EINE KONFIGURATIONSSPRACHE FÜR ADA

J. F. H. Winkler
Siemens AG München

KURZFASSUNG

Bei der praktischen Verwendung von Ada und ähnlichen Programmiersprachen in industriellen Projekten entstehen als Produkte in der Regel Programmfamilien mit Revisionen und Varianten. Derzeit enthalten Programmiersprachen keine Sprachelemente zur Beschreibung von solchen Revisionen und Varianten. Der Aufsatz beschreibt eine Methode, mit der die strukturellen Zusammenhänge in einer Familie von Programmen mit Revisionen und Varianten beschrieben werden können. Es wird gezeigt, wie die Methode mit dem Ada-Sprachmittel "Pragma" realisiert werden kann, ohne eine Spracherweiterung vorzunehmen.

ABSTRACT

The use of Ada and similar programming languages in industrial projects usually results in products that are program families with revisions and variants. For the time being programming languages do not contain language constructs for the description of such revisions and variants. This paper presents a method that can be used to describe the structural interconnections in a family of programs with revisions and variants. We show how this method can be realized by the use of Ada pragmas in such a way that no extensions of the language are necessary.

1 EINFÜHRUNG

Rechner und Programme, die Rechner steuern, werden infolge des technischen Fortschritts größer und größer. Ein Beispiel für solch ein großes Programm ist das Programm des EWSD (Elektronisches Wählsystem Digital), welches in Vermittlungstellen wie Ortsämtern oder Fernämtern des öffentlichen Fernsprechbereiches eingesetzt wird [Sie 81; BE 82a]. Dieses Programm ist in CHILL geschrieben und umfaßt etwa 500 000 Zeilen reinen Codes, aufgeteilt in etwa 1500 Module.

Solch große Programme können nicht mehr als monolithische Programme z.B. im Sinne von Pascal realisiert werden, da sie als Ganzes praktisch unhandhabbar sind. Eine Reihe von modernen Programmiersprachen wie z.B. Ada [Ref 83], CHILL [CCI

84], Mesa [MMS 79], Modula [Wir 80] und Pascal-XT [SS 85] enthalten daher Sprachmittel zur Modularisierung von Programmen, d.h. zur Gliederung von Programmen in eine Menge von textuell selbständigen Programmbausteinen. Bei den oben genannten Sprachen wird dann dieselbe semantische Konsistenz zwischen diesen Bausteinen gewährleistet wie innerhalb eines Bausteines selbst [Win 82]. Ältere Sprachen wie z.B. Fortran, Cobol und PL/I erlauben zwar auch die Gliederung des Programmes in verschiedene, textuell selbständige Bausteine, jedoch ist in der Regel keine semantische Konsistenz zwischen diesen Bausteinen garantiert. Dies liegt weniger an diesen Sprachen selbst sondern daran, daß diese Sprachen traditionellerweise nach dem Schema der unabhängigen Übersetzung implementiert wurden.

Große Programme als industrielle Produkte sind in der Regel nicht genau ein Programm im Sinne der Sprachdefinition, sondern stellen eine Familie verwandter Programme [Par 75] dar. Am Beispiel des EWSD zeigt sich das daran, daß es einerseits eine Reihe von funktionellen Varianten des EWSD-Programmes gibt, und andererseits jede Variante innerhalb der Lebensdauer eine Reihe von Revisionen durchläuft. Die Varianten ergeben sich z.B. dadurch, daß das EWSD-System sowohl für unterschiedliche Vermittlungsaufgaben (Ortsamt, Fernamt usw.) als auch in verschiedenen Ländern mit unterschiedlichen Vermittlungssystemen eingesetzt wird. Die Lebensdauer solcher Systeme liegt üblicherweise in der Größenordnung von Jahrzehnten, was zu einer ganzen Reihe von Revisionen führt.

Die oben erwähnten modernen Programmiersprachen bieten keine sprachliche Unterstützung zur Darstellung der strukturellen Zusammenhänge in solch einer Programmfamilie. In Ada ist es zwar möglich den Rumpf (Implementierungsteil) eines Bausteins auszutauschen, ohne den Rest des Programmes erneut übersetzen zu müssen; es ist aber nicht möglich auszudrücken, daß unter verschiedenen Bedingungen unterschiedliche Rümpfe in das Programm eingebaut werden sollen.

