PEARL Testsystem

P.-J. Brunner, K. Meffert, H. Windauer

Zusammenfassung

Dieser Artikel beschreibt ein sprachbezogenes Testsystem für PEARL. Für den Einzeltest von Programmkomponenten stehen komfortable Display-, Trace- und Unterbrechungsmöglichkeiten (Haltepunkte) zur Verfügung. Darüberhinaus unterstützt das System auch den Integrations- und Gesamttest durch die Möglichkeiten der Diagnose und Veränderung der Zustände von Tasks und Synchronisationsvariablen sowie durch Schnittstellen zur Simulation der Prozeßperipherie und durch Simulation der Zeitachse. Das Testsystem ist portabel in PEARL programmiert. Bisher ist es auf Rechnern vom Typ HP 3000, NORD 100 und Siemens 300-Serie installiert.

Summary

In this paper a PEARL debugger is presented. This debugger allows to perform the module test by means of comfortable display, trace and breakpoint possibilities. In addition, integration and system test is supported by possibilities for diagnosis and change of states of tasks and synchronization objects. Interfaces are offered to simulate process peripheral devices. The time scale is simulated. The debugger is programmed portable in PEARL. Up to now it is available for HP 3000, NORD 100 and Siemens 300 computers.

1. Einleitung

Kosten-, Zeit- und Sicherheitsgründe haben einen zunehmenden Einsatz höherer Echtzeit-Programmiersprachen bei der Automatisierung technischer Prozesse bewirkt. Dabei ist jedoch deutlich geworden, daß das "Werkzeug höhere Echtzeit-Programmiersprache" allein nicht genügt, die Kosten-, Zeit- und Sicherheitsanforderungen an die Entwicklung der Software ausreichend zu befriedigen. Hierzu muß dieses Werkzeug vielmehr im Rahmen einer Echtzeit-Programmproduktionsumgebung zur Verfügung stehen, die die Software-Entwicklung in allen ihren Phasen von der Anforderungsspezifikation bis zur Wartung und Modifikation unterstützt (vgl. [2], [3]).

Aufgrund der großen Bedeutung der Test- und Integrationsphase für die Automatisierung technischer Systeme, insbesondere sicherheitsrelevanter Systeme, müssen vor allem für diese Phase mächtige Werkzeuge verfügbar sein, die zumindest den folgenden Anforderungen genügen:

1. Der Test erfolgt auf Sprachebene, d.h. der Test soll anhand des Quellprogramms durchführbar sein, ohne das vom Compiler erzeugte Zielcode-Programm zu benutzen.
2. Der Software-Entwickler wird sowohl beim Einzeltest, als auch beim Integrations- und Gesamttest unterstützt.
   Beim Einzeltest wird üblicherweise jede Programm-Komponente, z.B. Prozedur oder Task oder Modul, für sich auf Richtigkeit überprüft. Hierzu sind Hilfsmittel zur Diagnose und Beeinflussung des Ablaufs und der Objekte der Komponenten erforderlich.
   Beim Integrationstest wird die Richtigkeit des Zusammenspiels von Programm-Komponenten überprüft. Hier werden also insbesondere Hilfsmittel zur Diagnose und Beeinflussung der Zustände von Tasks und Synchronisationsobjekten sowie eine Simulation der Zeitachse benötigt.
   Der Gesamttest dient der Überprüfung des gesamten Programms in seiner simulierten oder realen Umgebung. Beide Methoden erfordern Hilfsmittel zur Diagnose und Beeinflussung von zeitabhängigen Situationen sowie der Schnittstellen zu Standard- und Prozeßperipherie (E/A-Vorgänge, Interrupts). Zur Simulation der realen Umgebung ist zumindest eine Simulation der Peripherie-Schnittstellen notwendig.
3. Soweit sinnvoll und möglich, soll der Test auch auf

einem Gastrechner oder Entwicklungssystem, d.h. auf einem anderen, komfortabler oder umfangreicher ausgestatteten Rechner als dem für den Betrieb vorgesehenen Zielrechner durchgeführt werden können (vgl. [1]).
Zusätzlich zu den eben erwähnten Simulationsanforderungen müssen hier die für den Ablauf von PEARL-Programmen notwendigen Voraussetzungen auf dem Gastrechner oder Entwicklungssystem erfüllt werden.

