Digital Processes, 5 (1979) 213–222

THE PROGRAMMED ACTION MODULE: AN ELEMENT FOR SYSTEM MODELLING

Siegfried Wendt

Dept. of Electrical Engineering, University of Kaiserslautern,
P.O.B. 3049, D-6750 Kaiserslautern, Germany

(Received November 27, 1978)

Abstract

The purpose of this paper is to introduce a concept for the representation of the structure of program controlled systems. Based on a Petri net oriented program definition, an elementary system decomposition is introduced. This decomposition is refined by considering program components each of which belongs to a separate action module. The access graph is a representation of the access relation between action modules and memories, while the token flow between action modules is represented by Petri nets. General relations between automata, action modules and processors are discussed.

Keywords

Program, Petri net, decomposition, action module, memory, access graph, automaton, processor.

1. Introduction

The philosophy presented in this paper originated from a three years' project in which the attempt was made and successfully completed to analyse real industrial software (information retrieval system, operating system) in order to visualize its structure. The approach was strongly determined by the experiene of the analysts in the field of structuring complex hardware. Therefore, it was quite clear from the very beginning that software should not be considered independent from hardware but as an intergrated part of a program controlled system.

A representation of such a system as a set of interconnected modules must not necessarily describe a decomposition of the real physical system, but can be a so-called equivalent decomposition. In electrical engineering, the use of equivalent circuits is a well-known technique which is based on the fact that the system function usually does not completely determine the system structure. The various equivalent structures may differ very much with respect to their transparency: it often happens that the implemented structure is much more difficult to understand than an equivalent one. This is an important aspect for the kind of system modelling described in this paper.

The decomposition presented here is hierarchically refinable and uses only two types of components, which are called action modules and memories. The partitioning of the set of components into two subsets is not new, but has already been proposed by Petri [1]. There, the two types of components are called "instances" and

"channels"; in this form, they are even used in the German standards [2]. Although that system and the system presented here have much in common, they are not the same, and the difference will be pointed out. The intention of this paper, however, is not primarily to compare two decomposition philosophies, but to present one philosophy which proved itself useful in practical applications. Unfortunately, the limited size of such a paper makes it impossible to present examples of these applications.

It is necessary to point out that this paper is not a contribution to the field of synchronization though it deals with programs and Petri nets.

2. Programs and program components

A program is defined as a marked Petri net where instructions may be assigned to transitions and where conflicts may be solved by testing the value of program variables. Each instruction requests the application of a certain operator to certain program variables. Executing the program means selecting a sequence of transitions according to the firing rule and executing the assigned instructions in this sequence.

The common "naked" processors can only execute sequential programs; in this case the Petri net is a state machine graph. Since, however, it is possible to interconnect such processors or to multiplex such a processor by interrupt techniques, the execution of nonsequential programs is solvable. Therefore, this paper is not restricted to sequential programs.

However, the given program definition implies two restrictions with respect to the common idea of a program which must be discussed.

The first restriction concerns the instructions and is of minor importance. The common idea of an instruction includes more than the application of an operator to certain variables. As an example, we shall consider the instruction, "Blow the whistle for one minute". After this instruction has been executed, the value of the assigned variable is the same as before the execution began. This means that this instruction requests the consecutive application of two operators, namely, "Turn on" and "Turn off". Such instructions are not allowed in the given program definition, but, of course, they can always be considered as abbreviations for certain program sections. The example includes a second problem which is its reference to real time. This has nothing to do with applying an operator, but with the time intervals between the applications of operators. There are two possibilities to meet such real time requirements: Either the elementary intervals are known and a program section can be designed where the sum of the elementary intervals has the required value, or the elementary intervals are unknown but negligible with respect to the required duration. Then, a time variable can be introduced whose value is changed by a program independent clock, and this time variable is used in a condition for a program loop.

