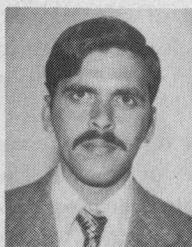


- [22] Z. Manna and P. Wolper, "Synthesis of communicating processes from temporal logic specifications," in *Proc. Workshop Logics of Programs (Springer-Verlag Lecture Notes in Comput. Sci.)*, vol. 131, 1981.
- [23] S. Owicki and L. Lamport, "Proving liveness properties of concurrent programs," *ACM Trans. Programming Languages Syst.*, vol. 4, pp. 455-495, July 1982.
- [24] A. Pnueli, "The temporal semantics of concurrent programs," in *Semantics of Concurrent Computation (Springer Lecture Notes in Comput. Sci.)*, vol. 70, June 1979, pp. 1-20.
- [25] —, "On the temporal analysis of fairness," in *Proc. 7th Annu. Symp. POPL*, Jan. 1980, pp. 163-173.
- [26] K. Ramamritham and R. M. Keller, "Specifying and proving properties of sentinel processes," in *Proc. 5th Int. Conf. Software Eng.*, Mar. 1981, pp. 374-382.
- [27] —, "On synchronization and its specification," in *Springer Lecture Notes in Comput. Sci.*, vol. 111, June 1981.
- [28] K. Ramamritham, "Specification and synthesis of synchronizers," Ph.D. dissertation, Univ. Utah, Aug. 1981.
- [29] H. A. Schmid, "On the efficient implementation of conditional critical regions and the construction of monitors," *Act Informatica*, vol. 6, pp. 227-249, 1976.
- [30] R. L. Schwartz and P. M. Melliar-Smith, "Temporal logic specifications of distributed systems," in *Proc. 2nd Int. Conf. Distributed Syst.*, Apr. 1981.
- [31] A. C. Shaw, "Software specification languages based on regular expressions," in *Proc. Software Tools Workshop*, May 1979, pp. 1-39.



Krithivasan Ramamritham received the B.Tech degree in electrical engineering and the M.Tech degree in computer science from the Indian Institute of Technology, Madras, India, in 1976 and 1978, respectively, and the Ph.D. degree in computer science from the University of Utah, Salt Lake City, in 1981.

Currently, he is an Assistant Professor in the Department of Computer and Information Science, University of Massachusetts, Amherst. His research interests include software engineering, operating systems and distributed computing.

Dr. Ramamritham is a member of the Association for Computing Machinery and the IEEE Computer Society.



Robert M. Keller received the B.S. and M.S.E.E. degrees from Washington University, St. Louis, MO, and the Ph.D. degree from the University of California, Berkeley.

Currently, he is a Professor of Computer Science at the University of Utah, Salt Lake City. From 1970-1976 he was an Assistant Professor of Electrical Engineering at Princeton University. His current research interests deal with numerous topics relating to multiprocessor implementations of functional languages, particularly using reduction and data-flow computation models.

Dr. Keller is a member of the Association for Computing Machinery and the IEEE Computer Society.

Distributed Software System Design Representation Using Modified Petri Nets

STEPHEN S. YAU, FELLOW, IEEE, AND MEHMET U. CAGLAYAN, MEMBER, IEEE

Abstract—A model for representing and analyzing the design of a distributed software system is presented. The model is based on a modified form of Petri net, and enables one to represent both the structure and the behavior of a distributed software system at a desired level of design. Behavioral properties of the design representation can be verified by translating the modified Petri net into an equivalent ordinary Petri net and then analyzing that resulting Petri net. The model emphasizes the unified representation of control and data flows, partially ordered software components, hierarchical component structure, abstract data types, data objects, local control, and distributed system state. At any design level, the distributed software system is viewed as a collection of software components. Software components are externally described in terms of their input and output control states, abstract data types, data objects, and a set of control and data transfer specifications. They are interconnected through the shared control states and through the shared data objects. A system component can be viewed internally as a collection of subcomponents, local control states, local abstract data types, and local data objects.

Index Terms—Control flow and data flow, design analysis, distributed software system, modified Petri net, software design representation.

I. INTRODUCTION

WE CONSIDER that a distributed computer system has a number of processing nodes connected by a message-based communication network. In addition to the physical distribution of hardware, conceptual distribution of both data and control is an essential characteristic of the system. To emphasize decentralized control, processing nodes will be highly autonomous in their availability, type of service they provide, their concern for protection of resources, and their reliability. The effects of actual choice of processing and communication hardware and system topology on design issues are not considered here.

The design of distributed software systems continues to be a challenging area of software engineering. A number of informal and formal design methods, which are primarily concerned with sequential software systems, have been proposed [1]-[5]. These methods do not directly address the design problems associated with parallel and distributed systems. Although

Manuscript received November 6, 1981; revised March 21, 1983. This work was supported by the U.S. Army Research Office under Contract DAA-C29-80-K-0092.

S. S. Yau is with the Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL 60201.

M. U. Caglayan was with the Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL 60201. He is now with the University of Petroleum and Minerals, Dhahran, Saudi Arabia.

some have facilities for the representation of concurrency, synchronization, and process relations, a general approach to the design of parallel, and distributed software systems is lacking.

Models which are suitable for representing distributed computer systems can be divided into a number of groups. The first group includes distributed programming languages recently developed for expressing direct communication, rather than communication and synchronization through shared variables, between program modules and processes. The two most important members of this class are distributed processes [6] and communicating sequential processes [7]. These languages are basically extensions of previously known programming languages with an expanded set of language constructs to handle new features like communication, nondeterminism, and unification of concepts like monitors [8], processes, and classes [9]. Another group includes three formal models that reflect the characteristics of distributed processing systems. These are actor systems [10], calculus of communicating systems [11], and networks of parallel processes [12]. A very good comparative evaluation of these models can be found in [13]. The third group consists of classical graph theoretic models, among which Petri nets [14]-[17] appear to be the most fully developed. Most of the recent work done on Petri nets and related models can be found in [18], and surveys of Petri nets in [19], [20].

Most of the above models of parallel and distributed systems are formal, computation oriented models which do not address the design issues directly. For the purpose of software design, several high level design methods have been proposed [21]-[27]. The method used in [23] is based on attributed grammars. SARA design methodology [25] is based on UCLA graphs. COSY [26] is a design specification language based on path expressions and later extended to specify distributed systems [27].

In this paper, we will present a design representation and analysis technique for parallel and distributed software systems. The representation technique is not a design method; it is only a means for representing, and thus documenting, the design of a distributed software system. The representation is based on a modified form of Petri net, and enables us to graphically represent the structure and dynamic behavior of a distributed software system. Both the control and data flows are included in a single graph representation of the design. Partially ordered software components, hierarchical component structure, abstract data types and data objects, local control, and distributed system state are emphasized. Software components are externally described in terms of their input and output control states, associated data types and data objects, and a set of control and data transfer specifications. Interconnection of software components is defined through shared control states and through shared data objects. A system component can be viewed internally as a collection of partially ordered subcomponents, local control states, local data types, and local data objects. The design representation in the form of a modified Petri net can be transformed into an equivalent Petri net for the purpose of analyzing the design for concurrency related properties such as mutual exclusion and deadlock-freeness.

II. MODIFIED PETRI NETS AND DESIGN REPRESENTATION

Petri nets are abstract, formal models of information and control flow in systems exhibiting concurrency and asynchronous behavior [20]. They are very simple and natural in structure, yet quite powerful in modeling and analyzing such systems. Informally, a Petri net, sometimes called a marked Petri net, consists of a Petri net graph, an initial marking, and fixed simulation rules. A Petri net graph is a collection of two types of nodes, called transitions and places, connected by directed arcs. Places are represented by circles and transitions by bars. Places can hold tokens, which are represented by small dots. The number of tokens in a place is called the marking of that place. The marking of the net is the collection of all place markings. Thus, the initial marking is the initial distribution of tokens to places. Simulation rules define how new markings can be obtained from a current marking. A transition whose input places hold tokens "fires" by removing tokens from its input places and adding tokens to its output places; this results in a new marking of the net. An example of a Petri net is shown in Fig. 1.

