Systemprogrammiersprachen und strukturiertes Programmieren

Gerhard Goos, Universität Karlsruhe

Zusammenfassung

Systemprogrammiersprachen sind höhere Programmiersprachen, die hinsichtlich ihrer Datenstrukturen und Grundoperationen auf die speziellen Bedürfnisse des Systemprogrammierers eingerichtet sind. Nach einem Überblick über den gegenwärtigen Bestand an Eigenschaften solcher Sprachen wird auf einige in Entwicklung befindliche Probleme, insbesondere im Bereich der Strukturierung von Daten und der Schnittstellenbeschreibung eingegangen.

1. EINFÜHRUNG

> Complete generality of programming
> implies complete absence of structure.
> P. Brinch Hansen

Höhere Programmiersprachen wurden vor allem entwickelt, um dem Programmierer die Bürde des Arbeitens mit einer Maschinensprache, oder einer maschinenorientierten Assembliersprache abzunehmen. Stattdessen werden Sprachelemente zur Verfügung gestellt, welche die Formulierung von Algorithmen im Rahmen der Terminologie und der Denkweise des jeweiligen Anwendungsgebietes erleichtern sollen; wir sprechen daher auch von problemorientierten Programmiersprachen. Jede solche Programmiersprache verdeckt eine Reihe von Eigenschaften der zugrundeliegenden Maschine und erlaubt dafür neue, "höhere" Konstruktionen. Z.B. wird der Gebrauch von Sprungbefehlen durch bedingte Anweisungen und Schleifen weitgehend überflüssig; die lineare Speicherstruktur wird verdeckt durch Kellerstrukturen oder eine Halde wie in ALGOL 68. Manche der neuen Eigenschaften werden durch das Betriebssystem zur Verfügung gestellt wie z. B. Dateien anstelle externer Speichermedien. Sieht man einmal von

der Art der Implementierung einer Eigenschaft ab, die hier durch Software erfolgt, während die Eigenschaften der Grundmaschine durch Hardware oder wenigstens durch Mikroprogrammierung realisiert sind, so definiert die Programmiersprache eine neue "abstrakte Maschine". Diese abstrakten Maschinen haben sich übrigens bisher als wesentlich stabiler erwiesen als die Hardware: Für den FORTRAN-Programmierer hat sich die Rechenanlage seit den Zeiten der IBM 650 bei weitem nicht so einschneidend verändert wie für den Programmierer in Assemblersprache. Um so verwunderlicher ist die immer noch zu beobachtende Sorglosigkeit bei der Entwicklung neuer Programmiersprachen verglichen mit der Sorgfalt und dem Aufwand, den man der Entwicklung neuer Hardware angedeihen läßt.

Programmiersprachen dienen nicht nur dem technischen Bedürfnis, Problemlösungen dem Rechner mitzuteilen. Wichtiger noch ist, daß sie weitgehend die Denkgewohnheiten der Programmierer beeinflussen. Sie tun das bereits in einem Stadium, in dem der Programmierer noch gar nicht an die explizite Formulierung in einer Sprache denkt; Möglichkeiten, die sich später nicht oder nur mit Mühe in der Programmiersprache darstellen lassen, werden möglichst frühzeitig im Entwurfsprozeß aus dem Denkprozess eliminiert.

Systemprogrammieren ist nach J. Sammet [10] die Erstellung von *Programmiersystemen*; das sind Systeme, welche nicht unmittelbar Lösungen für Anwendungsprobleme liefern, sondern die Grundlage für solche Problemlösungen in weiten Anwendungsbereichen bilden. Kurz ausgedrückt ist Systemprogrammieren also die Implementierung von Schnittstellen, auf denen andere Programme aufbauen, oder wie wir es oben ausdrückten - die Implementierung abstrakter Maschinen.

Lange Zeit schien Systemprogrammieren nur mit Assemblersprachen denkbar, da nach allgemeiner Ansicht die Eigenschaften der Hardware, sei es die Speicherstruktur, seien es spezielle Befehle, so stark ausgenutzt werden mußten, daß nur eine Programmiersprache in Betracht zu kommen schien, bei der die zugehörige abstrakte Maschine und die reale Hardware übereinstimmte. Auch wurde behauptet, daß nur auf diese Weise die "optimale" Ausnutzung des beschränkten Hauptspeichers sichergestellt werden können.

Das Entstehen von Systemprogrammiersprachen, von höheren Programmiersprachen für die Systemprogrammierung, ist das Ergebnis der Beobachtung, daß der Systemprogrammierer in Wahrheit das Hilfsmittel Assemblersprache gar nicht voll ausnutzt. Er beschreibt beispielsweise bedingte Anweisungen und Schleifen durch immer wiederkehrende Befehlssequenzen, die man genauso gut mechanisch erzeugen könnte. Er entwirft sich in Form von Unterprogrammbibliotheken zusammengesetzte

Operationen, die zusammengenommen neue Datenstrukturen charakterisieren. Insgesamt erhält er damit Programmiersprachen, die er allerdings von Hand in Assembliersprache übersetzt und die außerdem nicht standardisiert sind, sondern für jedes Problem und von unzähligen Programmierern täglich neu entwickelt werden.