The second restriction which is implied by the given program definition is a little bit more severe. The program and the variables to which the instructions can refer are two completely separate things. That means that the program cannot

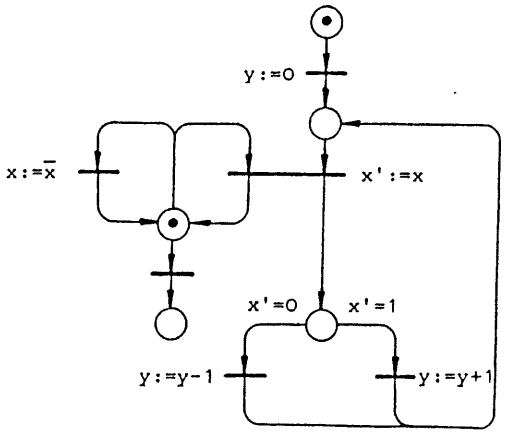

Fig. 1. Example of a program.

contain any instructions which change it. Whenever a self-changing program is discussed – which will only be in Section 4 – it will be referred to as a "processor program".

Figure 1 shows an example of a non-sequential program.

Programs can be so voluminous that it becomes desirable to decompose them according to semantic criteria. Such a decomposition also becomes necessary when a non-sequential program has to be prepared for execution on one or more processors for sequential programs. If the Petri net belonging to the program consists of unconnected parts, this is already a decomposition where the components can again be considered as individual programs according to the definition. If, however, the program is based on a connected Petri net, a decomposition results in components which must be able to transfer tokens between each other. That means that such a component not only contains instructions from the original program but also special instructions of the type: put a token on place *i* of component *j*.

3. Action modules, memories and access graphs

While program decomposition was discussed in the last section, we shall now consider system decomposition. The term "system" here means the entire real or imaginary physical structure in which the process of program execution can or could be observed. Since technology will not be of any relevance here, it is allowed and reasonable to use a very illustrative example for such a system, namely, a man in a certain environment. The environment consists of the stored unchangeable program, the changeable marking and the memory for the variables occurring in the instructions. This system is considered as composed of two components which are called action module and memory. The memory only contains the cells for the variables occurring in the instructions, the action module includes the rest of the system.

If the execution of a program component is considered, the relevant environment of the executing man has to be expanded by the changeable markings of those components into which he can put tokens, and by those men who execute other components and can change his own marking by introducing tokens. This expansion of the environment, however, is not a part of the considered action module, but belongs to other action modules.

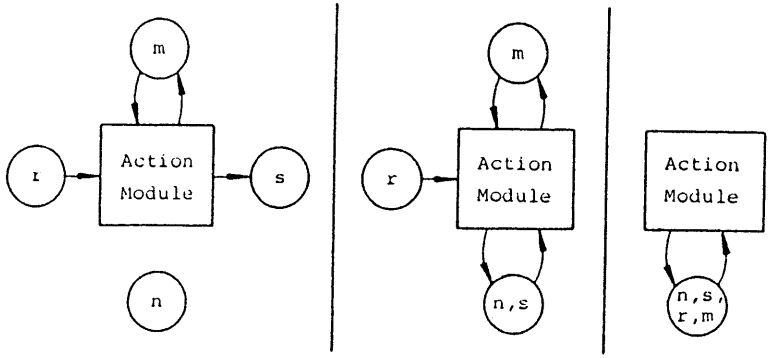

Fig. 2. Different versions of an access graph for the action $(m, s) := F(m, r)$.

The access relation between action module and memory is represented as a bipartite directed graph as shown in Figure 2. It can be decided arbitrarily whether the individual variables of the program are concentrated in a single memory or grouped according to sematic aspects and placed in different memories of smaller dimension. The access graph is no Petri net, that is, there are no tokens. There are three possible ways by which an action module can have access to a memory: namely, reading, setting or modifying. Reading access does not change the contents of the memory (memory r in Figure 2); setting access produces a new content overwriting the old content (memory s in Figure 2); and modifying access produces a new content depending somehow on the old content (memory m in Figure 2). There is a memory n in Figure 2 to which the action module has no access at all.

Modifying access is not necessarily the combination of reading and setting access; this can be seen in Figure 2. Since there is no reading access either to memory s or to memory n, it follows that there is no reading access to the combined memory (n, s). However, there exists the right for modifying access, since setting s together with not changing n means producing a new content depending on the old content. Though it would have been formally justified to introduce a special symbol for modifying access, this is not necessary in practice since it is always possible to represent more details on a lower level, as in the example of Figure 2.

The access graph is of special interest whenever it not only contains a single action module, as in Figure 2, but more action modules which partially have access to the same memories. This is usually the case when the different modules belong to different components of a single program.

