Konsistenzprüfung beim Integrieren von Basic PEARL Moduln

K. Lucas, München

Zusammenfassung

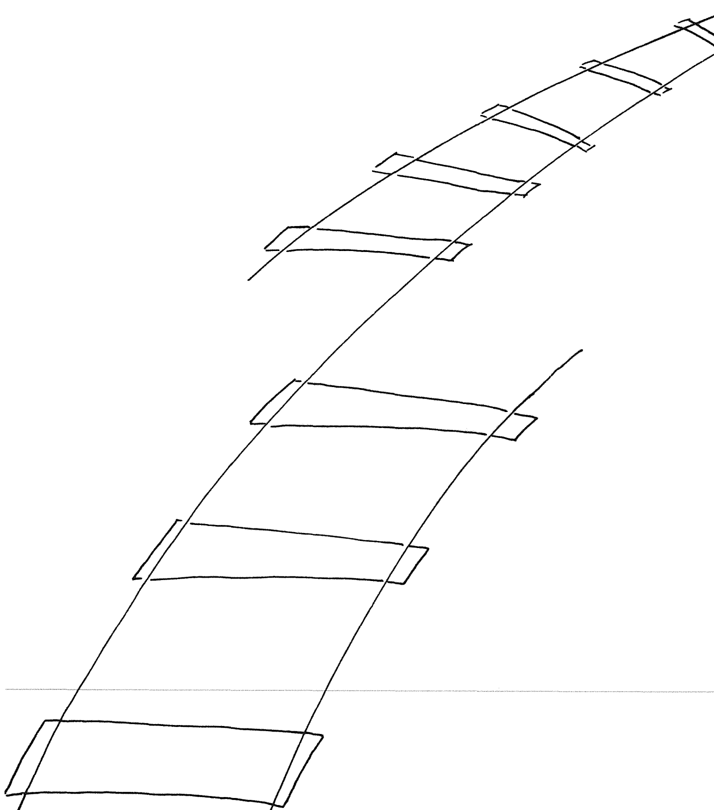
Es wird dargestellt, welchen Stellenwert die Integration in einem Übersetzungssystem für (Basic) PEARL einnimmt. Beim Zusammenstellen zu einem Programm müssen alle Module einer Prüfung unterzogen werden, um die konsistente Verwendung ihrer gegenseitigen Schnittstellen zu garantieren. Diesbezüglich unterschiedliche Vorgehensweisen in PEARL und Ada werden einander gegenübergestellt. Zuletzt wird die Implementierung eines portablen Konsistenzprüfprogrammes vorgestellt.

Summary

The importance of the integration phase in a Basic PEARL compilation system is described. During the linking phase of a program, the interfaces of all modules should be checked to guarantee consistent use of the global definitions.
The approaches taken in PEARL and Ada are compared. Finally, the implementation of a portable consistency checking program is presented.

Einleitung

Was nützt es dem PEARL-Anwender, wenn er glaubt, alle seine Module richtig programmiert zu haben und er erst zur Laufzeit feststellt, daß sie nicht aufeinander abgestimmt sind?



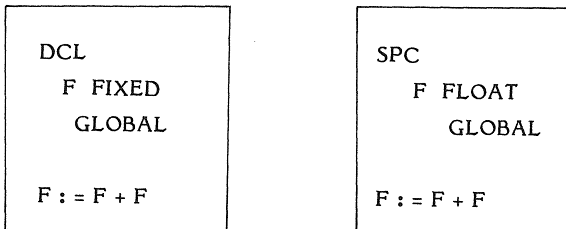
Die Konsistenzprüfung ist ein Beitrag zum Thema "PEARL in der Praxis". Sie widmet sich einer Phase in der Projektabwicklung, der i.a. wenig Beachtung geschenkt wird: der Integration. Die Erfahrung zeigt jedoch, daß die Integration einzelner Module zu einem funktionsfähigen Modulverband - dem Programm - häufig genug ein aufwendiges Unterfangen darstellt, vor allem dann, wenn das Zusammenspiel der Module von Phänomenen der oben dargestellten Art begleitet ist.

