NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, tests publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30.

Canada

A PETRI-NET-BASED GRAPHICAL SYSTEM FOR NETWORK PROTOCOL SYNTHESIS

by

Abderrazak Ghedamsi

B. Sc., University of Ottawa

Thesis

submitted to the School of Graduate Studies and Research

in partial fulfillment of the

requirements for the degree of

Master of Science

in

Computer Science

University of Ottawa

July 1987

ISBN 0-315-46772-X

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

To my parents Naceur and Mahbòuba

ABSTRACT

The production of error-free protocols is essential for network communications. Two main approaches exist for protocol design - analysis and synthesis. In this thesis, a Petri-net-based synthesizer called PNPS is proposed. Starting with a Petri net specification of the local protocol entity, PNPS creates the Petri net specification of its peer entity. In the process, the given Petri net representation is first converted to a matrix representation, so that most of the subsequent computations are mainly searching and duplication of the matrix elements. The last step of PNPS transforms the matrices created for the peer entity into a graphical representation. PNPS has been automated as a user-friendly, menu-driven graphical system called GSAPS on a MacIntosh microcomputer under the MacIntosh Pascal interpreter.

The generated protocol is guaranteed to be complete, bounded, live, deadlock-free and properly terminated, if the local entity satisfies the following properties: completeness, channel boundedness, liveness, absence of undesirable final states, absence of cycles of send transitions, absence of cycles of receive transitions and proper termination.

As illustrations, GSAPS is applied on three protocols: the Packet Radio Network Protocol, the Alternating Bit Protocol and the Session Establishment and Clearing Phase of the Session Protocol S.62. All the experimental results conform with specifications or data available in the literature.

ACKNOWLEDGEMENT

I would like to express my deepest appreciation to my thesis supervisor Professor Dr. T.Y. Cheung for his constant advice, support, encouragement and patience throughout my graduate research. He always found time for productive discussions concerning my work.

The invaluable assistance from the staff of the Department of Computer Science is also appreciated.

The financial support from the Tunisian Government is acknowledged with gratitude.

I am also indebted to the very special supports I received from my parents Mr. and Mrs. N. M. Ghedamsi, and my special friend M. Filion.

TABLE OF CONTENTS

List of Figures

List of Tables

Chapter 1

INTRODUCTION AND FUNDAMENTALS

1.1 Protocol Specification, Verification and Testing

A protocol is a set of rules governing the exchange of information between systems in a distributed environment. The complexity of these systems makes it desirable that the functions of a protocol be decomposed into layers. The International Organization for Standardization (ISO) Reference Model has developed an architecture called Open System Interconnection (OSI) [ZIMM 80]. This architecture classifies the functions of protocols into seven layers. Each layer provides a number of services to the layer above it. Within each layer, peer entities in the system interact so as to provide the services.

In order to describe clearly and precisely the services and protocols, formal specification techniques should be used. A protocol specification is a description of the interactions between the entities to provide the services. A service specification is a description of the behavior of a layer in terms of its input/output primitives. A variety of methods based on different models have been proposed in the literature for such purposes. These methods may be transition-oriented, such as finite state machines [BOCH 78, DANT 77a] and Petri nets [MERL 79]; programming language based [STEN 76]; formal languages based [HARA 79]; and temporal logic based [SCHW 82]. Some models used for specification are behavioral, such

as the calculus of communicating systems [MILN 80], communicating sequential processes [HOAR 78], and language for temporal ordering by specifications (LOTOS [ISO 85]).

Protocol development starts with a design and ends up with an implementation. Two of the main approaches for protocol design are analysis and synthesis. Analysis is mainly a detective approach, in which a draft design is first obtained by whatever means and a verification method is then applied to check whether the draft has any errors or not. If so, the draft will be modified and then reanalysed. A modification is usually based on human experience with the protocols. Two very popular methods for verification are reachability analysis and structural analysis. In the former, all the states are traversed so as to determine whether there are such errors as deadlocks, infinite loops, etc [VUON 86]. In the latter, the structure of a protocol is analysed using such techniques as place invariants and transition invariants [CHEU 84]. Synthesis is mainly a constructive approach, in which a protocol is constructed from a partially complete draft by using a certain method. For instance, in the case of a protocol with two communicating entities, only one of them is first designed and a semi-automated algorithm is used to produce the other, so that the pair will form an error-free protocol. Several methods used in the synthesis approach are reviewed in Chapter 2.

The synthesis approach for protocol design is based on the following fundamental principle of interactions between entities.

Principle of Compatible Interaction Sequences

An interaction sequence is a mixed sequence of transmission and reception transitions. An interaction sequence from one entity is said to be compatible with an interaction sequence from another entity if the following criteria are satisfied:

a) Completeness: All messages transmitted in one sequence are received in the other sequence. Hence, at the end of the interactions, the communication links must be free of the messages involved.
b) FIFO: Messages are received in the same order of their transmissions.
c) Transmit-before-receive: Messages received in one sequence must have been transmitted no later in the other sequence.

Figure 1.1 shows two compatible interaction sequences. The messages b and e transmitted in sequence S1 are all received later in sequence S2. Similarly, the messages a, c and d transmitted in sequence S2 are all received later in sequence S1. Hence, the two sequences satisfy the conditions of compatible interaction sequences.

A protocol implementation has to be checked as to whether it conforms to its protocol specification or not. This activity is called protocol conformance testing. A lot of research work has been directed towards such tests [BOCH 83]. Some investigations have focussed on developing test architectures [RAYN 82] and generating test sequences [SARI 82, URAL 84].

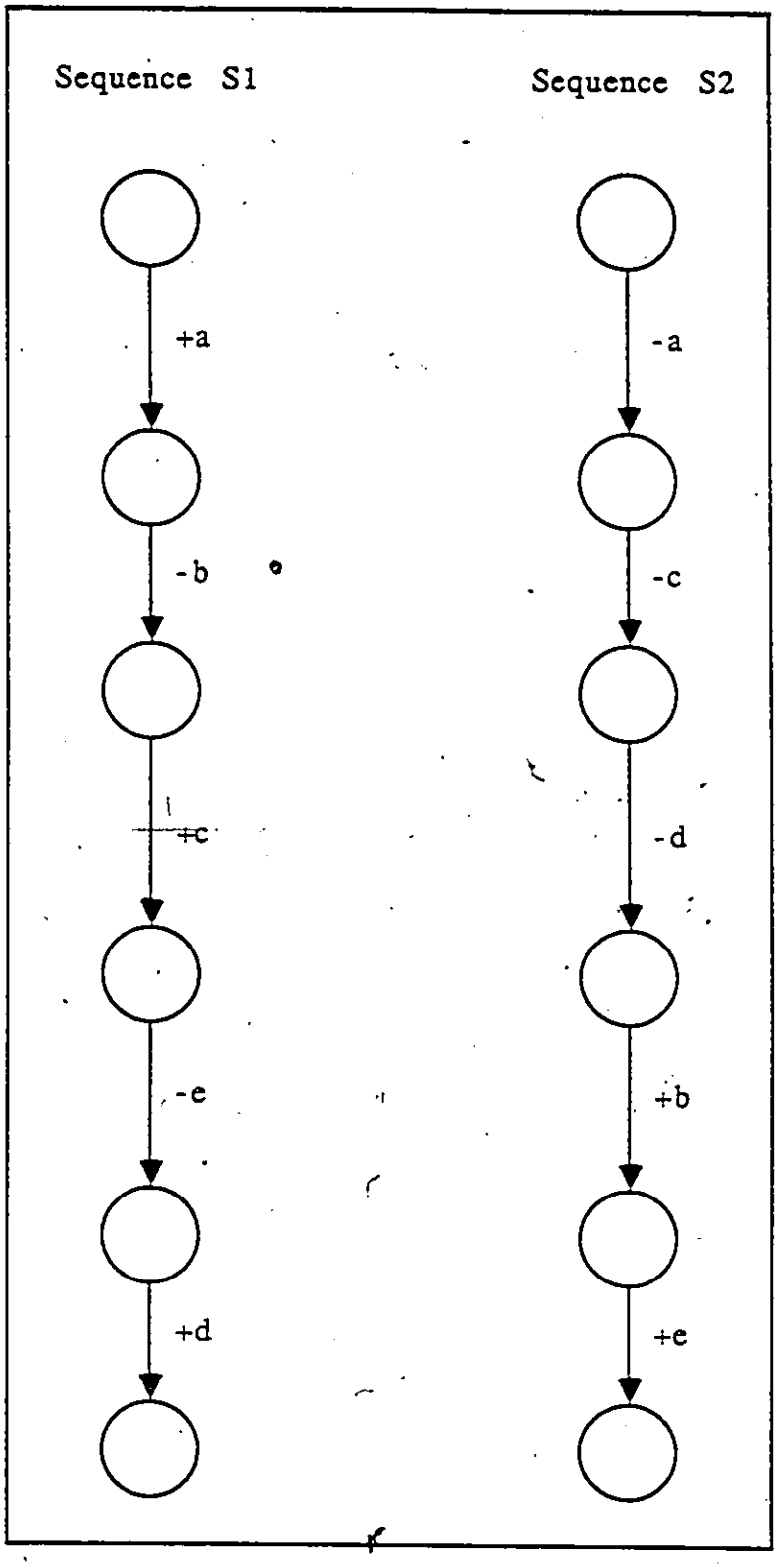


Figure 1.1 An example of two compatible interaction sequences

1.2 Graphical Models for Protocol specification and Verification

As mentioned in Section 1.1, many formal models have been proposed in the literature for the specification and verification of protocols. This section describes two formal graphical models which are the bases of the method used in this thesis, namely communicating finite state machines and Petri nets.

1.2.1 Communicating Finite State Machines

Figure 1.2 illustrates two communicating finite state machines (CFSM) [BRAN 83, RUDI 78, WEST 78]. Each process of the protocol is represented by a finite state machine whose state transitions are caused by the transmission or reception of messages. A message transmission is indicated by a minus "-" sign and a message reception is indicated by a plus "+" sign. These finite state machines communicate with each other through two channels, each represented by an FIFO queue in one direction. These queues are used to hold messages in transit. When a transmission (send) transition occurs, the corresponding message is inserted at the rear of the queue connecting the source machine to the destination machine. When a receive transition occurs, the message is removed from the front of the incoming queue.

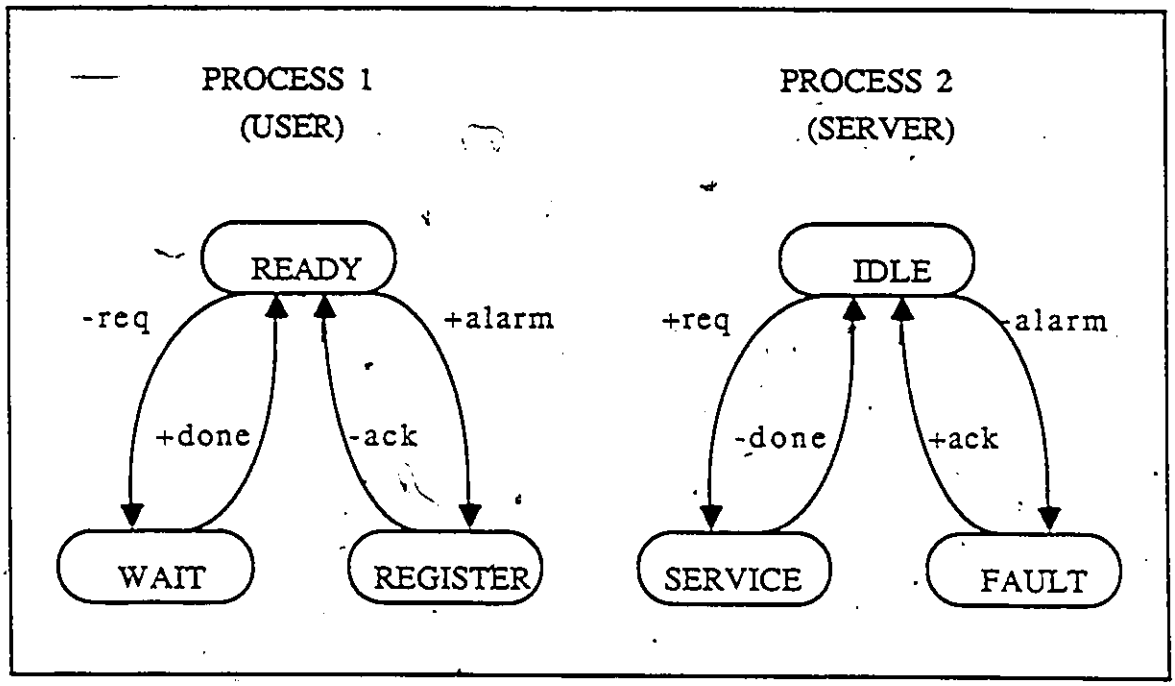


Figure 1.2 An example of two communicating finite state machines

The following formal definition of the CFSM model was introduced by Brand and Zafiropulo [BRAN 83].

A protocol P for N communicating processes can be specified as a quadruple:

$$P = << S_i >_{i=1,N}; < M_{ij} >_{i,j=1,N}; < o_i >_{i=1,N}; succ >$$

where,

- i, j, k are indices of the processes
- N is the number of processes
- S_i represents the set of states of Process i, i=1,...N
- M_{ij} represents the set of messages that can be sent from Process i to Process j, with M_{ii} being empty

- o_i, an element of S_i, represents the initial state of Process i,

- succ is a partial mapping function: $S_i \times (M_{ij} \cup M_{ji}) \longrightarrow S_i$

That is, succ(s,x) is the state reached by a process i after it receives or transmits message x in state s.

The transition is a send if x is from M_{ij}, and a receive if x is from M_{ji}, where $M_{ij} \cap M_{ji} = \phi$.

The definition of succ can be extended to a sequence X of messages: succ(s,ϕ) = s and succ(s,xX) = succ(succ(s,x),X), where ϕ is the empty message.

In reachability analysis, global states are generated from the initial global state by executing the local transitions one at a time. Some properties can be checked, such as absence of unspecified receptions, deadlocks, nonexecutable transitions and proper termination [ZAFI 80, SHER 82]. Several computer-aided systems have been developed for this purpose [ZAFI 80, VUON 81].

1.2.2 Petri Nets

A Petri net is an abstract and formal graph for expressing the flow and control of information in a system. It is particularly useful for those systems which exhibit nondeterministic, asynchronous and concurrent behaviors [DANT 77b, PETE 77, PETE 81].

A Petri net is a directed bipartite graph consisting of two types of nodes called places and transitions, usually denoted as circles and bars,

respectively.

Formally, a Petri net PN is a 5-tuple (P, T, M_0, I, O), where

P is a set of places;

T is a set of transitions;

M_0 is the initial marking;

I, O are mappings: T---> P^W (the power set of P) such that

I(t) is the set of input places of transition t; and

O(t) is the set of output places of transition t.

Figure 1.3 is an example of a Petri net specifying the stop-wait-acknowledgement communication protocol for two processes.

In a Petri net, places represent conditions and transitions represent events. Each transition represents an event with its associated input and output places representing the preconditions and postconditions, respectively. The fulfillment of a condition in a place is indicated by the existence of a token, represented as a dot in the place. The system status is represented by the pattern of tokens in the places and is called a marking. If M denotes a marking, M(p) denotes the number of tokens in place p. The initial marking denotes the initial system status.

A variation in specifying a protocol is **direct coupling**. In this strategy (Figure 1.4), the communication channels are not explicitly specified and each entity is represented by a distinct Petri net describing the behaviors of the elements inside. The interactions between local entities are not explicitly described, but can be easily perceived by looking at those places

Places: A - ready to send
B - buffer full
C - ready to receive
D - wait for ack.
E - message received
F - ack. received
G - buffer full
H - ack. sent

Transitions: t1 - send message
t2 - receive message
t3 - receive ack.
t4 - send ack.
t5 - prepare to send
t6 - prepare to receive

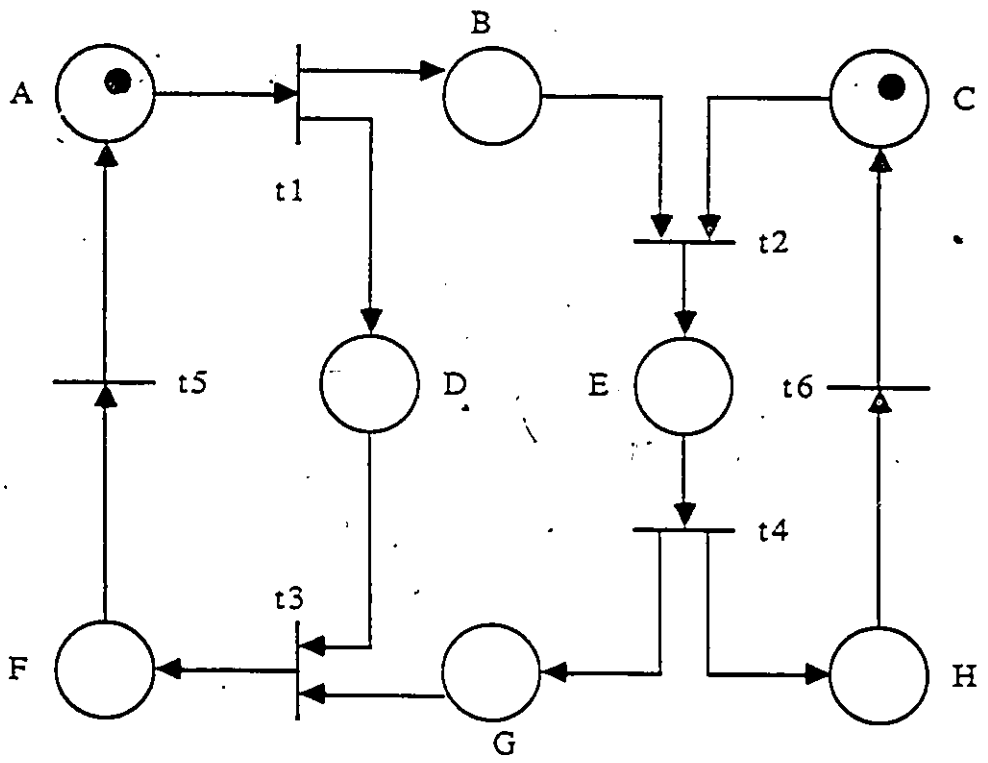


Figure 1.3 Petri net for the stop-wait-acknowledgement protocol

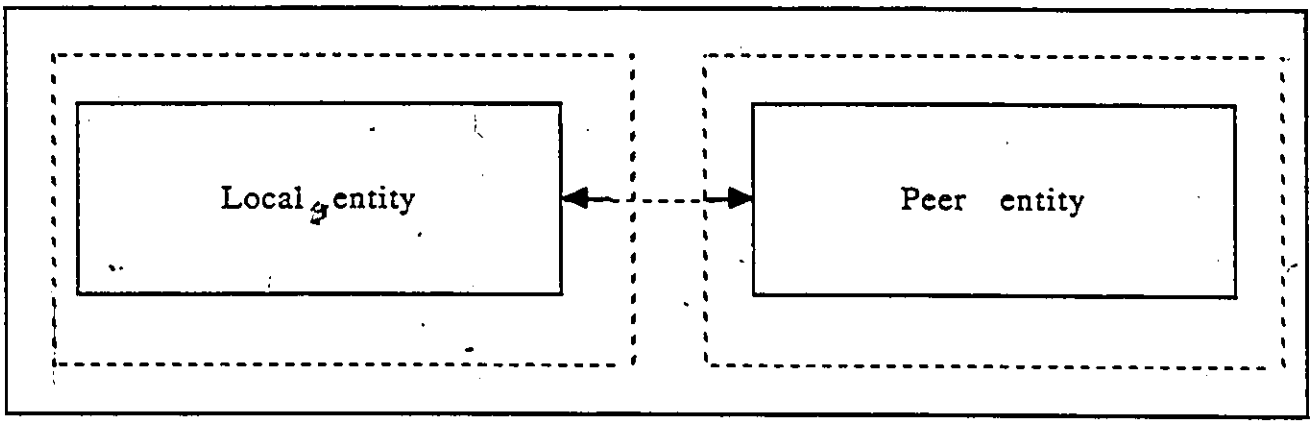


Figure 1.4 Direct coupling strategy (channels not included)

sharing common names in the individual local entities. As a result of the strategy of direct coupling, a local entity has three types of places: (1) internal places, (2) external input places and (3) external output places. Internal places represent the internal status of a local entity. In order to communicate with other entities, a number of external input or output places are defined in each entity. Deposit of a token into an external output place indicates the sending of a message and removal of a token from an external input place indicates the reception of a message.