Im folgenden wird ein Testsystem für PEARL vorgestellt, das den genannten Anforderungen gerecht wird. Es wurde in mehrjähriger Arbeit von dem Software-Haus WERUM, Lüneburg, entwickelt. Teile der Entwicklung wurden mit Mitteln des BMFT im Rahmen der DV-Gesetze der Bundesregierung innerhalb des Projekts PDV der Kernforschungszentrum Karlsruhe GmbH durchgeführt.

2. Allgemeine Eigenschaften des PEARL-Testsystems

Bevor in den nachfolgenden Abschnitten die Hilfsmittel für Einzel-, Integrations- und Gesamttest beschrieben werden, sollen hier die übergreifenden Eigenschaften des PEARL-Testsystems kurz dargestellt werden.

2.1 Test auf PEARL-Ebene

Alle Kommandos an das Testsystem, die das zu testende PEARL-Programm ("Testling") betreffen, beziehen sich ausschließlich auf den Quellcode des Testlings. Soweit sinnvoll, haben sie dieselbe Form wie die entsprechenden PEARL-Anweisungen.

2.2 Interaktiv

Das Testsystem erlaubt interaktives Arbeiten, d.h. alle Kommandos werden nach dem Übersetzen des Testlings vor seiner Ausführung oder bei seiner Ausführung an inzwischen eingeplanten Haltepunkten eingegeben. Außerdem kann die Ausführung des Testlings jederzeit zur Eingabe von Kommandos unterbrochen werden.
Man kann jedoch auch die Ausführung von Testsystem-Kommandos für den Zeitpunkt des Erreichens eines Haltepunktes einplanen, ohne daß das Testsystem dann in den Zustand "Dialog" übergeht. Dies ist z.B. für Snapshots oder kleinere Programmkorrekturen nützlich.

2.3 Gastrechner / Entwicklungssystem / Zielrechner

Aufgrund seiner PEARL-Betriebssystem-Schnittstellen und -Komponenten kann das Testsystem sowohl auf Gastrechnern oder Entwicklungssystemen als auch auf Zielrechnern installiert werden. Für Off-line-Tests werden die Schnittstellen zur Peripherie so simuliert, daß eine Simulation des anwendungsspezifischen Verhaltens der Peripheriegeräte einfach anschließbar ist.

2.4 Verfügbarkeit

Das Testsystem ist portabel in PEARL unter voller Ausnutzung des in [4] beschriebenen Sprachumfangs programmiert. Bisher ist es auf Rechnern vom Typ Hewlett-Packard HP 3000, Norsk Data NORD 10 S bzw. 100 sowie Siemens 330 im Einsatz.

2.5 Art der Implementierung

Der Testling wird nicht interpretiert sondern ausgeführt. Der größte Teil des Testsystems ist als eigenständiger Rechenprozeß realisiert, der den Adreßraum des Testlings nicht belastet. Die Kommunikation mit dem Testling erfolgt mittels einer übergeordneten Steuerung, die zum Testling gebunden wird.

3. Einzeltest

Im Einzeltest werden die verschiedenen Programm-Komponenten separat auf Richtigkeit geprüft.

3.1 Haltepunkte

Haltepunkte stellen Unterbrechungen im Ablauf des Testlings dar. Sie ermöglichen es, einzelne Teilaufgaben des Testlings zu trennen und somit separat zu testen. Die Kontrolle der Voraussetzung der nachfolgenden Aktionen verhindert Folgefehler und spart daher Zeit bei der Fehleranalyse.
Das Testsystem ermöglicht dem Benutzer, Haltepunkte einzuplanen durch folgende, sich auf den Quellcode beziehende Angaben:

- Quellanweisungszeile
- Markenbezeichner
- Prozedur- oder Task-Bezeichner
  (Zusatz: Unterbrechung bei Beginn oder Ende)
- Dation-Bezeichner
  (Zusatz: Unterbrechung bei Prozeßeingabe oder -ausgabe)
- Uhrzeitangabe
- Dauerangabe
  (Zusatz: einmalige oder zyklische Unterbrechung).