Behandelt werden in diesem Beitrag zur Konsistenzprüfung die folgenden Themen:

- Zuerst wird die Frage beantwortet, weshalb es in PEARL [1] überhaupt einer Konsistenzprüfung bedarf.
- Dann werden die verschiedenen Vorgehensweisen bei der Programmerstellung in Ada [2] und PEARL einander gegenübergestellt.
- Zuletzt wird die GPP-Implementierung eines portablen Konsistenzprüfprogrammes vorgestellt.

Warum bedarf es einer Konsistenzprüfung?

Eine Antwort auf diese Frage gibt folgendes Beispiel:



Es handelt sich hierbei um zwei PEARL-Module, deren Inhalt auf das hier wesentliche reduziert wurde. Jeder Modul stellt für sich betrachtet bezüglich der Zugriffsfunktion F eine korrekte Übersetzungseinheit dar. Somit ist auch in jedem Modul eine Addition mittels F erlaubt.
Erst die Integration in einen Modulverband läßt erkennen, daß die gegenseitigen Schnittstellen nicht aufeinander abgestimmt sind. Dabei werden diese unterschiedlichen Zugriffsfunktionen wegen des gleichnamigen Bezeichners F einander zugeordnet. In bezug auf die Verträglichkeit gilt dann, daß die SPC-Definition sich der DCL-Definition unterordnen muß. Vor allem darf sie keine weitergehende Semantik zulassen, als in der DCL-Definition vorgesehen ist. In diesem Beispiel liegt jedoch nicht einmal Übereinstimmung in den Typen vor. Findet bei der Integration keine Konsistenzprüfung statt, dann geht die Zuverlässigkeit, die in jedem Modul für sich vorliegt, im Modulverband verloren.

Zur Untermauerung sollen nun typische Fehler aus der Praxis zitiert werden. Auch diese sind auf das wesentliche reduziert.

1. <u>Gleicher Speicher:</u>

```
DCL   A   FIXED   (7)
SPC   A   BIT     (7)
```

Die Binder eines Zielsystems identifizieren i.a. die globalen Objekte anhand der gleichen Adresse, oder besser anhand des gleichen Namens. Damit haben hier beide Module Zugriff auf ein- und denselben Speicher, z.B. ein 16 Bit-Wort. Während des Programmablaufs wird nun mittels der SPC-Zugriffsfunktion die linke Hälfte des Wortes erfaßt, mittels der DCL-Zugriffsfunktion dagegen die rechte Hälfte.

2. <u>Vertauschung von Parametern</u>

```
P: PROC (X BIT, Y FIXED)
SPC P   ENTRY  (FIXED, BIT)
```

Dieser Fall tritt dann auf, wenn bereits entwickelte Module in einem weiteren Projekt wiederum Verwendung finden. Hier ist die Reihenfolge der Parameter nicht adaptiert worden.

3. <u>Falscher SYSTEM-Teil</u>

```
E: CONNECTION-TO-INTERRUPT
SPC E  SIGNAL
```

Hier hat der Verantwortliche für den SYSTEM-Teil eine Hardwaregegebenheit dargestellt, die mit der Spezifikation im PROBLEM-Teil nicht abgestimmt wurde.

Ohne Konsistenzprüfung ist das Aufdecken solcher Fehler nur zur Ausführungszeit, bestenfalls zur Testzeit möglich. In jedem Fall ist ihre Lokalisierung äußerst aufwendig und kann in der Regel nur von denen durchgeführt werden, die mit der Implementierung vertraut sind. Fehler bei modulübergreifenden Schnittstellen äußern sich nicht zu Beginn des Programmablaufs, sondern erst dann, wenn sich gegenseitig ausschließende Operationen auf ein Objekt stattgefunden haben. Dann wiederum ist man auf der Suche nach Anweisungen, deren vermeintlich falsche Implementierung dieses Objekt überschrieben haben könnten.