One of the advantages in using direct coupling is that the local entities can be directly implemented without the problem of inconsistent or incompatible decomposition of the same protocol [WEST 78] as in global modeling.

1.3 Logical Properties of a Protocol in Communicating FSM Model

The following definitions are used to describe the logical properties of a

protocol.

Definitions

Global state: A global state is a pair <S, C>, where S is a composite state, i.e., an N-tuple $<s_1, s_2, ..., s_N>$ with s_i representing the state of Process i, and C is an N^2-tuple $<c_{11},...,c_{1N},c_{21},...,c_{NN}>$, with each c_{ij} representing the contents of the channel from Process i to Process j.

Stable state: A stable global state, denoted as an N-tuple $<s_1,s_2,...,s_N>$, is one in which all channels are empty. A local state s_i of a global stable state S can also be referred to as a stable local state.

The 'reachability' relation |-- : A binary relation |-- is defined on the set of global states as follows:

<S, C> |-- <S', C'> iff all the elements of <S, C> and <S', C'> are equal except that there exist i, k, x_{ik} satisfying one of the following conditions:

i) $s'_i = succ(s_i,-x_{ik})$ and $c'_{ik} = c_{ik}.x_{ik}$

ii) $s'_k = succ(s_k,+x_{ik})$ and $c_{ik} = x_{ik}.c'_{ik}$

Reachable global state: A global state <S, C> is reachable iff $<S_0, C_0>$ |--* <S, C>, where |--* denotes the reflexive and transitive closure of |-- and $<S_0,$

C_0> = <<o_i>$_{i=1,N}$; < >$_{i,j=1,N}$>, i.e., every process is in its initial state o_i and all channels are empty.

Specified reception: The reception of message x at state s is said to be specified iff succ(s,+x) is defined.

Executable reception: The reception of message x is said to be executable at a local state s iff there exists a reachable global state <S, C> such that for some i and k, $s = s_k$ and $c_{ik} = xY$ for some sequence Y.

There are two classes of properties a protocol has to fulfill. One class is specific to an individual protocol, such as the quality of service of a connection request of the Class 4 Transport Protocol. To verify this class of properties, different techniques have to be used for different protocols. Another class of properties is common to all protocols. They can be verified by some general methods, such as reachability analysis. Most of these properties are logical. The common properties are defined below:

i) **Unspecified reception:** The reception of a message x is unspecified iff it is executable but not specified.

ii) **Nonexecutable reception:** The reception of a message x is nonexecutable iff it is specified but not executable.

iii) **Global state ambiguity:** State ambiguity means that the local state of one entity can coexist stably with several different states of another entity.

iv) **Deadlock state:** A deadlock state is a reachable non-final stable state S such that there are no i and x for which succ(s_i,-x) is defined.

v) **Livelock:** A protocol is in a livelock if it is in an infinite cycle accomplishing no useful work.

vi) **Completeness:** A protocol is said to be complete if there are no unspecified receptions in any of its entities.

vii) **Proper termination:** A system can terminate properly if every reachable state can reach at least one of the final states.

viii) **Bounded channel:** A communication channel is said to be bounded if it can contain only a fixed number of messages.

1.4 Motivation and Outline of the Thesis

A protocol is an important component of a communication network. Its correctness is crucial for the performance, reliability and availability of a network. In the past, errors and undesirable behaviors have been found in the design of many protocols. For example, 29 logical errors were found in

X.21 [WEST 78]. This is partly due to lack of formal design techniques . In the synthesis approach, a set of rules or a set of necessary and sufficient conditions is used to guide the design process. The resulting protocol will possess some desirable properties. This may shorten the development period of a protocol.

In this thesis, we propose a Petri-Net-based Protocol Synthesizer called PNPS and develop a system which is computationally powerful but also user-friendly. Starting with a Petri net specification of the local protocol entity, PNPS creates the Petri net specification of its peer entity. In the process, the given Petri net representation is first converted to a matrix representation, so that most of the subsequent computation is simply searching and duplication of the matrix elements. The last step of PNPS transforms the matrices created for the peer entity into a graphical representation. If the local entity possesses certain desirable properties, such as absence of deadlocks, completeness, liveness and boundedness, the resulting protocol is guaranteed to be logically correct. PNPS has been automated as a user-friendly, menu-driven graphical system called GSAPS.

GSAPS is closely related to APS, an Automated Protocol Synthesizer proposed by Dong (Section 2.4). Dong's method also starts with the Petri net representation of the local entity and ends with the Petri net representation of the peer entity. But it first transforms the Petri net representation of the local entity to a finite state machine representation, then applies six rules to produce a finite state machine representation of the peer entity. Lastly, it transforms the finite state machine

representation of the peer entity to a Petri net representation.

Following is a list of contributions of this thesis.

1) Both systems start and end with Petri net representations of the entities. But, GSAPS needs fewer steps than APS, because the former does not have to go through the steps of first converting the given Petri net of the local entity to a finite state machine and then converting the finite state machine representation of the peer entity to a Petri net. Instead, GSAPS converts the Petri net graphical representation of the local entity to two local matrices and lastly converts the peer matrices to a Petri net graphical reprensentation.

2) By carefully analysing the six rules of APS, we have been able to extract and regroup their functions and condense the computations into two steps (Steps II and III).

3) In applying the six rules, APS has to go through the time-consuming process of searching the graph of the finite state machine representation of the local entity for detecting the required conditions. By classifying the functions of the generation process, GSAPS has been able to perform all the operations as two kinds of matrix computations. Hence, not only does it take advantage of the fast speed of matrix computations, it also reduces tremendously the programming effort of creating data structures and algorithms for graph storage and searching as in the case of APS.

4) GSAPS is graphical and is thus visually more appealing and operationally more user-friendly than APS. Data can be modified before being synthesized. Intermediate results can be obtained. APS is not

graphical.

5) In GSAPS, the matrix operations are hidden from the designer because they are less well understood. The designer sees only the graphical screen representation.

The rest of the thesis is organized as follows. Chapter 2 includes a survey on three methods for synthesizing protocols. The details of Synthesizer PNPS are given in Chapter 3. Chapter 4 describes GSAPS - an implementation of PNPS as a graphical user-friendly system. In Chapter 5, GSAPS is applied to three protocols, namely the Packet Radio Network Protocol, the Alternating Bit Protocol and the Session Establishment and Clearing Phase of the Session Protocol S.62. Some concluding remarks and suggestions for future research are given in Chapter 5. Appendix A provides detailed descriptions of the functions of GSAPS.

Chapter 2

REVIEW ON SYNTHESIS METHODS FOR PROTOCOL VALIDATION

2.1 Introduction

As described in Chapter 1, a method is needed in the synthesis approach for constructing the protocol under design. This chapter reviews three such methods, proposed by Zafiropulo [ZAFI 80], Sidhu [SIDH 82a, SIDH 82b] and Dong [RAMA 85], respectively. One of the differences in these methods is the amount of information known initially. This is summarised in Table 2.1.

Besides the above three methods, two other methods [GOUD 84, CHOI 86] have also been proposed in the literature. Gouda's method constructs two communicating finite state machines M' and N' from a given machine such that the communication between M' and N' is logically correct. The method proposed by Choi generates "protocol sequences" or "protocol expressions", depending on whether the protocol is acyclic or cyclic. At the last step, these sequences or expressions are converted to finite state machines representing the protocol entities.

	Zafiropulo	Sidhu	Dong
no. of entities	2	N (N >= 2)	2
states	completely specified in both entities	completely specified in all entities	completely specified in local entity
transmission transitions	completely specified in both entities	completely specified in all entities	completely specified in local entity
reception transitions	none specified in any entity	completely specified in all entities	completely specified in local entity

Table 2.1 The given data of three methods for protocol synthesis

2.2 The Method of Zafiropulo

2.2.1 Description of the method

Zafiropulo [ZAFI 80] presents an interactive mechanism based on the model of communicating finite state machines for protocol construction. Application of this mechanism is limited to protocols with two entities. It assumes that the transmission events are given in both entities and generates their possible receptions. It is based on a tracking algorithm which determines where and when to apply some rules during the process of construction.

The mechanism prevents the occurrences of unspecified receptions and notifies the designer of the presence of state deadlocks and ambiguities.

The tracking algorithm uses an incremental construction process and requires a protocol designer's intervention whenever a semantics-related action is required. At the beginning of the process, the algorithm creates nodes labeled with 0 for both entities. These nodes correspond to their initial states. Then, an interactive process starts. The algorithm waits for the designer to add the next message transmission in any of the two entities according to the specification. Then, it automatically generates the corresponding message receptions according to the production rules described below. The process stops when all the specified message transmissions have been added into the protocol.

Production rules

Definition: Two messages, one from each entity, are said to collide if one is being received while the other is still in the communication channel. Collisions are identified by subscripts. For example, if message y (respectively x) is still in the channel, the reception of x (respectively y) is denoted by $+x_y$ (respectively, $+y_x$).

In the following, P1 and P2 denote the two communicating processes and s and s' represent two sequences of message transmissions.

Rule 1 (Figure 2.1): If, in P2, -e is appended to the state reached by +x, then, in P1,

a) append +e to the state reached by -x; and

b) append $+e_s$ to every state reached by the sequence -xs .

Rule 2 (Figure 2.2): If, in P2, -e is appended to the state reached by -x, then, in P1,

a) append +e (respectively, $+e_s$) to every state reached by +x (respectively, $+x_s$).

b) append $+e_{s'}$ (respectively, $+e_{s,s'}$) to every state reached by the sequence -s' attached to the state reached by +x (respectively, $+x_s$).

Rule 3 (Figure 2.3): If in P2, -e is appended to the state reached by $+v_{...,u}$, then, in P1,

a) append +e to the state reached by $+u_{...,v}$ and $+e_s$ to every state reached by $+u_{...,v,s}$;

b) append $+e_{s'}$ (respectively, $+e_{s,s'}$) to every state reached by the sequence -s' attached to the state reached by $+u_{...,v}$ (respectively, $+u_{...,v,s}$).

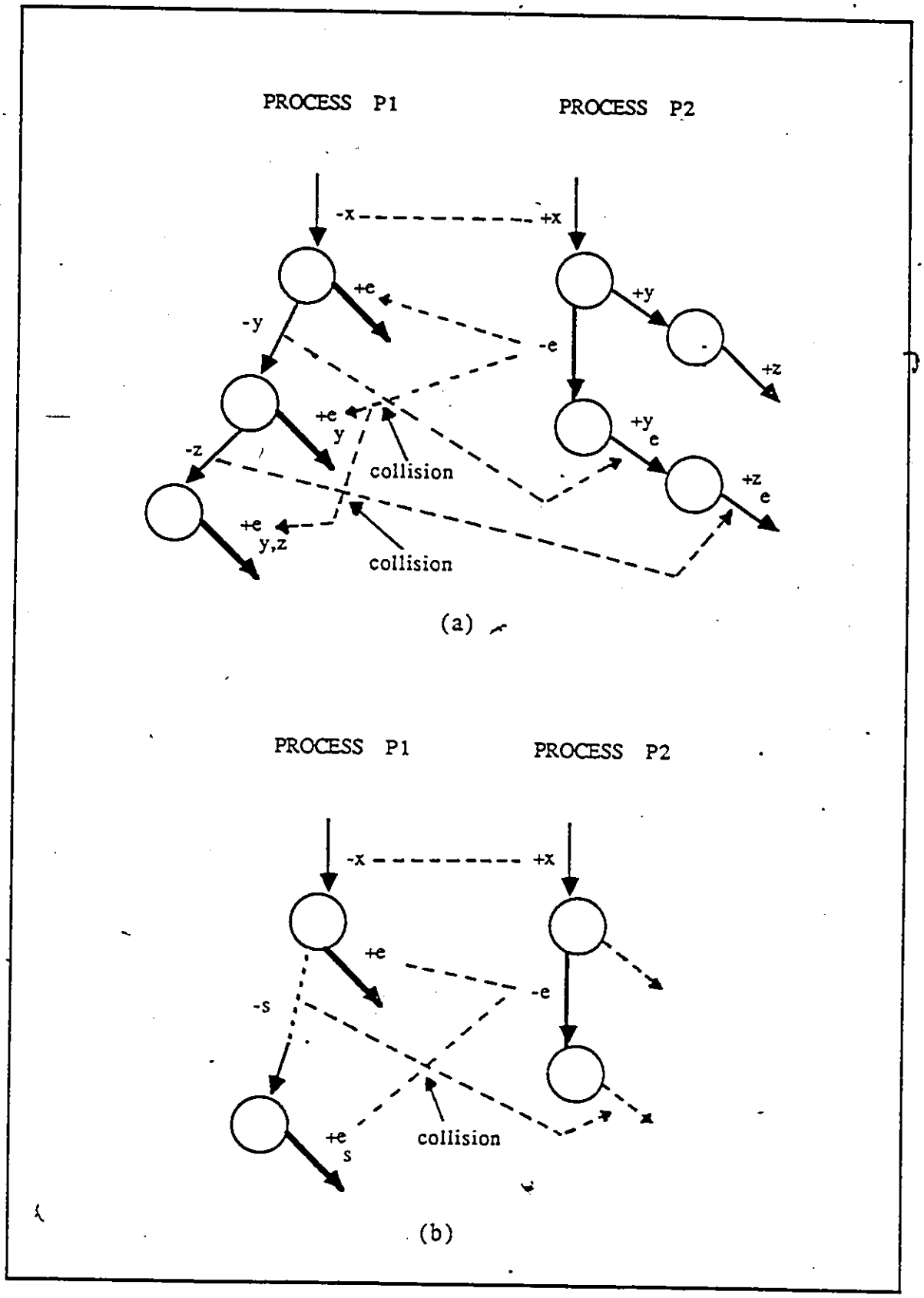


Figure 2.1 Production Rule 1 in the method of Zafiropulo.

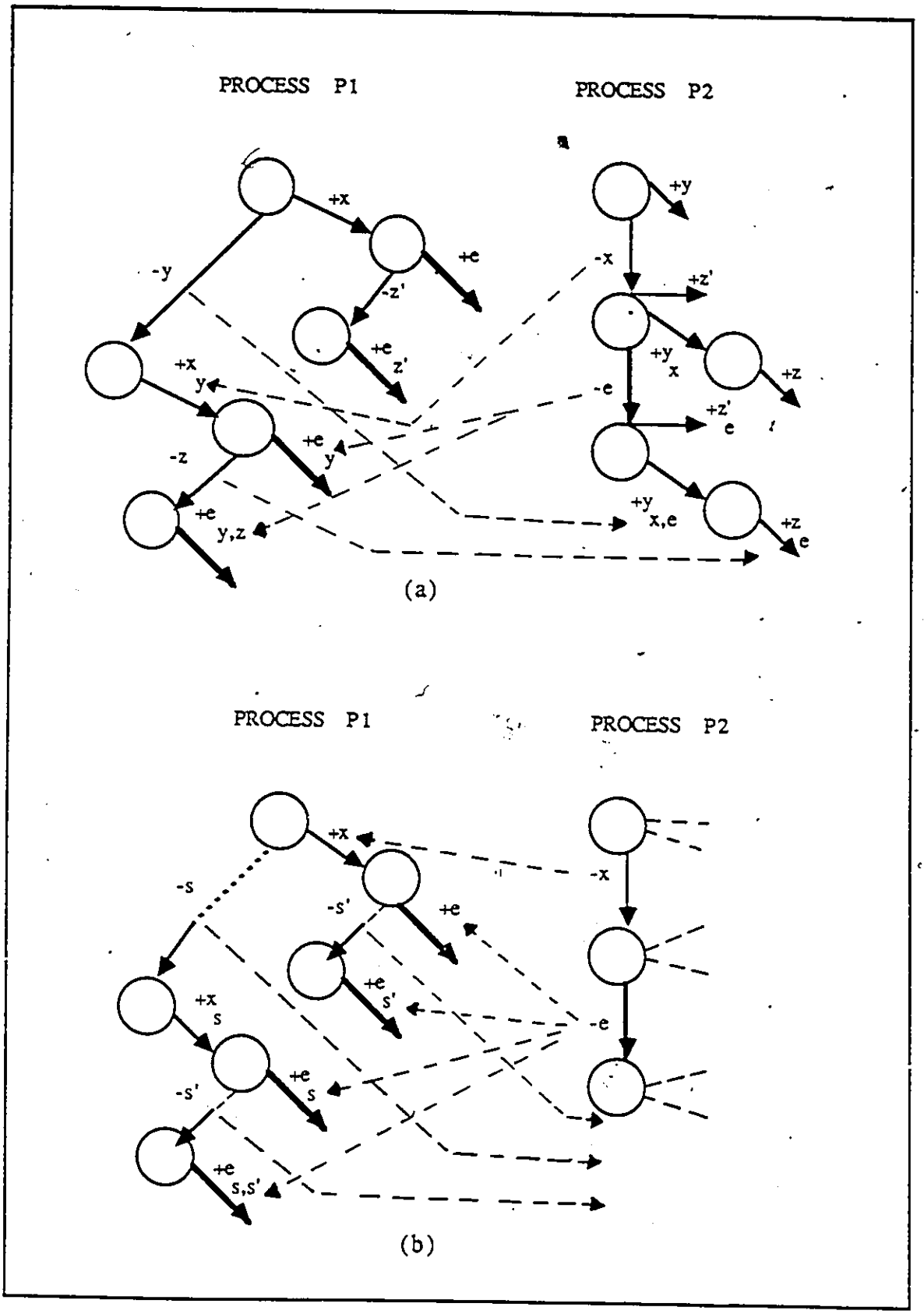


Figure 2.2. Production Rule 2 in the methdd of Zafiropulo

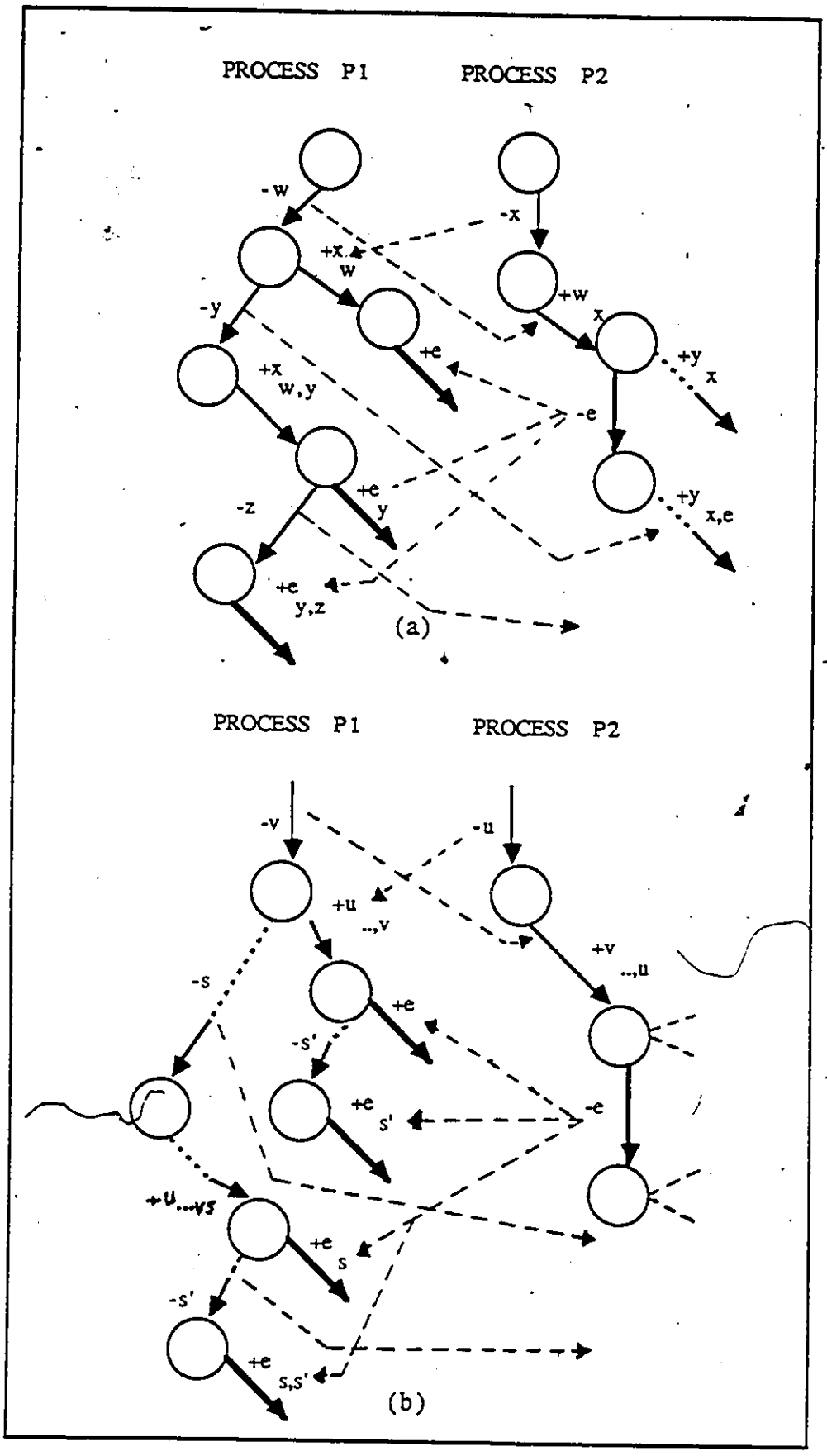


Figure 2.3 Production Rule 3 in the method of Zafiropulo.