Jeder Einplanung eines Haltepunktes können Testsystem-Kommandos angefügt werden. Sie werden bei Eintritt der Unterbrechung automatisch ausgeführt. Die Schachtelung von Einplanungen ist erlaubt und wird durch die mögliche Benennung der einzelnen Einplanungen unterstützt. Es stehen Kommandos zum Auflisten und Löschen der Einplanungen zur Verfügung.
Soll der Testlauf in Anlehnung an Beispiel 1 (s. Anhang) jeweils zu Beginn der Prozedur "STEUERUNG" des Moduls "LAGER" unterbrochen werden, sind z.B. folgende Kommandos nötig:

```
BREAK ON ENTRY STEUERUNG IN LAGER ;
    HALT 'Beginn von Steuerung' ;
END ;
```

Bei Eintritt der Unterbrechung wird die Unterbrechungsstelle (Zeile, Prozedur, Modul), die unterbrochene Task sowie die Uhrzeit gemeldet. Das Kommando "HALT" läßt das Testsystem anschließend auf Benutzer-Eingabe warten. Der dem HALT-Kommando beigefügte Text wird zuvor ausgegeben. Die Eingabe des Kommandos "GO;" veranlaßt die Fortsetzung des Testlaufes.
Mitunter ist es notwendig, in den Ablauf des Testlings einzugreifen, ohne daß zuvor ein Haltepunkt eingeplant wurde. Das interaktive Arbeiten gestattet es dem Benutzer, zu jedem Zeitpunkt die Kontrolle über den Testling zu erlangen. Das Testsystem hält dann den Testling vor der nächsten Quellanweisung an.
Fest vorgegebene Haltepunkte sind Programm-Beginn, Programm-Ende und Deadlock-Situationen.

3.2 Zugriff auf Variablen des Testlings

Werte von Variablen des Testlings lassen sich per Kommando sowohl ausgeben als auch verändern.
Die Ausgabe und Eingabe von Werten ist dem Typ der Variablen angepaßt. Die Ansprache erfolgt unter Verwendung der im Quellcode benutzten Bezeichner.
Ist eine Variable durch Angabe von Bezeichner und Modul nicht eindeutig gekennzeichnet, so kann die Zeile der Deklaration und/oder die die Deklaration beinhaltende Prozedur zusätzlich angegeben werden. Bei reentrant benutzten Prozeduren ist ebenfalls die Angabe einer der Tasks möglich, welche die Prozedur aufgerufen haben.
Um bei mehrfacher Objektansprache die Objektidentifizierung zu vereinfachen, ist eine Voreinstellung von Modul, Prozedur und Task möglich.
Bezugnehmend auf Beispiel 1 sind etwa folgende Testsystem-Kommandos möglich:

```
TEST MODULE LAGER ;    -- Voreinstellung: Modul
DISPLAY POS.X ;
DISPLAY POS.Y OF STEUERUNG ;
DISPLAY GERAET OF STEUERUNG FOR VERSAND ;
ERROR FROM 33 := '0'B ;
```

Wie die Beispiele zeigen, läßt sich die Objektansprache bis auf Eindeutigkeit reduzieren.
Bei der Ausführung der Kommandos wiederholt das Testsystem die vollständige Objektidentifikation.

3.3 Traces

Der Trace-Mechanismus gestattet es, alle Aktionen des Testlings zu verfolgen. Die Trace-Meldung enthält alle notwendigen Angaben über die gerade ausführende Task, den Modul, in dem sich die Task augenblicklich befindet, die Prozedur sowie die Zeilennummer der anstehenden Quellanweisung.

