Koordinierung und verteilte Systeme

F. Hofmann, Universität Erlangen

Zusammenfassung:

Zur Erzielung übersichtlicher Spezifikationen und Implementierungen werden neben den bekannten Funktions- und Datenabstraktionen noch Asynchronitäts- und Prozeßabstraktion untersucht und in ein allgemeines Strukturierungskonzept eingebettet. Anschließend werden Fragen der Realisierung diskutiert, wobei sich der 'remote-procedure-call' als besonders bedeutsam herausstellt. Auf dieser Basis werden als wichtige einschlägige, programmiersprachliche Überlegungen die 'communicating sequential processes' von Hoare und ihre Einbettung in ADA und eine realisierte PEARL-Erweiterung diskutiert.

Stichworte: Koordinierung asynchrone Prozesse, Programmiersprachen für verteilte Systeme.

Contents:

In order to obtain lucid specifications and implementations abstractions of asynchrony and processabstractions are investigated in addition to functional abstraction and data abstraction. These ideas are combined into a concept for structuring programs for distributed systems. Remote procedure call turns out to be a fundamental concept for realizing such programs. On this basis the ideas developped by Hoare in his paper on 'communicating sequential processes' are discussed along with their embeddings into ADA and into an implemented extension of PEARL.

Keywords: cooperating, asynchronous processes; programming languages for distributed systems.

1. Allgemeine Vorbemerkung

Die augenfälligste Besonderheit verteilter Systeme besteht darin, daß mehrere Befehlsströme gleichzeitig abgearbeitet werden, wobei nur eine geringe und unscharfe, zeitliche Kopplung zwischen der Abarbeitung verschiedener Befehlsströme existiert. Bei einem Monoprozessor mag zwar das Betriebssystem an der Benutzerschnittstelle ebenfalls diesen Sachverhalt suggerieren, tatsächlich wird aber zu jedem Zeitpunkt nur ein Befehl interpretiert.
Zu der Zeit, als die heute am weitesten verbreiteten Programmiersprachen entwikkelt wurden, spielten fast ausschließlich Monoprozessoren eine Rolle. Von daher ist es auch verständlich, daß sie keine oder nur sehr schwach ausgeprägte Mittel besitzen, um Programme, die mehrere, geignet kooperierende Befehlsströme zur Lösung einer Aufgabe benutzen, formulieren zu können. Ein weiterer Aspekt betrifft die Tatsache, daß in einem verteilten System nicht jeder Prozeß direkten Zugang zu jedem Speicherplatz hat und damit die Frage, wo Daten gelagert werden sollen und wo bestimmte Prozesse oder Teilstücke von Prozessen ausgeführt werden sollen, erheblichen Einfluß auf das Preis-/Leistungsverhältnis eines Programmsystems hat. Diese Problematik tritt beim Monoprozessor nicht auf und spielte deshalb auch bei der Entwicklung der heute gängigen Programmiersprachen - wenn überhaupt - nur eine untergeordnete Rolle. Die Folge ist, daß selbst dann, wenn die Sprache den Begriff des

Prozesses (meist als task bezeichnet) kennt, die Möglichkeiten zur Koordinierung der Aktivitäten verschiedener Prozesse recht schwach ausgelegt sind. Der Anwender kann nämlich in diesem Fall, sobald ein gegenseitiger Ausschluß bei der Bearbeitung von gemeinsamer Statusinformation erreichbar ist, die Möglichkeit einer Abstimmung mit Hilfe von allen Prozessen zugänglichen Informationen benutzen, um auch sehr komplizierter Formen der Zusammenarbeit zu erzielen. Hierzu ausreichende, einfach zu verstehende und zu realisierende Hilfsmittel wurden bereits 1968 von Dijkstra in Form der Semaphore und der auf sie anwendbaren P- und V-Operationen (identisch mit request und release in PEARL) entwickelt ([01]). Seine Vermutung, daß alle Koordinierungen mit Hilfe der P- und V-Operationen machbar sind, stützt sich wesentlich auf die oben skizzierten Eigenheiten von Monoprozessoren. Erst 1973 wurde von Lipton nachgewiesen, daß diese Anschauung bei Verwendung mehrerer Prozessoren nicht zutrifft ([02] und wesentlich erweitert in [03]). Damit war auch neu zu überdenken, welche Koordinierungsmöglichkeiten Programmiersprachen und Betriebssysteme für verteilte Systeme zur Verfügung stellen müssen.

