vm : M_v \longrightarrow (m', G_v)$ Vormontierer ist

mit:

$$vm(M_v) = ((n_{m'}, \beta(M_v), \varepsilon(M_v), \zeta(M_v)), G_v) \quad \text{und}$$

$$G_v = \bigcup \gamma(M_v \times M_v)$$

Beweis: Der Beweis wird konstruktiv erbracht, indem die Abbildungen $\beta, \gamma, \varepsilon, \zeta$ angegeben werden.
(Die Angabe von $n_{m'} \in A^+$ ist trivial).

(i) Die Abbildungen β, ε und ζ stellen wir mit Hilfe von Umbenennungen nach Def. 8 dar:

$$\beta = u_\beta \circ prb$$

$$\varepsilon = u_\varepsilon \circ pre$$

$$\zeta = u_\zeta \circ prc$$

mit: $u_\beta, u_\varepsilon, u_\zeta$ Umbenennungen,
prb, pre, prc Projektionen wie in Def. 5.

Für u_β muß jedoch zusätzlich gelten:

$$\bigwedge_{b \in M_v} u_\beta(prb(b)) \neq \emptyset ,$$

da Bezüge immer abgesättigt werden müssen.

Damit sind $E_{m'}$ und $C_{m'}$ bereits eindeutig bestimmt.
Sei $m \in M_v$, φ freies Montageobjekt.
Dann erhalten wir $B_{m'}$ durch:

$$B_{m'} = B_1 \cup \left\{ b \in A^+ \;\middle|\; (m, b, e_\varphi, \varphi) \in G \right\}$$

Ein offener Bezug entsteht also entweder dadurch, daß er explizit durch β erzeugt wird, oder daß er in G_v auf einen Eingang des freien Montageobjektes φ zeigt, d.h. daß er nicht abgesättigt werden konnte.

(ii) Die Abbildung γ zur Konstruktion des gerichteten Graphen G_v der symbolischen Bezüge.

1.) Zur besseren Verständlichkeit wollen wir zunächst die entsprechende Abbildung γ' betrachten, die wir auch im herkömmlichen Montierer vorfinden.
Danach wollen wir durch Modifizierung von γ' die Abbildung γ so konstruieren, daß der Vormontierer zu einem wirkungsvollen Instrument zur Erfüllung der Forderungen 3.3.3 und 3.3.4 wird.

Def. 9: Wir definieren eine Gewichtsfunktion:
Sei M eine Menge von Montageobjekten, $M_i \subset M$ Bibliotheken.

$w : M \longrightarrow \mathbb{N}$ heißt Hierarchie, wenn gilt:

$$\bigwedge_{\substack{m_1 \in M_j \\ m_2 \in M_k}} w(m_1) = w(m_2) \Longleftrightarrow M_j = M_k$$

Wir sagen, M_j sei in einer höheren Hierarchiestufe als M_k, wenn $w(m_1) > w(m_2)$ für $m_1 \in M_j$, $m_2 \in M_k$.

Mit Hilfe einer solchen Hierarchie können wir γ' darstellen:

Seien $m_1, m_2 \in M_v$, $m_1 \neq m_2$, $n \in prb(m_1)$.

$$(\gamma'(m_1,\varphi) \ni (m_1,n,e_\varphi,\varphi) \Longleftrightarrow \bigwedge_{m \in M_v} n \notin pre(m))$$

$$\wedge (\gamma'(m_1,m_2) \ni (m_1,n,n,m_2)$$

$$\Longleftrightarrow n \in pre(m_2) \wedge (\bigwedge_{m \in M_v} n \in pre(m) \Longrightarrow w(m) < w(m_2)))$$

Beim herkömmlichen Montierer gilt also: Ein symbolischer Bezug n wird nur durch einen gleichnamigen Eingang in einem anderen Montageobjekt abgesättigt, und zwar bei Mehr-

deutigkeit in demjenigen Montageobjekt, das in der höchsten Hierarchiestufe unter allen Montageobjekten, die diesen Eingang haben, ist.

Zur Konstruktion von γ benötigen wir noch einige Hilfsdefinitionen:

Def. 10: Sei $D \subset M_v \times B_v \times E_v \times M_v$ eine Menge von Elementen der Form:

$$D = \{(m_1, b_1, e_2, m_2) \text{ mit: } m_1 \neq \varphi \neq m_2\}$$

Wir nennen D die Menge der Direktbezüge.

Def. 11: Wir definieren auf M_v eine Struktur, die durch einen Baum darstellbar ist, dessen Knoten aus Teilmengen von M_v bestehen.

Sei K eine Zerlegung von $M_v \setminus \{\varphi\}$ mit:

a) $2^{M_v \setminus \{\varphi\}} \supset K = \{k_i \mid i \in I \subset \mathbb{N} \cup \{o\}, |I| < \infty\}$

b) $\bigcup_{i \in I} k_i = M_v \setminus \{\varphi\}$

c) $\bigwedge_{\substack{i,j \in I \\ i \neq j}} k_i \cap k_j = \emptyset$

d) $\bigwedge_{i \in I} k_i \neq \emptyset$

Sei $H \subset K \times K$ und sei $h : H \longrightarrow \mathbb{N} \cup \{o\}$ eine Abstandsfunktion.

Wir nennen (h,H) eine Baumstruktur über M_v, wenn gilt:

(i) $(k_1,k_2) \in H \wedge (k_2,k_3) \in H \Longrightarrow (k_1,k_3) \in H$

(ii) $\bigwedge_{k_1,k_2 \in K} (((k_1,k_2) \in H \wedge (k_2,k_1) \in H) \Longleftrightarrow k_1 = k_2)$

(iii) $\bigvee_{k_o \in K} \bigwedge_{k_i \in K} (k_o,k_i) \in H$

(IV) h ist definiert durch:

a) $h(k_o) = o$

b) $h(k_i) = n+1$

$$\Longleftrightarrow \bigvee_{k_j \in K} h(k_j) = n \wedge \bigwedge_{k_1 \in K} ((k_j,k_1) \in H \wedge (k_1,k_j) \in H \Longrightarrow k_j = k_1 \vee k_1 = k_i)$$

(V) $\bigwedge_{i \in I} m_1 \in k_i \wedge m_2 \in k_i \Longrightarrow pre(m_1) \cap pre(m_2) = \emptyset$

Bemerkung:

a) Wie man leicht sieht, handelt es sich um eine transitive (i) , reflexive (ii) , antisymmetrische (ii) Halbordnung mit einem minimalen Element k_o (iii) .

b) Wir nennen k_o die Baumwurzel .

c) Wir sagen, k_1 liegt näher an der Baumwurzel als k_2 , wenn $(k_1,k_2) \in H \wedge h(k_1) < h(k_2)$.

d) Wir sagen, m_1 und m_2 liegen im selben Knoten, wenn $m_1, m_2 \in k_i$, $i \in I$.

