Informatik-Fachberichte
GI-NTG Fachtagung, Struktur und Betrieb von Rechensystemen Kiel, März 1980

On the Partitioning of Computing Systems into Communicating Agencies

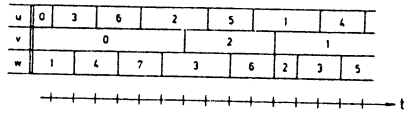
Siegfried Wendt
Department of Electrical Engineering
University of Kaiserslautern
D-6750 Kaiserslautern

1. Introduction

The paper deals with transparent modelling of information processing systems. Instead of considering hardware and software as the two complementary aspects of the system, the paper considers the physical structure and the informational structure. The informational structure is the abstraction which the human mind associates to the physical structure. In this sense, software also has a physical structure aspect, for example as a magnetic pattern on a disk, and hardware also has an informational structure aspect, for example a group of gates which is interpreted as an adder.

2. Processes, Agencies and Channels

A digital process is defined as a set of events where each event is assigned to a point of a discrete time scale. This assignment provides a partial ordering of the events; the ordering is only partial since it is permitted to assign more than one event to the same point of the time scale. An event is a change of value of an observation variable. A structure which describes completely or in some aspects the partial ordering of the events is called a structure of causality. Petri-Nets are used to describe structures of causality /1/. Sets or classes of events are assigned to transitions of the net. Fig. 1 shows an example with three observation variables u, v and w.



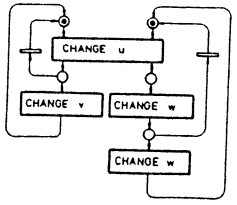
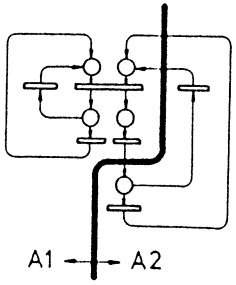


Fig.1

Now, the functional units are considered which produce the events; these units are called agencies /2/. Agencies are obtained by partitioning the set of transitions of the structure of causality; each block of the partition corresponds to an agency. Only such partitions are permitted where transitions with a common place are assigned to the same block, because the decision in a conflict situation must be made by one agency. If two transitions in different blocks of the partition are connected in the Petri-Net, the two corresponding agencies must communicate in order to guarantee the relation between the produced process and the causality structure. The communication requires that the agencies are interconnected via so-called channels. Since the channels are interpreted as containers for information, it is not necessary to restrict the use of channels to the communication between agencies but to allow the connection of a channel to a single agency which then represents a memory of that agency. There is always a channel for each observation variable, but these channels are not necessarily sufficient for the required communication. This is demonstrated in Fig. 2 :

→ formal analysis of petri-nets (a „fork" would not be taken into account!)



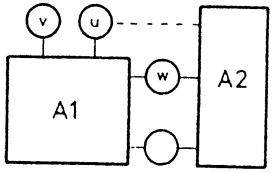


Fig.2

Two agencies A1 and A2 are defined by partitioning the set of transitions from Fig.1. Since all events concerning the observation variables u and v are assigned to the agency A1, the corresponding channels could be memories of A1. Only if the action of A2 depends somehow on the value of u, the agency A2 must have access to read u. Since the structure of causality shows the independence between the changes of v and the changes of w, there is no access from A2 to v. The channel for w, however, must be shared by A1 and A2, since both agencies can change the value of w. There must be yet another channel between A1 and A2 which follows from the structure of causality: A2 can observe the change at w produced by A1, and it interprets this as the start signal for its

own action. Since there are two alternatives for the action of A2, namely change w oder do nothing, A1 cannot decide whether A2 has finished its action or not, just by looking at w.

3. Programmed Agencies and Data Flow Graphs

Now, it is assumed that the function of an agency is defined by a program which must not necessarily be sequential. The program must correspond to the structure of causality. Instead of sets or classes of events, now instructions are assigned to the transitions of the Petri-Net - one instruction at most to one transition. An instruction has the general form S:=f(S) where S is the vector of all observation variables.

Fig. 3 shows the program corresponding to Fig. 1. One transition has been eliminated because the execution of the instruction $v:=(u)_{mod3}$ does not always produce a change of v. One transition has two alternative instruction assigned to it; in the given context, these two instructions are equivalent with respect to their effect on w; they are not equivalent, however, with respect to the resulting data flow graph.