Die Überlegungen über systematische Programmentwicklung, über hierarchischen Programmaufbau und ähnliche Prinzipien, die man heute unter dem Begriff "Strukturiertes Programmieren" zusammenfaßt, haben dazu geführt, daß die Gemeinsamkeiten dieser adhoc-Sprachen immer stärker in Erscheinung getreten sind und dann ihren Niederschlag in sprachlichen Formulierungen gefunden haben, die sich nicht mehr im Rahmen von Assembliersprachen bewegen. Schwerwiegender noch ist die Erkenntnis, die sich aus dem schichtenweisen Aufbau von Systemen ziehen läßt, wie ihn erstmals Dijkstra [6] paradigmatisch vorführte. Dijkstra zeigte nämlich, daß der Systemprogrammierer nur in der untersten Schicht seines Programmaufbaus einer Sprache bedarf, die genau auf die Hardware zugeschnitten ist. Danach entfernt er sich auch innerhalb von Betriebssystemen und erst recht in Übersetzern von der Hardware. Es ist daher gerechtfertigt, Sprachen zu benutzen, die in Ablaufsteuerung und Datenstrukturen eine von der Hardware abweichende abstrakte Maschine verwenden und eher höheren Programmiersprachen gleichen. Wenn es dann noch gelingt - z.B. in Form offener Unterprogramme und unter Ausnutzung spezieller Implementierungseigenschaften-, die Programmierung spezieller Hardware-Funktionen zugänglich zu machen, so ist man dem Ziel einer Programmiersprache, welche Assembliersprachen ablösen kann und zudem in weiten Bereichen maschinenunabhängig ist, schon sehr nahe.

Die Entwicklung von Systemprogrammiersprachen begann mit der Programmiersprache NELIAC [7], einem ALGOL 58-Dialekt, dessen Übersetzer in der eigenen Sprache geschrieben war. Eine auch heute noch bedeutende, sehr frühe Systemprogrammiersprache ist ESPOL [2], von der Firma Burroughs ab 1963 für die Maschinen der B5000 und später der B6000-Serie entwickelt. ESPOL enthält ALGOL 60 als Teilsprache und hat schon in frühen Zeiten, wenn auch unreflektiert, viele Entwicklungsprinzipien aufgezeigt, die auch heute noch von allgemeinem Interesse sind. Die allgemeine Entwicklung begann dann mit PL360 [12]. Schrittweise wurde zunächst die Ablaufsteuerung realer Maschinen, dann die lineare Speicheradressierung [14,3] und schließlich die Binärcodierung für Datenobjekte [3] durch entsprechende Konstruktionen höherer Programmiersprachen ersetzt (vgl. Abb. 1). Gegenwärtige Bemühungen zielen darauf ab, das noch unterentwickelte

Gebiet der Datenstrukturen in den Griff zu bekommen und die Maßnahmen zum Schutz gegen unbefugten Zugriff auf Datenobjekte und Operationen zu verbessern. Letzteres hängt mit der allgemeineren Aufgabe zusammen, das Zusammensetzen von Programmen aus Einzelmoduln modellmäßig zu erfassen, mit dem Ziel zu Schnittstellenbeschreibungen zu kommen, die vom Übersetzer auch auf semantische Konsistenz geprüft werden können, soweit das mit den Mitteln einer Programmiersprache überhaupt möglich ist.

2. SYSTEMPROGRAMMIERSPRACHEN HEUTE

Bevor wir uns der möglichen zukünftigen Entwicklung von Systemprogrammiersprachen zuwenden, ist es nützlich, den Bestand an wesentlichen Spracheigenschaften zu betrachten, auf dem diese Entwicklung aufbauen kann. Aufgrund der einleitenden Bemerkungen ist es nicht weiter verwunderlich, daß dies zugleich eine Bestandsaufnahme wünschenswerter Eigenschaften höherer Programmiersprachen darstellt. Dabei ist die Schreibweise der einzelnen Elemente von untergeordneter Bedeutung; barocke Wucherungen wie zum Beispiel unterschiedlichste Schreibweisen für die verschiedenen Formen der Wiederholungsanweisung sind natürlich negativ zu bewerten, treten aber häufig auf.

2.1 Programmaufbau und Ablaufsteuerung

Weithin akzeptierte Grundlage des Aufbaus von Programmen ist heute die Gliederung in Blöcke und Prozeduren. Diese Blockstruktur dient inhaltlich unterschiedlichen Zielen, aus denen für die Programmkonstruktion eine Reihe von Nebenbedingungen erwachsen (vgl. Abb. 2). Zum Beispiel kann eine Prozedur, welche nur der abkürzenden Schreibweise für eine Folge von Anweisungen dient, beliebige Seiteneffekte auf ihre Umgebung haben. Neue Grundoperationen können Seiteneffekte auf die Programmschicht haben, in der sie definiert sind, wie man am Beispiel von Speicherzuteilungs- und Freigabe- Prozeduren und den von ihnen verursachten Pegeländerungen sieht; hingegen dürfen sie keine Nebenwirkungen auf dem Abstraktionsniveau des Aufrufers solcher Operationen haben. Selbständige Programmoduln schließlich sollten nur über explizit definierte Parameter und das eventuelle Funktionsergebnis Veränderungen nach außen bewirken, um maximale Unabhängigkeit von der Umgebung zu erreichen.

Für die Steuerung des sequentiellen Ablaufs eines Programms benötigen wir Hilfsmittel (Abb.3) zur Darstellung der sequentiellen Folge, der zeitlichen Unabhängigkeit (Kollateralität), der Auswahl aus mehreren Alternativen, der aufzählenden und der iterierenden Schleife, des Aufrufs einer Prozedur, sowie für den Abgang aus einer zusammengesetzten Anweisung (Abschnitt, Block, Proze-

dur), wenn die Zielbedingung erreicht ist. Trotz aller gegenteiligen Argumente muß darüberhinaus dem Systemprogrammierer der Sprungbefehl erhalten bleiben, damit ein gleitender Übergang von der Ebene der Hardwareprogrammierung zu höheren Sprachelementen möglich ist. Dabei sind bei den in Abbildung 3 zusammengestellten Konstruktionen natürlich noch Vereinfachungen möglich; A bedeutet entweder eine einzelne Anweisung oder einen "Abschnitt" (eine Gruppe) aus mehreren Anweisungen, denen Vereinbarungen vorangehen können.

