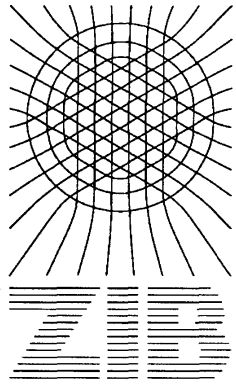
Konrad-Zuse-Zentrum für Informationstechnik Berlin
Heilbronner Str. 10, D-1000 Berlin 31



M. Grammel G. Maierhöfer G. Skorobohatyj

Parallel TRAPEX in Pool–T

(Die Implementierung eines adaptiven numerischen Algorithmus in einer parallelen objektorientierten Sprache)

Technical Report TR 90–5 (Oktober1990)

Herausgegeben vom
Konrad-Zuse-Zentrum für Informationstechnik Berlin
Heilbronner Str. 10
1000 Berlin 31
Verantwortlich: Dr. Klaus André
Umschlagsatz und Druck: Rabe KG Buch-und Offsetdruck Berlin

ISSN 0933-789X

M. Grammel G. Maierhöfer G. Skorobohatyj

Parallel TRAPEX in Pool–T

(Die Implementierung eines adaptiven numerischen Algorithmus in einer parallelen objektorientierten Sprache)

Abstract

Parallelizing a given sequential algorithm is usually done in such a way that it is decomposed into several concurrently executable parts based on an anlysis of data dependencies. The component parts themselves are executed sequentially and usually perform some exchange of data. A language for parallel programming should support this procedure by providing constructs for easy expressing of this kind of modularization and communication. Now it is quite possible to take any sequential programming language and extend it to contain some parallel constructs. It may be more adequate, however, to use programming languages which are by themselves built around the principles mentioned. Languages based on the object–oriented approach obviously fulfill this requirement. In this report we examine whether an object–oriented language is also suitable for efficient programming of a numerical algorithm by considering an example algorithm for numerical integration — TRAPEX — and the language POOL–T, which we used for implementing.

Key Words: parallel object–oriented programming language, message passing, abstract data typing, class hierarchie, module import mechanism, client/server principle, load balancing, adaptive numerical algorithms, order and stepsize control, Romberg quadrature.

M. Grammel G. Maierhöfer G. Skorobohatyj

Parallel TRAPEX in Pool–T

(Die Implementierung eines adaptiven numerischen Algorithmus in einer parallelen objektorientierten Sprache)

Zusammenfassung

Die Parallelisierung eines vorhandenen sequentiellen Programmes erfolgt im allgemeinen in der Weise, daß es auf Grund einer Analyse der Datenabhängigkeiten in mehrere parallel ausführbare Teile zerlegt wird, die ihrerseits sequentiell ablaufen und untereinander Daten austauschen. Eine parallele Programmiersprache sollte diese Vorgehensweise unterstützen, indem sie Sprachmittel zur Verfügung stellt, mittels derer sich die Modularisierung und Kommunikation bequem formulieren läßt. Nun läßt sich prinzipiell jede vorhandene sequentielle Programmiersprache um solche Konstrukte erweitern; günstiger in Hinblick auf die Strukturierung eines parallelen Programmes erscheint es aber möglicherweise, eine solche Programmiersprache zu verwenden, die die genannten Konzepte als elementare zur Verfügung stellt. Dies ist offensichtlich bei objektorientierten Programmiersprachen der Fall. Im vorliegenden Bericht wird an Hand eines Beispiels — TRAPEX —, das in POOL–T implementiert wurde, untersucht, inwieweit eine objektorientierte Sprache zur effektiven Programmierung eines numerischen Algorithmus geeignet ist.

Key Words: parallele objektorientierte Programmiersprache, Message Passing, Abstract Data Typing, Klassenhierarchi, Modul–Import–Mechanismus, Client/Server–Prinzip, Load Balancing, adaptive numerische Algorithmen, Ordnungs– und Schrittweitensteuerung, Romberg–Quadratur.

Inhalt

Das vollständige Listing wird Ihnen auf Wunsch gerne zugesandt.

0. Einleitung

Der vorliegende Bericht beschreibt die Implementierung eines adaptiven sequentiellen Algorithmus zur numerischen Integralwertberechnung (Romberg–Quadratur) – genannt TRAPEX – in der parallelen objektorientierten Sprache POOL–T und untersucht anhand dieses Beispiels die Eignung dieses objektorientierten Ansatzes für die Parallelisierung von adaptiven numerischen Algorithmen.

Für TRAPEX wurde einige Parallelisierungsansätze, welche die Adaptivität beibehalten, untersucht und für SUPRENUM– und Transputer–Architekturen in MIMD–Fortran bzw. Occam implementiert (Berichte [10] und [11]). Dieser Bericht steht mit den beiden genannten Arbeiten in engen Zusammenhang.

Die Sprache POOL–T — das Acronym steht für 'Parallel Object–Oriented Language, Target' — wurde bei Philips, Niederlande im Rahmen von ESPRIT entwickelt und steht in Zusammenhang mit einem weiteren Projekt, der Entwicklung einer Multi–Prozessor Architektur DOOM ('Decentralized Object–Oriented Machine'). Gemeinsame Zielsetzung ist es, auf Sprach– und Architekturebene den objektorientierten Ansatz auf konsistente Weise um ein Konzept von Parallelität zu erweitern.

Im ersten Abschnitt werden der sequentielle Algorithmus TRAPEX und einige Parallelisierungsansätze erläutert. Der zweite Abschnitt gibt eine kurze Einführung in die Konzepte objektorientierter Sprachen, stellt die wichtigsten Merkmale der Sprache POOL–T vor und erläutert die Interpretation paralleler POOL–T Programme. Der dritte Abschnitt beschreibt die POOL–T Implementierung von TRAPEX und die Ergebnisse des Vergleiches der parallelen und sequentiellen Versionen. Im vierten Abschnitt wird die Eignung des POOL–T Ansatzes anhand ausgewählter Konzepte der Sprache hinsichtlich des dargestellten Anwendungsgebietes, den adaptiven numerischen Algorithmen, diskutiert. Der Anhang enthält die Programmquellen der POOL–T Implementierung, sowie die Meßergebnisse der Testbeispiele.

1. TRAPEX (Grundlagen)

1.1 Der sequentielle Algorithmus TRAPEX

Der Integralwert einer Funktion über ein Intervall wird in TRAPEX als Summe von Werten über Teilintervalle ermittelt. Die Integralwerte der Teilintervalle werden ihrerseits durch ein polynomiales Extrapolationsverfahren mit Trapezsummen als Näherungswerten gewonnen. Zur Bildung der Trapezsummen wird ein Teilintervall unterteilt durch Zwischenpunkte, die durch die sogenannte Bulirsch–Folge (1/2h, 1/3h, 1/4h, 1/6h, 1/8h, 1/12h) gegeben sind, wobei h die Teilintervall–Länge ist, auch Schrittweite genannt. Die Anfangsschrittweite ist Eingabeparameter und die Bestimmung der Längen aufeinanderfolgender Teilintervalle ist Bestandteil des Algorithmus — der Ordnungs– und Schrittweitensteuerung — und hat zum Ziel, die Anzahl der Funktionsauswertungen über das Gesamtintervall zu minimieren (siehe [10], [6], [7]).

Der Algorithmus enthält drei geschachtelte Iterationen:

1. die Berechnung von Integralwerten bezüglich aufeinanderfolgender Teilintervalle (basic step) solange, bis das gesamte Intervall erreicht ist, oder eine Fehlerbedingung auftritt;

2. falls das Aitken–Neville–Verfahren in einem "Ordnungsfenster" nicht "konvergiert", die wiederholte Reduktion der Schrittweite;

3. die Berechnung der Trapezsummen und des Extrapolationsfehlers und deren Wiederholung mit inkrementierter Ordnung so oft, wie — vereinfacht ausgedrückt — noch keine Konvergenz vorliegt und sich eine Neuberechnung mit verkleinerter Schrittweite nicht lohnt. Nach "erfolgreicher" Berechnung eines Teilintervalls kann die Schrittweite auch wieder vergrößert werden.

Das — vereinfachte — Struktogramm veranschaulicht die Struktur von TRAPEX (die Bezeichnungen sind an jene in den POOL-T Programmquellen angelehnt):

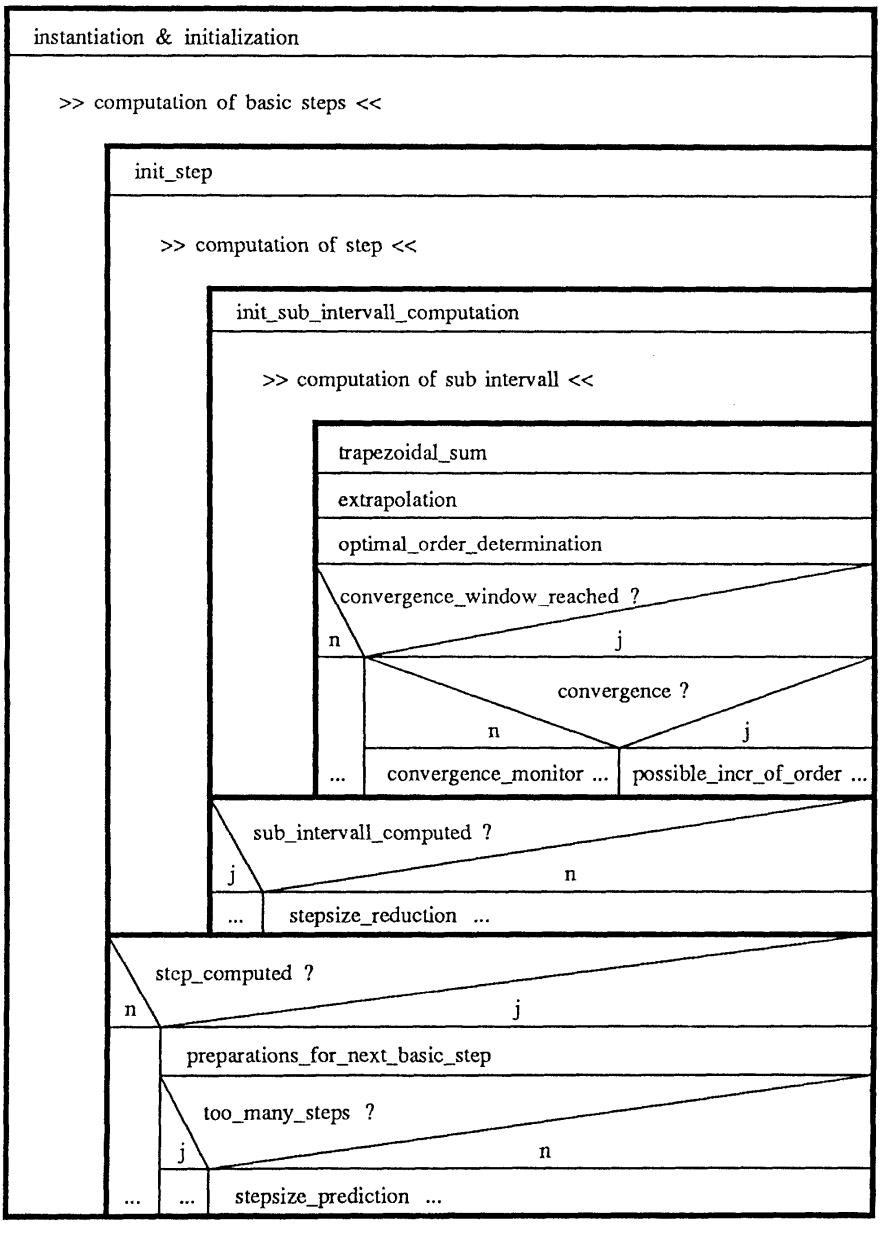


Bild 1.1

1.2 Vertikale Parallelisierung mit Lastabgabe

Eine ausführliche Untersuchung der Parallelisierungsmöglichkeiten des sequentiellen TRAPEX erfolgt in [10], deshalb seien nachfolgend die Ansätze einer Parallelisierung von TRAPEX nur umrissen.

Ein Ansatz zur Parallelisierung des Problems besteht darin, daß die Integralwerte von Teilintervallen, die zu Beginn fest vorgegeben werden, parallel berechnet werden können. Dies führt zu einer Master–Worker Konzeption derart, daß mehrere Worker, welche jeweils ein sequentielles TRAPEX darstellen, von einem Master mit einem Teilintervall gestartet werden und nach anschließender Berechnung das Ergebnis — den Integralwert des jeweiligen Teilintervalles — dem Master mitteilen.

Dieser Ansatz wird in [10] auch vertikale Parallelisierung genannt und kann verbessert werden, wenn die zweite Phase — die Reduktion der Schrittweite — als Ausgangspunkt für eine Aufteilung der Berechnung genommen wird. Dies bedeutet, daß ein Worker in dem Fall, daß er seine Schrittweite reduzieren muß, welches als "Lastzunahme" definiert wurde, einen Restteil seines Teilintervalles wieder an den Master zurückgeben kann und dieser es anderen Workern, die schon mit ihrer Berechnung fertig sind, zuteilt.

Es ist im allgemeinen zu erwarten, daß die Worker nicht zum gleichen Zeitpunkt ihre Berechnungen abschließen, weil die Kurvenverläufe in den einzelnen Teilintervallen sehr unterschiedlich sein können und deshalb für manche — aufgrund erhöhter Anzahl an Reduktionsschritten und kleinerer Schrittweiten — die Berechnung sehr viel aufwendiger ist und länger dauert. Die Unterschiede in der Auslastung können sehr groß sein, was die Effizienz der Parallelisierung erheblich mindert. Deshalb ist von der Lastabgabe im Falle der Reduzierung der Schrittweite an nicht beschäftigte Prozessoren aufgrund der besseren Lastverteilung (Load–balancing) eine deutliche Effizienzverbesserung zu erwarten.

Welches Beschleunigungsverhalten bezüglich der Anzahl der Knoten ist von diesem Ansatz 'Vertikale Parallelisierung mit Lastabgabe' für Trapex zu erwarten?

Einerseits bedeutet eine Parallelisierung der Berechnungen im günstigsten Fall einen proportionalen Anstieg der Beschleunigung, andererseits ergibt sich mit zunehmender Anzahl der Knoten nicht nur ein Anstieg des Kommunikations– und Verwaltungsaufwandes, es kann auch zu einem Mehraufwand an Berechnungen (Funktionsauswertungen) führen, wenn in erster Näherung die mittlere Intervallänge im parallelen Fall kleiner wird als die im sequentiellen Fall, und dadurch der Ordnungs– und Schrittweitensteuerung eine geringer werdende Wirksamkeit zukommt.

Bezüglich des Zusammenhanges von Beschleunigung und Anzahl der Knoten ist zu erwarten, daß mit zunehmender Parallelisierung die Beschleunigung zuerst ansteigt, dann aber hauptsächlich durch den zunehmenden Berechnungs- und Kommunikationsaufwand wieder absinkt, sodaß es für jedes Berechnungsbeispiel eine optimale Anzahl an Prozessoren bezüglich dieses Parallelisierungsansatzes gibt.

Eine "horizontale" Parallelisierung des Codes ergibt sich, wenn die Funktionswerte an den Stützstellen parallel berechnet werden, welches aber zu einem erhöhten Message–Aufkommen führt und Auslastungsprobleme der Knoten, die die Funktionswerte berechnen, nach sich zieht, so daß dieser Ansatz nicht effizient ist ([10]).

2. POOL–T (Grundlagen)

2.1 Konzepte objektorientierter Sprachen

Die Entwicklung objektorientierter Systeme geht zurück auf ein 1970 am Xerox PARC (Palo Alto Research Center) begonnenes Forschungsprojekt, welches zur Zielsetzung hatte, den Dialog zwischen Anwender und Maschine bequemer und einfacher zu gestalten. Als wichtigstes Ergebnis entstand die Programmiersprache Smalltalk, zugleich integrierte Arbeitsumgebung bestehend aus Multitasking–Betriebssystem, grafischer mausgesteuerter Benutzeroberfläche und kompletten Entwicklungswerkzeugen. Der Begriff 'objektorientiert' hat inzwischen eine weite Verbreitung gefunden, ist aber im engeren Sinne mit Konzepten und Sichtweisen der Sprache Smalltalk–80 verbunden, die im folgenden kurz erläutert werden (siehe [16], [8]).

Ein System objektorientiert darzustellen bedeutet im wesentlichen eine Klassifikation des Verhaltens seiner Objekte vorzunehmen. Der Begriff 'Objekt' bezeichnet in diesem Zusammenhang Einheiten des Systems — man betrachtet sie zunächst als "black boxes" —, denen ein "Verhalten" zukommt, welches herangezogen wird um sie zu kategorisieren. Hingegen gliedert eine prozedurale Sicht ein System typischerweise in Operationen und Datenstrukturen, d.h. es gibt passive Einheiten, genannt Daten, die Gegenstand schematischer Manipulationen, genannt Operationen, sind.

Vergleicht man diese beiden Sichtweisen, so stellt ein Objekt eine Zusammenfassung von Daten und Operationen dar derart, daß beide nicht mehr getrennt als solche vorkommen, sondern daß Objekte die elementaren Strukturierungsmittel sind, die ein System gliedern; anders ausgedrückt, in objektorientierten Sprachen gibt es keine Daten im prozeduralen Sinne (Werte), sondern an deren Stelle treten Objekte, denen die auf diesen Daten definierten Algorithmen als Eigenschaften zugeordnet sind.

Zum Beispiel kommt die Struktur der ganzen Zahlen in objektorientierter Sicht nicht als Menge der Zahlenwerte, auf welcher die elementaren arithmetischen und booleschen Operationen definiert sind, in Betracht, sondern als System von Objekten, denen als charakteristische Eigenschaft die Fähigkeit des Addierens, Subtrahierens usw zukommt, derart, daß jeder Zahl, in prozeduraler Sicht als Wert betrachtet, ein Objekt in objektorientierter Sicht entspricht.

Neben "Fähigkeiten" kann ein Objekt auch einen "Objektinhalt" aufweisen. Darunter versteht man einen jedem Objekt individuell zugewiesenen Satz an Variablen, der ausschließlich diesem zugänglich ist (zum Variablenbegriff in objektorientierten Sprachen siehe Kapitel 2.2 und 4.5). Diese Kapselung (encapsulation) der Objektinhalte ist charakteristisch für den Objektbegriff: ein Objekt

stellt für seine Umgebung eine "black box" dar und sowohl der Aufbau des Objektes, als auch sein jeweiliger Zustand ist nach außen hin verborgen.

Weiterhin sind Objekte — im Gegensatz zu Daten — "aktive" Einheiten, das bedeutet, ihnen kommt ein Laufzeitverhalten zu, wie es prozedurale Ausführungseinheiten aufweisen; insofern weisen sie Ähnlichkeit mit Prozessen auf — ohne jedoch bereits einen Begriff von Parallelität zu implizieren.

Es kann gesagt werden, Objekte sind die elementaren Strukturierungseinheiten eines objektorientierten Modells und ihnen kommt zu:

— ein Objektinhalt, d.h. ein Satz von Variablen, die jedem Objekt individuell zugeordnet und nur diesem zugänglich sind,

— eine Sammlung von objekteigenen Funktionen, welche die "Fähigkeiten" des Objektes bilden und Methoden (methods) genannt werden, sowie

— ein Laufzeitverhalten.

Ein Programmfluß entsteht durch Kommunikation zwischen Objekten, d.h. Objekte senden, empfangen und verarbeiten Botschaften (messages). Eine Botschaft besteht aus:

— der Angabe eines Zielobjektes,

— dem Namen einer Methode des Zielobjektes und eventuell

— Parametern, die der Methode des Zielobjekt als Objektreferenzen übergeben werden, sofern die aufgerufene Methode jene benötigt.

Dadurch ist Polymorphie möglich: Weil (objekteigene) Methoden bestimmen, was die Antwort eines Objektes auf eine Botschaft ist, kann eine Botschaft von verschiedenen Zielobjekten jeweils unterschiedlich (arteigen) interpretiert werden. Zum Beispiel könnten sowohl natürliche wie auch reelle Zahlen jeweils arteigen auf die gleiche Botschaft "square ()" reagieren.