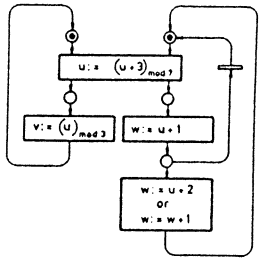


Fig.3

The data flow graph is the representation of the access relation between programmed agencies and the observation variables referred to in the instruction /3/. For the program in Fig. 3 and the partition in Fig. 2, the data flow graphs are shown in Fig. 4; the two versions correspond to the two alternative instructions in the program.

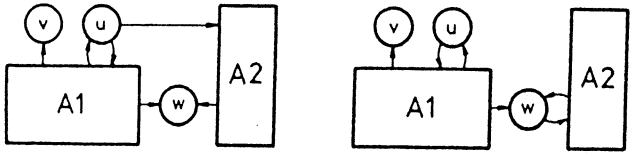


Fig.4

The data flow graph only shows those channels which are given by the observation variables; the communication via these channels is called shared variable communication. The communication via the other channels is called token communication. Since all communication must be finally implemented in form of shared variable communication, the token communication only exists on a certain level of abstraction, where it represents communication for synchronisation purposes. Fig. 5 shows, how token communication is implemented:

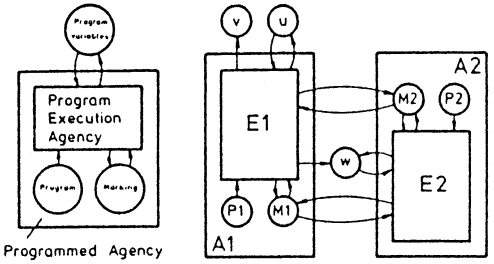


Fig.5

Each programmed agency can be decomposed into its program execution agency and the two memories for the program and its marking. Token communication between two agencies A1 and A2 now means that the program execution agency E1 of A1 has modifying access to the marking M2 of A2 or vice versa. While A1 and A2 are said to have token communication, E1 and E2 only have shared variable communication.

Of course, it is possible to show the token channel between A1 and A2 explicitly by decomposing the memory M2 into the memory access agency and the memory information container m2. Let M2 be a program counter built of flipflops and gates, then m2 is only a location on the output wires where the stored value can be measured. Figure 6 shows this de-

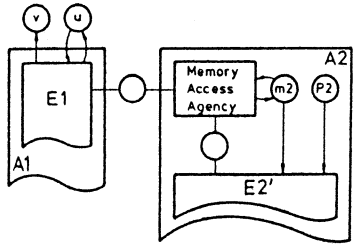


Fig.6

entscheidend

composition. Since the distinction between token communication and shared variable communication, however, is crucial for the transparent modelling of computing systems, the representation in Fig.5 is preferred to that in Fig. 6.

4. Mapping between Informational Structure and Physical Structure

Between an informational structure and a corresponding physical structure there is either the relationship of direct implementation or of indirect implementation. Direct implementation means that the physical structure can be partitioned such that there is a 1:1-mapping between the agencies and channels of the informational structure and the partition blocks of the physical structure (see Fig.7). The basic principle

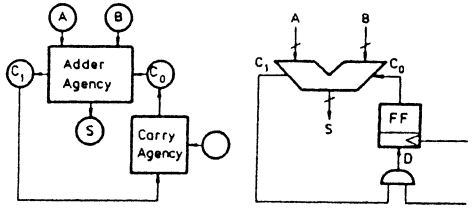


Fig.7

of indirect implementation is multiplex which means that a component of the physical structure in different intervals of time plays the role of different components of the informational structure. The most common case of multiplex is processor multiplex which means that one physical processor in different intervals of time plays the role of the program execution agency of different programmed agencies (see Fig. 5).

Of course, for each physical structure which can be interpreted as an information processing system, there exists an informational structure of which the physical structure is a direct implementation. But the same physical structure can be the indirect implementation of another informational structure. The two informational structures must be equivalent with respect to certain processes which they define. It shall be pointed out that the directly implemented informational structure cannot be obtained by stepwise refinement from the given indirectly implemented informational structure. For example, the interrupt agency which is needed for processor multiplex in a timesharing system cannot be found by stepwise refinement of the structure in Fig.5.

