LSE, SCA und PCA
Eine Einführung

Bernd Ulmann

27. Februar 2000

Inhaltsverzeichnis

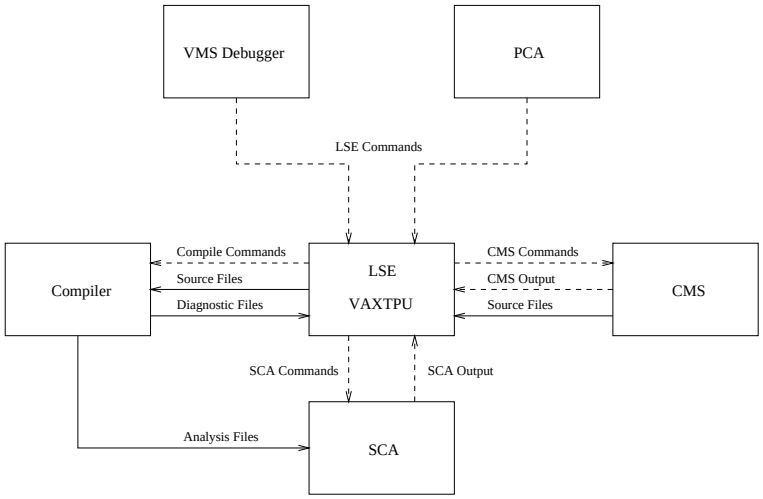

Abbildung 1: Die LSE-Umgebung

1 Überblick

Der Language-Sensitive-Editor, kurz LSE, ist ein leistungsfähiger Editor, dessen Fähigkeiten weit über die der meisten anderen Editoren hinausgehen. Er integriert vor allem die VAXSET-Utilities zu einer kompakten und vor allem in ihrer Bedienung konsistenten Entwicklungsumgebung unter VMS. Unter anderem bietet er die Möglichkeit, aus dem Editor heraus Compilerläufe durchzuführen, wobei er über Diagnostic-Files über hierbei eventuell aufgetretene Fehler informiert wird und einem die Möglichkeit bietet, direkt die vom Compiler gefundenen fraglichen Codestellen zwecks Korrektur etc. anzuspringen. Zusammen mit dem Source-Code-Analyser, dem SCA (siehe Abschnitt 2), bietet er darüber hinaus die Möglichkeit, beispielsweise unbekannte Programme zu analysieren. Zusammen mit dem Performance and Coverage-Analyser (PCA) können bereits vorhandene Programme hinsichtlich ihres Resourcenverhaltens optimiert werden, usf.[1]

Der LSE unterstützt die Sprachen Ada, BASIC, BLISS, C, CDD, COBOL, DATATRIEVE, DIBOL, FORTRAN, Pascal, PL/I sowie VAXELN Pascal. Das prinzipielle Zusammenspiel der einzelnen VAXSET-Komponenten mit dem LSE zeigt Abbildung 1, wobei das vorliegende Kursmaterial sich auf die Beschreibung der Utilities LSE, SCA und PCA beschränkt. Das Code-Management-System, CMS, wird nicht behandelt.

Der Source-Code-Analyzer, kurz SCA, stellt ein interaktives, multilinguales Werkzeug zur Erstellung von Cross-Referenzen sowie statischen Analysen von im Quellcode vorliegenden Programmen dar. Er ist ein in seiner Leistungsfähigkeit nicht zu unterschätzendes Hilfsmittel für Entwickler zum Verständnis des inneren Zusammenhangs und Aufbaus großer Software-Pakete – sowohl bei der Portierung, als auch bei der Wartung oder Weiterentwicklung großer Programmsysteme.

2 Der LSE

2.1 Einführung

Der LSE verfügt über zwei grundlegende Eingabemodi:

[1] Beispielsweise kann der LSE auch als Standardeditor für das VMS-MAIL-System eingesetzt werden, was durch das folgende DEFINE-Kommando erreicht wird:
`$ DEFINE MAIL$EDIT CALLABLE_LSE.`

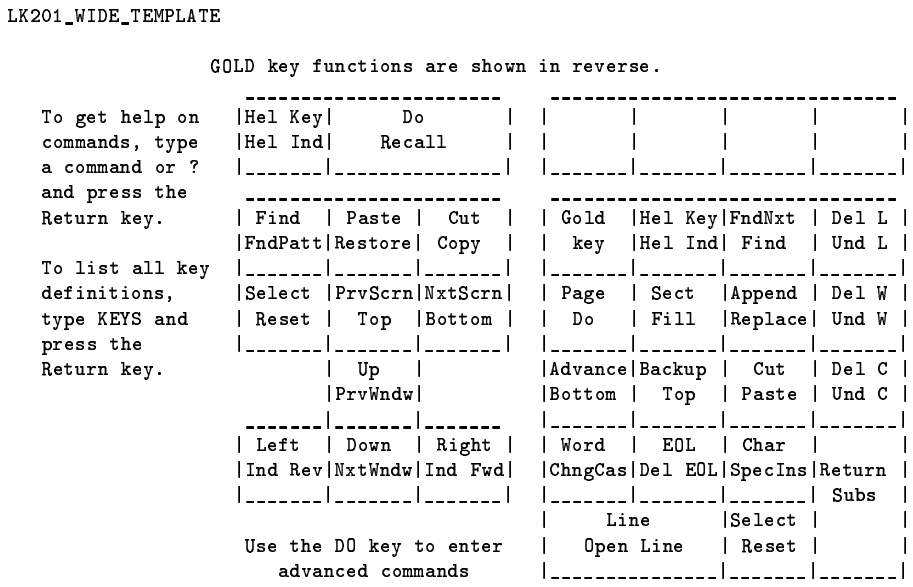

Abbildung 2: Die Keypad-Belegung

1. Der Texteingabemodus: Hier werden alle Eingaben in den aktuellen Textbuffer übernommen (abgesehen von Keypad-Eingaben).

2. Der Kommandomodus: In dieser Betriebsart können in der untersten Zeile des LSE-Fensters Kommandos an den Editor abgesetzt werden, mit deren Hilfe beispielsweise Compilerläufe o.ä. gestartet werden.

Zwischen den beiden Eingabevarianten kann mit Hilfe von CTRL/Z umgeschaltet werden. Nach dem Start befindet sich der LSE im Texteingabemodus; nach Drücken von CTRL/Z wird in den Kommandomodus umgeschaltet, der so lange aktiv bleibt, bis durch erneute Eingabe von CTRL/Z wieder in den Textmodus zurück geschaltet wird. Soll nur ein einziges Kommando eingegeben werden, kann der Kommandomodus auch durch die Tastenkombination PF1/KP7 erreicht werden. In diesem Fall wird nach erfolgter Eingabe eines Kommandos automatisch in den Textmodus zurückgeschaltet.

2.2 Das HELP-System

Durch Drücken der Taste PF2 kann jederzeit die in Abbildung 2 dargestellte Belegung des Keypads angezeigt werden. Eine ausführlichere Hilfe kann im Kommandomodus (zu erreichen durch CTRL/Z beziehungsweise PF1/KP7) durch den Befehl HELP erhalten werden.

2.3 Beenden des LSE

Der LSE kann auf zweierlei Weisen verlassen werden: Zum einen unter Abspeichern aller im Verlauf der Sitzung eingegebenen Änderungen, zum anderen ohne Speichern. Das Kommando EXIT verläßt den LSE unter Speichern aller Eingaben, während der Befehl QUIT den Editor verläßt und dabei alle Eingaben verwirft.

Wird der LSE durch EXIT verlassen und anschließend nur durch Eingabe von $ LSE (ohne Filename) wieder gestartet, lädt er das File, das vor dem letzten Verlassen mit EXIT bearbeitet wurde und springt an die zuletzt bearbeitete Stelle im Quellcode.

Diese Information wird allerdings nicht über ein Logout hinaus behalten, so daß der genannte Mechanismus nur innerhalb einer Login-Session angewandt werden kann.

2.4 Platzhalter und Token

Zwei grundlegende Begriffe im Zusammenhang mit dem LSE sind die sogenannten *Platzhalter* und *Token*:

Token: Tokens sind reservierte Schlüsselwörter oder Funktionsnamen, die als Template für entsprechende Sprachkonstrukte dienen. Beispielsweise kann das Token IF eingegeben und im LSE in eine vollständige IF-Konstruktion expandiert werden.

Platzhalter: Platzhalter sind generell von begrenzenden Zeichen umschlossen und werden vom LSE in den Editierbuffer eingefügt, sobald Tokens oder andere Platzhalter expandiert werden. Sie zeigen allgemein Positionen innerhalb einer expandierten Konstruktion an, an denen zusätzliche Angaben notwendig sind. Es wird zwischen optionalen und fakultativen Platzhaltern unterschieden; sie werden durch die Art der Begrenzungssymbole unterschieden – optionale Platzhalter sind von eckigen Klammern umschlossen, während fakultative innerhalb geschweifter Klammern stehen.

Wird der LSE beispielsweise durch `$ LSE BEISPIEL1.C` aufgerufen, erscheint folgende erste Zeile im Editorfenster:

```
{@compilation unit@}
[End of file]
```

Durch Drücken von CTRL/E kann dieser Platzhalter expandiert werden:

```
[@#module@]
[@module level comments@]
[@include files@]
[@macro definitions@]

[@preprocessor directive@]...

[@data type or declaration@]...;

[@function definition@]...;
[End of file]
```

Zwischen den einzelnen Platzhaltern und Tokens kann mittels CTRL/N resp. CTRL/P vor- und zurückgesprungen werden. Nach Expansion der Zeile `[@preprocessor directive@]` durch CTRL/E erscheint im unteren Fensterteil des LSE eine Auswahlliste möglicher Spracheinheiten, die an dieser Stelle eingesetzt werden können:

```
+->#define : Token replacement (replace the identifier with the given string)
|  #include : File inclusion (include the contents of the file)
|  #if : Conditional compilation (the constant expression is nonzero (TRUE)?)
|  #ifdef : Conditional compilation (the identifier is defined?)
|  #ifndef : Conditional compilation (the identifier is undefined?)
|  #line : Line number specification for purposes of error diagnostics
|  #undef : Token cancellation (cancel the preprocessor definition of the ident
  Choose one or press HELP key
```

Das gewünschte Item kann durch die Cursortasten und RETURN ausgewählt werden – in diesem Fall beispielsweise ein `#include`-Statement.