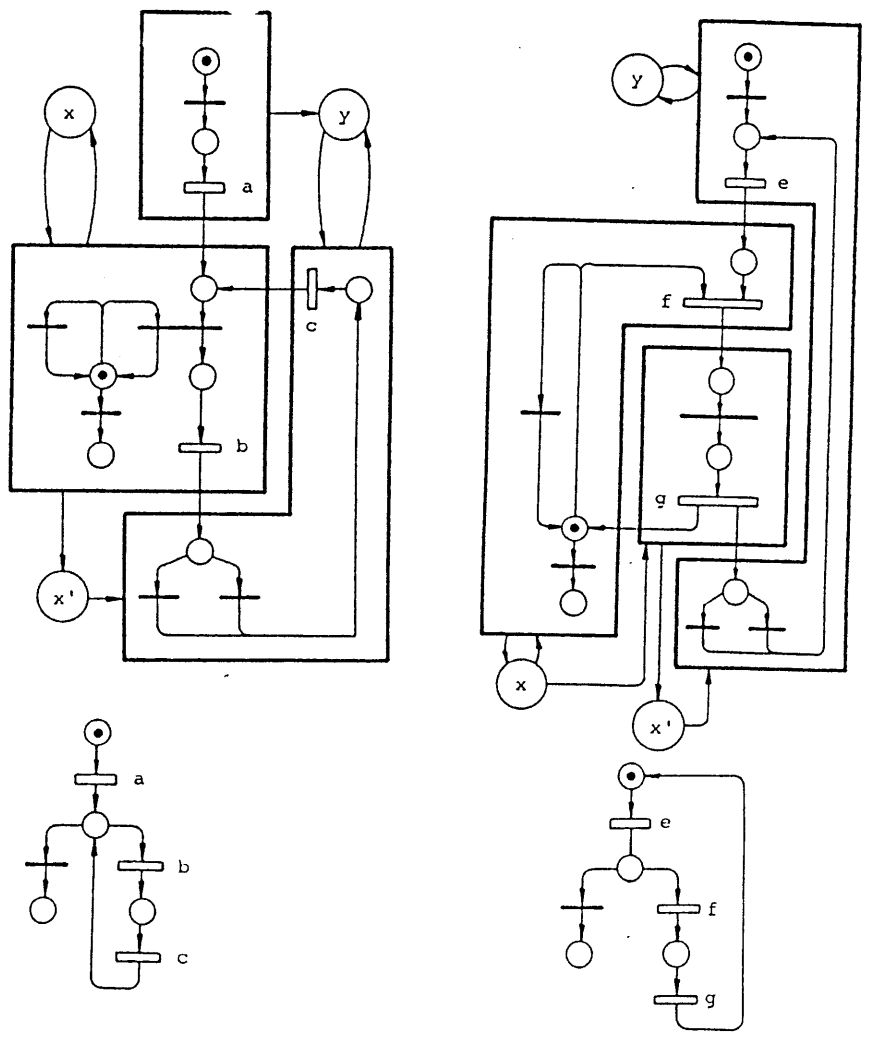

Fig. 3. Two different decompositions of the program from Figure 1.

Figure 3 shows two arbitrary decompositions of the program from Figure 1 into three components. For this, the original program had to be expanded by inserting the token transfer instructions a, b, c, respectively, e, f, g. Since the new programs have to be equivalent to the original program, the insertion of such token transfer instructions is somehow restricted [3]. If the rules are observed which guarantee the program equivalence, one obtains only such decompositions for which the conflicts lie within the action modules, namely, where for each conflict it is determined which action module has to decide it.

Figure 3 also shows the access graph. The action modules communicate with each other in two completely different ways, namely, via the memories and by token transfer. This is the main difference between the two decomposition philosophies mentioned in the introduction. While the channels are the only connections between the "instances", the memories are not the only connections between the action modules. This means that in the first case the channels are not classified, while they are of two different types in the second case.

If a decomposition is based on semantic aspects, the transfer of a token between two action modules always represents a semantically important event. Therefore, it is reasonable to represent a Petri net which only contains these events, because this usually helps to get more insight into the structure of causality of the system. Since the two decompositions in Figure 3 are not based on semantic aspects, the two separate Petri nets containing the token transfer events can, of course, not provide any further insight into the system structure. However, an example for which this could be demonstrated is beyond the scope of this paper.