Nun können wir die Abbildung γ konstruieren :

Sei (h,H) die Baumstruktur über M_v wie in Def. 11 , $u_v : E_v \cup B_v \longrightarrow A^+$ eine Umbenennung.

Seien $m_1, m_2 \in M_v$, $b_1 \in prb(m_1)$, $e_2 \in pre(m_2)$, $m_1 \in k_1$, $m_2 \in k_2$.

Dann gilt für γ :

$$\Big(\Big(\bigwedge_{m\in M_v}(m_1,b_1,pre(m),m)\notin D\Big)\wedge\bigwedge_{\substack{m\in M_v\\(m_1,m)\in H}}(u_v(b_1)\notin u_v(pre(m))\Big)$$

$$\Longrightarrow(m_1,b_1,e_\varphi,\varphi)\in\gamma(m_1,\varphi)) \qquad (a)$$

$$\wedge\,((m_1,b_1,e_2,m_2)\in D\Longrightarrow(m_1,b_1,e_2,m_2)\in\gamma(m_1,m_2)) \qquad (b)$$

$$\Big(\Big(\bigwedge_{m\in M_v}(m_1,b_1,pre(m),m),m\Big)\notin D\Big)\wedge\bigvee_{m_2\in M_v}u_v(b_1)=u_v(e_2)$$

$$\wedge\bigwedge_{m\in M_v}(u_1(b_1)\in u_v(pre(m))\wedge m\in k\wedge(k,k_1)\in H$$

$$\wedge\,h(k)\geq h(k_2)\Longrightarrow k=k_2)$$

$$\Longrightarrow(m_1,b_1,e_2,m_2)\in\gamma(m_1,m_2)) \qquad (c)$$

Das bedeutet:

Ein Element von G_v wird entweder durch ein Element der Direktbezüge definiert (b) oder durch einen mit der Baumstruktur und der Umbenennung nicht absättigbaren Bezug (a) oder durch einen Bezug, der (durch die Umbenennung) mit einem (umbenannten) Eingang eines Montageobjektes abgesättigt wird, das näher oder genauso nahe an der Baumwurzel liegt (c) , und zwar bei Mehrdeutigkeit desjenigen Montageobjektes, das in der Baumstruktur am nächsten liegt.

Damit ist $vm : M_v \longrightarrow (m', G_v)$ vollständig beschrieben.

4.2.3 Die Sprache zur Steuerung des Vormontierers

Um die geschilderte Struktur des Vormontierers realisieren zu können, wird ein Sprachmittel zu seiner Steuerung benötigt.

In Abb. 7 ist die syntaktische Beschreibung dieser Sprache in BNF - Notation (Backus-Naur-Form) dargestellt.

Sie ist so konstruiert, daß sie sich relativ leicht in die Kommandosprachen (job control languages) der verschiedenen Rechner einfügen läßt und insbesondere am TR440 leicht implementierbar ist; deswegen ist sie auf Keyword-Steuerung abgestellt.

4.2.3.1 Die Verbindung der Sprachelemente zur formalen Beschreibung.

Zunächst soll dargelegt werden, auf welche Art mit der hier beschriebenen Sprache die in 4.2.2 formal entwickelten Funktionen des Vormontierers realisiert werden können.

Die Umbenennung u_v wird durch die ⟨Referenzbeschreibung⟩ und die ⟨Deklarationsbeschreibung⟩ definiert, wobei damit nur diejenigen Elemente festgelegt werden, auf denen u_v nicht als id arbeitet.

Die Abbildungen β und ε werden durch die ⟨Offen-Bezugsbeschreibung⟩ , ζ durch die ⟨Commonbeschreibung⟩ definiert.

Mit Hilfe der ⟨Gruppenbeschreibung⟩ wird die Baumstruktur (h,H) der Montageobjekte definiert, wobei die transitive Hülle vom Vormontierer selbst gebildet wird (die $k_i, i \in I$ sind dabei die Gruppen).

Natürlich muß vom Vormontierer die Vollständigkeit (der Baum muß zusammenhängend sein) sowie die Widerspruchsfreiheit (Eindeutigkeit der Kanten zwischen je zwei Knoten) der Baumstruktur überprüft werden.

teuerung⟩::=⟨Steuerung⟩[⟨Trennung⟩⟨Steuerung⟩]∞

:= maschinenabhängig, z.B. durch verschiedene Spezifikationen auf Kommandoebene

::=⟨Gruppenbeschreibung⟩|⟨Referenzbeschreibung⟩|⟨Deklarationsbeschreibung⟩|⟨Zonen
|⟨Commonbeschreibung⟩|⟨Offen-Bezugsbeschreibung⟩|⟨Direkt-Bezugsbeschreibung⟩

nbeschreibung⟩::= GRUPPE=⟨Gruppendeklaration⟩[⟨Trennzeichen⟩⟨Gruppendeklaration⟩]

ennzeichen⟩::= maschinenabhängig, z.B. der Apostroph zum Abtrennen der Spezifikat

uppendeklaration⟩::=⟨neuer Gruppenname⟩(⟨Gruppenelement⟩[,⟨Gruppenelement⟩]∞)

⟨neuer Gruppenname⟩::=⟨Gruppenname⟩

⟨Gruppenelement⟩::=⟨Gruppenname⟩|⟨Montageobjektbezeichnung⟩

⟨Gruppenname⟩::=⟨Name⟩

⟨Montageobjektbezeichnung⟩::=⟨Montageobjektname⟩[(⟨Bibliotheksname⟩)]

nzbeschreibung⟩::= REFERENZ=⟨Symbolbeschreibungen⟩

mbolbeschreibungen⟩::=⟨Symbolbezeichnung⟩[⟨Trennzeichen⟩⟨Symbolbezeichnung⟩]∞

⟨Symbolbezeichnung⟩::=[⟨Montageobjektbezeichnung⟩]:⟨globaler Name⟩[⟨Umbenennung⟩

⟨Umbenennung⟩::= ([⟨neuer globaler Name⟩])

ationsbeschreibung⟩::= DEKLARATION=⟨Symbolbeschreibungen⟩

eschreibung⟩::= ZONE=⟨Symbolbeschreibungen⟩

Bezugsbeschreibung⟩::= OFFEN=⟨Symbolbeschreibungen⟩

beschreibung⟩::= COMMON=⟨Symbolbeschreibungen⟩

-Bezugsbeschreibung⟩::= DIREKT=⟨Bezugsbeschreibung⟩[⟨Trennzeichen⟩⟨Bezugsbeschrei

zugsbeschreibung⟩::=⟨Externbezug⟩:⟨Externdeklaration⟩

⟨Externbezug⟩::=⟨direkter Name⟩

⟨direkter Name⟩::=⟨globaler Name⟩[(⟨Montageobjektbezeichnung⟩)]

⟨Externdeklaration⟩::=⟨direkter Name⟩

Abb. 7

Durch die ⟨Direkt-Bezugsbeschreibung⟩ wird die Menge D definiert.