Nach Expansion der Zeile `[@function definition@]...;` wird folgende Auswahlliste angeboten:

```
+->function_definition : Defines a function
|  vararg_function_definition : Defines a function with a variable-length argum
+- main_function_definition : Defines a main function
  Choose one or press HELP key
```

Nach Auswahl des letzten Punktes (`main_function`) bietet sich folgendes Bild im Editorbuffer (das `#include`-Statement wurde in der Zwischenzeit durch ein `<stdio.h>` ergänzt):

```
[@#module@]
[@module level comments@]
[@include files@]
[@macro definitions@]

#include <stdio.h>
[@preprocessor directive@]...

[@data type or declaration@]...;



[@function level comments@]
{@main () OR main function that accept arguments from the command line@}
[@MAIN_PROGRAM@]
{
    [@block declaration@]...;

    {@statement@}...;
}

[@function definition@]...;
[End of file]
```

Durch Expansion der Zeile `@main () OR main function...` kann wieder in einer Auswahlliste zwischen verschiedenen Formen einer C-Hauptprogrammdefinition gewählt werden. Die Zeile sieht nun wie folgt aus:

```
{@main function name@} ()
```

Nun muß lediglich noch der Funktionsname expandiert werden, um eine gültige Deklaration eines Hauptprogrammes in C zu erhalten:

```
main ()
```

Das restliche Programm kann nun ebenfalls durch entsprechende Expansionsoperationen generiert werden, wobei natürlich gewünschte Parameter explizit eingegeben werden müssen. Letzten Endes hat das in diesem Beispiel erzeugte Programm folgende Gestalt:

```
[@#module@]
[@module level comments@]
[@include files@]
[@macro definitions@]

#include <stdio.h>
[@preprocessor directive@]...

[@data type or declaration@]...;



[@function level comments@]
{@main () OR main function that accept arguments from the command line@}
```

```
[@MAIN_PROGRAM@]
{
    [@block declaration@]...;

    printf ("Hi!\n");
    [@statement@]...;
}

[@function definition@]...;
[End of file]
```

Nun sind die nicht länger benötigten Token und Platzhalter zu beseitigen, was durch die Tastenkombination CTRL/K (*Kill*) geschehen kann:

```
#include <stdio.h>



main ()
{

    printf ("Hi!\n");
}
[End of file]
```

Wie dieses Beispiel zeigt, können mit Hilfe des LSE syntaktisch korrekte Programme fast automatisch erzeugt werden.

2.5 Pseudocode

Der LSE bietet die Möglichkeit, während der Programmentwicklung sogenannten Pseudocode einzugeben, um beispielsweise Designentscheidungen zu dokumentieren, o.ä. Ein Pseudocodeabschnitt wird durch Drücken von PF1/SPACEBAR eingeleitet:

```
#include <stdio.h>



main ()
{
Kleine Hallo-Welt-Demonstration
    printf ("Hi!\n");
}
[End of file]
```

An dieser Stelle (})kann nun beliebiger Text, Pseudocode, eingegeben werden. Vor dem eigentlichen Übersetzungslauf müssen alle Pseudocodeteile in Kommentare umgewandelt werden, was durch das Kommando PF1/B geschieht:

```
#include <stdio.h>
```

```
main ()
{
/*
**  Kleine Hallo-Welt-Demonstration
*/
{@tbs@}
    printf ("Hi!\n");
}
[End of file]
```

Der Platzhalter `@tbs@`, der automatisch nach der Umwandlung von Pseudocode in Kommentarzeilen erzeugt wird, kann durch einfaches Überschreiben mit Code ersetzt werden und dient dazu, direkt Code einzugeben, sobald ein Pseudocodeabschnitt konkretisiert wird.

2.6 Übersetzen von Code

Das Übersetzen eines Programmes aus dem LSE heraus geschieht durch das Kommando COMPILE. Nach erfolgreicher Übersetzung stehen folgende Zeilen im unteren Fensterteil:

```
Starting compilation: CC/DECC $1$DIA2:[ULMANN.BASTEL]BEISPIEL1.C;4 /DIAGNOSTICS=$
Compilation of buffer BSP1.C completed
```

Einige Compiler, wie zum Beispiel PASCAL bieten die Möglichkeit, Platzhalter, Tokens und Pseudocode, die während der Programmentwicklung mit dem LSE erzeugt wurden, zur Übersetzungszeit zu überlesen, so daß sie nicht explizit gelöscht werden müssen. Hierfür dient in der Regel der Qualifizierer /DESIGN[2].

Für das folgende Beispiel wurde der Strichpunkt am Ende des printf-Statements entfernt, um einen Fehler zu provozieren. Wird ein Compilerlauf im LSE gestartet, legt der Compiler alle während des Übersetzungsvorganges auftretenden Messages in einem Diagnosticfile ab, das der LSE seinerseits wieder einlesen kann. Das einfache COMPILE-Statement startet lediglich den Übersetzerlauf, leitet jedoch nicht automatisch die Auswertung des Diagnosticfiles durch den LSE ein, so daß im Normalfall eine Übersetzung mit COMPILE/REVIEW gestartet werden sollte. In obigem Beispiel zieht dies folgende Ausgabe nach sich:

```
Line  885:      }
%CC-E-NOSEMI, Missing ";".

Line  885:      }
%VCG-I-NOBJECT, No object file produced.

[EOB]
 Buffer: $REVIEW                                  | Read-only | Insert | Forward

#include <stdio.h>



main ()
{
/*
**  Kleine Hallo-Welt-Demonstration
*/

    printf ("Hi!\n")
```

[2] Qualifier können nach dem COMPILE-Statement in Form von $/QUALIFIER angegeben werden.

```
}
[End of file]
```

Das Editorfenster wird zweigeteilt, wobei im oberen Teil die vom Compiler generierten Fehler- und Warnungsmeldungen stehen, während im unteren Fenster der entsprechende Sourcecode angezeigt wird.

Zwischen einzelnen Meldungen im REVIEW-Buffer kann mit CTRL/F und CTRL/B vor- und zurückgesprungen werden, während CTRL/G an die Stelle im Quellcode springt, die vom Compiler als fehlerhaft oder zumindest bedenklich erkannt wurde.

Der Wechsel zwischen verschiedenen Teilfenstern im LSE ist durch die Kommandos PF1/UP beziehungsweise PF1/DOWN möglich.

2.7 Buffer im LSE

Wie die meisten Editoren unterstützt der LSE die gleichzeitige Verwendung mehrerer Buffer innerhalb einer Editor-Session. Jedem Buffer ist hierbei ein eindeutiger Name zugeordnet, über den er im folgenden angesprochen werden kann. Darüberhinaus besitzt jeder Buffer eine Reihe von Attributen, die im folgenden kurz dargestellt sind:

Insert/Overstrike: Dieses Attribut unterscheidet zwischen den beiden grundlegenden Eingabemodi des LSE: Zum einen der Einfügemodus, in dem bereits bestehender Text auseinandergerückt wird, falls neue Zeichen eingefügt werden; zum anderen der Überschreibmodus, in dem bereits vorhandene Zeichen durch neue Zeichen überschrieben werden können.

Zwischen diesen beiden Modi kann mit Hilfe von CTRL/A umgeschaltet werden, beziehungsweise durch die Kommandos SET INSERT und entsprechend SET OVERSTRIKE.

Forward/Reverse: Durch dieses Attribut wird die Ausführungsrichtung einer Reihe von Kommandos, wie beispielsweise Suchoperationen, etc. festgelegt. Das Kommando KP4 bzw. SET FORWARD schaltet in den Forwardmodus, während KP5 resp. SET REVERSE in den Reversemodus schaltet.

Input/Output: Jedem Buffer ist ein Eingabe- beziehungsweise ein Ausgabefile zugeordnet. Das Eingabefile ist die Datei, aus der der Bufferinhalt gelesen wurde (beispielsweise durch ein GOTO FILE-Kommando), während der Inhalt des Buffers in das Ausgabefile geschrieben wird. Das einem Buffer zugeordnete Ausgabefile kann durch das Kommando SET OUTPUT_FILE explizit spezifiziert werden.

Read/Write: Ein Buffer mit dem Attribut READ wird bei Beenden des Editors nicht in eine Datei gespeichert, wie dies bei einem WRITE-Buffer der Fall wäre.

Modifiable/Unmodifiable: Ein Buffer kann entweder ein ausschließlicher Lesebuffer sein, oder ein modifizierbarer Buffer. Standardmäßig sind Buffer modifizierbar. Soll ein Readonly-Buffer erzeugt werden, kann dies durch das Kommando GOTO FILE/READ erfolgen. Zwischen den beiden Modi kann auch durch die Kommandos SET WRITE beziehungsweise SET READ umgeschaltet werden.

Language: Jedem Buffer ist eine Programmiersprache zugeordnet, die im allgemeinen aus der Extension des geladenen Files bestimmt wird. Darüberhinaus kann die einem Buffer zugeordnete Sprache jedoch auch durch das Kommando SET LANGUAGE eingestellt werden.

Wrap/Nowrap: Ein Buffer mit dem Attribut Nowrap besitzt die Eigenschaft, daß Text, der über den rechten Textrand hinausragt, nicht automatisch umgebrochen wird. Das gewünschte Attribut kann durch SET WRAP bzw. SET NOWRAP gesetzt werden.

2.8 COLLAPSE und EXPAND

Mit Hilfe des COLLAPSE-Kommandos können ganze Blöcke eines Programmes in eine Zeile zusammengefaßt werden (sie kollabieren). Das Expandieren solcher zusammengefaßten Blöcke geschieht durch das EXPAND-Kommando oder kurz CTRL/E. Die Kurzform des COLLAPSE-Kommandos ist CTRL/\`

Durch Eingabe des Kommandos VIEW SOURCE kann eine ganze Gruppe von Blöcken, die durch ein COLLAPSE-Kommando zusammengefaßt wurden, expandiert werden.

2.9 Suchen und Ersetzen im LSE

Der LSE verfügt über eine mächtige Suchfunktion, die auch die Verwendung von Wildcards bei der Spezifizierung von Suchausdrücken unterstützt. Die einfachste Form eines Suchkommandos kann durch PF1/KP3 gestartet werden. Nach Eingabe dieses Kommandos muß ein Suchstring eingegeben werden, der danach – entsprechend des Forward-/Reverse-Attributes des Buffers vorwärts oder rückwärts im Buffer gesucht wird. Wird der Suchstring gefunden, springt der Cursor an den Anfang der entsprechenden Textstelle. Drücken von KP3 wiederholt die letzte Suche, so daß durch mehrfache Eingabe von KP3 das n-te Auftreten des gewünschten Strings gesucht wird.

Sollen komplexere Suchstrings formuliert werden, die auch Wildcards enthalten dürfen, muß explizit das SEARCH/PATTERN-Kommando eingesetzt werden[3].

Zulässige Wildcards bei solchen Suchoperationen sind * und %, wobei der Stern eine beliebige Anzahl beliebiger Zeichen ersetzt, während das Prozentsymbol genau ein beliebiges Zeichen repräsentieren kann. Soll innerhalb eines Suchstrings ein Stern oder ein Prozentzeichen auftreten, muß das betreffende Symbol durch Voranstellen eine Backslashes (\) gequotet werden.

Das Ersetzen von Zeichenketten im LSE kann durch das Kommando SUBSTITUTE gestartet werden. Beispielsweise ersetzt das Kommando

```
SUBSTITUTE printf my_printf
```

jedes Auftreten der Zeichenkette `printf` durch `my_printf`. Zu beachten ist in diesem Zusammenhang, daß die obige Form des Kommandos den ersetzten String automatisch in Großbuchstaben konvertiert. Um dies zu vermeiden, ist folgender Aufruf notwendig:

```
SUBSTITUTE/CASE_MATCHING printf my_printf
```

Dieser Aufruf führt dazu, daß eine Zeichenkette aus Kleinbuchstaben durch eine Zeichenkette aus Kleinbuchstaben ersetzt wird, während ein String aus Großbuchstaben auch durch einen String aus Großbuchstaben substituiert wird (ausschlaggebend für die Steuerung dieses Verhaltens ist jedoch nur das erste Zeichen des zu ersetzenden Strings).