Several subclasses and extensions of ordinary Petri nets [20] have been studied either to increase the modeling power or modeling convenience, or to facilitate the solution of analysis problems. Generalized Petri nets [20] are Petri nets with multiple arcs between a place and a transition. They are equivalent in modeling power to ordinary Petri nets. A fundamental extension to Petri nets is the incorporation of inhibitor arcs into the Petri net. An inhibitor arc allows a transition to fire if its associated input place has a marking of zero tokens. It is shown in [28] that Petri nets with inhibitor arcs are equivalent in modeling power to Turing machines. Other extensions such as coordination nets [29] and timed-nets [30] are also equivalent to Turing machines. Several subclasses have been proposed in order to facilitate the analysis of Petri nets. State machines are Petri nets in which transitions can have only one input place and one output place. Decision-free nets, also known as marked graphs, are Petri nets in which each place can have only one input transition and one output transition [31]. Other subclasses include free-choice nets, conflict-free nets, persistent nets, simple Petri nets, and restricted Petri nets [20].

Petri nets can be used to model both the static and dynamic properties of systems. Static properties of systems are represented by the graphical part of a Petri net. Dynamic properties of a system can be determined from the Petri net graph, the initial marking, and the simulation rules.

There are many potential advantages of modeling a dynamic system using Petri nets: the ability to produce a precise, graphical representation, the existence of analysis tools for determining and verifying the dynamic behavior of systems from their structure, and the capability of designing systems using top-down and/or bottom-up approaches. The capabilities of representing local control and concurrent, conflicting, non-deterministic, and asynchronous events, are the most useful properties of Petri nets for modeling a system. A Petri net defines a partial ordering of event occurrences and no assumptions are made regarding the flow, measurement, and direction of time.

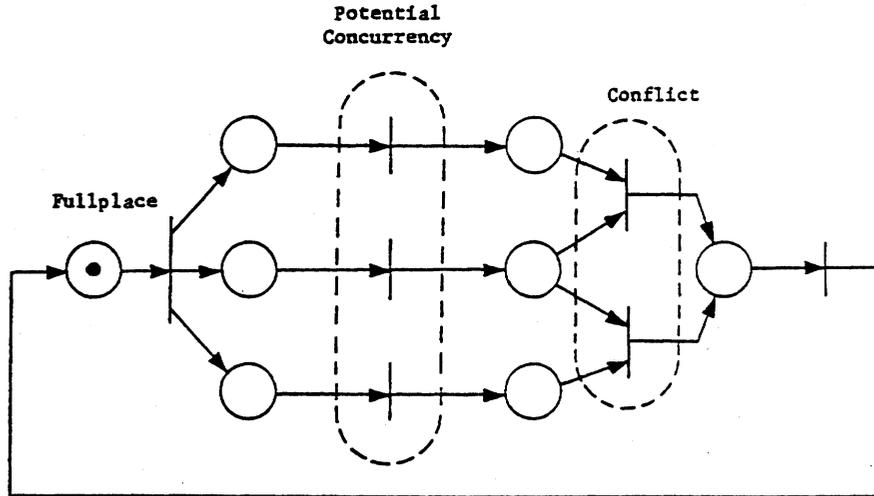


Fig. 1. An example of a Petri net.

Petri net models are especially suitable for describing the control flow aspects of a system. Data flow is generally ignored and data-dependent control flow is represented as a part of the nondeterministic behavior of the modeled system. Although this aspect of modeling control flow can only be attributed to the "low-level" nature of Petri nets, it is obvious that there is a need for "higher level" models which will directly relate to the issues in software design.

There have been modifications to Petri nets which relate to our objective in modeling distributed software systems. Evaluation nets [32], macro E-nets [33], and pro-nets [34] proposed different modifications to the transition firing rules and attempted to represent data flow by associating attributes with tokens. The data flow language EDDA [35] uses a modified Petri net for requirements analysis and system specification. In Valette's model [36], control flow is represented by a Petri net and the data flow, data operations, and memory locations by a separate data graph. By associating a set of data graph predicates with each Petri net transition, the operations of the data graph are controlled by the Petri net.

In this paper, we will modify Petri nets for the representation of control flow and data flow in distributed software systems. Our modified Petri net consists of a set of control state variables, a set of abstract data types, a set of data objects, and a set of software components which are connected to each other through the control state variables and through the data objects. The control state variables correspond to the places of a Petri net. Software components correspond to the nonprimitive transitions of a Petri net, where a nonprimitive transition has a Petri net subgraph as its inner structure and does not fire instantaneously [37]. The execution of a software component can be regarded as the firing of a nonprimitive transition. The nonprimitive transition firing rule, which is fixed in ordinary Petri nets, is generalized in modified Petri nets by associating with each software component a control transfer specification, which gives the control flow through that component. Each component has associated with it a data transfer specification which represents the data flow through that component. An example of a modified Petri net is shown in Fig. 2. Initially, the system is in the total control state such that

$p_1 = \text{enabled}$ and $p_2 = p_3 = \dots = p_{11} = \text{disabled}$, where p_i , $i = 1, \dots, 11$ are the control state variables; and the component C_1 starts to be executed. After the execution of C_1 is completed, components C_2 and C_4 will start to be executed in parallel and they will add some information to the data objects D_1 and D_2 , respectively. The execution of C_3 is synchronized with the termination of both C_2 and C_4 . C_3 will use the information in data objects D_1 and D_2 and will add information to D_3 during its execution. Components C_5 and C_6 will be executed after C_3 has completed its execution. Note that C_5 and C_6 share the information in D_3 . After the execution of C_5 (or C_6) terminates, either C_2 (or C_4) is restarted or the control state set represented by $p_{10} = \text{enabled}$ (or $p_{11} = \text{enabled}$), independent of the condition of all other control state variables is reached. The system completes its execution if both p_{10} and p_{11} are enabled. If either C_5 or C_6 decides to enable p_{10} or p_{11} , respectively, then the system may be assumed to deadlock. In the following sections, the concepts of control state, component, control and data transfer specifications, and component interconnections will be explained in detail.

System States and Control States

The total state of a distributed software system is defined as the combination of the total control state of the system and the total state of the data objects in the system. The total control state of the system is the collection of the individual control states of the system. In Fig. 2, after C_3 completes its execution, but before C_5 or C_6 begins theirs, the total control state is $p_6 = p_7 = p_8 = p_9 = \text{enabled}$, and all remaining p 's disabled. The total state of the data objects is the collection of states of the individual data objects, i.e., D_1, D_2, D_3 in Fig. 2. The current state of a data object is the current value associated with that data object.

Let $S = \{s_i | i = 1, \dots, n\}$ be the set of control state variables of a distributed software system, where each $s_i \in S$ is enabled or disabled. Whether s_i is enabled or disabled is determined by the marking function

$$M: S \rightarrow \{0, 1, 2, \dots\}$$

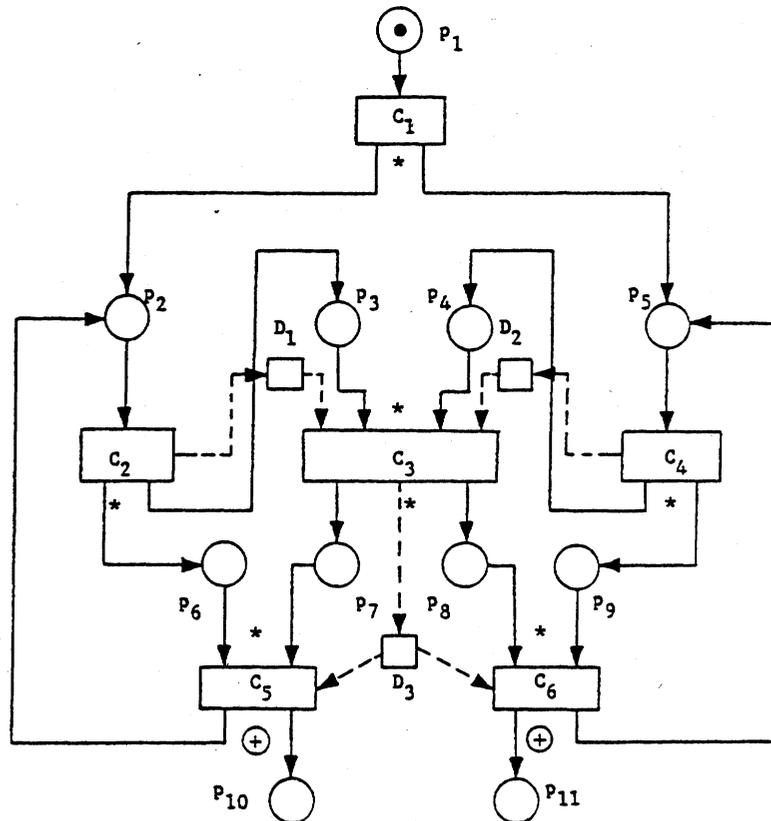


Fig. 2. An example of a modified Petri net.