Mit dieser Sprache hat der Benutzer des Vormontierers die Möglichkeit, alle Funktionen des Vormontierens selbst variabel steuern zu können.

4.2.3.2 Beschreibung der Syntax und semantische Ergänzungen

A) Die ‹Gruppenbeschreibung›

Sie dient zum Definieren der Gruppen und gleichzeitig ihrer Baumstruktur. Sie besteht aus dem neu zu definierenden Gruppennamen und in Klammern dahinter ihren ‹Gruppenelementen› . Diese können entweder Namen von Untergruppen sein oder Namen von Montageobjekten, evtl. gefolgt von einem Bibliotheksnamen in Klammern. Tritt der Name einer Gruppe als Gruppenelement auf, so wird damit ein neuer Baumast angelegt, so daß zwei Gruppen, deren Namen Elemente derselben Gruppe sind, getrennte parallele Äste darstellen.

Jede Gruppe muß genau einmal definiert werden und darf höchstens einmal als ‹Gruppenelement› auftreten. Genau eine Gruppe muß vorhanden sein, die zwar deklariert wird, aber nicht ‹Gruppenelement› sein darf; sie bildet die Wurzel des Baumes, und ihr Name wird gleichzeitig der Name des neu zu erzeugenden Montageobjektes.

B) Die ‹Symbolbeschreibungen›

Mit ihrer Hilfe werden globale Namen fixiert und evtl. umbenannt. Sie bestehen aus einer ‹Montageobjektbezeichnung› (optional), gefolgt von einem Doppelpunkt und daran anschließend dem ‹globalen Namen›, evtl. gefolgt von einer ‹Umbenennung›. Fehlt die ‹Umbenennung›, so dient die ‹Symbolbeschreibung› lediglich der eindeutigen Zuordnung des globalen Namens zu einem bestimmten Montageobjekt.

C) Die ‹Umbenennung›

Sie besteht zunächst aus einem Klammernpaar, das leer sein kann; in diesem Fall wird der ‹globale Name› "gelöscht", d.h. er hat nur noch lokale Bedeutung in dem jeweiligen Montageobjekt (das ist natürlich verboten bei einem symbolischen Externbezug, da dieser abgesättigt werden muß).

Im anderen Fall erhält er den ‹neuen globalen Namen›, der in der Klammer steht, und zwar in allen Montageobjekten, falls keines durch eine ‹Montageobjektbezeichnung› spezifiziert wurde. Diese ‹Symbolbeschreibungen› werden benutzt in:

1) der ‹Referenzbeschreibung› zur Fixierung und Umbenennung von globalen Bezügen,

2) der ‹Deklarationsbeschreibung› zur Fixierung und Umbenennung von globalen Deklarationen,

3) der ‹Zonenbeschreibung› zur Umbenennung von Zonen (diese Unterscheidung ist notwendig, da zu Zonen gleichnamige globale Deklarationen existieren können, z. B. Commonzonen),

4) der ‹Offen-Bezugsbeschreibung› zur Kennzeichnung und Umbenennung von Namen, die im neu zu erzeugenden Montageobjekt offene globale Namen bleiben sollen (nicht abgesättigte Externbezüge bleiben in jedem Fall global, Externbezüge, die hier aufgeführt sind, werden auch dann nicht abgesättigt, wenn die entsprechende globale Deklaration in einem der zusammengefügten Montageobjekte vorhanden ist) und

5) der ‹Commonbeschreibung› zur Kennzeichnung und Umbenennung von Commonzonen, die in dem neu zu erzeugenden Montageobjekt Commonzonen bleiben sollen.

D) Die ‹Direkt-Bezugsbeschreibung›

Sie erlaubt beliebige Querverbindungen von symbolischen Bezügen unter Umgehung der Baumstruktur.
Sie besteht aus ‹Bezugsbeschreibung›en, die jeweils einen ‹Externbezug› und die ‹Externdeklaration›, durch die der ‹Externbezug› abgesättigt werden soll, enthalten, durch Doppelpunkt getrennt. Beide können durch Angabe einer ‹Montageobjektbezeichnung› genau spezifiziert werden.

E) Die ⟨Montageobjektbezeichnung⟩

Sie dient zur Kennzeichnung eines bestimmten Montageobjektes. Der ⟨Montageobjektname⟩ kann näher spezifiziert werden durch einen ⟨Bibliotheksnamen⟩; fehlt dieser, so wird das Montageobjekt nach Maßgabe der Bibliothekshierarchie identifiziert (⟨Montageobjektnamen⟩ müssen innerhalb eine Bibliothek eindeutig sein).

Ein ⟨globaler Name⟩, bei dem die ⟨Montageobjektbezeichnung⟩ fehlt, muß eindeutig unter allen Montageobjekten aller Bibliotheken sein.

Zum besseren Verständnis wollen wir ein Beispiel und dazu den analogen Aufbau auf Quellebene einer ALGOL6o-Struktur betrachten. Dabei sollen uns die ALGOL-Prozedurnamen zugleich als Montageobjektnamen und als symbolische Namen dienen - der Aufruf einer Prozedur ist also ein symbolischer Bezug, während die Deklaration einer Prozedur eine symbolische Deklaration ist.

Dem Zusammenfassen von Montageobjekten in Gruppen entspricht dann das Umfassen von Prozeduren durch einen ALGOL6o-Block; die Namen aller in diesem Block deklarierten Prozeduren sind ja in diesem und allen weiteren eingeschachtelten Blöcken globale Namen.

Wir betrachten eine Menge von Prozeduren Pi (i=o,...,8) und strukturieren sie mittels folgender ‹Gruppenbeschreibung› :

```
GRUPPE=STDHP(Po,G1,G2,G3)'G1(P1,P2)'
       G2(P3,P4,G4,G5)'G3(P5)'G4(P6)'G5(P7,P8)
```

Dem entspricht eine Baumstruktur nach Abb. 9 .
In Abb. 8 sehen wir die analoge Blockstruktur für ALGOL und in Abb. 1o eine Tabelle für die Namensgültigkeiten (die Prozedur-Rümpfe und Block-Statements sind durch "..." symbolisiert).