Die komplexeste Form des Ersetzens im LSE wird durch das Kommando

```
SUBSTITUTE/PATTERN
```

eingeleitet. So ersetzt beispielsweise

```
SUBSTITUTE/PATTERN 1%3 103
```

alle dreibuchstabigen Zeichenketten der Form 1%3 durch die feste Zeichenkette 103.

[3] Das einfache SEARCH-Kommando entspricht dem Kommando PF1/KP3.

2.10 Aufruf des LSE aus dem Debugger

Wie bereits in Abbildung 1 dargestellt wurde, besteht die Möglichkeit, aus dem Debugger heraus den LSE aufzurufen, um beispielsweise direkt Änderungen im Quellcode vornehmen zu können, sobald Fehler im Debugger erkannt worden sind. Von Vorteil hierbei ist, daß der LSE bei einem solchen Aufruf direkt an die Stelle im Quellcode springt, die gerade im Debugger näher untersucht wurde.

Zunächst muß jedoch ein Programm für die Ausführung im Debugger entsprechend übersetzt werden, was durch das Kommando

```
COMPILE/REVIEW $/DEBUG
```

geschehen kann. Nach Verlassen des LSE durch das Kommando EXIT kann das Programm wie gewohnt mit

```
LINK/DEBUG BEISPIEL1
```

gelinkt und mit

```
RUN BEISPIEL1
```

ausgeführt werden[4].

Aus dem Debugger heraus kann nun an beliebiger Stelle durch Eingabe des Kommandos EDIT der LSE aufgerufen werden. Nach Verlassen des LSE wird wieder in den Debugger zurückgekehrt. Da dieses Verhalten in den meisten Fällen unerwünscht ist, existiert das Kommando EDIT/EXIT, das, analog zu EDIT, den LSE aufruft, nach dem Beenden des LSE allerdings auch den Debugger beendet, so daß ein zusätzliches EXIT im Debugger unterbleiben kann.

[4]Um sicherzustellen, daß der Debugger nicht unter **DECwindows**, sondern im Textmode gestartet wird, sollte zu Beginn der Session – beispielsweise im LOGIN.COM folgendes Kommando ausgeführt werden: `$ DEFINE DBG$DECW$DISPLAY .` Hierdurch wird der Debugger gezwungen, im Textmodus abzulaufen.

A Die wichtigsten Befehle im LSE

CTRL/A	Schaltet wechselweise zwischen Insert- und Overstrikemodus um.
CTRL/B	Springt im REVIEW-Buffer eine Meldung rückwärts.
CTRL/E	Expandiert einen Platzhalter oder ein Token.
CTRL/F	Springt im REVIEW-Buffer eine Meldung vorwärts.
CTRL/G	Springt vom REVIEW-Buffer in den Quellcode.
CTRL/K	Löscht einen Platzhalter und springt eine Position vorwärts.
CTRL/N	Springt einen Platzhalter, ein Token vorwärts.
CTRL/P	Springt einen Platzhalter, ein Token rückwärts.
CTRL/\	Faßt einen Programmabschnitt zusammen (COLLAPSE).
KP3	Wiederholt die letzte durch PF1/KP3 gestartete Suche.
KP4	Schaltet in den Forwardmodus.
KP5	Schaltet in den Reversemodus.
PF1/KP3	Startet eine neue Suche.
PF1/B	Wandelt Pseudocode in Kommentar um.
PF1/SPACEBAR	Startet die Eingabe von Pseudocode.
PF1/UP	Springt ein Fenster nach oben.
PF1/DOWN	Spring ein Fenster nach unten.
COLLAPSE	Faßt einen Programmabschnitt zusammen.
COMPILE	Übersetzt das im aktuellen Buffer befindliche Programm.
EXIT	Verläßt den Editor mit Abspeichern.
GOTO FILE	Lädt ein File in den Editor.
QUIT	Verläßt den LSE ohne Speichern der Eingaben.
SEARCHPATTERN	Startet eine Suche, deren Suchstring auch Wildcards enthalten darf.
SET FORWARD	Schaltet in den Forwardmodus.
SET INSERT	Schaltet einen Buffer in den Insertmodus.
SET LANGUAGE	Ordnet einem Buffer eine bestimmte Sprache zu.
SET NOWRAP	Löscht das Wrap-Attribut des aktuellen Buffers.
SET OUTPUT_FILE	Ordnet einem Buffer ein Ausgabefile zu.
SET OVERSTRIKE	Schaltet einen Buffer in den Overstrikemodus.
SET READ	Schaltet den aktuellen Buffer in den Readonlymodus.
SET REVERSE	Schaltet in den Reversemodus.
SET WRAP	Setzt das Wrap-Attribut für den aktuellen Buffer.
SET WRITE	Schaltet den aktuellen Buffer in den Zustand modifizierbar.
SUBSTITUTE	Ersetzt Zeichenketten durch andere Zeichenketten.
VIEW SOURCE	Expandiert eine ganze kollabierte Gruppe.

2 Der SCA

Das in den folgenden Abschnitten über den Einsatz des SCA verwendete FORTRAN-Programm E1.FOR stellt einen Emulator für eine experimentelle CPU dar, was allerdings für die Ausführungen im einzelnen unerheblich ist und an dieser Stelle lediglich der Vollständigkeit halber Erwähnung finden sollte.

Die folgenden Abschnitte stellen die wichtigsten Eigenschaften des SCA sowie des Zusammenspiels von SCA und LSE dar, wobei eine Reihe von Features unerwähnt bleibt, was aufgrund der Kürze dieser Einführung unvermeidlich ist.

2.1 Ein unbekanntes Programm – was nun?

Eine der Hauptanwendungen des SCA ist sicherlich in der Untersuchung unbekannter beziehungsweise unübersichtlicher oder schlecht (gar nicht?) dokumentierter Programme, die sich im (durchaus realistischen) schlimmsten Fall über eine Vielzahl einzelner Module erstrecken können, zu sehen.

Auch bei der Wartung bekannter Quellcodes ist der SCA von Nutzen, da er innerhalb kürzester Zeit wertvolle Crossreferenceinformationen liefern kann und dadurch schnell unerwartete Datenabhängigkeiten, etc. aufzudecken vermag. Darüberhinaus sind im Hinblick auf Anzahl und Typ der Argumente von Funktionsaufrufen Konsistenzchecks möglich, die weit über das, was die meisten Compiler zur Verfügung stellen, hinausgehen.

Um ein gegebenes Programm einer Analyse durch den SCA zu unterziehen, sind eine Reihe vorbereitender Schritte notwendig. Zunächst muß eine SCA-Library angelegt werden, in der der Analyzer Informationen über das zu untersuchende Programm ablegt, auf die im folgenden zugegriffen werden kann. Hier werden beispielsweise die Namen der verwendeten Variablen, die Orte ihres Auftretens, ihre Typen, etc. verwaltet.

Der SCA ist in der Lage, Programme in allen Sprachen zu analysieren, für die ein Compiler zur Verfügung steht, der in der Lage ist, entsprechende Analyse-Files zu generieren, die im allgemeinen anhand ihrer Extension .ANA zu erkennen sind. Der erste Schritt in der Analyse eines Programmes besteht nun aus der Erzeugung eines solchen Analyse-Files:

```
$ FORTRAN/ANALYSIS_DATA E1
```

Durch diesen Compilerlauf wurde nun das File E1.ANA generiert, das die notwendigen Informationen für die folgenden Schritte enthält. Als nächstes muß das SCA-Environment für das zu untersuchende Programm E1 erzeugt werden. Dies umfaßt im einzelnen:

- Das Erzeugen eines Verzeichnisses, das die SCA-Library enthalten wird,
- das Generieren der Library an sich sowie
- das Laden des Analysis-Files in die Bibliothek.

Im einzelnen sind folgende Schritte notwendig:

```
$ CREATE/DIRECTORY [.LIB]
$ SCA
SCA> CREATE LIBRARY [.LIB]
%SCA-S-LIB, your SCA Library is $1$DIA2:[ULMANN.DEMO.LIB]
%SCA-S-NEWLIB, SCA Library created in $1$DIA2:[ULMANN.DEMO.LIB]
SCA> LOAD E1
%SCA-S-LOADED, module UEPEMU loaded
%SCA-S-LOADED, module EXECUTE loaded
%SCA-S-LOADED, module RUN loaded
%SCA-S-LOADED, module DEPREG loaded
%SCA-S-LOADED, module DEPOSIT loaded
%SCA-S-LOADED, module DISASM loaded
%SCA-S-LOADED, module SAVE loaded
%SCA-S-LOADED, module SNAP loaded
%SCA-S-LOADED, module RESTART loaded
%SCA-S-LOADED, module RDMP loaded
%SCA-S-LOADED, module LOAD loaded
%SCA-S-LOADED, module DUMP loaded
%SCA-S-LOADED, module NTOA loaded
%SCA-S-LOADED, module ATOI loaded
%SCA-S-LOADED, module SPLIT loaded
%SCA-S-LOADED, module TRIM loaded
%SCA-S-LOADED, module ENABLE_CTRLC loaded
%SCA-S-LOADED, module CTRLC_PRESSED loaded
%SCA-S-LOADED, module ALU32 loaded
%SCA-S-LOADED, module ALU4 loaded
%SCA-S-LOADED, module STAGE1 loaded
%SCA-S-LOADED, module HELP loaded
%SCA-S-COUNT, 22 modules loaded (22 new, 0 replaced)
SCA> EXIT
```

```
$
```

Nach Abschluß der vorbereitenden Maßnahmen für die Analyse des Programmes E1.FOR zeigen die nun folgenden Abschnitte die wichtigsten Möglichkeiten auf, die der SCA bei der Untersuchung von Quellcode bietet.

2.2 Der SCA mit dem LSE als Frontend

Obwohl prinzipiell die Möglichkeit besteht, den SCA als Standaloneprogramm zu verwenden, wird im folgenden stets der bereits kurz vorgestellte Editor LSE als Frontend verwendet, da die Kombination dieser beiden Programme um ein Vielfaches praktischer und leistungsfähiger als der SCA alleine ist.

Zunächst ist der LSE durch Eingabe des Kommandos $ LSE zu starten. Im Anschluß hieran muß die zu verwendende SCA-Bibliothek deklariert werden, um sicherzustellen, daß LSE und SCA auf alle notwendigen Daten des Analyselaufes des Compilers zugreifen können:

```
LSE Command> SET LIBRARY [.LIB]
```

Das verwendete Beispielprogramm besitzt zwei Vorteile: Zum einen besteht es aus lediglich einem einzigen Modul, das sich auch in nur einem Verzeichnis befindet, zum anderen ist dieses Verzeichnis das aktuelle Defaultverzeichnis, unter dem gearbeitet wird. In komplexeren Fällen sind diese vereinfachenden Voraussetzungen jedoch nicht gegeben, so daß dem LSE mitgeteilt werden muß, in welchen Verzeichnissen er von Fall zu Fall nach dem Quellcode einzelner Module oder Routinen zu suchen hat. Angenommen, es handelt sich bei dem zu untersuchenden Programm um drei Module, die sich in den Verzeichnissen

```
DISK$USER:[ULMANN.EMULATOR.SOURCE.FRONTEND]
DISK$USER:[ULMANN.EMULATOR.SOURCE.MACHINE]
DISK$USER:[ULMANN.EMULATOR.SOURCE.MISC]
```

befinden, so kann dem LSE diese etwas komplexere Sachlage mit Hilfe des folgenden Kommandos bekannt gemacht werden, so daß er jederzeit auf alle benötigten Sourcefiles zugreifen kann:

```
LSE Command> SET SOURCE_DIR DISK$USER:[ULMANN.EMULATOR.SOURCE.FRONTEND], ...
```

Über die aktuelle Bibliothek, beziehungsweise die aktuellen Sourcecode-Verzeichnisse kann man sich mit den Kommandos

```
LSE Command> SHOW LIBRARY/FULL
LSE Command> SHOW SOURCE_DIRECTORY
```

informieren.