2.2 Datenarten

Einfache Datenobjekte können charakterisiert werden durch ihren Umfang (Anzahl von Bits). Eine solche *typfreie* Kennzeichnung erlaubt zusammen mit der Adresse den Zugriff auf das Objekt; für weitergehende Operationen muß man jedoch wissen, nach welcher Codierungsvorschrift die Objekte zu interpretieren sind. Abgesehen davon, daß es lästig ist, ständig wissen zu müssen, daß 0, 1, 2, 3, 4, 5 in dieser Reihenfolge etwa die Farben rot, gelb, grün, blau, violett, purpur bedeuten und durch 3 Bit codiert werden können, wird hierdurch Fehlinterpretationen Tür und Tor geöffnet.

Die Kennzeichnung der einfachen Objekte durch Datentypen, die nicht nur den Umfang, sondern auch die Codierung und die zulässigen Operationen charakterisieren, greift daher auch bei Systemprogrammiersprachen um sich (Abb. 4). Man sollte dabei sorgfältig unterscheiden zwischen der Umfangsangabe, die zugleich die Verarbeitungsbreite bei Operationen wiedergibt, und Ausschnittsangaben, welche besagen, daß ein Objekt mit einem geringeren Umfang gespeichert werden kann, auch wenn es beim Zugriff sofort auf die Verarbeitungsbreite (z.B. Registerlänge) verlängert wird.

Allerdings genügen die Möglichkeiten, wie sie durch Datentypen geboten werden, nicht für alle Aufgaben der Systemprogrammierung. Schwierigkeiten bereiten vor allem Algorithmen zur Speicherverwaltung, welche beispielsweise Keller oder Halden auf den linearen Speicher abbilden und demselben Speicherbereich wechselnde Decodierungsvorschriften, also unterschiedliche Datentypen, zuordnen. Der gegenwärtig häufigste Ausweg besteht darin, einen Datentyp word einzuführen, welcher zusätzlich die Möglichkeiten typfreier Sprachen eröffnet. Damit öffnet man natürlich die Büchse der Pandora wieder, die man durch Einführung der Datentypen gerade geschlossen hatte. Es ist daher sinnvoll, syntaktisch "unsichere" Programmoduln einzuführen und nur in diesen den wortweisen Zugriff zu erlauben. Neben einer Erhöhung der Zuverlässigkeit erzwingt ein solches Vorgehen größere Klarheit über die Objekte, mit denen man umgeht.

Bei *zusammengesetzten Datenobjekten* unterscheiden wir ein- oder mehrstufige Reihungen, bei denen die Elemente durch berechenbare Indizes selektiert werden, und Verbunde, bei denen die Elemente durch fest vorgegebene Bezeichner angesprochen werden. Im Bereich der Systemprogrammierung lohnt es sich, bei Reihungen solche mit Deskriptor und ohne Deskriptor begrifflich zu unterscheiden; letztere haben einen zur Übersetzungszeit berechenbaren Umfang. PASCAL bietet zusätzlich Mengen als Objekte sowie sequentielle, dafür aber dem Umfang nach unlimitierte, Strukturen an, die speziell als Modell für Daten auf Hintergrundspeichern gedacht sind. Schließlich kann man in dieser Sprache die Elemente zusammengesetzter Objekte "packen", um Speicher zu sparen. Diese Möglichkeit bildet zusammen mit Mengenobjekten die wesentliche Voraussetzung dafür, daß man keine Formulierung für den bitweisen Zugriff in den Speicher benötigt. Zur Bildung komplizierterer Strukturen steht gegenwärtig nur noch der Referenzbegriff, also die Nachbildung von Adressen, zur Verfügung, so daß wir zur Zusammenstellung in Abb. 5 gelangen.

Die Entwicklung der Beschreibungshilfen für Datenstrukturen ist von einer bemerkenswerten Unlogik gekennzeichnet. Bei schrittweiser Verfeinerung des Programmentwurfs ergeben sich Datenstrukturen bereits in den allerersten Schritten. Trotzdem hat man zur Modellbildung mit Reihungen, Verbunden und Referenzen in Programmiersprachen nur Begriffe zur Verfügung, welche gemeinhin das physische Nebeneinander der Elemente oder die Verweisstruktur wiedergeben, also maschinen- und implementierungsorientierte Begriffe, welche auf diesem Niveau noch gar keine Rolle spielen sollten und eliminiert werden müßten. (Die Relationenmodelle und andere Beschreibungshilfen, die für den Entwurf von Datenbanken entwickelt wurden, zeigen eine mögliche Alternative). Andererseits hat man bisher zur globalen Organisation der Speicherverteilung außer der Keller- und der Haldenorganisation keine Hilfsmittel zur Verfügung. Alles Weitere muß mithilfe von Reihungen simuliert werden, obwohl dieses Hilfsmittel für diesen Zweck wiederum zu hoch gegriffen ist. (Der bisher einzige weitergehende Versuch, nämlich die Speicherorganisation des AED-Systems (Ross [9]) hat sich zwar bewährt, hat aber keine Nachfolger gefunden.)

Im Zusammenhang mit Geflechten aus zusammengesetzten Objekten, etwa bei Listenstrukturen, ist schließlich noch die Frage zu lösen, wie man zu einer einheitlichen Artangabe für Verbunde mit Komponenten unterschiedlicher Art gelangt (vgl. Abb.6). Geht man davon aus, daß jeder Verbund bzw. jede Verbundvariable nur eine fixierte Komponentenart erlaubt, während verschiedene Verbunde der gleichen Art unterschiedliche Komponentenart haben können, so gelangt man zur Methode der Verbundvarianten, wie sie PASCAL benutzt. ALGOL 68 [11] erlaubt mit seiner Vereinigung von Arten, daß sich die Komponenten-