5. Examples The first example is taken from a complex software system: GOLEM from SIEMENS in BS 1000. Fig. 8 shows a system where m users simultaneously want to do interactive retrieval on a common data pool.

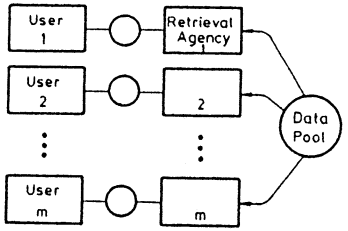


Fig.8

Instead of implementing this system directly, it is more economical to implement only a single retrieval agency and multiplex it. Fig.9 shows the informational structure for this multiplex. This structure, again,

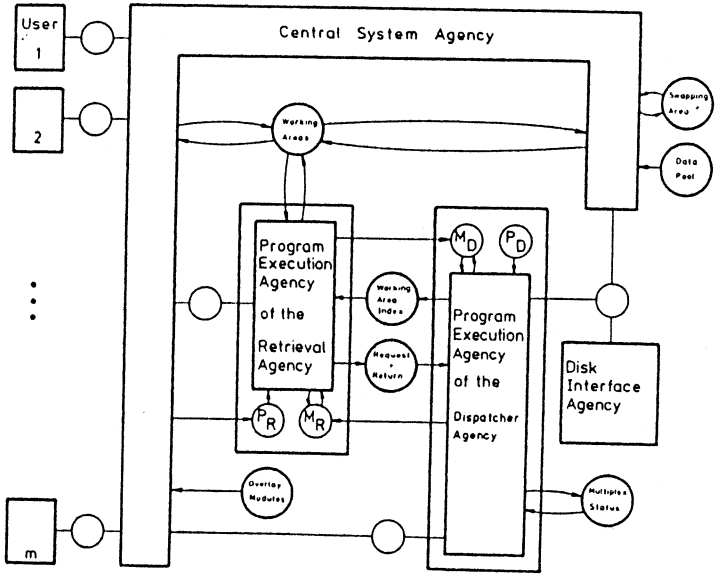


Fig.9

is not directly implemented, since it contains more than one program execution agency which finally will be implemented by multiplex, too. But except of this processor multiplex, the structure in Fig. 9 is already very close to the implementation.

Whenever the retrieval agency is active, it works for one of the users, but it must not know which one; this information is part of the multiplex status. The working area index tells the retrieval agency which of the working areas in the main memory is the one of the actual user. The number of working areas in the main memory is less than the number of users; therefore a swapping area on disk is needed. The program of the retrieval agency is too long for the available program space P_R in the main memory; therefore there must be a path for overlay modules from disk to main memory. All transfers between the main memory and peripheral sources and sinks are done by the central system agency which has a hardware section (channels, devices) and a software section (operating system). The structure of causality for the activities of the central system agency contains concurrency. Requests for activities of the central system agency can be issued in form of supervisor calls by the retrieval agency, the dispatcher agency and the disk interface agency. The retrieval agency cannot only send requests to the central system agency, but also to the dispatcher agency; these are the requests for dialogue with the user or for a disk transfer of data or overlay. The dispatcher agency finally forwards these requests to the central system agency or the disk interface agency, respectively. The token communication between the retrieval agency and the dispatcher agency is such that the sum of tokens in the union of M_R and M_D is constantly one. When the retrieval agency places the token into M_D, the channel 'request + return' contains the information for the dispatcher agency what the request is and where the token should be placed in M_R after the requested activity has been completed. Once the dispatcher agency has obtained the token in M_D, it will initiate the requested activity, update the multiplex status and decide from this status for which user the retrieval agency has to work next. This decision sometimes may cause a swapping of working areas.

It has already been said that the final implementation differs from Fig. 9 with respect to processor multiplex. Fig. 10 shows the structure of causality for the actions of those agencies which share the same program execution agency. There are three reasons why the action of the dispatcher agency or the retrieval agency can end: It can be interrupted (i), or it can call another agency, either the central system agency (n) or the partner in the pair (N).