Eine Liste aller Module, aus denen sich der zu untersuchende Quellcode zusammensetzt, läßt sich durch das Kommando SHOW MODULE generieren, das eine detaillierte Auflistung aller analysierten Programmeinheiten liefert. Im vorliegenden Beispiel führt dies zu folgender Ausgabe[5]:

[5] Hierbei werden die einzelnen Routinen in alphabetischer Reihenfolge aufgelistet.

```
   Module              #  Ident    Language    Compiled

ALU32                  1  01       FORTRAN  19-Feb-2000 20:57
ALU4                   1  01       FORTRAN  19-Feb-2000 20:57
ATOI                   1  01       FORTRAN  19-Feb-2000 20:57
CTRLC_PRESSED          1  01       FORTRAN  19-Feb-2000 20:57
DEPOSIT                1  01       FORTRAN  19-Feb-2000 20:57
DEPREG                 1  01       FORTRAN  19-Feb-2000 20:57
DISASM                 1  01       FORTRAN  19-Feb-2000 20:57
DUMP                   1  01       FORTRAN  19-Feb-2000 20:57
ENABLE_CTRLC           1  01       FORTRAN  19-Feb-2000 20:57
EXECUTE                1  01       FORTRAN  19-Feb-2000 20:57
HELP                   1  01       FORTRAN  19-Feb-2000 20:57
LOAD                   1  01       FORTRAN  19-Feb-2000 20:57
NTOA                   1  01       FORTRAN  19-Feb-2000 20:57
RDMP                   1  01       FORTRAN  19-Feb-2000 20:57
RESTART                1  01       FORTRAN  19-Feb-2000 20:57
RUN                    1  01       FORTRAN  19-Feb-2000 20:57
SAVE                   1  01       FORTRAN  19-Feb-2000 20:57
SNAP                   1  01       FORTRAN  19-Feb-2000 20:57
SPLIT                  1  01       FORTRAN  19-Feb-2000 20:57
STAGE1                 1  01       FORTRAN  19-Feb-2000 20:57
TRIM                   1  01       FORTRAN  19-Feb-2000 20:57
UEPEMU                 1  01       FORTRAN  19-Feb-2000 20:57
```

Um herauszufinden, welche Routinen das Hauptprogramm `UEPEMU` aufruft, kann der Befehl `FIND CALLED_BY` eingesetzt werden. Mit seiner Hilfe ist es möglich, Aufrufketten zu verfolgen. So liefert beispielsweise ein

```
LSE Command> FIND CALLED_BY UEPEMU
```

die Ausgabe

```
UEPEMU procedure calls
   ATOI function
   DEPOSIT routine
   DEPREG routine
   DISASM routine
   DUMP routine
   ENABLE_CTRLC routine
   EXECUTE function
   HELP routine
   IAND function
   IBCLR function
   IBSET function
   ISHFT function
   LIB$STOP routine
   LOAD routine
   NTOA routine
   RDMP routine
   RESTART routine
   RUN routine
   SAVE routine
```

```
   SNAP routine
   SPLIT routine
   SYS$ASSIGN function
   TRIM function

Query: 1   | Command: FIND CALLED_BY UEPEMU | Forward
[End of File]
```

Die Ausgabe des obigen Kommandos erscheint in einem eigenen Fenster, innerhalb dessen wie gewohnt mit Hilfe der Cursortasten, etc. navigiert werden kann. Um beispielsweise an die Stelle im Quellcode zu springen, an der die Routine ATOI aufgerufen wird, kann innerhalb des Query-Buffers in der entsprechenden Zeile das Kommando GOTO SOURCE, beziehungsweise CTRL/G verwendet werden. Dies ist jedoch nicht direkt möglich, sondern erst nach Expansion der gewünschten Zeile (durch CTRL/E)[6]:

```
   ATOI function
    UEPEMU\4427          call reference
    UEPEMU\4475          call reference
```

Nach der Expansion werden alle Referenzen, die zu dem jeweiligen Eintrag gehören, angezeigt – nur diese können direkt durch CTRL/G angesprungen werden. Zu jedem Vorkommen werden hierbei Modul sowie Zeilennummer des Aufrufes angegeben.

Nach CTRL/G auf den ersten Eintrag obiger Liste ergibt sich folgendes Bild:

```
        CALL SPLIT (LINE, ARG, ',', ARGNUM)
        DO 5 I = 1, 3
          J = TRIM (ARG (I))
5         ARGVAL (I) = ATOI (ARG (I), '$')
        ABORT = .FALSE.
        IF (CMND .EQ. 'QUIT' .OR. CMND .EQ. 'EXIT') THEN
          WRITE (*, '(X, A, $)') 'MAIN> DO YOU REALLY WANT TO QUIT ? (Y/N) '
          READ (*, '(A)') LINE
          IF (LINE .EQ. 'Y') GOTO 9999
        ELSE IF (CMND .EQ. 'RESET') THEN
          WRITE (*, '(X, A, $)') 'MAIN> DO YOU REALLY WANT TO RESET ? (Y/N) '
          READ (*, '(A)') LINE
          IF (LINE .EQ. 'Y') THEN
            DO 20 I = 0, MAXMEM
 Buffer: E1.FOR                                    | Write | Insert | Forward
UEPEMU procedure calls
   ATOI function
    UEPEMU\4427          call reference
    UEPEMU\4475          call reference
   DEPOSIT routine
   DEPREG routine
   DISASM routine
   DUMP routine
   ENABLE_CTRLC routine
   EXECUTE function
   HELP routine
   IAND function
   IBCLR function
   IBSET function
Query: 1   | Command: FIND CALLED_BY UEPEMU | Forward
[End of File]
```

[6]Wie bereits bei anderen Gelegenheiten kann auch hier im Queryfenster mit CTRL/B und CTRL/F rückwärts und vorwärts gesprungen werden.

2.3 Queries

Queries, Abfragen, sind ein zentraler Bestandteil der SCA-Benutzerschnittstelle.

2.3.1 Einfache Abfragen

Ein Beispiel für eine einfache Query war bereits im vorangegangenen Abschnitt in Form des Kommandos `FIND CALLED_BY` zu sehen. Im allgemeinen nimmt das Kommando `FIND` die tragende Stellung in allen SCA-Queries ein. So sucht beispielsweise das Kommando

```
LSE Command> FIND SCRATCH
```

alle Stellen des Quellcodes, an denen das Symbol `SCRATCH` verwendet wird:

```
SCRATCH variable
    EXECUTE\10          variable declaration
    EXECUTE\129         write reference
    EXECUTE\135         read reference
    EXECUTE\136         read reference
    EXECUTE\144         read reference
    EXECUTE\147         read reference
    EXECUTE\152         read reference
    EXECUTE\152         read reference
    EXECUTE\154         read reference
    EXECUTE\155         read reference
    EXECUTE\163         read reference
    EXECUTE\165         read reference
    EXECUTE\171         read reference
    EXECUTE\172         read reference
    EXECUTE\180         read reference
    EXECUTE\182         read reference
    EXECUTE\184         read reference
    EXECUTE\184         read reference
    EXECUTE\303         write reference
    EXECUTE\322         read reference
Query: 3   | Command: FIND SCRATCH | Forward
```

Die einzelnen Einträge dieser Liste können im übrigen direkt, d.h. ohne vorangehendes `EXPAND` mit `CTRL/G` angesprungen werden.

Das `FIND`-Kommando verfügt über die Möglichkeit, Wildcards in Queries zu verwenden, wobei das bereits in Abschnitt 2.9 Gesagte gilt. So sucht beispielsweise der Befehl

```
LSE Command> FIND *B*T
```

alle Vorkommnisse von Symbolen, in deren Namen die Buchstaben `B` und `T` auftreten – jeweils angeführt von einer beliebigen Anzahl anderer Zeichen. Für das Beispielprogramm ergeben sich hierbei unter anderem Einträge wie diese:

```
ABORT variable
    CTRLC_PRESSED\5     variable declaration
    CTRLC_PRESSED\6     variable declaration
    CTRLC_PRESSED\8     write reference
```

```
ABORT variable
ABORT variable
ABORT variable
ABORT variable
ABORT variable
ABORT variable
BIT_OUT variable
BREAK_POINT variable
BREAK_POINT variable
BREAK_POINT variable
BREAK_POINT variable
CNDBIT variable
IBSET function
IO$M_ABORT constant
IO$M_BRDCST constant
IO$M_LPBEXT constant
IO$M_LPBINT constant
Query: 4   | Command: FIND *B*T | Forward
```

Wie man sieht, enthält diese Liste durchaus Einträge, die man (in Kenntnis des zu untersuchenden Quellcodes) vielleicht nicht unbedingt erwartet hätte (beispielsweise `IO$M_ABORT`). Diese Symbole sind das Ergebnis einer Reihe von Includes, die auch ursächlich für die teilweise extrem anmutenden Zeilennummern verantwortlich zeichnen.

Wie das obiege Beispiel zeigt, werden verschiedene Zugriffsformen auf Variablen unterschieden. Dies sind im einzelnen:

- Vereinbarungen, gekennzeichnet durch `PRIMARY`,
- Lesezugriffe, `READ` sowie
- Schreibzugriffe, `WRITE`.

Ähnliche Unterscheidungen treffen auch auf Unterprogrammaufrufe zu, wie folgendes Beispiel anhand von

```
LSE Command> FIND DISASM
```

zeigt:

```
DISASM routine
    DISASM\2            SUBROUTINE or PROGRAM declaration
    EXECUTE\54          call reference
    EXECUTE\54          SUBROUTINE or FUNCTION declaration (hidden)
    UEPEMU\4508         call reference
    UEPEMU\4508         SUBROUTINE or FUNCTION declaration (hidden)
```

In dieser Auflistung verbergen sich nun die eigentliche Vereinbarung der Routine selbst, sowie ihre Aufrufe innerhalb des restlichen Programmes, wobei jedem Aufruf ein weiterer Deklarationsteil zugeordnet ist, der jedoch implizit (verborgen, `hidden`) ist. Bei Unterroutinen können folgende Zugriffsformen unterschieden sowie bei Abfragen gezielt eingesetzt werden:

- Die Prozedur-/Funktionsvereinbarung, `declaration`, auch `PRIMARY` genannte,
- implizite Vereinbarungen beim Aufruf, `declaration (hidden)` und
- die eigentlichen Aufrufe der jeweiligen Routine, `call reference`.