Mittels einer ‹Direkt-Bezugsbeschreibung›

```
DIREKT=A(P1):B(P7)
```

könnte man erreichen, daß ein in P7 deklarierter Name "B" (in Abb. 8 nicht aufgeführt) als "A" in P1 bekannt wäre, also ein gezieltes Durchbrechen der Blockstruktur, das in ALGOL6o nicht möglich ist.

```
begin comment STDHP ;
      procedure PO ;
            ...
      begin comment G1 ;
            procedure P1 ;
                  ...
            procedure P2 ;
                  ...
      end G1 ;
      begin comment G2 ;
            procedure P3 ;
                  ...
            procedure P4 ;
                  ...
            begin comment G4 ;
                  procedure P6 ;
                        ...
            end G4 ;
            begin comment G5 ;
                  procedure P7 ;
                        ...
                  procedure P8 ;
                        ...
            end G5 ;
      end G2 ;
      begin comment G3 ;
            procedure P5 ;
                  ...
      end G3 ;
end STDHP ;
```

Abb. 8

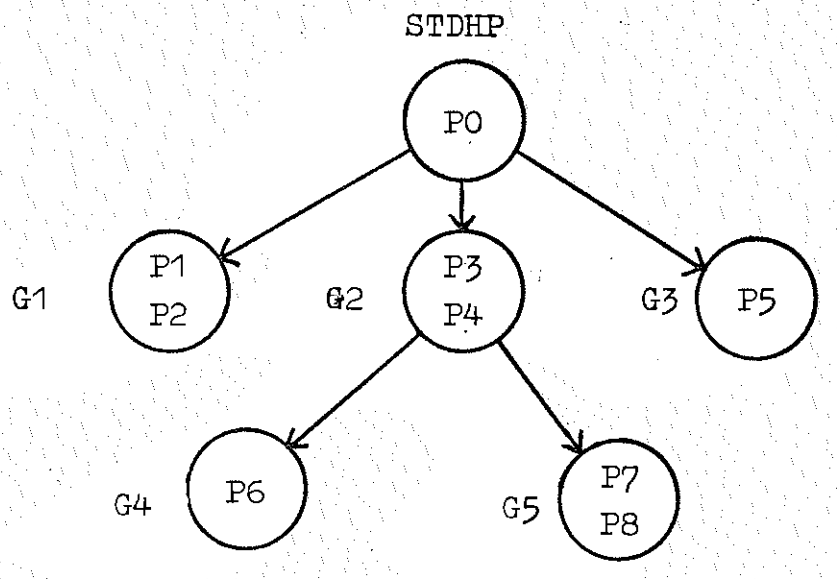


Abb. 9

gültig in \ Name	P0	P1	P2	P3	P4	P5	P6	P7	P8
P0	x								
P1	x	x	x						
P2	x	x	x						
P3	x			x	x				
P4	x			x	x				
P5	x					x			
P6	x			x	x		x		
P7	x			x	x			x	x
P8	x			x	x			x	x

Abb. 10

4.2.4 Der Auswertungsalgorithmus für die Sprache

Es wird nun ein Algorithmus für die Auswertung dieser Sprache beschrieben (siehe auch Abb. 11 und 12).

Zunächst wird die ⟨Gruppenbeschreibung⟩ ausgewertet, um die Gruppen-Baumstruktur zu erzeugen. Dazu werden alle ⟨Gruppendeklaration⟩en untersucht und die darin definierten ⟨Gruppenname⟩n in der Grüppenliste zusammengefaßt. Dann werden der Reihe nach die ⟨Gruppenelement⟩e aller Gruppen untersucht.

Jedes ⟨Gruppenelement⟩, das in der Gruppenliste ist, bekommt einen Verweis auf die übergeordnete Gruppe, deren Element sie ist. Die übrigen ⟨Gruppenelemt⟩e werden als ⟨Montageobjektbezeichnung⟩en interpretiert und in die Montageobjektliste eingefügt. Dabei erhält jedes Montageobjekt in der Montageobjektliste einen Rückverweis auf seine Gruppe und die Gruppe einen Verweis auf ihr erstes Montageobjekt sowie eine Angabe über die Anzahl ihrer Montageobjekte.

Daran schließt sich die Erzeugung der Deklarationsliste und der Referenzliste an. In der Deklarationsliste werden zunächst alle Deklarationen aus allen Montageobjekten der Montageobjektliste aufgeführt mit entsprechenden Verweisen in der Montageobjektliste und Rückverweisen in der Deklarationsliste sowie dem Typ der Deklaration (Zone, Commonzone, Eingang, absolute Größe etc.).

Mit Hilfe der ⟨Zonenbeschreibung⟩ und der ⟨Deklarationsbeschreibung⟩ werden dann Umbenennungen in der Deklarationsliste vorgenommen, wobei die alten durch die neuen Namen ersetzt werden, die alten aber gemerkt werden müssen, um sie hinterher im Montageobjekt wiederzufinden.

Dann werden mit der ⟨Direkt-Bezugsbeschreibung⟩ die explizit aufgeführten globalen Bezüge abgesättigt und mit einem Verweis auf das entsprechende Element der Deklarationsliste versehen.

BEISPIEL :

```
GRUPPE=G1(A,G2,G3,TEST(BIB1),I)'G2(C,D,G4,G5)'G4(E,F)
      'G5(G,H(BIB2),K(BIB3))'G3(L,M,N,G6)'G6(P)

REFERENZ=TEST(BIB1):ENTRY1(ENTRY87)'D:EINGGA':EING1(EING2)

OFFEN=TEST(BIB1):ENTRY87'EING2'E4

DIREKT=EXT1(c):DEF4(a)'EXT5(TEST(BIB1)):DEF7(H(BIB2))
```