where $s_i \in S$ is enabled if $M(s_i) \neq 0$ and disabled if $M(s_i) = 0$. The value of $M(s_i)$ is the control state value associated with s_i , i.e., the marking of a Petri net place. Control state values other than 0 or 1 will be considered to be the result of an inconsistent or faulty design. A control state variable s_i , for which $M(s_i) = k_i$, is graphically represented by a small circle containing k_i tokens. For a distributed software system with n control state variables, the total control state at a j th time instance is the n -tuple $m_j = \langle k_{j1}, k_{j2}, \dots, k_{jn} \rangle$, where $M(s_i) = k_{ji}$, $s_i \in S$, $i = 1, \dots, n$. The total control state space is the set of n -tuples $\{m_j, j = 0, 1, \dots\}$. It should be noted that m_j is equivalent to the marking of all Petri net places and the set $\{m_j, j = 0, 1, \dots\}$ is the reachability set of the Petri net, i.e., the set of markings reachable from the initial marking. There is a set of final control state variables F , which is a subset of the power set of S . If all $s_i \in F$ are enabled, execution of the system is considered to be complete. There is also an initial marking function M_0 , which determines the initial total control state m_0 , such that

$$M_0: S \rightarrow \{0, 1\}.$$

A total control state is distributed by its very nature since a control state variable cannot determine the total control state alone. Emphasis is placed on the representation and manipulation of the distributed control states since we are interested in the distributed components of the system and the locality of the control states rather than the total system state. This is due to the fact that information on the total state of a large distributed software system is typically incomplete. The system control states will basically be used for tracking the control flow.

Abstract Data Types and Data Objects

Let T be the set of system-wide abstract data types and D the set of system-wide data objects. A data object is a triple $\langle d_n, t_n, v \rangle$, where $d_n \in D$ is a data object name, $t_n \in T$ is a data type name, and v is the current value of the data object, which is of data type t_n . The abstract data types and the data objects are graphically represented by small squares in our modified Petri net.

An abstract data type is a collection of values and operations. The operations of an abstract data type will subsequently be defined in terms of the components of our model. The specification of abstract data types will be carried out using techniques similar to Guttag's algebraic approach [38], [39]. Although the algebraic specification approach has its problems, it is a well developed formal specification technique and is close to the set theoretic nature of our design representation due to its abstract algebra foundation.

An algebraic specification of an abstract data type consists of a syntactic specification, a semantic specification, and a restriction specification [39]. The syntactic specification identifies type names, domain and range value sets, operation names, and type checking information. The semantic specification is an axiomatic definition of the meaning of operations. The restriction specification basically identifies limitations placed on values and operations, like error conditions. Semantic and restriction specifications include the control states under which the execution of an operation is started and completed. Some of the operations in a data type can be designated as atomic or indivisible operations, which, when executed, will either be carried out to their completion, or will be aborted

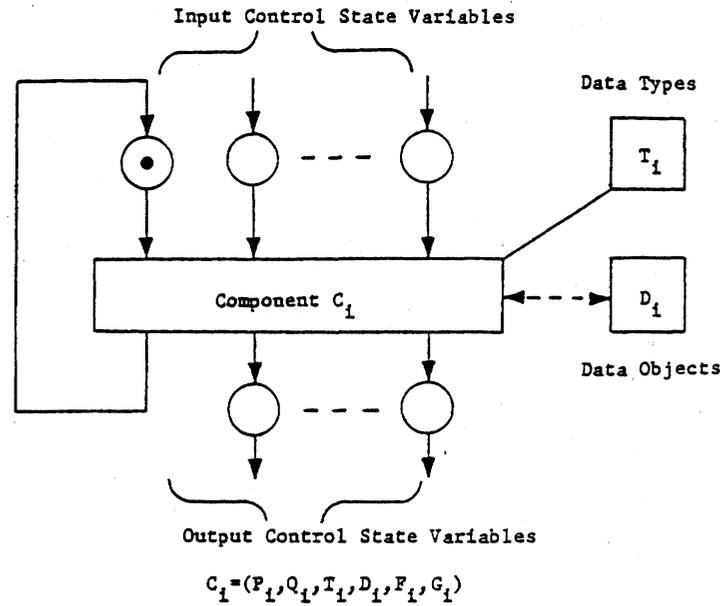


Fig. 3. The external view of a software component.

leaving the state of the data objects associated with them unchanged.

A system-wide set of abstract data types is partially known by each component. This set contains elementary data types like integer, real, Boolean, character, string, etc., with well known value and operation sets, plus user defined abstract data types.

Software Components (External View)

A software component $C_i \in C$, or simply a component, corresponds to a nonprimitive transition of a Petri net and is represented by the 6-tuple

$$C_i = (P_i, Q_i, T_i, D_i, F_i, G_i)$$

where P_i and Q_i are the set of input control state variables and the set of output control state variables, respectively; both are nonempty subsets of S . T_i and D_i are the set of abstract data types and the set of data objects of the component; these are subsets of T and D , respectively. F_i and G_i are the control transfer specification and the data transfer specification of the component, respectively.

A software component is shown in Fig. 3. Graphically, a software component is represented as a rectangle. With respect to a component, the connection of the control state variables are represented by directed solid lines, the connection of data types by solid lines, and the connection of the data objects by directed dotted lines. The set of input control state variables is used to start the execution of the component. During its execution, the component changes the values of its output control state variables. This continues until the execution of the component terminates. The sets of input and output control state variables need not be mutually exclusive. Control state variables common to both of these sets may be used to model the ready (or busy) state of the component and/or to model recursion. The sets T_i and D_i contain the abstract data types and data objects external to the component. These sets may be shared with the other system components.

Two transfer specifications, F_i and G_i , completely identify

all the control and data flows through a system component C_i . The control transfer specification F_i is the pair $(F_{in}^{(i)}, F_{out}^{(i)})$. $F_{in}^{(i)}$ is the input control transfer specification which provides the input control states which initiate the execution of C_i and also describes how the input control state variables change immediately after the execution of C_i begins. $F_{out}^{(i)}$ is the output control transfer specification which provides the sequence of output control states which will exist during the execution of C_i . Each $F_{in}^{(i)}$ is an expression over the input control state variables with the operators “*”, “+”, “++”, and “⊕”. Application of these operators to the input control state variables are defined in Table I. A parenthesized integer that follows a control state variable defines the priority assignment for that variable. It is assumed that a component C_i will start to be executed if the expression for $F_{in}^{(i)}$ evaluates to “enabled.” Note that because the order of application of the operators can affect the outcome of $F_{in}^{(i)}$, the expression for $F_{in}^{(i)}$ should be parenthesized in order to explicitly indicate the order of application of these operators. Operators “*”, “+”, “++”, and “⊕” are graphically represented by the symbol being placed between the arcs, or groups of arcs, that connect the input control state variables to the component. Priority assignments are shown similarly.

The output control specification $F_{out}^{(i)}$ specifies the sequence of output control states which will exist during C_i 's execution. $F_{out}^{(i)}$ can be given in either a data-independent, data-dependent, or mixed form. $F_{out}^{(i)}$ is data-independent if it is an expression of the output control state variables of C_i together with the operators “*”, “+”, “⊕”, and “;”. Application of these operators to the output control state variables is defined in Table II. Similar to $F_{in}^{(i)}$, it is necessary to parenthesize the expression of $F_{out}^{(i)}$ in order to indicate the order of application of the operators.

$F_{out}^{(i)}$ is data-dependent if it is given in the general form

$$F_{out}^{(i)} = F_{out1}^{(i)} ; F_{out2}^{(i)} ; \dots ; F_{outn}^{(i)}$$

where $F_{outj}^{(i)}$ is the specification of the j th change to the output control state variables of C_i . Each $F_{outj}^{(i)}$ is a data-dependent

TABLE I
INPUT CONTROL TRANSFER SPECIFICATION OPERATORS

States of input control state variables before execution		Results generated by the operator	Values of input control state variables after execution	
P_1	P_2	$P_1 * P_2$	P_1	P_2
D	D	D	--	--
D	E	D	--	--
E	D	D	--	--
E	E	E	$M(P_1) - 1$	$M(P_2) - 1$
$P_1 + P_2$				
D	D	D	--	--
D	E	E	--	$M(P_2) - 1$
E	D	E	$M(P_1) - 1$	--
E	E	E	$M(P_1) - 1$	$M(P_2) - 1$
$P_1 ++ P_2$				
D	D	D	--	--
D	E	E	--	$M(P_2) - 1$
E	D	E	$M(P_1) - 1$	--
E	E	E	$M(P_1) - 1$	--
$P_1(i) ++ P_2(j)$				
D	D	D	--	--
D	E	E	--	$M(P_2) - 1$
E	D	E	$M(P_1) - 1$	--
E	E	E	$M(P_1) - 1$	--
$P_1 \oplus P_2$				
D	D	D	--	--
D	E	E	--	$M(P_2) - 1$
E	D	E	$M(P_1) - 1$	--
E	E	D	--	--