Eine Sprache, welche strukturelle Beziehungen in einer Familie mit Revisionen und Varianten auszudrücken gestattet, so daß die Elemente der Familie aus der gesamten Bausteinmenge der Familie heraus konfiguriert werden können, soll im folgenden als Konfigurationssprache bezeichnet werden. Eine solche Konfigurationssprache ist eng verwandt mit den "Module Interconnection Languages" (MIL) [Ber 84; DK 76; Nar 85].

In der vorliegenden Arbeit wird eine Konfigurationssprache vorgestellt und gezeigt, wie sie sich in Ada, ohne eine Spracherweiterung vornehmen zu müssen, realisieren läßt. In Abschnitt 2 wird das der Sprache zugrunde liegende Modell dargestellt. Abschnitt 3 zeigt die Realisierung mittels Ada-Pragmata, und Abschnitt 4 enthält einige Beispiele.

2 MODELL DER KONFIGURATIONSSPRACHE

2.1 PROGRAMMBAUSTEINE

Unter einem Programmbaustein wird im folgenden im wesentlichen das Gleiche verstanden wie unter einer Übersetzungseinheit (compilation unit) in Ada: ein textuell selbständiges Gebilde, welches einer Gruppe von ausgezeichneten Konstrukten der betreffenden Programmiersprache angehört. Solche ausgezeichneten Konstrukte sind z.B. Unterprogramm, Paket, Modul.

Ein Baustein hat einen Namen, der auch ein gegliederter Name sein kann; wie z.B. der Name einer Untereinheit in Ada. Jeder Baustein hat genau einen Namen, jedoch können mehrere Bausteine denselben Namen haben. Solche Bausteine müssen sich dann aber in der Version unterscheiden. Darauf wird weiter unten eingegangen.

Die Beziehungen zwischen verschiedenen Bausteinen, die ein Programm bilden sollen, kommen im Text dieser Bausteine dadurch zum Ausdruck, daß ein Baustein auf andere Bausteine Bezug nehmen kann. Ein Programm kann dann so definiert werden, daß es aus einem bestimmten Baustein B und allen Bausteinen besteht, auf die B direkt oder indirekt Bezug nimmt. In Ada ist B das Hauptprogramm.

Bausteine sind organisatorisch zu Bibliotheken zusammengefaßt. Diese Bibliotheken ergeben jeweils einen Kontext, in welchem Bausteinnamen und -versionen interpretiert werden. Im folgenden wird aus Gründen der Einfachheit angenommen, daß die Bausteine einer Programmfamilie in einer Bibliothek enthalten sind. Der Grund dafür ist, daß in der vorliegenden Arbeit das Problem der Interbibliotheksreferenzen nicht behandelt wird.

2.2 REVISIONEN

Wie bereits in der Einleitung erwähnt sind die Revisionen eine meist zeitlich geordnete Folge von Ausprägungen eines Bausteins. Meist ist damit die Intention verbunden, daß eine spätere Revision frühere ablösen soll. In der Praxis kommt aber häufig der Fall vor, daß eine Reihe (meist aufeinander folgender) Revisionen im Einsatz sind. Diesem Umstand wird im vorliegenden Modell dadurch Rechnung getragen, daß es zu einem Baustein generell mehrere Revisionen geben kann.

Die Revisionen werden beschrieben durch eine Menge REV von Revisionsindikatoren:

$$REV = \{r1, \ldots, rn\}$$

REV ist im einfachsten Falle totalgeordnet; z.B. im Falle von Zeitmarken oder Revisionsnummern, wie sie für industrielle Produkte typischerweise verwendet werden.

Werden hierarchisch gegliederte Benummerungen verwendet wie z.B. in [Tic 82], dann kann REV auch partiell geordnet sein. In [Tic 82] handelt es sich um eine spezielle Halbordnung, nämlich um eine Ordnung mit Baumstruktur. Dies bedeutet insbesondere, daß dabei nicht vorgesehen ist, verschiedene Revisionen wieder zusammenzuführen. Bei einer allgemeinen Halbordnung wäre auch ein solches Zusammenführen möglich.

Ein konkreter Baustein kann auch zu mehreren Revisionen gehören. Dies ist z.B. dann erforderlich, wenn Programmversionen homogen in den Revisionsindikationen sein sollen [MTW 84]. Um diesem Fall auch Rechnung zu tragen, wird eine Revisionsangabe als eine Teilmenge von REV definiert. Damit ist die Menge der Revisionsangaben RevA:

$$RevA := Pot(REV)$$

wobei "Pot" der Potenzmengenoperator ist.