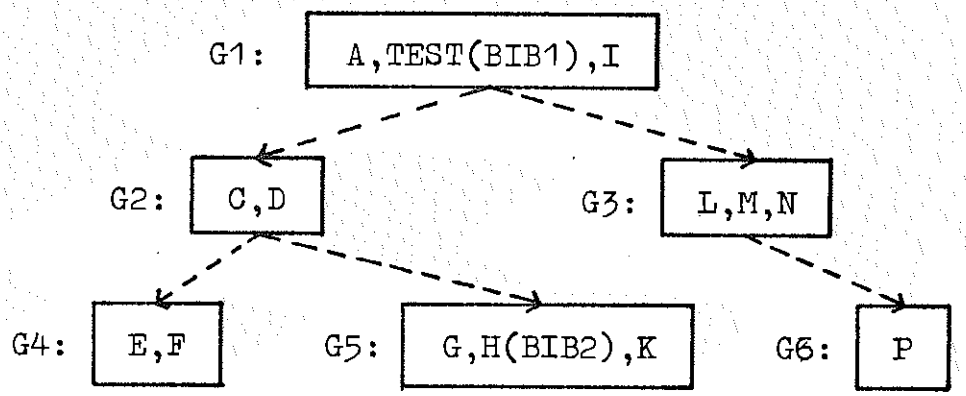
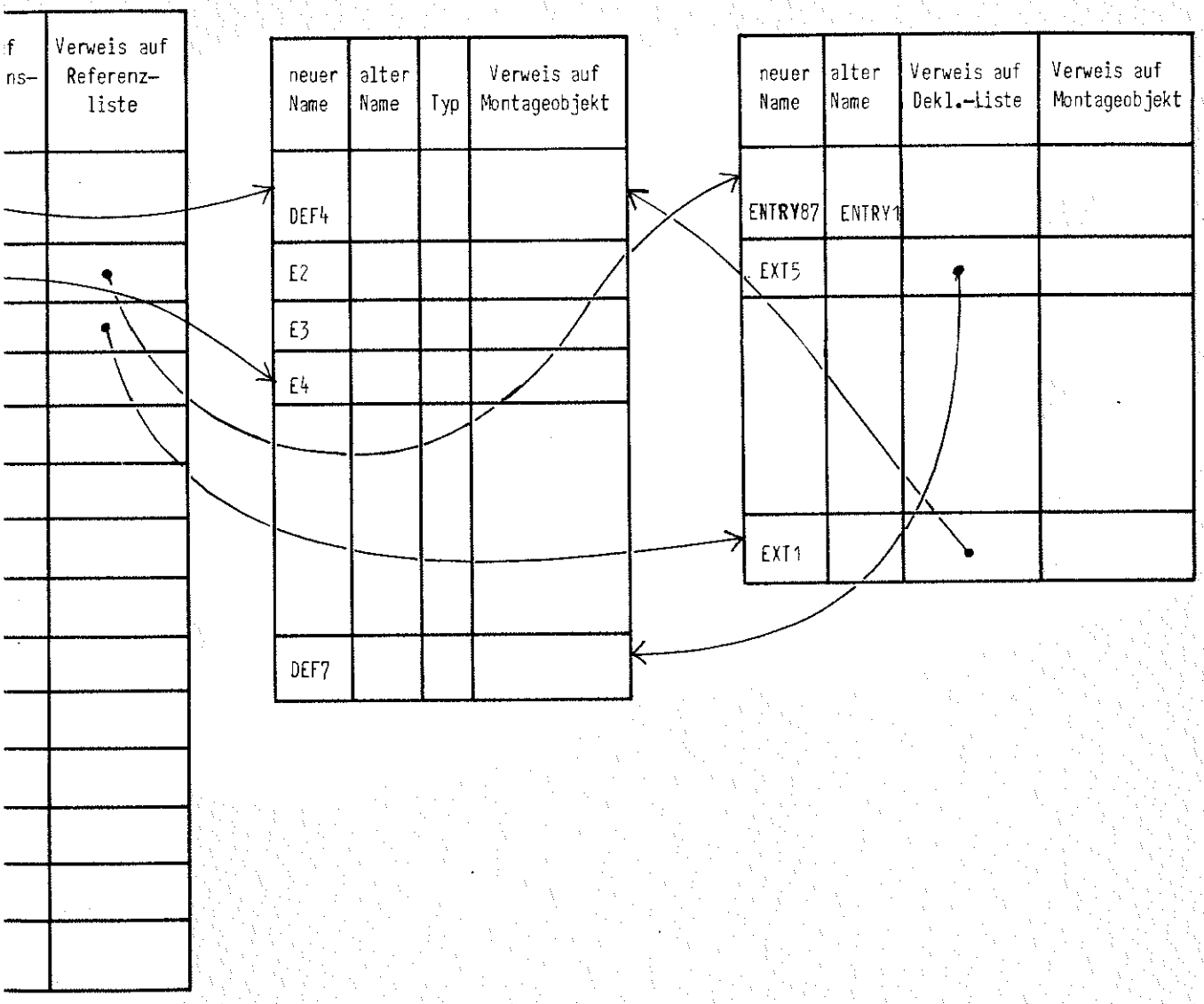


Abb. 11

E

DEKLARATIONSLISTE

REFERENZLISTE



Anhand der ‹Offen-Bezugsbeschreibung› wird dann

a) eine Liste derjenigen globalen Deklarationen angelegt, die global bleiben sollen in dem neuen Montageobjekt (dazu wird noch die ‹Commonbeschreibung› herangezogen) und
b) diejenigen Bezüge in der Referenzliste als "nicht absättigbar" gekennzeichnet, die als globale Bezüge erhalten bleiben sollen.

Anschließend werden alle Elemente der Montageobjektliste daraufhin untersucht, ob der zugehörige Teil der Referenzliste noch offene Bezüge enthält, die nicht als "nicht absättigbar" markiert sind.

Zunächst wird innerhalb der Gruppe des Montageobjektes eine passende globale Deklaration in der Deklarationsliste gesucht. Falls dort keine gefunden wird, wird die Baumstruktur anhand der Verweise auf die übergeordneten Gruppen und ihre zugehörigen Montageobjektlisten-Teile durchsucht.

Zum Schluß bleiben nur noch diejenigen globalen Deklarationen und Bezüge übrig, die entweder in der ‹Offen-Bezugsbeschreibung› enthalten sind oder nicht abgesättigt werden konnten (Bezüge).

Nach dieser Manipulation der globalen Deklarationen und Bezüge wird dann der eigentliche Code des neu zu erstellenden Montageobjektes erzeugt:

Alle unbenannten Zonen werden, soweit ihre Ablage- und Speicherschutz-Bedingungen dies zulassen, zu einer oder auch mehreren Zonen zusammengefaßt und die Relativbezüge in ihnen entsprechend korrigiert, d.h., falls nötig, werden die zonenrelativen Adressen so mit additiven Konstanten versehen, daß die Translationen auf den Anfang der neuen, größeren Zonen stimmen.

Dabei ist evtl. zu berücksichtigen, daß es wie bei IBM eine maximal zulässige Größe von Zonen (4o96 Bytes je

Basisregister, das für die Zone zur Verfügung steht) geben kann, die ihren Grund in der Anzahl der Bits hat, die im Adreßteil der Befehle für die Relativadressen zu den Basisregistern vorgesehen sind [12]. Bei der Überschreitung der Maximalgröße muß eine Zonenzusammenfassung natürlich unterbleiben - es sei denn, die Basisregister der ursprünglichen Zonen sind so gewählt, daß sie nicht kollidieren.