2. Klassifikation von Koordinierungsaufgaben

Die Entwicklung oder Erweiterung einer höheren Programmiersprache setzt stets Vorstellungen über eine begriffliche Strukturierung der zu beschreibenden Algorithmen voraus. Es soll deshalb nachfolgend der Versuch unternommen werden, für verteilte Systeme geeignete Strukturierungsvorstellungen möglichst unbeeinflußt von programmiersprachlichen Zwängen zu entwickeln, wobei hier die Koordinierung asynchroner Aktivitäten im Vordergrund steht. Für die weiteren Überlegungen bedeutsam ist die Beobachtung, daß es im wesentlichen drei unterschiedliche Gründe für Koordinierungsmaßnahmen gibt ([04]):

1. Gegenseitiger Ausschluß
Manche Manipulationen an Datenstrukturen dürfen nicht zeitlich überlappend mit anderen ausgeführt werden. Dies kann erforderlich sein, um eine Datenstruktur in einem konsistenten Zustand zu halten oder Veränderungen während einer langwierigen Auswertung zu verhindern. Ein typisches Beispiel sind verkettete Listen, wo das Ein- oder Ausketten zu Schwierigkeiten führt, wenn gleichzeitig eines der Nachbarelemente lesend oder schreibend bearbeitet wird. Dieser Fragenkreis spielt bereits bei der Implementierung von Betriebssystemen für Monoprozessoren eine erhebliche Rolle und hat in diesem Zusammenhang zur Entwicklung des Monitorkonzeptes geführt ([05]). Ein Monitor wird dabei angesehen, als eine Datenstruktur und eine Reihe darauf anwendbarer Operationen, die generell unter gegenseitigem Ausschluß abgewickelt werden.
Der gegenseitige Ausschluß ist hier nicht nur im üblichen, von Dijkstra geprägten Sinne zu verstehen, sondern auch im Sinne atomarer Aktionen, wie sie vor allem im Zusammenhang mit verteilten Datenbanken diskutiert werden. Dabei wird im wesentlichen nur verlangt, daß das Ergebnis mehrerer, zeitlich überlappend ausgeführter Operationen auch bei serieller Ausführung erreichbar gewesen wäre ([06]).

2. Kommunikation
Hier geht es darum, daß Prozesse zusammenarbeiten, um ein Problem gemeinsam zu lösen und zu diesem Zweck Information austauschen müssen. Im wesentlichen ist dies auf zwei Arten möglich:
- durch Informationsaustausch über gemeinsame Objekte und
- durch Nachrichtenkopplung.

In der Vergangenheit war die Benutzung gemeinsamer Objekte vorherrschend. Sie bildet z. B. den zweiten wesentlichen Gesichtspunkt des Monitorkonzepts von Concurrent Pascal ([07]).
Für verteilte Systeme ist zu einer Kommunikation zwischen Prozessen auf verschiedenen Rechnern nur der zweite Weg gangbar. Er fand jedoch bei Monoprozessoren nur geringe Verbreitung, da er in vielen Fällen dynamisch aufwendiger ist als die Benutzung gemeinsamer Objekte.

3. Reihenfolgen (Synchronisation)
Wenn mehrere Operationen zur Ausführung

anstehen, die unter gegenseitigem Ausschluß ablaufen, müssen sie letztlich in irgend einer Reihenfolge abgearbeitet werden. Die Festlegung dieser Reihenfolge wird meist dem Betriebssystem überlassen, was erheblichen Zusatzaufwand bedeutet, wenn seine Strategien für die spezielle Anwendung unzweckmäßig sind. Als Antwort auf diese Problematik stellen Betriebssysteme gelegentlich 'dynamische' Prioritäten zur Verfügung, die aber kaum Eingang in Programmiersprachen gefunden haben. So kennt z. B. auch PEARL nur Prioritäten, die beim Start eines Prozesses festgelegt werden und die er unabhängig von dem gerade zu bearbeitenden Datenobjekt beibehält.
Oft sind bestimmte Reihenfolgen einzuhalten, um das korrekte Zusammenwirken zu erreichen, man denke etwa an die Erzeuger-Verbraucher Situation. Bei Echtzeitsystemen kann es wesentlich sein, daß Reihenfolgen wie first-in-fist-out eingehalten werden.
Zur Lösung von Reihenfolgeproblemen wurden zahlreiche Mechanismen entwickelt, die sich für die Bearbeitung derartiger Aufgabenstellungen mehr oder weniger gut eignen. Beispiele sind Signale, Ereignisvariable, Prioritätswarteschlangen und Pfadausdrücke.