Es stehen drei in sich gestaffelte Trace-Arten zur Verfügung:

- CALL TRACE (Trace-Meldung bei Beginn und Ende von Prozeduren und Tasks)
- PATH TRACE (wie CALL TRACE, zusätzlich Meldung bei Überlaufen von Marken)
- LINE TRACE (wie PATH TRACE, zusätzlich Meldung bei Überlaufen jeder Quellanweisungszeile).

Der Trace läßt sich hierbei generell oder beschränkt auf bestimmte Module, Tasks, Prozeduren oder Zeilenbereiche einschalten.
So erhält man nach dem Kommando:

```
LINE TRACE OF STEUERUNG IN LAGER ON ;
```

folgende Ausgabe (vgl. Beispiel 1):

```
LINE 30 ENTRY STEUERUNG IN LAGER
                    TASK : VERSAND IN M2
LINE 40 OF STEUERUNG IN LAGER
                    TASK : VERSAND IN M2
   ...
LINE 72 OF STEUERUNG IN LAGER
                    TASK : VERSAND IN M2
   ...
LINE 99 EXIT STEUERUNG IN LAGER
                    TASK : VERSAND IN M2
```

Soll der Ablauf von nur einer Task verfolgt werden, so besteht die Möglichkeit, den Trace mit einer entsprechenden Bedingung zu versehen.

3.4 Snapshots / kleine Programm-Korrekturen

Oftmals ist es hilfreich, an bestimmten Stellen im Testling den Wert verschiedener Variablen auszugeben ohne in den Dialogmodus überzugehen.
Das Testsystem bietet hier die Möglichkeit, über die im Abschnitt 3.1 erwähnten Haltepunkte Programmstellen anzugeben, an denen ein spezifizierter Satz von Variablen ausgegeben wird.

Ebenfalls ist es möglich, kleinere Programmkorrekturen (Zuweisungen), wie zum Beispiel fehlende oder falsche Initialisierungen, durchzuführen, um so Zeit, die beim erneuten Übersetzen, Binden und Laden des Programms anfallen würde, zu sparen.

3.5 Rückverfolgung

Das Testsystem führt Buch über die begonnenen und noch nicht abgeschlossenen Aktionen des Testlings. Hierzu gehören vor allem noch nicht abgeschlossene Unterprogramm-Aufrufe sowie Belegungen von Synchronisationsvariablen.

Im Fehlerfall, aber auch an beliebigen Programmstellen, sind diese Informationen ein wichtiges Hilfsmittel zur Kontrolle des Ablaufes im Testling.
Entsprechend Beispiel 1 könnte eine solche Meldung lauten:

```
TASK VERSAND
    IN STEUERUNG IN LAGER AT LINE 72
    CALLED FROM
        VERSAND IN M2 AT LINE 60
    PRIO = 10 ACTIVE
    WAITING FOR MASCHINE
```

Entsprechend der Schachtelungstiefe werden alle Prozeduren mit ihrer Aufrufzeile gemeldet. Zusätzliche Angaben über den Task-Status ergänzen die Information (vgl. hierzu ebenfalls Abschnitt 4.1).

3.6 Ausführung im Einzeltakt

Um kritische Programmstellen kontrolliert zu durchlaufen, ist es möglich, den Ablauf des Testlings im Einzeltakt durchzuführen. Der Benutzer spezifiziert lediglich im GO-Kommando die Schrittweite in Anzahl Quellanweisungszeilen. Nach Überlaufen entsprechend vieler Quellanweisungszeilen wird der Testling unterbrochen. Eine erneute Kommando-Eingabe ist nun möglich.