The second example is the philosophers problem /4/. Fig.11 shows the non-sequential program decomposed in such a way that each philosopher agency has a sequential program and only the fork administration agency has a non-sequential program. It is now assumed that the system shall

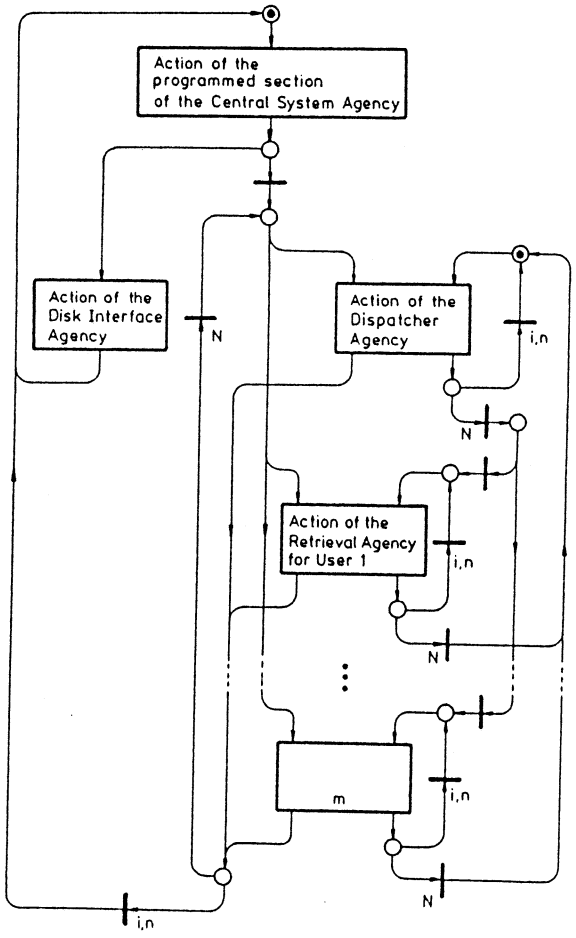


Fig.10

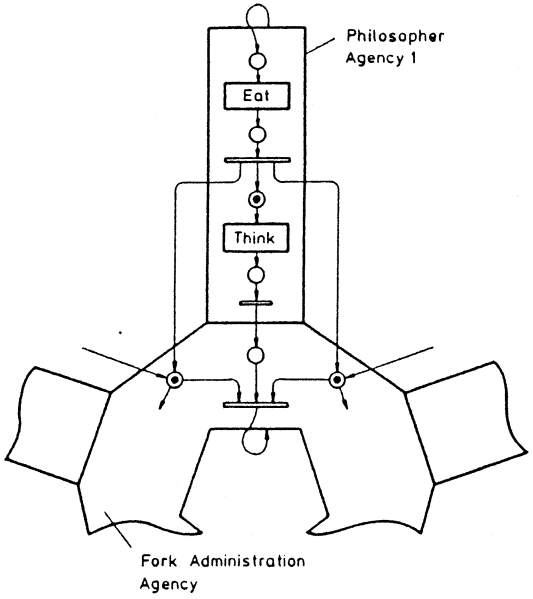


Fig.11

be implemented using a given operating system which supports semaphor communication via the instructions P(s) and V(s). That means that a so-called synchronization agency is given, the non-sequential program of which is shown in Fig.12; this program only deals with transfer of tokens and does not refer to any observation variables. The problem now is to find an implementation of the fork administration agency in form of communicating agencies, where one is the synchronization agency and the others are agencies with sequential programs which can have shared variable communication. Fig. 13 shows the data flow graph of the system which corresponds to the solutions proposed by Dijkstra, Hoare and others /5/. The token communication channels are shown as dashed lines. Usually, the philosopher agency j and the assistant agency j are combined to one agency with a sequential program,called 'the philosopher j'.