Nach den derzeitigen theoretischen Überlegungen besitzen diese Klassen jeweils so spezifische Eigenschaften, daß ihnen zweckmäßigerweise gesonderte sprachliche Ausdrucksmittel gewidmet werden. Dies gilt insbesondere, wenn noch Fragen des Korrektheitsnachweises eines Programmsystems berücksichtigt werden.

3. Ein Strukturierungskonzept

Um zu möglichst universellen, für verteilte Systeme brauchbaren Strukturierungskonzepten zu gelangen, kann man gedanklich vom fertigen Programm ausgehen und betrachten, welche Möglichkeiten bestehen, von solchen Implementierungsdetails zu abstrahieren, die keinen grundsätzlichen Einfluß auf die Brauchbarkeit der Lösungsbeschreibung ausüben.
In der Literatur werden seit langem zwei Abstraktionsmechanismen untersucht ([08]):

1. Funktionale Abstraktion
Hier werden einzelne Algorithmen in separate Einheiten, die Prozeduren, abgetrennt. Die Anwendung eines Algorithmus erfolgt durch Aufruf der zugehörigen Prozedur. Der Aufrufer abstrahiert von der Arbeitsweise des Algorithmus und benötigt nur noch die Kenntnis seiner Wirkung. Die Lösungsbeschreibung braucht bei Verwendung dieser Abstraktionsform lediglich festzulegen, welche Änderungen durch eine Prozedur an den Daten hervorgerufen werden, ohne näher zu präzisieren, wie dies erreicht wird.

2. Datenabstraktion
Häufig werden eine Reihe von Datenanteilen nur mit bestimmten Operationen bearbeitet. Schon aus Gründen besserer Wartbarkeit erscheint es zweckmäßig, in einem solchen Fall die betreffenden Datenanteile und die darauf anwendbaren Operationen zu einer Einheit zusammenzufassen und nach außen lediglich die Operationen verfügbar zu machen. Als Folge davon benötigt der Benutzer einer solchen Einheit keine Kenntnis über die tatsächliche Darstellung der entsprechenden Datenanteile. Daten zusammen mit den sie manipulierenden Operationen werden als Datenobjekte bezeichnet, ihre Beschreibung als abstrakter Datentyp.

Abstrakte Datentypen stellen lediglich einem Benutzer durch ihn aktivierbare Möglichkeiten zur Verfügung, beschreiben aber keine Gebilde, die von sich aus aktiv werden. Solange einzelne sequentielle Programme betrachtet werden, erscheint dies auch ausreichend. Programmsysteme zur Überwachung und Steuerung von Produktionsanlagen bestehen dagegen aus mehreren in sich sequentiellen und deterministischen Abläufen, die zeitlich durchmischt abgearbeitet werden. Um sich bei derartigen Programmsystemen von Implementierungseinzelheiten zu befreien, kann man sich zweier weiterer Abstraktionsmechanismen bedienen ([04]):

3. Asynchronitätsabstraktion
Wird ein abstrakter Datentyp von mehreren Benutzern gleichzeitig oder zeitlich verschachtelt ausgeführt, so sind zu einer konfliktfreien Bearbeitung häufig Koordinierungsmaßnahmen erforderlich. Zur Lösung solcher Koordinierungsprobleme wurde eine

große Zahl von Mechanismen entworfen und realisiert, die fast alle die Tendenz haben, die Koordinierungsmaßnahmen über die Programme zu verstreuen und damit Koordinierungseigenschaften schwer durchschaubar zu machen. Betrachtet man jedoch Programme, die sich der Idee abstrakter Datentypen bedienen, so stellt man fest, daß Koordinierungsmaßnahmen meist an der Schnittstelle zwischen Datentypen auftreten. Diese Überlegungen lassen es empfehlenswert erscheinen, die Koordinierung der Operationen eines Datentyps mit in die Struktureinheit abstrakter Datentyp einzubeziehen. Werden Operationen eines abstrakten Datentyps aufgerufen, so braucht der Aufrufer nun keine eigenen Vorkehrungen für die Koordinierung zu treffen, d. h. er kann von der Notwendigkeit bestimmter Koordinierungsmaßnahmen abstrahieren.
Es spricht vieles dafür, daß abstrakte, synchronisierte Datentypen die grundlegende Strukturierungseinheit für Systeme sind, in denen mehrere kooperierende Programme gleichzeitig ablaufen.