4. Integrationstest

Um den Test paralleler Aktivitäten des Testlings zu unterstützen, enthalten die meisten Testsystem-Kommandos eine optionale Task-Angabe. Sie bewirkt, daß eine Ausführung des Kommandos nur durch die angegebene Task erfolgt.
So können Testausgaben in Grenzen gehalten und Korrekturmaßnahmen gezielt für eine bestimmte Task durchgeführt werden.

```
BREAK ON ENTRY STEUERUNG IN LAGER
                    FOR WARTUNG ;
    HALT 'WARTUNG RUFT STEUERUNG' ;
END ;
```

Die geforderte Unterbrechung des Testlaufes wird nur dann wirksam, wenn die reentrant benutzte Prozedur "STEUERUNG" von der Task "WARTUNG" aufgerufen wird (vgl. Beispiel 1).
Sollen in Beispiel 1 nur die Aktionen der Task "VERSAND" verfolgt werden, so ist folgendes Kommando möglich:

```
CALL TRACE FOR VERSAND ON ;
```

Ferner stehen dem Benutzer umfangreiche Status-Ausgaben zur Verfügung. Sie enthalten die relevanten Informationen über den aktuellen Zustand der Tasks sowie Informationen über die Historie.

4.1 Status-Ausgabe für Tasks und Synchronisationsvariablen

Zu jedem Zeitpunkt, auch innerhalb von Haltepunkt-Einplanungen, läßt sich der Status des Testlings abrufen. Die Task-Status-Meldung enthält hierbei

- Priorität
- augenblicklichen Standort mit Aufrufhierarchie
- falls wartend: worauf
- ob suspendiert
- eventuelle Einplanungen.

Die Status-Meldungen für Synchronisationsvariablen enthalten den Zustand (gesperrt, frei, n-mal geentert ...) sowie eine Liste der Tasks, welche die Synchronisationsvariablen belegt haben. Zusätzlich wird auf Anforderung die Aufrufhierarchie der Task zum Zeitpunkt der Belegung ausgegeben.
Während Deadlock-Situationen, die alle Aktivitäten im Testling blockieren, automatisch erkannt werden, ist die per Kommando ausführbare Deadlock-Untersuchung hilfreich bei der Aufdeckung von Teilblockierungen. Hierbei werden Blockierungszyklen gemeldet, die nur durch eine am Zyklus unbeteiligte Task aufgehoben werden können.

Um die Benutzung von Synchronisationsvariablen sowie Änderungen der Task-Zustände zu verfolgen, steht ein STATUS TRACE zur Verfügung. Die Meldung beinhaltet die ausgeführte Task-Steueranweisung bzw. Task-Koordinierungsanweisung, die betreffende Task und die gegebenenfalls beteiligten Synchronisationsvariablen.

4.2 Status-Änderung für Tasks und Synchronisationsvariablen

Bei Fehlern im Synchronisationsmechanismus des Testlings muß nicht unbedingt der Testlauf abgebrochen werden. Das Testsystem stellt alle notwendigen Task-Steueranweisungen (ACTIVATE, SUSPEND ...) und Task-Koordinierungsanweisungen (REQUEST, RELEASE, ENTER ...) in PEARL-Form bereit.
Die Task-Steueranweisungen enthalten selbst keine Startbedingung, jedoch ist ein entsprechendes Verhalten durch die Verwendung von zeitabhängigen Haltepunkten realisierbar.

5. Gesamttest

Der Gesamttest erfordert eine möglichst exakte Simulation der tatsächlichen Ablaufumgebung. Das Testsystem unterstützt dies durch eine Simulation der Zeitachse, die den korrekten Ablauf paralleler Aktivitäten im Testling gewährleistet. Wie im Abschnitt 3.1 (Haltepunkte) schon erwähnt, kann der Benutzer selbst Testsystem-Kommandos zeitabhängig einplanen oder ausführen lassen.

Für die Prozeß-Eingabe und -Ausgabe stehen Schnittstellen für einen einfachen Anschluß von anwendungsabhängigen

Programmen zur Verfügung, die die Prozeßperipherie simulieren. Hierbei ist es ebenfalls möglich, Haltepunkte für bestimmte Prozeß-Eingaben oder -Ausgaben einzuplanen.

Anweisungen zum Verkehr mit Standardperipherie werden mittels der entsprechenden Geräte des Testrechners (Gastrechner oder Entwicklungssystem) ausgeführt.