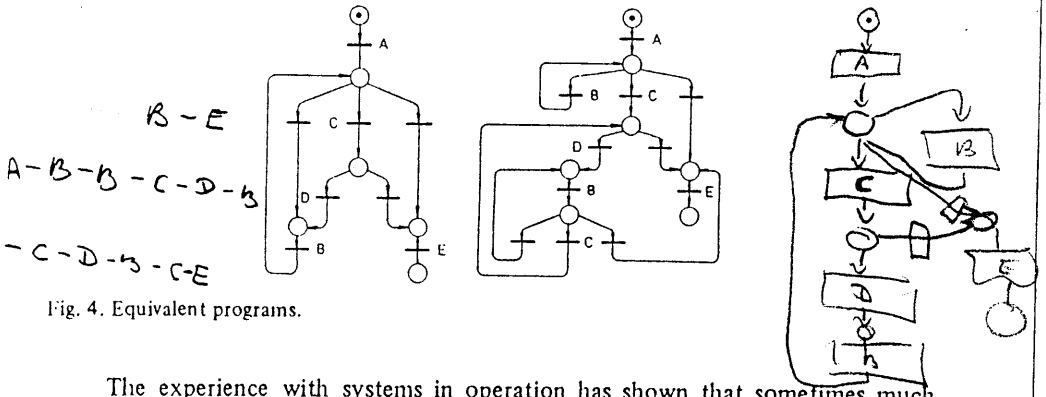

Fig. 4. Equivalent programs.

The experience with systems in operation has shown that sometimes much transparency can be gained by representing a decomposition of a redundant equivalent program instead of the implemented program code. Such redundancy is introduced by simply taking loops apart, as shown in Figure 4. Of course, in this example it would not make sense to ask which version is more transparent since here the instructions assigned to the transitions are only symbols without semantics. Again, it must be said that a convincing example which demonstrates the discussed effect cannot be given in this paper since this would require too much space.

4. Action modules, automata and processors

Action modules, automata and processors are components for systems. We shall discuss how they are related and how they differ. Based on a consideration of the behaviour of an automaton with respect to its input and its output, it is rather simple to find general relationships and differences between the three types of components. When an automaton is implemented as an electronic sequential circuit, it is

well known that there is a difference between a level signal and a pulse signal. This difference, however, can not only be found in the field of electronics, but is also very meaningful with respect to other implementations of automata. The everyday automata always have "pulse signals" as inputs and outputs: inserting a coin corresponds to an input pulse, delivering the paid object – e.g., a box of cigarettes – corresponds to an output pulse. That means that pulse signals in the field of electronics correspond to moving tokens in the field of mechanics. A mechanical input for a level signal could be a selector switch which can be switched to one of several different positions and a mechanical output for a level signal could be an indicator which can be read.

An automaton with a pulse input is seen as waiting passively for an external start pulse which makes it active. An automaton with a level input, however, is seen as being driven by its own energy to sample the input level with internally defined frequency; each sampling event provides an input element which then is used in the argument of the transition and output functions. Correspondingly, an automaton with a pulse output is seen as pushing someone to become active, while an automaton with a level output is providing a signal to be sampled.

It is obvious that an automaton with pulse input and output can always be seen on a lower level as an automaton with level input and output. This requires the expansion of the sets of input and output elements. When the original input set, for example, was a set of coins, the new element, "No Coin", has to be added. In general, however, it is not possible to interpret any given automaton with level input and output on a higher level as an automaton with pulse input and output.

Now we shall consider the action module without token transfer. The input of the module is connected to the output of a memory, and this is a level signal connection. The activity of the action module is generated internally; that is the module is not driven by external pulses, but has to sample the state of the memory. The output of the action module is connected to the input of the memory, and this is a pulse signal connection since the module has access to set or modify the state of the memory.

So, the action module without token transfer is a special type of automaton, namely, with level input and pulse output; while the memory is an automaton of the opposite type, that is, with pulse input and level output.