Problemlos ist die Behandlung gleichnamiger Commonzonen: sie werden nur einfach angelegt, wobei die Angabe ihrer Größe aus der maximal vorkommenden bestimmt wird. Wenn wie beim TR440 eine Vorbesetzung von Commonzonen möglich ist [29], muß diese wie sonst beim Montieren aus allen beteiligten Montageobjekten zusammengesucht werden.

Wenn der Name der Commonzone nicht in der ‹Commonbeschreibung› aufgeführt ist, wird sie nach der Zusammenlegung zu einer einfachen Zone gemacht, so daß sie bei der späteren Montage nicht mehr als Commonzone benutzt wird und somit ihr Name wieder frei verfügbar ist.

4.2.5 Vormontage zu einem teilweise fertigen Operatorkörper

Grundsätzlich gilt zunächst alles, was über die Vormontage mehrerer Montageobjekte zu einem einzigen gesagt wurde, auch, wenn das Ergebnis der Vormontage nicht wieder ein Objekt im Montageobjekt-Zwischencode sein soll, sondern ein Objekt im Maschinencode. Das bedeutet, daß die im letzten Kapitel entworfene Syntax zur Steuerung der Gültigkeit von globalen Namen auch in diesem Fall angewendet werden kann. Auch der in 4.2.4 beschriebene Algorithmus zur Realisierung ist im wesentlichen derselbe.

Zusätzlich wird das Operatorkörper-Gerüst bereits angelegt wie unter 4.1.3 beschrieben, bestehend aus dem Code der Kontrollprozedur sowie den statischen Tabellen wie der Symboltabelle und der Kontrolltabelle.

Darüber hinaus wird aber auch schon der Maschinencode aller an der Vormontage beteiligten Montageobejekte generiert und in den Operatorkörper eingefügt, da davon ausgegangen werden kann, daß diese ausgetestet sind und nicht ersetzt werden müssen - die Zeiger auf Maschinencode in der Kontrolltabelle werden also schon alle auf gültig gesetzt.

Das hat gegenüber der Verknüpfung zu einem Montageobjekt die Vorteile, daß erstens ein großer Teil der Montagezeiten einer auf diese Art vormontierten Bibliothek entfallen kann, und daß zweitens nach einer Modifizierung anderer Montageobjekte keine Umgebungs-Untersuchungen in den vormontierten Montageobjekten durchgeführt werden müssen, da einfach wieder ein Kopie des gesamten vormontierten Komplexes im Operatorkörper ersetzt werden kann, was als reiner Transportvorgang kaum Rechenzeit beansprucht.

Auch mehrere solcher vormontierten Komplexe können in einem Operatorkörper benutzt werden, nur müssen dann die mehrfach vorhandenen statischen Tabellen verknüpft werden, und der Code der Kontrollprozedur darf nur einmal abgelegt werden.

5. REALISIERUNG UNTER BERÜCKSICHTIGUNG SPEZIELLER RECHNEREIGENSCHAFTEN

Beim Vormontierer treten eigentlich keine speziellen Schwierigkeiten auf - es handelt sich dort lediglich um konventionelle Standardtechniken wie Listenmanipulationen, Auswertung von Startinformation sowie leichte Veränderungen des Montageobjekt-Zwischencodes durch Änderung von Translationen, Zusammenfassung von Zonen etc.
Beim Nachmontierer hingegen tauchen Probleme auf, die teils gar nicht, teils nur mit bestimmten Forderungen an die Hardware sowie die Systemsoftware lösbar sind.
Das letzte Kapitel soll speziellen Implementierungsschwierigkeiten gewidmet sein, die sich daraus ergeben.

5.1 Der Sprung in die Kontrollprozedur

Diesem Punkt muß besondere Aufmerksamkeit geschenkt werden, denn solche Sprünge treten statisch und dynamisch sehr häufig auf; d.h. es muß dafür gesorgt werden, daß sie sowohl an Speicherplatz als auch an Rechenzeit sehr wenig benötigen.

Außerdem muß, will man den Montageobjekt-Zwischencode nicht strengen Einschränkungen unterwerfen, sichergestellt sein, daß ein Sprung in die Kontrollprozedur, im folgenden kurz Kontrollsprung genannt, nicht mehr Code in Anspruch nimmt als der Befehl, der durch ihn ersetzt wird. Anderenfalls würde auch zuviel Zeit benötigt für eine Umordnung des Montageobjekt-Zwischencodes, da durch Verschieben des nachfolgenden Codes die meisten Adressen abgeändert werden müßten.

Dem steht gegenüber, daß die Kontrollprozedur mit mehreren Daten versorgt werden muß, nämlich:

a) mit einer Spezifizierung der symbolischen Deklaration, auf die sich der Befehl bezieht (also mit einem Verweis auf ein Element der Symboltabelle),

b) mit dem Operationscode des Befehls, der durch den Kontrollsprung ersetzt wurde und

c) mit der Rücksprungadresse.

d) Außerdem sollen noch alle Register und Indexregister unverändert erhalten bleiben, da sie im weiteren Programmablauf evtl. noch benötigt werden.

Diese Forderungen schließen einen normalen Unterprogramm-Sprung in die Kontrollprozedur aus.

Lediglich der spezielle Sprung in die Kontrollprozedur beim Sprung in ungültigen Code (siehe 4.1.3.2) kann ein Unterprogramm-Sprung sein, da er keinerlei Versorgung benötigt; in der Kontrollprozedur selbst kann die Information gespeichert werden, zu welchem (ersetzten) Montageobjekt der ungültige Code gehörte.

Als einzige Alternative zum Unterprogramm-Sprung bleibt eine spezielle Hardwareunterbrechung (Interrupt) zur Realisierung des Kontrollsprunges.
Dazu bieten sich zwei Möglichkeiten an:

a) ein programmierter Speicherschutzalarm
b) ein programmierter Makroalarm.

Der Speicherschutzalarm könnte so verwendet werden, daß der originale Operationscode erhalten bleibt, der Adreßteil jedoch durch eine Adresse ersetzt wird, die nicht zugeteilt ist (einen Schutzmechanismus gegen Zugriff auf nicht zugewiesene Speicherteile gibt es bei allen Rechnern).

Gleichzeitig muß jedoch eine eindeutige Zuordnung zwischen dieser Adresse und einem Element der Symboltabelle möglich sein, so daß u.U. ein ziemlich großer Adreßraum freigehalten werden muß und somit nicht benutzbar ist.