art während der Lebensdauer der Variablen ändert. In beiden Fällen ist die Menge der Arten, denen die Komponenten angehören können, a priori bekannt. Anders ist dies in SIMULA [4]. Hier kann eine Klasse A (vergleichbar einer Verbundart) als Präfix der Vereinbarung einer oder mehrerer Klassen B benutzt werden. Dies bewirkt die Aufnahme der sämtlichen Komponenten von A in die Klasse B. Umgekehrt kann sich dann ein Verweis auf Klassen A auch auf Klassen der Art B beziehen. Bei der Definition der Klasse A müssen die Klassen B noch nicht einmal ihrer Anzahl nach bekannt sein. Wir können das Verfahren zum Beispiel dazu benutzen, um zunächst die sämtlichen Verbundkomponenten zu definieren, welche für eine spezielle Speicherverwaltungsstrategie allen Verbunden gemeinsam sein sollen. Dies kann in einer niedrigeren Programmschicht geschehen als die Definition der restlichen Verbundkomponenten, die etwa in wechselnden Benutzerprogrammen erfolgt. Weder in PASCAL noch in ALGOL 68 ist ein derartiges Vorgehen möglich.

3. PROGRAMM- UND DATENMODULN

Unter (*Programm-*) *Moduln* verstehen wir Programmstücke mit der Eigenschaft, daß das Zusammensetzen solcher Moduln zu größeren Einheiten keine Kenntnis des inneren Arbeitens der einzelnen Moduln verlangt und die Korrektheit der einzelnen Moduln ohne Kenntnis der Einbettung in das Gesamtprogramm nachprüfbar ist. Diese Charakterisierung (nach Dennis [5]) sagt aus, daß sowohl der interne Aufbau des Moduls als auch seine externe Verwendung lediglich von einer genau zu definierenden *Schnittstelle* abhängen soll. Die entstehenden größeren Einheiten sollten wieder als Moduln aufgefaßt werden können, der Modulbegriff sollte also rekursiv sein.

In den üblichen Programmiersprachen werden zur Wiedergabe von Programmoduln lediglich Prozeduren als sprachliches Hilfsmittel zugelassen. Diese sind in vielen Fällen nicht ausreichend. Allgemeinere Moduln werden benötigt,

- wenn modulspezifische Datenbestände vor dem ersten Aufruf initialisiert oder zumindest zwischen zwei Aufrufen aufbewahrt werden müssen,

- wenn mehrere Prozeduren auf gemeinsamen Datenbeständen arbeiten und daher mit diesen zusammen einen umfassenderen Modul bilden,

- wenn mehrere Prozeduren auf gemeinsamen Hilfsfunktionen aufbauen, die zum Programmodul gezählt werden müssen.

Beispiele hierfür sind etwa die Kollektion von Prozeduren, welche zusammen die verschiedenen Zugriffsfunktionen auf eine Datei oder irgendeine andere Datenstruktur realisieren. Mir sind keine Beispiele modularen Aufbaus von Programmen bekannt, bei denen sich die Moduln nicht in dieser Form charakterisieren lassen.

Ein solcher allgemeiner Programmodul hat (n+1) Eingänge, die sämtlich Prozeduraufrufe darstellten. Der erste Aufruf führt zum *Modulaufbau*; er definiert und initialisiert die lokalen Daten des Moduls. Die weiteren Aufrufe *aktivieren* den Modul oder genauer ausgedrückt jeweils eine der Prozeduren des Moduls. Die Dauer dieser Aktivierungen ist zu unterscheiden von der Lebensdauer des gesamten Moduls. Letzere beginnt mit dem Modulaufbau und überdeckt alle möglichen Aktivierungen.

Zur sprachlichen Formulierung eignet sich eine leichte Verallgemeinerung der Klassen aus SIMULA [4] (vgl. Abb. 7). Abbildung 8 zeigt die Definition eines Kellers für ganze Zahlen.

Auch die Untersuchung weiterer Beispiele zeigt, daß *Datenmoduln*, die Beschreibung der Implementierung einer Datenstruktur zu den häufigsten Anwendungen von Programmoduln zählen. Zugleich ergeben sich damit Möglichkeiten die Speicherorganisation mit solchen Programmoduln zu erledigen - Hilfsmittel wäre dann eine lokale Reihung von Worten - und sich damit von den Beschränkungen frei zu machen, welche wir im letzten Abschntit kritisierten.

Implementierungstechnisch lassen sich Moduln ohne weiteres in die übliche kellerartige Speicherorganisation eingliedern, sofern man verabredet, daß die Lebensdauer eines Moduls am Ende des Blocks endet, in dem der Modul aufgebaut wurde. Um den häufig nicht unbeträchtlichen organisatorischen Aufwand beim Prozeduraufruf zu vermeiden, ist es zweckmäßig, bei den von außen zugänglichen Prozeduren wahlweise auch offenen Einbau zuzulassen. Diese Maßnahme bewährt sich vor allem bei der Realisierung der Zugriffsfunktionen auf Datenstrukturen durch solche Prozeduren.

Weist man jedem Modul und den von ihm aufgerufenen Prozeduren ein eigenes Kellersegment zu, so lassen sich damit unabhängige *sequentielle Prozesse* oder *Koroutinen* realisieren. Letzere lassen sich übrigens auch in die normale Kellerorganisation einbetten, sofern man die Abschnitte der Koroutinen als Prozeduren formuliert wie aus Abbildung 9 ersichtlich.

Eine andere Erweiterung des Konzepts bilden die von Brinch Hansen und Hoare eingeführten *Monitore* (vgl. [1]). Hier kann zu jedem Zeitpunkt höchstens ein Modulaufruf aktiviert sein. Treten in einem Prozeßsystem während dieser Aktivierung weitere Aufrufe des Moduls auf, so werden sie in eine oder mehrere modulspezifische Warteschlangen eingereiht und bis zur Beendigung der laufenden Aktivierung verzögert.