2.3 VARIANTEN

Varianten eines Bausteins sind Alternativen (z.B. in konstruktiver oder funktioneller Hinsicht), die zu einem Zeitpunkt gleichberechtigt nebeneinander stehen [SW 78]. Die Varianten werden charakterisiert durch die Werte, die bestimmte Merkmale annehmen.

Sei MN die Menge der Merkmalnamen:

$$MN := \{MN1, \ldots, MNm\}$$

Die Menge der Merkmalausprägungen sei MA:

$$MA := \{MA1, \ldots, MAp\}$$

Eine Variantenangabe ordnet jedem Merkmalnamen eine (möglicherweise leere) Teilmenge der Merkmalausprägungen zu. Die Menge der Variantenangaben ist damit:

$$VarA := MN \rightarrow Pot(MA)$$

Dabei bedeuten $A \rightarrow B$ die Menge der partiellen Abbildungen von A in B.

Die Menge der Varianten über MN und MA ist die Menge VAR:

$$VAR := MN \rightarrow MA$$

Jede Variantenangabe bestimmt eine Menge von Varianten. Die Variantenmenge einer Variantenangabe v ist:

$$VarM(v) := \{f \in VAR \mid \forall a \in dom(f): f(a) \in v(a)\}$$

Man könnte die Menge der Ausprägungen gliedern in namensspezifische Teilmengen, wie es z.B. in [SW 78] gemacht wurde, wo Ausprägungen als Werte von Datentypen (z.B. Zahlen, Aufzählungswerte, Zeichenketten) realisiert wurden. Diese Gliederung der Ausprägungsmenge ist für das hier vorgestellte Modell einer Konfigurationssprache nicht essentiell. Um die Darstellung in dieser Arbeit einfach zu halten, wird auf die Gliederung der Ausprägungsmenge verzichtet.

2.4 VERSIONEN EINES BAUSTEINS

Eine einzelne Version ist durch die Angabe einer Revisionsindikation und einer Variante charakterisiert. Daher ist die Menge der Versionen wie folgt definiert:

$$VERS := REV \times VAR$$

Eine Versionsangabe beschreibt eine Menge einzelner Versionen. Entsprechend ist die Menge der Versionsangaben VerA wie folgt definiert:

$$VerA := REV \rightarrow VarA$$

2.5 VERSIONEN EXTERNER BAUSTEINE

Der Zusammenhang zwischen den Programmbausteinen eines Programmes erfolgt, wie bereits oben erwähnt, durch Externreferenzen. Programmiersprachen fordern hierbei in der Regel die Angabe des Namens des Programmbausteins, auf den Bezug genommen werden soll. In der Konfigurationssprache soll es darüberhinaus möglich sein, gezielt eine bestimmte Version des externen Bausteins in der Externreferenz anzugeben. Die Version des externen Bausteins sollte dabei von der

aktuellen Version des referierenden Bausteins abhängen können. Diese Abhängigkeit kann auf unterschiedliche Art und Weise ausgedrückt werden. Sei (r,a) die Version des referierenden Bausteins und (re, ae) die Version des referierten, externen Bausteins. Folgende Auswahlmöglichkeiten bestehen:

- re hängt von r ab;
- ae hängt von r ab;
- ae hängt von a ab;
- (re, ae) hängt von r ab;
- (re, ae) hängt von (r,a) ab.

2.6 BIBLIOTHEKEN

Ein Bibliothekselement ist charakterisiert durch seinen Namen und durch seine Versionsangabe. Die Menge der Bibliothekselemente BE ist daher:

$$BE := N \times VerA$$

wobei N die Menge der Namen ist.

Jedes Bibliothekselement bestimmt eine Menge von Bausteinversionen. Wenn b = (n,v) ein Bibliothekselement ist, dann ist die Menge der Bausteinversionen von b:

$$BV(b) := \{(n,r,a) \mid r \in dom(v) \wedge a \in VarM(v(r))\}$$

Eine Bibliothek ist eine Menge B von Bibliothekselementen, welche die folgende Bibliotheksbedingung erfüllt:

$$BIB(B) := (\forall a,b \in B: a \neq b \Rightarrow BV(a) \cap BV(b) = \emptyset)$$

d.h. eine Bausteinversion identifiziert eindeutig ein Bibliothekselement. In Kapitel 3 gilt für ein einzelnes Bibliothekselement, welches in einer Bibliothek enthalten ist, stets $BV(b) \neq \emptyset$.