2.3.2 Zusammengesetzte Abfragen

Mit Hilfe zusammengesetzter Abfragen können nun sehr viel komplexere Betrachtungen des vorliegenden Quellcodes vollzogen werden. Sollen beispielsweise alle Stellen des Programms herausgesucht werden, an denen schreibend auf die Variable ABORT zugegriffen wird, kann dies mit Hilfe der folgenden Eingabe geschehen:

```
LSE Command> FIND ABORT AND OCCURRENCE=WRITE
```

Für das Programm E1.FOR liefert diese Query folgende Ausgabe:

```
ABORT variable
    CTRLC_PRESSED\8      write reference
ABORT variable
```

Hierbei wird nur der erste Eintrag automatisch expandiert – alle folgenden müssen explizit mit CTRL/E aufgefächert werden.

Entsprechend läßt sich beispielsweise nach der Vereinbarung der Prozedur DISASM mit Hilfe des folgenden Statements suchen:

```
LSE Command> FIND DISASM AND OCCURRENCE=PRIMARY
```

Kommt statt PRIMARY das Attribut CALL zum Einsatz, liefert die Query alle Positionen zurück, an denen die nämliche Routine aufgerufen wird.

Außer den hiermit vorgestellten Einschränkungen eines Suchvorganges nach Zugriffsformen (lesend, schreibend, vereinbarend, rufend), besteht auch die Möglichkeit, ihn nach Klassen einzuteilen, wobei Klassen in diesem Zusammenhang

- Unterroutinen, ROUTINE,
- Variablen, VARIABLE und
- Konstanten, CONSTANT

sind. So liefert beispielsweise der Befehl FIND SYMBOL=ROUTINE eine Liste aller Unterroutinen zurück, die sich im Quellcode finden. Im vorliegenden Fall ergibt sich hierdurch eine Liste von 323 Symbolen, unter denen auch alle (implizit) verwendeten Systemroutinen, etc. zu finden sind. Durch Verwendung logischer Verknüpfungen, läßt sich diese Suche nun beispielsweise auf alle Aufrufe aller Routinen einschränken:

```
LSE Command> FIND SYMBOL=ROUTINE AND OCCURENCE=CALL
```

Dies liefert für E1.FOR die nunmehr überschaubare Menge von 32 Symbolen zurück. Suchkriterien lassen sich durch Verwendung der Verknüpfungen AND beziehungsweise OR beliebig einschränken, so daß sich in der Praxis eine schier unerschöpfliche Vielfalt an möglichen Abfragen ergibt, die sich genau auf den jeweils vorliegenden Problemtyp zurechtschneiden lassen.

2.3.3 Mehr zu Subroutinen

In den vorangegangenen Abschnitten wurden bereits Queries vorgestellt, mit deren Hilfe herausgefunden werden konnte, welche Routinen aufgerufen werden – beispielsweise liefert ein

```
LSE Command> FIND CALLED_BY EXECUTE
```

eine Liste aller Routinen, die von der Routine EXECUTE aufgerufen werden. Diese Liste weist jedoch eine schwerwiegende Einschränkung auf – sie enthält zwar alle aufgerufenen Routinen, aber nicht die Routinen, die ihrerseits von diesen Routinen aufgerufen werden, usf. Sie hat mithin eine Tiefe von 1, da sie nur die erste Ebene von Subroutinenaufrufen zurückliefert. Um eine hierarchisch strukturierte Liste aller Funktionen, die im Verlauf von EXECUTE aufgerufen werden, zu erhalten, kann das Kommando

```
LSE Command> FIND CALLED_BY (EXECUTE, DEPTH=ALL)
```

verwendet, das folgendes Ergebnis produziert:

```
EXECUTE function calls
   ALU32 function calls
   .  ALU4 function calls
   .  .  IAND function
   .  .  IEOR function
   .  .  IOR function
   .  .  ISHFT function
   .  .  NOT function
   .  .  STAGE1 routine calls
   .  .     IAND function  (See above)
   .  .     IOR function  (See above)
   .  .     NOT function  (See above)
   .  IAND function  (See above)
   .  IOR function  (See above)
   .  ISHFT function  (See above)
   DISASM routine calls
   .  IAND function  (See above)
   .  INDEX function
   .  ISHFT function  (See above)
   .  NTOA routine calls
   .     IAND function  (See above)
   .     ISHFT function  (See above)
   IAND function  (See above)
   IBCLR function
   IBSET function
   IEOR function  (See above)
   IOR function  (See above)
   ISHFT function  (See above)
   NTOA routine  (See above)

Query: 10  | Command: FIND CALLED_BY (EXECUTE, DEPTH=ALL) | Forward
```

Hierbei ist zu bemerken, daß das Argument von DEPTH= beliebige (durchaus auch konkrete) ganzzahlige Werte annehmen kann. In derartigen Fällen wird die sich ergebende Baumstruktur der einzelnen Aufrufe nur bis zu der explizit spezifizierten Tiefe durchsucht.

In manchen Fällen ist auch die umgekehrte Vorgehensweise erwünscht: Es interessiert, welche Routinen eine bestimmte Routine aufrufen, um beispielsweise abschätzen zu können, welche Auswirkungen geplante Änderungen in einer Subroutine nach sich ziehen könnten. Hierfür steht der Befehl FIND CALLING zur Verfügung. Soll beispielsweise untersucht werden, an welchen Stellen die Routine IOR aufgerufen wird, kann dies folgendermaßen geschehen:

```
LSE Command> FIND CALLING IOR
```

Es ergibt sich folgende Ausgabe:

```
IOR function is called by
   ALU32 function
   ALU4 function
   ATOI function
   EXECUTE function
   STAGE1 procedure
```

Anhand dieser Liste kann nun, wie bereits erwähnt, beispielsweise abgeschätzt werden, welche Programmteile von geplanten Änderungen in dieser Routine betroffen sein könnten – eine in der Praxis sehr häufig auftretende Fragestellung.

Eine weitere Hilfe, die der SCA dem Entwickler in Bezug auf Unterroutinen bietet, ist die Überprüfung von Modulen auf Konsistenz, was Subroutinenaufrufe betrifft. Beispielsweise kann geprüft werden, ob die Routine EXECUTE andere Routinen fehlerhaft aufruft, d.h. mit unkorrekter Anzahl von Argumenten oder falschem Argumententyp, etc. Hierzu dient der Befehl CHECK CALLS:

```
LSE Command> CHECK CALLS EXECUTE
```

Dieses Kommando ist in neueren SCA-Versionen entfallen und sollte durch

```
LSE Command> INSPECT (EXECUTE AND SYMBOL=ROUTINE)/CHARACTERISTICS=ALL
```

ersetzt werden. Sind alle untersuchten Aufrufe in Ordnung, meldet der SCA dies durch Ausgabe von

```
No errors found based on your selection criteria
```

Das INSPECT-Kommando bietet folgende Möglichkeiten zur Untersuchung des Quellcodes:

CHARACTERISTICS=TYPE: Hiermit werden die verlangten Typen einer Unterroutine mit den Typen der tatsächlich übergebenen Argumente verglichen.

CHARACTERISTICS=USAGE: Sucht nach Variablen, auf die entweder lesend, aber nicht initialisierend zugegriffen wird, sowie nach Variablen, die zwar beschrieben, aber nie gelesen werden.

CHARACTERISTICS=IMPLICIT_DECLARATIONS: Es wird nach dem Auftreten nicht explizit deklarierter Symbole gesucht.

CHARACTERISTICS=UNIQUENESS: Wird ein Symbol mehrfach deklariert, kann hiermit die Widerspruchsfreiheit solcher Deklarationen überprüft werden.

CHARACTERISTICS=UNUSED_SYMBOLS: Vereinbarte, aber nie verwendete Symbole werden hiermit aufgelistet.

CHARACTERISTICS=ALL: Dies stellt den Defaultwert für das INSPECT-Kommando dar: Es wird nach allen fünf vorangegangenen Kriterien gesucht.

2.3.4 Sprachspezifische Checks

Die hier beschriebene Testmöglichkeit fällt in gewisser Weise in die Gruppe der zu Unterroutinen gehörenden Abfragen, da sie sich auf den gesamten Quellcode bezieht und einen routinenübergreifenden Konsistenzcheck ermöglicht. Das Kommando

```
LSE Command> CHECK LANGUAGE FORTRAN/DEFINITIONS
```

untersucht den gesamten zur Verfügung stehenden Quellcode nach folgenden Kriterien:

- Undefinierte Symbole,
- nichtreferenzierte Symbole,
- inkonsistente Symbolvereinbarungen.

2.3.5 Typen

Ein weiterer großer Bereich, der Behandlung verdient, ist die Frage nach dem Typ einzelner Variablen oder Konstanten. Soll beispielsweise der Typ der Variablen `SCRATCH` herausgefunden werden, so kann dies mit Hilfe der Abfrage

```
LSE Command> FIND TYPING SCRATCH
```

erreicht werden:

```
SCRATCH variable is typed by
   INTEGER scalar type
```

Nach Expansion ergibt sich folgende Ausgabe:

```
SCRATCH variable is typed by
    EXECUTE\10           variable declaration
   INTEGER scalar type
    EXECUTE\10           reference
```

Umgekehrt können selbstverständlich auch alle Variablen oder Konstanten eines bestimmten Typs gesucht werden. Sollen beispielsweise alle als `LOGICAL` vereinbarten Variablen aufgelistet werden, kann dies durch

```
LSE Command> FIND TYPED_BY LOGICAL
```

geschehen:

```
LOGICAL scalar type types
   array component
   array component
   array component
   array component
   array component
   array component
   return value
   return value
   ABORT variable
   ABORT variable
```

```
   ABORT variable
   ABORT variable
   ABORT variable
Query: 16  | Command: FIND TYPED_BY LOGICAL | Forward
```

2.3.6 Wer enthält was?

In diesem Abschnitt sollen die Suchbegriffseinschränkungen mit Hilfe von CONTAINED_BY behandelt werden. Als einführendes Beispiel möge die Query

```
LSE Command> FIND CONTAINED_BY EXECUTE
```

dienen, die folgende Ausgabe nach sich zieht:

```
EXECUTE function contains
   array
   return value
   return value
   return value
   return value
   return value
   return value
   return value
   return value
   10 label
   20 label
   30 label
   40 label
Query: 17  | Command: FIND CONTAINED_BY EXECUTE | Forward
```

Diese Liste setzt sich aus einer Aufzählung *aller* in EXECUTE vorgefundenen Symbole zusammen. Hierunter fallen nicht nur Variablen, Konstanten, COMMON-Blöcke und ähnliche Strukturen, sondern auch Rückgabewerte aufgerufener Funktionen, die aufgerufenen Funktionen selbst, Labels, usf. Es handelt sich also wirklich um eine Liste *aller* Dinge, die sich innerhalb von EXECUTE finden.

Sollen nun alle in EXECUTE enthaltenen Labels gesucht werden, kann dies wie folgt erreicht werden:

```
LSE Command> FIND CONTAINED_BY EXECUTE AND SYMBOL=LABEL
```

Als Ausgabe ergibt sich hierdurch:

```
10 label
    EXECUTE\59           read reference
    EXECUTE\61           label declaration
20 label
30 label
40 label
50 label
60 label
70 label

Query: 18  | Command: FIND CONTAINED_BY EXECUTE AND SYMBOL=LABEL | Forward
```

2.3.7 Query-Buffer

Nach dem bereits gesagten stellt sich mitunter die Frage, was eigentlich mit bereits abgesetzten Queries geschieht – diese Überlegung erscheint umso interessanter, als manche Suchvorgänge doch ein erkleckliches Quantum an Zeit benötigen, so daß es von Vorteil wäre, könnte man ihre Ergebnisse wieder und wieder verwenden, ohne den Suchvorgang an sich wiederholen zu müssen.