4. Prozeßabstraktion
Abstrakte, synchronisierte Datentypen als zentrale Zerlegungseinheiten in Prozeßsystemen, so wie sie bisher entwickelt wurden, sind in einer Reihe von Fällen in ihrer Ausdruckskraft noch unbefriedigend. Betrachtet man etwa eine Komponente 'Systemuhr', so könnte sie durch einen abstrakten Datentyp mit den Operationen clockup, clockdown und readclock charakterisiert werden. Dann wäre noch zum Ausdruck zu bringen, daß die Operationen clockup und clockdown unabhängig von den Aktivitäten anderer Programme alternierend ablaufen. Geht man den Weg, daß Aktivitätsträger, die Operationen von Datenobjekten in sequentieller Reihenfolge aufrufen und zur Ausführung bringen, als Prozesse bezeichnet, so gehört offenbar zur Beschreibung der Systemuhr ein Prozeß der zyklisch die Operationen clockup und clockdown aufruft. Man sieht sofort, daß konzeptionell ein solcher Prozeß zur Idee der Systemuhr gehört und deshalb auch Bestandteil des beschreibenden abstrakten, synchronisierten Datentyps sein sollte.
Es ist Aufgabe des Datentyps selbst, Aktivitätsträger, die zur Lösung seiner Aufgabe erforderlich sind, bereitzustellen. Übergeordnete Prozesse, die einen solchen Datentyp benutzen, können dann von datentypinternen, prozeduralen Vorgängen abstrahieren.

Neben diesen vier Abstraktionsformen sind noch eine Reihe weiterer bedeutsam, wie etwa bezüglich Recoveryverhalten und Zugriffsschutz. Da sie von weniger fundamentaler Bedeutung sind, werden sie hier nicht weiter verfolgt.

Gebilde, die bei Berücksichtigung der vier dargestellten Abstraktionsmechanismen als Einheit zu sehen sind, werden im weiteren als Module bezeichnet. Zu ihrer Beschreibung gehört:

1. eine Datenstruktur, die den Wertevorrat charakterisiert und
2. eine Menge von Operationen, die die Datenstruktur manipulieren können.

Die Operationen ihrerseits zerfallen in:

1. Prozeduren, die von Prozessen aufgerufen werden können, und
2. Prozesse, als Träger von Aktivitäten.

Für Strukturüberlegungen ist es dabei irrelevant, wie die Auswirkung der Operationenoder Teiloperationen von Prozeduren und Prozessen beschrieben werden. Im wesentlichen gibt es zwei Möglichkeiten:
1. Beschreibung durch Angabe von Algorithmen
2. durch Ausdrücke einer (mehr oder weniger formalen) Theorie, die dem Anwendungsgebiet angemessen ist.

Im ersten Fall spricht man im allgemeinen von Implementierungen, im zweiten von Spezifikationen.
Der Asynchronitätsabstraktion wird dadurch Rechnung getragen, daß (Teil-)Operationen, soweit erforderlich, noch eine Nichtblockierungsbedingung beigegeben wird. Sie beschreibt, bei welchen Objektzuständen eine auszuführende (Teil-)Operation tatsächlich durchgeführt werden kann. Ist die Nichtblockierungsbedingung nicht erfüllt, so wird die Ausführung bis zu ihrem Erfülltsein (verursacht durch andere Prozesse) aufgeschoben.

Die Prozeßabstraktion findet ihren Niederschlag in zwei Strukturkonzepten:

1. In einem Modul können aktive Komponenten existieren. Sie werden nicht aufgerufen, sondern stehen, wenn sie nicht gerade aktiv sind, immer zur Ausführung an.
2. Operationen können als Sequenzen von Teiloperationen dargestellt werden, wobei jede Teiloperation durch eine Nichtblockierungsbedingung und eine Effektbeschreibung charakterisiert ist.