2.7 KONFIGURIERUNG

Bei der Konfigurierung wird aus einer oder mehreren Bibliotheken ein Programm aus Bausteinen zusammengefügt. Dieser Montageprozeß wird gestartet mit der Angabe einer Bausteinversion für den Baustein, der das Hauptprogramm (z.B. i.S. von Ada) sein soll. Der Rest des Programmes wird dann automatisch aufgrund der Externreferenzen konfiguriert. Die transitive Hülle der vom Hauptprogramm ausgehenden Externreferenzen bestimmt die Menge der zum Programm gehörenden Bausteinversionen.

3 REALISIERUNG IN ADA

Die Realisierung einer Konfigurationssprache, wie sie im vorangehenden Abschnitt beschrieben wurde, kann auf zwei Arten erfolgen:

- es wird eine eigene Konfigurationssprache definiert. In dieser Sprache wird dann auf die Programmbausteine selbst durch Verweise Bezug genommen. Oft sind diese Verweise sogar nur Namen von Dateien, welche die Programmbausteine enthalten. Dadurch sind Name und Inhalt in der Regel völlig entkoppelt, so daß Konsistenzprüfungen erst beim Konfigurieren selbst durchgeführt werden können [Win 85]. Ein solcher Ansatz wurde in der Vergangenheit verschiedentlich verfolgt [DK 76; Fel 79; HP 80; Nar 85]. Häufig gingen diese Ansätze von klassischen Programmiersprachen aus, die keine Sprachelemente für das Programmieren im Großen enthalten (wie z.B. Pascal, C). Außerdem definierten solche Konfigurationssprachen oft auch Sprachelemente für die Formulierung des Exports und Imports von Größen. Diese Größen wurden meistens als "resources" bezeichnet. Die modernen Programmiersprachen enthalten nun ihrerseits auch Sprachelemente für den Export und Import von Größen. Will man nun eine klassische Konfigurationssprache mit einer modernen Programmiersprache kombinieren, dann ergibt sich das Problem, ob die zwei Gruppen von Sprachelementen für den Export und den Import von Größen miteinander verträglich sind.

- hinzufügen von Sprachelementen zur Realisierung der Konfigurationssprache zu einer existierenden Programmiersprache. Dieser Weg liegt nahe bei Programmiersprachen, die bereits Sprachelemente für das Programmieren im Großen enthalten. Dies ist der Fall bei modernen Programmiersprachen wie Ada, CHILL, Mesa, Modula und Pascal-XT. Dadurch entsteht eine Sprache, die sowohl Elemente für das Programmieren im Kleinen als auch für das im Großen enthält, also eine Art Breitbandsprache.

Im folgenden wird der zweite Weg verfolgt und als Basissprache Ada gewählt. Hierbei bestehen nun wieder zwei verschiedene Möglichkeiten:

- echte Spracherweiterung, d.h. Realisierung der Konfigurationssprache mit Sprachkonstruktionen, die in Ada nicht vorhanden sind und über seinen syntaktischen Rahmen hinausgehen;

- Realisierung der Funktionen der Konfigurationssprache als Pragmata. Dies ist keine echte Spracherweiterung, da eine Implementierung von Ada zusätzliche Pragmata definieren darf.

Im folgenden wird der zweite Weg gewählt, um keine echte Spracherweiterung vornehmen zu müssen. Bekanntermaßen dürfen ja von Ada keine Unter- oder Obermengen gebildet und mit dem Namen "Ada" bezeichnet werden.

Zu diesem Zweck werden zwei Gruppen von Pragmata definiert:

- Pragmata zur Definition der Versionen eines Bausteins, und
- Pragmata zur Auswahl der Version eines externen Bausteins.

3.1 DEFINITIONSPRAGMATA

Es werden drei Definitionspragmata (D-Pragmata) definiert. Diese definieren eine resultierende Menge VRes von Bausteinversionen eines Bausteines mit Namen "n". Die drei D-Pragmata sind:

pragma DEF_REV (R1,...,Rk);

mit der Bedeutung r = {R1,...,Rk}, wobei angenommen wird, daß die Liste R1,...,Rk keine Wiederholungen enthält. Die Liste R1,...,Rk enthält mindestens ein Element. Dies gilt entsprechend auch für die anderen in den folgenden Pragmata vorkommenden Listen. Die Ri sind Revisionsindikationen. Ist für ein Bibliothekselement kein Pragma DEF_REV angegeben, dann ist r = ∅. Die Menge der Revisionsindikatoren wird durch die Implementierung oder jeweils für ein Projekt festgelegt.