Gleichartige Objekte werden zu Klassen (classes) zusammengefaßt. Hierbei bedeutet "gleichartig", daß alle Objekte einer Klasse sich in der Struktur ihres Inhaltes und ihres Verhaltens, d.h. Art und Anzahl der Variablen und der Methoden, gleichen. In diesem Zusammenhang werden Objekte als Instanzen (instances) ihrer Klasse bezeichnet, weil sie als Vertreter ihrer Klasse angesehen werden können.

Die strukturellen Eigenschaften aller Instanzen einer Klasse (Klasseninstanzen), d.h. ihre Variablen (instance variables) und ihre Methoden (instance methods), werden "Instanzeigenschaften" (instance features) genannt und in einer

Klassendeklaration vereinbart. Des weiteren gehört dazu eine Beschreibung des Laufzeitverhaltens der Instanzen einer Klasse, durch die festgelegt ist, in welchem Zustand eine Instanz einer anderen eine Botschaft sendet bzw. bereit ist, eine bestimmte Botschaft zu empfangen.

Eine Klasseninstanz zu erstellen bedeutet, daß ein neues Individuum dieser Klasse geschaffen wird, also ein Objekt, dessen Variablen und Methoden nach Art und Anzahl sowie dessen Laufzeitverhalten der Klassendeklaration entnommen sind. Dieser Vorgang ist in Smalltalk–80 der jeweiligen Klasse zugeordnet und wird von ihr selbst, nicht von anderen Objekten durchgeführt. Aus diesem Grunde gibt es auch Botschaften, die nicht an Objekte sondern an Klassen gerichtet sind, z.B. mit der Aufforderung eine neue Instanz einzurichten.

Solche Botschaften beziehen sich dann auf Klassenmethoden (class methods), die der Klasse selbst und nicht ihren Instanzen "gehören". Entsprechend können einer Klasse auch Variablen zugeordnet sein (class variables), diese sind dann genau einmal in der Klassendefinition selbst gespeichert und können von allen Instanzen der Klasse benutzt werden. In dieser Hinsicht erscheinen in Smalltalk–80 Klassen wiederum als besondere Objekte. Klassenmethoden und –variablen werden gemeinsam als "Klasseneigenschaften" (class features) bezeichnet; Instanz– und Klasseneigenschaften zusammen heißen einfach "Eigenschaften" (features).

Das Ordnungsprinzip, Objekte als Instanzen von Klassen zu organisieren, wird um ein weiteres ergänzt: die Organisation der Klassen zu einer Klassenhierarchie (class hierarchie). Dies bedeutet, daß eine Klasse in eine Oberklasse eingebettet sein und zugleich Unterklassen enthalten kann. In Smalltalk–80 ist diese Klassenhierarchie baumartig angeordnet, andere Sprachen erlauben hingegen allgemeine azyklische Graphen (z.B. $C++$).

Mit der Klassenhierarchie ist ein Vererbungsmechanismus (inheritance) verbunden, welcher Instanzeigenschaften auf Unterklassen überträgt derart, daß Variablen und Methoden der Oberklasse gleichzeitig in allen Unterklassen vorhanden sind. Gleichzeitig haben diese aber auch die Möglichkeit, ererbte Methoden durch eigene zu überlagern, die — semantisch gesehen — gegenüber der überlagerten Methode gewissermaßen einen Spezialfall darstellen. Auf diese Weise sind in der Klassenhierachie Klassen, die allgemeinere Eigenschaften repräsentieren, näher der Hierarchiewurzel angeordnet gegenüber solchen mit spezielleren Eigenschaften.

Zusammenfassend gesehen, sind Objekte individuelle Vertreter ihrer Klasse, die eventuell Unterklasse einer allgemeineren Art ist, und sie sind ausgestattet mit eigenen (spezifischen) und ererbten (allgemeineren) Eigenschaften. Das Wesen objektorientierter Sprachen ist ihr ausgeprägtes Abstraktionsvermögen, das sich in Konzepten wie Kapselung, Polymorphie, Klassenhierarchie und Vererbung

äußert; die dadurch entstehenden Vorteile sind gute Wiederverwendbarkeit und leichte Erweiterungsmöglichkeit des Systems (Stichworte: Prototyping, evolutionäre Softwareentwicklung).

2.2 Konzepte von POOL–T

Die Sprache POOL–T — das Acronym steht für 'Parallel Object-Oriented Language, Target' — wurde als Projekt im Rahmen von ESPRIT entwickelt; in Zusammenhang damit steht als weiteres Projekt die Entwicklung einer Multi–Prozessor Architektur DOOM ('Decentralized Object–Oriented Machine'). Gemeinsame Zielsetzung ist es, auf Sprach– und Architekturebene den objektorientierten Ansatz auf eine konsistente Weise um ein Konzept von Parallelität zu erweitern. Einige Konzepte objektorientierter Sprachen wie Kapselung, Botschaftenaustausch etc. unterstützen Parallelität sehr gut, andere allerdings erforden weitere Entwurfsentscheidungen. POOL–T hat nicht alle Merkmale (sequentieller) objektorientierter Sprachen.

POOL–T sei anhand der wichtigsten Merkmale vorgestellt, so daß ein Überblick über die Sprache und ein Verständnis der TRAPEX–Implementierung möglich ist, (für eine genaue Sprachbeschreibung siehe [1]):

- **Hierarchie, Vererbung:**
  POOL–T enthält keine Klassenhierarchie, also die Möglichkeit Unterklassen von Klassen zu bilden, und einen damit verbundenen Vererbungsmechanismus, sondern ein Modulkonzept ('units'), welches es gestattet Klassenspezifikationen zu im– und exportieren.

  Vergleichbar dem Modulkonzept von MODULA–2 gibt es korrespondierende 'specification –' und 'implementation units'. Eine 'specification unit' enthält ein oder mehrere Schnittstellenbeschreibungen von Klassen, die 'class specifications', welche von anderen 'units' importiert werden können. Eine zugehörige 'implementation unit' enthält die Implementierungen der Klassen, die 'class definitions' (oben Klassendeklaration genannt).

  Eine 'class definition' enthält:

  1. die Deklaration der Instanzvariablen,
  2. die Instanzmethoden ('method definition'),
  3. die Klassenmethoden, welche Routinen heißen ('routine definition'), und
  4. den Objektrumpf ('body'), welcher das Programm der Instanz darstellt.

Klassen müssen keine 'class specification' aufweisen, sie sind dann nur in der 'implementation unit', in welcher ihre 'class definition' steht, bekannt. Wechselseitige Importe zwischen Modulen sind erlaubt, d.h. es ist keine hierarchische Organisation der Module und der in ihnen enthaltenen Klassen vorgeschrieben. Auf diese Weise ist eine gleichberechtigte, beidseitige Kommunikation zwischen Instanzen verschiedener Klassen möglich, und zwar unabhägig davon, wie diese in Module zusammengefaßt und organisiert sind — es muß die Schnittstelle des Kommunikationspartners bekannt sein.

Eine dritte Modulsorte sind 'root units', welche in jedem Programm genau einmal enthalten sind und die Funktion eines "main program" erfüllen. Beim Programmstart wird ein initiales 'root object' als Instanz der letzten der in der 'root unit' enthaltenen Klassendeklaration erzeugt, welches anschließend die Inkarnation weiterer Instanzen veranlassen kann und auf diese Weise das System der Objekte startet.

- **Kapselung:**
  Die Kontrolle einer Instanz über ihre Variablen, ihr Verhalten und ihre Lebensdauer liegt ausschließlich bei ihr selbst, d.h. nach der Inkarnation der Instanz geht die Kontrolle an ihren Rumpf über. In POOL–T gibt es keine Klassenvariablen, dadurch gibt es in dieser Sprache nur lokale Daten – ausschließlich solche, die als Objektinhalt einer Instanz zugeordnet sind.

- **Parallelität:**
  In POOL–T ist Parallelität auf der Ebene der Instanzen eingeführt – nicht hingegen auf der Ebene der Anweisungen oder Ausdrücke. Dies bedeutet, daß Instanzen nach ihrer Inkarnation parallel zueinander laufen, aber jeweils in sich strikt sequentiell. Die Granularität eines Programms ist somit abhängig vom Entwurf der Klassenstruktur.

  Da es keine gemeinsamen Variablen (Klassenvariablen) gibt und Instanzen sequentiell arbeiten, tritt auch das Problem des gegenseitigen Ausschlusses (mutual exclusion) von Zugriffen auf solche nicht auf. Die globale Synchronisation der Objekte, d.h. die notwendige Sequentialisierung auf Grund von Datenabhängigkeiten, wird über Kommunikation erreicht: ein Empfänger muß auf eine benötigte Nachricht mindestens solange warten, bis der Sender sie abgeschickt hat.

- **Kommunikation:**
  Erhält ein Objekt eine Botschaft von einem anderen, sendet es stets eine Antwort an jenes zurück. Das Kommunikationsprotokoll entspricht einem synchronen, nicht–anonymen Nachrichtenaustausch (Rendezvous–Technik).

Ein Objekt schickt eine Botschaft, indem es eine Methode des Zielobjektes mit entsprechenden Argumenten aufruft und auf das Ergebnis der Methode wartet. Die Syntax des Aufrufes lautet:

```
<send-expression> ::=
  <send-destination-expression>
  ! <method-id> <actual-parameter-list>
```

und ermöglicht ein fortlaufendes (kaskadiertes) Senden,

```
z.B. 3 ! add (5) ! equal (9) ! not (),
```

weil ein Zielobjekt auch aus einem Ausdruck hervorgehen kann ('send–destination–expression'), und eine 'send–expression' ihrerseits wieder ein Ausdruck ist, der als Ergebnis des Methodenaufrufes eine Referenz auf ein Objekt liefert.

An Objekte gerichtete Botschaften werden jederzeit von diesen entgegengenommen und zur weiteren Bearbeitung in der Reihenfolge ihres Eintreffens aufbewahrt. Die Bearbeitung einer Botschaft erfolgt erst dann, wenn das Zielobjekt ein passendes 'answer statement' ausführt:

```
<answer-statement> ::= ANSWER <method-id-list>.
```

Die Vorgehensweise ist dabei folgende: wenn für eine der in der Liste des 'answer–statement' aufgeführten Methoden eine Botschaft vorliegt, dann führt das Objekt — vergleichbar einem Prozeduraufruf — diese Methode aus und teilt dem Sender das Ergebnis mit; ansonsten wartet das Objekt eine passende Botschaft ab.

Es sei hinzugefügt, daß es eine optionale 'post processing section' einer Methode gibt, die es ermöglicht, den Rückgabewert dieser Methode, schon an der Stelle, an der er vollständig berechnet vorliegt, dem Aufrufer mitzuteilen und anschließend weitere Anweisungen, sozusagen "Nachberechnungen", auszuführen. Weiterhin kann ein Objekt auch direkt seine eigenen Methoden aufrufen, d.h. ohne sich selbst als Zielobjekt angeben zu müssen:

```
<method-call-expression> ::=
  <method-id><actual-parameter-list>.
```

Botschaften an Klassen, die in POOL–T "Routinenaufrufe" heißen, werden konzeptuell betrachtet, von der aufgerufenen Klasse beantwortet. Bei der Ausführung hingegen arbeitet das aufrufende Objekt die Routine

selbst ab, d.h. der Kontrollfluß verzweigt in die Routine und kehrt anschließend mit dem Ergebnis an seine Aufrufstelle zurück. Dies bedeutet, daß eine Routine zu einem bestimmten Zeitpunkt mehrmals aktiv sein kann; jeder Aufrufer ist deshalb mit einem Satz lokaler Routinenvariable ausgestattet. Die Syntax des Routinenaufrufes lautet:

```
<routine-call-expression> ::=
  <class-name>.<routine-id> <actual-parameter-list>.
```

- **Kontrolle:**
  Es gibt in POOL–T drei Kontrollanweisungen: Verzweigung, Wiederholung und Auswahl. Die Syntax der Verzweigung erlaubt Bedingungskaskaden und nicht nur der 'else'–Zweig, sondern auch jeder 'then'–Zweig ist optional:

```
<if-statement>  ::=
  IF <condiditon> [ THEN <statement-sequence> ]
  { ELSIF <condition> [ THEN <statement-sequence> ] }
  [ ELSE <statement-sequence> ]
  FI.
```

  Die Wiederholung entspricht der abweisenden Schleife, ihre Syntax lautet:

```
<do-statement> ::=
  DO <condidion> [ THEN <statement-sequence> ] OD.
```

  Die Auswahl entspricht Dijkstras bewachten Anweisungen (guarded commands), wobei neben der (optionalen) Bedingungen auch die Empfangsdirektive 'answer' zugelassen ist, welche ein selektives Empfangen von Botschaften als Synchronisationsmittel ermöglicht. Ansonsten wird die zeitliche Reihenfolge der eingetroffenen Botschaften berücksichtigt. Die Syntax lautet:

```
<select-statement> ::=
  SEL <guarded-command> { OR <guarded-command> } LES

<guarded-command> ::=
  [ <condition> ]
  [ ANSWER <method-id-list> ]
  [ THEN <statement-sequence> ].
```

- **Variablen:**
  Der Variablenbegriff in objektorientierten Sprachen unterscheidet sich vom

Begriff der "Von–Neuman–Variable"; er stellt keinen "bezeichneten Behälter" für einen Wert bestimmter Ausdehnung dar — es gibt keine Daten im prozeduralen Sinne —, sondern immer eine Referenz auf ein Objekt. Eine Variable ändern, also ihr etwas zuzuweisen, heißt demnach, sie auf eine andere Instanz der gleichen Klasse zeigen zu lassen. In Anlehnung daran ist das Symbol für die Zuweisung in POOL–T '$\leftarrow$'. Variablen sind mit 'nil' initialisiert, was so interpretiert wird, daß sie auf ein artspezifisches Objekt 'nil' zeigen, welches aber kommunikationsunfähig ist.

- **Weitere Konzepte:**
  Ohne weiteren "syntaktischen Zucker" wären Ausdrücke — streng botschaftsorientiert — nur unbequem formulierbar. Beispielsweise sähe der Ausdruck

  ```
  3 + 5 ~= 9
  ```

  so aus:

  ```
  3 ! add (5) ! equal (9) ! not ().
  ```

  Diese Umformulierung wird automatisch vorgenommen, wobei die Zuordnung von Operationsymbolen zu Methodenbezeichnern fest vorgegeben ist und für alle Klassen (Operatorüberlagerung, Polymorphie) benutzt werden kann (z.B. '$A = B$' wird zu 'A ! equal (B)' expandiert). Allen Operationssymbolen ist eine Priorität zugeordnet.

POOL–T enthält die vordefinierten Standardklassen 'Boolean', 'Integer', 'Char' und 'String'; hingegen ist die Klasse 'Real' kein POOL–T Standard, sondern eine Option des POOL–T Interpreters. Diese Standardklassen werden für Von–Neuman–Rechner aus Effizienzgründen wie in prozeduralen Sprachen als Werte und nicht als Objekte codiert. Variable dieser Klassen sind daher technisch Von–Neuman–Variable und keine Objektreferenzen, hingegen ist der Variablenbegriff der Sprache hierdurch nicht berührt, denn dies betrifft ausschließlich die Ebene der Implementierung.

Vorläufig enthält die Standard–Systemumgebung die Klassen '`Read_File`' und '`Write_File`', welche auch die Ein–/Ausgabe übernehmen, sowie die Klasse '`Unix_Shell`', die als generelle Schnittstelle zu UNIX dient. Eine Weiterentwicklung dieser Systemumgebung derart, daß diese eine hinreichende Programmbibliothek auch zur Systemprogrammierung enthält, ist geplant (siehe [3]).

Eine weiteres Konzept sind parametrisierbare Klassen. So kann die Standardklasse '`Array (Element_Class)`' mit beliebigen, d.h. auch mit 'Arrays', Klassen als Feldelement instanziiert werden. Allerdings sind selbstdefinierte parametrisierte Klassen nicht POOL–T Standard sondern wiederum eine Option des Interpreters.

2.3 Simulation eines parallelen POOL–T Systems

POOL–T–Programme können durch den Interpreter 'pl' und den Simulator 'parpl' ausgeführt werden.

Der Interpreter 'pl' unterstützt den vollen Sprachumfang von POOL–T sowie zusätzlich die Klasse Real und parametrisierte Klassen; weiterhin enthält er umfangreiche symbolische Debug–Möglichkeiten. Der Interpreter wechselt in den Debug–Modus, wenn ein Interrupt auftritt (z.B. ein Laufzeitfehler oder C–c) oder wenn im Quellkode die Direktive '`##BREAK`' erscheint. Man kann im Debug–Modus Objekte schrittweise weiterlaufen lassen, einen Auflistung über den Status aller Objekte erhalten sowie Objektinhalt (Instanzvariablen) und Objektstack (Methodenvariablen) beliebiger Objekte betrachten. In POOL–T gibt es keine explizite Verwaltung der Objekte — sie wird dem Benutzer abgenommen. Deshalb enthält der Interpreter einen "Garbage Collector", welcher nicht erreichbare Objekte einsammelt.

Der Simulator 'parpl' is eine virtuelle DOOM–Maschine, die die reale möglichst gut wiederspiegeln soll. Die DOOM–Architektur stellt eine lose gekoppelte 'local memory'–Maschine dar, deren Knoten durch ein Punkt–zu–Punkt Netzwerk verbunden sind. Der Simulator:

— unterstützt den vollen Leistungsumfang des Interpreters 'pl';

— emuliert drei universelle Topologien: Generalisierter Chordal Ring (GCR), Generalisierter Petersen Graph (GPG) und Generalisierter Hypercube (GHC) (für ein Diskussion dieser Topologien siehe [14]);

— unterstützt die Objektallokation auf Knoten;

— enthält im Vergleich zu 'pl' zusätzliche Debug-Möglichkeiten, neben der Verwaltung von Objekten sind auch Informationen über Knoten erhältlich, beispielsweise eine Auflistung der momentan auf einem Knoten alloziierten Objekte;

— errechnet bestimmte Simulationsergebnisse einer Ausführung.

Die Objektallokation, d.h. die Zuordnung von Objekten zu Knoten, geschieht durch 'allocation pragmas' im Quellkode. Hierfür sind die Klassen 'Node', '`Node_Set`' und '`Loc (C)`' von parpl vordefiniert, d.h. die Ausführungsknoten des Netzwerkes erscheinen auf Sprachebene als Instanzen der Klasse Node. Die Knoten sind durch Nummern, von Null beginnend, adressiert. Die Zuordnung von Nummern zu Knoten nimmt eine Routine der Klasse Node vor, diese liefert jenen Knoten, welcher der Nummer "i MOD Anzahl der Knoten" entspricht.

Eine Methode der gleichen Klasse bildet die Umkehrfunktion; sie gibt die Nummer des Knotens der angesprochenen Instanz zurück.

In Verbindung mit der Simulation verschiedener Topologien ist es von entscheidendem Interesse, Ergebnisse über Programmläufe abzuleiten. Während eines Laufes können Zwischenergebnisse im Debug–Modus und anschließend Endergebnisse in Ausgabedateien in tabellarischer oder graphischer Form betrachtet werden. Die vom Simulator zur Verfügung gestellten Ergebnisse umfassen Informationen über die emulierte Topologie, lokale Meßergebnisse der Knoten und davon abgeleitete globale Ergebnisse. Dem Meßverfahren und der Definition der Parameter liegen Annahmen und Vereinfachungen zugrunde, so daß die Ergebnisse nicht als absolute, sondern nur als relative Aussagen gewertet werden sollten.

Es sei bereits hier erwähnt, daß für die Bewertung der Versionen von TRAPEX nur elementare Meßergebnisse der Simulation für anschließend von uns durchgeführte Berechnungen benutzt werden. Deswegen seien die Ergebnisse, wie sie parpl nach einer Simulation ableitet, und ihre Grundlagen im folgenden nur aufgeführt, ohne näher auf sie einzugehen (siehe [9]).

Einige Annahmen und Vereinfachungen sind:

— Die Ausführungszeiten sind auf einen Motorola 68020 Prozessor bezogen (geschätzt) und die Übertragungszeiten mit 10 MBit pro Sekunde angegeben.

— Pakete können immer auf dem kürzesten Wege übertragen werden, das Verbindungsnetzwerk ist nie blockiert oder eingeschränkt. Die Übertragungszeit ist allein proportional abhängig von der Anzahl der vermittelnden Knoten.

— Jeder Knoten hat eine unbeschränkte Speicherkapazität.