4. SCHNITTSTELLENBESCHREIBUNGEN

Die Beschäftigung mit kleinen Beispielprogrammen, wie sie beim Lehren des Programmierens üblich und notwendig sind, hat bisher vielfach den Blick versperrt für solche Eigenschaften von Programmiersprachen, deren Notwendigkeit erst bei sehr großen Programmen und beim Arbeiten im Team sichtbar wird. Dies gilt nicht nur für Programmiersprachenentwicklungen im akademischen Bereich.

Von A. Perlis stammt die Bemerkung, er habe noch nie ein ALGOL-Programm lauffen sehen, sondern stets nur ein Programmsystem, in dem das ALGOL-Programm einen Teilmodul bildete. Trotzdem halten die Programmiersprachen an der Fiktion des "Hauptprogramms" fest. Diesem kann man zwar getrennt übersetzte Prozeduren beifügen, aber das Verfahren ist meist einstufig wie in FORTRAN und erlaubt keineswegs die baumartigen Beziehungen beim Zusammenbau von Teilmoduln zu größeren Moduln wiederzugeben. Für Systemprogrammiersprachen scheint es angebracht, nur (getrennt übersetzbare) Prozeduren und allgemeinere Programmoduln zu unterscheiden. Daneben gibt es noch ein "Strukturprogramm", welches den Zusammenbau und die Initialisierung des Modulsystems beschreibt. Strukturprogramme zeichnen sich dadurch aus, daß sie die Lücken aufzeigen, in welche die anderen Moduln eingesetzt werden sollen. Da sich insgesamt wieder ein Modul ergeben soll, sind Strukturprogramme nur begrifflich, nicht aber syntaktisch von sonstigen Moduln unterscheidbar (vgl. Abb.10).

Wie in Abbildung 8 bereits angedeutet definiert jeder Modul die Prozeduren, Untermoduln, Artvereinbarungen und Objekte, welche von außen zugänglich, also *öffentlich* sind. Umgekehrt muß er eine Beschreibung aller externen Größen enthalten, welche in diesem Modul benutzt, aber nicht definiert werden. Technisch ist diese Beschreibung externer Größen eine Vorbesetzung der Übersetzertabellen für die Übersetzung des Moduls, gibt also auch Auskunft über die Art von Objekten usw. und erschöpft sich nicht in der bloßen Auflistung von Bezeichnern wie in Assemblierspachen üblich.

Während diese Techniken in einigen wenigen Sprachen schon üblich sind, fehlt

es in fast allen Sprachen am nächsten Schritt, nämlich der Überwachung der genauen Einhaltung der Schnittstellendefinition mithilfe des Übersetzers bzw. des Binders. Hierzu ist es notwendig die Angaben über externe Größen in Modul A zu vergleichen mit den Originalvereinbarungen (als öffentliche Größen) in Modul B,C,D... Eine Untersuchung in einer Softwareabteilung ([8]) zeigt, daß man mit einer solchen Kontrolle, die insbesondere auch die Anzahl und Arten von Prozedurparametern erfaßt, ein Drittel des Zeitaufwandes für die Suche nach Laufzeitfehlern bereits zur Übersetzungszeit abfangen kann.

Eine Möglichkeit, den nicht unbeträchtlichen Aufwand beim gegenseitigen Kontrollieren der Schnittstellenbeschreibungen zu reduzieren, besteht darin, die gesamte Beschreibung der externen und öffentlichen Größen der Moduln im Strukturprogramm zu wiederholen oder - falls das Strukturprogramm als vor den Teilmoduln übersetzt angenommen werden kann - sie überhaupt ins Strukturprogramm zu verlegen. Die Prüfung der Konsistenz erfolgt dann in mehreren Schritten. Einerseits ist die Konsistenz der einzelnen Schnittstellenbeschreibungen innerhalb des Strukturprogramms zu überprüfen, andererseits muß die Übereinstimmung der zusammengehörigen Beschreibungen im Strukturprogramm und im Untermodul geprüft werden.

5. ZUGRIFFSBESCHRÄNKUNGEN

Die erwähnte Wiederholung der Schnittstellenbeschreibung im Strukturprogramm erlaubt dem Schreiber des Strukturprogramms nun auch weitergehende Eingriffe in die Möglichkeiten der Kommunikation verschiedener Teilmoduln. Durch das Aufführen oder Nichtaufführen einer Größe als externe Größe in der Schnittstellenbeschreibung eines Teilmoduls B wird bestimmt, auf welche Teilmoduln sich der Gültigkeitsbereich dieser in einem Teilmodul A als öffentlich definierten Größe erstreckt. Unsere frühere Charakterisierung der Programmoduln macht verständlich, warum diese Festlegung vom Strukturprogramm und nicht vom Teilmodul A ausgeht.

Ein derartiges Verfahren erlaubt eine genaue Kontrolle der Zugriffsrechte der einzelnen Moduln und codifiziert daher im Programm Absprachen von Programmierern, welche sonst nur mündlich oder in der Programmdokumentation festgehalten werden können. Ein Vergleich mit ähnlichen Sicherheitsmaßnahmen in Dateisystemen zeigt jedoch, daß es noch Möglichkeiten zur weiteren Differenzierung gibt, deren Übertragung auf Programmiersprachen nützlich erscheint. Als Minimum sollte man zwischen der Erlaubnis zum Lesen eines Objekts und der Erlaubnis zum Verändern des Objekts (Zuweisung nach eventuellem Lesen) unterschei-

den. Weitere Möglichkeiten, auf deren Realisierung wir hier nicht eingehen, wären etwa die Erlaubnis in eine verkettete Liste oder eine ähnliche Struktur zu schreiben ohne jedoch den Aufbau des Geflechts zu ändern, oder die Erlaubnis, ein Geflecht zu vergrößern, es aber nicht zu verkleinern, usw.