2.2.2 Comments on the Method

The method of Zafiropulo, el al. has several defects:

i) Not all receptions of a specified message transmission are created simultaneously. Some of them may be added later because of the message transmissions newly added by the designer. This phenomenon may confuse the users and affect the correctness of the design.

ii) The complexity of the synthesis procedure depends greatly on the order of adding the message transmissions.

iii) The method does not provide any guideline for the designer to specify the state each transition will reach. If these states are not properly assigned, the resulting protocol may have deadlocks or endless loops. This means that the logical correctness of the resulting protocol is not guaranteed.

2.3 Sidhu's Method

2.3.1 Description of the method

Sidhu [SIDH 82a, SIDH 82b] considers N (N >= 2) processes with lossless, error-free and FIFO (or non-FIFO) channels. A protocol is first informally but completely specified as a collection of finite state machines. The method constructs the reachability tree for the global system and builds a directed graph for describing the activities of each protocol entity so that the protocol satisfies some specified properties. The method uses four design rules for constructing the protocol.

For the reachability tree, the global state is represented by an NxN matrix (Figure 2.4). The diagonal element a_{ii} defines the state of entity i. The off-diagonal element a_{ij} defines the state of the channel from entity i to entity j.

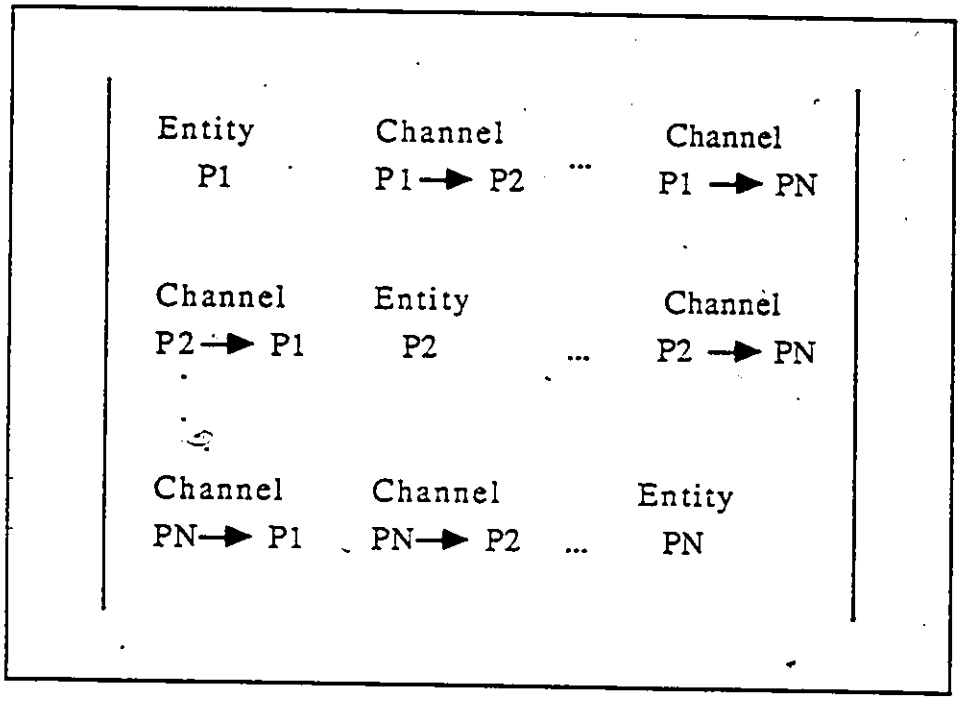


Figure 2.4 Representation of global states

The process starts by drawing N nodes, all labeled with 0, to denote the initial states s_0 of the N entities. These nodes represent the roots of the entity trees. The algorithm applies the design rules to generate new global state by perturbations. That is, at each time, only one local entity executes a transition. At the end of the algorithm, a tree-like structure (reachability

tree) is generated.

Protocol design rules

Rule 1: If entity P_i executes a transition from state s_1 to state s_2 on transmitting a message x to entity P_j, then, in the entity tree for P_i,

a) add a new state s_2 ,

b) draw a directed arc labeled with -x from state s_1 to state s_2,

c) add message x to the message queue in the channel P_i ---> P_j,

d) label the (i, i)$^{-th}$ element of the new system state with s_2, and

e) draw a directed arc labeled with -x from the old to the new global state.

Rule 2: If P_i executes a transition from state s_1 to state s_2 on receiving a message y from channel P_j ---> P_i, then, in the entity tree for P_i,

a) add a new state s_2,

b) draw a directed arc labeled with +y from state s_1 to state s_2,

c) delete message y from channel P_j ---> P_i,

d) label the (i, i)$^{-th}$ element of the new system state with s_2, and

e) draw a directed arc labeled with +y from the old to the new global state.

Rule 3: If the channel P_j ---> P_i contains messages $<x_1, x_2,...,x_n>$ (ordered from left to right according to their order of transmissions) at the time Rule 2 is applied, then,

a) if the channel is FIFO, Rule 2 is applied to message x_1 only,

b) if the channel is non-FIFO, Rule 2 is applied to each of the messages $x_1,...,x_n$.

Rule 4: If the protocol being synthesized satisfies the protocol properties listed in Table 2.2, each application of Rule 1, 2 or 3 must not violate the property maintenance conditions of Table 2.2.

2.3.2 Comments on the method

In Sidhu's method, the designer must specify all the interactions (i.e., both message transmissions and receptions) between the communicating entities. It includes four design rules. In fact, the fourth rule cannot really be considered as a rule because all it states is that the other three rules must not violate some protocol conditions. Strictly speaking, Sidhu's method is not a pure synthesizer. It combines both the analysis and synthesis approaches for protocol design.

Protocol property	Condition for ensuring property maintenance
Completeness	Make sure that a specified reception of a message x in an entity P, at a state s is indicated by a directed arc from that state in the entity tree
Freedom from deadlock.	Make sure that every stable state has at least one next transmission allowed by an entity (exept for the final state)
Freedom from livelock or tempo-blocking	Make sure that a new reception or transmission does not generate a global state that is one of the states on a path from the initial to the current state (exept for the initial state for for cyclic protocols)
Termination (Cyclic behavior)	Make sure that every interaction path starting from the global initial state and through a sequence of global states leads to the global final (initial) state.
Boundedness	Make sure that whenever a message is added to a channel, its total number of messages does not exceed a specified upper bound.
Liveness or absence of non-executable interactions	Make sure that every newly created global state is reached from a global state already generated.

Table 2.2 Conditions for maintaining properties in Sidhu's method

2.4 Dong's Method (APS)

Dong proposes an automated protocol synthesizer (APS) [RAMA 82, DONG 83, RAMA 85] for designing protocols with two entities.

2.4.1 Description of the Method

In APS, the direct coupling strategy is used and each communicating entity is modeled by a separate Petri net. Each send-transition has exactly one external output place and each receive transition has exactly one external input place.

Assuming that a local entity is given, APS constructs the corresponding peer entity so that the resulting protocol (specified by both the local and peer entities) is complete, bounded, live, deadlock-free, and properly terminated. The method has the following five steps.

Step 1 specifies the local entity by a Petri net.

Step 2 transforms this Petri net into a state transition graph (STG1) by using a state exploration procedure.

Step 3 makes sure that the local entity satisfies some logical properties by examining the structure of STG1.

Step 4 constructs the state transition graph (STG2) for the peer entity from STG1 according to certain transformation rules (presented in the coming subsections).

Step 5 constructs the Petri net representation of the peer entity from STG2.

More details about these steps follow:

Validation of local properties:

The given local entity should be validated to ensure that the following properties are correct: local boundedness, local completeness, local liveness, no undesirable terminal states, no cycles of send transitions, no cycles of receive transitions, and every reachable state can reach at least one desirable terminal state. This process can be done by examining the structure of STG1.

Construction of the peer state transition graph

APS traverses STG1 in a certain order, such as depth-first-search or breadth-first-search, detects some target arcs (denoted by solid lines) in STG1 and generates one or two arcs (denoted by solid lines) in STG2 according to the Transformation Rules listed in Table 2.3. Dashed lines indicate the reference arcs which specify the conditions to be satisfied in order to apply the Transformation Rules. During the traversal, only one target arc is examined each time.

In STG1, there are two situations when a message, say x, is received.

a) If the channel from STG1 to STG2 is empty, the reception transition of x is called originating.

b) If there is at least one message in the channel from STG1 to STG2, we say the messages collide and the reception of x is called a subordinate transition. ".c" is appended to each transition of this kind.

a) Transformation rules

The transformation rules of Table 2.3 handle all the three types of transitions: send, originating receive and subordinate receive.

Transformation Rules 1 and 2 handle send transitions and originating receive transitions, respectively. The transitions generated in STG2 are just their "reverse" transitions.

Transformation Rules 3 to 6 deal with subordinate message receptions. The philosophy underlying these rules is that any transition sequence from state k to state j in STG2 should be compatible with the transition sequence from k to j in STG1.

In Rules 3 and 4, the subordinate receive transition +w.c follows a send transition -x. The difference between these two rules in that -x is not specified at state 1 for Rule 3, but it is specified at state 1 for Rule 4. In both cases, two possible sequences, (-w, +x) and (+x, -w) from state k to state j can exist in STG2. Both sequences are compatible with the transition sequence (-x, +w.c) from state k to state j in STG1. Hence, both transition sequences are incorporated into STG2 and end at state j. In Rule 4, state h in STG2 is augmented by state j and let all the circumstances applicable to j or h be also applicable to the combined state of j&h. The transition from state 1 to state j&h in STG2 is set to be +x.c instead of +x as obtained from Rule 1.

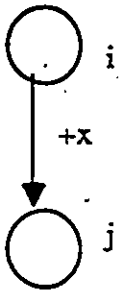
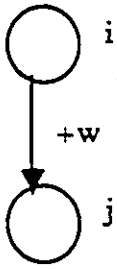
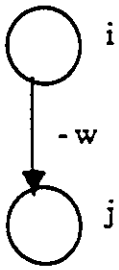
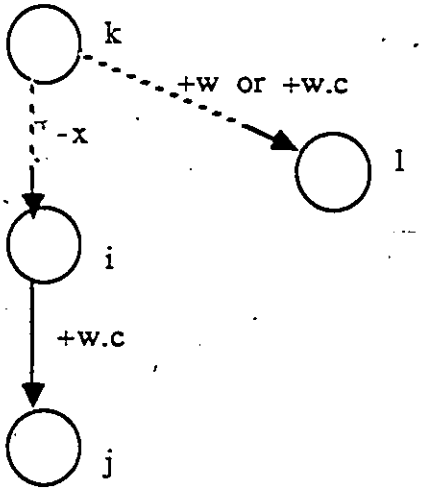
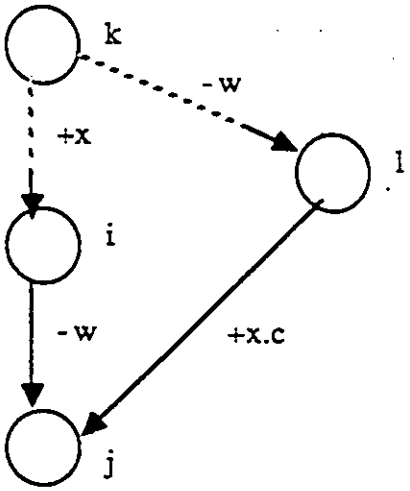
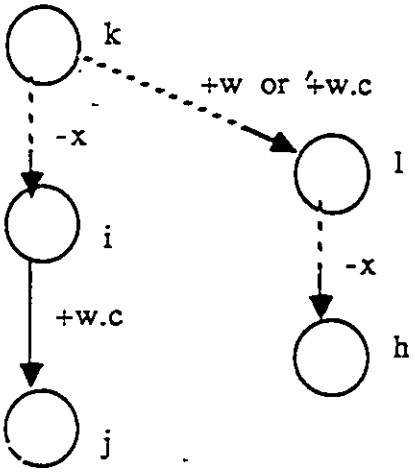
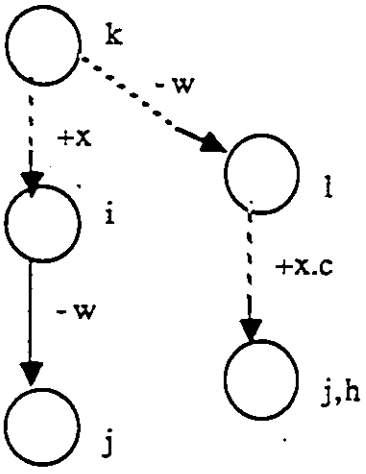
	STG 1	STG 2
RULE 1		
RULE 2		
RULE 3		
RULE 4		

Table 2.3 Transformation Rules for arc (i, j) in STG 1

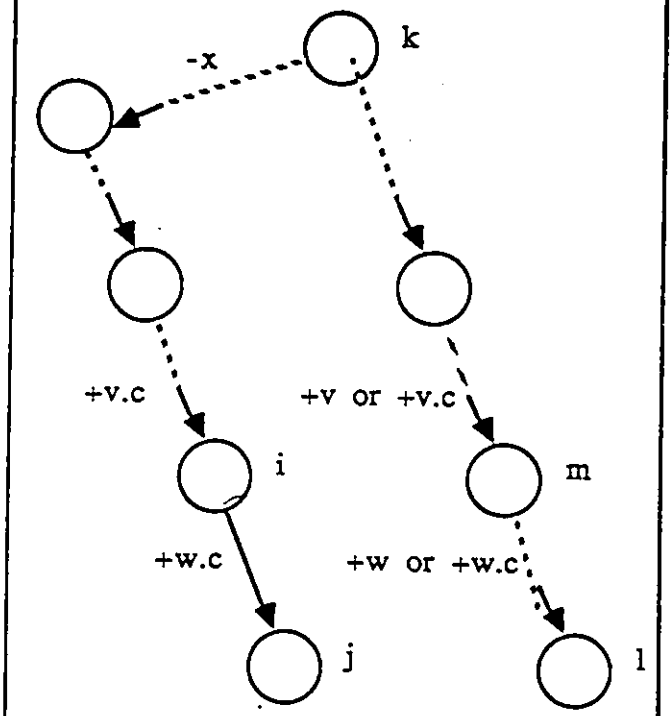
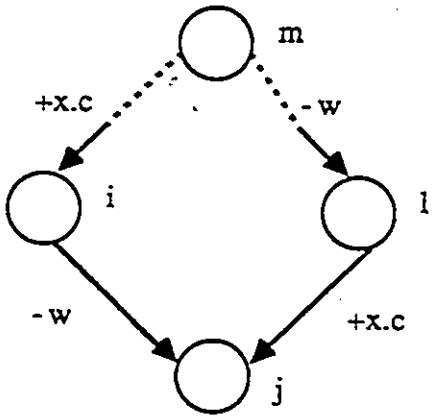
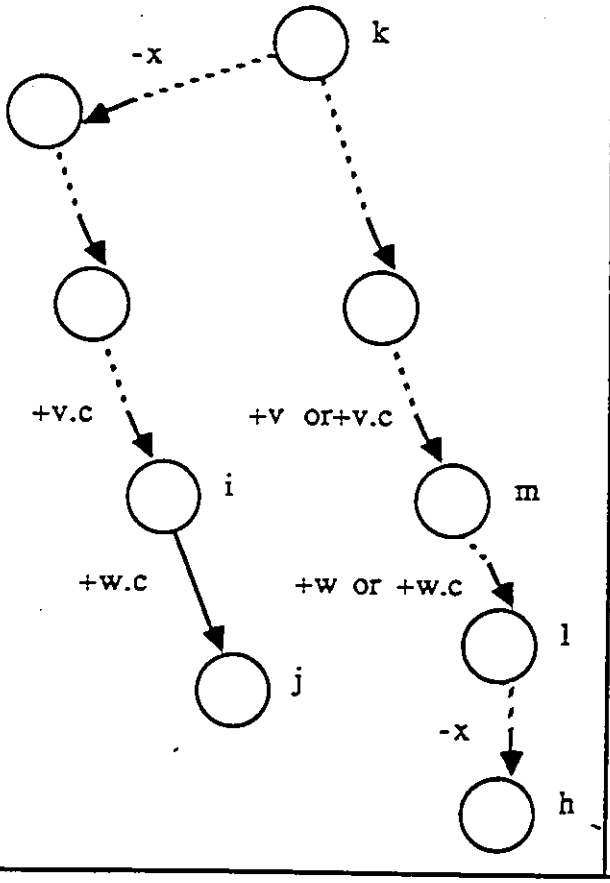
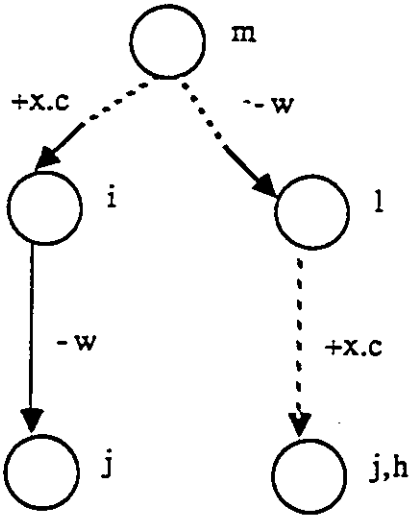
	STG 1	STG 2
RULE 5		
RULE 6		

Table 2.3 (Continued)

Lastly, Rules 5 and 6 handle the situation where a subordinate receive transition follows another subordinate receive transition. They are the extentions of Rules 3 and 4. In Rules 3 and 4, the subordinate receive transition is the first of a sequence of subordinate receive transitions (if there is any) whereas in Rules 5 and 6, it is a subordinate receive transition subsequent to the first one. These Rules are derived from Rules 3 and 4 by taking into account the induction on the length of sequences of subordinate receive transitions.

Construction of the peer entity

After STG2 has been constructed as a finite state machine, the Petri net representation of the peer entity can be created by the following steps

a) Create the external input places and external output places to represent the received and transmitted messages, respectively.
b) Represent each state of STG2 by an internal place.
c) Create transitions corresponding to arcs in STG2.
d) Define the input and output functions of the transitions based on the labels as well as the initiating and ending states of the corresponding arcs. Hence, an arc with label -x from state i to state j in STG2 is represented by a transition t with {i} as its input place and {j,x} as its output places. Similarly, if the label is +y, then i and y are the input places, and j is the only output place for the transition t.

e) Define the initial marking of the petri net to be the initial state of STG2.

Dong guarantees that the resulting protocol is correct if: the local entity satisfies the local properties stated in Section 1.3.

2.4.2 Comments on the Method

The method has the following shortcomings:

1) The method assumes that there are only two communicating entities. Protocols with N entities cannot be applied.

2) The method transforms the Petri net representation of the local entity into a state transition graph STG1, constructs the peer state transition graph STG2 from STG1 and then constructs the Petri net representation of the peer entity from STG2. This process will be greatly improved if the Petri net representation of the peer entity can be directly created from the Petri net representation of its local entity without passing through the intermediate steps.