<u>Worauf ist eine derartige Fehlersituation zurückzuführen?</u>

Die Ursache dafür liegt im Konflikt der beiden Ziele

. Typkonzept
. Modulare Programmerstellung

Der Compiler übernimmt die Aufgabe eine Übersetzungseinheit hinsichtlich der in der Sprachdefinition vorgebenen Kriterien zu analysieren und sie schließlich, sofern sie diesen Kriterien genügt, aus der Quellsprache in eine Zielsprache zu transformieren. In PEARL -wie in allen aktuellen Programmiersprachen mit ausgeprägtem Typkonzept - liegt der Schwerpunkt der statischen Programmanalyse auf der Überprüfung der Typdefinitionen und deren korrekter Verwendung. Dadurch wird die Sicherheit und Zuverlässigkeit der Programme erhöht.
Andererseits ist für die Entwicklung von Software das Zusammensetzen eines Programmes aus einzelnen Programmteilen, den Moduln, von großer Wichtigkeit. In modernen Programmiersprachen handelt es sich nicht nur darum, Unterprogramme bereitzustellen; vielmehr ist auch zu berücksichtigen, daß die Unterprogramme ersetzenden Module in

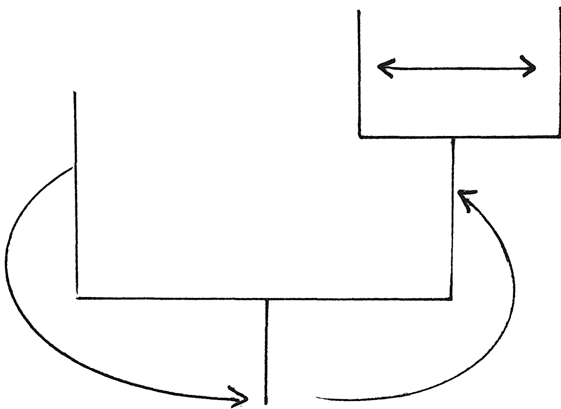
vielfacher komplexer Weise zu höheren, problemangepaßten Einheiten zusammengesetzt werden können. Ihre Wiederverwendung ist schon aus ökonomischen Gründen geboten.

Die Lösung des aufgezeigten Konfliktes ist über eine Konsistenzprüfung erreichbar. Schließlich soll die Sicherheit und Zuverlässigkeit eines Programmes nicht darunter leiden, daß es aus mehreren Moduln gebildet wurde. Diese Prüfung muß Teil einer statischen Programmanalyse sein. Ein Übersetzungssystem muß die korrekte Verwendung der Typen gemäß ihrer Definitionen auch im Modulverband garantieren.

Welche Aufgaben übernimmt die Konsistenzprüfung?

Da ist vor allem die Typprüfung. Die Vollständigkeit und Verträglichkeit aller globalen Bezüge ist zu überprüfen. Dadurch wird die Aussage des Compilers über die Zuverlässigkeit eines Moduls auf den gesamten Modulverband ausgedehnt. Alle Module müssen bei ihrer Integration dieser Prüfung unterzogen werden, um die konsistente Verwendung ihrer gegenseitigen Schnittstellen zu garantieren.

Als weiteres folgen Aufgaben, die nicht in der Programmiersprache selbst begründet sind, sondern die sich daraus ergeben, daß zum Ablauf nicht genügend Platz vorhanden ist und deshalb eine Überlagerungsstruktur erzeugt werden muß.



Globale Bezüge in einem segmentierten Programm

Ein Segmentbaum nimmt Einfluß auf die Sichtbarkeit der globalen Bezüge. Die Konsistenzprüfung muß solche Einflüsse auf die Zuordnung berücksichtigen.