Wir betrachten das Problem anhand der Parameterübergabe für Prozeduren. Technisch läßt sich diese auf die beiden Fälle der Übergabe einer Adresse des aktuellen Parameters und der Übergabe einer Kopie des Wertes des als aktueller Parameter auftretenden Objekts reduzieren. Üblicherweise werden diese beiden Möglichkeiten gleichgesetzt mit der Erlaubnis, den aktuellen Parameter zu lesen und Zuweisungen an ihm vorzunehmen (Schreiberlaubnis) bzw. den Wert nur zu lesen (Leseerlaubnis). Ihren extremen Ausdruck findet diese Auffassung im Referenzkonzept von ALGOL 68. Dabei wird übersehen, daß auch Konstante eine Adresse haben und daß insbesondere bei zusammengesetzten Objekten das Einsparen des Kopierens durch die Übergabe einer Adresse ohne Schreiberlaubnis einen erheblichen Fortschritt darstellen würde. Interessanter noch wäre die Möglichkeit, die Adresse einer Variablen zu übergeben, ohne damit gleichzeitig das Schreiben zu erlauben. Zukünftige Systemprogrammiersprachen sollten nicht nur bei Prozedurparametern, sondern in allen Arten von Schnittstellenbeschreibungen die Unterscheidung zwischen Adresse und Wert eines Objekts sorgfältig von der Unterscheidung zwischen Schreib- und Leseerlaubnis trennen.

Literatur

[1] Brinch Hansen, P., *"Operating System Principles"*. Prentice-Hall, Englewood Cliffs, N.J., 1973.

[2] Burroughs, "ESPOL Language, Information Manual" Burroughs Corp., Detroit, Form 5000094

[3] Clark, B.L. and Horning, J.J., "The System Language for Project SUE". SIGPLAN Notices vol. 6 no. 9 (1971).

[4] Dahl, O.J., Myhrhaug, B. and Nygaard U., "SIMULA 67, Common Base Language". Norwegian Computer Center, Oslo, 1967.

[5] Dennis, J.B., "Modularity". In: Bauer, F.L. (ed.), Advanced Course on Software Engineering. Lecture Notes in Economics and Mathematical Systems 81. Springer, Berlin-Heidelberg-New York 1973.

[6] Dijkstra, E.W., "The Structure of the "THE" Multiprogramming System". Comm. ACM 11, 341-346 (1968).

[7] Halstead, M.H., *"Machine-Independent Programming"*. Spartan Books, Washington, D.C., 1962.

[8] Klunder, J., "Experiences with SPL". Working Paper, Conference on Maschine-Oriented High Level Languages. Trondheim 1973.

[9] Ross, D.T, "The AED Free Storage Package". Comm. ACM 10, 481-492 (1967)

[10] Sammet, J.E., "A Brief Survey of Languages Used in Systems Implementation". SIGPLAN Notices, Volume 6, Number 9, (1971)

[11] Wijngaarden, A. v. (ed.), "Report on the Algorithmic Language ALGOL 68. Num. Math. 14, 79-218 (1969).

[12] Wirth, N., "PL360", A Programming Language for the 360 Computers". Journal ACM 15, 37-74 (1968).

[13] Wirth, N., "The Programming Language PASCAL (Revised Report)". ETH Zürich, Berichte der Fachgruppe Computer-Wissenschaften Nr. 5, 1972.

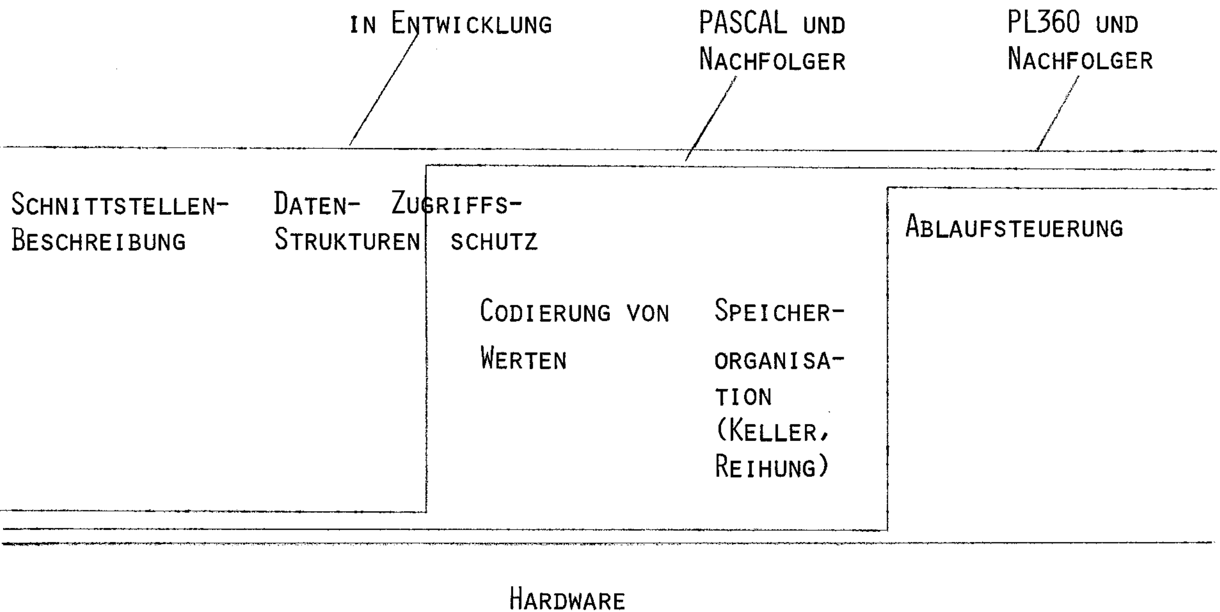


ABB. 1 ABLÖSUNG DER SYSTEMPROGRAMMIERSPRACHEN (SPS) VON DER HARDWARE

BLOCKSTRUKTUR

BLÖCKE DIENEN

- DER SYNTAKTISCHEN ZUSAMMENFASSUNG
- DER KENNZEICHNUNG ZUSAMMENGESETZTER OPERATIONEN
- DER KENNZEICHNUNG NEUER PROGRAMMSCHICHTEN, WELCHE AUF DIE HILFSMITTEL DER UMFASSENDEN, UNTERLIEGENDEN SCHICHT AUFBAUEN

BEGIN **END**

BEGIN **END**

BLÖCKE SIND ÄQUIVALENT ZU PARAMETERLOSEN PROZEDUREN.