Chapter 3

PNPS - A PETRI-NET-BASED PROTOCOL SYNTHESIZER

3.1 Introduction

In this chapter, we present in details the algorithmic aspects of our research results - a Petri-net-based protocol synthesizer called PNPS. Implementation of PNPS is reported in Chapter 4.

PNPS is applicable to protocols with only two directly coupled communicating entities (i.e., without explicit representation of the communication links). In our investigation, only interactions between the communicating entities but not their internal functions are considered. Hence, each protocol entity consists of only "send", "receive" transitions and internal transitions. Since the communication channels are not represented in the model, their functions are expressed by including external input and output places in the entities. Without loss of generality, we assume that each "send" transition has exactly one external output place, one internal input place and one internal output place, and that each "receive" transition has exactly one external input place, one internal input place and one internal output place. Figure 3.1 shows an example of these two transitions.

To apply PNPS, the local entity should be specified as a Petri net and should satisfy the following properties: completeness, channel boundedness, liveness, absence of undesirable final states, absence of

cycles of send transitions, absence of cycles of receive transitions, and proper termination. PNPS will generate the peer entity. The resulting protocol composed of both specified entities satisfy the following logical properties: deadlock-freeness, boundedness, completeness, liveness and proper termination.

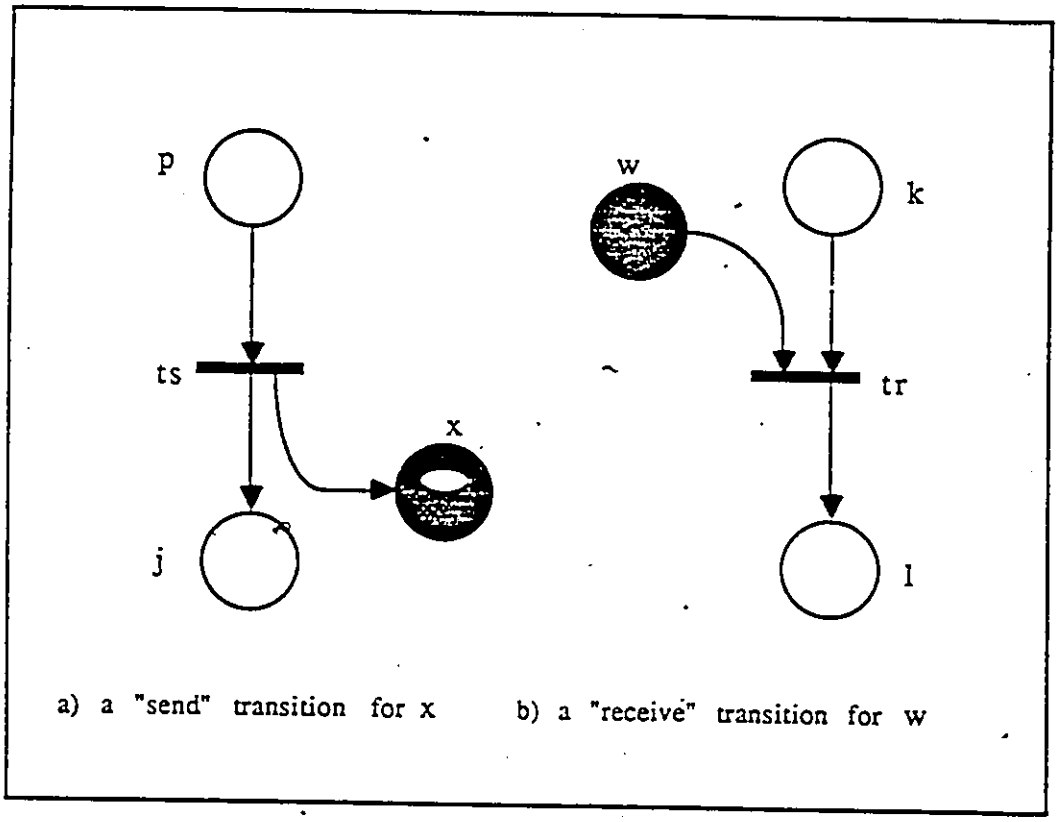


Figure 3.1 An example of the "send" and "receive" transitions

3.2 The Petri-net-based Protocol Synthesizer PNPS

Since a Petri net can be represented either as a graph or as a matrix,

PNPS is described below in terms of both representations.

PNPS obtains the result in four steps. The first step transforms the given graphical representation of the local entity to a matrix representation. The second step initializes the matrix representation of the peer entity. The third step adds transitions to this matrix. The fourth step generates the graphical representation of the peer entity.

Motivation for Steps II and III of PNPS

According to the Principle of Compatible Interaction Sequences stated in Section 1.1, the transition sequences of the local entity and the peer entity should be compatible with each other. In Table 3.1, the local entity column shows some transition sequences, with lengths one or two, which start at place m of the local entity shown in Figure 3.2.a. The peer entity column shows the compatible transition sequence or sequences which should be generated (Figure 3.2.b). For example, if there is a transition sequence t(m, i; -x).t(i, j; +w) in the local entity, then, by the Principle of Compatible Interaction Sequences, the sequences t(m, i; +x).t(m, h; -w) and t(i, j; -w).t(h, j; +x) should be generated in the peer entity. Step II of PNPS generates the first three transitions and Step III the last one (i.e., t(h, j; +x)).

The correspondence between Table 3.1 and Figure 3.2. is explained below:

t(m, i; -x) = t1	t(m, i; +x) = t'1
t(m, h; +w) = t2	t(m, h; -w) = t'2
t(i, j; +w) = t3	t(i, j; -w) = t'3
	t(h, j; +x) = t'4

Local entity	Peer entity
t(m, i; -x)	t(m, i; +x)
t(m, h; +w)	t(m, h; -w)
t(m, i; -x).t(i, j; +w)	t(m, i; +x).t(i, j; -w) or t(m, h; -w).t(h, j; +x)

Note: t(i, j; $\pm$x) denotes a transition with internal input place i, internal output place j, and external input place (+x) or external output place (-x).

Table 3.1 Compatible transition sequences

STEP I : Construct the matrix representation of the local entity from its graphical representation

Suppose the given graphical representation of the local entity has N1 internal places, N2 external input places , N3 external output places and M transitions. For convenience in description, it is assumed that the transitions and each type of places are numerically labeled.

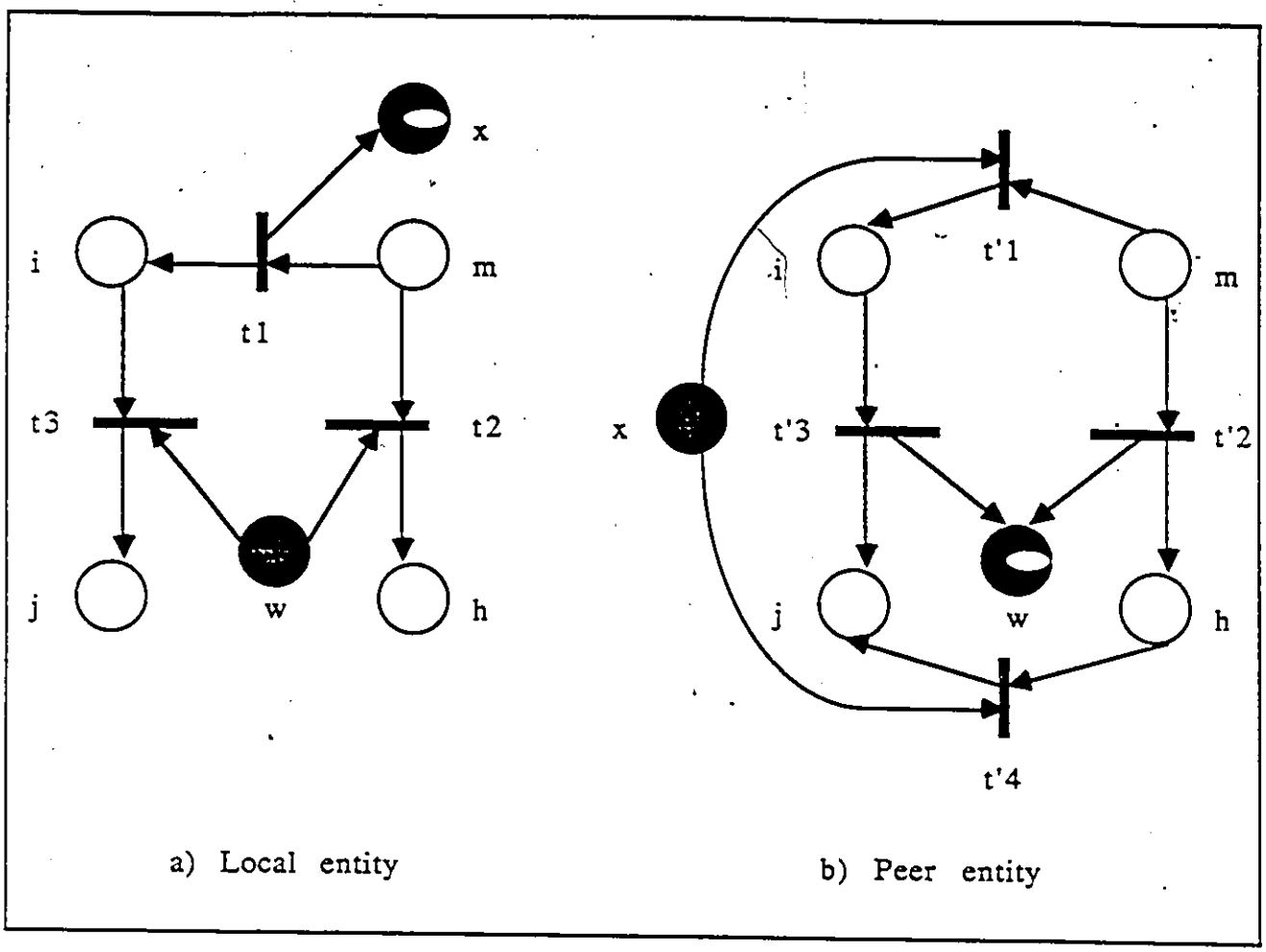


Figure 3.2 Compatible transition sequences between entities

The input matrix L- and output matrix L+ of the local entity both have dimension M*N, where N= N1 + N2 + N3. Each row of L- and L+ corresponds to a transition in the given graphical representation of the local entity. Columnwise, as shown in Figure 3.3, L- is divided into three submatrices S, I and O and L+ is divided into three submatrices S', I' and O'.

These submatrices are defined below:

S has N1 columns, each representing an internal place. That is, for i = 1, ... ,M; j = 1, ... ,N1,

$$S(i,j) = \begin{cases} 1 & \text{if place j is an internal input place of transition i} \\ 0 & \text{otherwise} \end{cases}$$

I has N2 columns, each representing an external input place. That is, for i = 1, ... ,M; j = 1, ... ,N2,

$$I(i,j) = \begin{cases} 1 & \text{if place j is an external input place of transition i} \\ 0 & \text{otherwise} \end{cases}$$

O is an M*N3 null matrix.

S' has N1 columns, each representing an internal place. That is, for i = 1 ..., M; j = 1, ... ,N1,

$$S'(i,j) = \begin{cases} 1 & \text{if place j is an internal output place for transition i} \\ 0 & \text{otherwise} \end{cases}$$

I' is an M*N2 null matrix.

O' has N3 columns, each representing an external output place. That is, for i = 1, ... ,M; j = 1, ... ,N3,

$$O'(i,j) = \begin{cases} 1 & \text{if place j is an external output place for transition i} \\ 0 & \text{otherwise} \end{cases}$$

$$L- = \begin{vmatrix} S(1;1)\ldots\ldots S(1,N1) & I(1,1)\ldots\ldots I(1,N2) & O(1,1)\ldots\ldots O(1,N3) \\ \vdots \quad\quad \vdots & \vdots \quad\quad \vdots & \vdots \quad\quad \vdots \\ S(M,1)\ldots\ldots S(M,N1) & I(M,1)\ldots I(M,N2) & O(M,1)\ldots O(M,N3) \end{vmatrix}$$

$$L+ = \begin{vmatrix} S'(1,1)\ldots\ldots S'(1,N1) & I'(1,1)\ldots\ldots I'(1,N2) & O'(1,1)\ldots\ldots O'(1,N3) \\ \vdots \quad\quad \vdots & \vdots \quad\quad \vdots & \vdots \quad\quad \vdots \\ S'(M,1)\ldots S'(M,N1) & I'(M,1)\ldots I'(M,N2) & O'(M,1)\ldots O'(M,N3) \end{vmatrix}$$

Figure 3.3 Input matrix L- and output matrix L+ of the local entity

STEP II: Create the initial elements of the peer entity

This step starts with an empty peer entity. Then, corresponding to each internal place, each external input place and each external output place of the local entity, an internal place, an external output place and an external input place is added to the peer entity, respectively.

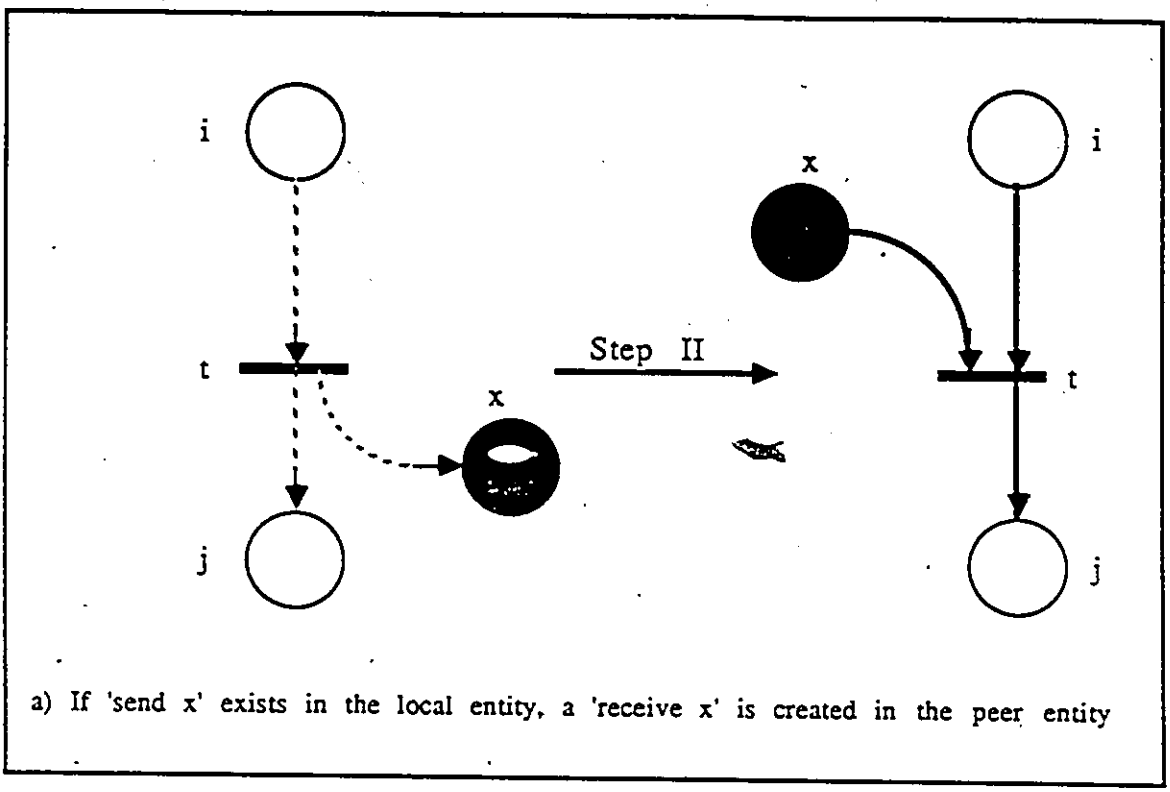
a) Description in terms of the graphical representation

Figure 3.4 shows the graphical creation process. First, for each place (internal or external) of the local entity, a place with an identical label is created for the peer entity. Then, for each send (respectively, receive)

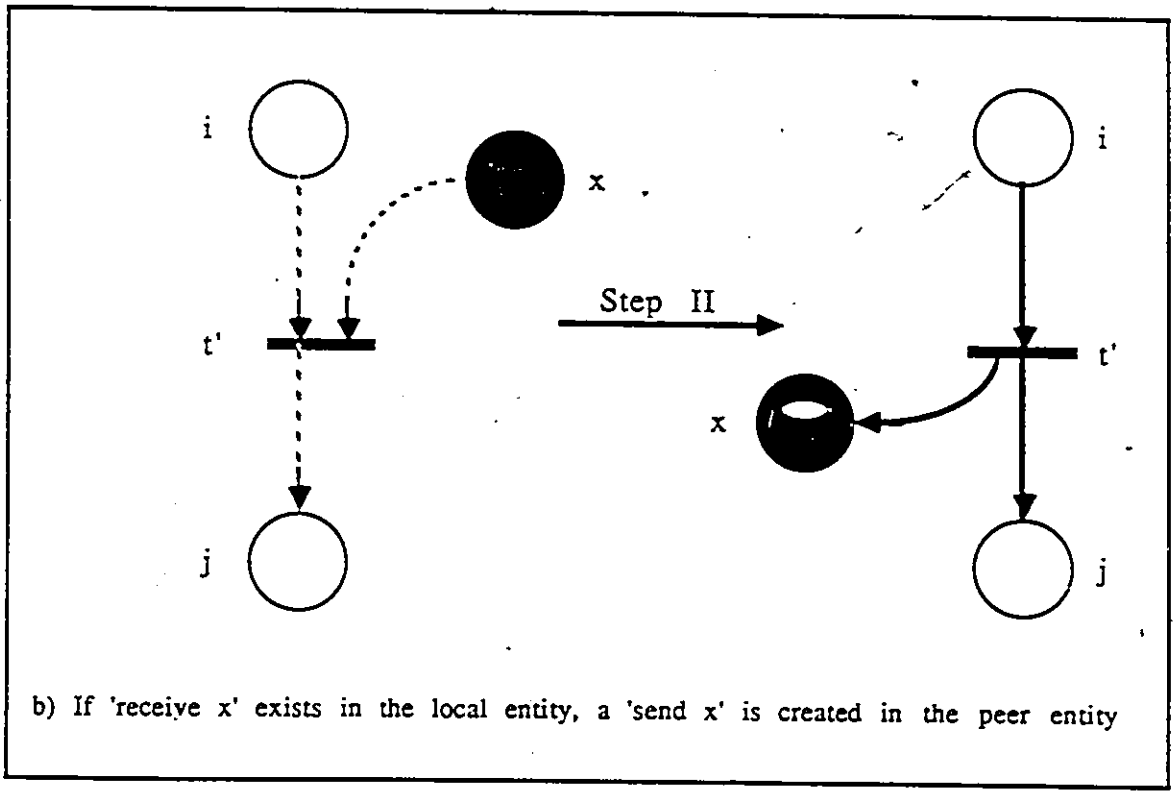
transition for a message x in the local entity, this step generates a receive (respectively, send) transition for x in the peer entity. More specifically, for each transition t of the local entity having internal place i, internal output place j and external output (respectively, input) place x, a corresponding transition t' is created in the peer entity, having internal input place i, internal output place j and external input (respectively, output) place x.

b) Description in terms of matrix representation

In this step, an input matrix P- and an output matrix P+ are created for the peer entity from the input and output matrices L- and L+ of the local entity. In the graphical description, if a place i is an internal input (respectively, output) place of a transition t for the local entity, the same place i is an internal input (respectively, output) place of transition t for the peer entity. Hence, in matrix representation, the submatrix S (respectively, S') of matrix L- (respectively, L+) should occupy the first N1 columns of P- (respectively, P+) . If a place x is an external input (respectively, output) place of a transition t in the local entity, the corresponding transition in the peer entity has the same place x as an external output (respectively, input) place. Hence, the submatrices I' and O' of L+ constitute the second and third submatrices of P-, respectively. Similarly, the matrices I and O of L- constitute the second and third submatrices of P+, respectively. Figure 3.5 shows the generated matrices P- and P+ for the peer entity.



a) If 'send x' exists in the local entity, a 'receive x' is created in the peer entity



b) If 'receive x' exists in the local entity, a 'send x' is created in the peer entity

Figure 3.4 Initialization of the peer entity in Step II

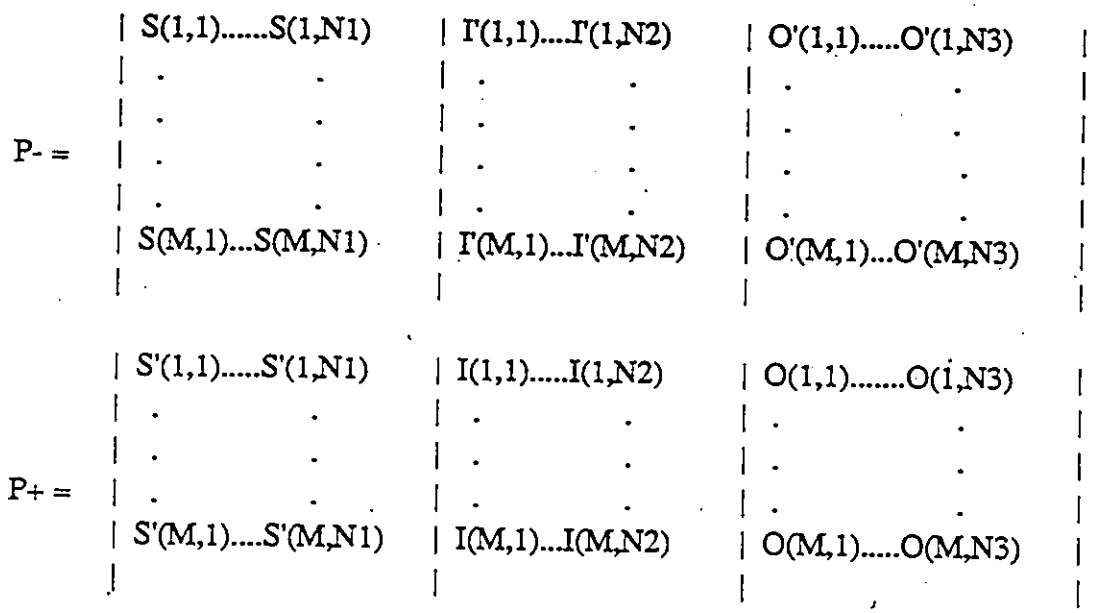


Figure 3.5 Initialized matrices P- and P+ of the peer entity

Symbolically, the creation process of Step II is shown below:

L- = | S | I | O | L+ = | S' | I' | O' |

| Step II

P- = | S | I' | O' | P+ = | S' | I | O |

The following example illustrates Steps I and II of the Synthesizer PNPS.