D : Disabled ($M(p_i) = 0$)

E : Enables ($M(p_i) > 0$)

-- : No change in the value

$M(p_i)$: The value of control state variable p_i , which is the number of tokens in p_i .

function,

$$F_{out}^{(i)}(D_i): Q_i \rightarrow Q_{ij}$$

where $Q_{ij} \subseteq Q_i$, and

$$M(q_k) \leftarrow M(q_k) + 1$$

for all $q_k \in Q_{ij}$. That is, a subset Q_{ij} of the set of output control state variables is selected according to the current

TABLE II
OUTPUT CONTROL TRANSFER SPECIFICATION OPERATORS

Operator	Values of output control state variables	
	P_1	P_2
$P_1 * P_2$	$M(P_1) + 1$	$M(P_2) + 1$
	$M(P_1) + 1$	--
$P_1 + P_2$	or --	$M(P_2) + 1$
	or $M(P_1) + 1$	$M(P_2) + 1$
$P_1 \oplus P_2$	$M(P_1) + 1$	--
	or --	$M(P_2) + 1$
$P_1 ; P_2$	First	Then
	$M(P_1) + 1$	$M(P_2) + 1$

values of data objects and the values of these control state variables are incremented by one. An example of a data-dependent output control transfer specification is the following:

$$F_{out} = \text{if cond then } \{q_1, q_2\} \text{ else } \{q_3\}$$

where "cond" is a Boolean expression on some data objects. A similar modification of output control state variables can be specified in a data-independent way by the expression $F_{out} = (q_1 * q_2) + q_3$.

F_{out} can also be given in a mixed form which is as a combination of data-independent and data-dependent specifications. This can be done by specifying some F_{outj} of the general form data-dependent F_{out} specification in the data-independent form. F_{out} is generally data-dependent at the lower levels of a design representation, and is data-independent at the higher levels. Thus, data-dependent specifications are postponed to lower levels of design.

When a long (or infinite) and repeating sequence of the same subexpression exists in the expression for F_{out} , such a sequence can be represented in a shorter notation by writing the number of times a subexpression is repeated ("*" if infinitely many) as a superscript of that subexpression. As an example, $F_{out} = (p_1 + p_2); (p_1 + p_2); (p_1 + p_2)$ can be written as $F_{out} = (p_1 + p_2)^3$, and $F_{out} = (p_1 + p_2); \dots; (p_1 + p_2)$ as $F_{out} = (p_1 + p_2)^*$. Note that the superscript is not a new operation; it is introduced for notational convenience only. Simple examples of graphical representations of F_{in} and F_{out} are shown in Fig. 4. It seems that complicated expressions of both F_{in} and F_{out} cannot clearly be represented in the graphical form.

The data transfer specification G_i of a component C_i allows C_i to manipulate the values of the data objects associated with it. Let RD_i and WD_i be the sets of input and output data objects of C_i , respectively, where RD_i and WD_i are not necessarily disjoint. The data transfer specification can be given as a set of functions $G_i = \{g_{ij}, j = 1, \dots, k\}$, where $wd_{ij} = g_{ij}(RD_i)$, and $wd_{ij} \in WD_i$. The operations that can be used in each g_{ij} are those allowed by the abstract data types. The operations can be combined to compose sequential components (processes) using the ordinary language constructs for sequential, decision, and loop structures. Nondeterministic constructs are

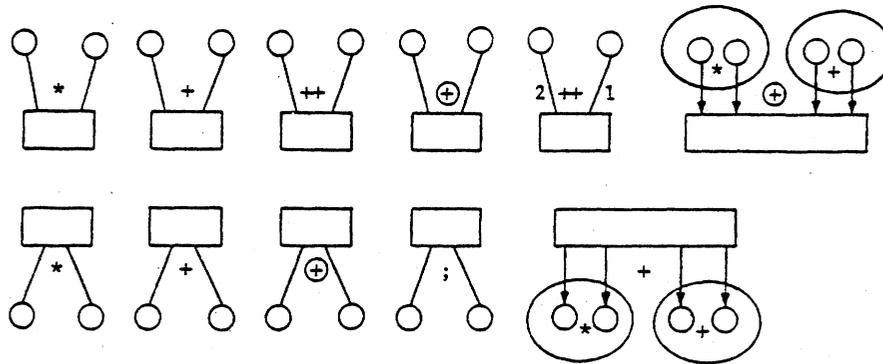


Fig. 4. The graphical representation of the operators in input and output control transfer specifications.

represented by guarded commands [40] and the independent parallel operations by a set of operations.

Interconnection of Components

To complete the system structure, an interconnection of system components can be defined from two points of view. First, components are interconnected through their control states variables. This is called control interconnection. Second, they are interconnected through data objects if they share them. This is called data interconnection.

The interconnection relation R is a binary relation on the Cartesian product of the components:

$$R \subseteq C \times C.$$

The actual interconnection of system components is made through system control state variables, or through the system data objects by selecting P_i 's and Q_i 's, or D_i 's and D_j 's such that if $(C_i, C_j) \in R$, then $Q_i \cap P_j \neq \phi$ or $D_i \cap D_j \neq \phi$. That is, if $q \in Q_i$ and $q \in P_j$, then components C_i and C_j are connected through the control state variable q , or if $d_{mn} \in D_i$ and $d_{mn} \in D_j$, then components C_i and C_j are connected through the shared data object d_{mn} . Thus, the control and data interconnection relations can be defined as follows:

$$R_{1,ij} = \{(C_i, q_k, C_j) | q_k \in Q_i \cap P_j\}$$

and

$$R_{2,ij} = \{(C_i, d_{mn}, C_j) | d_{mn} \in D_i \cap D_j\}.$$

Now, $R = (R_1, R_2)$, where $R_1 =$ union over $R_{1,ij}$ and $R_2 =$ union over $R_{2,ij}; i = 1, 2, \dots$ and $j = 1, 2, \dots$.

System Structure

With respect to all the previous definitions, a distributed software system can be completely represented by the 7-tuple

$$SYSTEM = (S, T, D, C, R, M_0, F)$$

where

- S : set of control state variables
- T : set of abstract data types
- D : set of data objects
- C : set of components
- R : interconnection relation
- M_0 : initial marking function (or initial marking m_0)
- F : final control state variables ($F \subseteq$ powerset of S).

With respect to ordinary Petri nets, S corresponds to the set of places, C to the set of nonprimitive transitions, R to the set of arcs, and M_0 to the initial marking function. The dynamic behavior of the system is controlled by the initial control state, the initial data state, and the individual transfer functions of the system components.

The 7-tuple SYSTEM determines the global system structure and dynamic behavior from the point of view of a global observer. However, such a global system view can only be possible at the design phase, as opposed to the operational phase. Although complete or partial information on the system's structure and behavior can be gathered either by a central component or by any component in a distributed manner, such information usually reflects the status of the system considerably before the current time. This is due to the delay in system data communication. Such a view of system models is commensurate with large-scale distributed systems in which the exact current status of the system is no longer required to be known, and, in fact, may not be feasibly determined. An example system representation is shown in Fig. 2. The 7-tuple system representation will basically be used to represent the internal structure of components rather than the global system view.

Component Internal Structure

Central to the modeling of distributed software systems is the concept of a component. Components will externally be known by their external specifications. Internally, they will be regarded as subsystems with their own structure and dynamic behavior. Using the top-down design approach, the designer determines the internal structure and dynamic behavior of a component from its external specifications; and using the bottom-up approach, the designer determines the external specification of a component from its internal structure and dynamic behavior. Externally, a component starts execution as soon as a selected set of input control state variables is enabled. Then, the output control state variables and data objects of the component are modified as specified, and the execution of the component is completed.