Grundsätzlich reichen zwar die soweit entworfenen Konzepte zur Behandlung aller drei Klassen von Koordinierungsaufgaben aus, ihre alleinige Benutzung würde aber zu schwer durchschaubaren Darstellungen führen. Sowohl der gegenseitige Ausschluß als auch Reihenfolgeangaben haben im Grunde einen relationalen Charakter ([08]). Es erscheint daher sinnvoll, Module noch durch eine 'incompatible'-Relation und eine 'prior'-Relation zu ergänzen. Die 'incompatible'-Relation beschreibt, welche (Teil-)Operationen eines Moduls (eventuell noch abhängig von den Aufrufparametern) nur unter gegenseitigem Ausschluß bearbeitet werden dürfen. Die 'prior'-Relation zeigt an, daß die Ausführung gewisser (Teil-)Operationen (eventuell noch abhängig von den Aufrufparametern) nicht begonnen werden darf, solange noch bestimmte andere auf Ausführung warten. Prioritätsangaben betreffen in diesem Konzept nicht Prozesse sondern (Teil-)Operationen und unterscheiden sich dadurch wesentlich von der üblichen Verwendung.

Insgesamt ergibt sich somit die in Bild 1 skizzierte Modulstruktur. Zur Veranschaulichung der Verwendung obiger Relationen ist in Bild 2 als einfaches Beispiel die Koordinierung des bekannten zweiten Leser-Schreiberproblems dargestellt, bei dem Schreibaufrufe möglichst stark bevorzugt werden sollen, auch auf die Gefahr hin, daß dadurch Leseaufrufe beliebig lange verzögert werden können.

Es sei noch angemerkt, daß beide Relationen für die praktische Anwendung etwas differenziertere Ausdrucksmöglichkeiten benötigen ([04]), die hier nicht näher

```
MODULE        Modulname(Parameterliste);

DECLARATIONS  Beschreibung der möglichen Objektwerte,
              die auch als Objektzustände bezeichnet
              werden;

SYNCHRONIZATION
  x incompatible y: Beschreibung der Unverträglich-
                    keitsrelation;
  x prior y:        Beschreibung der Prioritätsrela-
                    tion;

OPERATIONS
  Operationsbezeichnung(Parameterliste)
            → Ergebnisvariable;
    NBL     Nichtblockierungsbedingung; } erste Teil-
    EFFECTS Effektbeschreibung;         } operation
      .
      .
      .
    NBL     Nichtblockierungsbedingung; } letzte Teil-
    EFFECTS Effektbeschreibung;         } operation
  .
  .
  .
  Prozeßbezeichnung;
    CYCLIC
    NBL     Nichtblockierungsbedingung; } erste Teil-
    EFFECTS Effektbeschreibung;         } operation
      .
      .
      .
    NBL     Nichtblockierungsbedingung; } letzte Teil-
    EFFECTS Effektbeschreibung;         } operation
END_MODULE
```

Bild 1. Struktureller Aufbau eines Moduls

```
MODULE        reader_writer_database;

DECLARATIONS ...
              /* Beschreibung der möglichen Zustände
              der Datenbasis */;

SYNCHRONIZATION
  x incompatible y: x ∈ WRITE ∧ y ∈ READ ∪ WRITE ;
              /* Mit großen Buchstaben geschriebene
              Operationsbezeichnungen bezeichnen die
              Menge aller noch nicht beendeten Aufrufe
              der entsprechenden Operation.
              Im vorliegenden Fall wird also zum Aus-
              druck gebracht, daß Schreiboperationen
              nur unter gegenseitigem Ausschluß zu Lese-
              oder anderen Schreiboperationen bearbei-
              tet werden dürfen. */
  x prior y:          (x ∈ WRITE ∧ y ∈ READ) ∨
                      (x, y ∈ WRITE ∧ x before y) ;
              /* Die erste Zeile bringt zum Ausdruck,
              daß mit der Bearbeitung von Leseaufrufen
              nur dann begonnen werden darf, wenn keine
              Schreibaufrufe zur Ausführung anstehen.
              Die zweite Zeile besagt, daß die Schreib-
              aufrufe in der Reihenfolge des Aufrufs
              zur Ausführung zu bringen sind.
              Die Reihung von Leseaufrufen ist offenge-
              lassen, so daß Compiler und/oder Betriebs-
              system hier beliebige Strategien wählen
              können. */

OPERATIONS
  read(...) → ...;

    NBL     TRUE;
    EFFECTS ...;
              /* Effektbeschreibung der Leseoperation
              */

  write(...);

    NBL     TRUE;
    EFFECTS ...;
              /* Effektbeschreibung der Schreibopera-
              tion */

END_MODULE
```

Bild 2. Darstellung der Datenbasis für das zweite Leser-Schreiber-Problem

erörtert werden, da sie keine Rückwirkung auf die bisherigen Überlegungen haben.