Example 3:1 The local entity shown in Figure 3.6 has two transitions t1 and t2, two internal places p1 and p2, one external input place req2 and one external output place req1. t1 represents the "send" transition of message req1. t2 represents the "receive" transition of message req2.

The graphical representations for the given local entity and the created peer entity are shown in Figure 3.6. Figure 3.7 shows the input matrix L- and output matrix L+ for the local entity. Figure 3.8 shows the created input matrix P- and output matrix P+ for the peer entity.

For instance, L-(1,1) =1 means that internal place p1 is an input place of transition t1. L-(1,2) = 0 means that internal place p2 is not an input place of transition t1. L-(2,3) = 1 means that external place r2 is an input place to transition t1.

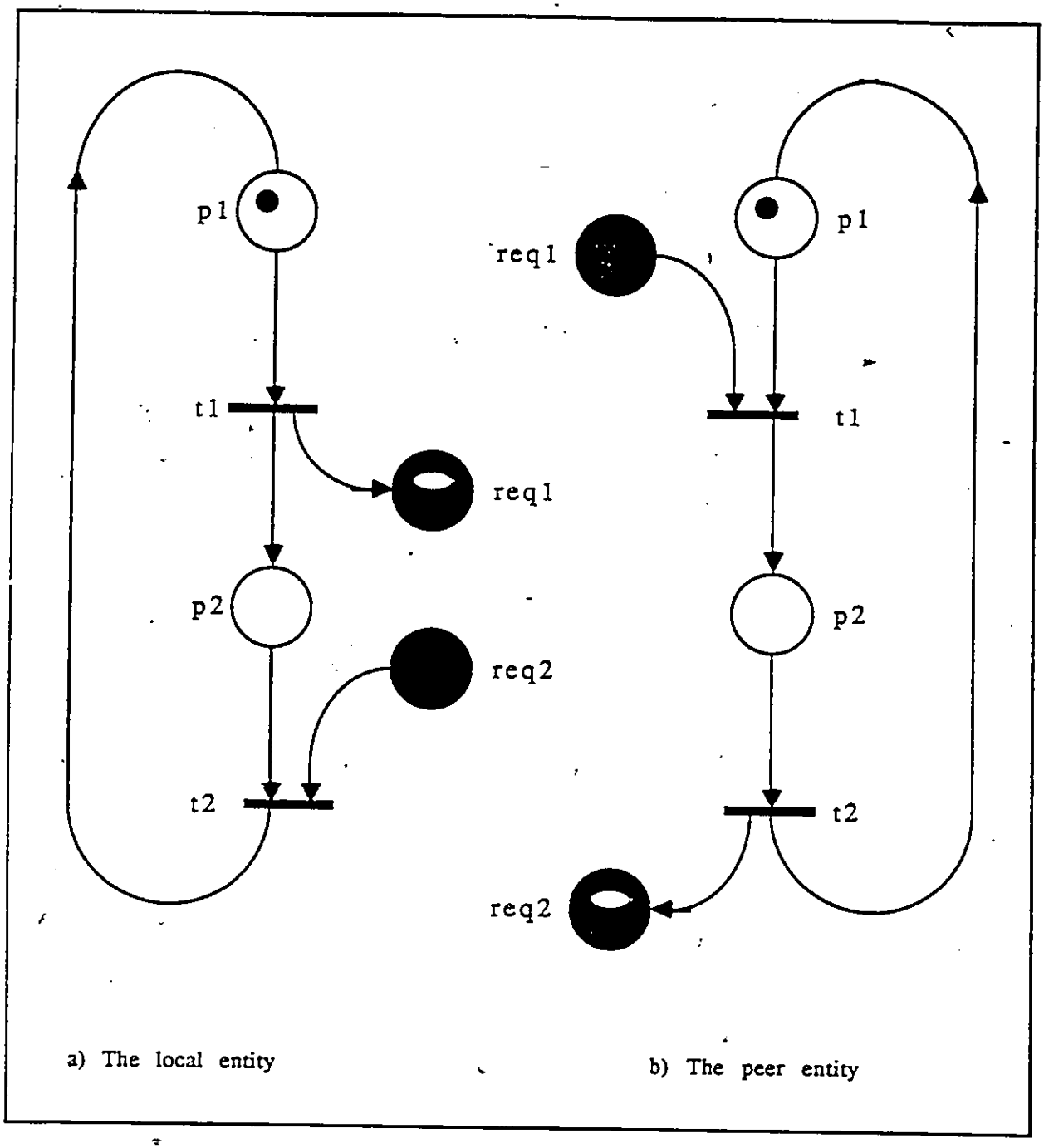


Figure 3.6 The local entity and peer entity in graphical representation

Matrix L- :

	p1	p2	req2	req1
t1	1	0	0	0
t2	0	1	1	0

Matrix L+ :

	p1	p2	req2	req1
t1	0	1	0	1
t2	1	0	0	0

Figure 3.7 Input and output matrices for the local entity

Matrix P- :

	p1	p2	req2	req1
t1	1	0	0	1
t2	0	1	0	0

Matrix P+ :

	p1	p2	req2	req1
t1	0	1	0	0
t2	1	0	1	0

Figure 3.8 Input and output matrices for the peer entity

STEP III : Adding transitions into the peer entity

This step is an iterative process. Each iteration adds a transition to the peer entity if some conditions of the local and the peer entities are satisfied.

a) Description in terms of graphical representation

Apply the following process repeatedly until it is no longer possible (i.e., one or more of the conditions cannot be satisfied).

Process (Figure 3.9): A transition t4 with internal input place h, internal output place j and external input place x, is added to the peer entity if the following three conditions are detected:

(i) In the local entity, there is an external input place w for two receive transitions t1 and t2, where t1 has internal input place i and internal output place j, and t2 has internal input place m and internal output place h.

(ii) In the peer entity, there is an external input place x for a receive transition t3 having internal input place m and internal output place i.

(iii) In the local entity, there is no send transition having x as an external output place and h as an internal input place.

b) Description in terms of matrix representation

In this step, the following process is applied repeatedly until it is no longer possible (i.e, one or more of the conditions cannot be satisfied). Essentially, it checks the matrices L- and L+ of the local entity and the matrices P- and P+ of the peer entity. If some conditions (Figure 3.10) are satisfied, a row is added at the bottoms of the matrices P- and P+ of the peer entity.

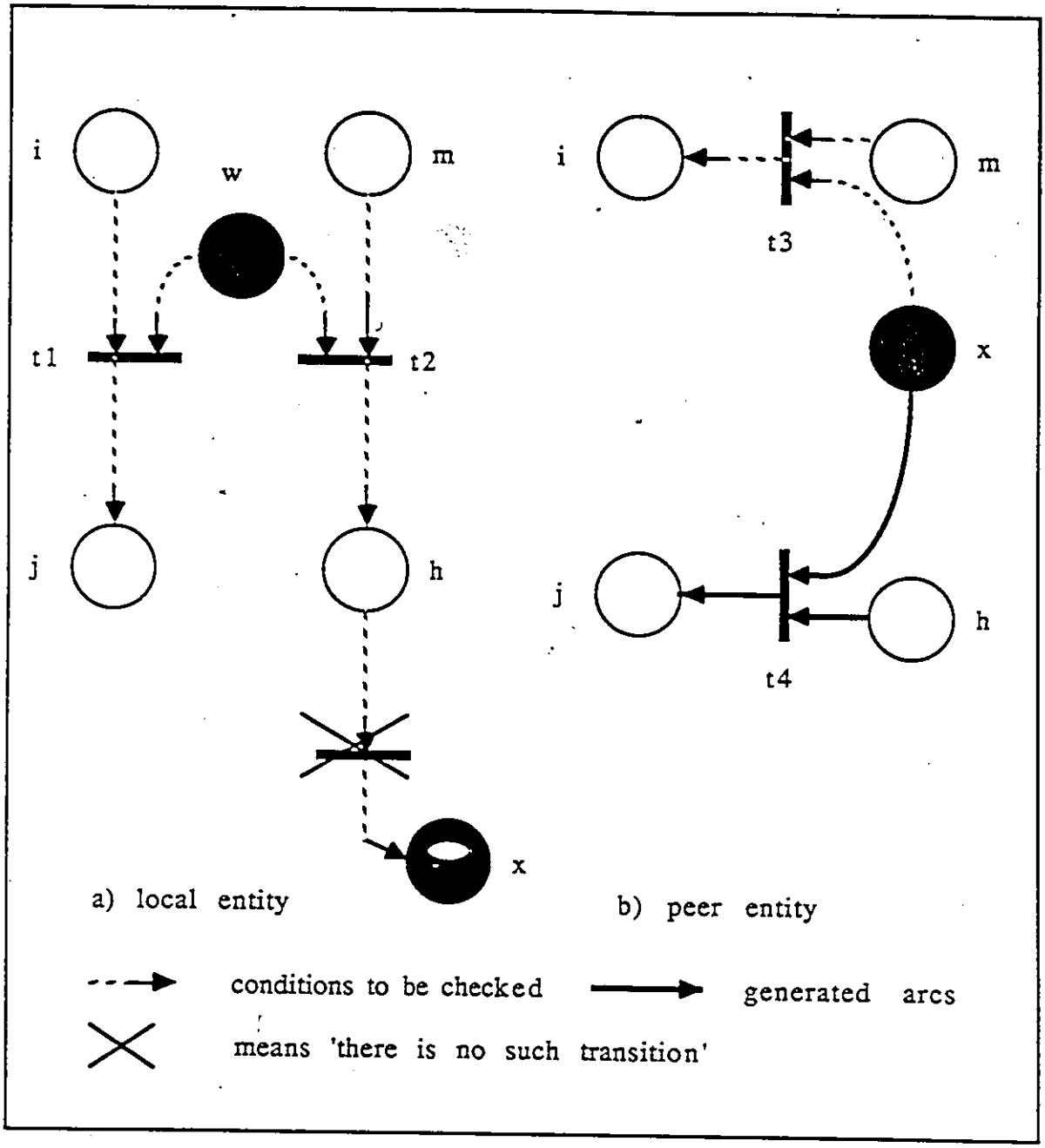
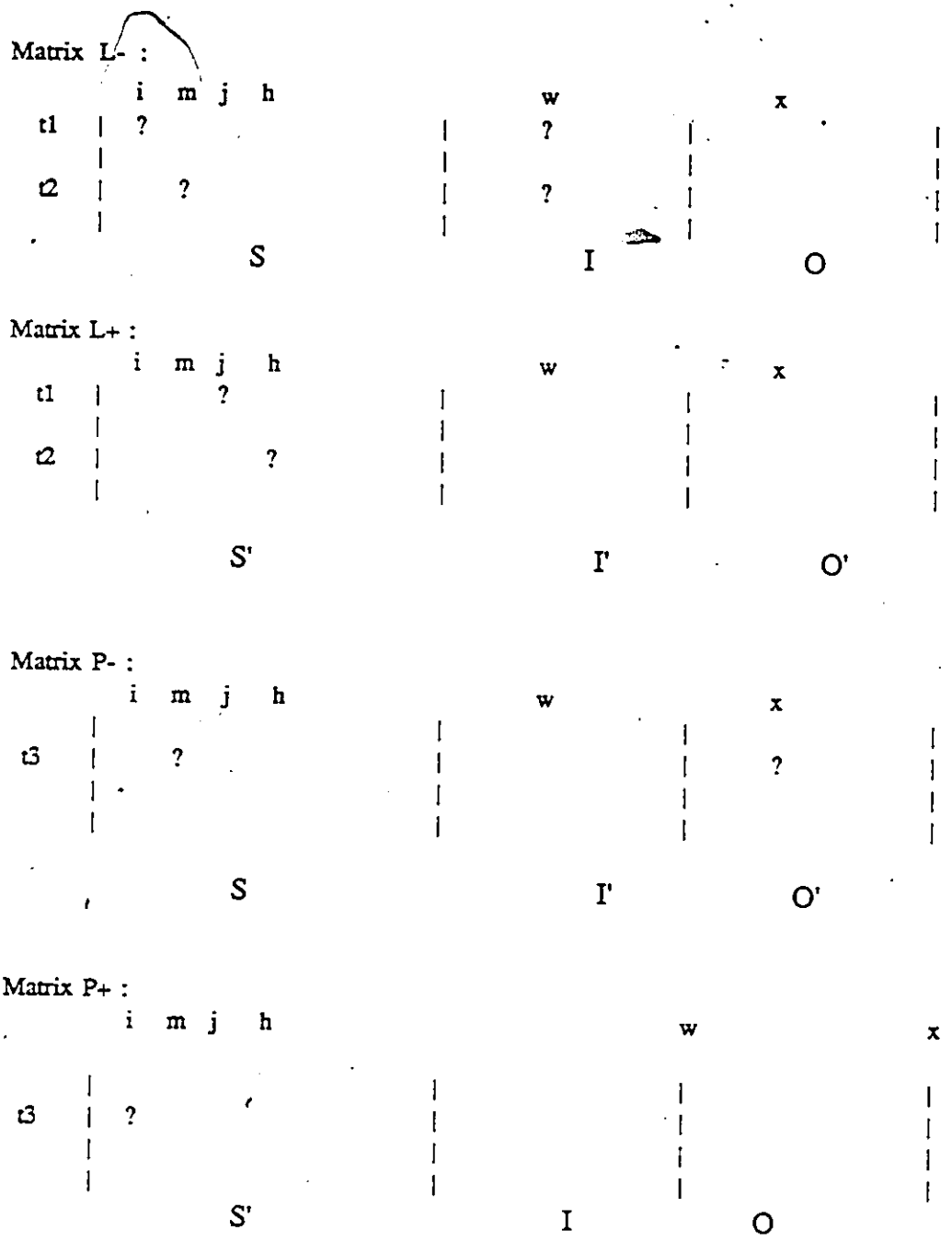


Figure 3.9 Graphical representation of Step III



Note: ? indicates those elements to be checked

Figure 3.10 Conditions of Step III in matrix representation

Process: Suppose the following three conditions are satisfied:

(i) In L- and L+, there exist two rows t1 and t2 and five columns i, j, m, h and w, such that

$S(t1,i) = S'(t1,j) = I(t1,w) = 1$

$S(t2,m) = S'(t2,h) = I(t2,w) = 1$

(ii) In P- and P+, there exist a row t3 and three columns m, i and x, such that

$S(t3,m) = S'(t3,i) = O'(t3,x) = 1$

(iii) In L- and L+ , $S(t,h) = O'(t,x) = 0$, for all t where $1 \leq t \leq M$ and $t \neq t1, t2$.

Then, the following row

(0; 0; ...; S(u+1, h); ...; O'(u+1, x); ...; 0), where S(u+1, h) = O'(u+1, x) = 1

should be added at the bottom of P- and the following row

(0; ...; S'(u+1, j); ...; 0), where S'(u+1 ,j) = 1

should be added at the bottom of P+, where u is the number of rows of both P- and P+ before addition.

STEP IV : Construct the graphical representation of the peer entity from its matrix representation

Once P- and P+ have been determined in Step III, the graphical representation of the peer entity can be constructed as follows:

Corresponding to the first N1 columns, the next N2 columns and the last N3 columns of P- and P+, we construct N1 internal places, N2 external output places and N3 external input places, respectively. If the number of rows of P- and P+ is R, R transitions are created in the peer entity as follows: An arc is generated from internal place j (respectively, external place j) to transition i, if S(i,j) =1 (respectively, O'(i,j) = 1). An arc is generated from transition i to internal place j (respectively, external place j), if S'(i,j) = 1 (respectively, I(i,j) = 1).

Remarks on PNPS

Following are some remarks on PNPS.

(i) If the protocol to be synthesized starts with the graphical representation of the local entity and the result is required in both the matrix and graphical formats, then all four steps should be executed. On the other hand, if the problem starts with the matrix representation of the local entity and the result is also required just in matrix representation, then only Steps II and III have to be executed.

(ii) PNPS is computationally faster than Dong's APS, because, during the checking of the conditions, PNPS searches matrices whereas APS scans finite state machines.

(iii) The peer entity generated by PNPS has the same number of places as the local entity. The former may have more transitions because step III adds rows to the bottoms of P- and P+.

3.3 Validity of Synthesizer PNPS

The goal of PNPS is to generate error-free protocols (Section 3.1). It has been proven that Dong's APS [RAMA 85] generates an error-free protocol. Hence, showing the equivalence between Steps II and III of PNPS and Dong's Transformation Rules constitutes a complete proof of the validity of PNPS. To show that PNPS is equivalent to APS, we have to prove that they perform exactly the same functions.

Functions achieved by **Rules 1 to 6 of Dong's method** (Section 2.4):

1) If a target transition in the local entity has an internal input place i and an internal output place j, then the corresponding transition generated in the peer entity also has internal input place i and internal output place j.

2) If in the local entity the target transition is a receive transition for a message x, then, a send transition for the same message x is created in the peer entity.

3) If in the local entity the target transition is a send transition for the message x, then a receive transition for the same message x is created in the peer entity.

Rules 3 and 5 of APS have the following additional function:

4) In the peer entity some necessary transitions are created which do not correspond to any transition in the local entity (see Table 3.1 and Figure 3.10).

Table 3.2 explains how the above four functions of APS are achieved in Steps II and III of PNPS.

Function of APS	Corresponding operations in PNPS
1	Submatrices S and S' occupy the first N1 columns of P- and P+, respectively (Step II).
2	Submatrices I and O are created in columns (N1+1) to (N1+N2) and columns (N1+N2+1) to (N1+N2+N3) of P+, respectively (step II).
3	Submatrices I' and O' are created in columns (N1+1) to (N1+N2) and columns (N1+N2+1) to (N1+N2+N3) of P-, respectively (Step II).
4	Step III adds one or many rows into P- and P+. These rows represent transitions in the peer entity which do not correspond to any transition in the local entity

Table 3.2 Correspondence between APS and PNPS

Note: In APS, Rule 3 is a special case of Rule 5. In Rule 3, the receive transition of w comes right after the send transition of x in the local entity; whereas in Rule 5, that receive transition may not be the first one coming after the send transition of the message x. Step III of PNPS includes the function of Rule 5 and hence takes care of both Rule 3 and Rule 5.