PROZEDUREN

PROZEDUREN DIENEN

- DER ZUSAMMENFASSUNG HÄUFIG WIEDERKEHRENDER ANWEISUNGEN (EFFIZIENZFRAGE)
- ALS NEUE GRUNDOPERATIONEN
- ALS EIGENSTÄNDIGE PROGRAMMODULN (KEINE SEITENEFFEKTE)

ABB. 2 HILFSMITTEL ZUM PROGRAMMAUFBAU

SEQUENZ:	A_1 ; A_2
KOLLATERALITÄT:	A_1 , A_2
AUSWAHL:	IF B THEN A_1 ELSE A_2 FI
	CASE FORMEL OF M_1 : A_1,
	.
	.
	.
	M_N : A_N
	OUT A_0 ENDCASE
	(D.H. IF(FORMEL)=M_1 THEN A_1 ELSE...)
AUFZÄHLUNG:	FOR ZÄHLER LAUFLISTE, LAUFLISTE,..., DO A DONE
	FROM F_1 BY F_2 TO F_3
ITERATION:	WHILE B DO A DONE
	DO A DONE
	DO A DONE UNTIL B
PROZEDURAUFRUF:	P
	P(AP_1,..., AP_N)
ABGANG:	M:BEGIN...EXIT M WITH ERGEBNIS...END
SPRUNG:	GOTO M

ABB. 3 SEQUENTIELLE ABLAUFSTEUERUNG

TYPEN:

<u>INT</u>	GANZE ZAHLEN	MIT IMPLIZITER OBERSCHRANKE
<u>REAL</u>	GLEITPUNKTZAHLEN	BZW. GENAUIGKEITSGRENZE

ENDLICHE MENGEN:

<u>BOOL</u> = (<u>FALSE</u>,<u>TRUE</u>)
<u>WOCHENTAG</u> = (SONNTAG, MONTAG, ..., SAMSTAG)
<u>CHAR</u>

AUSSCHNITTE:

<u>INT</u> (0 : 13)
<u>WERKTAG</u> = <u>WOCHENTAG</u> (MONTAG:SAMSTAG)

ABB. 4 DATENTYPEN

ZUSAMMENGESETZTE OBJEKTE

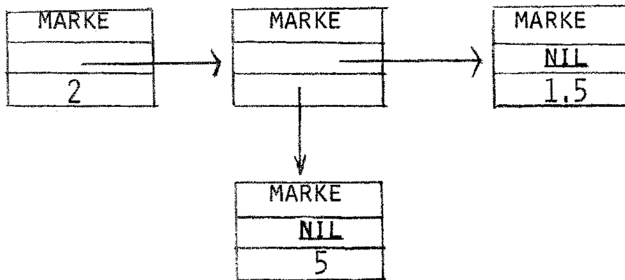
```
ROW [1:100] INT A                                   A[I]
ARRAY [1:N, L:M] BOOL B                             B[J,*] , B[J,K]
STRUCT (REAL REALTEIL, IMAGINAERTEIL) C             C.REALTEIL
STRUCT (PACKED(INT(1:100)JAHR,
               INT(1:12) MONAT,
               INT(1:31)TAG),
        STRUCT(PACKED(INT(0:23)STUNDE,
                      INT(0:59)MINUTE))
                              UHRZEIT)D             D.UHRZEIT.STUNDE


SET WOCHENTAG ARBEITSTAGE          IF MONTAG IN ARBEITSTAGE THEN.....
FILE CHAR TEXT
```

ZUR BILDUNG VON GEFLECHTEN:

```
REF DATENART
z.B.
   LISTELEMENT = STRUCT (REAL INHALT, REF LISTELEMENT NEXT)
```

ABB. 5 DATENSTRUKTUREN

PROBLEM: DARSTELLUNG DER ELEMENTE DER LISTE




```
PASCAL: LISTELEMENT = RECORD  MARKE : BOOL; ZEIGER :↑LISTELEMENT;
                              CASE KENNZEICHEN : (ZAHL, UNTERLISTE) OF
                              ZAHL :(X : REAL);
                              UNTERLISTE :(X :↑ LISTELEMENT)
                      END
```

```
ALGOL 68:  MODE LISTELEMENT=STRUCT (BOOL MARKE, REF LISTELEMENT ZEIGER,
                                    UNION(REAL,REF LISTELEMENT)X)
```

```
SIMULA:    CLASS A; BEGIN BOOLEAN MARKE; REF(A) ZEIGER; END;
         A CLASS LISTELEMENT1; BEGIN REAL X; END;
         A CLASS LISTELEMENT2; BEGIN REF (A) C; END

         REF(A) AA; AA:-NEW LISTELEMENT1;
              (AA QUA LISTELEMENT1).X := 5;
           INSPECT AA WHEN LISTELEMENT1 DO X:=X+1
```

ABB. 6   VERBUNDE MIT KOMPONENTEN VARIABLER ART

<u>MODULE</u> Modulbezeichner

(formale Parameter für den Modulaufbau):

<u>BEGIN</u>

Vereinbarungen	lokaler Daten
	lokaler Prozeduren und Untermoduln
	von aussen zugänglichen Prozeduren und Untermoduln;