Token transfer between two action modules means that there is a direct connection between these modules, and the question is whether this is a level or pulse signal connection. This question is answered by decomposing the action module according to its definition. Figure 5(a) shows the elements which were introduced to define the action module: the memory for the program, the memory for the marking and the executing unit ("the man"). The executing unit can have its individual auxiliary memory, which makes it possible to execute a single instruction from the program in more than one step. A token transfer between two action modules requires a connection between the executing unit of one module and the memory for the marking of the other module, and this is a pulse signal connection. Since the execution units themselves are action modules on a lower level, the com-

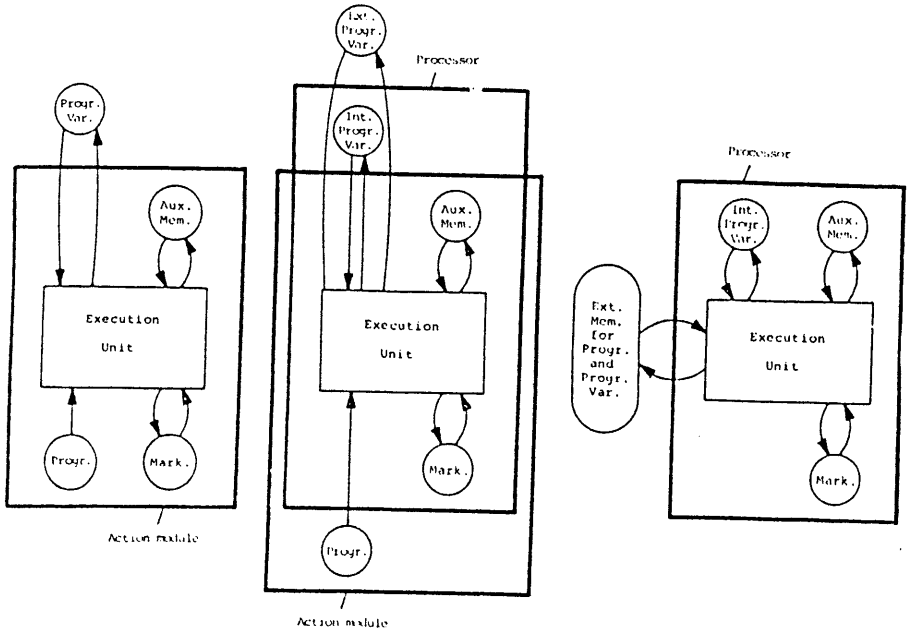

Fig. 5. Relations between action modules, execution units and processors.

munication between two action modules via token transfer on a higher level means communication via memory on a lower level.

In the same way as the action module in Figure 5(a) has been decomposed, the executing unit again can be decomposed and a lower-level executing unit will appear, and so on. The lowest level has been reached when the innermost program has a trivial structure consisting of a loop with only one transition and one marked place. The single instruction of this program is executed periodically and has the form $Z := f(Z)$, where Z is the state of all memories and f is a transition function which in general is very complex.

Figure 5(b) shows the relation between an action module and a processor: The memory for the program must lie outside of the processor but inside of the action module, and the memory for the program variables which lies outside of the action module, may be partially included in the processor, usually in the form of processor registers.

If the program can change itself during its execution, the memories for the program and for the program variables can no longer be separated. Such a program, of course, can be executed by a processor (see Fig. 5(c)) which itself is an action module and contains an execution unit, but there is a great difference to the decomposition in Figure 5(a): While the action module in Figure 5(a) is described by its program, the action module in Figure 5(c), which is a processor, is described as a combination of other given action modules and memories. This is not a specialty of

the example in Figure 5 but can be generalized: an action module can be introduced by describing its program or by combining given modules and memories to a hierarchically higher module. In the latter case there also exists a program for the new module, but this program usually is not easy to derive. However, this hardly matters since the program is also usually of no interest.

Again we must emphasize what we have mentioned in the introduction: the discussed modelling of systems does not require that the action modules and memories really are implemented as individual, physically separable components. It is sufficient that such an implementation can be imagined and that it is functionally equivalent to the actual implementation.