The computational details of each step of PNPS are illustrated in Example 5.1.

Chapter 4

GSAPS - A GRAPHICAL SYSTEM FOR AUTOMATING PROTOCOL SYNTHESIZER PNPS

4.1 Introduction

GSAPS, a Graphical System for Automating the Protocol Synthesizer PNPS described in Chapter 3, has been implemented on an Apple MacIntosh microcomputer. It inputs the Petri net specification of the local entity graphically and interactively and outputs the graphical representation of the peer entity or both the local and peer entities.

Functionally, GSAPS is divided into two subsystems: one subsystem for graphical support (Sections 4.2), permitting the inputting, outputting and modification of graphical objects, such as circles, bars and arcs; another subsystem for executing the synthesizer PNPS (Section 4.3). A record structure called Net is used to store the information about the local and peer entities. Some programming aspects of GSAPS are described in the last sections of this chapter.

GSAPS is a menu-driven system. The user executes his requests through the various system menus. This chapter summarises the main functions of these menus. Detailed descriptions of these functions at the operational level are provided in Appendix A. The following description assumes that the readers are familiar with menu-driven, icon-based operations on a

microcomputer.

4.2 Subsystem for Graphical Support

The graphical support subsystem of GSAPS allows a designer to graphically create or alter the graphical representation of a local entity. It also provides the facilities for displaying the graphical representations of the local and peer entities. It has the following three menus: Create Net, Modify Net and File.

* **Create Net:** A user selects this menu in order to create a local entity to work on. The local entity may be totally new or be obtained by adding some elements to an existing one. The menu includes the following entries:

Entry	Function
- Int-place	create internal places
- Ext-input-place	create external input places
- Ext-output-place	create external output places
- Transition	create transitions
- Arc	create arcs between places and transitions
- Name	assign names to places and transitions
- Token	assign number of tokens to internal places

* **Modify Net:** The user selects this menu in order to delete or move the elements of an existing Petri net representation of the local entity. The menu includes the following entries:

Entry	Function
- Delete node	delete places or transitions
- Delete arc	delete arcs between places and transitions
- Move node	change the positions of places or transitions
- Draw local	display the graphical representation of the local entity
- Display or hide name	display the names of places and transitions if hidden and hide them if displayed
- Display or hide token	display the number of tokens in each internal places if hidden and hides them if displayed

* File: This menu includes the following options:

Entry	Function
- New	prepare the system for the creation of a new local entity
- Open	open an existing file containing the Petri net of a local entity.
- Save: for GSAPS	save the data of a local entity for future use by GSAPS
- Save: for MacPaint	save the graphical representation of an entity for future use by MacPaint
- Quit	exit from the GSAPS system

4.3 Subsystem for Executing the Synthesizer

This subsystem uses the matrices of the local entity created during the graphical interactions. It generates the matrix representation of the peer entity. The only menu of this subsystem is **Synthesis** which includes the following entries:

Entry	Function
- Apply Step II	initialize the peer entity
- Apply Step III	generate the final peer entity
- Draw peer	display the graphical representation of the peer entity
- Draw both	display the graphical representation of both the local and peer entities

4.4 Net - A Record Structure for GSAPS

A record structure Net is used in GSAPS to store the locations, names of the places and transitions and the arcs connecting these elements. Figure 4.1 lists the names and the maximum sizes of its fields. Figure 4.2 shows more details of the fields Name, Location and LL in the record structure Net.

Table 4.1 describes the different fields of Net. Part (a) of the table shows their types and sizes while Part (b) describes their meanings.

p	t
m	j
maxp	maxt
MAXM	MAXJ
Names for internal places	
Names for Transitions, Ext-input-places and Ext-output-places, respectively	
Locations	
Tokens	
Incoming arcs	Outgoing arcs

Figure 4.1 The fields of the record structure Net

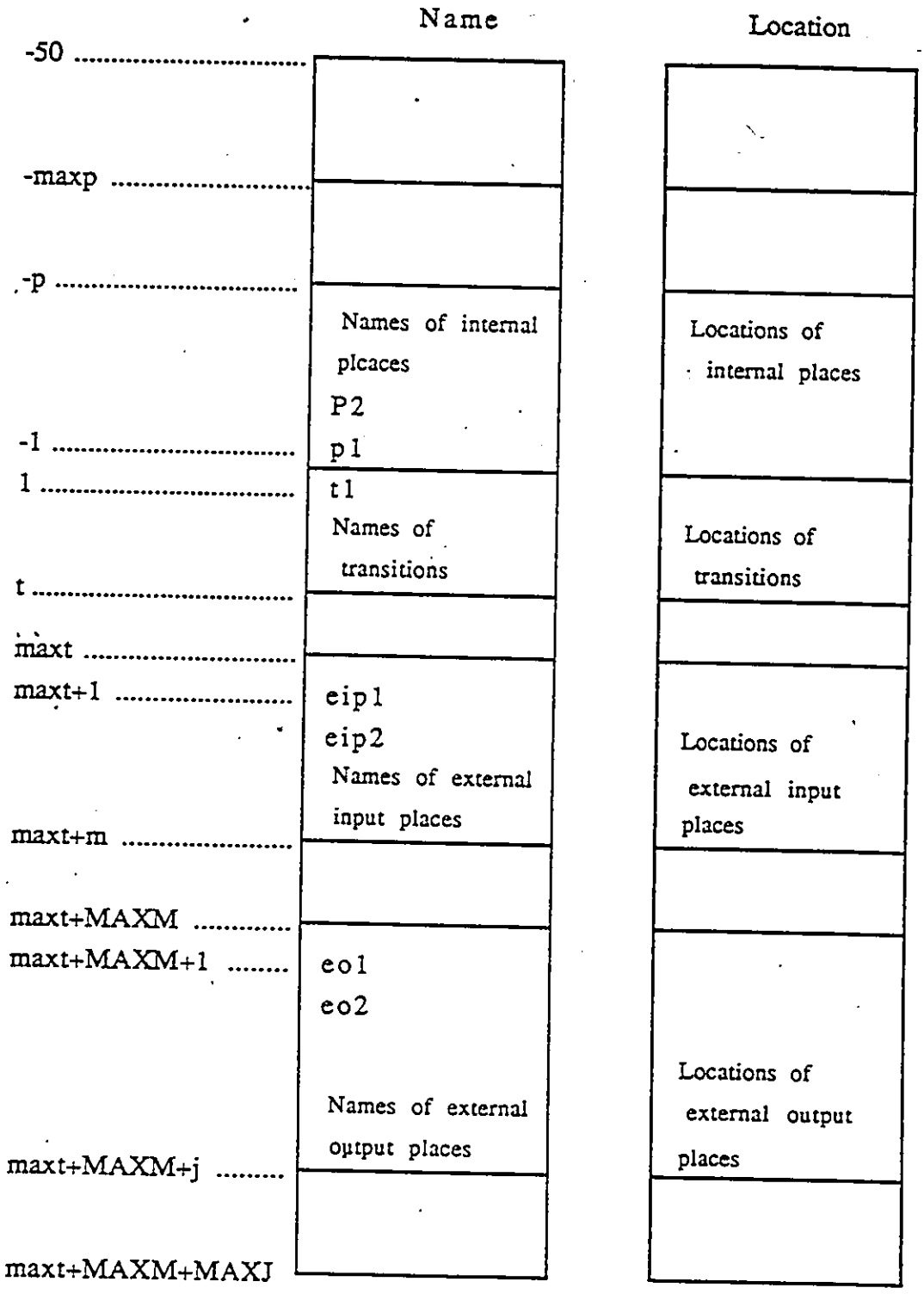


Figure 4.2 The fields Name and Location of the record structure Net

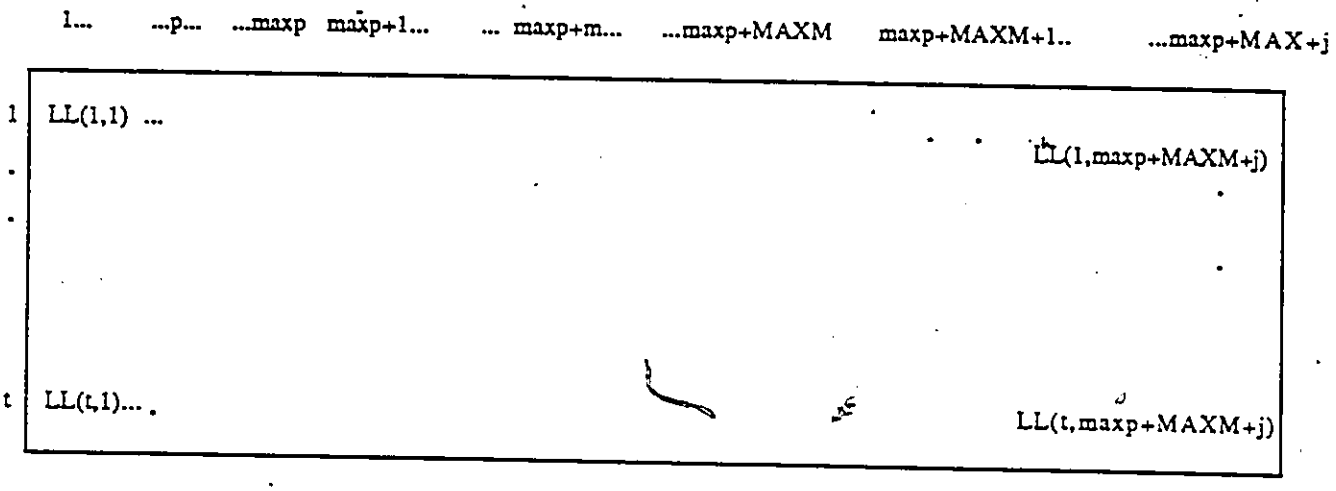


Figure 4.2 (continued) LL: matrix representation for arcs from places to transitions

```
Net = Record
        p, t, m. j, maxp, maxt, MAXM, MAXJ : integer;
        name : array [-50..80] of string [16];
        l : array [-50..80] of point;
        token : array [-50..-1] of integer;
        LL,LLP : array [1..50,1..80] of integer;
      end;
```

Table 4.1(a). Types and sizes of the fields of the record structure Net.

Field	Description
p, t, m, j	* number of internal places, transitions, external input places and external output places, respectively.
maxp, maxt, MAXM, MAXJ	* maximum numbers of the above items allowed in the net, respectively.
name	* array for storing the names of the above items.
token	* array for storing the distribution of tokens among the internal places.
LL, LLP	* arrays for storing arcs from places to transitions and from transitions to places respectively.

Table 4.1(b). Description of the fields of the record structure Net.

4.5 Pseudo-code Description of GSAPS

Figure 4.3 shows a skeleton pseudo-code of GSAPS. It contains essentially a loop in which actions are taken in response to the user's entry. Handle Menu, the main procedure of the system, calls the appropriate procedures or functions to create or modify the Petri net specification for a local entity or generates its peer entity.

Appendix B contains the program listing and detailed descriptions of each function and procedure .

```
Procedures and Functions for graphical support

Procedures for initializing the matrices of the peer entity

Procedures for searching the matrices of the local and peer
      entities to generate the final matrices of the peer entity
Procedure Handle Menu:
   Case selection of
      ...
      ...
      ...
   end case;
end Handle Menu;

Program Main;
   Loop
   Handle Menu;
   Until exit from GSAPS;
end Main;
```

Figure 4.3 Skeleton pseudo-code of the GSAPS program.

CHAPTER 5
EXAMPLES AND CONCLUSION

In this chapter, we first present three examples of applying GSAPS for the synthesis of three protocols: the Packet Radio Network Protocol [RAMA 85], the Alternating Bit Protocol [DAVI 79] and the Session Establishment and Clearing Phase of Recommendation S.62 [CCITT 81]. In particular, the first example illustrates the computational details of the four steps of PNPS, while the last two examples just show the inputs and outputs of GSAPS. Some concluding remarks and suggestions for future research are given at the end of the chapter.

5.1 Examples

Example 5.1: The Packet Radio Network Protocol

This example is taken from Ramamoorthy's article [RAMA 85]. It demonstrates the computational details of PNPS.

Suppose there are two stations in a packet radio network. Both stations are responsible for the network control and have the complete network information, such as its topology and transmission routes. To keep the data consistent, once a station detects any change in the network status, it updates its own copy and also sends the data to the other station for

updating. A protocol is needed to coordinate this process.

Suppose the graphical representation of Station 1 is given as in Figure 5.1. GSAPS will generate the entity of the other station so that their interactions are compatible and logically correct.

Initially, Station 1 is in the ready state. When a request for updating the data arrives, Station 1 will update its database and then send a "DONE" message to acknowledge Station 2. From the initial state, Station 1 can also initiate an update request to Station 2 and then enter the wait state from which Station 1 will go back to the ready state once a "DONE" message is received. However, Station 2 may send an update request while Station 1 is waiting. In this case, Station 1 will be reactivated
and process the incoming request as shown in the Figure 5.1. Further, when Station 1 is in state a, the "DONE" message for the previously sent request may be ready. Thus, Station 1 has two options:

1) absorb the "DONE" message and then process the incoming request, or

2) temporarily suppress the handling of the "DONE" message until the processing of the already received request is complete.

The application of Step I of PNPS produces the input and output matrices L- and L+ shown in Figure 5.2.

Applying Step II of PNPS generates the matrices P- and P+ (shown in Figure 5.3) for Station 2.

In this example, the iterative step (i.e., Step III) of PNPS is applied only once. That is, after Step II, P- and P+ are each increased by just one row at the bottom. The "*" in Figures 5.2 and 5.3 indicates those rows which satisfy the three conditions of Step III. The matrices P- and P+ generated

for the peer entity are shown in Figure 5.4, where the "+" indicates the added rows.

R: Ready	R1: Request 1
P: Process incoming request	R2: Request 2
C: Outgoing request has been completed	D1: Done 1
W: Wait for outgoing request to be done	D2: Done 2
A: Be activated to process incoming request	

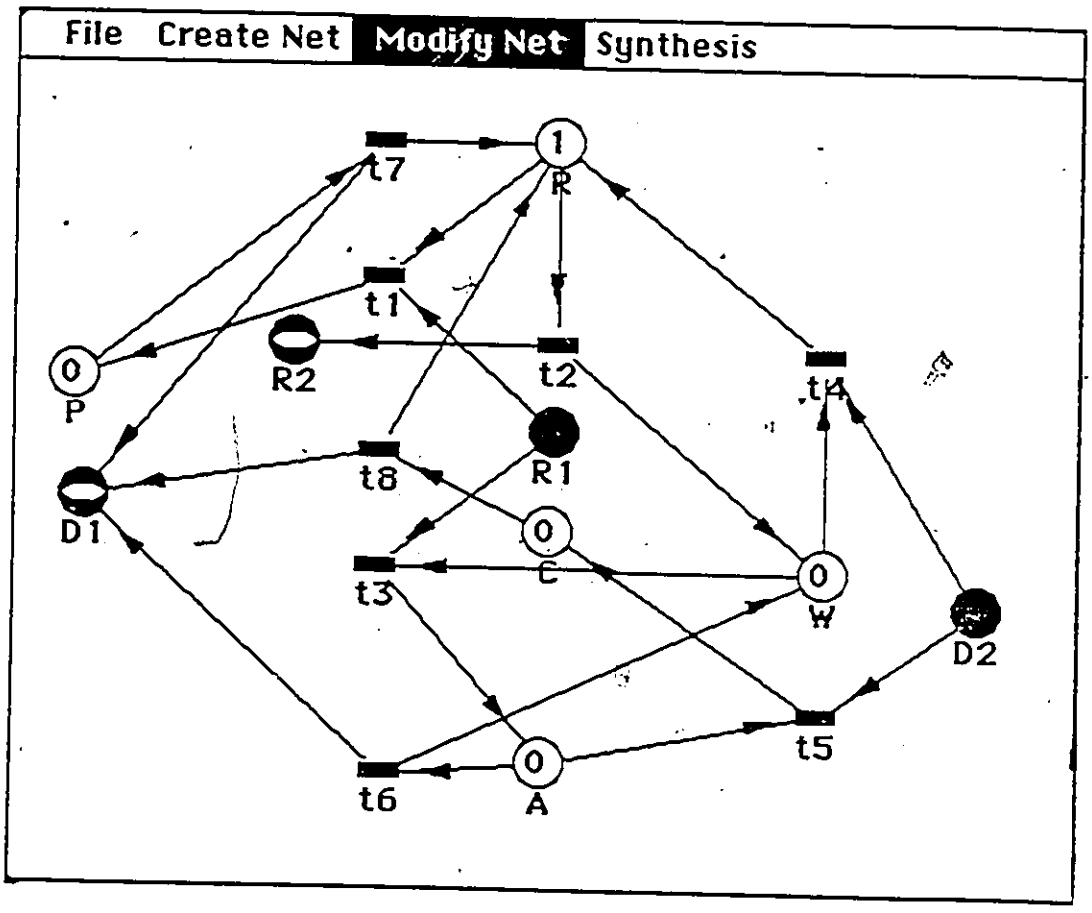


Figure 5.1 The Petri net representation of Station 1.

L- =

	R	P	W	A	C	R1	D2	R2	D1
*	1	0	0	0	0	1	0	0	0
	1	0	0	0	0	0	0	0	0
*	0	0	1	0	0	1	0	0	0
	0	0	1	0	0	0	1	0	0
	0	0	0	1	0	0	1	0	0
	0	0	0	1	0	0	0	0	0
	0	1	0	0	0	0	0	0	0
	0	0	0	0	1	0	0	0	0

L+ =

*	0	1	0	0	0	0	0	0	0
	0	0	1	0	0	0	0	1	0
*	0	0	0	1	0	0	0	0	0
	1	0	0	0	0	0	0	0	0
	0	0	0	0	1	0	0	0	0
	0	0	1	0	0	0	0	0	1
	1	0	0	0	0	0	0	0	1
	1	0	0	0	0	0	0	0	1

Figure 5.2 Input and output matrices for Station 1

P- =

R	P	W	A	C	R1	DZ	R2	D1
1	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	1	0
0	0	1	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0
0	0	0	1	0	0	0	0	1
0	1	0	0	0	0	0	0	1
0	0	0	0	1	0	0	0	1

P+ =

0	1	0	0	0	1	0	0	0
0	0	1	0	0	0	0	0	0
0	0	0	1	0	1	0	0	0
1	0	0	0	0	0	1	0	0
0	0	0	0	1	0	1	0	0
0	0	1	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0

Figure 5.3 Initial input and output matrices for Station 2

P-

	R	P	W	A	C	R1	D2	R2	D1
	1	0	0	0	0	0	0	0	0
	1	0	0	0	0	0	0	1	0
	0	0	1	0	0	0	0	0	0
	0	0	1	0	0	0	0	0	0
	0	0	0	1	0	0	0	0	0
	0	0	0	1	0	0	0	0	1
	0	1	0	0	0	0	0	0	1
	0	0	0	0	1	0	0	0	1
+	0	1	0	0	0	0	0	1	0

P+ =

	0	1	0	0	0	1	0	0	0
	0	0	1	0	0	0	0	0	0
	0	0	0	1	0	1	0	0	0
	1	0	0	0	0	0	1	0	0
	0	0	0	0	1	0	1	0	0
	0	0	1	0	0	0	0	0	0
	1	0	0	0	0	0	0	0	0
	1	0	0	0	0	0	0	0	0
+	0	0	0	1	0	0	0	0	0

Figure 5.4 Final input and output matrices for Station 2

The graphical represention of the peer entity (Station 2 of the Packet Radio Network Protocol) generated in step IV of PNPS is shown in Figure 5.5

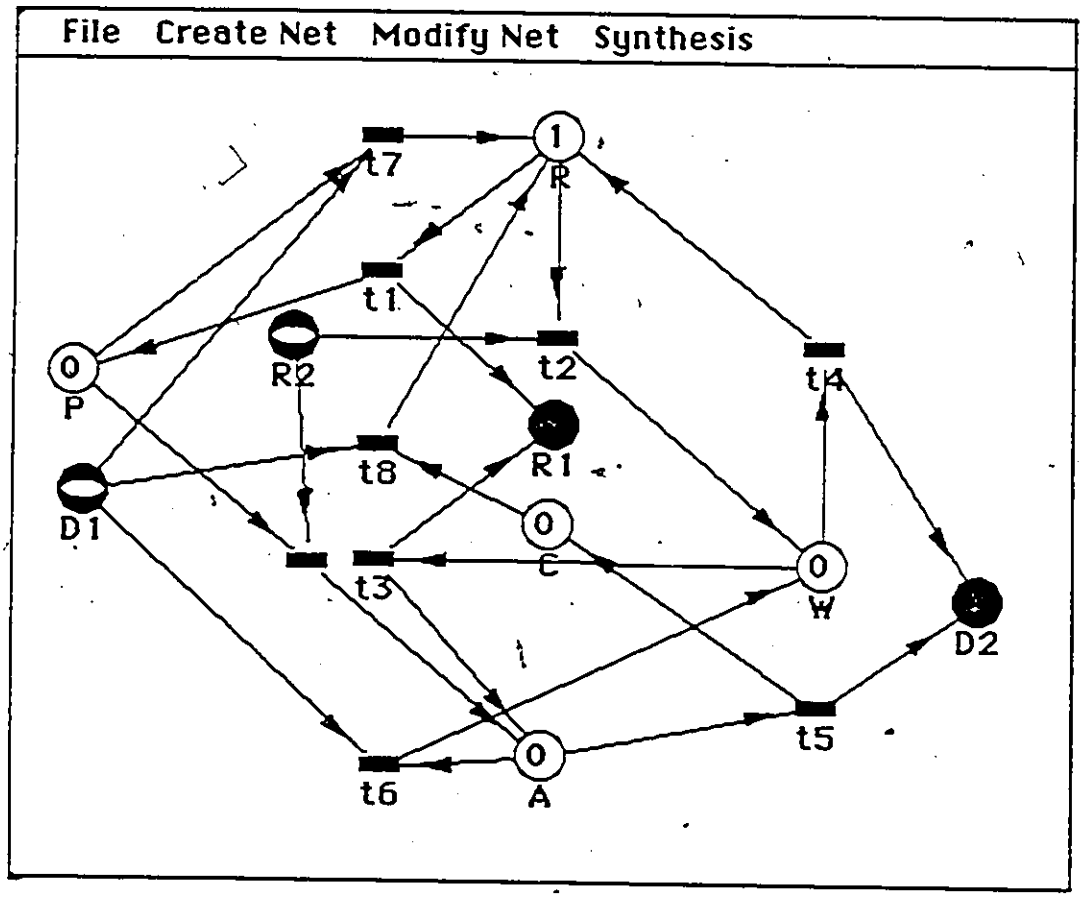


Figure 5.5 Generated Petri net representation of Station 2.