— Die aktiven Objekte eines Knotens werden nach der 'non–preemptive round–robin scheduling strategy' behandelt.

Grundlage der abgeleiteten Simulationsergebnisse sind lokale Zeitmessungen der Knoten im Netz. Jedem Knoten i ist zugeordnet, eine:

— lokale virtuelle Uhr;

— kumulierte "idle time", welche die Summe der bisherigen Leerlaufzeiten darstellt;

— kumulierte "communication time", welche die Summe der bisherigen Rechenzeiten, die für Interknoten–Kommunikation aufgewendet wurden, beinhaltet.

Dies bedeutet, daß für jeden Knoten jeweils Rechen–, Leerlauf– und Kommunikationszeiten auf der Basis des angewendeten Meßverfahrens und der erwähnten Annahmen bekannt sind.
Die wichtigsten Parameter einer Simulation, wie sie parpl nach einem Programmlauf berechnet, sind:

1. Informationen über die emulierte Topologie:
   - Topologiedefinition: GCR, GPG, GHC, Anzahl der Knoten;
   - Diameter des Graphen;
   - Grad des Graphen;
2. lokale Meßergebnisse der Knoten:
   - Wert der lokalen Uhr;
   - Auslastungsfaktor;
   - Kommunikationszeitfaktor;
   - Anzahl der auf diesem Knoten alloziierten Objekte;
   - Anzahl der vom Garbage–Collector eingesammelten Objekte;
3. abgeleitete globale Ergebnisse:
   - Laufzeit des Programms;
   - Beschleunigungsfaktor;
   - Parallel Index;
   - Systemauslastung;
   - Kommunikationszeit in Prozent;
   - durchschnittliche Paketdistanz.

3. POOL-T Implementierung von TRAPEX

3.1 Ein abstrakter objektorientierter Entwurf

Nachfolgend ist ein abstrakter Entwurf von TRAPEX vorgestellt, dessen Hauptziel es ist, das Abstraktionsvermögen des objektorientierten Ansatzes zu zeigen. Wie schon oben erwähnt ist für diesen Ansatz bezeichnend, daß es keine Daten im prozeduralen Sinne gibt und daß Algorithmen als Eigenschaften von Objekten angesehen werden.

Der Ausgangspunkt dieses Entwurfes ist, daß der Algorithmus TRAPEX den Integranden, d.h. reelwertigen Funktionen, als Eigenschaft zugeordnet wird. Genauer: die reelwertigen Funktionen einer reellen Variablen werden als Klasse von Objekten betrachtet, die sich wesentlich durch eine Auswertungsfunktion der Signatur "$R \rightarrow R$" auszeichnen, im folgenden als Evaluationsmethode 'at (x)' bezeichnet; diese Klasse wird um eine Methode 'trapex' erweitert, die — eine wesentliche Voraussetzung — für hinreichend oft differenzierbare Funktionen ein im Rahmen der geforderten Genauigkeit liegendes Ergebnis liefert.

Die Klasse der Funktionen der Signatur "$R \rightarrow R$" läßt sich in Unterklassen unterteilen. Eine gebräuchliche Gliederung wurde [5] entnommen, wobei — einschränkend — nur die elementaren reelwertigen Funktionen betrachtet werden.

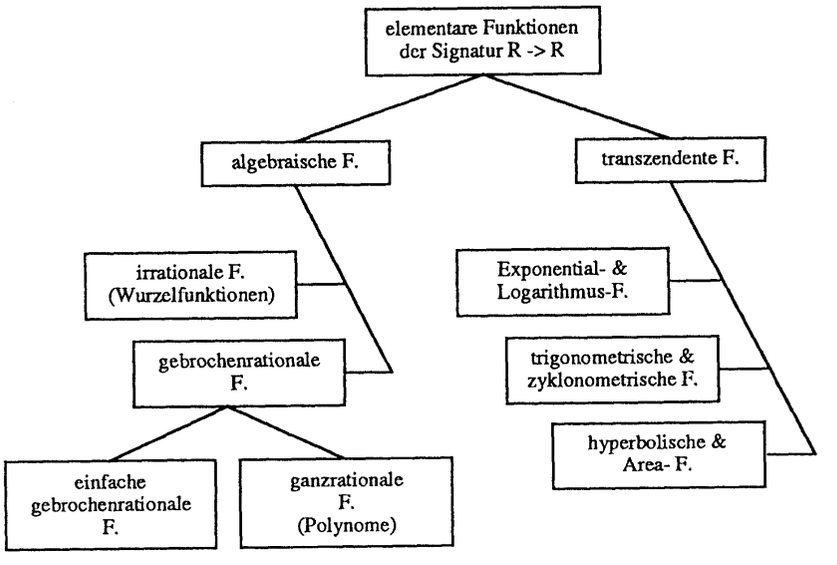


Bild 3.1

Diese Gliederung der Funktionen kann direkt in eine objektorientierte Klassenhierarchie übertragen werden. Jedes Objekt dieser Klassenhierarchie weist wesentlich eine individuelle Evaluationsmethode 'at (x)' auf, welche in ihren Eigenschaften die (Unter–)Klassenzugehörigkeit desselben bestimmt. Die Methode 'trapex' hingegen ist bezüglich dieser Gliederung eine allgemeine Eigenschaft, welche allen (unter der genannten Voraussetzung) reellwertigen Funktionen ungeachtet ihrer (Unter–)Klassenzugehörigkeit zugeordnet werden kann.

Für eine Implementierung folgt daraus, daß TRAPEX als Methode von Instanzen der Klasse "elementare Funktionen der Signatur $R \rightarrow R$" definiert wird. Diese Klasse kann weitere Unterklassen enthalten, wobei Instanzen dieser Klassenhierarchie durch eine artspezifische Evaluationsmethode 'at (x)' ausgezeichnet sind. Instanzen von Unterklassen erben die Methode 'trapex' als allgemeine Eigenschaft von ihrer Oberklasse, ohne daß Änderungen oder Erweiterungen dieser Methode notwendig sind. Genauer gesagt, ist die gesamte Unterklassenorganisation für die Methode 'trapex' irrelevant, so daß eine Umorganisation oder schrittweise Erweiterung dieser Klassenhierarchie die Implementierung von TRAPEX nicht berührt.

Weiterhin sei angemerkt, daß, z.B. aus Optimierungsgründen, Unterklassen um arteigene Integrationsmethoden ergänzt werden können, welche dann der ererbten, allgemeinen Methode 'trapex' nebengeordnet werden, oder diese auch überlagern können. Beispielsweise könnte für die (Unter–) Klasse der "ganzrationalen Funktionen" eine wesentlich effizientere Integrationsmethode entwickelt werden, welche auf analytischen und nicht — wie TRAPEX — auf numerischen Prinzipien beruht.

Der vorgestellte Entwurf ist insofern typisch für eine objektorientiertes Vorgehen, als von einer hierarchischen Gliederung des "Problembereiches" ausgegangen wird, welche unter Bezugnahme auf die schon implementierte Klassenhierarchie schrittweise umgesetzt und verfeinert wird. Dabei wird versucht, eine algorithmische Erweiterung einer schon bestehenden, möglichst allgemeinen Klasse als Methode zuzuordnen, damit entsprechend viele Unterklassen an der neuen Methode partizipieren können — beispielsweise wurde die Methode 'trapex' der Klasse 'Function' und nicht einer eigenen, neuen Klasse zugeordnet.

Wenn andererseits eine Erweiterung als ein Spezialfall gegenüber einer bestehenden Anwendung anzusehen ist, dann wird dies durch eine neue Unterklasse dargestellt ("Spezialisierung"), wobei bezüglich des Speziallfalles weiterhin zutreffende Eigenschaften ererbt, und nicht zutreffende Eigenschaften überlagert werden. Beispielsweise könnte die vorgestellte Funktionsklassenhierarchie um eine Unterklasse "hyperbolische & Area–Funktionen" erweitert werden — wenn es diese noch nicht gibt —, wobei die Methode 'trapex' ererbt werden kann und die artspezifische Methode 'at (x)' neu implementiert werden muß. Auf

diese Weise führt der objektorientierte Ansatz zu einem Softwareentwurf, der ein stufenweiser Erweiterungsprozeß ist, wobei aufgrund der Vererbung und Überlagerung in jeder Stufe jeweils nur die Unterschiede zum bereits bestehenden System implementiert werden brauchen (programming by difference).

3.2 Grobentwurf

Die Implementierung dieses abstrakten Entwurfes bereitet in POOL–T Schwierigkeiten. Wie schon erwähnt, enthält POOL–T anstelle eines Unterklassenkonzepts mit Vererbungsmechanismus ein Modulkonzept ('units') mit Export–Schnittstellen.

Verfolgte man den Ansatz, die Vererbung von Eigenschaften durch ihren Import nachzubilden, dann führte dies zu einer Umkehrung des Klassenbaumes in Bild 3.1, wenn man eine Darstellungsweise der Modulhierarchie zugrunde legt, in der die Exportrelation als "nach oben" gerichteter Graph gezeichnet wird. Da in einer Klassenhierarchie allgemeine Klassen mit grundlegenden Eigenschaften näher an der Wurzel sind, würden sie in einer Modulhierarchie — in der Klassen anderer Module diese Eigenschaften nur nutzen können, indem sie jene Klassen importieren — auf diese Weise gegenüber Klassen spezielleren Charakters untergeordnet sein und diesen sozusagen in einer "dienenden Funktion" zuarbeiten.

Diese Situation ist für eine Modulhierarchie mit Import/Export–Schnittstellen insofern typisch, als grundlegende Datentypen eine Basisschicht bilden, auf der benutzerspezifische Schichten aufbauen — aber auch die entgegengesetzte Situation ist denkbar, in der ein allgemeineres Modul speziellere Module unterordnet ("nach oben verbirgt"), um z.B. unterschiedliche Lösungen eines Problembereiches den übergeordneten Applikationen unter einer einheitlichen Schnittstelle zur Verfügung zu stellen.

In Bezug auf TRAPEX liegen beide Situationen gleichzeitig vor. Einerseits sollte TRAPEX — getreu dem Ansatz, Vererbung durch Import zu ersetzen — den Modulen, die Funktionsklassen enthalten, untergeordnet werden, so daß neu hinzukommende Funktionsmodule TRAPEX nur zu importieren brauchen, und TRAPEX seinerseits von einer Erweiterung seiner Anwendungsumgebung nicht betroffen ist; andererseits ist TRAPEX in gewisser Weise von seinen Anwendungen abhängig: für die Integration werden die Funktionswerte des Integranden benötigt, und im Falle einer Parallelisierung ist, um einen Engpaß bei der Berechnung der Funktionswerte zu vermeiden, eine Vervielfachung von "Funktionsservern" erforderlich. Das bedeutet, TRAPEX benötigt bestimmte Informationen seiner Integranden — den Klassennamen, die Evaluationsmethode 'at (x)' und die Erzeugungsroutine 'new (...)' — und kann deshalb ihnen nicht untergeordnet werden.

Dies führt zu einer Modulstruktur mit wechselseitigen Import/Export–Verhältnissen. Genauer, die Klassenspezifikation 'Function' bildet die Schnittstelle zu den implementierten Funktionen und stellt die Evaluationsmethode 'at (x)' und die Erzeugungsroutine 'new (...)' für TRAPEX und andere Anwendungen zur Verfügung — wobei das Vorhandensein und die Applikation lokal implementierter Klassen wie '`Algebraic_F.`', '`Transcendent_F.`' usw verborgen bleibt; umgekehrt exportiert die Klasse 'Trapex' die Methode 'trapex' exklusiv an die Klasse 'Function', so daß diese eine Integration für übergeordnete Anwendungen anbieten kann:

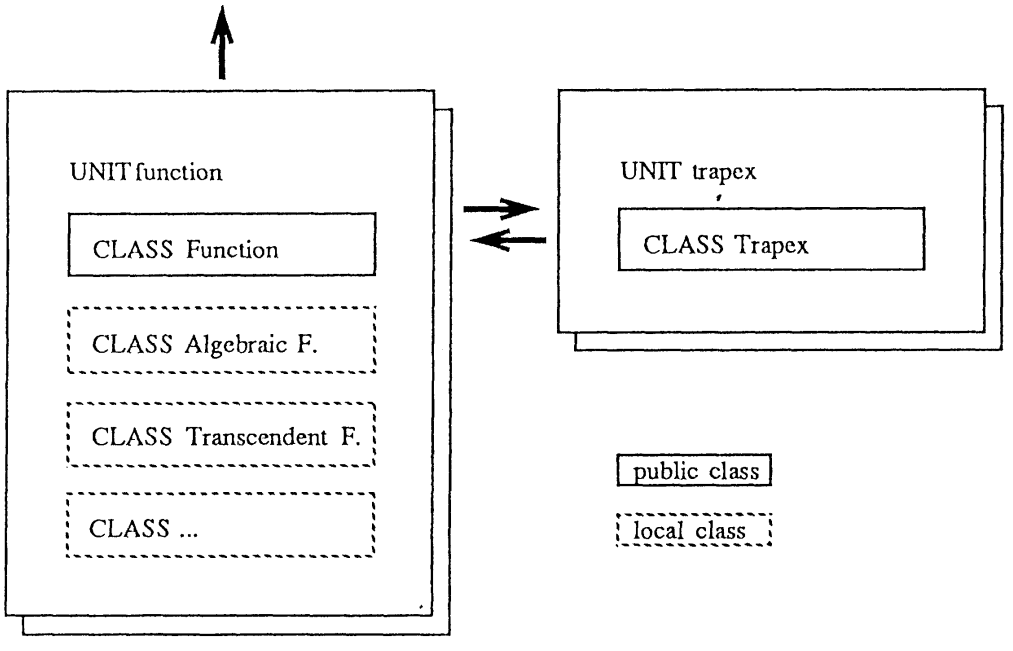


Bild 3.2

Es sei angemerkt, daß die — nicht zwingende — Zuordnung der Integrationsmethode 'trapex' und der Evaluationsmethode 'at (x)' zu verschiedenen Klassen, als auch deren Aufteilung in getrennte Module, durch den Ansatz, Vererbung durch Import zu ersetzen, entstanden ist, und eine konsequente Anwendung des Geheimnisprinzips (information hiding) darstellt.

Auf die in Bild 3.2 dargestellte Untergliederung des Modul 'function' in '`Algebraic_F.`', '`Transcendent_F.`' usw wird verzichtet, weil im Rahmen dieser Arbeit dem Modul lediglich die Aufgabe zukommt, für TAPEX eine Reihe von Testfunktionen bereitzustellen, so daß sich für diese Implementierung das Bild vereinfacht:

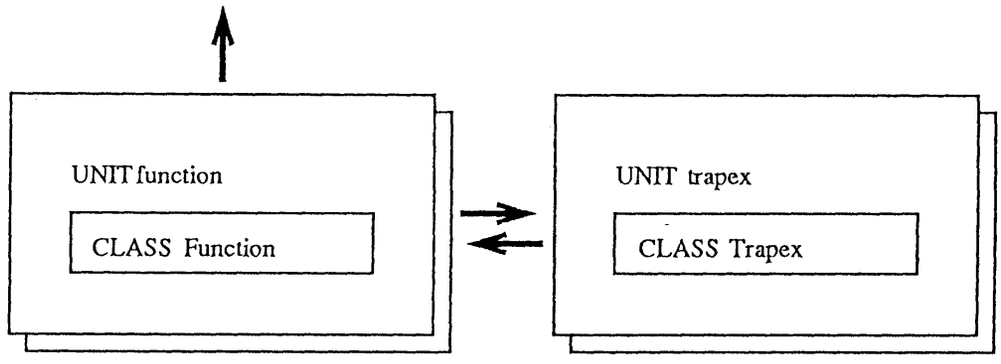


Bild 3.3

Es stellt sich noch die Frage, wie flexibel hinsichtlich der Erweiterbarkeit um neue Funktionen der vorgestellte Ansatz ist — insbesondere im Vergleich zur abstrakten, am Unterklassenkonzept mit Vererbungsmechanismus orientierten Modellierung. Auf diesen Punkt wird noch in Kapitel 4.1 eingegangen, wie auch auf bisher nicht erwähnte Klassen und Module ('Real', 'Quintuple' usw), wie sie der vollständige Modulgraph der TRAPEX–Implementierung im Anhang zeigt.

3.3 Feinentwurf

Zur Implementierung wurde der Algorithmus TRAPEX nicht weiter in unterschiedliche Objekte aufgegliedert. Dies hat seinen Grund in der aus der FORTRAN–Implementierung übernommenen Struktur, welche durch eine große Anzahl globaler Variable ausgezeichnet ist, auf die durchgehend im gesamten Code zugegriffen wird. Eine Feinmodellierung von TRAPEX in verschieden Objekte hätte zur Folge, daß die Zugriffe auf globale Variable nicht mehr durch

Seiteneffekte realisert werden könnten und zu 'message passing' führen würden. Dies bedeutet aber eine Erhöhung des Kommunikationsaufwandes und zunehmende Redundanz, ohne daß Transparenz und Flexibilität des Codes wesentlich verbessert würden.

Andererseits ist es aber möglich den Code in objekteigene Methoden aufzugliedern, damit er insgesamt übersichtlicher wird. Dies ist geschehen (ohne Erläuterung, siehe Anhang), und zwar können die einzelnen Methoden zu folgenden Gruppen zusammengefaßt werden:

— 'control methods', welche die logische Ablaufsteuerung des Algorithmus bilden (die Ablaufstruktur des Algorithmus könnte durch Zustände und Zustandsübergänge eines endlichen deterministischen Automates repräsentiert werden);

— 'computation methods', welche numerische Berechnungen durchführen und deren Ergebnisse über (dokumentierte) Seiteneffekte globalen Varablen zuweisen;

— boolesche 'test methods', welche die für die Berechnung und Steuerung notwendigen Prädikate formulieren.

Die Parallelisierung des sequentiellen TRAPEX ist nur mit geringfügigen Änderungen und Erweiterungen verbunden. Das zugrundeliegende Master/Worker Modell wurde bereits in Kapitel 1.2 vorgestellt. In dieser Implementierung heißen Master- und Worker-Objektklasse jeweils '`Trapex_Manager`' und '`Trapex_Server`'. Die Kommunikation zwischen '`Trapex_Server`'- und '`Trapex_Manager`'- Objekten ist einseitig gerichtet derart, daß nur die Server den Manager kennen und dessen öffentliche Methoden aufrufen und nicht umgekehrt. Dies bedeutet, daß freie Server um noch nicht berechnete Teilintervalle konkurrieren und ein Manager die Aufgabe übernimmt, die Resource "Teilintervall" zu verwalten und zuzuteilen.

Die Klassendeklaration '`Trapex_Manager`' umfaßt neben der Methode 'trapex', welche von den Funktionen aufgerufen wird, die öffentlichen Methoden '`drop_intervall`', '`get_intervall`' und '`get_result`', welche von den Servern benutzt werden um Teilintervalle zu erhalten, bei Lastzunahme abzugeben und um Integralwerte abzuliefern. Die privaten Methoden des Managers haben die Aufgabe, die Server zu erzeugen bzw. deren Termination anzuzeigen, den Intervallstack zu initialisieren und, wesentlich, die Anfragen der Server zu verwalten. Für den Intervallstack wurde die generische Klasse '`Stack (Element_Class)`' implementiert.

Die Klassendeklaration '`Trapex_Server`' unterscheidet sich von der von 'Trapex' wesentlich nur darin, daß sie um eine weitere Art von Methoden, den 'commu-

nication methods', welche die Kommunikation zum '`Trapex_Manager`'-Objekt übernehmen, ergänzt ist. Die 'communication methods' realisieren, indem sie die erwähnten Methoden des Managers aufrufen, folgende Aufgaben:

— ein Teilintervall, welches die Angaben zu Intervallanfang,

— ende und Anfangsschrittweite umfaßt, vom Manager abholen;

— ein Restintervall im Falle der Reduktion mit neu berechneter Schrittweite beim Manager zur weiteren Berechnung abgeben;

— das Ergebnis der Berechnung, d.h. den Status der Berechnung, den Integralwert des Teilintervalls und den extrapolierten Fehler, beim Manager abliefern.