Alle im Verlauf einer SCA-Session bearbeiteten Queries werden in einem Buffer, dem sogenannten Query-Buffer, gespeichert, dessen Inhalt sich mit Hilfe des Kommandos SHOW QUERY anzeigen läßt.

```
    Name      Query expression         Description

    1         CALLED_BY UEPEMU         (none)
    2         SCRATCH                  (none)
    3         SCRATCH                  (none)
    4         *B*T                     (none)
    5         ABORT AND OCCURR=WRITE   (none)
    6         DISASM                   (none)
    7         SYMBOL=ROUTINE           (none)
    8         SYMBOL=ROUTINE AND OCCURRENCE=CALL
                                       (none)
    9         SYMBOL=ROUTINE AND OCCURRENCE=CALL
                                       (none)
    10        CALLED_BY (EXECUTE, DEPTH=ALL)
                                       (none)
    11        CALLING IOR              (none)
    13                                 (none)
    14        TYPING SCRATCH           (none)
    15        TYPING SCRATCH           (none)
    16        TYPED_BY LOGICAL         (none)
    17        CONTAINED_BY EXECUTE     (none)
(*) 18        CONTAINED_BY EXECUTE AND SYMBOL=LABEL
                                       (none)
```

Eine beliebige Query aus dieser Liste kann nun mit Hilfe des Kommandos GOTO QUERY rezykliert werden, ohne sie erneut ausführen zu müssen. So zeigt beispielsweise

```
LSE Command> GOTO QUERY 2
```

das Ergebnis der Query Nummer 2 erneut an. Außer der Möglichkeit, eine bereits abgearbeitete Abfrage direkt, d.h. durch Angabe ihrer laufenden Nummer, zu adressieren, kann auch relativ vorgegangen werden. Hierfür stehen die Kommandos NEXT QUERY beziehungsweise PREVIOUS QUERY zur Verfügung, mit deren Hilfe – ausgehend von der aktuellen Abfrage, in obiger Liste durch ein vorangestelltes (*) markiert – eine Abfrage vor- resp. zurückgesprungen werden kann.

Ebenso kann aus der Liste der Queries ersatzlos gelöscht werden, indem das Kommando DELETE QUERY zur Anwendung kommt. So löscht beispielsweise ein

```
LSE Command> DELETE QUERY 3
```

die Abfrage mit der Nummer 3 aus dem Query-Buffer, so daß auf sie in allen nachfolgenden Schritten nicht mehr Bezug genommen werden kann.

2.3.8 Bewegen im Quellcode

Allen vorangegangenen Beispielen ist eines gemein: Sie greifen über Suchkommandos auf den Quelltext zu – zuerst wird nach einem bestimmten Kriterium gesucht, danach kann mit CTRL/G oder GOTO SOURCE an die korrespondierende Stelle im Quellcode gesprungen werden, um ihn einer eingehenderen Untersuchung zu unterziehen.

Ein wichtiger Punkt, der sicherlich Beachtung verdient, ist die Frage nach der Vereinbarung einer Variablen, etc. Sicherlich kann dieses Problem mit Hilfe einer entsprechenden Query gelöst werden, was der Eleganz entbehrt, sofern man sich im Sourcecode-Fenster selbst befindet und gerne wissen möchte, an welcher Stelle die Variable deklariert wird, auf der sich gerade der Cursor (und mit ihm hoffentlich auch die Aufmerksamkeit des Benutzers) befindet. Für dieses ganz spezielle Problem stellt der SCA das Kommando

```
LSE Command> GOTO DECLARATION/INDICATED/PRIMARY
```

zur Verfügung, das – zu einem einfachen CTRL/D abgekürzt – zu einem nahezu unersetzlichen Hilfsmittel beim interaktiven Verstehen eines unbekannten Programmes werden kann. Mit seiner Hilfe kann mit einer einzigen Tastenkombination sofort zu der Stelle im Quellcode gesprungen werden, an der die Variable, auf der der Cursor stand, deklariert wird.

3 Der PCA

Der PCA dient einerseits zur Untersuchung eines Programmes hinsichtlich seines Laufzeitverhaltens und seines Ressourcenbedarfs (Performance-Analyser), während er andererseits die Möglichkeit bietet, die Überdeckung, d.h. Ausführung beliebiger Programmteile näher zu beleuchten (Coverage-Analyser).

3.1 Grundlagen

Der Einsatz des PCA gliedert sich in zwei Schritte:

1. Ein Sammellauf, währenddessen alle interessierenden Parameter und das Verhalten des zu untersuchenden Programms mitprotokolliert werden sowie
2. ein Auswertungslauf, in dessen Verlauf die im ersten Schritt gesammelten Daten näher analysiert werden können.

Vor dem eigentlichen Datensammeln muß das zu untersuchende Programm mit /DEBUG übersetzt und gelinkt werden, um alle für den PCA notwendigen Hilfsroutinen und Tabellen in das Executable einzubinden. Darüberhinaus muß ein Logical umdefiniert werden, um zu verhindern, daß das Programm unter der Kontrolle des Debuggers läuft, sondern an seiner Statt durch den Collector, den PCAC, gesteuert wird. Die folgenden Abschnitte verwenden das bereits vorgestellte FORTRAN-Programm E1.FOR, das folgendermaßen für eine Untersuchung mit Hilfe des PCA vorbereitet wird:

```
$ FORTRAN/DEBUG/NOOP E1
$ LINK/DEBUG E1
$ DEFINE LIB$DEBUG PCA$COLLECTOR
$ DEFINE PCA$DECW$DISPLAY " "
```

Das Logical LIB$DEBUG zeigt auf die Bibliothek, die als Debug-Library verwendet werden soll. In diesem Fall wird der PCA-Collector als „Debugger“ eingesetzt. Um zu verhindern, daß der PCACunter seiner DECwindows-Oberfläche läuft, die der Kommandooberfläche durchaus unterlegen ist, wird das Logical PCA$DECW$DISPLAY umdefiniert.

Nach Starten des Programmes mit `$ RUN E1`, meldet sich der PCAC mit folgendem Prompt:

```
        DIGITAL PCA Collector Version V4.6-4


PCAC>
```

Ab diesem Zeitpunkt obliegt die Steuerung des Programmlaufes sowie des Datensammelns dem PCAC. Sofern nicht anders vorgegeben, werden die während des folgenden Laufes gesammelten Meßdaten in einem File mit der Extension .PCA abgelegt (in diesem Fall also einer Datei E1.PCA). Dieses Meßdatenfile kann unter Umständen sehr groß werden, so daß es sich empfiehlt, es nicht in Verzeichnissen abzulegen, die mit Diskqoutas eingeschränkt sind! Mit Hilfe des Kommandos

```
PCAC> SET DATAFILE <filename>
```

kann das gewünschte Ausgabefile festgelegt werden, ohne dieses Kommando wird der Name des zu untersuchenden Programmes mit der Extension .PCA als Defaultfilename verwendet.

3.2 Nodespecs

Die grundlegende Frage bei der Untersuchung eines Programmes hinsichtlich seines Laufzeit- und Ressourcenverhalten ist, in welcher Auflösung das Datensammeln geschehen soll. Welche Zeitschritte sind als kleinste Einheit zu verwenden? Welche Programmabschnitte sollen wie aufgelöst werden, etc.

Hierzu werden sogenannte *Nodespecs* (Knotenspezifikationen) eingesetzt, mit deren Hilfe festgelegt wird, wie das Programm für den Kollektorlauf segmentiert wird. Der allgemeine Aufbau einer solchen Spezifikation hat folgende Gestalt:

<grob> [BY <fein>]

Hierbei legt grob die zu betrachtende Grobstruktur fest, während durch die Angabe des optionalen Wertes <fein> diese weiter spezifiziert werden kann. Nodespecs können folgende Werte annehmen[7]:

`LINE [pathname\] %LINE n`: Nur Zeile n wird gesampelt.

`LINE [pathname\] %LINE n BY [n] BYTES`: Der Code, aus dem sich die selektierte Zeile zusammensetzt, wird byteweise untersucht.

`LINE [pathname\] %LINE n BY CODEPATH`: S.o.

`MODULE pathname`: Nur das ausgewählte Modul wird untersucht.

`MODULE pathname BY [n] LINES`: Zeilenweises Sampling innerhalb des gewünschten Moduls, wobei durch Angabe des optionales Parameters n jeweils n Zeilen zu einer Untersuchungseinheit zusammengefaßt werden können.

`MODULE pathname BY [n] BYTES`: Byteweise Segmentierung.

`MODULE pathname BY ROUTINE...`

[7] Anhang A enthält eine Beschreibung des allgemeinen Aufbaus von Pfadangaben im Zusammenhang mit dem PCA.

`MODULE pathname BY CODEPATH...`

`MODULE pathname BY VINSTRUCTION`: Vektorinstruktionen als Feinstruktur.

`PROGRAM_ADDRESS BY MODULE`: Hierbei wird das gesamte Programm untersucht, was wohl in den meisten Fällen das gewünschte Verhalten des Kollektors darstellen dürfte. Feinstruktur stellen in diesem Falle eventuell (falls spezifiziert) die Module dar, aus denen sich das betreffende Programm zusammensetzt.

`PROGRAM_ADDRESS BY [n] LINES`: Zeilenweises Datensammeln.

`PROGRAM_ADDRESS BY [n] BYTES`: Byteweise.

`PROGRAM_ADDRESS BY ROUTINE...`

`PROGRAM_ADDRESS BY CODEPATH...`

`PROGRAM_ADDRESS BY VINSTRUCTION...`

`ROUTINE pathname`: Untersuchungseinheit ist hierbei eine einzelne Routine des vorliegenden Programmes.

`ROUTINE pathname BY [n] LINES...`

`ROUTINE pathname BY [n] BYTES...`

`ROUTINE pathname BY ROUTINE...`

`ROUTINE pathname BY CODEPATH...`

`ROUTINE pathname BY VINSTRUCTION...`

Um beispielsweise das Programm `E1.FOR` auf Zeilenebene zu untersuchen, ist das Kommando

```
PCAC> SET COUNTER PROGRAM BY LINE
```

zu verwenden.

In manchen Fällen sind die Möglichkeiten zur Segmentierung eines zu untersuchenden Programmes anhand der obigen Nodespecs nicht ausreichend – sei es, daß sie zu fein sind, sei es, daß sie zu grob sind. Um auch Spezialfälle erfassen zu können, existiert neben den bereits genannten Arten der Aufteilung der Meßwerte die Möglichkeit, sogenannte *Eventmarker* zu verwenden. Ein Eventmarker stellt einen Breakpoint dar, bei dessen Erreichen alle bis dahin gesammelten Meßwerte in das Datenfile geschrieben werden. Hiermit kann ein Programm vollkommen wahlfrei in Blöcke unterteilt werden, die nicht unbedingt herkömmlichen Strukturelementen entsprechen müssen. Beispielsweise können mit Hilfe dieser Technik bestimmte Bereiche des Programmes mit hoher Auflösung abgetastet werden, während andere, minder interessante, nur grob in die Meßdaten einfließen. Beispielsweise setzt das Kommando

```
PCAC> SET EVENT CALCULATE LINE %LINE 110
```

einen Eventmarker des Namens `CALCULATE` an die Stelle der Zeile 110.