Außerdem gibt es Befehle, die nur das Laden des Adreßteiles in ein Register bewirken (z. B. TR440 : BA ; IBM : LA), wobei keine Überprüfung der Adresse auf Gültigkeit erfolgt, die also noch keinen Speicherschutzalarm auslösen.

Aus diesen Gründen empfiehlt sich der Makroalarm als Kontrollsprung.

Der Makroalarm ist eine Hardwareunterbrechung, die durch einen ungültigen Operationscode ausgelöst wird.
"Ungültig" kann dabei bedeuten, daß der Operationscode mit gar keiner Bedeutung belegt ist, oder daß ein bestimmter Befehl nicht von der Hardware sondern von der Software interpretiert wird (z. B. auch bei kleinerem Rechnerausbau).

Beim Auftreten eines Makroalarmes werden zunächst hardwaremäßig alle Register sowie alle den aktuellen Prozeß beschreibenden Werte (z. B. das PSW bei IBM [14]) an wohldefinierter Stelle abgespeichert (im sogenannten Alarmkeller).

Der Systemkern überprüft anhand des Operationscodes, ob es sich um einen zu interpretierenden Befehl handelt; wenn nicht, so setzt er den Prozeß, unter dessen Regie der Makroalarm auftrat, auf einer bestimmten Adresse, der Alarmadresse, fort. Der Prozeß kann dann seinerseits entscheiden, was zu tun ist - im Normalfall handelt es sich um einen Programmierfehler (z. B. einen Sprung in einem Datenbereich o.ä.), der zum Programmabbruch führen muß.

Im Falle unseres programmierten Makroalarmes muß die Alarmadresse in der Kontrollprozedur liegen. Aus dem Alarmkeller kann sich die Kontrollprozedur die benötigte Versorgung beschaffen, nämlich die Adresse, an der der Makroalarm auftrat, den Operationscode des Makroalarms sowie dessen Adreßteil.

Bei allen betrachteten Rechnern ist es nun möglich, durch die Kombination Makrocode/Adreßteil gleichzeitig eindeutig den echten Operationscode und den Verweis auf die Symboltabelle darzustellen.
Dazu drei Beispiele:

a) CDC: von 128 Befehlen sind 8 nicht belegt. Ein Speicheroperand wird durch 18 Bits dargestellt. Werden davon 4 Bits mit zur Kennzeichnung des Operationscodes herangezogen, bleiben 14 Bits zur Adressierung des Symboltabellenelementes.

b) IBM: über 1oo von 256 Befehlen sind nicht belegt. Da nicht mehr als 1oo Befehle Speicheroperanden haben, ist damit eine Zuordnung bereits eindeutig herstellbar. Für die Symboltabellenelemente stehen 16 Bits zur Verfügung, da für jeden Speicheroperanden im Befehlsformat 16 Bits vorgesehen sind.

c) TR44o: Bei diesem Rechner ist die eindeutige Zuordnung am knappsten, denn es sind (im Normalmodus) 27 von 256 Befehlscodes nicht belegt. Etwa 18o Befehle haben einen Speicheroperanden, so daß unter Hinzunahme weiterer 3 Bits aus dem Adreßteil eine eindeutige Zuordnung der Makrocodes zu den Operationscodes möglich ist. Dann bleiben noch 13 Bits im Adreßteil zur Kennzeichnung eines von maximal 8192 möglichen Symboltabellenelementen - diese Zahl dürfte jedoch in allen Fällen ausreichen.

Vorbedingungen für die Verwendung des Makroalarmes als Kontrollsprung sind natürlich auf der einen Seite, daß sowohl Hardware als auch System-Software es erlauben, auf unterster Programmierebene solche Alarme abzuhandeln und danach normiert den Programmlauf fortzusetzen (evtl. nach einem eingeschachtelten Montierer-Zwischenlauf), auf der anderen Seite, daß die Alarmadresse nicht vom Assembler-Programmierer zwischenzeitlich verändert wird.

5.2 Adreßkonstanten

Am meisten Schwierigkeiten bei der Implementierung des Nachmontierers bereiten Adreßkonstanten von symbolischen Bezügen.

Eigentlich würde es genügen, wenn ein Kontrollsprung bei der Auswertung des Inhalts einer solchen Adreßkonstanten erfolgte, denn erst dann wird der symbolische Bezug ja tatsächlich angesprochen.

Da es aber zu aufwendig bzw. häufig sogar unmöglich ist, den dynamischen Ablauf vorher zu untersuchen, muß bereits jeder Befehl, der die Zelle, in der sich die Adreßkonstante befindet, anspricht, durch einen Kontrollsprung ersetzt werden. Dabei ist jedoch zu beachten, daß die Kontrollprozedur in der Lage sein muß, zu unterscheiden, ob der Grund für den Kontrollsprung das Ansprechen der Speicherzelle selbst (a) oder das Ansprechen ihres Inhalts (b) war.

Beispiel im TR440:

a) EXTERN TEST, SFBE TEST,
b) EXTERN TEST, SFBE(TEST/A),

Diese Unterscheidungsmöglichkeit läßt sich nur durch Reservierung eines weiteren Bits im Adreßteil des generierten Makrobefehls realisieren, was allerdings bedeutet, daß sich die maximal mögliche Anzahl von Symboltabellenelementen halbiert, also z. B. beim TR440 auf 4096 verringert. In Anbetracht der Tatsache, daß im Falle eines Überschreitens dieser Zahl im Durchschnitt nur ca. 30 Ganzworte Speicher pro Element (z. B. pro Unterprogramm) zur Verfügung stünden, erscheint dies jedoch kaum als wesentliche Einschränkung.

Bisher wurde davon ausgegangen, daß eine Adreßkonstante direkt angesprochen wurde, daß also ihre Adresse direkt Adreßteil eines Befehles war. In der Tat müssen alle anderen Möglichkeiten ausgeschlossen werden (benutzen kann sie sowieso nur der Assemblerprogrammierer), da es aussichtslos ist,

alle im Programmlauf dynamisch erzeugten Adreßteile schon im voraus während einer (Teil-) Montage daraufhin zu untersuchen, ob etwa das Ergebnis eine symbolische Adreßkonstante bezeichnet.
(TR440-Beispiel: R MCFU A, SFBE TAB-- TAB= Tabelle von symbolischen Adreßkonstanten --).