3. Implementationsgesichtspunkte

Will man asynchrone Prozeßsysteme implementieren, so wird man zuerst nach einer geeigneten Programmiersprache Ausschau halten. Bei Verwendung von Monoprozessoren kann man dabei verhältnismäßig leicht zu brauchbaren Lösungen kommen, da die Gesamtheit der Module in diesem Falle (zumindest für die Implementierung) ohne schädliche Auswirkungen auf das dynamische Verhalten zu einem einzigen Modul zusammengefaßt werden kann. Wenn nämlich das unterlagerte Betriebssystem erlaubt

- für getrennt übersetzte und geladene Programme gemeinsame Datenbereiche einzurichten,
- zur Erzielung des gegenseitigen Ausschlusses Prozeßumschaltungen nur als Folge von expliziten Betriebssystemaufrufen zu erlauben (eine Zuordnungsstrategie, die Betriebssysteme für Prozeßrechner im allgemeinen zur Verfügung stellen) und
- dynamische Prioritäten (für Reihenfolgefestlegungen) zur Verfügung stellt,

kann man im Prinzip jede Programmiersprache verwenden, die Datenbereiche benennen kann (z. B. FORTRAN mit labelled common) und die die Möglichkeit besitzt, Betriebssystemaufrufe in Form von 'Code-Prozeduren' zugänglich zu machen.

Ganz anders liegen die Verhältnisse bei Betrachtung verteilter Systeme. Hier ist die Modularisierung vom technischen Aufbau des Rechensystems her erzwungen, so daß lediglich zu überlegen bleibt, welche Mechanismen eine Programmiersprache zur Verfügung stellen muß, um ein System kooperierender Module realisieren zu können. Aus Gründen eines effizienten Datenzugriffs wird im weiteren vorausgesetzt, daß ein Modul stets vollständig auf einem Rechner residiert. Offensichtlich treten zwei sehr unterschiedliche Klassen von Koordinierungen auf:

1. Die Intra-Modul-Koordinierung, die dieselben Fragen aufwirft, wie die Realisierung eines Systems asynchroner Prozesse auf einem Monoprozessor, und
2. die Inter-Modul-Koordinierung, die bei dem vorgestellten Strukturkonzept mit dem in der neueren Literatur gebräuchlichen Begriff 'remote-procedure-call' (RPC) identisch ist.

Da der Teilpunkt 1 unter Benutzung üblicher Prozeßrechner-Betriebssysteme erledigt werden kann, bleibt lediglich Teilpunkt 2 genauer zu untersuchen. Es gibt eine Reihe von Arbeiten und Pilotimplementierungen, die zeigen, daß ein für den Benutzer transparenter RPC-Mechanismus durchaus im Bereich Machbaren liegt, so daß auch von hier aus keine besonderen Anforderungen an eine Programmiersprache (sehr wohl aber an ihren Compiler und das Betriebssystem!) gestellt werden. Da man jedoch davon ausgehen muß, daß industriell eingesetzte Betriebssysteme in nächster Zukunft keinen RPC-Mechanismus zur Verfügung stellen und nach dem derzeitigen Kenntnisstand ein universeller RPC-Mechanismus für Echtzeitanwendungen einen zu hohen statischen und dynamischen Aufwand verursacht, wird es zweckmäßig sein, in den Programmiersprachen lediglich einfachere Mechanismen zur Verfügung zu stellen, die aber eine gute Basis für die Realisierung von 'remote-procedure-calls' bilden müssen.