Des weiteren sei auf die Instantiierung der konstanten Datenstrukturen der Klasse '`Trapex_Server`' hingewiesen, welche parallel erfolgt und — unter Berücksichtigung der Datenabhängigkeiten dieser Stukturen von der geforderten Genauigkeit — gegenüber der sequentiellen Version in einer leicht veränderten Reihenfolge geschieht (auf Konstantendeklarationen in POOL-T wird noch in Kapitel 4.6 eingegangen).

3.4 Simulation

Im Rahmen dieser Arbeit sind einige Testbeispiele — eine Untermenge der in [10] untersuchten Funktionen — mit dem Simulator parpl gerechnet worden. Ein wesentliches Ergebnis von [10] läßt sich an den Meßdaten im Anhang erkennen: bezüglich dieses Parallelisierungsansatzes — "Vertikale Parallelisierung mit Lastabgabe" — gibt es für jede Funktion eine Anzahl an Prozessoren, die nicht überschritten werden sollte, weil in diesem Fall der Berechnungsaufwand aus den in 1.2 erläuterten Gründen stark zunimmt.

In Kapitel 2.3 wurden Meßergebnisse und davon abgeleitete Parameter eines Simulationslaufes, wie sie der Simulator berechnet, vorgestellt. Für diese Simulation wurden nur die vom Simulator angegebenen Ausführungszeiten der einzelnen Knoten benutzt. Andere abgeleiteten Parameter, die u.a. über die "Güte" der Parallelisierung Auskunft geben sollen, kommen aufgrund der Weise, in der sie berechnet werden, nicht in Betracht. Das Meßverfahren, welches der Simulator für die Meßergebnisse benutzt, ist in [9] nicht näher erläutert und steht unter dem Vorbehalt der in 2.3 aufgeführten Annahmen und Vereinfachungen. Aus diesem Grund sind die vorgestellten Laufzeiten als Näherungen zu betrachten.

Die günstigste Topologie für das verwendete parallele TRAPEX ist ein Stern, in dem der zentrale Knoten für den "Manager" vorbehalten ist. Diese Topologie ist nicht homogen und wird von parpl nicht unterstützt. Für die Simulation wurde

die Option "Generalisierter Chordal Ring" als Topologie verwendet, wobei diese unter Berücksichtigung der Anzahl der Knoten als "Optimaler Chordal Ring" ausgelegt wurde (siehe [9] und [14]). Die Anzahl der Knoten des Netzwerkes ist um "1" erhöht, weil der Knoten "0" der Ein-/Ausgabe vorbehalten ist, so daß den Knoten "1" bis "n" ausschließlich die Aufgabe zukommt, TRAPEX zu berechnen.

4. Diskussion des Ansatzes von POOL–T

4.1 Abstraktionsvermögen

Der objektorientierte Ansatz ist durch zwei Paradigmen gekennzeichnet. Auf sie ist das Abstraktionsvermögen zurückzuführen, das diesen Ansatz auszeichnet (siehe [13] und [15]):

1. Objekte des Systems werden als Implementierungen Abstrakter Datentypen (abstract data types) beschrieben.

   Abstrakte Datentypen (ADT) in Programmiersprachen sind definiert als Zusammenfassung und Kapselung von Operationen und Daten derart, daß die Benutzung allein anhand der prozeduralen Schnittstelle erfolgen kann. Sprachen, die die Definition Abstrakter Datentypen durch einen Kapselungs–Mechanismus ermöglichen, werden auch objekt–basiert genannt (z.B. MODULA–2, ADA). Die Vorteile des "Abstract Data Typing" sind durch das Lokalitätsprinzip begründet:

   — leichte Änderbarkeit, d.h. Anwendungen sind von Änderungen nicht betroffen, soweit die Schnittstelle und deren Semantik eines ADT unverändert ist, und

   — gute Wiederverwendbarkeit, d.h. ein ADT kann, da frei von Seiteneffekten, in einem beliebigen Kontext gemäß seiner Semantik benutzt werden.

   Im Unterschied zu objekt–basierten Sprachen gilt für objektorientierte Sprachen der Grundsatz, daß alle Module, im Sinne gekapselten Einheiten, Typen sind, und umgekehrt — es gibt keine expliziten Typdeklarationen. An deren Stelle treten "Klassendeklarationen". Die Klassenbildung integriert Typ– und Modulaspekte. Dadurch wird das Programmieren Abstrakter Datentypen nicht nur ermöglicht, sondern gefordert.

   Ein objektorientiertes Entwurfsprinzip ist primär an den Datenstrukturen, und nicht am Kontrollfluß, orientiert. Dieser Umstand unterstützt eine

   — leichte Erweiterbarkeit, denn Datenstrukturen erweisen sich im Lebenszyklus eines Softwaresystems als relativ stabil — Erweiterungen betreffen selten Art und Anzahl der Datenstrukturen, als vielmehr deren Funktionalität, d.h. die Operationen, die auf ihnen implementiert sind ([13]).

2. Klassen werden als Erweiterungen oder Spezialisierungen anderer Klassen definiert (Hierarchie, Vererbung).

Die Vererbung ermöglicht eine Programmiermethodik, in der für neu hinzukommende Programmteile nur die Unterschiede zu bereits bestehenden implementiert werden müssen (programming by difference). In dieser Sicht wird der gesamte Entwurf als ein Prozeß der abgestuften Erweiterung und Verfeinerung der bereits vorhandenen Umgebung betrachtet. Dabei kann eine neu hinzukommende Unterklasse gegenüber ihrer Oberklasse einen — z.B. vorher bewußt nicht betrachteten — Spezialfall formulieren, oder sie stellt eine Erweiterung hinsichtlich neuer Fähigkeiten dar. In beiden Fällen braucht lediglich das wirklich Neue, die Unterschiede zum Bestehenden, hinzugefügt werden. Die Überlagerung sorgt für die spezifische, arteigene Interpretation; die Vererbung für die Verfügbarkeit gemeinsamer Eigenschaften.

Analog zur Spezialisierung gibt es auch die Möglichkeit der Generalisierung, d.h. wenn sich im Laufe der Entwicklung Gemeinsamkeiten zwischen Unterklassen herausstellen, können diese in eine Oberklasse "herausgezogen" werden. Dadurch wird Coderedundanz abgebaut — und die Gefahr von Inkonsistenzen durch nachträglichen Korrekturen oder Erweiterungen. Die Applikation solcher Klassen bleibt davon unberührt.

Im Hinblick auf die Implementierung des Algorithmus TRAPEX kann gesagt werden, daß, wie in den Kapiteln 3.2 und 3.3 erläutert, das Konzept der ADT nur global Anwendung fand und auf eine Feingliederung von TRAPEX in getrennte Objekte verzichtet wurde. Insofern gilt auch das gleiche für das Konzept der Klassenhierarchie. Allerdings zeigte gerade der in Kapitel 3.1 vorgestellte abstrakte Entwurf (dieser ging von einer Klassengliederung der Funktionen aus), daß das Konzept einer Klassenhierarchie, verbunden mit der Möglichkeit der Vererbung und Überlagerung, gegenüber der vorgenommen Implementierung in POOL–T auf der Basis eines Modulkonzeptes mit Import–Mechanismus angemessener ist.

Unter anderem wird das deutlich in Verbindung mit der strengen Typisierung und der fehlenden Möglichkeit einer Variantenbildung in POOL–T. Die Verknüpfung der (allgemeinen) Methode TRAPEX mit den jeweils (speziell) zu integrierenden Funktionen durch eine Importrelation führte zu einer Lösung, die gegenüber dem abstrakten, objektorientierten Entwurf mit Nachteilen versehen ist. Auf diesen Punkt wird noch ausführlich in Kapitel 4.4 eingegangen. Es muß aber hinzugefügt werden, daß das Modulkonzept von POOL–T durch die Möglichkeit zyklischer Importrelationen insofern hinreichend universell ist, als es möglich ist, beliebige Kommunikationsverbindungen paralleler Objekte darzustellen.

Wenn davon ausgegangen werden kann, daß bei der Implementierung numerischer Algorithmen in der Regel wenige, meist von vornherein bekannte Datenstruk-

turen verwendet werden (z.B. ganze, reelle Zahlen, Matrizen), und daß die Umsetzung von Verfahren, die weitgehend schon in mathematischer Notation vorliegen, im Vordergrund steht, dann bedeutet dies, daß dem ausgeprägten Abstraktionsvermögen des objektorientierten Ansatzes eine geringere Bedeutung zukommt, als in anderen Anwendungsbereichen, bei denen die Implementierung primär an der Modellierung der Datenstrukturen orientiert ist. Bislang werden solche Anwendungsbereiche als Domäne objektorientierter Sprachen betrachtet, weil sie durch ihr Abstraktionsvermögen u.a. einen "evolutionären" Softwareentwurf (Prototypen) besonders gut unterstützen.

Von Vorteil kann allerdings der objektorientierte Ansatz auch für numerische Algorithmen sein, die gemäß dem MIMD-Konzept parallelisiert werden sollen. Das hohe Abstraktionsvermögen und die Paradigmen dieses Ansatzes unterstützen eine Einführung von Parallelität gut. Anders als in imperativen Sprachen, die vom strikt sequentiellen Ansatz der Von-Neuman-Architektur herrühren, ist Parallelität objektorientierten Sprachen inhärent. Dies wird an mehren Punkten deutlich. Im wesentlichen bewirkt eine Gliederung in selbständige, gekapselte Einheiten — in Objekte —, deren Applikation immer eine Kommunikation mit ihnen bedeutet, daß diese als Träger von Parallelität in Frage kommen, ohne daß weitere Maßnahmen zur Gliederung in parallele Einheiten und zur Synchronisation von deren Speicherbereichen aufgrund möglicher Seiteneffekte notwendig werden.

Abschließend sei festgestellt, daß das im Vergleich zu den imperativen Sprachen höhere Abstraktionsvermögen des objektorientierten Ansatzes, welches von den Paradigmen des "Abstract Data Typing" und der Vererbung in einer Klassenhierarchie herrührt,

- zu besserer Änderbarkeit, Wiederverwendbarkeit und Erweiterbarkeit führen und nicht zuletzt das Verständnis und die intellektuelle Beherrschbarkeit größerer Systeme erleichtert;
- die Einführung der Parallelität unterstützt in der Hinsicht, daß die objektorientierte Gliederung zu einer vorgegebenen Granularisierung hinsichtlich potentiell parallel ablauffähiger Einheiten führt.

Hingegen ist der objektorientierte Ansatz — im Sinne einer Taxonomie der Programmiersprachen — stärker mit den klassischen imperativen Sprachen verwandt, als z.B. die (deklarativen) funktionalen und logischen Sprachen. Das zeigt sich daran, daß die Übertragung von Software aus imperativen Sprachen, wie am Beispiel TRAPEX von FORTRAN nach POOL-T geschehen, verhältnismäßig leicht ist. Inwieweit diesem Umstand eine Bedeutung beigemessen wird, hängt davon ab, ob das geringere Abstraktionsvermögen der klassischen im-

perativen Sprachen (Von–Neuman–Sprachen) in Bezug auf numerischer Algorithmen als ein Nachteil angesehen wird. Für die immer noch weitverbreitete Verwendung von FORTRAN in der Numerik allerdings noch andere Gründe ausschlaggebend sein.

4.2 Parallelisierung

Am Beispiel TRAPEX lassen sich Anforderungen an Programmiersprachen im Hinblick auf eine Parallelisierung adaptiver numerischen Algorithmen ableiten:

1. Der Anteil der Berechnung arithmetischer Ausdrücke bezüglich des Codes und der Ausführungszeit ist relativ hoch. Das bedeutet, daß eine Parallelisierung "im Kleinen", speziell arithmetischer Ausdrücke, sich lohnen würde, falls dies die zugrunde liegende Architektur, z.B. durch "Vektorpipelining" oder "Multiple Floating Point Units", entsprechend unterstützt. Eine feine Granulierung kann durch sprachliche Ausdrucksmittel ermöglicht, oder — vorteilhafter — vom Übersetzer vorgenommen werden: eine Optimierung und Parallelisierung des Codes, speziell der Applikation der elementaren numerischen Klassen, auf der Ebene der arithmetischen Ausdrücke, Schleifen etc. Auf diese Weise würde die implizite Parallelität des Codes aufgrund von Daten–(un)abhängigkeiten automatisch erkannt und, auf die konkrete Architektur bezogen, umgesetzt werden.

2. Bei TRAPEX wurde eine explizite Parallelisierung verwendet (Vertikale Parallelisierung mit Lastabgabe), die aus der Theorie der Integralrechnung folgt und die durch keinen Autoparallelisierer zu realisieren ist. Dies ist insofern typisch, als es sich bei der Parallelisierung "im Großen" oft mehr um eine Parallelisierung des Problems als des Codes handelt. Es ist daher zu fordern, daß die abstrakte parallele Modellierung des Algorithmus im Rahmen der sprachlichen Ausdrucksmittel für Parallelität, Kommunikation und Synchronisation leicht umzusetzen sein soll. Eine gute Unterstützung einer derartigen Parallelisierung bedeutet daher, daß der Programmieraufwand bezüglich der Aufgliederung in parallele Einheiten und für deren Kommunikation gegenüber der seqüntiellen Lösung möglichst klein ist.

Zu 1) ist zu sagen, daß POOL–T keine sprachlichen Möglichkeiten zur Formulierung der Parallelität "im Kleinen", speziell arithmetischer Ausdrücke, aufweist; die Ebene, auf der Parallelität in POOL–T zum Einsatz kommt, ist die der Instanzen und vielmehr vergleichbar mit der von sogenannten "Tasks". Eine Parallelisierung und Optimierung "im Kleinen", wie sie oben beschrieben

wurde, müßte somit vom Übersetzer geleistet werden. Hingegen ist es Aufgabe der beiden vorgestellten POOL–T–Interpreter, im Rahmen des Programmentwurfes Parallelität, genauer gesagt eine parallele Ausführungsumgebung, nur zu simulieren, insofern erübrigt sich natürlich eine derartige Autoparallelisierung; auch wird eine Codeoptimierung — unseres Wissens nach — nicht vorgenommen.

Zu 2) kann gesagt werden, daß POOL–T sowohl die Entwicklung paralleler Systeme als auch die Parallelisierung sequentieller Programme sehr gut unterstützt. Der Grund liegt darin, daß — im Gegensatz zu den sequentiellen imperativen Sprachen, welche im nachhinein um parallele Ausführungs– und Kommunikationskonstrukte erweitert wurden — der objektorientierte Ansatz (Botschaftenaustausch, Kapselung) auf konsistente Weise um ein Parallelitätskonzept (nebenlaufige Instanzen) erweitert wurde. Bei diesem Konzept korrespondiert daher die Granularität, im Sinne der Parallelität, mit der Modularität eines Programmes. Dies bedeutet, daß das Abstraktionsvermögen des objektorientierten Ansatzes durch seine Gliederungskraft auch eine natürliche Parallelisierung bewirkt.

POOL–T stellt nur den synchronen Botschaftenaustausch zur Verfügung. Das führt dazu, daß in einer Situation, in der eine parallele Modellierung einen asynchronen Botschaftenaustausch vorsieht, dieser durch ein "Pufferobjekt" realisiert werden muß, welches die Botschaft zwischen Produzent und Konsument puffert und auf diese Weise beide entkoppelt. Allerdings kann eine Klasse derartiger Objekte, z.B. die Klasse "Ringpuffer", generisch geschrieben und mit der Klasse der zu speichernden Objekte und der Größe des Ringpuffers parametrisiert werden. Sie kann dann, obwohl einmal implementiert, verschieden erzeugt und benutzt werden. Andererseits stellt sich die Frage, ob der objektorientierte Ansatz mit diesem Parallelitätskonzept nicht auch um ein asynchrones Kommunikationsmodell erweiterbar ist und inwieweit dies andere Aspekte (Sicherheit, Verifikation) berührt.

In POOL–T ist es nicht möglich, Objekte "von außen" zu unterbrechen und beispielsweise eine entsprechende Ausnahmemethode abarbeiten zu lassen — es gibt kein "Interrupt–" bzw. "User–Event–Handling". Diese Situation tritt bei TRAPEX nur am Rande auf: wenn einer der Server eine Fehlerbedingung anzeigt, dann könnte anderen, noch rechnenden Servern, deren Termination signalisiert werden. Bei anderen im Hause untersuchten adaptiven Algorithmen (siehe [12]) tritt jedoch der Fall ein, daß parallele Berechnungen u.U. "prophylaktischer" Art sind, d.h. sie können während der Berechnung aufgrund anderer Ergebnisse überholt sein. In solch einem Fall ist es wünschenswert den überholten Berechnungsprozeß sofort zu stoppen, weil sein Rechenaufwand unnötig ist. Es bleibt nur die Möglichkeit, diese Situation, in der z.B. ein Objekt

ein anderes unterbrechen will, um ihm eine Bedingung zu signalisieren, durch gezieltes Abfragen seitens des zweiten Objektes (polling) zu implementieren.

Die Objektallokation in POOL–T wurde oben bereits vorgestellt. Es sei hier noch einmal hervorgehoben, daß das verwendete Konzept elegant und flexibel ist. Die Prozessoren eines Netzwerkes erscheinen auf Sprachebene als Instanzen der vordefinierten Klasse 'Node' und '`Node_Set`'. Auf diese Weise sind "Operationen auf Knoten" leicht möglich. Hinsichtlich der Objektallokation besteht seitens der Sprache keine Einschränkung. Z.B. ist in TRAPEX der 'TRAPEX–Manager' mit einer Knotenmenge parametrisiert, auf die er dann die TRAPEX–Server plaziert.

4.3 Reals, Standardklassen

Wie bereits erwähnt wurde, enthält POOL–T die vordefinierten Standardklassen 'Boolean', 'Integer', 'Char' und 'String'; hingegen ist die Klasse 'Real' kein POOL–T Standard, sondern eine Option des POOL–T Interpreters. Diese Standardklassen werden aus Effizienzgründen, wie in prozeduralen Sprachen, als Werte und nicht als Objekte codiert, was zur Folge hat, daß Instanzen dieser Klassen nicht alloziierbar sind; ansonsten erscheinen sie auf Sprachebene konsequent gemäß des objektorientierten Ansatzes. Gerade für diese Klassen ist eine effiziente Implementierung besonders wichtig, denn zur Laufzeit dürften die meisten Objekte Instanzen dieser Basisklassen sein. Für numerische Algorithmen kommt hinzu, daß sie eine umfangreiche, leistungsfähige und an den Standards orientierte (IEEE) Bibliothek an mathematischen Funktionen erfordern.

Leider erfüllte die uns zur Verfügung stehende Implementierung von POOL–T diese Anforderungen nicht: es fehlten wichtige Angaben zur Genauigkeit und zum Wertebereich der Klasse 'Real' (siehe [4]), sowie Logarithmus– und trigonometrische Funktionen. Die Klassen '`Real_Server`' und '`Integer_Server`' wurden von uns implementiert und enthalten die zusätzlichen Funktionen, die für die TRAPEX–Implementierung benötigt wurden. Ein weiter Nachteil ist, daß es für Instanzen der Klasse 'Real' keine Denotation gibt (z.B. "3.05", "$305e-2$"), 'Real'–Zahlen können allein durch Routinen und Methoden ihrer Klasse erzeugt werden. Beispielsweise haben die Vereinbarungen von Variablen und Konstanten (siehe 4.5) der Klasse Real zu Beginn fast aller Klassendeklarationen lediglich die Aufgabe häufiger benutzte rationale Zahlen mit einem Namen zu bezeichen, um sie nicht jedesmal neu durch den Routinenaufruf 'Real.create (...,...)' zu erzeugen.

4.4 Tupel & Varianten

POOL–T bietet — im Gegensatz zu ebenfalls neueren Vorschlägen für objektorientierte Sprachen — keine Möglichkeiten zur Tupelbildung. Die "Records" oder "Structs" der klassischen imperativen Sprachen als Bestandteile der "zusammengesetzten Typen", haben ihr objektorientiertes Gegenstück in — vom Benutzer zu programmierenden — "strukturierten Objekten", welche als Funktionalität die entsprechenden Konstruktor– und Selektormethoden aufweisen.

Für TRAPEX wurden die generischen Klassen 'Triple' und 'Quintuple' implementiert, weil z.B. die Methode 'trapex' der Klasse 'Function' fünf Daten (Fehlercode, Integralwert, extrapolierter Fehler, Anzahl der Intregrationsschritte und der Funktionsauswertungen) als eine Instanz der Klasse 'Quintupel (Integer, Real, Real, Integer, Integer)' zurückgibt; Instanzen der Klasse 'Tripel (Real, Real, Real)' bilden die Elemente des Intervallstacks, sie enthalten die drei Angaben Intervallanfang, –ende und Anfangsschrittweite.