Anweisungen zur Initialisierung

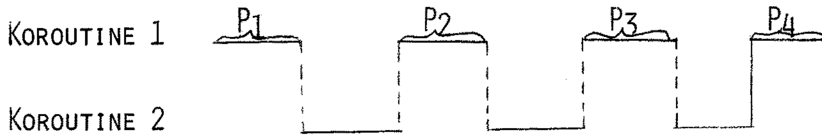
<u>END</u>

Abb. 7 Allgemeine Programmoduln

```
MODULE  STACK
        (INT MAXIMALE TIEFE, PROC ÜBERLAUF, PROC INT KELLER LEER):
  BEGIN
        ARRAY[1:MAXIMALE TIEFE] INT K;
        INT TIEFE;
        PUBLIC PROC PUSH   (INT A):
             IF TIEFE = MAXIMALE TIEFE THEN ÜBERLAUF
             ELSE TIEFE := TIEFE+1; K [TIEFE] := A
             FI;
        PUBLIC PROC POP :
             IF TIEFE = 0 THEN KELLER LEER ELSE TIEFE := TIEFE -1 FI;
        PUBLIC PROC VAL   INT:
             IF TIEFE = 0 THEN KELLER LEER ELSE K [TIEFE] FI;
        PUBLIC PROC DEPTH   INT:
             TIEFE;
  INITIALISIERE:
        TIEFE := 0
  END

  STACK (10, ÜBERLAUF, KELLERLEER) S;
  S.PUSH (17)
```

ABB. 8   EIN MODUL ZUR KELLERORGANISATION

ZEITLICHER ABLAUF:




```
MODULE KOROUTINE 1 :
   BEGIN
      PUBLIC PROC KOROUTINENEINGANG (*EINE PROZEDURVARIABLE*);
      VEREINBARUNG LOKALER DATEN;
      PROC P1:  BEGIN .... KOROUTINENEINGANG := P2 END;
      PROC P2:  BEGIN .... KOROUTINENEINGANG := P3 END;
            .
            .
            .
INITIALISIERE:
      KOROUTINENEINGANG := P1
   END
```

ABB. 9 REALISIERUNG VON KOROUTINEN

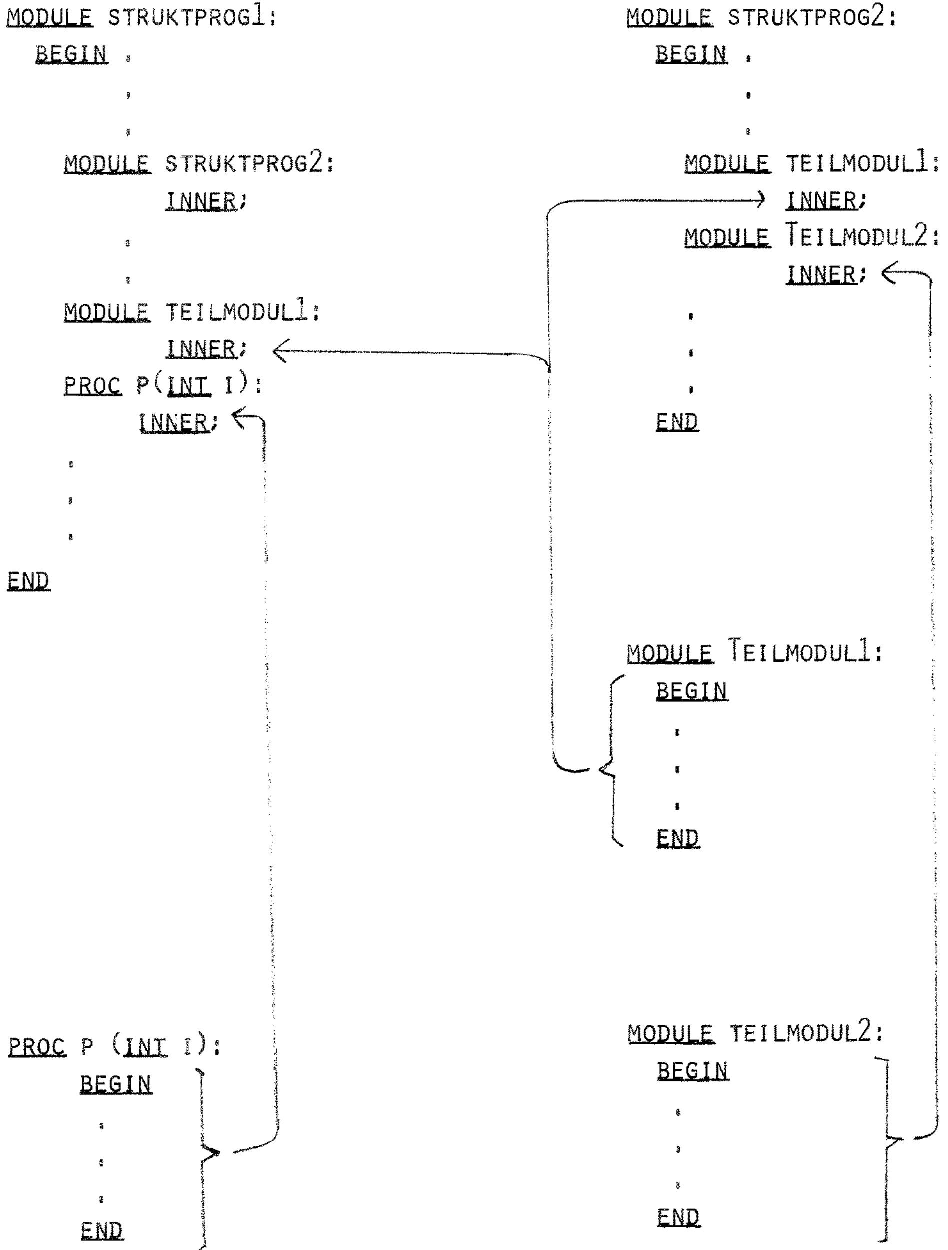


ABB. 10 AUFBAU VON PROGRAMMEN AUS EINZELNEN MODULN