6. Schrifttum

[1] G l a s s , Robert L.: Real-Time: The "Lost World" of Software Debugging and Testing. Comm. ACM, May 1980, Vol. 23, Number 5, p. 264 - 271.

[2] G r i m m , R. und H e r t l i n , I.: Wege zu Programmsystem-Produktionsmitteln. Regelungstechnische Praxis, 22. Jahrgang, 1980, Heft 9, S. 314 - 321.

[3] H ü n k e , H. (Ed.): Software Engineering Environments. Proc. Symposium $S^2 E^2$, June 1980, Lahnstein, Germany, North Holland 1981.

[4] W e r u m , W. und W i n d a u e r , H.: PEARL. Beschreibung mit Anwendungsbeispielen. Vieweg 1978.

Anhang

Beispiel 1:

```
1  MODULE ( LAGER ); PROBLEM ;
30 STEUERUNG: PROC (
              GERAET FIXED,
              POS STRUCT [(X,Y,Z) FIXED] IDENT,
              ERROR BIT(1) IDENT
             ) REENTRANT GLOBAL ;
40 . . .
70 CASE GERAET
      ALT  /* Geraet1 */
72       REQUEST MASCHINE ;
         Ermittlung der Verweilzeit ;
90       Fortsetzung nach entsprechender Dauer
      ALT  /* Geraet2 */
   . . .
99 END ;
   MODEND ;
- - - - - - - - -
1  MODULE ( M2 ); PROBLEM ;
20 VERSAND: TASK PRIO 10 ;
60       Aufruf von Steuerung
   END ;
100 NACHSCHUB: TASK PRIO 5 ;
150      Aufruf von Steuerung
   END ;
200 WARTUNG: TASK PRIO 15 ;
230      Aufruf von Steuerung
   END ;
   MODEND ;
```

Beispiel 2:

Im Beispiel 1 sei eine Blockierung entstanden.
Die Task 'WARTUNG' hat nach durchgeführten Arbeiten die Maschine nicht wieder freigegeben.
Das STATUS TASK-Kommando gibt folgende Auskunft:

```
TASK VERSAND
   IN STEUERUNG IN LAGER AT LINE 72 CALLED FROM
      VERSAND IN M2 AT LINE 60
      PRIO = 10, ACTIVE
      WAITING FOR MASCHINE
TASK NACHSCHUB
   PRIO = 5, INACTIVE
TASK WARTUNG
   PRIO = 15, INACTIVE
```

Das STATUS SEMA-Kommando erklärt die Blockierung:

```
SEMA MASCHINE PRESET : 1
   BLOCKED BY : WARTUNG SINCE 10:05:00.000
   WAITING TASK : VERSAND
```

Beispiel 3:

Es soll überprüft werden, ob die übrigen Tasks auf andere Geräte ausweichen, wenn die Task 'WARTUNG' aus Beispiel 1 MASCHINE reserviert.

```
B1: BREAK ON 72 IN LAGER FOR WARTUNG ;
    B2: BREAK ON 72 IN LAGER ;
        HALT 'WARTUNG IN GEFAHR' ;
    END ;
    B3: BREAK ON EXIT STEUERUNG IN LAGER
                          FOR WARTUNG ;
        CANCEL B2 ;
    END ;
END ;
```

Beispiel 4:

Dauert die Belegung einer Maschine in Beispiel 1 länger als 5 Minuten, soll dies angezeigt werden.

```
B1: BREAK ON ENTRY STEUERUNG IN LAGER ;
    B2: BREAK AFTER 5 MIN ;
        STATUS TASK ;
    END ;
    B3: BREAK ON EXIT STEUERUNG IN LAGER ;
        CANCEL B2, B3 ;
    END ;
END ;
```

Anschrift der Autoren:

Peter J. Brunner, Klaus Meffert und Dr. Hans Windauer
Entwicklungsbüro Wulf Werum
Glogauer Straße 2 a
2120 Lüneburg
Tel.: 04131 / 53344, 53066