Example 5.2: The Alternating Bit Protocol

In the Alternating Bit Protocol (ABP) without error recovery [DAVI 79], every message has a one-bit sequence number whose value alternates between 0 and 1 during transmission. The local entity does not send the next message until the one sent previously has been positively acknowledged with the proper value of the sequence number. Figure 5.6 shows the local sender entity of the ABP. Transitions t1 and t3 correspond to the transmission of messages m1 and m2, respectively. Transitions t2 and t4 correspond to the receptions of k1 and k2, respectively. it1 and it2 are internal transitions for preparing the sending of the next message to the peer entity.

The graphical representation of the local entity (Figure 5.6) is input to GSAPS. By clicking the appropriate entries of the menus, the graphical representation of the peer entity (Figure 5.7) is generated as an output. In Figure 5.7 transitions t1 and t3 correspond to the reception of messages m1 and m2 by the receiver, respectively. Transitions t2 and t4 correspond to the transmissions of acknowledgements k1 and k2, respectively. it1 and it2 are internal transitions for preparing the reeeption of the next message from the sender.

A1: Ready to send m1

w1: Wait for k1

E1: Prepare m2

A2: Ready to send m2

w2: Wait for k2

E2: Prepare m2

m1: Message with seq. num. 0

k1: Acknowledgement for m1

m2: Message with seq. num. 1

k2: Acknowledgement for m2

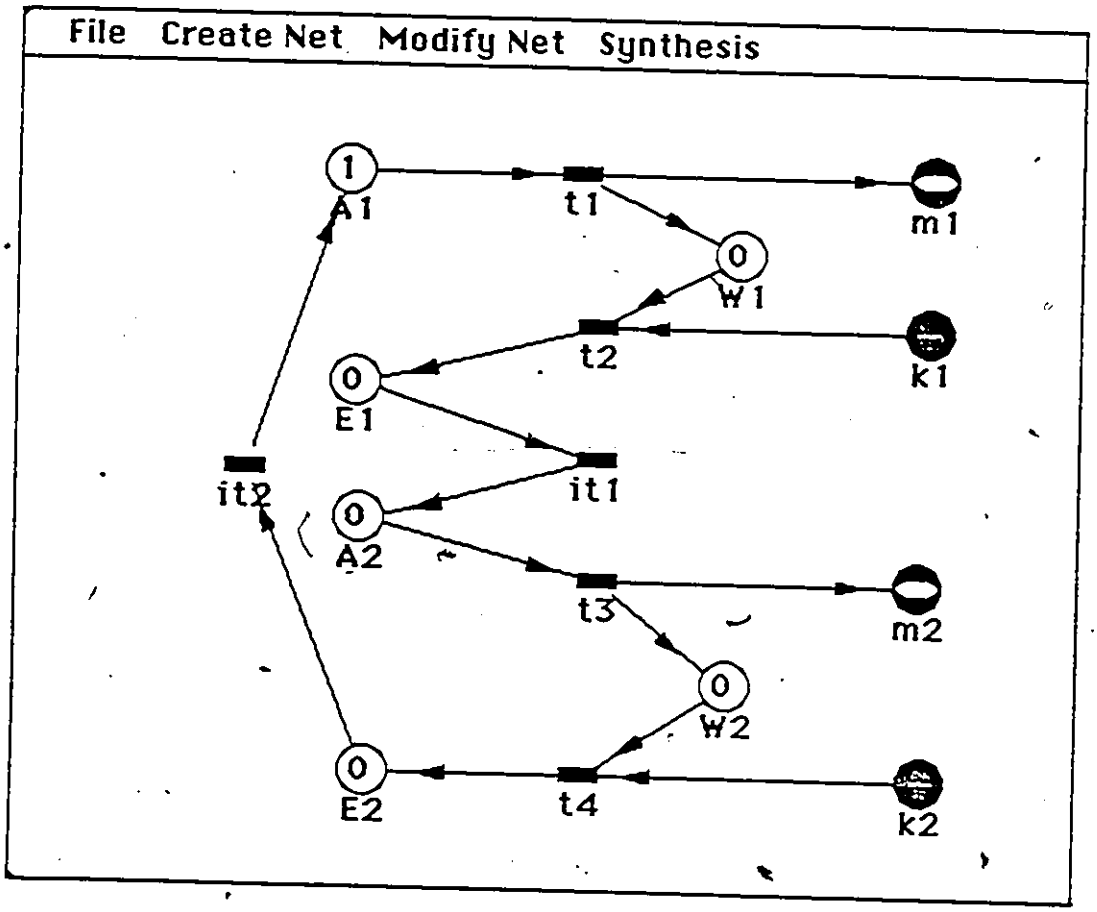


Figure 5.6 Graphical representation of the local entity (sender) of ABP

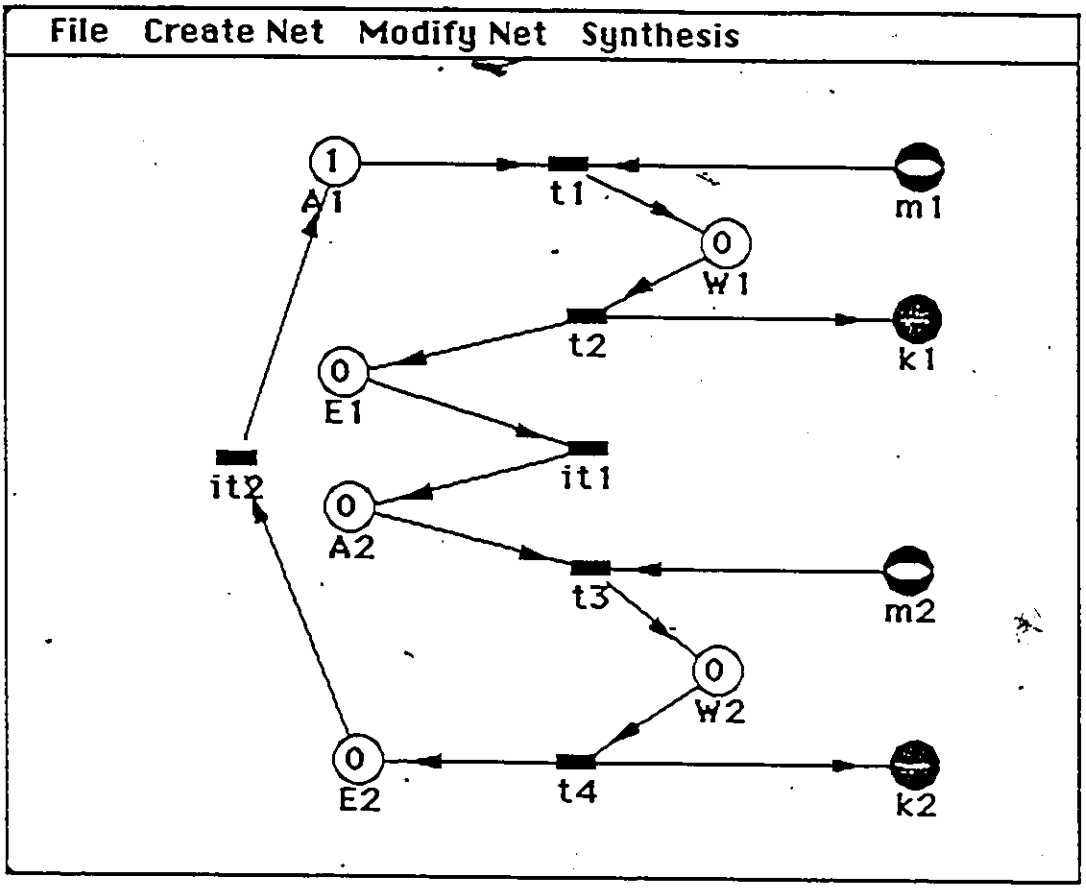


Figure 5.7 Graphical representation of the peer entity (receiver) of ABP

Example 5.3: Session Establishment and Clearing Phase of the Session Protocol S.62

Recommendation S.62 [CCITT 81] is the session protocol for teletex services. In this example, GSAPS is applied for synthesizing its Session Establishment and Clearing Phase.

The Petri net shown in Figure 5.8 represents the calling station. By applying GSAPS, the Petri net representing the called station is obtained, as shown in Figure 5.9.

The external places in the two entities have the following meanings:

cse: message for command session end
css: message for command session start
cscc: message for command session change control
cstw: message for command session two way simultaneous
rsep: message for response session end positive
rssn: message for response session start negative
rssp: message for response session start positive
rsccp: message for response session change control positive
rstwn: message for response session two way simultaneous negative
rstwp: message for response session two way simultaneous positive

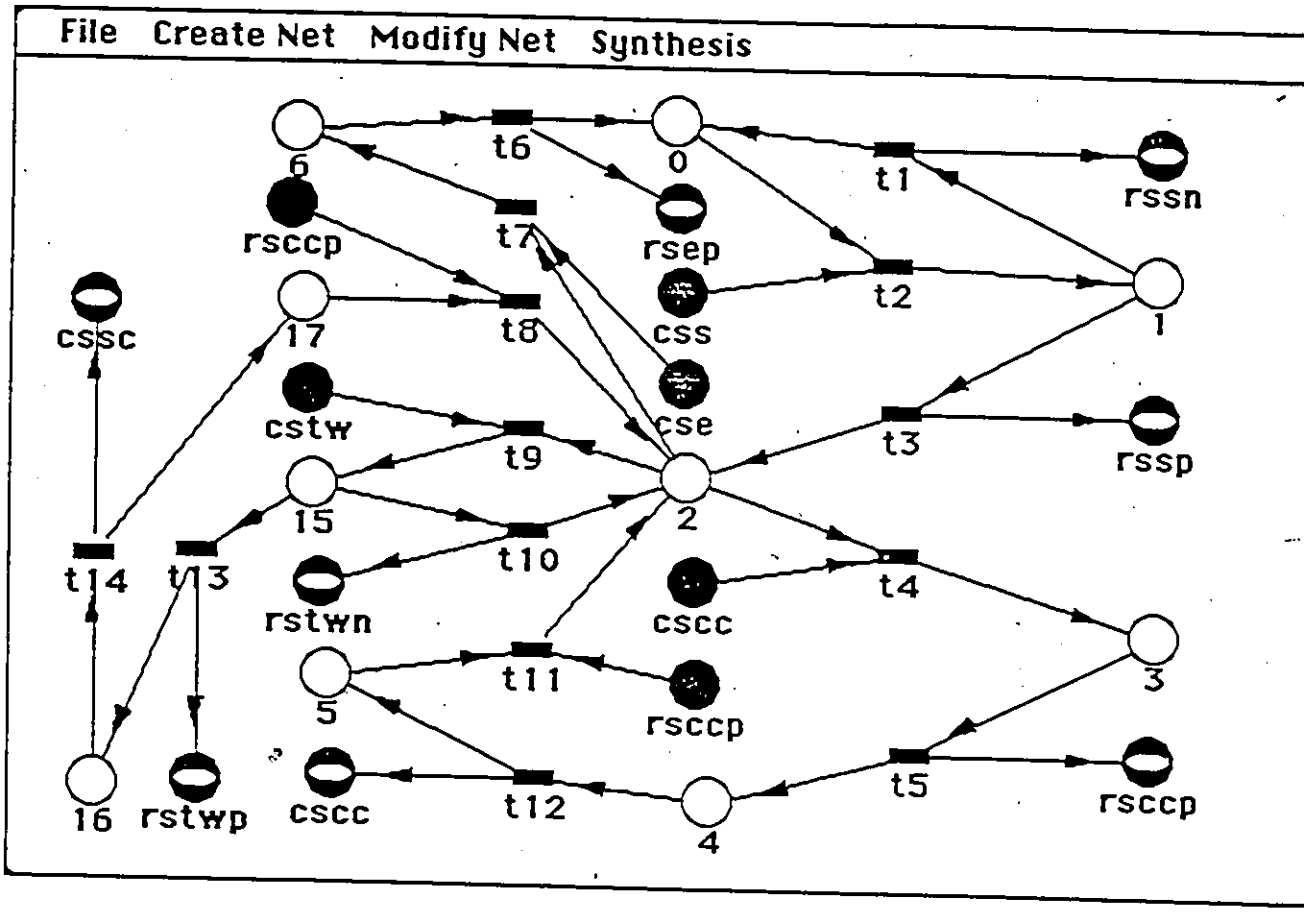


Figure 5.8 The local station in the Establishment and Clearing Phase of S.62

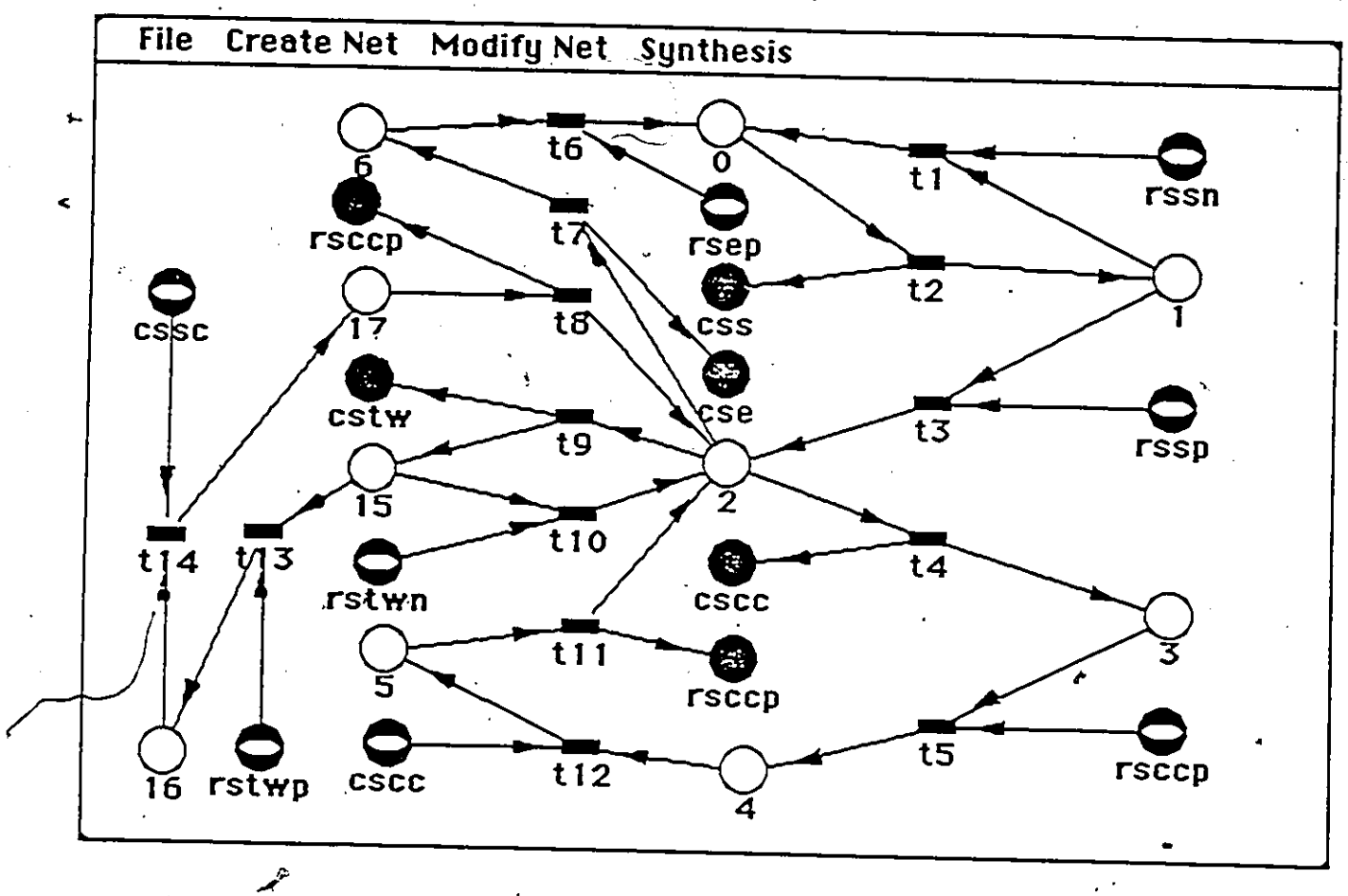


Figure 5.9 Graphical representation of the peer station in the Establishment and Clearing Phase of S.62

Note: All the results of GSAPS illustrated in the above three examples conform with specifications reported in the literature.

5.2 Conclusion

In this thesis, we propose a Petri-net-based protocol synthesizer PNPS which generates a peer protocol entity from its local entity. We have also

developed a software package, GSAPS, for automating PNPS. GSAPS is implemented on a MacIntosh microcomputer under the MacIntosh Pascal interpreter. It accepts its inputs (i.e., a Petri net representation for a local entity) graphically and interactively. It executes interactively the steps of PNPS to generate graphical outputs (i.e., the Petri net representation of the peer entity). GSAPS has been tested in synthesizing three real-life protocols, the Packet Radio Network Protocol, the Alternating Bit Protocol and the Session Establishment and Clearing Phase of the Session Protocol S.62. All the results of the tests conform with specifications or results reported in the literature.

As far as we know, GSAPS is the first graphical implementation of an interactive system used for protocol synthesis. In comparison with other systems, it has the advantages of being user-friendly and visually appealing and of being computationally efficient because of its matrix representation. It is certainly a big improvement over APS, an existing automated protocol synthesizer [RAMA 85].

GSAPS has several shortcomings which require further investigations.

1) One of the technical difficulties encountered when developing GSAPS was that the MacIntosh Pascal interpreter did not accept large programs. As a result, we have not been able to add some procedures for printing the matrix representations. When both entities are displayed on the screen simultaneously, the names of the places and transitions cannot be put at the right positions.