pragma DEF_VAR (M1:(A11,...,A1n1), M2:(A21,...,A2n2),...);

mit der Bedeutung a = {(M1, {A11,...,A1n1}), (M2, {A21,...,A2n2}),...}. Die einzelnen Listen von Ausprägungen (z.B. A11,...,A1n1) sollen keine Wiederholungen enthalten. Ebenfalls soll kein Mi mehrfach vorkommen. Die Aij sind Merkmalausprägungen. Ist für ein Bibliothekselement kein Pragma "DEF_VAR" angegeben, dann ist a = ∅. Die Menge der Merkmalausprägungen wird durch die Implementierung oder auch für einzelne Projekte festgelegt. Wird statt einer Ausprägungsliste das Schlüsselwort **all** angegeben, dann ist dies gleichbedeutend mit der Angabe aller Ausprägungen.

pragma DEF_VAR(**all**);

ist gleichbedeutend mit der Angabe von **all** für alle Merkmalnamen.

pragma DEF_VERS (R1:(var-angabe), R2:(var-angabe), ...);
hat die Bedeutung v = {(R1, {var-angabe}), (R2, {var-angabe}),...). Ist für ein Bibliothekselement kein Pragama DEF_VERS angegeben, dann ist v = ∅, d.h. v ist ein Element aus REV >→ VarA. Außerdem soll kein Ri mehrfach vorkommen. "var-angabe" ist analog zu Pragma DEF_VAR zu interpretieren. Die Angabe von **all** ist ebenfalls wie bei DEF_VAR möglich.

Die drei Pragmata können auch nebeneinander angegeben werden, jedoch jedes für ein Bibliothekselement höchstens einmal. Die Menge der Bausteinversionen eines Bibliothekselementes mit Namen "n" ist nun durch die drei Mengen r, a und v wie folgt bestimmt. Falls a ∪ v ≠ ∅, dann ist:

$$VRes := \{ (n,p,q) \mid q \in VarM(v(p)) \vee (p,q) \in r \times VarM(a) \}$$

Falls keine Variantenangabe gemacht ist, dann gehört das Bibliothekselement zu allen Varianten, d.h. wenn a ∪ v = ∅, dann ist die Menge der Bausteinversionen:

$$VRes := \{(n,p,q) \mid (p,q) \in r \times VAR \}$$

Wenn VRes leer ist, dann ist das betreffende Bibliothekselement fehlerhaft und wird nicht in eine Bibliothek aufgenommen. Außerdem muß die in Abschnitt 2.6 erwähnte Bibliotheksbedingung erfüllt sein, um das betreffende Bibliothekselement in eine Bibliothek aufzunehmen.

Die D-Pragmata stehen unmittelbar in einem Baustein oder folgen unmittelbar auf den Baustein, wenn der Baustein nur aus einer UP-Spezifikation oder einer Exemplarbildung (Instantiierung) besteht.

3.2 AUSWAHLPRAGMATA

Die Auswahlpragmata sind Teil der Externreferenzen und definieren die jeweils auszuwählende Version eines referierten Bausteins.
Der Auswahlprozeß geht stets von der Version des referierenden Bausteins aus. Für das Hauptprogramm wird diese Angabe bei der Initiierung einer Programmkonfigurierung von außen vorgegeben. Im folgenden sei (r,a) die aktuelle Version des referierenden Bausteins. (re,ae) bezeichnet im folgenden die ausgewählte Version des externen Bausteins.

Jeder Auswahlschritt muß genau ein Bibliothekselement auswählen. Es ist ein Fehler, wenn kein Bibliothekselement mit der Version (re,ae) und dem betreffenden Namen in der Bibliothek existiert.

pragma SEL_REV(R);
wobei R eine Revisionsindikation ist. Es gilt: (re,ae) = (R,a) Das bedeutet, daß eine feste Revision des externen Bausteins gefordert wird.

pragma SEL_REV (R1 ⇒ R1', R2 ⇒ R2', ...);
wobei das Argument ein Element f aus REV >→ REV ist. Es gilt:

(re,ae) = IF r ∈ dom(f) THEN (f(r),a) ELSE (r,a) FI

pragma SEL_VAR(A);
wobei A eine Variante aus VAR ist. Es gilt : (re,ae) = (r,A), d.h. es wird eine bestimmte Variante ausgewählt.

pragma SEL_VAR (A1 ⇒ A1', A2 ⇒ A2', ...);
wobei das Argument ein Element f aus VAR >→ VAR ist. Es gilt