1. Abwärtsreferenzen, die für Daten und Prozeduren gleichermaßen unproblematisch sind.

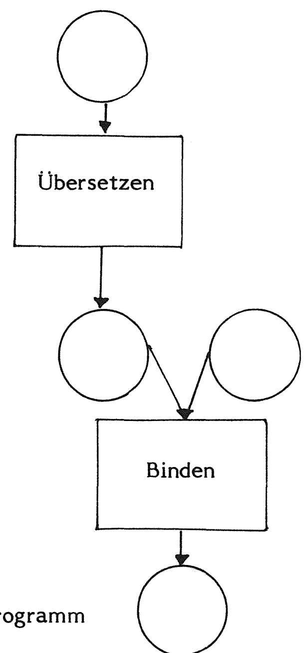
2. Aufwärtsreferenzen, die für Daten unzweckmäßig sind; für Prozeduren dagegen den Grund zum Segmentwechsel darstellen.

3. Seitenreferenzen, die in jedem Fall unzulässig sind.

Eine weitere Aufgabe stellt das Thema des sog. Vorbindens dar. Das Programm wird schrittweise aus Moduln zusammengestellt. Dabei werden nicht unbedingt alle globalen Bezüge zugeordnet. Solche im Prinzip nicht ablaufberechtigten Einheiten werden bspw. in Modulbibliotheken aufbewahrt. Die Eigenständigkeit solcher Einheiten kann es auch erforderlich machen, daß bei der weiteren Integration nicht mehr alle globalen Bezüge zur Verfügung stehen sollen; dann muß die Gültigkeit solcher Definitionen abgebaut werden. Man denke dabei an das Zusammenstellen eines mathematischen Paketes aus vielen Moduln mit vielen gegenseitigen Schnittstellen, das hinterher für seine Anwendung nurmehr wenige Schnittstellen aufweisen soll.

Programmerstellung in PEARL und Ada

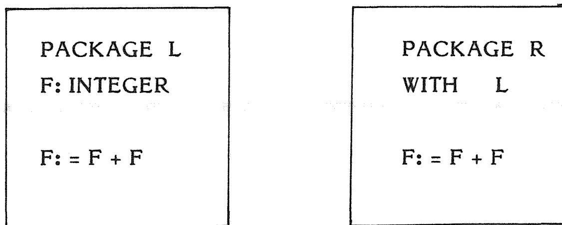
In der Sprachdefinition von PEARL sind hinsichtlich der Programmerstellung keine speziellen Restriktionen enthalten. Demnach ist hier auch die klassische Vorgehensweise (Übersetzen-Binden-Laden) angebracht



Die einzelnen Arbeitsphasen sind speziellen Aufgaben zugeordnet. Beim Übersetzen werden die Module aus einer Sprache in eine andere transformiert. Auf diese Weise werden Anwendersprachen, eventuell über mehrere Zwischensprachen, auf den ausführbaren Zielcode überführt. Beim Binden hingegen werden Module aus nur einer Sprachebene zu einem Modulverband zusammengesetzt. Dabei werden zwar Referenzen gelöst und Adressen zugeordnet, eine Sprachtransformation jedoch findet nicht statt.

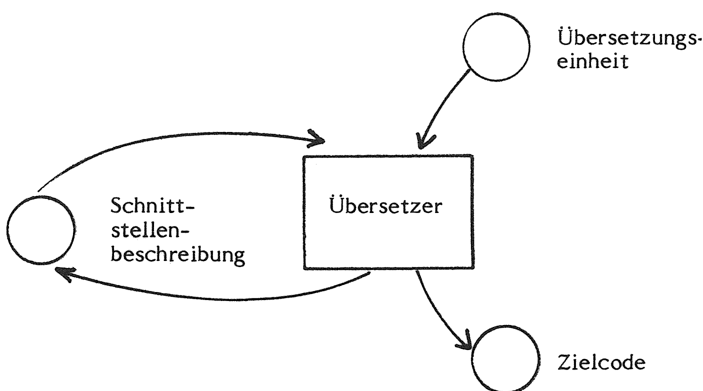
Als Gegensatz dazu soll jetzt die Vorgehensweise in Ada - stellvertretend für alle artverwandten Sprachen - skizziert werden.