Die Klassen 'Triple' und 'Quintuple' sind zwar generisch, aber in der Anzahl ihrer Parameter festgelegt, so daß für jedes n eines benötigten n–Tupels genau eine generische Klassendeklaration mit n Parametern erforderlich wird. Dieser Programmieraufwand, für jedes n der benutzten n–Tupel eine generische Klasse zu implementieren, erscheint hinsichtlich der grundlegenden Bedeutung dieser Klassen als unnötig, zumal die gleichartig aufgebauten Klassendeklarationen sich nur in der Anzahl der Parameter unterscheiden.

Auch die Anwendung dieser generischen Klassen ist aus unserer Sicht umständlich, da Objekte diese Klassen mit "komplizierteren" vom Benutzer definierten Klassen auf eine Stufe gestellt werden, wobei es sich hingegen "nur" um "gewöhnliche" Tupel handelt. Weiterhin stellt jede Instanz dieser Klassen ein Laufzeitobjekt dar, während Tupel z.B. in (getypten) funktionalen Sprachen oder als "Records" in imperativen Sprachen gewöhnlich als reine "Datenobjekte" realisiert sind.

Ähnliches gilt auch für die Variantenbildung in POOL–T. Als Beispiel soll hier die Implementierung der Klasse 'Function' herangezogen werden. Zuvor sei erwähnt, daß die nachfolgend vorgestellte Problematik im abstrakten Entwurf des Kapitels 3.1 nicht auftritt, da dort das Konzept der Klassenhierarchie, verbunden mit einem Vererbungs– und Überlagerungsmechanismus, benutzt wurde. Hingegen ist die Grundlage des in Kapitel 3.2 erläuterten Grobentwurfes für POOL–T ein Modulkonzept mit einem Import–Mechanismus. Dieser Entwurf sieht vor, daß die Integranden von TRAPEX Instanzen der Klasse 'Function' sind, deren Methode 'at (x : Real) Real' eine Funktion auf ein gegebenes Argument anwendet. TRAPEX benutzt zur Kommunikation mit seinem Integranden den Namen, Methoden und Routinen der Klasse 'Function' und importiert hierfür deren Schnittstellenbeschreibung.

Wenn mehrere Funktionen integriert werden sollen, müssen deren Zuordnungsvorschriften in der Klasse 'Function' implementiert werden. Bei der Einrichtung von Instanzen dieser Klasse muß dann festgelegt werden, welche der implemetierten Funktionen die neue Instanz repräsentiert, d.h. welche Zuordnungsvorschrift bei einem Aufruf der Methode 'at (x : Real) Real' benutzt wird. Daß bedeutet, bei jedem Aufruf der Methode 'at' muß eine Selektion durchlaufen werden, bei der die entsprechende Zuordnungsvorschrift ausgewählt wird — diese bleibt dann nach der Instantiierung immer die gleiche.

Für die Implementierung bestehen folgende Möglichkeiten:

— Die Zuordnungsvorschriften der Funktionen können (funktionswertige) Ausdrucke, Methoden einer eigenen, wie geschehen, oder fremden Klasse (Unterklassen in Bild 3.2) sein.

— Die Selektion kann durch eine Verzweigung ('if–then–else') oder durch eine Auswahl ('select') realisiert werden. Bei der ersten Möglichkeit wird bei jedem Aufruf von 'at' die Bedingungskaskade solange durchlaufen, bis die geforderte Funktion gefunden ist, bei der zweiten werden sogar alle Bedingungen ausgewertet.

Weil nach der Instantiierung die ausgewählte Zuordnungsvorschrift immer die gleiche bleibt, ist der Berechnungsaufwand für die Selektion unnötig — die anderen Zuordnungsvorschriften kommen nicht mehr in Betracht. Weiterhin braucht die neue Instanz nicht mit deren Code ausgestattet zu sein.

Ein erster Lösungsvorschlag ist, eine Variantenbildung auf Klassenebene vorzunehmen: jede individuelle Zuordnungsvorschrift definiert eine eigene Klasse, sozusagen eine "spezielle Funktionsklasse"; neben der Methode 'at (x : Real) Real' enthalten Instanzen der Klasse 'Function' eine Variable, beispielsweise mit Namen '`selected_funtion`', die nach ihrer Instantiierung auf eine Instanz derjenigen "speziellen Funktionsklasse" zeigt, welche die ausgewählte Funktion bezeichnet. Die Evaluationsmethode 'at' der Klasse 'Function' würde dann nur noch die spezielle Evaluationsmethode 'at' jener, durch die Variable bezeichneten, Instanz ausführen (Polymorphie). Der Typ der Variable '`selected_function`' wird durch eine Typvereinigung aller in Betracht kommenden "speziellen Funktionsklassen" definiert. In dieser Sicht stellt die Klasse 'Function' eine Vereinigung ihrer "speziellen Funktionsklassen" dar.

Diese vorgestellte Lösung hat noch den Nachteil, daß für jede neu hinzukommende Funktion, die Variantenbildung der Variable '`selected_function`' ebenso, wie die (noch verbleibende) Fallunterscheidung bei der Instantiierung der Variable, um die neue Klasse erweitert werden müßte. Die in 3.1 vorgestellte Klassenhierarchie der Funktionen könnte allerdings die Variantenbildung und die Fallunterscheidung auf Unterklassen verteilen.

Eine weiterer Lösungsvorschlag, der diesen Nachteil ganz vermeidet, ist, die Klasse 'Function' generisch mit einer Parameterklasse auszustatten, deren Bedeutung der jener "speziellen Funktionsklassen" entspricht. Bei der Erzeugung einer Instanz der Klasse 'Function' würde dann eine Instanz der ausgewählten Funktionsklasse als Parameter übergeben, und die Evaluationsmethode 'at' der Klasse 'Function' würde dann nur jene der Parameterinstanz aufrufen. Für die Klasse 'Function' sind dann die "speziellen" Funktionsklassen nicht mehr mit ihrem Namen bekannt, sondern nur durch den Parameter bezeichnet.

Beide Lösungen sind in POOL–T nicht erlaubt, die erste setzt Konzept zur Variantenbildung voraus, die zweite verlangt, daß Methoden von parametrisierten Klassen aufgerufen werden könnten, welches u.a. zur Folge hätte, daß erst zur Laufzeit überprüft werden kann, ob die angesprochene Methode der parametrisierten Klasse auch wirklich existiert.

Varianten– und Tupelbildung sind in der Typtheorie als "direkte Summe" (Vereinigung) und "direktes Produkt" (Cartesisches Produkt) bekannt. Weil sie einfache, fundierte und, wie die aufgeführten Beispiele gezeigt haben, für die Praxis elementare Konzepte sind, wird, auf rein konzeptueller Ebene ohne Betrachtung von Implementierungsaspekten, ein Tupelkonzept für POOL–T skizziert:

— Eine Tupelklasse wird durch die Aufzählung der Elementklassen angegeben (z.B. bei Methodendeklarationen):

```
(Elementklasse_1, ... , Elementklasse_n).
```

— Entsprechend wird ein Tupel, also eine Instanz der jeweiligen n–Tupelklasse, durch Auflistung der Elementobjekte gebildet, wobei diese — wie sonst üblich — durch einen Ausdruck oder einer Variablen bestimmt sein können, z.B.

```
(expression_1, ... , expression_n).
```

— Auf diese Weise kann ganz auf Tupel–Konstruktorfunktionen (Injektionen, als Routinen realisiert) verzichtet werden; das gleiche gilt für Selektorfunktionen (Projektionen, als Methoden realisiert), wenn die Zuweisungs–Anweisung '←' auf Tupel von Variablen erweitert wird:

```
(var_1, ... , var_n) <- (expr_1, ... , expr_n).
```

Die Tupelklassen müssen übereinstimmen.

Dieser Vorschlag lehnt sich mehr an eine "anonyme" Tupelbildung Funktionaler Sprachen an, als an "Pascal" bzw. "C", deren "Records" oder "Structs" individuell mit Typnamen bezeichnet sind.

4.5 Speicherverwaltung

Wie bereits in 2.2 erläutert wurde, unterscheidet sich der Variablenbegriff in POOL–T von dem imperativer Sprachen. Eine Variable in POOL–T repräsentiert immer eine Referenz auf eine Instanz der Klasse ihres Typs und nicht — wie es eine "Von–Neuman–Variable" darstellt — einen bezeichneten "Behälter" für Daten (mit bestimmter Ausdehnung). Demnach bedeutet eine Zuweisung an eine Variable, diese auf eine neue Instanz derselben Klasse "zeigen zu lassen". Dies hat zur Folge, daß, im Unterschied zu "Von–Neuman–Variable", deren alter Wert nach einer Zuweisung "überschrieben" und damit "zerstört" ist, Instanzen entstehen können, die nach einer Zuweisung nicht mehr von einer Variablen referenziert werden.

Der, aus diesem Grunde benötigte, "Garbage Collector" des Interpreters 'pl' und des Simulators 'parpl' arbeitet nach der "Mark & Sweep"–Strategie (die Aufruffrequenz des "Garbage Collectors" kann als Option angegeben werden; bei 'parpl' ist der Default–Wert: alle 20.000 Instruktionen). Im Vergleich zu imperativen Sprachen — hier liegt die Allokation und Deallokation von Speicherplatz "auf dem Heap" gewöhnlich vollständig in Benutzerverantwortung — läßt sich als Vorteil von POOL–T (generell aller "Garbage Collector"–Sprachen) festhalten, daß der Benutzer eben jene Speicherverwaltung von Objekten "geschenkt" bekommt.

Ein Nachteil hingegen ist, daß der "Garbage Collector", besonders bei rekursiven Objektstrukturen (z.B. Bäume), einen sehr hohen Anteil der gesamten Ausführungszeit in Anspruch nehmen kann, während für gewöhnlich die Freigabe von Objekten durch den Benutzer effizienter ist, weil sie erstens frühest möglich erfolgt — bei guter Programmierung — und zweitens ohne Rechenaufwand für eine Analyse unreferenzierter Objekte zur Laufzeit auskommt. Ein weiterer Nachteil ist gewöhnlich die nicht vorhersehbare Dauer eines "Garbage Collector"–Laufes — wenn keine Verfahren angewendet werden, die, auch bei rekursiven Objektstrukturen, Echtzeitanforderungen genügen ("Compile Time Garbage Collection").

4.6 Konstanten

In 4.3 wurde schon auf spezielle Variablenvereinbarungen der Klasse Real hingewiesen, welche im Grunde genommen Konstantenvereinbarungen — ausgedrückt in Kommentaren — darstellen. In POOL–T sind keine Konstantendeklarationen vorgesehen; es liegt an dem Benutzer dies durch Variable auszudrücken und selbst darauf zu achten, daß er diese nach der Instantiierung nicht mehr verändert. Dies kann eine Fehlerquelle sein, die schwer zu entdecken ist, z.B. wenn aufgrund einer versehentlichen Veränderung, eine "konstante Variable"

nicht mehr jenes Objekt referenziert, welches sie — möglicherweise durch ihren Namen intendiert — in den Augen des Benutzers bezeichnet. Es wäre wünschenswert konstante Objektbezeichnungen zur Verfügung zu haben.

Schlußbetrachtungen

Das Hauptleistungsmerkmal objektorientierter Programmiersprachen ist ihr ausgeprägtes Abstraktionsvermögen, welches von den Paradigmen des objektorientierten Ansatzes herrührt:

1. Objekte des Systems werden als Implementierungen Abstrakter Datentypen (abstract data types) beschrieben;
2. Klassen werden als Erweiterungen oder Spezialisierungen anderer Klassen definiert (Hierarchie, Vererbung).

Am Beispiel des adaptiven numerischen Algorithmus TRAPEX und der parallelen objektorientierten Sprache POOL–T wurde gezeigt, daß dieser Ansatz aufgrund des genannten Abstraktionsvermögens und dadurch, daß Sprachkonstrukte zur Modularisierung, Parallelität Kommunikation als elementare Konzepte dieses Ansatzes zur Verfügung gestellt werden, die Implementierung des gewählten parallelen Modells von TRAPEX besonders gut unterstützt. Allgemein kann gesagt werden, daß der objektorientierte Ansatz für eine Parallelisierung, welche dieses hohe Abstraktionsniveau zum Ausgangspunkt nimmt, gut geeignet ist.

Andererseits wurde am Beispiel TRAPEX auch deutlich, daß einzelne Eigenschaften der Sprache POOL–T einer hinreichend effizienten, sicheren und einfachen Implementierung von numerischen Algorithmen entgegenstehen. Hauptsächlich zu nennen sind:

— die fehlende Standardisierung und Unterstützung der Klasse "Real";

— die fehlende Möglichkeit eines "Interrupt–" bzw. "User–Event–Handling";

— die fehlende Möglichkeit von Konstantendeklarationen;

— der Verzicht auf eine mit einem Vererbungsmechanismus verbundene Klassenhierarchie;

— in Verbindung mit letzterem Punkt, die ungenügende Unterstützung bei der Tupel– und Variantenbildung.

Literatur

1. P. America: *Definition of the programming language POOL–T.*
2. L. Augusteijn: *POOL–T user manual.*
3. L. Augusteijn: *POOL–T Standard Environment.*
4. M. Beemster, H. Muller: *User manual of pl, a POOL–T Interpreter.*
5. I.N. Bronstein, K.A. Semendjajew: *Taschenbuch der Mathematik.*
6. P. Deuflhard: *Order and Stepsize Control in Extrapolation Methods.* Preprint Nr. 93, Universität Heidelberg (1980).
7. P. Deuflhard, H.J. Bauer: *A Note on Romberg Quadrature.* Preprint Nr. 169, Universität Heidelberg (1982).
8. ix Multiuser–Multitasking–Magazin, Schwerpunktheft 1/90: "OOPS unter UNIX".
9. J.P. Katoen, P. den Haan: *User manual of parpl, a simulator for a parallel POOL–T system.*
10. G. Maierhöfer, G. Skorobohatyj: *Parallel–TRAPEX (Ein paralleler adaptiver Algorithmus zur numerischen Integration; seine Implementierung für SUPRENUM–artige Architekturen mit SUSI).* Technical Report TR 88–5, Konrad–Zuse–Zentrum für Informationstechnik Berlin (1988).
11. G. Maierhöfer, G. Skorobohatyj: *Implementierung des parallelen TRAPEX auf Transputern.* Report TR 89–8, Konrad–Zuse–Zentrum für Informationstechnik Berlin (1989).
12. G. Maierhöfer, G. Skorobohatyj: *Implementierung von parallelen Versionen der Gleichungslöser EULEX und EULSIM auf Transputern.* Technical Report TR 90–2, Konrad–Zuse–Zentrum für Informationstechnik Berlin (1990).
13. Bertand Meyer: *The road to object–orientedness.* Object–Oriented Software Construction, Prentice Hall (1988).
14. E.A.M. Odijk, R.A.H. van Twist: *Networks for Parallel Computers.* Proceedings COMP EURO Conference 1987, IEEE, New York, pp. 779–782.
15. Peter Wegner: *Learning the Language.* Byte (1989).
16. Klaus Zerbe: *Vorbild Smalltalk.* ct Magazin für Computertechnik, 4/90, 120 pp.

Anhang

A. TRAPEX - Simulationsergebnisse

```
Implementation:     POOL-T
Version:            trapex05
Date:               03.09.90
```

1. Netzwerk-Topologie "Generalized Chordal Ring" (s. [9] und [14])

Topology	#Nod	Diam	Deg	C/P	#Chd	Chd1	Chd2
seq-cr1	2	1	1	1	1	1	-
par-cr1	2	1	1	1	1	1	-
par-cr2	3	1	2	2	1	1	-
par-cr4	5	1	4	4	2	1	3
par-cr8	9	2	4	8	2	1	3
par-cr16	17	3	4	12	2	1	5
par-cr32	33	4	4	16	2	1	9
par-cr64	65	7	4	28	2	1	11
par-cr128	129	8	4	32	2	1	15
par-cr256	257	15	4	60	2	1	23

```
Parameter:
#Nod  ==  Number of nodes
Diam  ==  Diameter
Deg   ==  Degree
C/P   ==  Cost/Perf.
#Chd  ==  Number of chords
Chd1  ==  Chord [ 1]
Chd2  ==  Chord [ 2]
```

2. Simulationsergebnisse

*** Testbeispiel : 1, f (x) = exp (x)

Eingabeparameter:

- Intervall: [0.000000e+00, 1.000000e+00]
- maximale Schrittweite: 1.000000e+00
- erste Schrittweite: 1.000000e+00
- Fehler Epsilon: 2.000000e-15

Topology	Time	Nr of steps	Nr of eval.
seq-cr1	54.25	2	27
par-cr1	54.95	2	27
par-cr2	38.31	2	26
par-cr4	37.54	4	36
par-cr8	38.08	8	72
par-cr16	38.59	16	112
par-cr32	45.56	32	231
par-cr64	61.46	64	560
par-cr128	101.26	128	748
par-cr256	238.45	256	1280

*** Testbeispiel : 9, f (x) = 2 / (2 + sin (10*pi*x))

Eingabeparameter:

- Intervall: [0.000000e+00, 1.000000e+00]
- maximale Schrittweite: 1.000000e+00
- erste Schrittweite: 1.000000e-01
- Fehler Epsilon: 2.000000e-15

Topology	Time	Nr of steps	Nr of eval.
seq-cr1	1537.36	52	909
par-cr1	1762.20	63	6797
par-cr2	862.81	58	3513
par-cr4	588.43	58	2372
par-cr8	328.52	65	1802
par-cr16	213.06	59	1320
par-cr32	163.65	56	1088
par-cr64	82.47	64	920
par-cr128	125.24	128	1324
par-cr256	261.67	256	2116

```
*** Testbeispiel : 17,  f (x) = if x = 0 then
                                 50
                               else
                                 (sin (50*pi*x))**2 / 50*(pi*x)**2
                               fi
```

Eingabeparameter:

- Intervall: [0.000000e+00, 1.000000e+00]
- maximale Schrittweite: 1.000000e+00
- erste Schrittweite: 1.000000e+00
- Fehler Epsilon: 2.000000e-15

Topology	Time	Nr of steps	Nr of eval.
seq-cr1	2831.85	107	1699
par-cr1	3235.12	121	22950
par-cr2	2050.79	139	17890
par-cr4	1387.08	147	9971
par-cr8	623.95	111	3179
par-cr16	373.04	132	4185
par-cr32	256.16	133	3328
par-cr64	178.77	128	2652
par-cr128	136.21	128	1932
par-cr256	276.28	256	2788

```
*** Testbeispiel : 20,  f (x) = 1 / (x*x + 1.005e-3)
```

Eingabeparameter:

- Intervall: [-1.000000e+00, 1.000000e+00]
- maximale Schrittweite: 2.000000e+00
- erste Schrittweite: 2.000000e+00
- Fehler Epsilon: 2.000000e-15

Topology	Time	Nr of steps	Nr of eval.
seq-cr1	165.78	8	163
par-cr1	181.45	10	328
par-cr2	94.20	8	192
par-cr4	64.97	8	168
par-cr8	43.13	8	128
par-cr16	42.18	16	168
par-cr32	46.62	32	306
par-cr64	58.80	64	567
par-cr128	102.42	128	896
par-cr256	238.56	256	1768

*** Testbeispiel : 22, f (x) = 1 / (x*x + 1e-4)

Eingabeparameter:
- Intervall: [-1.000000e+00, 1.000000e+00]
- maximale Schrittweite: 2.000000e+00
- erste Schrittweite: 2.000000e+00
- Fehler Epsilon: 2.000000e-15

Topology	Time	Nr of steps	Nr of eval.
seq-cr1	806.30	54	933
par-cr1	796.02	57	4046
par-cr2	453.83	59	3118
par-cr4	303.30	58	1743
par-cr8	246.50	57	1206
par-cr16	223.52	58	998
par-cr32	172.15	67	1064
par-cr64	124.52	87	1265
par-cr128	183.22	146	1544
par-cr256	265.14	264	2296

B. TRAPEX - Programmquellen

1. Moduldiagramm

Das Bild veranschaulicht die Grobstruktur der POOL-T Implementierung von TRAPEX:

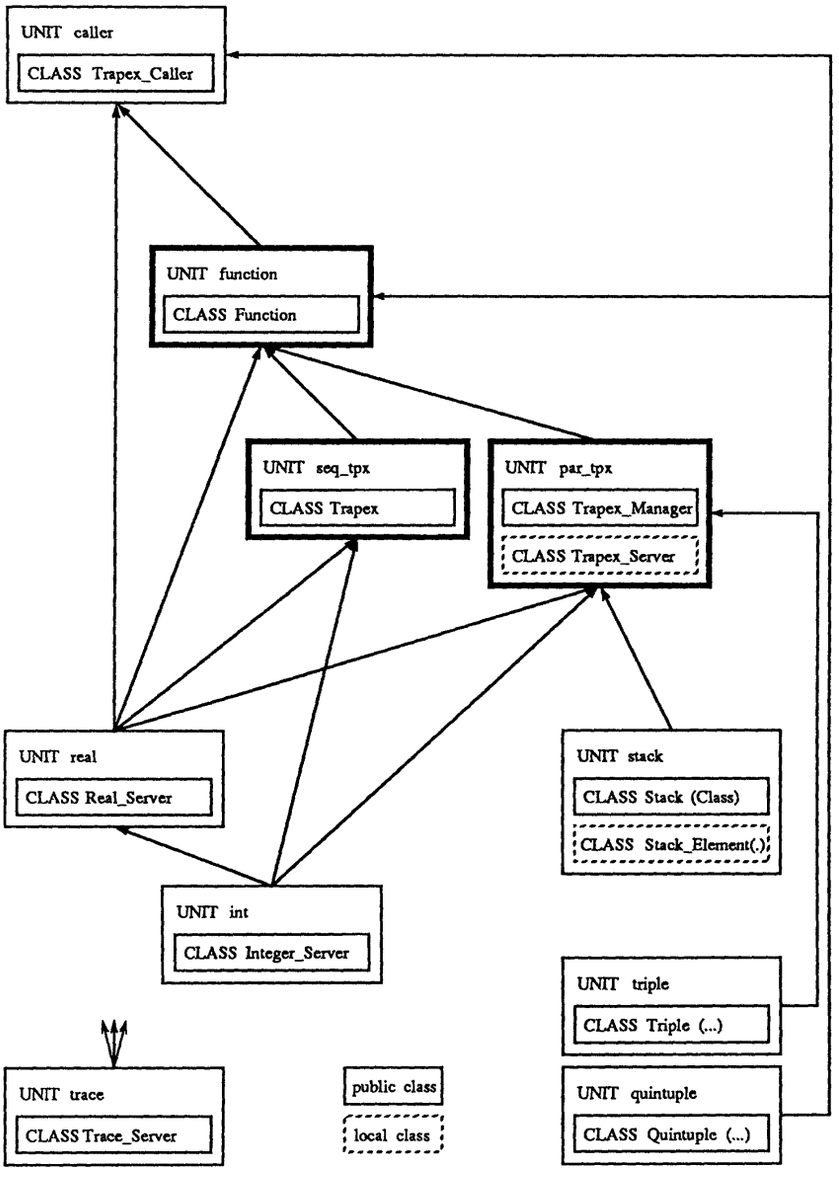


Bild 1

Der Übersicht wegen ist zwischen Definitions- und Implementationsmodulen nicht näher unterschieden worden. Lokale Klassendeklarationen, d.h. solche, die keine exportierte Schnittstellenbeschreibung im Definitionsmodul haben, sind durch eine gestrichelte Umrahmung gekennzeichnet. Man beachte den wechselseitigen Import zwischen den Modulen 'function' und '`seq_tpx`' bzw. '`par_tpx`'.
Es folgt eine Kurzbeschreibung der Module:

— 'caller' veranlaßt die Instantiierung einer — vom Benutzer ausgewählten — Testfunktion und ruft anschließend deren Methode 'trapex' mit Eingabeparametern auf;

— 'function' implementiert den ADT "Funktionen der Signatur $R \rightarrow R$" (s. Kapitel 3.1 und 3.2) und enthält die Zuordnungsvorschriften der Testfunktionen;

— '`seq_tpx`' stellt für 'function' die sequentielle Implementierung von TRAPEX zur Verfügung,

— '`seq_tpx`' enthält die Klassen '`Trapex_Manager`' und '`Trapex_Server`' der parallelen Implementierung von TRAPEX (s. Kapitel 3.3);

— 'real' und 'int' enthalten Funktionen der zugehörigen mathematischen Klassen, welche nicht Teil der POOL–T Standardklassen 'Real' und 'Integer' sind (s. Kapitel 4.3);

— 'triple' und 'quintuple' implementieren den zugehörigen abstrakten Datentyp (s. Kapitel 4.4);

— 'stack' implementiert den ADT 'Stack';