3.3 Meßwerttypen

Während das zu untersuchende Programm ausgeführt wird, sammelt der Kollektor eine Reihe von Daten, indem er in festen Zeitabständen Δt eine Reihe von Zählern ausliest, mit deren Hilfe der Verbrauch gewisser Ressourcen mitprotokolliert wird[8] Protokolliert werden folgende Werte:

PC sampling: Der Stand des Programmzählers wird mitprotokolliert – dies dient als grobes Maß für die verbrauchte Realzeit.

Page fault sampling: Ebenso wird die Anzahl an Page faults pro Programmsegment gespeichert.

System services sampling: Diese Sorte Meßpunkt speichert die Anzahl von System service-Aufrufen pro Segment und Zeiteinheit.

IO-data sampling: Mitprotokolliert wird auch die Anzahl der ausgeführten IO-Zugriffe.

Execution counts: Dieser Zähler entspricht der Anzahl von Aufrufen bestimmter Teile des zu untersuchenden Programms.

Coverage data: Hiermit werden Daten über Aufrufe von Routinen gesammelt.

3.4 Die Steuerung des PCAC

Vor dem eigentlichen Meßlauf muß dem Kollektor mitgeteilt werden, welche Daten zu erfassen sind. Dies geschieht mit Hilfe des SET-Kommandos, das folgende Parameter setzen kann:

PC_SAMPLING: Bei Auswahl dieser Option wird die tatsächlich benötigte Zeit für die Ausführung der einzelnen Programmteile mitprotokolliert. Diese setzt sich aus der eigentlichen CPU-Zeit, der IO-Zeit, etc. zusammen. PC_SAMPLING ist die Defaulteinstellung des PCAC. Problematisch an dieser Meßwertform ist die Tatsache, daß die verbrauchte Systemzeit stark von Faktoren abhängig ist, die unter Umständen außerhalb des Verantwortungsbereiches des zu untersuchenden Programmes liegen, wie beispielsweise der Pagefaultrate des Systems, Wartezeiten bei IO-Operationen, etc.

CPU_SAMPLING: Im Gegensatz zu PC_SAMPLING wird in diesem Fall nicht die verbrauchte Systemzeit, sondern die wirklich benötigte CPU-Zeit mitprotokolliert.

PAGE_FAULTS: Hierbei wird lediglich die Anzahl der durchgeführten Pagefaults pro Programmsegment mitgeschrieben.

Coverage: Diese Option ermöglicht die Untersuchung von Fragen der Art: „Wird eine bestimmte Funktion überhaupt aufgerufen?“, etc. Beispielsweise untersucht der PCAC das Verhalten des Programmes nach Eingabe des Kommandos

```
PCAC> SET COVERAGE PROGRAM BY ROUTINE
```

dahingehend, welche Routinen überhaupt aufgerufen werden.

SERVICES: Die Anzahl von Systemservice-Aufrufen wird gesampled.

COUNTERS <nodespec>: Diese Form des SET-Kommandos gestattet es, Messung auf den Kontext <nodespec> einzuschränken.

[8] $\Delta t = 10$ms. Dieser Defaultwert kann durch den Qualifier /INTERVAL: modifiziert werden.

EVENT: Hiermit können Eventmarker gesetzt werden.

Mit Hilfe des SET-Kommandos gesetzte Einstellungen können durch das Kommando CANCEL zurückgesetzt werden. Sollen alle bisherigen Einstellungen gelöscht werden, kann dies durch

```
PCAC> CANCEL ALL
```

geschehen.

3.5 Beispiel

Das folgende Beispiel zeigt einen einfachen Analyselauf über das Programm E1.EXE, der gemäß Abschnitt 3.1 übersetzt und gelinkt wurde:

```
$ RUN E1

        DIGITAL PCA Collector Version V4.6-4


PCAC> GO
%PCA-I-DEFDATFIL, set datafile required in this context, creating '[]E1.PCA'
%PCA-I-BEGINCOL, data collection begins
%PCA-I-DATADEFPC, defaulting to collecting PC sampling data
U-EP-EMULATOR - VERSION 1.0
MAXMEM : 000003E8

MAIN> LOAD TEST
LOAD> READING FILE : TEST.BIN
LOAD> LOWEST ADDRESS : 00000000 ---> HIGHEST ADDRESS : 0000000A
MAIN> RUN
RUN > HALT !
RUN >       196612 INSTRUCTIONS EXECUTED.
MAIN> EXIT
MAIN> DO YOU REALLY WANT TO QUIT ? (Y/N) Y
     196612 INSTRUCTIONS EXECUTED.
%PCA-I-ENDCOL, data collection ends
$
```

Da keine näheren Angaben über die Art der Meßdatenerfassung gemacht wurden, geht der PCAC von der Einstellung PC_SAMPLING aus – weiterhin wird automatisch das Analysefile []E1.PCA generiert, da kein SET DATAFILE-Kommando eingegeben wurde. Die durch einen solchen Programmlauf unter der Kontrolle des PCAC gesammelten Daten können im nun folgenden Schritt mit Hilfe des Analysators, des PCA, eingehend untersucht werden:

3.6 Die Analyse der Meßdaten

Wie auch der PCAC, verfügt der PCA über zwei Oberflächen – eine Kommandooberfläche und eine DECwindows-Oberfläche, die weitgehend selbsterklärend ist und für einfache Analysen sicherlich ausreicht. Soll die DECwindows-Oberfläche des PCA verwendet werden, ist darauf zu achten, daß das Logical PCA$DECW$DISPLAY nicht definiert ist, bzw. auf den aktuellen Desktop zeigt. Abbildung 3 zeigt den Zustand des PCA nach Start durch das Kommando PCA <filename>, wobei <filename> im vorliegenden Fall den Wert E1.PCA besitzt. Entsprechend zeigt Figur 4 den Zustand nach Ausführung einer einfachen Query. Nach Anklicken von VIEW > ergibt sich eine Anzeige wie in Abbildung 5 dargestellt.

Unter der Kommandooberfläche des PCA sind die beiden Befehle PLOT beziehungsweise TABULATE besonders hervorzuheben, da mit ihrer Hilfe Diagramme der in Frage

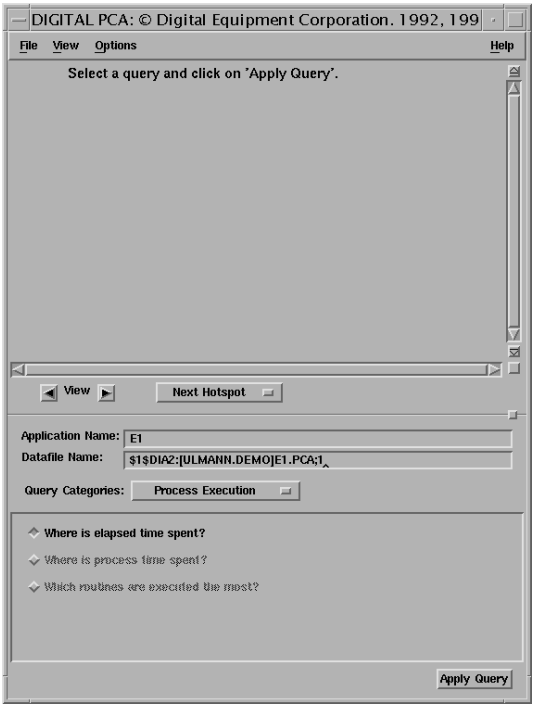

Abbildung 3: Die `DECwindows`-Oberfläche des `PCA`

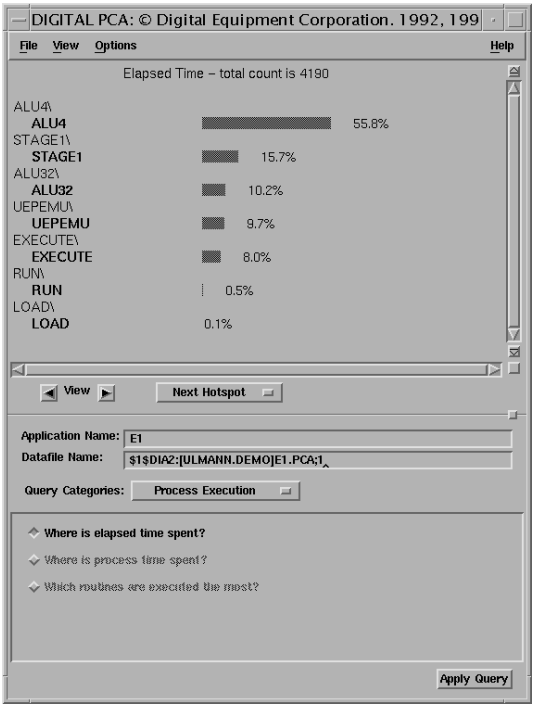

Abbildung 4: Der `PCA` nach einer einfachen Query

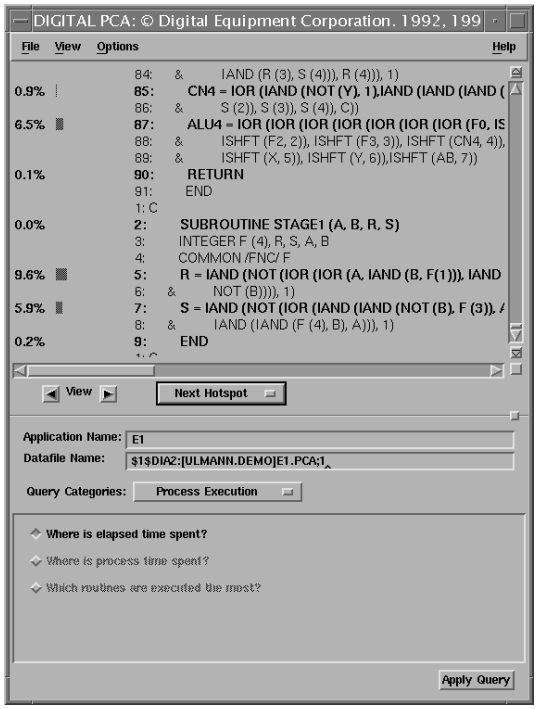

Abbildung 5: Detailliertere Auswertung im PCA

kommenden Meßdaten angefertigt werden können. Der generelle Aufbau der genannten Kommandos ist

{PLOT|TABULATE} [/QUALIFIER] <nodespec>

, wobei unter anderem folgende Qualifier möglich sind:

/IO_SERVICE: Plotted/tabuliert die Anzahl der Aufrufe von IO-Routinen.

/PHYSICAL_IO: Die Menge physikalischer IOs wird dargestellt.

/PC_SAMPLING: Verbrauchte Systemzeit wird abgetragen.

/READ_COUNT: Dargestellt wird die Anzahl physikalischer Lesezugriffe.