(re,ae) = IF a ∈ dom(f) THEN (r,f(a)) ELSE (r,a) FI

pragma SEL_VERS(R,A);
wobei (R,A) ∈ REV x VAR. Es gilt dann (re,ae) = (R,A), d.h. es wird eine bestimmte Version ausgewählt.

pragma SEL_VERS (V1 ⇒ V1', V2 ⇒ V2', ...);
wobei das Argument ein Element f aus VERS >→ VERS ist. Es gilt

(re,ae) = IF (r,a) ∈ dom(f) THEN f((r,a)) ELSE (r,a) FI

Die Auswahlpragmata stehen stets unmittelbar nach der Externreferenz, zu der sie gehören. Im folgenden wird angenommen, daß höchstens ein Auswahlpragma zu einer Externreferenz angegeben wird. Man könnte auch die Angabe mehrerer zulassen; zB ein SEL_REV und ein SEL_VAR. Darauf soll aber hier nicht weiter eingegangen werden. Ist für eine Externreferenz kein Auswahlpragma angegeben, dann gilt (re,ae) = (r,a). Ein Auswahlpragma, welches sich auf keine Externreferenz bezieht, bezieht sich auf den Rumpf der Spezifikation, zu welcher das Auswahlpragma gehört.

Ein Baustein kann auch in mehreren Versionen in das zu konfigurierende Programm eingebaut werden, jedoch dürfen sich die Gültigkeitsbereiche dieser verschiedenen Versionen nicht überlappen.

4 BEISPIELE

Das erste Beispiel zeigt die Anwendung von DEF_VERS und von SEL_VAR. Das Hauptprogramm MAIN ist ein Bibliothekselement mit den zwei Versionen (1:(Speed:(High))) und (1:(Speed:(Low))). In MAIN wird auf eine Untereinheit SORT Bezug genommen, wobei SEL__VAR verwendet wird. Wenn MAIN die Variante (Speed:High) hat, dann wird die Variante (Kind:Quick_Sort) von SORT ausgewählt, wenn MAIN die Variante (Speed:Low) hat, dann wird (Kind:Bubble__Sort) ausgewählt.

Durch die Zuordnungsmöglichkeiten in SEL__VAR ist eine Umbenennung der Merkmale an der Schnittstelle zwischen Bausteinen möglich. Diese Möglichkeit ist insbesondere dann von großer Bedeutung, wenn bereits vorhandene Bausteine eingesetzt werden sollen, bei deren Erstellung kein Einfluß auf die Namenswahl bestand. Ähnliche Umbenennungen bietet Ada auch für Programmgrößen.

```
procedure MAIN is
   pragma DEF_VERS (1:(Speed:(High,Low)));
   -- ...
   procedure SORT( ... ) is separate;
      pragma SEL_VAR ((Speed:High ⇒ Kind:Quick_Sort),
                      (Speed:Low  ⇒ Kind:Bubble_Sort));
   -- ...
end MAIN;

            separate(MAIN)
procedure SORT( ... ) is
   pragma DEF_VERS (1:(Kind:(Quick_Sort)));
   -- ...
end SORT;

            separate(MAIN)
procedure SORT( ... ) is
   pragma DEF_VERS (1:(Kind:(Bubble_Sort)));
   -- ...
end SORT;
```

Beispiel 1

Das zweite Beispiel zeigt, daß sich Programmversionen auch im funktionellen Umfang unterscheiden können. Das Hauptprogramm MAIN liegt in zwei Bibliothekselementen vor, die entsprechend unterschiedliche Versionen haben. Beide Versionen benutzen das Paket STACK__PACK, das ebenfalls in zwei

Bibliothekselementen mit entsprechenden Versionen vorliegt. Die schnelle Version von STACK_PACK benutzt eine Implementierung mittels einer Reihung und bietet nur die Operationen PUSH, POP und TOP an. Die allgemeine Version (Kind:General) von STACK_PACK benutzt eine Implementierung als verkettete Liste und bietet zusätzlich die Operation LENGTH an. Diese Operation kann dann auch in der entsprechenden Version von MAIN benutzt werden.

```
        with STACK_PACK; use STACK_PACK;
procedure MAIN is
    pragma DEF_VERS (1:(Kind:(Fast)));
    -- ...
end  MAIN;

        with STACK_PACK; use STACK_PACK;
procedure MAIN is
    pragma DEF_VERS (1:(Kind:(General)));
    -- ...
    ... LENGTH(...) ...
    -- ...
end  MAIN;

package STACK_PACK is
    pragma DEF_VERS (1:(Kind:(Fast)));
    type  STACK is limited private;
            -- array implementation
    procedure PUSH ... ;
    procedure POP ... ;
    function TOP ...;
    -- ...
end  STACK_PACK ;

package STACK_PACK is
    pragma DEF_VERS (1:(Kind:(General)));
    type  STACK is limited private;
            -- list implementation
    procedure PUSH ... ;
    procedure POP ... ;
    function TOP ...;
    function LENGTH ...;
    -- ...
end  STACK_PACK ;
```