The basic idea for the development of the component concept is to be able to leave the degree of concurrency to be provided at a level of system description undefined. Although the existence of concurrency can be determined by evaluating the input and output control specifications, the degree of

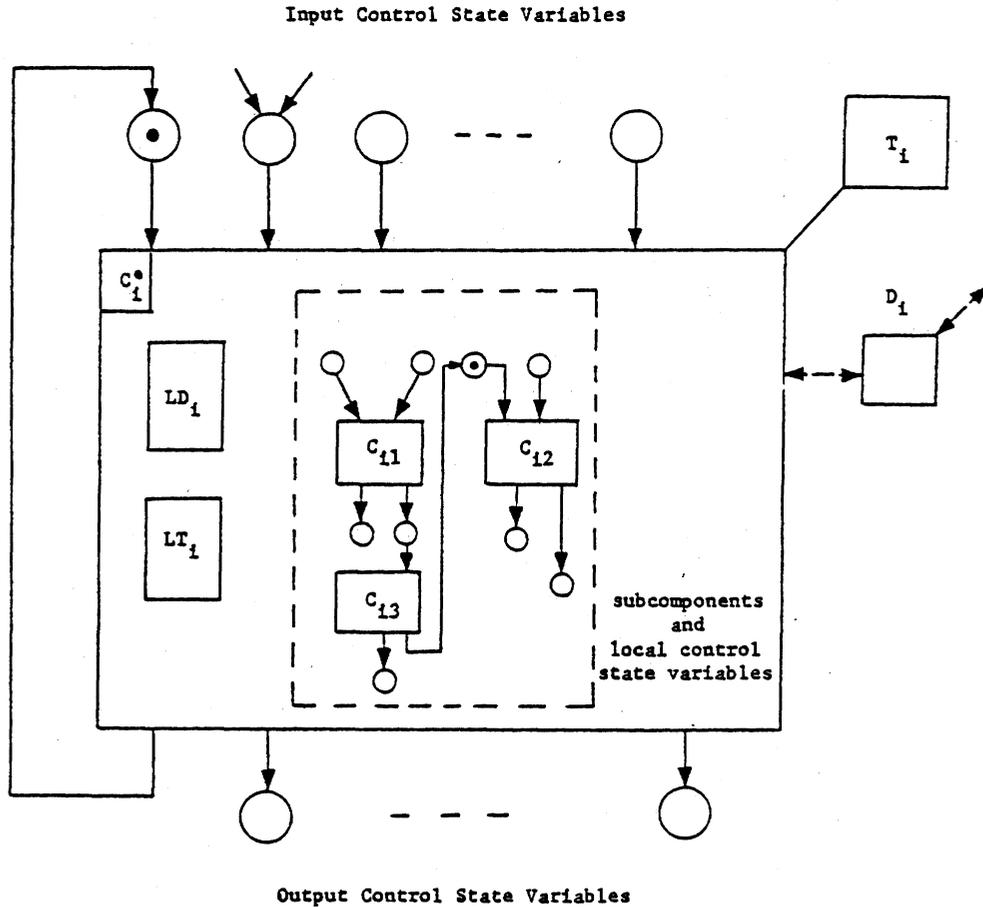


Fig. 5. The internal view of a software component.

concurrency can only be determined through the knowledge of the internal structure of the component. Similarly, from a design point of view, the degree of concurrency for a given design level can be decided when the component is designed, and that component's design can be used to design higher level components without requiring the knowledge of its internal structure.

For large-scale distributed software systems, a system specification at a certain level may result in complex descriptions of components. Therefore, components can be thought of as subsystems. Internally, a system component C_i , which is shown in Fig. 5, can be completely characterized as the subsystem

$$(P_i \cup Q_i \cup LS_i, T_i \cup LT_i, D_i \cup LD_i, \{C_{i1}, \dots, C_{in}\}, R_i, M_{0i}, F_i)$$

where P_i , Q_i , T_i , and D_i are as defined before. LS_i is the set of local control state variables, LT_i is the set of local abstract data types, LD_i is the set of local data objects, $\{C_{i1}, \dots, C_{in}\}$ is the set of subcomponents of C_i , R_i is the interconnection relation, M_{0i} is similar to M_0 , and F_i is similar to F . The number of subcomponents in C_i , number of local control state variables, number of data types and objects, interconnection relation, and subcomponent transfer functions will be determined by a proper functional decomposition of the control and data transfer specifications of C_i .

III. EXAMPLES

Now, we would like to use some examples in order to illustrate our design representation technique. Some common operations are graphically shown in Fig. 6. These are typically used in producing more detailed operations and are self-explanatory.

Fig. 7 illustrates the first two levels of the design representation for a simple producer-consumer system. One producer communicates through a shared buffer with one consumer. The producer can place information into a buffer if it is empty and the consumer can get information from the buffer if it is full. If the buffer is full, the producer waits until the buffer is emptied by the consumer and is informed by the consumer that it has done so. If the buffer is empty, the producer places information into the buffer; and if the consumer is waiting for the buffer to be filled, it informs the consumer that it has done so. The design shown in Fig. 7 is further decomposed into subcomponents; one of these subcomponents is illustrated in Fig. 8.

The data objects shared by the consumer and producer components are BUF, a buffer with capacity one, and the Boolean variables WP and WC used to represent the waiting states of the producer and the consumer. Note that the wait state of the producer (or the consumer) is represented by the data and control states of the producer (or consumer). The producer and consumer can manipulate these data objects only exter-

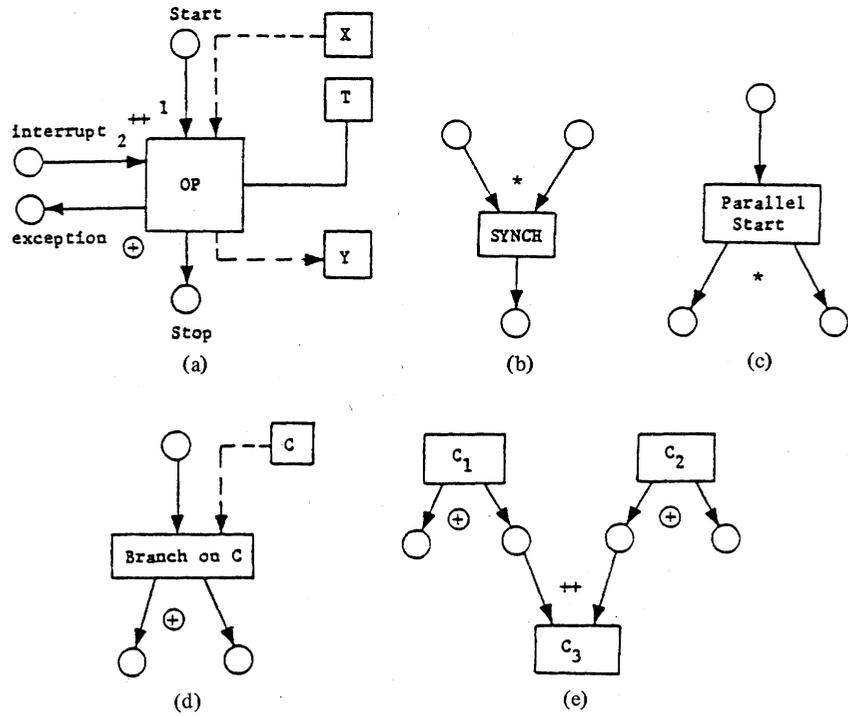


Fig. 6. The graphical representation of some common operations. (a) An operation with interrupt and exception conditions. (b) A "join" operation. (c) A "fork" operation. (d) A conditional branch. (e) Possible double execution of component C₃.