Das eingangs formulierte Beispiel für globale Schnittstellen präsentiert sich in Ada folgendermaßen:



Es handelt sich hier um zwei Übersetzungseinheiten, deren Inhalt auf das hier Wesentliche reduziert wurde. In L wird das Objekt F definiert; eine Addition ist zulässig. In R dagegen wird F nicht definiert. Um eine Addition durchzuführen, muß die Definition von F sichtbar gemacht werden. Dies geschieht durch die Anweisung WITH L. Durch sie werden alle Definitionen aus L importiert, und damit auch die von F. Grundsätzlich wird der Bezug auf globale Schnittstellen nicht explizit zum Ausdruck gebracht, sondern implizit anhand der Namen von Übersetzungseinheiten. Auf diese Weise besteht zwischen den Übersetzungseinheiten eine baumartige Ordnung.

Um konsistente Schnittstellen innerhalb eines Programmes zu garantieren, sieht die Sprachdefinition ein sog. libraryfile vor. Es bildet die Datenbasis der Schnittstellenbeschreibungen aller Übersetzungseinheiten eines Programmes.



Für jedes zu erstellende Programm wird ein solches library-file angelegt. Die Buchführung in dieser Datenbasis wird vom Compiler übernommen. Bei jeder Übersetzung müssen sowohl die globalen Definitionen aus vorangegangenen Übersetzungen entsprechend den Sichtbarkeitsregeln herangezogen, als auch Neuzugänge integriert werden.

Folgende Gegenüberstellung verdeutlicht die unterschiedlichen Vorgehensweisen bei der Programmentwicklung.

Modulentwicklung

In PEARL ist die Entwicklung eines Moduls unabhängig von anderen Moduln. Deshalb kann er auch für sich alleine übersetzt werden und in einer speziell für ihn zugeschnittenen Umgebung getestet werden.

In Ada ist eine Übersetzungseinheit fast immer von anderen Übersetzungseinheiten abhängig. Ihre Übersetzung ist erst dann möglich, wenn das library file alle Vorgänger enthält; dabei wird sie dann selbst integriert.

Modulhaltung

Übersetzte PEARL-Module lassen sich in Bibliotheken halten. Dort können sie auch mit Moduln aus anderen Programmiersprachen zusammentreffen.

Übersetzte Ada-Einheiten lassen sich nur zentral im library file des jeweiligen Programms halten. Eine Änderung im bereits erstellten library file zieht eine erneute Übersetzung und Integration aller abhängigen Einheiten nach sich.

Integration

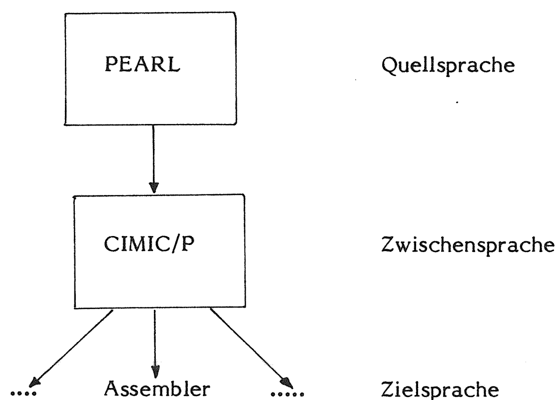
Bei der klassischen Vorgehensweise ist das Zusammenstellen eines Programms aus Moduln - die Integration - eine spezielle Arbeitsphase. In PEARL bedarf es dabei der Überprüfung der gegenseitigen Schnittstellen. Wird aus Platzgründen das Programm segmentiert, dann muß dabei zusätzlich die Sichtbarkeit der globalen Definitionen berücksichtigt werden.