Beispiel 2

Das dritte Beispiel zeigt eine etwas größere Programmfamilie, die in grafischer Form in [MTW 84] dargestellt ist. Die Familie besteht aus zwei Revisionen: 1 und 2. Die Revision 1 hat drei Varianten und die Revision 2 zwei Varianten.

```
      with B, C;
procedure A is
   pragma DEF_VERS
      (1:(Var:(1,2)), 2:(Var:(1)));
   --...
end A;
```

```
package B is
   pragma DEF_VERS
    (1:(Var:(1)), 2:(Var:(1,2)));
   -- ...
end B;
```

```
   with B;
      pragma SEL_VAR (Var:(2));
   with D;
procedure A is
   pragma DEF_VERS (1:(Var:(3)));
   --...
end A;
```

```
package B is
   pragma DEF_VERS
            (1:(Var:(2)));
   --...
end B;
```

```
      with B, C, E;
procedure A is
   pragma DEF_VERS (2:(Var:(2)));
   --...
end A;
```

```
package C is
   pragma DEF_REV(1);
   --...
end C;
```

```
package D is
   pragma DEF_REV(1);
   --...
end D;
```

```
package C is
   pragma DEF_REV(2);
   --...
end C;
```

```
package E is
   pragma DEF_REV(2);
   --...
end E;
```

Beispiel 3

5 ZUSAMMENFASSUNG

Die vorliegende Arbeit stellt eine Konfigurationssprache vor, mit welcher die strukturellen Zusammenhänge in Familien modularer Programme mit Revisionen und Varianten beschrieben werden können. Die Sprache erlaubt einerseits die Beschreibung der Versionen, zu denen ein bestimmter Programmbaustein gehört, und andererseits, welche Versionen die Bausteine haben müssen, die von einem Baustein aus referiert werden. Die Version des referierten Bausteins kann dabei von der des referierenden abhängig gemacht werden.

Am Beispiel von Ada wird gezeigt, wie die Konfigurationssprache sich in eine bestehende Programmiersprache integrieren läßt. Diese Integration ist ein wesentlicher Unterschied des hier vorgestellten Ansatzes im Vergleich zu bisherigen Konfigurationssprachen, die meist als selbständige Sprachen definiert wurden [DK 76; HP 80; Nar 85].

Die wesentlichen Eigenschaften des hier vorgestellten Ansatzes sind:

- Konfigurationssprache als "natürliche" Erweiterung einer Programmiersprache;
- Beschreibung von Versionen als Paare von Revisionen und Varianten;
- das "Wissen" über die Programmkonfigurationen ist explizit in Form von Fakten und Ableitungsregeln im Programmtext enthalten;
- die Wissensdarstellung ist an die Struktur des Programmes angepaßt;
- die Konstruktion einer bestimmten Programmversion ist automatisch möglich.

DANKSAGUNG

Der Autor dankt T. Mehner für wertvolle Hinweise zu einer früheren Fassung des Aufsatzes.

6 LITERATUR

BE 82a Botsch, Dietrich; Eberding, Hans
EWSD, A Real-Time Communication System with High-Level Language Software
IEEE COM 30,6(1982)1337..1342.

Ber 84 Berry, Daniel M.
On the Use of Ada as a Module Interconnection Language
Proc. Seventeenth Ann. Hawaii International Conference on System Sciences, 1984, 294..302.

DK 76 DeRemer, F.L.; Kron, H.H.
Programming-in-the-large versus Programming-in-the-small
= [NS 76: 80..89].

Fel 79 Feldman, Stuart I.
Make - A Program for Maintaining Computer Programs
Software - Practice & Experience 9(1979)255..265.

HP 80 Habermann, A. Nico; Perry, Dewayne E.
System Composition and Version Control for Ada
Dept. of Computer Science, Carnegie Mellon Univ., May 1980.