5. Conclusion

While in classical disciplines there exists a stable and widely accepted repertoire of concepts and models, the corresponding repertoire in computer science still is very small and unstable. Hence continuously new concepts and models are proposed which then compete for acceptance. There are two types of concepts, namely structure-oriented and procedure-oriented ones. The first will help to put down design decisions in the form of formal and transparent "plans", the second will help to formalize the design procedure by alogorithms which support the search for optimal design decisions. The second always requires the first. In this sense, this paper presents a structure-oriented concept as a base for transparent plans of program controlled systems. The application of this concept will be the subject of another paper.

References

[1] Petri, C. A., Kommunikationsdisziplinen, *ISF-76-1* (Gesellschaft für Mathematik und Datenverarbeitung, Bonn).
[2] DIN 66200, Teil 1 (Beuth Verlag, Köln)
[3] Ullrich, G. (unpublished), "Der Entwurf von Steuerstrukturen für parallele Abläufe mit Hilfe von Petri-Netzen" (*PhD thesis*, University of Hamburg, 1976).

Résumé

L'objectif de cet article est d'introduire un concept pour représenter la structure des systèmes contrôlés par programmation. Une décomposition élémentaire des systèmes est introduite, basée sur une définition des programmes en tant que réseaux de Petri. Cette décomposition est affinée par la prise en considération de programmes composants appartenant chacun à un module d'action séparé. Le graphe d'accès représente la relation d'accès entre modules d'action et mémoires, alors que l'écoulement des marques entre les modules d'action est représenté par des réseaux de Petri. Des relations générales entre automates, modules d'action et processeurs sont discutées.

Zusammenfassung

In der Arbeit wird ein Konzept eingeführt für die Darstellung der Struktur programmgesteuerter Systeme. Basierend auf einer Petri-Netz orientierten Programm-Definition wird

eine elementare Systemzerlegung eingeführt. Diese Zerlegung wird verf durch die Betrachtung von Programmkomponenten, die einem separaten Aktionsm angehören. Der Zugriffsgraph ist eine Darstellung der Zugriffsbeziehungen zwischen Aktionsmoduln und Speichern, während der Marken fluss zwischen Aktionsmoduln durch Petri-Netze dargestellt wird. Allgemeine Beziehungen zwischen Automaten, Aktionsmoduln und Prozessoren werden diskutiert.

Digital Processes, 5 (1979) 223–234

A METHOD FOR DECOMPOSING INTERPRETED PETRI NETS AND ITS UTILIZATION

J.M. Toulotte and J.P. Parsy

Centre d'Automatique, Université des Sciences et Techniques de Lille,
B.P. 36, 59650 Villeneuve d'Ascq, France.

(Received September 20, 1978)

Abstract

Petri nets are a very powerful and now commonly employed tool for describing logical processes.

For an easy realization by different technologies, and particularly by programmable controllers, we propose a method for decomposition into sub-nets which is directly implementable. This heuristic method gives an algorithm using an examination of all transitions but without enumerating all markings.

Keywords

Petri net, decomposition, programmable controllers, Grafcet.

1. Introduction

Decomposition of Petri nets has always been a research problem because of its importance in logical automata realization. We propose here a heuristic method for obtaining a decomposition in specific state machines. For safe nets, this is a partition in respect of place marking. It is possible to deduce different realizations but the method seems particularly well suited for implementation on programmable controllers.

2. Petri graph

A Petri graph is defined by $G=\{P,T,E,S\}$ where P is a finite set of places denoted by circles $P=\{p_1,\ldots,p_n\}$, T is a finite set of transitions denoted by bars $T=\{t_1,\ldots,t_m\}$. E and S are forward and backward incidence relations, applications, respectively, from $P\times T$ and $T\times P$ in $\Pi=\{0,1\}$:

$$E(p_j,t_i)=1$$

if there is an arc from place p_j to transition t_i

$$S(t_m,p_e)=1$$

if there is an arc from transition t_m to place p_e.

Then p_j is an input place of t_i, p_e an output place of t_m. We define by ${}^{\cdot}t$ and $t^{\cdot}$, respectively, the set of input and output places of t transition. More generally, E and S are applications from $P\times T$ and $T\times P$ in $\mathbb{N}$; we have then generalized nets with $E(p_j,t_i)=k$ if there is a k-arc from p_j to t_i and $S(t_m,p_e)=k'$ if there is a k'-arc from t_m to p_e.