Wichtig für symbolische Adreßkonstanten ist weiterhin, daß für die Kontrollprozedur entscheidbar sein muß, ob eine solche Adreßkonstante bereits umgewandelt ist oder nicht (wenn sie an verschiedenen Stellen angesprochen wird). Da aber bei allen Rechnern mindestens die ersten 2 Bits innerhalb einer Adreßkonstanten frei sind (weil wir nur Programme betrachten, die unter der Regie eines Betriebssystems laufen, können wir solche Adreßkonstanten wie die vom Typ Y bei IBM[12] außer acht lassen), kann eines davon zur Kennzeichnung des Umwandlungs-Zustandes verwendet werden (steht dann aber auch dem Assemblerprogrammierer nicht mehr zur freien Verfügung).

[1] Barron, D.W.: Assembler und Lader.
Carl Hanser Verlag. München 1970

[2] Brauer, W.; Indermark, K.: Algorithmen, rekursive Funktionen und formale Sprachen.
BI-Hochschulskripten 817

[3] Buchmann, R.: BODAT, ein schnelles und platzsparendes System zur Datenmanipulation und -speicherung in ALGOL6o und FORTRAN. Arbeitsbericht des Rechenzentrums der Ruhr-Universität Bochum Nr. 7405

[4] Burroughs: B6700
Espol Language Information Manual

[5] Burroughs: B6700
Master Control Program Information Manual

[6] Burroughs: B6700
Program Binder Information Manual

[7] Control Data Corporation: CYBER 170 Model 175

[8] Control Data Corporation: CYBER 170
Hardware Reference Manual

[9] Control Data Corporation: CYBER 170
Loader Reference Manual

[10] Honeywell Bull General Electric:
GE-635 System Manual

[11] Honeywell Bull General Electric:
GE-645 System Manual

[12] IBM: Betriebssystem /360
Die Assemblersprache

[13] IBM: Betriebssystem /360
Linkage Editor und Loader

[14] IBM: Betriebssystem /360
Maschinenlogik und Aufbau der Instruktionen

[15] IBM: System /370
Der virtuelle Speicher - das moderne Konzept der Datenverarbeitung

[16] IBM: VM / 370 = CP + CMS
Systemsteuerprogramm für den Simultanbetrieb virtueller Maschinen im Dialog auf IBM Systemen /370

[17] IBM: System /360
Operating System Linkage Editor and Loader

[18] Kandzia, P.; Langmaack, H.: Informatik: Programmierung. Teubner Studienbücher Informatik

[19] McCarthy, J.; Corbato, F.J.; Daggett, M.M.:
The linking segment subprogram language and linking loader. Comm.ACM 6,7(July 1963), Seite 391-395

[20] Mohn, K.-H.; Rosendahl, M.; Zoller, H.:
AIDA, eine Dialogsprache für den TR44o. Arbeitsbericht des Rechenzentrums der Ruhr-Universität Bochum Nr. 7107

[21] Nudds: Conversation of programs between computers: Interpreters, Simulators, Compilers.
Journal Comp. System Sci. 7(1973) Seite 597-614

[22] Presser, White: Linkers and Loaders
Comp. Surveys 4(1972) Seite 149-167

[23] Rank Xerox: Universal Time-Sharing System (UT3)
Sigma 6/7/9 Computers System Management Reference Manual

[24] Rank Xerox: Universal Time-Sharing System (UT3)
Sigma 6/7/9 Computers Batch Processing Reference Manual

[25] Rank Xerox: Sigma 9 Computer Reference Manual

[26] Reynolds: Definitional interpreters for higher order programming languages.
ACM Annual Conf. 1972 Seite 717-740

[27] Telefunken: TR440 BCPL Sprachbeschreibung

[28] Telefunken: TR440 Befehlslexikon

[29] Telefunken: TR440 Die Assemblersprache TAS

[30] Telefunken: TR440 FORTRAN

[31] Telefunken: TR440 Große Befehlsliste

[32] Telefunken: TR440 Systemdienste BS3 Unterlagensammlung

[33] Univac: 1108 Multiprocessor System
Assembler Programmers Reference Manual

[34] Univac: 1108 Multiprocessor System
Executive Programmers Reference Manual

[35] Univac: 1108 Multiprocessor System
Executive Functions General Description

[36] Univac: 1108 Multiprocessor System
System Description

[37] Wilkes, M.V.: Time-Sharing-Betrieb bei digitalen Rechenanlagen. Carl Hanser Verlag. München 1970

Bisher erschienene Arbeitsberichte des Rechenzentrums

der Ruhr-Universität Bochum

Nr. 7101: K.-H. Mohn, M. Rosendahl, H. Zoller
AIDA; eine Dialogsprache für den TR 440 (vergriffen)

Nr. 7102: K.-H. Mohn, M. Rosendahl, H. Zoller
AIDA, ein Dialogsystem und seine Implementierung in ALGOL (vergriffen)

Nr. 7103: K.-H. Mohn, M. Rosendahl, H. Zoller
AIDA, Manual für den Benutzer (vergriffen)

Nr. 7104: 4. Jahresbericht des Rechenzentrums (Juni 1970 bis Juni 1971)

Nr. 7105: H. Wupper
WR MB02 - Ein einfaches Band-Betriebssystem für einen mittleren Rechner

Nr. 7201: H. Windauer
Existenzsätze zur (0,1,..,R-2,R) - Interpolation

Nr. 7202: W. Schelongowski
DIATRACE, Ein System zur interaktiven Assemblerprogrammierung

Nr. 7203: M.Jäger, M. Rosendahl, R. Staake
Einführung in die Listenverarbeitung anhand der Dialogsprache AIDA

Nr. 7204: R. Mannshardt, P. Pottinger
Einführung in die Benutzung des Teilnehmer-Rechensystems TR 440 in der RUB (vergriffen)

Nr. 7205: 5. Jahresbericht des Rechenzentrums (1.7.1971 bis 30.6.1972)

Nr. 7206: M. Rosendahl
BOGOL-TAS, ein Weg zur systemnahen Programmierung in ALGOL am TR 440

Nr. 7207: W. Stark
ILW, Programmsystem zur Berechnung des Instationären Ladungswechsels von Verbrennungskraftmaschinen (Modulbeschreibung und Eingabekonventionen)

Nr. 7208: W. Stark
ILW, Programmsystem zur Berechnung des Instationären Ladungswechsels von Verbrennungskraftmaschinen (Regelmechanismus und Berechnung der Rohrströmung)

Nr. 7209: H. Ehlich
Anregung und Kritik zum Betriebs- und Programmiersystem der TR 440

Nr. 7210: M. Rosendahl
BOGOL-STRING, eine flexible Zeichenkettenverarbeitung in ALGOL 60

Nr. 7211: H.Camici, H. Claus, H. Ehlich, D. Kipp
Arbeitsbericht über ein Programm zur Haushaltsführung