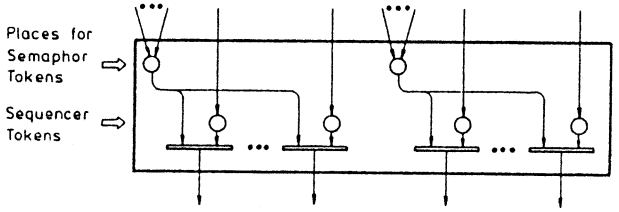


Fig.12

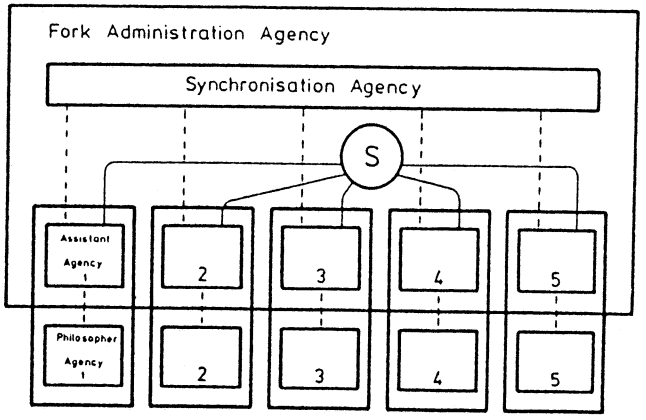


Fig.13

References :

/1/ Petri, C.A. Kommunikationsdisziplinen.
ISF - 76 - 1, GMD, Bonn, 1976

/2/ Petri, C.A.et.al. Advanced Course on General Net Theory of Processes and Systems.
Dept. of Computer Science, University of Hamburg, 1979.

/3/ Wendt, S. The programmed action module: an element for system modelling.
Digital Processes, 1979.

/4/ Dijkstra, E. Hierarchical ordering of sequential processes.
In "Operating System Techniques" ed. Hoare,Perrot.
Academic Press London-New York, 1972.

/5/ Pieper, F. Einführung in die Programmierung paralleler Prozesse.
R. Oldenbourg Verlag, München, 1977.

Entwurf, Beschreibung und Implementation von Systemen mit Hilfe der nebenläufigen Programmiersprache CAP

Franz J. Rammig
Universität Dortmund
Abt. Informatik I

Kurzfassung:

Es wird eine Sprache vorgestellt, die aufbauend auf einem einheitlichen Konzept (zeitbewertete interpretierte Petri Netze) Beschreibungen auf sehr unterschiedlichen Abstraktionsebenen erlaubt. Diese Abstraktionsebenen reichen von der Ebene der Verschaltung von integrierten Bausteinen hin bis zu Warteschlangenmodellen. Auf die Möglichkeit, nebenläufige Prozesse beschreiben zu können, wurde besonderes Wert gelegt (CAP steht für Concurrent Algorithmic Programming Language) /RA1/.

In diesem Beitrag wird das Grundkonzept der Sprache, insbesondere die CAP-Netze, erläutert und spezielle Spracheigenschaften von CAP anhand von fünf kleinen Beispielprogrammen vorgestellt. Diese Beispielprogramme sind:
- *auf der IC-Ebene: Beschreibung eines "8-bit-ripple-counter"*
- *auf der RT-Ebene: Beschreibung eines synchronen sequentiellen Komplementierers*
- *auf der RT-Ebene: Beschreibung eines asynchronen sequentiellen Komplementierers*
- *auf der BS-Ebene: Beschreibung eines Resourcenverwalters*
- *auf der Systeminteraktionsebene: Beschreibung einer Tankstelle.*

Abschließend wird die bisher implementierte CAP-Software (Compiler, Interpreter/Simulator, Debugger, Dokumentationsgenerator, optimierender Cross Code Generator, Maschinenmodellgenerator) kurz vorgestellt.

1. Grundkonzept

Das CAP-Grundkonzept läßt sich als alphanumerisches Äquivalent zu zeitbewerteten interpretierten Petri Netzen charakterisieren, wobei die Interpretation (= Datenmanipulation) in PL/1-artigen Sprachkonstrukten ausgedrückt wird.

Für die Beschreibung des Petri Netzes (Kontrollstruktur der Sprache) werden Label zur Identifikation von Stellen benutzt, während "ON CONDITIONS" auf Labeln zur Beschreibung von Transitionen dienen. Formal erhält man dadurch Sprachkonstrukte, die nicht unähnlich zu Dijkstras "Guarded Commands" /DI1/ sind. Z.B.:

ON(A, B) : C : D : F := G+H; entspricht einer Transition mit Eingangsstellen A und B, Ausgangsstellen C und D sowie der Interpretation F := G+H. In diesem Beispiel ist die Transition nicht zeitbewertet.