Implementiert man die Prozeduren eines Moduls als zyklische Prozesse, so werden im wesentlichen zwei Mechanismen benötigt:

1. eine Interprozeßkommunikation (IPC) zum Austausch von mit einem Typ versehenen Nachrichten zwischen Prozessen und
2. ein Mechanismus, der es Prozessen erlaubt, auf Nachrichten verschiedenen Typs gleichzeitig zu warten und die Reaktion vom Typ einer empfangenen Nachricht abhängig zu machen.

Wie in [04] nachgewiesen wurde, kann für große Klassen von Nichtblockierungsbedingungen, 'incompatible'- und 'prior'-Relationen automatisch eine Realisierung gewonnen werden, die sich lediglich obiger Mechanismen bedient. Die Aktivitäten eines Moduls werden dabei am Anfang und Ende durch teilweise recht komplizierte Programmstücke ergänzt. Eine Rückführung auf noch einfachere Koordinierungsmechanismen ist zwar möglich, bleibt aber nur dann

halbwegs durchschaubar, wenn Koordinierungsmuster einer IPC eingehalten werden. Dies sicherzustellen ist aber gerade eine wesentliche Leistung, die von einer guten Programmiersprache zusammen mit ihrem Compiler erwartet wird.

4. Beispiele für Sprachkonstrukte

Als erstes Beispiel werden die 'communicating sequential processes' von Hoare ([09]), die auch die Basis für sehr ähnliche Konstrukte in ADA bilden, aus der hier entwickelten Sicht beleuchtet.
Neben herkömmlichen Sprachkonstrukten enthält CSP drei weitere, die im Hinblick auf verteilte Systeme konzipiert sind:

1. Es gibt eine Parallelanweisung, die es gestattet nebenläufige Prozesse zu formulieren.
2. Zur Kommunikation zwischen Prozessen gibt es eine Ausgabe-Anweisung P!x, die die Nachricht x mit dem durch x bestimmten Typ an Prozeß P sendet und eine Eingabe-Anweisung P?y, die eine Nachricht des durch y bestimmten Typs von Prozeß P entgegennimmt und den Inhalt in y hinterlegt. Die Synchronisation erfolgt nach dem 'rendez-vous'-Prinzip, d. h. der ausgebende (empfangende) Prozeß wird bis zur tatsächlichen Ablieferung (Ankunft) einer Nachricht blokkiert.
   Bis auf die Forderung, daß der Empfänger angeben muß, von welchem Sender er eine Nachricht erwartet, handelt es sich um einen Mechanismus der eine unmittelbare RPC-Implementierung mit Hilfe zyklischer Prozesse gestattet, wenn der Prozedurrumpf lediglich einen Aufrufparameter in einem lokalen Datenbereich hinterlegt (also in einem sehr eingeschränkten Fall).
3. Disjunktives Warten mit selektiver Fortsetzung wird durch die 'guarded commands' von Dijkstra erreicht. Sie haben die Form
   $G_1 \rightarrow S_1 \square G_2 \rightarrow S_2 \square \ldots \square G_n \rightarrow S_n$.
   Die G_i sind Nichtblockierungsbedingungen für die S_i. Sind mehrere G_i erfüllt, so wird eine beliebige der zugehörigen Anweisungen ausgeführt. Der 'guarded command' gilt dann als abgearbeitet. Hoare läßt in den 'guards' auch Eingabe-Anweisungen zu, was letztlich eine ungepufferte Interprozeßkommunikation mit disjunktivem Warten bedeutet.

Die Weiterentwicklung von CSP in ADA geht vor allem in Richtung RPC. Der Nachrichtenempfänger muß nicht mehr den Absender benennen und kann explizit angeben, wann das 'rendez-vous' beendet ist, womit leicht ein allgemeiner RPC realisierbar ist.
Für die Intermodulkoordinierung stehen in beiden Fällen keine weiteren Möglichkeiten zur Verfügung, was nach den Überlegungen des Abschnitts 2 im Hinblick auf den ohnehin sehr schwierigen Korrektheitsnachweis verteilter Systeme als unbefriedigend anzusehen ist.