In Ada entfällt diese Arbeitsphase völlig; die Integration ist Bestandteil der Übersetzung. Eine Überlagerungsstruktur läßt sich nicht explizit beschreiben. Implementationsabhängig kann der Aufbau eines Segmentbaumes mittels entsprechender PRAGMA's gesteuert werden. Allerdings muß dann das gesamte Programm so oft übersetzt werden, bis eine günstige Platzaufteilung gefunden ist.

Implementierung eines portablen Konsistenz-Prüfprogrammes

Beim Konzipieren eines Konsistenzprüfprogrammes für Basic PEARL Module erhebt sich die Frage nach der Sprachebene, auf der diese Überprüfung stattfinden soll.

Die GPP-Implementierung führt diese Prüfung auf der Zwischensprache CIMIC/P [3] durch.



Bei der Übersetzung eines PEARL-Moduls überprüft das front-end des Compilers die sprachspezifischen semantischen Zusammenhänge und transformiert ihn in CIMIC/P. Diese Zwischensprache ist einer abstrakten, PEARL ausführenden Maschine zugeordnet. Der Übergang auf reale Maschinen wird jeweils mit einem speziellen back-end vorgenommen.

Die Überprüfung der globalen Schnittstellen auf der Quellsprachebene durchzuführen ist nicht zweckmäßig. Bedeutet es doch eine Aussage zu treffen, die erst nach erfolgreicher semantischer Überprüfung des Moduls selbst sinnvoll ist. Andererseits ist auf der Ebene der Zielsprache wegen des bei der Übersetzung erfolgten Informationsverlustes über Typdefinitionen das Integrationsproblem nicht vollständig lösbar. In CIMIC/P dagegen liegen die nötigen Informationen über die Modulschnittstellen noch vor. Sie sind ihrer Bedeutung entsprechend in verschiedenen Listen aufbereitet. Somit kann eine schnelle Überprüfung der Konsistenz eines Modulverbandes auf Zielrechner-unabhängige Weise erfolgen.

Die Leistungen der portablen GPP-Implementierung eines Konsistenzprüfprogrammes für Basic PEARL beinhalten

Typprüfung

Sämtliche in DIN 66253 für Basic PEARL festgehaltenen Konsistenzregeln zwischen globalen Definitionen werden präzise geprüft.

Überlagerungsstruktur

Der Einfluß der Segmentierung auf die Sichtbarkeit von Definitionen wird berücksichtigt.

Vorbinden

Sowohl stufenweises Aufbauen eines Programmes als auch eventuelles Zudecken globaler Definitionen für weitere Integrationsschritte wird unterstützt.

Crossreference

Eine Übersicht über Deklaration und Spezifikation aller globalen Bezüge im Modulverband wird erstellt.

Das Konsistenzprüfprogramm ist Teil des GPP-Übersetzungssystems für Basic PEARL, dessen Verfügbarkeit aus der folgenden Tabelle hervorgeht.

Übersetzungs-Rechner ↑ / Ziel-Rechner →	PDP-11/ /03 /23	INTEL 8086	MICRO NOVA
DATA-GENERAL NOVA			X
INTERDATA 7/32		X	
PDP-11/23 /34	X	X	
SIEMENS 330	X	X	X
SIEMENS 7.531	X	X	X

Stand: 01-Dez.81

Schrifttum

[1] DIN 66253: Programmiersprache PEARL;
Teil 1 - Basic PEARL (Vornorm), Teil 2 - Full PEARL (Normentwurf). Beuth-Verlag, Berlin/Köln, 1980.

[2] United States Department of Defense:
Reference Manual for the Ada Programming Language; proposed standard document; july 1980

[3] B.F. Eichenauer:
Spezifikation der Zwischensprache CIMIC/C;
PDV-Entwicklungsnotizen PDV-E88; GfK Karlsruhe 1976
B.F. Eichenauer, R. Henn, K. Lucas, A. Zeh:
Spezifikation von CIMIC/P; Technische Notiz GPP/5/77;
GPP, München 1977.