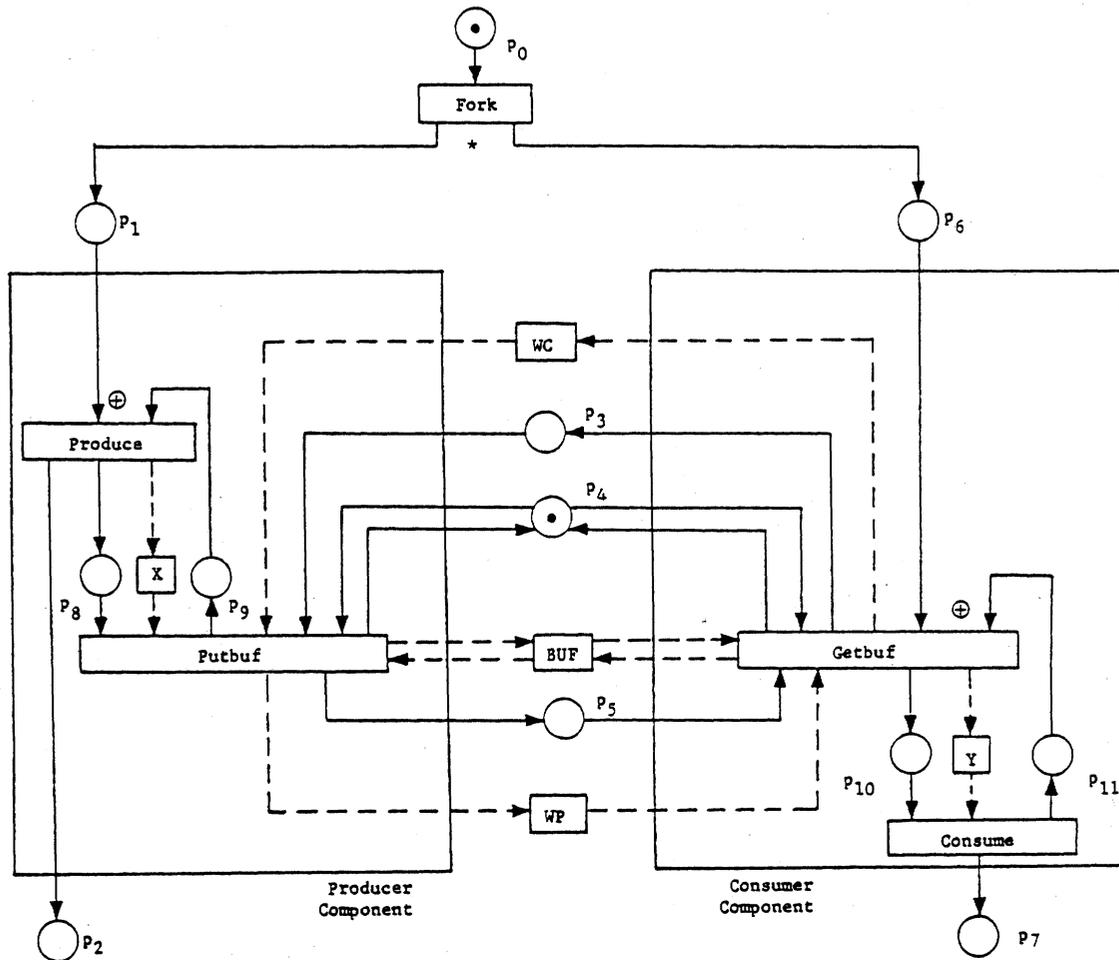


Fig. 7. The graphical representation of the first two design levels of a producer-consumer system.

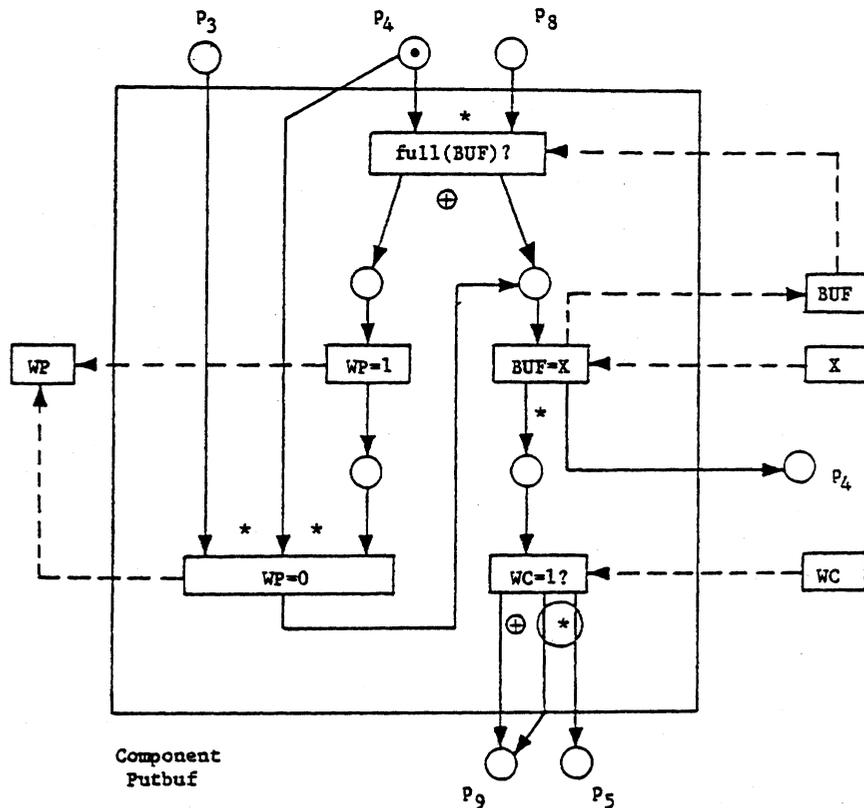


Fig. 8. A detailed design representation for the component "Putbuf."

nally, and only by the operations allowed by the types of these data objects. Types may be assumed as buffer for *BUF* and integer for *WP* and *WC*.

IV. DESIGN ANALYSIS

A given modified Petri net design representation can be analyzed by transforming the control flow aspects of the design representation into an equivalent Petri net and then analyzing that resultant Petri net. Given a modified Petri net as the representation of a distributed software system design, some of the analysis problems that we are interested in are the safety of the control states, the mutually exclusive execution of some software components, the proper termination of software components, and the deadlock-freeness of the components or the whole system.

A control state is said to be safe if its marking can only have a value of 0 or 1 during the course of the execution of the system. A system is safe if all of its control states are safe. Two or more software components are mutually exclusive if their execution durations do not overlap. A set of components may be required to be mutually exclusive if they update a shared data object in parallel. A modified Petri net component terminates properly if no local control state variable of the component remains enabled after the execution of the component is terminated. Deadlock-freeness (liveness) of a system that is represented by a modified Petri net must be categorized in terms of the levels of deadlock-freeness. An elementary component is deadlock-free at level 0 if it can never be executed during the execution of the system. Such an elementary component is called dead. Other levels can be

defined in a way similar to the way the liveness of a Petri net transition is defined in [20], [31]. The deadlock-freeness of the system can then be determined in terms of the deadlock-freeness of a single component at a certain level, or of a set of components, or of a set of components at different levels of deadlock-freeness. The same concept applies to the deadlock-freeness of a component if a component has a number of subcomponents.

Petri net representations of components with different input and output control transfer specifications are shown in Figs. 9 and 10, respectively. When a component is transformed into its Petri net representation, some number of redundant Petri net places and transitions may be generated. In addition, some of the transitions are timed-transitions [30], and inhibitor arcs may also exist in the equivalent Petri net representation.

There are two basic techniques for the analysis of Petri nets. One is the analysis using the reachability graph (also called a reachability tree), and the other is the analysis by linear algebra. Advantages and disadvantages of both techniques with respect to their capabilities to solve the Petri net problems are reviewed in [20]. The reachability graph is a finite, compact representation of the reachability set and the transition firings of a Petri net. The reachability set is the set of all markings that are reachable from an initial marking. A marking m_1 is reachable from a marking m_2 if m_1 can be obtained from m_2 by firing a finite number of transitions. An algorithm to construct the reachability graph of a Petri net is given in [20]. The linear algebraic techniques use two matrices to represent a Petri net graph and a vector for the initial marking. Then, the net can be analyzed by solving matrix equations that in-