— 'trace' ist ein untergeordnetes Modul, welches Methoden für eine Textausgabe im Falle eines Fehlers oder Programmabruchs bereitstellt.

```
## ************************************************************************
## *                                                                      *
## * PROJECT: trapex                                                      *
## * VERSION: trapex05                                                    *
## * FILE:    function.spec                                               *
## * CHANGED:                                                            *
## *   M.G., 11.09.90                                                     *
## *                                                                      *
## ************************************************************************

SPECIFICATION UNIT function

USE trace, quintuple

CLASS Function

  METHOD expr () String

  METHOD at (x : Real) Real

  METHOD size () Integer

  ##
  ## method: trapex (t, tend, eps, hmax, h) (status, y, error, nstep, f_eval)
  ##
  ## input parameter:
  ## t          > Real          actual position (orginal: transient)
  ## tend       > Real          final point of integration
  ## eps        > Real          relative precision
  ## hmax       > Real          maximum permitted stepsize,
  ## h          > Real          stepsize proposal (orginal: transient)
  ##
  ## result parameter:
  ## status       Integer >     return code of trapex result (new)
  ## y            Real >        final value of integration (orginal: transient)
  ## error        Real >        global relative error estimate
  ## nstep        Integer >     number of integration steps
  ## f_eval       Integer >     number of function evaluations
  ##
```

```
  METHOD seq_trapex
    (t_begin,
     t_end,
     eps,
     max_h,
     first_h     : Real) Quintuple (Integer, Real, Real, Integer, Integer)

  METHOD par_trapex
    (t_begin,
     t_end,
     eps,
     max_h,
     first_h            : Real,
     server_nodes       : Node_Set)
    Quintuple (Integer, Real, Real, Integer, Integer)

  ## only for trapex servers:

  METHOD get_seq_trapex_result
    (status     : Integer,
     y,
     error      : Real,
     nr_steps   : Integer) Function

  METHOD get_par_trapex_result
    (status     : Integer,
     y,
     error      : Real,
     nr_steps,
     nr_evals   : Integer) Function

  ## Routines

  ROUTINE new
    (node_set     : Node_Set,
     trace        : Trace_Server,
     function_nr  : Integer) Function

  ROUTINE copy
    (node_set    : Node_Set,
     trace       : Trace_Server,
     f           : Function) Function

END Function
```

```
## ************************************************************************
## *                                                                      *
## * PROJECT: trapex                                                      *
## * VERSION: trapex05                                                    *
## * FILE:    function.impl                                               *
## * CHANGED:                                                            *
## *   M.G., 11.09.90                                                     *
## *                                                                      *
## ************************************************************************

IMPLEMENTATION UNIT function

USE trace, real, quintuple, seq_tpx, par_tpx

CLASS Function

  VAR
    ### servers
    trace       : Trace_Server,
    real        : Real_Server,
    seq_tpx     : Trapex,
    par_tpx     : Trapex_Manager,

    ### constants
    zero,
    one,
    two,
    three,
    four,
    ten,
    fifty,
    half,
    pi,
    pi_trunc    : Real,
    first_f_nr,
    last_f_nr   : Integer,

    ## run-time-constants
    f_nr        : Integer,              ## kind of function

    ## variables
    prev_eval_nr,                       ## nr of evaluations at trapex call
    act_eval_nr         : Integer,      ## actual nr of function evaluations
```

```
  trapex_result       : Quintuple (Integer, Real, Real, Integer, Integer)

## internal startup methods

METHOD instantiation
  (p_trace    : Trace_Server,
   p_f_nr     : Integer) Function :
LOCAL
  r : Function
IN
  trace       <- p_trace;
  f_nr        <- p_f_nr;
  real        <- Real_Server.new (Node.my_Node () ! get_singleton (),
                                  trace);
  zero        <- Real.float (0);
  one         <- Real.float (1);
  two         <- Real.float (2);
  three       <- Real.float (3);
  four        <- Real.float (4);
  ten         <- Real.float (10);
  fifty       <- Real.float (50);
  half        <- Real.create (5, -1);
  pi          <- real ! pi ();
  pi_trunc    <- Real.create (314159, -5);
  act_eval_nr <- 0;
  RETURN SELF
END instantiation

## internal function-expression methods

METHOD expr_f1 () String :
  RETURN "exp (x)"
END expr_f1

METHOD expr_f2 () String :
  RETURN "exp (2 * |x - 0.5|)"
END expr_f2

       ⋮

METHOD expr_f28 () String :
  RETURN "(1 - x * x) ** 0.5"
END expr_f28

## internal function-evaluation methods
```

```
METHOD eval_f1 (x : Real) Real :
  RETURN real ! exp (x)
END eval_f1

METHOD eval_f2 (x : Real) Real :
  RETURN real ! exp (two * real ! abs (x - half))
END eval_f2

        ⋮

METHOD eval_f28 (x : Real) Real :
  RETURN real ! power (one - x * x, half)
END eval_f28

## external methods

METHOD expr () String :
  ## == expression of function as string
LOCAL
  r : String
IN
  IF f_nr = 1 THEN
    r <- expr_f1 ()
  ELSIF f_nr = 2 THEN
    r <- expr_f2 ()
  ELSIF f_nr = 3 THEN

        ⋮

  ELSIF f_nr = 28 THEN
    r <- expr_f28 ()
  FI
  RETURN r
END expr

METHOD at (x : Real) Real :
  ## == f(x)
LOCAL
  r : Real
IN
  IF f_nr = 1 THEN
    r <- eval_f1 (x)
  ELSIF f_nr = 2 THEN
    r <- eval_f2 (x)
```

```
  ELSIF f_nr = 3 THEN

        ⋮

  ELSIF f_nr = 28 THEN
    r <- eval_f28 (x)
  FI;
  act_eval_nr <- act_eval_nr + 1
  RETURN r
END at

METHOD get_nr () Integer :
  RETURN f_nr
END get_nr

METHOD size () Integer :
  RETURN act_eval_nr
END size

METHOD seq_trapex
  (t_begin,
   t_end,
   eps,
   max_h,
   first_h    : Real) Quintuple (Integer, Real, Real, Integer, Integer) :
LOCAL
  trapex_finished : Boolean
IN
  prev_eval_nr <- act_eval_nr;
  IF Trapex.id (seq_tpx, NIL) THEN
    seq_tpx <- Trapex.new (Node.my_Node () ! get_singleton (), trace, SELF)
  FI;
  seq_tpx ! do_trapex (t_begin, t_end, eps, max_h, first_h);
  trapex_finished <- FALSE;
  DO ~trapex_finished THEN
    SEL
      ANSWER (at)
    OR
      ANSWER (get_seq_trapex_result) THEN
        trapex_finished <- TRUE
    LES
  OD
  RETURN trapex_result
END seq_trapex
```

```
METHOD par_trapex
  (t_begin,
   t_end,
   eps,
   max_h,
   first_h            : Real,
   server_nodes       : Node_Set)
  Quintuple (Integer, Real, Real, Integer, Integer) :
LOCAL
  trapex_finished : Boolean
IN
  prev_eval_nr <- act_eval_nr;
  IF Trapex_Manager.id (par_tpx, NIL) THEN
    par_tpx <- Trapex_Manager.new (Node.my_Node () ! get_singleton (),
                                   trace, SELF)
  FI;
  par_tpx ! do_trapex (t_begin, t_end, eps, max_h, first_h, server_nodes);
  trapex_finished <- FALSE;
  DO ~trapex_finished THEN
    SEL
      ANSWER (get_nr)
    OR
      ANSWER (get_par_trapex_result) THEN
        trapex_finished <- TRUE
    LES
  OD
  RETURN trapex_result
END par_trapex

## only for trapex servers:

METHOD get_seq_trapex_result
  (status     : Integer,
   y,
   error      : Real,
   nr_steps   : Integer) Function :
  ## SIDE-EFFECT TO: trapex_result
LOCAL
  not_used : Boolean
IN
  IF status = 0 THEN
    trapex_result
      <- Quintuple (Integer, Real, Real, Integer, Integer).new
```

```
          (Node.my_Node () ! get_singleton ())
          ! get_elements (status, y, error, nr_steps,
                          act_eval_nr - prev_eval_nr)
  ELSE
    trapex_result
      <- Quintuple (Integer, Real, Real, Integer, Integer).new
          (Node.my_Node () ! get_singleton ())
          ! get_elements (status, NIL, NIL, NIL, NIL)
  FI
  RETURN SELF
END get_seq_trapex_result

METHOD get_par_trapex_result
  (status     : Integer,
   y,
   error      : Real,
   nr_steps,
   nr_evals   : Integer) Function :
  ## SIDE-EFFECT TO: trapex_result
LOCAL
  not_used : Boolean
IN
  IF status = 0 THEN
    trapex_result
      <- Quintuple (Integer, Real, Real, Integer, Integer).new
          (Node.my_Node () ! get_singleton ())
          ! get_elements (status, y, error, nr_steps, nr_evals)
  ELSE
    trapex_result
      <- Quintuple (Integer, Real, Real, Integer, Integer).new
          (Node.my_Node () ! get_singleton ())
          ! get_elements (status, NIL, NIL, NIL, NIL)
  FI
  RETURN SELF
END get_par_trapex_result

## Routines

ROUTINE new
  (node_set    : Node_Set,
   trace       : Trace_Server,
   function_nr : Integer) Function :
LOCAL
  r : Function
```

```
  IN
    IF (1 <= function_nr) & (function_nr <= 28) THEN
      r <- NEW  ##$ ALLOC node_set;
      r ! instantiation (trace, function_nr)
    ELSE
      r <- NIL
    FI
    RETURN r
  END new

  ROUTINE copy
    (node_set    : Node_Set,
     trace       : Trace_Server,
     f           : Function) Function :
    RETURN NEW  ##$ ALLOC node_set
      ! instantiation (trace, f ! get_nr ())
  END copy

  BODY
    ANSWER (instantiation);
    DO TRUE THEN
      ANSWER (expr, at, get_nr, size, seq_trapex, par_trapex)
    OD

END Function
```

```
## ************************************************************************
## *                                                                      *
## * PROJECT: trapex                                                      *
## * VERSION: trapex05                                                    *
## * FILE:    par_tpx.spec                                                *
## *   M.G., 11.09.90                                                     *
## *                                                                      *
## ************************************************************************

SPECIFICATION UNIT par_tpx

USE trace, function

CLASS Trapex_Manager

  METHOD do_trapex
    (t_begin,
     t_end,
     eps,
     max_h,
     first_h            : Real,
     server_nodes       : Node_Set) Trapex_Manager

  ROUTINE new
    (node_set    : Node_Set,
     trace       : Trace_Server,
     f           : Function) Trapex_Manager

END Trapex_Manager
```

```
  ## ************************************************************************
  ## *                                                                      *
  ## * PROJECT: trapex                                                      *
  ## * VERSION: trapex05                                                    *
  ## * FILE:    par_tpx.impl                                                *
  ## * CHANGED:                                                            *
  ## *   M.G., 11.09.90                                                     *
  ## *                                                                      *
  ## ************************************************************************

IMPLEMENTATION UNIT par_tpx

USE trace, real, int, triple, stack, function

  ## short description of POOL-T variables:
  ##    C       == constant
  ##    RTC     == run time constant
  ##    IP      == instantiated by an input parameter of a public method
  ##    TP      == related to a transient parameter of a public method
  ##    OP      == instantiates an output parameter of a public method
  ##    CSE     == names a common sub expression (substituteable)
  ##    IV      == induction variable in loop (range in parentheses)

CLASS Trapex_Manager

  ## ************************************************************************
  ## *                        constants & variables                         *
  ## ************************************************************************

  VAR
    ## trapex servers
    trace       : Trace_Server,         ## trace system
    stack                               ## intervall stack
                : Stack (Triple (Real, Real, Real)),
    f           : Function,             ## RTC, IP, integrand

    ## some Real numbers
    zero,

    ## trapex Real variables
    error,                              ## OP, global relative error estimat
    y           : Real,                 ## OP, actual result of integration
```

```
  ## trapex Integer variables
  nr_servers,                          ## IP, number of trapex servers
  nr_evals,                            ## OP, number of function evaluations
  nr_steps,                            ## OP, number of steps
  status      : Integer,               ## OP, status return code, new

  ## trapex Boolean variables
  computation_finished : Boolean       ## becomes TRUE if computation is done

## ************************************************************************
## *                            instantiation                             *
## ************************************************************************

METHOD instantiation
  (p_trace    : Trace_Server,
   p_f        : Function) Trapex_Manager :
LOCAL
  not_used : Boolean
IN
  trace       <- p_trace;
  f           <- p_f;
  stack       <- Stack (Triple (Real, Real, Real)).new
                   (Node.my_Node () ! get_singleton (), trace);
  zero        <- Real.float (0)
  RETURN SELF
END instantiation

## ************************************************************************
## *                            local methods                             *
## ************************************************************************

METHOD instantiate_servers
  (eps,
   max_h              : Real,
   server_nodes       : Node_Set)  Trapex_Manager :
  ## SIDE-EFFECT TO: nr_servers
LOCAL
  last_node_nr,
  i           : Integer       ## IV (0..last_node_nr)
IN
  last_node_nr <- Node.number_of_nodes () - 1;
  nr_servers   <- 0;
  i <- 0;
```

```
  DO i <= last_node_nr THEN
    IF server_nodes ! contains (Node.from_Int (i)) THEN
      Trapex_Server.new (i, trace, SELF, f, eps, max_h);
      nr_servers <- nr_servers + 1
    FI;
    i <- i + 1
  OD
  RETURN SELF
END instantiate_servers

METHOD initialize_stack
  (t_begin,
   t_end,
   first_h     : Real) Trapex_Manager :
LOCAL
  i            : Integer,              ## IV (1..nr_servers)
  h,                                   ## CSE, intervall division stepsize
  t,                                   ## left border of actual sub-intervall
  tn           : Real,                 ## right border of actual sub-intervall
  triple       : Triple (Real, Real, Real)
IN
  ## !!! {t_end > t_begin} !!!
  h <- (t_end - t_begin) / Real.float (nr_servers);
  tn <- t_begin;
  i <- 1;
  DO i < nr_servers THEN
    t  <- tn;
    tn <- tn + h;
    triple <- Triple (Real, Real, Real).new
                (Node.my_Node () ! get_singleton ())
                ! get_elements (t, tn, first_h);
    stack ! push (triple);
    i <- i + 1
  OD;
  triple <- Triple (Real, Real, Real).new
              (Node.my_Node () ! get_singleton ())
              ! get_elements (tn, t_end, first_h);
  stack ! push (triple)
  RETURN SELF
END initialize_stack

METHOD manage_servers () Integer :
  ## SIDE-EFFECT TO: nr_working_servers, status, computation_finished
LOCAL
```

```
  nr_working_servers : Integer        ## number of busy trapex servers
IN
  nr_working_servers   <- 0;
  status               <- 0;
  computation_finished <- FALSE;
  DO ~computation_finished THEN
    SEL
      #stack > 0 ANSWER (drop_intervall) THEN
        nr_working_servers <- nr_working_servers + 1
    OR
      ANSWER (get_intervall)
    OR
      ANSWER (get_result) THEN
        nr_working_servers <- nr_working_servers - 1
    LES;
    computation_finished
      <- (#stack = 0 & nr_working_servers = 0) | status > 0
  OD
  RETURN status
END manage_servers

METHOD finitialize_servers () Trapex_Manager :
LOCAL
  i : Integer
IN
  i <- nr_servers;
  DO i > 0 THEN
    ANSWER ALL;
    i <- i - 1
  OD
  RETURN SELF
END finitialize_servers


## ************************************************************************
## *                          public methods                              *
## ************************************************************************

METHOD drop_intervall () Triple (Real, Real, Real) :
LOCAL
  r : Triple (Real, Real, Real)
IN
  ## pop intervall from stack or terminate server
  IF ~computation_finished THEN
    r <- stack ! pop ()
```

```
  ELSE
    r <- NIL
  FI
  RETURN r
END drop_intervall

METHOD get_intervall
  (t_begin,
   t_end,
   new_h : Real) Trapex_Manager :
LOCAL
  r : Trapex_Manager
IN
  IF ~computation_finished THEN
    r <- SELF
  ELSE
    r <- NIL
  FI
  RETURN r
POST
  ## push intervall on stack
  IF ~computation_finished THEN
    stack ! push (Triple (Real, Real, Real).new
                  (Node.my_Node () ! get_singleton ())
                  ! get_elements (t_begin, t_end, new_h));
  FI
END get_intervall

METHOD get_result
  (p_status   : Integer,
   p_y,
   p_error    : Real,
   p_nr_steps,
   p_nr_evals : Integer) Trapex_Manager :
  ## SIDE-EFFECT TO: status, y, error, nr_steps, nr_evals
LOCAL
  r : Trapex_Manager
IN
  IF ~computation_finished THEN
    r <- SELF
  ELSE
    r <- NIL
  FI
  RETURN r
```

```
POST
  ## evaluate result
  IF ~computation_finished THEN
    IF p_status = 0 THEN
      y           <- y + p_y;
      error       <- error + p_error;
      nr_steps    <- nr_steps + p_nr_steps;
      nr_evals    <- nr_evals + p_nr_evals
    ELSE
      status      <- p_status
    FI
  FI
END get_result

METHOD do_trapex
  (t_begin,
   t_end,
   eps,
   max_h,
   first_h            : Real,
   server_nodes       : Node_Set) Trapex_Manager :
LOCAL
   new_server_nodes : Node_Set
IN
  RETURN SELF
POST
  new_server_nodes <- server_nodes - Node.no_Node ();
  IF #new_server_nodes > 0 THEN
    instantiate_servers (eps, max_h, new_server_nodes);
    initialize_stack (t_begin, t_end, first_h);

    ## initialisation of output parameters
    y           <- zero;
    error       <- zero;
    nr_steps    <- 0;
    nr_evals    <- 0;

    status <- manage_servers ();
    finitialize_servers ();
    f ! get_par_trapex_result (status, y, error, nr_steps, nr_evals)
  ELSE
    trace ! fail ("Trapex_Manager$do_trapex", "no physical nodes specified")
  FI
END do_trapex
```

```
  ROUTINE new
    (node_set    : Node_Set,
     trace       : Trace_Server,
     f           : Function) Trapex_Manager :
    RETURN NEW  ##$ ALLOC node_set
      ! instantiation (trace, f)
  END new

  BODY
    ANSWER (instantiation);
    DO TRUE THEN
      ANSWER (do_trapex)
    OD

END Trapex_Manager

CLASS Trapex_Server

  ## ************************************************************************
  ## *                        constants & variables                         *
  ## ************************************************************************

  VAR
    ## trapex servers
    trace       : Trace_Server,         ## trace system
    int         : Integer_Server,       ## Integer extension
    real        : Real_Server,          ## Real extension
    manager     : Trapex_Manager,       ## RTC, IP, my trapex manager
    f           : Function,             ## RTC, IP, integrand

    ## some Real numbers
    zero,
    one,
    two,
    half,
    one0one,

    ## trapex Real constants
    fmin,                               ## C, minimum of fcm
    ro,                                 ## C,
    safe,                               ## C,
    scale,                              ## C,
    small,                              ## C, square-root (Real.min ())
```

```
eph,                                  ## RTC, (= ro * eps)
eps          : Real,                  ## RTC, IP, relative precision

## trapex Integer constants
km,                                   ## RTC, max column number (1..6,<- eps)
jm,                                   ## RTC, associated max row number
                                      ## (2..7, jm = km + 1, <- eps)
nstmax,                               ## C, max permitted number of steps
jrmax        : Integer,               ## C, max permitted nr. of reductions

## trapex Array constants
nj           : Array (Integer),       ## C, stepsize sequence
a            : Array (Real),          ## C, associated normal. work per step
d,                                    ## C (<- nj), extrapolation
al           : Array (Array (Real)), ## RTC (<- eph, a)

## trapex Real variables
t,                                    ## TP, left border of act. sub-interv.
tn,                                   ## right border of actual sub-intervall
tend,                                 ## IP, right border of integration
teps,                                 ## not a RTC anymore (<- t, tend, eps)
h,                                    ## IP, stepsize
hr,                                   ## OP, last h
h1,                                   ## tail intervall
q,                                    ## quotient h1 / h
hmax,                                 ## IP, actual maximum of h
hmaxu,                                ## RTC (= |initial hmax|)
err,                                  ## extrapolated normalized error
errh,                                 ## extrapolated error
error,                                ## OP, global relative error estimate
fc,                                   ## error for stepsize and order control
fcm,                                  ## actual minimum of fc
fco,                                  ## error for stepsize and order control
omj,                                  ## normalized work for order control
omjo,                                 ## normalized work for order control
red,                                  ## factor of stepsize reduction
y,                                    ## OP, actual result of integration
yr,                                   ## trapezoidal sum at j
yn,                                   ## sum of Stuetzpunkthoehen at j
yl,                                   ## sum of Stuetzpunkthoehen at j
yl1,                                  ## yl of previous j
zq1,                                  ## f (tn)
zq           : Real,                  ## f (t)
```

```
  ## trapex Integer variables
  k,                                  ## IV (1..km), actual column
  ko,                                 ## (1..km), optimal column
  koh,                                ## (1..km), previous optimal column
  j,                                  ## IV (2..jm), actual order
  jo,                                 ## (2..jm), optimal order
  joh,                                ## (2..jm), previous optimal order
  jred,                               ## IV (0..jrmax), number of reductions
  nstep,                              ## IV (0..nstmax), OP, number of steps
  status      : Integer,              ## OP, status return code, new

  ## trapex Array variables
  incr,                               ## init_step & possible.. & step_red..
  nred        : Array (Integer),      ## possible.. & stepsize_reduction
  dt          : Array (Real)          ## extrapolation

## ************************************************************************
## *                          instantiation                               *
## ************************************************************************

METHOD instantiation
  (p_trace    : Trace_Server,
   p_manager  : Trapex_Manager,
   p_f        : Function,
   p_eps,
   p_max_h    : Real) Trapex_Server :
  ## SIDE-EFFECT TO: trace, manager, int, real,
  ##                 zero, one, two, half, ono0one,
  ##                 km, jm, nstmax, jrmax, fmin,
  ##                 ro, safe, scale, small,
  ##                 nj, a
LOCAL
  not_used : Boolean
IN
  RETURN SELF
POST
  ## instantiation of trapex servers
  trace       <- p_trace;
  manager     <- p_manager;                  ## my trapex manager
  int         <- Integer_Server.new
                   (Node.my_Node () ! get_singleton (), trace);
  real        <- Real_Server.new
                   (Node.my_Node () ! get_singleton (), trace);
```

```
  ## instantiation of some Real numbers
  zero        <- Real.float (0);              ## 0.0
  one         <- Real.float (1);              ## 1.0
  two         <- Real.float (2);              ## 2.0
  half        <- Real.create (5, -1);         ## 0.5
  oneOone     <- Real.create (101, -2);       ## 1.01

  ## instantiation of integer constants
  km          <- 6;                           ## 1 <= km <= 6
  jm          <- km + 1;                      ## 2 <= jm <= 7, jm = km + 1
  nstmax      <- 1000;                        ## 1000
  jrmax       <- 5;                           ## 5

  ## instantiation of real constants
  fmin        <- Real.create (1, -2);         ## 0.01
  ro          <- Real.create (25, -2);        ## 0.25
  safe        <- Real.create (5, -1);         ## 0.5
  scale       <- Real.create (1, 0);          ## 1.0, changed (^ original)
  small       <- Real.create (1, -35);        ## 1.0E-35

  ## instantiation of nj, a
  nj          <- Array (Integer).new (1, 7)   ##$ ALLOC HERE
                   ! put (1, 1)               ## Bulirsch sequence
                   ! put (2, 2)
                   ! put (3, 3)
                   ! put (4, 4)
                   ! put (5, 6)
                   ! put (6, 8)
                   ! put (7, 12);
  a           <- Array (Real).new (1, 7)      ##$ ALLOC HERE
                   ! put (1, one)             ## associated work per step
                   ! put (2, Real.float (2))
                   ! put (3, Real.float (4))
                   ! put (4, Real.float (6))
                   ! put (5, Real.float (8))
                   ! put (6, Real.float (12))
                   ! put (7, Real.float (16));

  ## evaluation of parameters f, eps, max_h
  evaluate_f_eps_hmax (p_f, p_eps, p_max_h);

  ## instantiation of al, d
  instantiate_al_d ()
END instantiation
```

```
## ************************************************************************
## *                        trapex control methods                        *
## ************************************************************************

METHOD compute_trapex () Boolean :
LOCAL
  r : Boolean
IN
  IF ~boundaries_near_epmach () THEN
    IF ~stepsize_too_high () THEN
      init_computation ();
      r <- basic_steps_done ()
    ELSE
      fail_exit (2);
      r <- FALSE
    FI
  ELSE
    fail_exit (1);
    r <- FALSE
  FI
  RETURN r
END compute_trapex

METHOD basic_steps_done () Boolean :
LOCAL
  r,
  next_step   : Boolean
IN
  next_step <- TRUE;
  DO next_step THEN
    next_step <- FALSE;
    init_step ();
    IF step_computed () THEN
      preparations_for_next_basic_step ();
      IF ~too_many_steps () THEN
        stepsize_prediction ();
        IF ~stepsize_estimate_too_small () THEN
          IF ~boundaries_near_epmach () THEN
            IF ~stepsize_too_high () THEN
              next_step <- TRUE
            ELSE
              solution_exit ();
              r <- TRUE
```

```
          FI
        ELSE
          solution_exit ();
          r <- TRUE
        FI
      ELSE
        fail_exit (5);
        r <- FALSE
      FI
    ELSE
      fail_exit (4);
      r <- FALSE
    FI
  ELSE
    r <- FALSE
  FI
 OD
 RETURN r
END basic_steps_done

METHOD step_computed () Boolean :
LOCAL
 r,
 step_red    : Boolean
IN
 step_red <- TRUE;
 DO step_red THEN
   step_red <- FALSE;
   init_sub_intervall_computation ();
   IF ~sub_intervall_computed () THEN
     stepsize_reduction ();
     IF ~too_many_reductions () THEN
       IF ~tail_intervall_too_small () THEN
         IF tail_intervall_shared () THEN
           step_red <- TRUE
         ELSE
           fail_exit (6);
           r <- FALSE
         FI
       ELSE
         step_red <- TRUE
       FI
     ELSE
       fail_exit (3);
```

```
        r <- FALSE
      FI
    ELSE
      r <- TRUE
    FI
  OD
  RETURN r
END step_computed

METHOD sub_intervall_computed () Boolean :
  ## SIDE-EFFECT TO: j
LOCAL
  extrapol,
  r           : Boolean
IN
  j <- 2;
  extrapol <- TRUE;
  DO extrapol & j <= jm THEN
    trapezoidal_sum ();
    extrapolation ();
    optimal_order_determination ();
    IF last_sub_intervall () THEN
      IF convergence () THEN
        extrapol <- FALSE;
        r <- TRUE
      ELSE
        IF convergence_window_reached () THEN
          IF convergence_monitor () THEN
            j <- j + 1
          ELSE
            extrapol <- FALSE
          FI;
          r <- FALSE
        ELSE
          j <- j + 1
        FI
      FI
    ELSE
      IF convergence_window_reached () THEN
        IF convergence () THEN
          possible_increase_of_order ();
          extrapol <- FALSE;
          r <- TRUE
        ELSE
```

```
          IF convergence_monitor () THEN
            j <- j + 1
          ELSE
            extrapol <- FALSE
          FI;
          r <- FALSE
        FI
      ELSE
        j <- j + 1
      FI
    FI
  OD
  RETURN r
END sub_intervall_computed

METHOD convergence_monitor () Boolean :
  ## SIDE-EFFECT TO: red
LOCAL
  jk  : Integer,      ## CSE
  r   : Boolean
IN
  jk <- int ! min (km, joh);
  IF k >= jk THEN
    red <- one / fco;
    r <- FALSE
  ELSE
    IF ko >= koh THEN
      red <- one / fco
    ELSE
      red <- al@koh@ko / fco
    FI;
    IF al@jk@ko < fco THEN
      r <- FALSE
    ELSE
      r <- TRUE
    FI
  FI
  RETURN r
END convergence_monitor
```

```
## ************************************************************************
## *                      trapex computation methods                      *
## ************************************************************************

METHOD evaluate_f_eps_hmax
  (p_f         : Function,
   p_eps,
   p_hmax      : Real) Trapex_Server :
  ## SIDE-EFFECT TO: f, eps, eph, hmaxu, hmax
  ##
  ## instantiation of f, eps, eph, hmaxu, hmax
  f            <- Function.copy
                    (Node.my_Node () ! get_singleton (),
                     trace, p_f);
  eps          <- p_eps;
  eph          <- ro * eps;
  hmaxu        <- real ! abs (p_hmax);
  hmax         <- hmaxu
  RETURN SELF
END evaluate_f_eps_hmax

METHOD evaluate_t_tend_h
  (p_t,
   p_tend,
   p_h         : Real) Trapex_Server :
  ## SIDE-EFFECT TO: t, tend, teps, h1, h, q
  ##
  ## instantiation of t, tend, teps, h1, h, q
  t            <- p_t;
  tend         <- p_tend;
  teps         <- (real ! abs (t) + real ! abs (tend)) * real ! eps ();
  h1           <- tend - t;
  h            <- real ! min (p_h, h1); ## new ! pre-conditions for t, tend
  q            <- h1 / h
  RETURN SELF
END evaluate_t_tend_h

METHOD instantiate_al_d () Trapex_Server :
  ## SIDE-EFFECT TO: al, km (!), jm (!), d
LOCAL
  r, s, u, v, w,                       ## CSEs
  s_old                : Real,
  j,                                   ## IV (2..jm)
```

```
  k                    : Integer,      ## IV (1..km)
  exit_j_loop,
  exit_k_loop          : Boolean
IN
  ## instantiation of al (al <- eph, a)        ## (2..jm-1, 1..km-1)
  al <- Array (Array (Real)).new (2, 6)        ##$ ALLOC HERE
  ;
  s_old <- zero;
  exit_j_loop <- FALSE;
  j <- 2;
  DO ~exit_j_loop & j < jm THEN                ## j <= jm -1
    al ! put (j, Array (Real).new (1, 5)       ##$ ALLOC HERE
             );
    v <- a@(j + 1);
    w <- (v - a@1) + one;
    k <- 1;
    exit_k_loop <- FALSE;
    DO ~exit_k_loop & k < j THEN               ## k <= j - 1
      r <- a@(k + 1);
      u <- real ! power (eph, (r - v) / (w * Real.float (2 * k + 1)));
      (al@j) ! put (k, u);
      s <- u * r;
      IF k > 1 & s * oneOone > s_old THEN
        exit_k_loop <- TRUE;
        exit_j_loop <- TRUE
      ELSE
        s_old <- s;
        k <- k + 1
      FI
    OD;
    IF ~exit_j_loop THEN
      IF v * oneOone > s_old THEN
        exit_j_loop <-TRUE
      ELSE
        j <- j + 1
      FI
    FI
  OD;

  ## re-instantiation of km, jm
  IF exit_j_loop THEN
    jm <- int ! max (2, j - 1);
    km <- jm - 1
  FI;
```

```
  ## instantiation of d (d <- nj)              ## (2..jm, 1..km)
  d <- Array (Array (Real)).new (2, jm)        ##$ ALLOC HERE
  ;
  j <- 2;
  DO j <= jm THEN
    v <- Real.float (nj@j);
    d ! put (j, Array (Real).new (1, km)       ##$ ALLOC HERE
            );
    k <- 1;
    DO k < j THEN
      w <- v / Real.float (nj@k);
      (d@j) ! put (k, w * w);
      k <- k + 1
    OD;
    j <- j + 1
  OD
  RETURN SELF
END instantiate_al_d

METHOD init_computation () Trapex_Server :
  ## SIDE-EFFECT TO: incr, nred, dt,
  ##                 nstep, koh, joh, omjo, zq, y, error, hr
LOCAL
  k                    : Integer       ## IV (1..km)
IN
  ## initialisation of incr, nred, dt
  incr         <- Array (Integer).new (1, jm)          ##$ ALLOC HERE
  ;                                                    ## 1..7 wg. 1..jm
  nred         <- Array (Integer).new (1, km)          ##$ ALLOC HERE
  ;                                                    ## 1..6 wg. 1..km
  dt           <- Array (Real).new (1, jm)             ##$ ALLOC HERE
  ;                                                    ## 1..7
  k <- 1;
  DO k <= km THEN
    incr       ! put (k, 0);
    nred       ! put (k, 0);
    dt         ! put (k, zero);
    k <- k + 1
  OD;
  incr         ! put (jm, -1);
  dt           ! put (jm, zero);

  ## initialisation of nstep, koh, joh, omjo, zq, y, error, hr
```

```
  nstep       <- 0;
  koh         <- km;
  joh         <- jm;
  omjo        <- zero;
  zq          <- f@(t);
  y           <- zero;        ## original version: y is transient parameter
  error       <- zero;
  hr          <- hmax
  RETURN SELF
END init_computation

METHOD init_step () Trapex_Server :
  ## SIDE-EFFECT TO: hr, h, hmax, jred, incr
LOCAL
  not_used : Boolean
IN
  IF q < oneOone THEN
    hr <- h;
    h  <- h1
  FI;
  hmax <- real ! min (h1, hmaxu);
  jred <- 0;
  k <- 1;
  DO k <= km THEN
    incr ! put (k, incr@k + 1);
    k <- k + 1
  OD
  RETURN SELF
END init_step

METHOD init_sub_intervall_computation () Trapex_Server :
  ## SIDE-EFFECT TO: tn, fcm, zq1, yn, yl, yr, dt
LOCAL
  not_used : Boolean
IN
  ## init right border
  IF h1 = h THEN
    tn <- tend
  ELSE
    tn <- t + h
  FI;

  ## init order control
  fcm <- real ! max (real ! abs (h) / hmax, fmin);
```

```
  ## first trapezoidal sum
  zq1 <- f@(t + h);    ## <- f@(tn) ?
  yn <- zero;
  yl <- (zq1 + zq) * half;
  yr <- yl * h;        ## <- yl * h / Real.float (nj@1)
  dt ! put (1, yr)
  RETURN SELF
END init_sub_intervall_computation

METHOD trapezoidal_sum () Trapex_Server :
  ## SIDE-EFFECT TO: yr, yn, yl, yl1
LOCAL
  fi,                  ## CSE
  g    : Real,         ## CSE
  i,                   ## IV
  m    : Integer       ## CSE
IN
  m <- nj@j;
  g <- h / Real.float (m);
  IF (m // 3) = 0 THEN
    i <- 1;
    DO i <= m THEN
      fi <- Real.float (i);
      yn <- yn + f@(t + fi * g) + f@(tn - fi * g);
      i <- i + 6
    OD;
    yr <- (yn + yl1) * g
  ELSE
    yl1 <- yl;
    i <- 1;
    DO i <= m THEN
      fi <- Real.float (i);
      yl <- yl + f@(t + fi * g);
      i <- i + 2
    OD;
    yr <- yl * g
  FI
  RETURN SELF
END trapezoidal_sum

METHOD extrapolation () Trapex_Server :
  ## SIDE-EFFECT TO: dt, errh, yr
LOCAL
```

```
  c,                  ##
  ta,                 ## CSE
  u,                  ## CSE
  v    : Real,        ##
  jk,                 ## CSE
  k    : Integer      ## IV (2..jm)
IN
  v <- dt@1;
  c <- yr;
  dt ! put (1, c);
  ta <- c;
  k <- 2;
  DO k <= j THEN
    jk <- (j - k) + 1;
    u  <- (c - v) / (d@j@jk - one);
    c  <- d@j@jk * u;
    v  <- dt@(k);
    dt ! put (k, u);
    ta <- u + ta;
    k <- k + 1
  OD;
  errh <- real ! abs (u);
  yr <- ta;
  ta <- real ! abs (y + yr);
  IF ta < scale THEN  ## orginal version: ta < small
    ta <- one
  FI;
  err <- (errh / ta) / eph
  RETURN SELF
END extrapolation

METHOD optimal_order_determination () Trapex_Server :
  ## SIDE-EFFECT TO: k, fc, fco, omj, omjo, ko, jo
LOCAL
  not_used : Boolean
IN
  k   <- j - 1;
  fc  <- real ! max (real ! power (err, one / Real.float (j + k)), fcm);
  omj <- fc * a@j;
  IF (j = 2 | omj * one0one <= omjo) & k <= joh THEN
    ko   <- k;
    jo   <- j;
    omjo <- omj;
    fco  <- fc
```

```
  FI
  RETURN SELF
END optimal_order_determination

METHOD possible_increase_of_order () Trapex_Server :
  ## SIDE-EFFECT TO: nred, fc, fco, ko, jo
LOCAL
  not_used : Boolean
IN
  IF ko >= k & incr@j >= 0 THEN
    IF nred@ko > 0 THEN
      nred ! put (ko, nred@ko - 1)
    FI;
    IF j < jm THEN
      fc <- real ! max (fco / al@j@k, fcm);
      IF a@(j + 1) * fc * oneOone <= omjo THEN
        fco <- fc;
        ko  <- jo;
        jo  <- jo + 1
      FI
    FI
  FI
  RETURN SELF
END possible_increase_of_order

METHOD stepsize_reduction () Trapex_Server :
  ## SIDE-EFFECT TO: h, red, jred, nred, incr
LOCAL
  l : Integer          ## IV
IN
  red <- red * safe;
  h   <- h * red;
  IF ~first_step () THEN
    nred ! put (koh, nred@koh + 1);
    l <- koh;
    DO l <= km THEN    ## km - koh - 1 executions
      incr ! put (l, -2 - nred@koh);
      l <- l + 1
    OD
  FI;
  jred <- jred + 1
  RETURN SELF
END stepsize_reduction
```

```
METHOD preparations_for_next_basic_step () Trapex_Server :
  ## SIDE-EFFECT TO: t, h1, y, zq, error, nstep
LOCAL
  not_used : Boolean
IN
  t  <- tn;
  h1 <- tend - t;
  y  <- y + yr;
  zq <- zq1;
  error <- error + errh;
  nstep <- nstep + 1
  RETURN SELF
END preparations_for_next_basic_step

METHOD stepsize_prediction () Trapex_Server :
  ## SIDE-EFFECT TO: h, hr, q, koh, joh
LOCAL
  not_used : Boolean
IN
  IF fco ~= fcm THEN
    hr <- h
  FI;
  h   <- h / fco;
  q   <- h1 / h;
  koh <- ko;
  joh <- koh + 1
  RETURN SELF
END stepsize_prediction

METHOD solution_exit () Trapex_Server :
  ## SIDE-EFFECT TO: error, status
LOCAL
  not_used : Boolean
IN
  error  <- error / real ! max (real ! abs (y), small);
  status <- 0
  RETURN SELF
END solution_exit

METHOD fail_exit (error_code : Integer) Trapex_Server :
  ## SIDE-EFFECT TO: status
LOCAL
  not_used : Boolean
IN
```

```
  status <- error_code
  RETURN SELF
END fail_exit

## ************************************************************************
## *                          trapex test methods                         *
## ************************************************************************

METHOD tail_intervall_too_small () Boolean :
LOCAL
  r : Boolean
IN
  IF tend - tn <= two * h THEN
    r <- TRUE
  ELSE
    r <- FALSE
  FI
  RETURN r
END tail_intervall_too_small

METHOD last_sub_intervall () Boolean :
LOCAL
  r : Boolean
IN
  ## delete first condition ?
  IF tn > tend THEN
    trace ! fail ("Trapex_Manager$last_sub_intervall", "tn > tend")
  ELSIF tn = tend THEN
    r <- TRUE
  ELSE
    r <- FALSE
  FI
  RETURN r
END last_sub_intervall

METHOD first_step () Boolean :
LOCAL
  r : Boolean
IN
  IF nstep = 0 THEN
    r <- TRUE
  ELSE
    r <- FALSE
  FI
```

```
  RETURN r
END first_step

METHOD convergence_window_reached () Boolean :
LOCAL
  r : Boolean
IN
  IF j >= koh | first_step () THEN
    r <- TRUE
  ELSE
    r <- FALSE
  FI
  RETURN r
END convergence_window_reached

METHOD convergence () Boolean :
LOCAL
  r : Boolean
IN
  IF err * eph < eps THEN
    r <- TRUE
  ELSE
    r <- FALSE
  FI
  RETURN r
END convergence

METHOD boundaries_near_epmach () Boolean :
LOCAL
  r : Boolean
IN
  IF real ! abs (h1) <= teps THEN
    r <- TRUE
  ELSE
    r <- FALSE
  FI
  RETURN r
END boundaries_near_epmach

METHOD stepsize_too_high () Boolean :
LOCAL
  r : Boolean
IN
  IF
```

```
    q <= real ! eps ()
  THEN
    r <- TRUE
  ELSE
    r <- FALSE
  FI
  RETURN r
END stepsize_too_high

METHOD too_many_steps () Boolean :
LOCAL
  r : Boolean
IN
  IF nstep > nstmax THEN
    r <- TRUE
  ELSE
    r <- FALSE
  FI
  RETURN r
END too_many_steps

METHOD too_many_reductions () Boolean :
LOCAL
  r : Boolean
IN
  IF jred > jrmax THEN
    r <- TRUE
  ELSE
    r <- FALSE
  FI
  RETURN r
END too_many_reductions

METHOD stepsize_estimate_too_small () Boolean :
LOCAL
  r : Boolean
IN
  IF real ! abs (h) <= teps THEN
    r <- TRUE
  ELSE
    r <- FALSE
  FI
  RETURN r
END stepsize_estimate_too_small
```

```
## ************************************************************************
## *                        communication methods                         *
## ************************************************************************

METHOD intervall_received () Boolean :
  ## SIDE-EFFECT TO: t, tend, h
LOCAL
  r           : Boolean,
  intervall   : Triple (Real, Real, Real)
IN
  intervall <- manager ! drop_intervall ();
  IF ~Triple (Real, Real, Real).id (intervall, NIL) THEN
    evaluate_t_tend_h
      (intervall ! return_element1 (),
       intervall ! return_element2 (),
       intervall ! return_element3 ());
    r <- TRUE
  ELSE
    r <- FALSE
  FI
  RETURN r
END intervall_received

METHOD tail_intervall_shared () Boolean :
  ## SIDE-EFFECT TO: tend, h1, hmax, teps
LOCAL
  r : Boolean
IN
  IF ~Trapex_Manager.id (manager ! get_intervall (tn, tend, h), NIL) THEN
    tend <- tn;
    h1   <- tend - t;
    hmax <- real ! min (hmax, h1);
    teps <- (real ! abs (t) + real ! abs (tend)) * real ! eps ();
    r <- TRUE
  ELSE
    r <- FALSE
  FI
  RETURN r
END tail_intervall_shared

METHOD result_transferd () Boolean :
LOCAL
  r : Boolean
```

```
IN
  IF status = 0 THEN          ## solution
    IF ~Trapex_Manager.id
        (manager ! get_result (status, y, error, nstep, #f), NIL) THEN
      r <- TRUE
    ELSE
      r <- FALSE
    FI
  ELSIF status <= 5 THEN      ## failure
    IF ~Trapex_Manager.id
        (manager ! get_result (status, zero, zero, 0, 0), NIL) THEN
      r <- TRUE
    ELSE
      r <- FALSE
    FI
  ELSIF status = 6 THEN       ## abort
    r <- FALSE
  FI
  RETURN r
END result_transferd

## ************************************************************************
## *                        main control method                           *
## ************************************************************************

METHOD trapex_done () Boolean :
LOCAL
  r : Boolean
IN
  IF intervall_received () THEN
    compute_trapex ();
    IF result_transferd () THEN
      r <- TRUE
    ELSE
      r <- FALSE
    FI
  ELSE
    r <- FALSE
  FI
  RETURN r
END trapex_done

ROUTINE new
  (node_nr    : Integer,
```

```
      trace       : Trace_Server,
      manager     : Trapex_Manager,
      f           : Function,
      eps,
      max_h       : Real) Trapex_Server :
    RETURN NEW  ##$ ALLOC AT node_nr
      ! instantiation (trace, manager, f, eps, max_h)
  END new

  BODY
    ANSWER (instantiation);
    DO trapex_done () OD

END Trapex_Server
```

```
  ## ************************************************************************
  ## *                                                                      *
  ## * PROJECT: trapex                                                      *
  ## * VERSION: trapex05                                                    *
  ## * FILE:    real.spec                                                   *
  ## * CHANGED:                                                            *
  ## *   M.G., 11.09.90                                                     *
  ## *                                                                      *
  ## ************************************************************************

USE trace

SPECIFICATION UNIT real

CLASS Real_Server

  ## constant methods

  ## Real constants wrt. implementation of class Real

  METHOD max_real () Real
    ## == 1.7976931348623157e+308 (?),
    ## [Ay: (y ~= 'Infinity') -> (|y| <= max_real < 'Infinity')]

  METHOD min_real () Real
    ## == 2.2250738585072014E-308 (?),
    ## [Ay: (y ~= 0) -> (|y| >= min_real > 0)]

  METHOD eps () Real
    ## == 2e-15
    ## Real precision = 1e-15, [Axy: (|x| > 1e-15) -> (y ~= y + x) ?]

  ## Real constants of Irrational values

  METHOD e () Real
    ## == e

  METHOD ln2 () Real
    ## == ln (2)

  METHOD log_e () Real
    ## == lg (e)
```

```
METHOD pi () Real
  ## == pi

METHOD pi_by2 () Real
  ## == pi : 2

METHOD pi_by4 () Real
  ## == pi : 4

METHOD pi_by6 () Real
  ## == pi : 6

METHOD root2 () Real
  ## == square_root (2)

METHOD inv_root2 () Real
  ## == 1 : square_root (2)

## monadic methods

METHOD abs (x : Real) Real
  ## == |x|

METHOD exp (x : Real) Real
  ## == exp (x)

METHOD ln (x : Real) Real
  ## == ln (x), {x > 0}

METHOD sin (x : Real) Real
  ## == sin (x)

METHOD cos (x : Real) Real
  ## == cos (x)

METHOD sinh (x : Real) Real
  ## == sinh (x)

METHOD cosh (x : Real) Real
  ## == cosh (x)

## dyadic methods
```

```
  METHOD min (a, b : Real) Real
    ## == min (a, b)

  METHOD max (a, b : Real) Real
    ## == max (a, b)

  METHOD power (base, exp : Real) Real
    ## == base ** exp

  ## conversion methods

  METHOD string (x : Real) String
    ## == x as string representation, wrt. Real precision - 1

  ## routines

  ROUTINE new
    (node_set    : Node_Set,
     trace       : Trace_Server) Real_Server

END Real_Server
```

```
  ## ************************************************************************
  ## *                                                                      *
  ## * PROJECT: trapex                                                      *
  ## * VERSION: trapex05                                                    *
  ## * FILE:    int.spec                                                    *
  ## * CHANGED:                                                            *
  ## *   M.G., 11.09.90                                                     *
  ## *                                                                      *
  ## ************************************************************************

USE trace

SPECIFICATION UNIT int

CLASS Integer_Server

  ## constant methods

  METHOD max_int () Integer
    ## == 2 ** 30 - 1 = 1073741823 (10 Stellen)

  METHOD min_int () Integer
    ## == -2 ** 30 = -1073741824 (10 Stellen)

  ## monadic methods

  METHOD is_even (x : Integer) Boolean
    ## == x MOD 2 = 0

  METHOD is_odd (x : Integer) Boolean
    ## == x MOD 2 <> 0

  METHOD abs (x : Integer) Integer
    ## == |x|

  METHOD fac (x : Integer) Integer
    ## == fac (x)

  ## dyadic methods

  METHOD min (a, b : Integer) Integer
    ## == min (a, b)
```

```
  METHOD max (a, b : Integer) Integer
    ## == max (a, b)

  ## conversion methods

  METHOD char (x : Integer) Character
    ## == x as char representation, {0 <= x <= 9}

  METHOD string (x : Integer) String
    ## == x as string representation, leeding '0's are skipped

  ## routines

  ROUTINE new
    (node_set    : Node_Set,
     trace       : Trace_Server) Integer_Server

END Integer_Server
```

```
## ************************************************************************
## *                                                                      *
## * PROJECT: trapex                                                      *
## * VERSION: trapex05                                                    *
## * FILE:    triple.spec                                                 *
## * CHANGED:                                                            *
## *   M.G., 11.09.90                                                     *
## *                                                                      *
## ************************************************************************

SPECIFICATION UNIT triple

CLASS Triple (Class1, Class2, Class3)

  METHOD return_element1 () Class1

  METHOD return_element2 () Class2

  METHOD return_element3 () Class3

  METHOD get_elements
    (e1 : Class1,
     e2 : Class2,
     e3 : Class3) Triple (Class1, Class2, Class3)

  ROUTINE new (node_set : Node_Set) Triple (Class1, Class2, Class3)

END Triple
```

```
  ## ************************************************************************
  ## *                                                                      *
  ## * PROJECT: trapex                                                      *
  ## * VERSION: trapex05                                                    *
  ## * FILE:    quintuple.spec                                              *
  ## * CHANGED:                                                            *
  ## *   M.G., 11.09.90                                                     *
  ## *                                                                      *
  ## ************************************************************************

SPECIFICATION UNIT quintuple

CLASS Quintuple (Class1, Class2, Class3, Class4, Class5)

  METHOD return_element1 () Class1

  METHOD return_element2 () Class2

  METHOD return_element3 () Class3

  METHOD return_element4 () Class4

  METHOD return_element5 () Class5

  METHOD get_elements
    (e1 : Class1,
     e2 : Class2,
     e3 : Class3,
     e4 : Class4,
     e5 : Class5) Quintuple (Class1, Class2, Class3, Class4, Class5)

  ROUTINE new
    (node_set : Node_Set) Quintuple (Class1, Class2, Class3, Class4, Class5)

END Quintuple
```

```
## ************************************************************************
## *                                                                      *
## * PROJECT: trapex                                                      *
## * VERSION: trapex05                                                    *
## * FILE:    stack.spec                                                  *
## * CHANGED:                                                            *
## *   M.G., 11.09.90                                                     *
## *                                                                      *
## ************************************************************************

SPECIFICATION UNIT stack

USE trace

CLASS Stack (Element_Class)

  METHOD push (element : Element_Class) Stack (Element_Class)

  METHOD pop () Element_Class

  METHOD top () Element_Class

  METHOD size () Integer

  ROUTINE new
    (node_set   : Node_Set,
     trace      : Trace_Server) Stack (Element_Class)

END Stack
```

```
  ## ************************************************************************
  ## *                                                                      *
  ## * PROJECT: trapex                                                      *
  ## * VERSION: trapex05                                                    *
  ## * FILE:    trace.spec                                                  *
  ## * CHANGED:                                                            *
  ## *   M.G., 11.09.90                                                     *
  ## *                                                                      *
  ## ************************************************************************

SPECIFICATION UNIT trace

CLASS Trace_Server

  METHOD entry (method : String) Trace_Server

  METHOD exit (method : String) Trace_Server

  METHOD msg (name, msg : String) Trace_Server

  METHOD msg_int (name, msg : String, value : Integer) Trace_Server

  METHOD msg_real (name, msg : String, value : Real) Trace_Server

  METHOD warn (name, warn : String) Trace_Server

  METHOD warn_int (name, warn : String, value : Integer) Trace_Server

  METHOD warn_real (name, warn : String, value : Real) Trace_Server

  METHOD fail (name, fail : String) Trace_Server

  METHOD fail_int (name, fail : String, value : Integer) Trace_Server

  METHOD fail_real (name, fail : String, value : Real) Trace_Server

  ROUTINE new
    (node_set   : Node_Set,
     silent     : Boolean) Trace_Server :

END Trace_Server
```