/WRITE_COUNT: S.o., jedoch für physikalische Schreibzugriffe.

/COUNTER: In Verbindung mit entsprechenden SET COUNTERS-Kommandos können hiermit die solchemaßen gesammelten Daten visualisiert werden.

/ASCENDING: Die Ausgabe wird in aufsteigender Reihenfolge nach den jeweiligen Zählerständen sortiert.

/DESCENDING: S.o., jedoch absteigend sortiert.

/MAXIMUM=n: Hierdurch kann die Ausgabe von Zählern, die den Wert n überschreiten, verhindert werden. Durch NOMAXIMUM wird eine solche Einstellung unterdrückt.

/MINIMUM=n: S.o.

/NOSORT: Die Ausgabe erfolgt unsortiert.

Für die wie oben beschrieben gesammelten Daten eines E1-Programmlaufes erzeugt beispielsweise das Kommando PLOT/PC_SAMPLING folgende Ausgabe:

```
                DIGITAL Performance and Coverage Analyzer              Page 1

        Program Counter Sampling Data (4190 data points total) - "*"

Bucket Name          +----+----+----+----+----+----+----+----+----+----+
ALU4\                |
 ALU4 . . . . . . . .|************************************************  55.8%
STAGE1\              |
 STAGE1 . . . . . . .|**************                                    15.7%
ALU32\               |
 ALU32  . . . . . . .|*********                                         10.2%
UEPEMU\              |
 UEPEMU . . . . . . .|*********                                          9.7%
EXECUTE\             |
 EXECUTE  . . . . . .|*******                                            8.0%
RUN\                 |
 RUN  . . . . . . . .|                                                   0.5%
LOAD\                |
 LOAD . . . . . . . .|                                                   0.1%
ATOI\                |
 ATOI . . . . . . . .|                                                   0.0%
CTRLC_PRESSED\       |
 CTRLC_PRESSED  . . .|                                                   0.0%
DEPOSIT\             |
 DEPOSIT  . . . . . .|                                                   0.0%
DEPREG\              |
 DEPREG . . . . . . .|                                                   0.0%
DISASM\              |
 DISASM . . . . . . .|                                                   0.0%
DUMP\                |
 DUMP . . . . . . . .|                                                   0.0%
ENABLE_CTRLC\        |
 ENABLE_CTRLC . . . .|                                                   0.0%
HELP\                |
 HELP . . . . . . . .|                                                   0.0%
NTOA\                |
 NTOA . . . . . . . .|                                                   0.0%
RDMP\                |
 RDMP . . . . . . . .|                                                   0.0%
RESTART\             |
 RESTART  . . . . . .|                                                   0.0%
SAVE\                |
 SAVE . . . . . . . .|                                                   0.0%
SNAP\                |
 SNAP . . . . . . . .|                                                   0.0%
                     |
                     +----+----+----+----+----+----+----+----+----+----+
```

Entsprechend erzeugt das Kommando TABULATE/PC_SAMPLING folgendes Bild:

```
                DIGITAL Performance and Coverage Analyzer              Page 1

        Program Counter Sampling Data (4190 data points total) - "*"

                             Data             95% Conf
Bucket Name                  Count  Percent   Interval
ALU4\
 ALU4 . . . . . . . . . .    2340   55.8% +/-  1.5%
STAGE1\
 STAGE1 . . . . . . . . .     657   15.7% +/-  1.1%
ALU32\
 ALU32  . . . . . . . . .     426   10.2% +/-  0.9%
UEPEMU\
 UEPEMU . . . . . . . . .     405    9.7% +/-  0.9%
```

```
EXECUTE\
 EXECUTE  . . . . . .   337    8.0% +/-  0.8%
RUN\
 RUN  . . . . . .  .    20    0.5% +/-  0.2%
LOAD\
 LOAD . . . . . .  .     5    0.1% +/-  0.1%
ATOI\
 ATOI . . . . . .  .     0    0.0%
CTRLC_PRESSED\
 CTRLC_PRESSED  . . . . .     0    0.0%
DEPOSIT\
 DEPOSIT  . . . . . .     0    0.0%
DEPREG\
 DEPREG . . . . . .  .     0    0.0%
DISASM\
 DISASM . . . . . .  .     0    0.0%
DUMP\
 DUMP . . . . . .  .  .     0    0.0%
ENABLE_CTRLC\
 ENABLE_CTRLC . . . . . .     0    0.0%
HELP\
 HELP . . . . . .  .  .     0    0.0%
NTOA\
 NTOA . . . . . .  .  .     0    0.0%
RDMP\
 RDMP . . . . . .  .  .     0    0.0%
RESTART\
 RESTART  . . . . . .     0    0.0%
SAVE\
 SAVE . . . . . .  .  .     0    0.0%
SNAP\
 SNAP . . . . . .  .  .     0    0.0%
```

Da Listen, wie die beiden Beispiele des PLOT- bzw. TABULATE-Kommandos bereits andeuten, unter Umständen sehr umfangreich werden können, bietet der PCA die Möglichkeit, innerhalb einer solchen Ausgabe nach Marken zu suchen. Soll beispielsweise der Ressourcenverbrauch der Routine RDMP bestimmt werden, kann mit Hilfe des Kommandos

```
PCAC> FIND RDMP
```

an die entsprechende Stelle in der vorliegenden Ausgabeliste gesprungen werden, die zusätzlich durch einen vorangestellten Pfeil markiert wird.

Das Kommando

```
PCAC> PLOT PROGRAM BY LINE
```

erzeugt eine Auflistung des Programms mit zeilenweiser Segmentierung unter Angabe des jeweiligen Ressourcenverbrauchs.

Mit Hilfe des Kommandos NEXT kann an die nächstgelegene Stelle der Ausgabe (beispielsweise eine PLOT PROGRAM BY LINE) mit signifikantem Ressourcenverbrauch gesprungen werden. Gezieltes Blättern in der Ausgabe ist durch PAGE NEXT beziehungsweise PAGE PREVIOUS möglich. Der Befehl PAGE SUMMARY zeigt eine Seite mit allgemeinen Informationen an – beispielsweise:

```
                    DIGITAL Performance and Coverage Analyzer            Page 330

          Program Counter Sampling Data (4190 data points total) - "*"

                  PCA Version V4.6-4       27-FEB-2000 01:27:56

PLOT Command Summary Information:
```

```
Number of buckets tallied:                              809

Program Counter Sampling Data - "*"

Data count in largest defined bucket:                  1097    26.2%
Data count in all defined buckets:                     4190   100.0%
Data count not in defined buckets:                        0     0.0%
Portion of above count in P0 space:                       0     0.0%
Number of PC values in P1 space:                          0     0.0%
Number of PC values in system space:                      0     0.0%
Data points failing /STACK_DEPTH or /MAIN_IMAGE:          0     0.0%

Total number of data values collected:                 4190   100.0%

Command qualifiers and parameters used:
  Qualifiers:
    /PC_SAMPLING /NOSORT /NOMINIMUM /NOMAXIMUM
    /NOCUMULATIVE /SOURCE /ZEROS /NOSCALE /NOCREATOR_PC
    /NOPATHNAME /NOCHAIN_NAME /WRAP /NOPARENT_TASK /NOKEEP /NOTREE
    /FILL=("*","0","x","@",":","#","/","+")
    /NOSTACK_DEPTH /MAIN_IMAGE
  Node specifications:
    PROGRAM_ADDRESS BY LINE

No filters are defined
```

Um das Resultat des letzten PLOT- oder TABULATE-Kommandos in eine Datei zu schreiben, kann das Kommando FILE <name> verwendet werden, wobei <name> den gewünschten Filenamen für die Schreiboperation spezifiziert. Analog hierzu kann der Befehl PRINT eingesetzt werden, wobei als Ausgabedevice das Logical SYS$PRINT Verwendung findet.

A Pfadangaben

```
a-char ::==
    'a' |
    'A'

alpha-char ::==
    'a' .. 'z' |
    'A'..'Z'

b-char ::==
    'b' |
    'B'

digit-char ::==
    '0' .. '9'

dot-char ::==
    '.'

double-colon-char ::==
    ':'

double-quote-char ::==
    '"'

e-char ::==
    'e' |
    'E'

i-char ::==
    'i' |
```

```
    'I'

l-char ::==
    'l' |
    'L'

m-char ::==
    'm' |
    'M'

n-char ::==
    'n' |
    'N'

percent-char ::==
    '%'

single-quote-char ::==
    '''

space-char ::==
    space |
    horizontal-tab

special-char ::==
    '^' |
    '~' |
    '|' |
    '#' |
    '$' |
    '-' |
    '=' |
    '&' |
    '+' |
    '<' |
    '>' |
    '*' |
    '_'

zero-char ::==
    '0'

operator-char ::==
    space-char |
    '[' |
    ']' |
    ',' |
    '(' |
    ')' |
    '/' |
    '!' |
    '\'

quote-char ::=
    single-quote-char |
    double-quote-char

separator-char ::==
    '\' |
    dot-char (Ada only)

terminator-char ::==
    space-char |
    carriage-return |
    '[' |
    ']' |
    ',' |
```

```
    '(' |
    ')' |
    '/' |
    '!'

double-quoted-char ::==
    alpha-char |
    digit-char |
    dot-char |
    double-colon-char |
    operator-char |
    percent-char |
    single-quote-char |
    special-char

single-quoted-char ::==
    alpha-char |
    digit-char |
    double-colon-char |
    double-quote-char |
    dot-char |
    operator-char |
    percent-char |
    special-char

unquoted-char ::==
    alpha-char |
    digit-char |
    dot-char (not Ada) |
    double-quote-char |
    percent-char |
    single-quote-char |
    special-char

double-quote-quote ::==
    double-quote-char double-quote-char

double-quote-sequence ::==
    double-quoted-char |
    double-quote-quote

double-quoted-identifier ::==
    double-quote-char double-quote-sequence [{double-quoted-sequence}]
    double-quote-char

single-quote-quote ::==
    single-quote-char single-quote-char

single-quote-sequence ::==
    single-quoted-char |
    single-quote-quote

single-quoted-identifier ::==
    single-quote-char single-quote-sequence [{single-quoted-sequence}]
    single-quote-char

double-colon-colon ::==
    double-colon-char double-colon-char

unquoted-identifier ::==
    {[ [{unquoted-char}] double-colon-colon]}
    unquoted-char [{unquoted-char}]

quoted-identifier ::==
    double-quoted-indentifier |
    single-quoted-identifer
```

```
token-identifier ::==
    quoted-identifier | unquoted-identifier

label-identifier ::==
     percent l-char a-char [b-char e-char l-char]
     space-char [{space-char}]
     token-identifer

line-identifier ::==
     percent l-char i-char [n-char e-char]
     space-char [{space-char}]
     [{'0'}] digit-char {digit-char}
     [dot [{'0'}] digit-char {digit-char}]

name-identifier ::==
     percent n-char a-char [m-char e-char]
     space-char [{space-char}]
     token-identifer

identifier ::==
     line-identifier |
     label-identifier |
     name-identifier |
     token-identifier

path-name :==
     [{identifier separator-char}] identifier terminator-char
```

The following BNF operator definitions are used:

```
::==                          assignment operator.
|                             exclusive OR operator.
[]                            optional operator.
..                            range operator.
{}                            repeat operator.
```