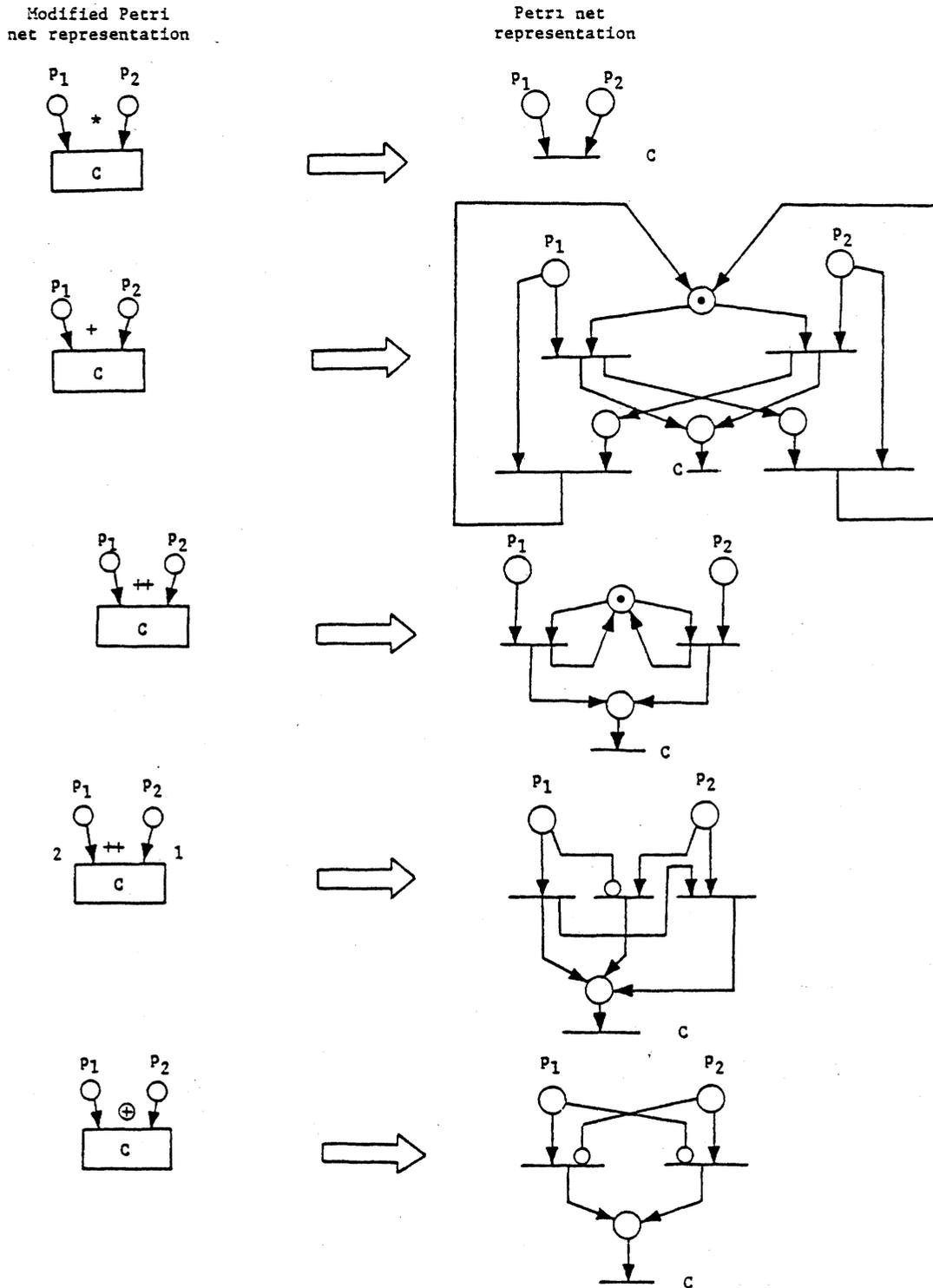


Fig. 9. Petri net representation of simple input control transfer specifications in modified Petri nets.

volve these matrices [20]. Since the matrix representation cannot represent self-loops in a Petri net, the analysis by the reachability graph technique is preferred.

The property of a modified Petri net first to be analyzed must be the safety of all control states. Since a control state is either enabled or disabled, it is sufficient to represent a control state s_i by the marking $M(s_i) = 1$. Then the markings for which $M(s_i) > 1$, i.e., a control state being enabled more than once simultaneously, are not meaningful and can be consid-

ered as design errors which must be detected during the analysis. If the Petri net that is being analyzed is an unbounded net (i.e., some place can have an infinite marking), analysis by the reachability graph cannot in general answer the deadlock-freeness question. Some other analysis problems are also suspected to be undecidable in this case. Timed-transitions, inhibitor arcs, and redundant places and transitions in the resultant Petri net complicate the generation and analysis of the reachability graph.

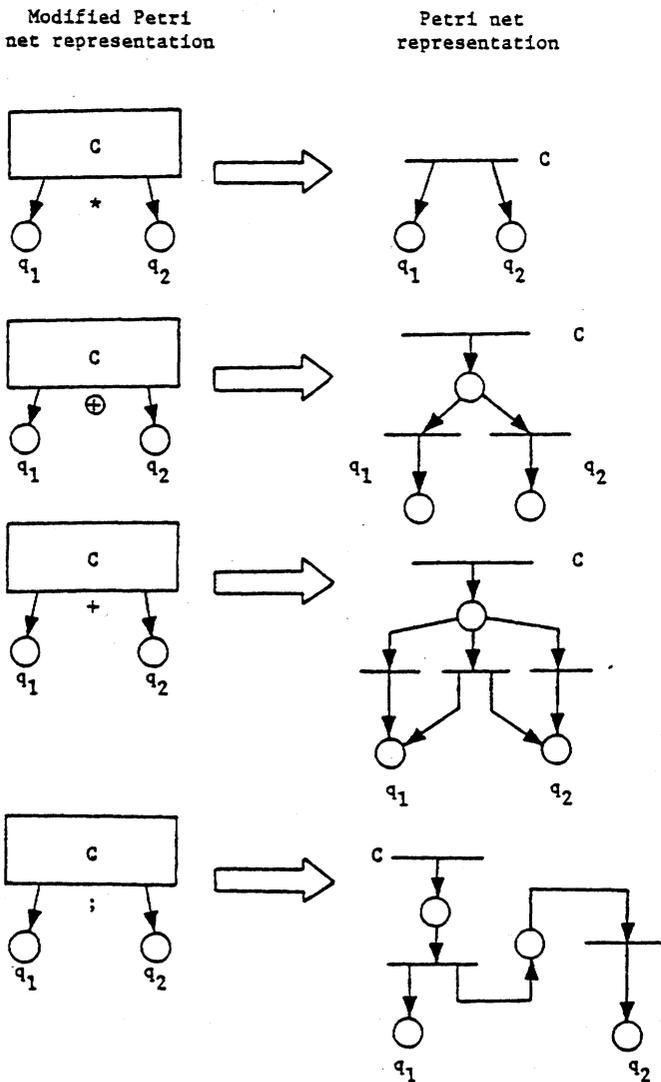


Fig. 10. Petri net representation of simple output control transfer specifications in modified Petri nets.

Currently we are developing some tools for performing the transformation of the design representations into Petri nets and for analyzing these resulting Petri nets. Using this technique, we are limited to performing control flow analysis of the type described previously. Thus, direct analysis of the modified Petri nets both from control and data flow points of view is also being studied.

V. DISCUSSION

A model for representing and analyzing the design of a distributed software system has been presented. The model is based on a modified form of Petri nets and enables one to represent both the structure and the behavior of a distributed software system. A representation in the form of a modified Petri net can be transformed into an ordinary Petri net which can be analyzed in order to verify certain behavioral properties.

Several modifications to ordinary Petri nets have been carried out in order to obtain the modified Petri net model. Petri net places, which are renamed as control state variables, are exclusively used to describe the control information in a design representation. Transitions are treated as executable units of

software and are called software components. The simple transition firing rule of Petri nets is generalized to input and output control transfer specifications. Modifications to the transition firing rule are sufficiently general so that both a large number of conditions under which a software component can start to be executed and the mechanism by which a software component interacts with its control environment can be specified in a compact form. Data objects which are connected to software components by directed arcs, and data types which are also connected to software components are added to a Petri net so that data aspects of a design representation can be described. A data transfer specification is associated with a software component to describe how a component modifies its data environment.

For the purpose of design analysis, structural and behavioral properties of a design representation given in the form of a modified Petri net are identified. The modified Petri net is first transformed into an equivalent ordinary Petri net, then that resulting Petri net is analysed in order to verify behavioral properties such as safeness, deadlock-freeness, mutual exclusion, and proper termination. Although certain analysis problems for Petri nets are known to be undecidable, the limitation of boundedness enforced through the transformation process results in the removal of any undecidable problems. Still, the exponential complexity of boundedness and liveness problems makes the analysis quite difficult.

More work needs to be done to determine whether the input and output control transfer specifications in their current form can completely specify all possible forms of the execution of components and all possible forms of output control state set modifications. For large-scale distributed software systems, some design experimentation is necessary to determine the type and amount of detail a final design level must include for the purpose of separating the design and programming issues. The modified Petri nets are currently unable to represent the performance and reliability constraints as part of the design representation. Also, the software is assumed to have a static structure. Therefore, there is a need to extend modified Petri nets so that additional constraints and the dynamic structure of distributed software systems can be represented during the design process. Such an extension of the representation technique will require the development of extensive analysis techniques for validating the performance and reliability constraints. Finally, the representation technique must be integrated into an overall design methodology. The methodology must address the critical issue of system partitioning and it must be compatible with the representation scheme.

ACKNOWLEDGMENT

The authors would like to express their thanks to S. Shatz of Northwestern University for many helpful discussions.

REFERENCES

- [1] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. Ass. Comput. Mach.*, vol. 15, no. 12, pp. 1053-1058, 1972.
- [2] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming*. New York: Academic, 1972.