Veröffentlichungen des Konrad-Zuse-Zentrum für Informationstechnik Berlin
Technical Reports Oktober 1990

TR 86-1. H. J. Schuster. *Tätigkeitsbericht (vergriffen)*

TR 87-1. H. Busch; U. Pöhle; W. Stech. *CRAY-Handbuch. - Einführung in die Benutzung der CRAY.*
TR 87-2. H. Melenk; W. Neun. *Portable Standard LISP Implementation for CRAY X–MP Computers. Release of PSL 3.4 for COS.*
TR 87-3. H. Melenk; W. Neun. *Portable Common LISP Subset Implementation for CRAY X–MP Computers.*
TR 87-4. H. Melenk; W. Neun. *REDUCE Installation Guide for CRAY 1 / X-MP Systems Running COS Version 3.3*
TR 87-5. H. Melenk; W. Neun. *REDUCE Users Guide for the CRAY 1 / X-MP Series Running COS. Version 3.3*
TR 87-6. R. Buhtz; J. Langendorf; O. Paetsch; D. A. Buhtz. *ZUGRIFF - Eine vereinheitlichte Datenspezifikation für graphische Darstellungen und ihre graphische Aufbereitung.*
TR 87-7. J. Langendorf; O. Paetsch. *GRAZIL (Graphical ZIB Language).*

TR 88-1. R. Buhtz; D. A. Buhtz. *TDLG 3.1 - Ein interaktives Programm zur Darstellung dreidimensionaler Modelle auf Rastergraphikgeräten.*
TR 88-2. H. Melenk; W. Neun. *REDUCE User's Guide for the CRAY 1 / CRAY X-MP Series Running UNICOS. Version 3.3.*
TR 88-3. H. Melenk; W. Neun. *REDUCE Installation Guide for CRAY 1 / CRAY X-MP Systems Running UNICOS. Version 3.3.*
TR 88-4. D. A. Buhtz; J. Langendorf; O. Paetsch. *GRAZIL-3D. Ein graphisches Anwendungsprogramm zur Darstellung von Kurven- und Funktionsverläufen im räumlichen Koordinatensystem.*
TR 88-5. G. Maierhöfer; G. Skorobohatyj. *Parallel-TRAPEX. Ein paralleler, adaptiver Algorithmus zur numerischen Integration ; seine Implementierung für SUPRENUM-artige Architekturen mit SUSI.*

TR 89-1. *CRAY-HANDBUCH. Einführung in die Benutzung der CRAY X-MP unter UNICOS.*
TR 89-2. P. Deuflhard. *Numerik von Anfangswertmethoden für gewöhnliche Differentialgleichungen.*
TR 89-3. A. R. Walter. *Ein Finite-Element-Verfahren zur numerischen Lösung von Erhaltungsgleichungen.*
TR 89-4. R. Roitzsch. *KASKADE User's Manual.*
TR 89-5. R. Roitzsch. *KASKADE Programmer's Manual.*
TR 89-6. H. Melenk; W. Neun. *Implementation of Portable Standard LISP for the SPARC Processor.*
TR 89-7. F. A. Bornemann. *Adaptive multilevel discretization in time and space for parabolic partial differential equations.*
TR 89-8. G. Maierhöfer; G. Skorobohatyj. *Implementierung des parallelen TRAPEX auf Transputern.*

TR 90-1. K. Gatermann. *Gruppentheoretische Konstruktion von symmetrischen Kubaturformeln.*
TR 90-2. G. Maierhöfer; G. Skorobohatyj. *Implementierung von parallelen Versionen der Gleichungslöser EULEX und EULSIM auf Transputern.*
TR 90-3. *CRAY-Handbuch. Einführung in die Benutzung der CRAY X-MP unter UNICOS 5.1*
TR 90-4. H.-C. Hege. *Datenabhängigkeitsanalyse und Programmtransformationen auf CRAY-Rechnern mit dem Fortran-Präprozessor* ***fpp.***
TR 90-5. M. Grammel; G. Maierhöfer; G. Skorobohatij. *Trapex in POOL; Implementierung eines numerischen Algorithmus in einer parallelen objektorientierten Sprache.*
TR 90-6. P. Deuflhard; A. Hohmann. *Einführung in die Numerische Mathematik.*