MMS 79 Mitchell, James G.; Maybury, William; Sweet, Richard
Mesa Language Manual
Xerox, Palo Alto Research Center, Palo Alto 1979; Version 5.0, CSL-79-3.

MR 85 Morgenbrod, H.; Remmele, W. (Hrsg.)
Entwurf großer Software-Systeme
B.G. Teubner, Stuttgart 1985.

MTW 84 Mehner; T.; Tobiasch, R.; Winkler, J.F.H.
A Proposal for an Integrated Programming Environment for CHILL
Third CHILL Conference, Cambridge 1984-09-23..28, p.65..71.

Nar 85 Narayanaswamy, K.
NuMIL - A Language to describe Software System Architecture
Comp. Science Dept., Univ. of Southern California, Los Angeles, USC TR 85-328.

NS 76 Schneider, H.-J.; Nagl,M. (eds.)
Programmiersprachen - 4. Fachtagung der GI
Springer, Berlin etc. 1976.

Par 75 Parnas, D.L.
On the design and development of program families
TH Darmstadt, FB Informatik, Bericht BS I 75/2, 2.7.75.

Ref 83 Reference Manual for the Ada Programming Language
ANSI/MIL-STD 1815A
United States Dept. of Defense, Washington D.C., Jan. 1983.

Sie 81 Siemens AG
EWSD Digital Switching System
telcom report, Vol.4 (1981) Special Issue.

SS 85 Sommer, M.; Stadel, M.
Erfahrungen beim Implementieren eines modernen Compilers
=[MR 85: 449..455].

SW 78 Winkler, J.F.H.; Stoffel, C.
Methode zur Betriebssystemgenerierung und -modularisierung für Prozeßrechensysteme
Kernforschungszentrum Karlsruhe KfK - PDV 153, Februar 1978.

Tic 82 Tichy, Walter F.
Adabase: A Data Base for Ada Programs
AdaTEC Conference 1982 p.57..65; ACM, New York 1982.

Win 82 Winkler, J.F.H.
Ada: die neuen Konzepte
Elektron. Rechenanlagen 24,4(1982)175..186.

Win 85 Winkler, J.F.H.
Language Constructs and Library Support for Families of Large Ada Programs
Workshop on Software Engineering Environments for Programming-in-the-Large, Cape Cod, June 1985.

Wir 80 Wirth, Niklaus
MODULA-2
Eidgenössische Technische Hochschule Zürich, Institut für Informatik. Bericht Nr. 36, March 1980.

J.F.H. Winkler
Siemens AG
Otto-Hahn-Ring 6
D-8000 München 83

Berichte des German Chapter of the ACM

Im Auftrag des German Chapter
of the ACM herausgegeben durch den Vorstand

Band 26

Die Reihe dient der schnellen und weiten Verbreitung neuer, für die Praxis relevanter Entwicklungen in der Informatik. Hierbei sollen alle Gebiete der Informatik sowie ihre Anwendungen angemessen berücksichtigt werden.

Bevorzugt werden in dieser Reihe die Tagungsberichte der vom German Chapter allein oder gemeinsam mit anderen Gesellschaften veranstalteten Tagungen veröffentlicht. Darüber hinaus sollen wichtige Forschungs- und Übersichtsberichte in diese Reihe aufgenommen werden.

Aktualität und Qualität sind entscheidend für die Veröffentlichung. Die Herausgeber nehmen Manuskripte in deutscher und englischer Sprache entgegen.

Software-Architektur und modulare Programmierung

Herausgegeben von

Prof. Dr.-Ing. Hans-Wilm Wippermann
Universität Kaiserslautern

B. G. Teubner Stuttgart 1986

CIP-Kurztitelaufnahme der Deutschen Bibliothek

Software-Architektur und modulare Programmierung :
(am 24. u. 25.02.1986 in Kaiserslautern)
hrsg. von Hans-Wilm Wippermann.
Stuttgart : Teubner, 1986.
(Tagung ... des German Chapter of the ACM ; 1986, 1)
(Berichte des German Chapter of the ACM ; Bd. 26)
ISSN 0724-9764
ISBN 3-519-02445-4

NE: Wippermann, Hans-Wilm (Hrsg.); Association for Computing Machinery / German Chapter : Tagung ... des ...; Association for Computing Machinery / German Chapter: Berichte des German...

Printed in Germany
Gesamtherstellung: J. Illig Offsetdruck, Göppingen
Umschlaggestaltung: M. Koch, Reutlingen