Ein Beispiel für die Erweiterung einer bestehenden Programmiersprache stellt die in Erlangen realisierte PEARL-Erweiterung dar ([10]), die folgende Strukturierungsmittel bietet:

1. Die Zusammenfassung von Daten, Prozeduren und Prozessen zu Moduln ist bereits Bestandteil von PEARL.
2. Zur Kommunikation zwischen Prozessen wurden die Anweisungen transmit und receive zum senden bzw. empfangen von Nachrichten in die Sprache aufgenommen. Sie stellen insofern eine Erweiterung der Aus- und Eingabeanweisungen von CSP dar, als sowohl die synchrone Arbeitsweise des 'rendez-vous'-Konzeptes als auch eine asynchrone Arbeitsweise (durch Nachrichtenpufferung) formuliert werden kann.
3. Disjunktives Warten mit selektiver Fortsetzung wird ebenfalls durch 'guarded commands' ermöglicht.

Diese PEARL-Erweiterung kann also als Verallgemeinerung von CSP angesehen werden. Darüberhinaus zeigt sie, daß die für die praktische Anwendung wichtige Frage der Zeitüberwachung bei der Kommunikation sich ohne Schwierigkeiten in die hier vorgestellten Strukturüberlegungen einpassen läßt.
Die an CSP angebrachte Kritik trifft allerdings auch hier zu, da die in PEARL vorhandenen Mittel zur Intramodulkoordinierung

zu elementar gehalten sind und die Erweiterung keine speziellen Sprachmittel für Verträglichkeits- und Reihenfolgefestlegungen enthält.

5. Schlußbemerkung

Die besonderen Anforderungen, die die Koordinierung für verteilte Systeme an höhere Programmiersprachen stellt, haben zu einer Reihe interessanter Entwicklungen geführt. Allerdings liegen derzeitige Sprachkonstrukte im Vergleich zum Stand bei der Datenstrukturierung und der Beschreibung sequentieller Kontrollstrukturen auf einer verhältnismäßig niederen Abstraktionsebene. Dies gilt insbesondere bezüglich solcher Koordinierungen, die von relationalem Charakter sind. In dieser Hinsicht ist sicherlich noch erhebliche Entwicklungsarbeit zu leisten.
Für ausführliche Beispiele zu den hier entwickelten Ideen und für Überlegungen zu ihrer Formalisierung als Voraussetzung für die Verifikation sei auf [11] und [04] verwiesen.

Schrifttum:

[01] Dijkstra, E. W.: Cooperating Sequential Processes. In 'Programming Languages', ed. F. Genuys, Academic Press London - New York, 1968, pp. 43-112.

[02] Lipton, R. J.: On Synchronization Primitive Systems. Dep. of Comp. Science, Carnegie-Mellon, 1973.

[03] Bathelt, P.: Vergleich von Synchronisationsmechanismen. Arbeitsberichte des IMMD, Universität Erlangen-Nürnberg, Band 15, Nr. 3 (1982).

[04] Mackert, L.: Modellierung, Spezifikation und korrekte Realisierung von asynchronen Systemen. Arbeitsberichte des IMMD, Universität Erlangen-Nürnberg, Band 16, Nr. 7 (1983).

[05] Hoare, C. A. R.: Monitors: An Operating System Structuring Concept. CACM vol. 17, no. 10, (1974), pp. 271-281.

[06] Allchin, J. E.; McKendry, M. S.: Synchronization and Recovery of Actions. Second ACM Symp. on Principles of Distributed Computing, Montreal, Quebec, Aug. 17-19, 1983, pp.31-44.

[07] Brinch Hansen, P.: The Architecture of Concurrent Programs. Prentice-Hall, Englewood Cliffs, N. J., 1977.

[08] Keramidis, S.; Mackert, L.: Specification and Implementation of Parallel Activities on Abstract Objects. Proc. 4th Int. Conf. on Software Engineering, 1979, pp. 203-211.

[09] Hoare, C. A. R.: Communicating Sequential Processes. CACM, vol. 21, no. 8 (1978), pp.666-677.

[10] Fleischmann, A.; Holleczek, P.; Klebes, G.; Kummer, R.: Synchronisation und Kommunikation verteilter Automatisierungsprogramme. Angewandte Informatik 7 (1983), pp. 290-297.

[11] Keramidis, S.: Eine Methode zur Spezifikation und korrekten Implementierung von asynchronen Systemen. Arbeitsberichte des IMMD, Universität Erlangen-Nürnberg, Band 15, Nr. 4 (1982).

Anschrift:

Hofmann, Fridolin
Institut für Mathematische
Maschinen und Datenverarbeitung (IV)
der Universität Erlangen-Nürnberg
Martensstr. 3
8520 Erlangen