- [3] W. P. Stevens, G. F. Myers, and L. C. Constantine, "Structured design," *IBM Syst. J.*, vol. 13, no. 2, pp. 115-139, 1974.
- [4] E. Yourdon and L. L. Constantine, *Structured Design*. New York: Yourdon, Inc., 1975.
- [5] M. A. Jackson, *Principles of Program Design*. London: Academic, 1975.
- [6] P. Brinch-Hansen, "Distributed processes: A concurrent programming concept," *Commun. Ass. Comput. Mach.*, vol. 21, no. 11, pp. 934-941, Nov. 1978.
- [7] C. A. R. Hoare, "Communicating sequential processes," *Commun. Ass. Comput. Mach.*, vol. 21, no. 8, pp. 666-677, Aug. 1978.
- [8] — "Monitors: An operating system structuring concept," *Commun. Ass. Comput. Mach.*, vol. 17, no. 10, pp. 549-557, Oct. 1974.
- [9] O. J. Dahl, B. Myrhaug, and K. Nygaard, *Simula 67-Common Base Language*. Oslo: Norwegian Comput. Cen., May 1968.
- [10] C. Hewitt and R. Atkinson, "Parallelism and synchronization in actor systems," in *Proc. 4th ACM Symp. Principles of Programming Languages*, Jan. 1977, pp. 267-280.
- [11] R. Milner, *A Calculus of Communicating Systems (Lecture Notes in Comput. Sci.)*, vol. 92. New York: Springer-Verlag, 1980.
- [12] G. Kahn, "The semantics of a simple language for parallel programming," in *Proc. IFIP Congr. 74*, Aug. 1974, pp. 471-475.
- [13] D. B. MacQueen, "Models for distributed computing," IRIA Tech. Rep. 351, Apr. 1979.
- [14] C. A. Petri, "Kommunikation mit Automaten," Ph.D. dissertation (in German); trans. by C. F. Greene, Suppl. 1 to RADC-TR-65-337, vol. 1, Rome Air Develop. Cen., Griffiss AFB, NY, 1965.
- [15] A. W. Holt *et al.*, "Final report of the information system theory project," RADC-TR-68-305, Rome Air Develop. Cen., Griffiss AFB, NY, Sept. 1968.
- [16] A. W. Holt, and F. Commoner, "Events and conditions," in *Rec. Project MAC Conf. on Concurrent Syst. and Parallel Comput.*, 1970, pp. 3-52.
- [17] J. B. Dennis, "Modular asynchronous control structures for a high performance processor," in *Rec. Project MAC Conf. on Concurrent Syst. and Parallel Comput.*, 1970, pp. 55-80.
- [18] W. Brauer, *Net Theory and Applications (Lecture Notes in Comput. Sci.)*, vol. 84. New York: Springer-Verlag, 1980.
- [19] J. L. Peterson, "Petri nets," *ACM Comput. Surveys*, vol. 9, pp. 223-252, Sept. 1977.
- [20] —, *Petri Net Theory and the Modeling of Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1981.
- [21] S. S. Yau, C. C. Yang, and S. M. Shatz, "An approach to distributed computing system software design," *IEEE Trans. Software Eng.*, vol. SE-7, pp. 427-436, July 1981.
- [22] L. J. Mekly, and S. S. Yau, "Software design representation using abstract process networks," *IEEE Trans. Software Eng.*, vol. SE-6, pp. 420-435, Sept. 1980.
- [23] P. M. Lu, and S. S. Yau, "A methodology for representing the formal specification of distributed computing system software design," in *Proc. 1st Int. Conf. Distributed Comput. Syst.*, Oct. 1979, pp. 212-221.
- [24] S. S. Yau, and S. M. Shatz, "On communication in the design of software components of distributed computer systems," in *Proc. 3rd Int. Conf. Distributed Comput. Syst.*, Oct. 1982, pp. 280-287.
- [25] I. M. Campos, and G. Estrin, "Concurrent software system design," in *Proc. 3rd Int. Conf. Software Eng.*, May 1978, pp. 230-242.
- [26] P. E. Lauer, P. R. Torrigiani, and M. W. Shields, "COSY—A system specification language based on paths and processes," *Acta Informatica*, vol. 12, pp. 109-158, 1979.
- [27] P. E. Lauer, M. W. Shields, and E. Best, "Design and analysis of highly parallel and distributed systems," in *Abstract Software Specifications (Lecture Notes in Comput. Sci.)*, vol. 86, D. Bjorner, Ed. New York: Springer-Verlag, 1980, pp. 451-503.
- [28] T. Agerwala, "An analysis of controlling agents for asynchronous processes," *Dep. Comput. Sci., Johns Hopkins Univ.*, Rep. TR-35, Aug. 1974.
- [29] S. S. Patil, "Coordination of asynchronous events," Ph.D. dissertation, TR-72, Project MAC, MIT, Rep. TR-72, June 1970.
- [30] C. Ramchandani, "Analysis of asynchronous systems by Petri nets," Ph.D. dissertation, Project MAC, MIT, Rep. TR-120, 1973.
- [31] F. Commoner *et al.*, "Marked directed graphs," *J. Comput. Syst. Sci.*, vol. 5, pp. 511-523, Oct. 1971.
- [32] G. J. Nutt, "Evaluation nets for computer system performance analysis," in *Proc. 1972 Fall Joint Comput. Conf.*, vol. 41, pp. 279-236.
- [33] J. D. Noe, and G. J. Nutt, "Macro E-nets for representation of parallel systems," *IEEE Trans. Comput.*, vol. C-22, pp. 718-727, Aug. 1973.
- [34] J. D. Noe, "Hierarchical modeling with pro-nets," in *Proc. Nat. Electron. Conf.*, vol. 32, Oct. 1978, pp. 155-160.
- [35] W. Tractnig and H. Kerner, "EDDA—A very high-level programming and specification language in the style of SADT," in *Proc. Compsac 80*, Oct. 1980, pp. 436-443.
- [36] R. Valette, and M. Diaz, "Top-down formal specification and verification of parallel control systems," *Digital Processes*, vol. 4, pp. 181-199, 1978.
- [37] C. A. Petri, "Interpretations of net theory," GMD-ISF Rep. 75-07, July 1975.
- [38] J. V. Guttag and J. J. Horning, "The algebraic specification of abstract data types," *Acta Informatica*, vol. 10, pp. 27-52, 1978.
- [39] J. Guttag, "Notes on type abstraction (version 2)," *IEEE Trans. Software Eng.*, vol. SE-6, pp. 13-24, Jan. 1980.
- [40] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Commun. Ass. Comput. Mach.*, vol. 18, pp. 453-457, Aug. 1978.



Stephen S. Yau (S'60-M'61-SM'68-F'73) received the B.S. degree from the National Taiwan University, Taipei, Taiwan, China, in 1958, and the M.S. and Ph.D. degrees from the University of Illinois, Urbana, in 1959 and 1961, respectively, all in electrical engineering.

He joined the faculty of the Department of Electrical Engineering, Northwestern University, Evanston, IL, in 1961, and is now Professor and Chairman of the Department of Electrical Engineering and Computer Science. He is currently

interested in reliability and maintainability of computing systems, software engineering, and distributed computer systems. He has published numerous technical papers in these and other areas.

Dr. Yau is a Fellow of the Franklin Institute from which he received the Louis E. Levy Medal in 1963; he is also a Fellow of the American Association for the Advancement of Science. He received the Golden Plate Award of the American Academy of Achievement in 1964, and the first Richard E. Merwin Award of IEEE Computer Society in 1981. He was the President of the IEEE Computer Society in 1974-1975, the Division V (Computer Society) Director of the IEEE in 1976-1977, and the Chairman of the IEEE Technical Activities Board Development Committees in 1979. He has been a Director of the American Federation of Information Processing Societies (AFIPS) from 1972 to 1982 and is now the Vice President of AFIPS. He is also now the Editor-in-Chief of IEEE COMPUTER magazine. He was the Conference Chairman of the First Annual IEEE Computer Conference, Chicago, 1967, and the General Chairman of the 1974 National Computer Conference, Chicago; the General Chairman of the IEEE Computer Society's First International Computer Software and Applications Conference, Chicago, 1977 (COMPSAC '77), and the Chairman of National Computer Conference Board in 1982-1983. He is also a member of Association for Computing Machinery, Society for Industrial and Applied Mathematics, American Society for Engineering Education, Sigma Xi, Tau Beta Pi, and Eta Kappa Nu.

Mehmet U. Caglayan (S'76-M'82) was born in Ankara, Turkey, on July 21, 1951. He received the B.S. degree in electrical engineering and the M.S. degree in computer science from the Middle East Technical University, Ankara, Turkey, in 1973 and 1975, respectively, and the Ph.D. degree in computer science from Northwestern University, Evanston, IL, 1982.

He is now an Assistant Professor at the University of Petroleum and Minerals, Dhahran, Saudi Arabia.

Dr. Caglayan is a member of Turkish Society of Electrical Engineers and the Association for Computing Machinery.