2) Also, because of the small size of the MacIntosh screen, if the entity displayed has a lot of places and transitions, the graphical representation of that entity will look very crowded.

3) PNPS accepts only a special class of Petri nets for specifying protocol entities consisting of only send, receive and internal transitions. Further investigations are required for problems with more general Petri nets.

4) GSAPS does not check the correctness of the logical properties in the local entity. The addition of this feature represents an important improvement to our system.

5) In our method, it is assumed that there are only two directly coupled communicating entities. It is not clear how the protocol synthesis procedure can be extended to cover a circumstance with N communicating entities.

REFERENCES

[BOCH 78] Bochmann, G.V, 'Finite state description of communication protocols', Computer Networks, Vol. 2, No. 4/5, (Sept.- Oct. 1978).

[BOCH 83] Bochmann, G.V., 'Testing transport protocol implementations', Proc. of CIPS Conference 1983, Ottawa, (May 1983), pp. 123-129.

[BRAN 83] Brand, D. and Zafiropulo, P., 'On communicating finite state machines', J. ACM, Vol. 30, No. 2, (April 1983), pp. 323-342.

[CCITT 81] CCITT Recommendation S.62, Yellow Book, Vol. VII, Fascicle VII.2, Geneva, (1981).

[CHEU 84] Cheung, T.Y. and Qiu, X., 'PNPUO - A Petri-net based software package for protocol validation', Technical Report TR84-06, Dept. of Computer Science, Univ. of Ottawa, (Nov. 1984).

[CHOI 86] Choi, T.Y., 'A sequence method for protocol construction', Proc. 6th IFIP International Workshop on Protocol Specification, Testing, and Verification, Montreal, Grey Rocks inn.,, (June 86), pp. 9/1-9/18

[DANT 77a] Danthine, A. and Bremer, R.J., 'Modelling and verification of end-to-end transport protocols', Computer Networks, Vol. 2, No. 4/5, (Oct. 1978), pp. 381-395.

[DANT 77b] Danthine, A.S., 'Petri nets for protocol modeling and verification', Proc. Computer Networks and Teleprocess. Symp., (Oct. 1977), pp. 663-685.

[DAVI 79] Davies, D.W., 'Computer networks and their protocols', John Wiley & Sons, (1979).

[DONG 83] Dong, S.T., 'The modeling, analysis and synthesis of communications protocols', Ph.D. dissertation, Dept. of Electrical Engineering, Univ. of California, Berkeley, (June 1983).

[GOUD 84] Gouda, M.G. and Yu, Y.T., 'Synthesis of communicating finite state machines with guaranteed progress', IEEE Trans. on Communications, Vol. COM-32, No. 7, (July 1984), pp. 779-788.

[HARA 79] Harazango, J., 'Protocol definition with formal grammars', Proceedings of the symposium of computer communication protocols, IFIP T.C.6, Liege, Belgium, (Feb. 1979).

[HOAR 78] Hoare, C.A.R, 'Communicating sequential processes', Comm. of the ACM, Vol. 21, No. 8, (Aug. 78), pp. 666-677.

[ISO 86] Working Document on LOTOS- Draft Proposal. DP8007- (1986).

[MERL 79] Merlin, P.M, 'Specification and validation of protocols', IEEE Trans. on Communications, Vol. COM-27, No. 11, (Nov. 1979), pp. 1671-1679.

[MILN 80] Milner, R., A calculus of Communicating Systems, Lecture Notes in Computer Science 92, (1980).

[PETE 77] Peterson, J. L., "Petri nets", ACM Computing Surveys, Vol. 9, No. 3, (Sept. 1977), pp. 223-252.

[PETE 81] Peterson, J. L., Petri nets theory and modeling of systems, Prentice-Hall, (1981).

[RAMA 82] Ramamoorthy, C. V., and Dong, S. T., "Communication protocol synthesis", IEEE Proc., COMSAC 82, (Nov. 1982), pp. 217-225.

[RAMA 85] Ramamoorthy, C.V., Dong, S. T. and Usuda, Y., 'An implementation of an automated protocol synthesizer (APS) and its application to the X.21 protocol', IEEE Trans. on Software Engineering, Vol. SE-11, NO. 9, (Sept. 1985), pp. 886-908.

[RAYN 82] Rayner, D., 'A system for testing protocol implementations', Protocol Specification, Testing and Verification, (ed. Sunshine C.), North-Holland, (1982), pp. 539-553.

[RUDI 78] Rudin, H., West, C., and Zafiropulo, P., 'Automated protocol validation: one chain of development', Computer Networks, Vol. 2, No. 4/5, (Oct. 1978), pp. 373-380.

[SARI 82] Sarikaya, B. and Bochmann, G.V., 'Some experience with test sequence generation for protocols', Protocol Specification, Testing and Verification, (ed. Sunshine C.), North-Holland, (1982), pp. 555-567.

[SCHW 82] Schwartz, R.L and Melliar-Smith, P.M., 'From state machine to temporal logic: specifications methods for protocol standards', IEEE Trans. on Communications, Vol COM-30, No. 12, (Dec. 1982), pp. 2486-2496.

[SHER 82] Sherman, M. and Rudin, H., 'Using automated validation techniques to detect lockups in packet-switched networks', IEEE Trans. on Communications, Vol. COM-30, No. 7, (July 1982), pp. 1762-1767.

[SIDH 82a] Sidhu, D.P., 'Rules for synthesizing correct communication protocols', ACM SIGCOMM Comput. Commun., Rev., Vol. 12, No. 1, (Jan. 1982), pp. 35-51.

[SIDH 82b] Sidhu, D.P., 'Protocol design rules', Protocol Specification, Testing and Verification, (ed. Sunshine C.), North-Holland, (1982), pp. 283-300.

[STEN 76] Stenning, N.V., 'A data transfer protocol', Computer Networks, Vol. 12, (Sept. 1976), pp. 99-110.

[URAL 84] Ural, H. and Probert, R.L., 'Systematic and selective generation of test sequences', Technical Report TR-84-16, Dept. of Computer Science, Univ. of Ottawa, (1984).

[VUON 81] Vuong, S. T. and Cowan, D. D., 'Automated protocol validation via resynthesis: the CCITT X.75 packet level recommendation as an example', Research Report CS 80-39, Dept. of Computer Science, Univ. of Waterloo, (May 1981).

[VUON 86] Vuong, S. T., Hui, D. D. and Cowan, D. D., 'Valira - a tool for protocol validation via reachability analysis', 6th International Workshop on Protocol Specification, Testing and Verification, Montreal, Grey Rocks inn., (June 1986), pp. 2/26-2/39.

[WEST 78] West, C.H. and Zafiropulo, P., 'Automated validation of a communication protocol: the CCITT X.21 recommendation', IBM Journal of Research and Development, Vol. 22, (Jan. 1978), pp. 60-71.

[ZAFI 80] Zafiropulo, p.,, 'Towards analyzing and synthesizing protocols', IEEE Trans. on Communications, Vol. COM-28, No. 4, (Apri. 1980), pp. 651-661.

[ZIMM 80] Zimmermann, H., 'OSI reference model- The ISO model of architecture for open systems interconnections', IEEE Trans. on Communications, Vol. COM-28, No. 4, (Apri. 1980), pp. 651-661.

Appendix A

A USER'S GUIDE TO GSAPS

A.1 Introduction

GSAPS - the Graphic System for Automating Protocol Synthesizer PNPS is a menu-driven graphic system for creating the peer protocol entity from a given local protocol entity. The principle and computational details are described in Chapter 3. This appendix describes GSAPS at the operational level. Section A.2 describes its hardware and software requirements. Section A.3 explains how it can be started and Section A.4 describes its functions.

A.2 Hardware/Software Requirements of GSAPS

GSAPS is a Pascal application program to be executed on an Apple MacIntosh microcomputer. A memory of at least 512 Kbytes is needed.

The software required includes the MacIntosh Pascal interpreter, the built-in graphic procedures: quickdraw1 and quickdraw2, and MacPaint.

A.3 Starting GSAPS

To start GSAPS, the user first opens the file containing the GSAPS

program by clicking twice at its icon and then selects the entry GO from the menu RUN. The GSAPS program will then be compiled and a welcome message appears on the screen (Figure A.1).

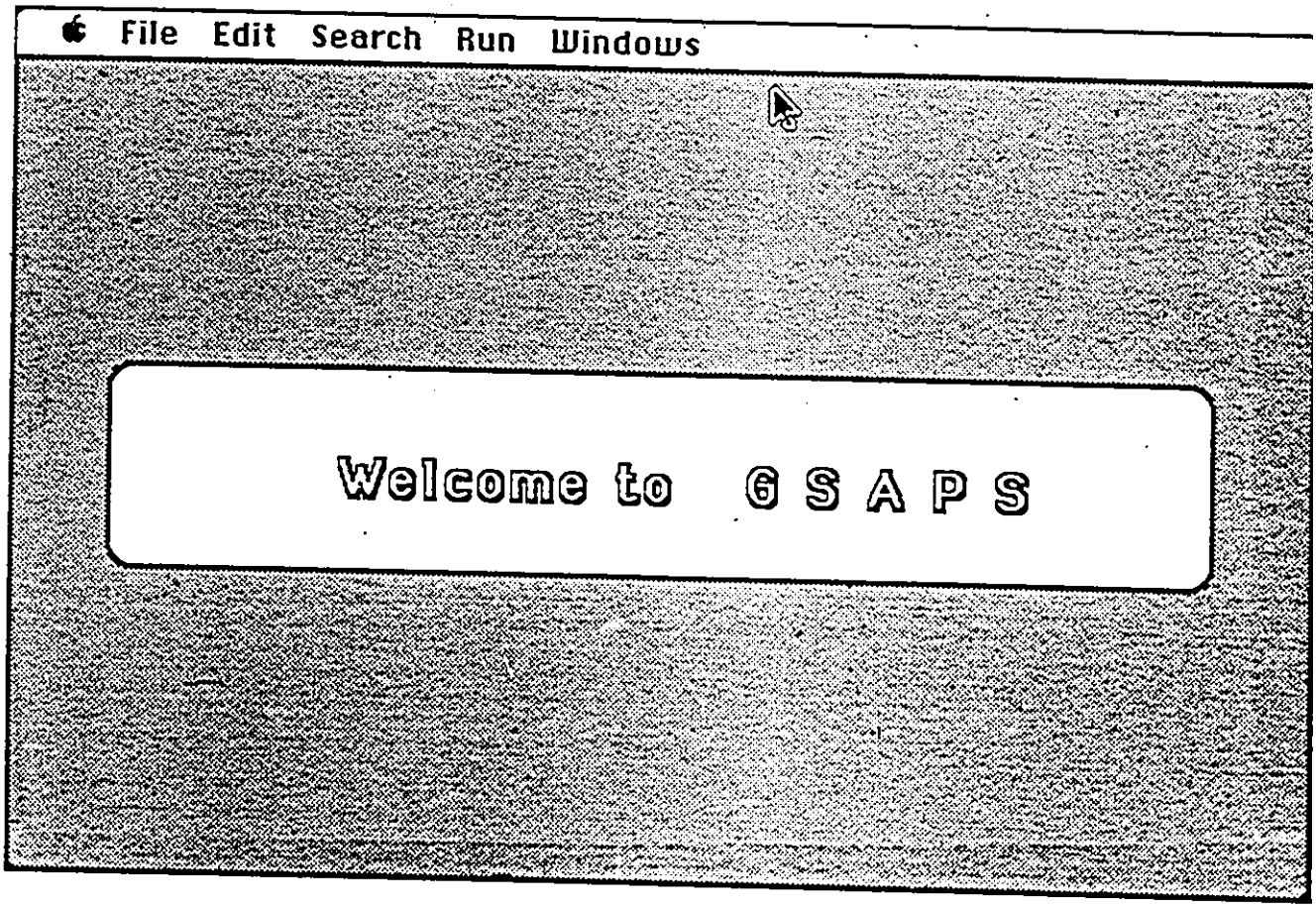


Figure A.1 The welcome message of GSAPS

GSAPS then shows (Figure A.2) a list of existing files in the disk. The user either selects one of the Petri net files by clicking its name twice, or starts a new problem by clicking CANCEL.

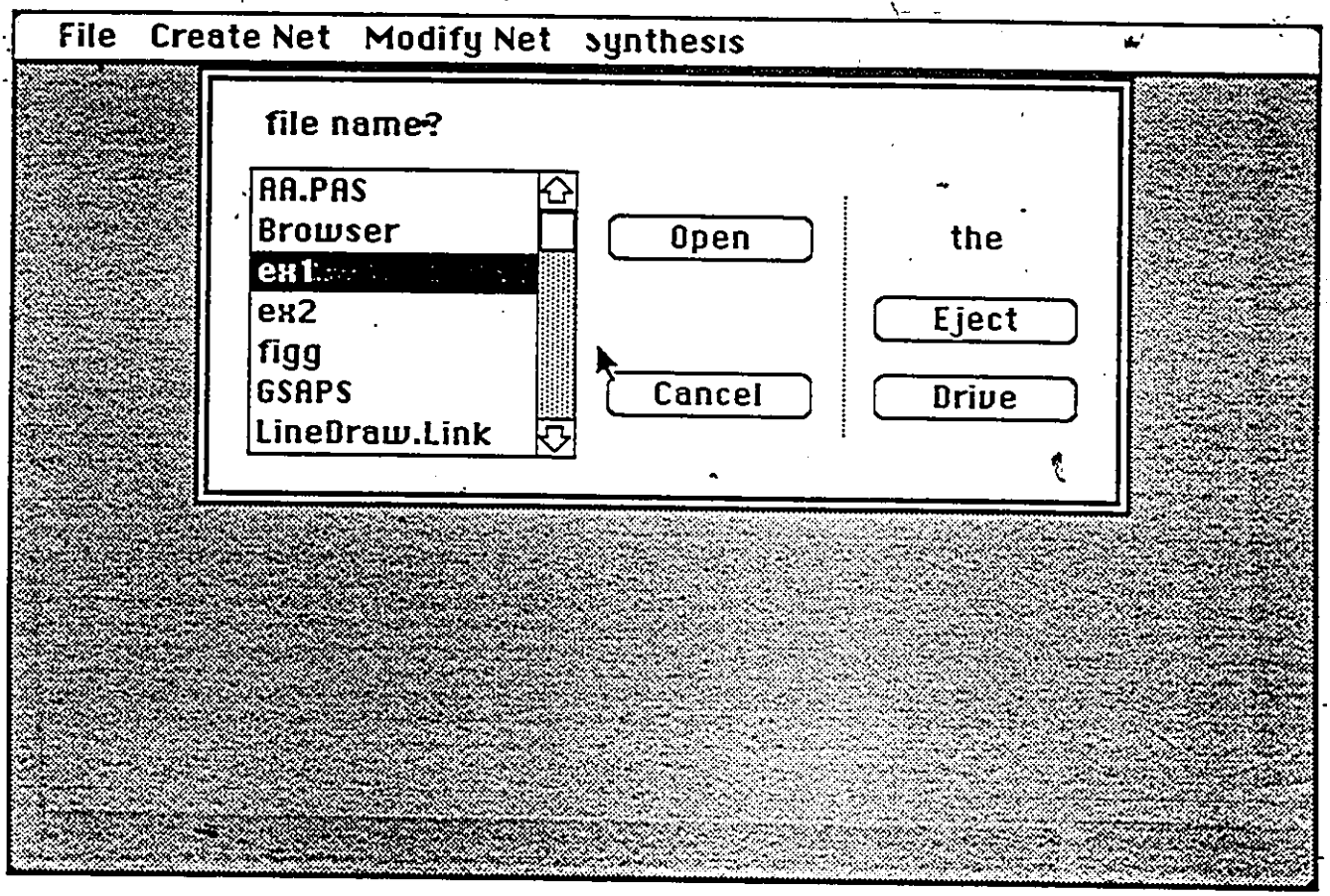


Figure A.2 The option of opening an existing or a new file

If an existing Petri net file is selected, GSAPS displays its graphic on the screen. The user may modify the local Petri net file, call synthesis procedures to design the peer Petri net and quit the sustem by selecting the appropriate entry from the menus.

If a new design is chosen, GSAPS opens a small window where the user is asked to enter values for the parameters: the maximum number of

transitions, external input places and external output places for the local Petri net (Figure A.3). The total of maximums can not exceed 80. After entering a value, the user hits RETURN. After these values have been entered, GSAPS clears the screen (Figure A.4). The user then starts the design of the local Petri net graphically by choosing the appropriate entry from the menus.

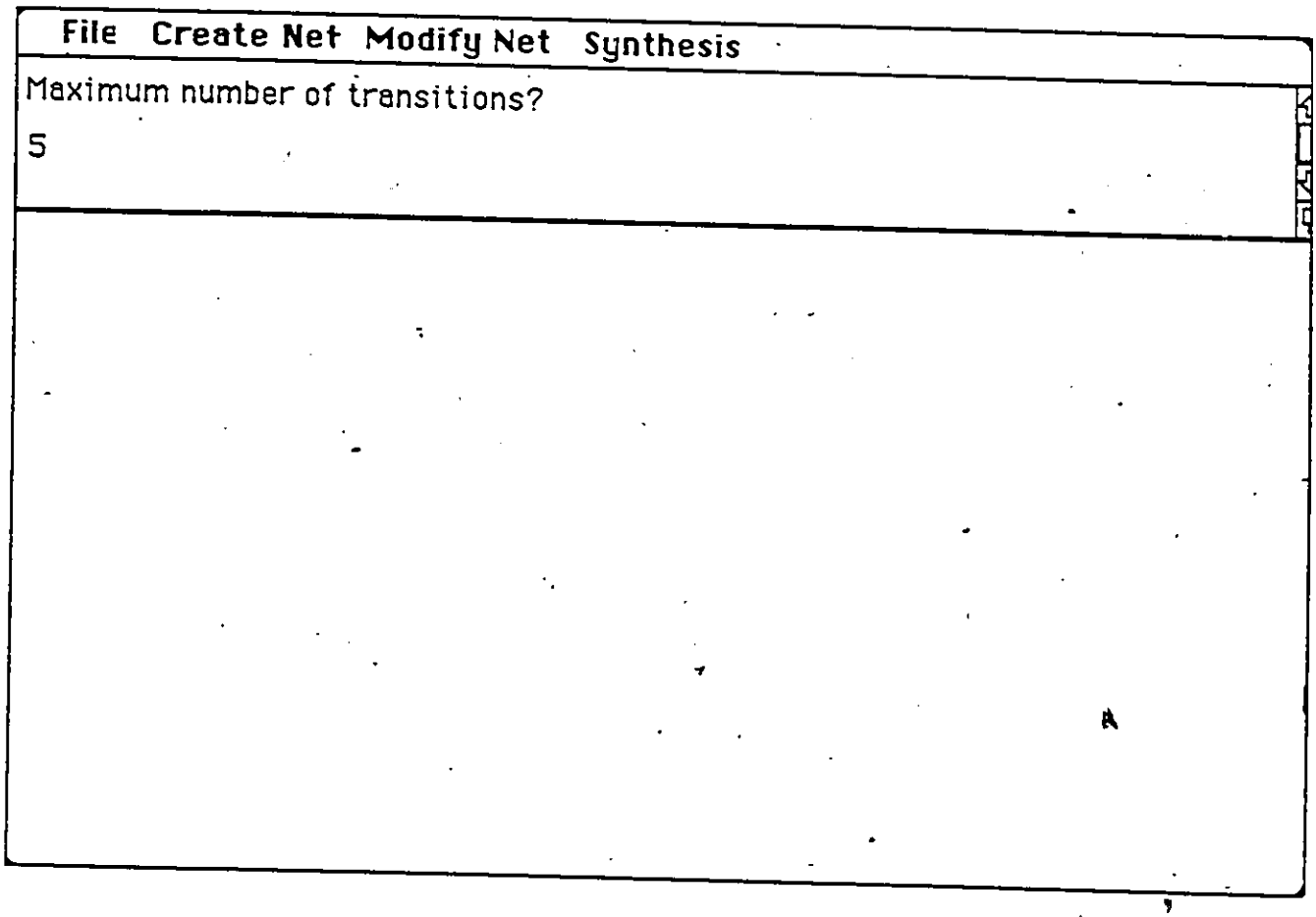


Figure A.3 The window for reading the parameters

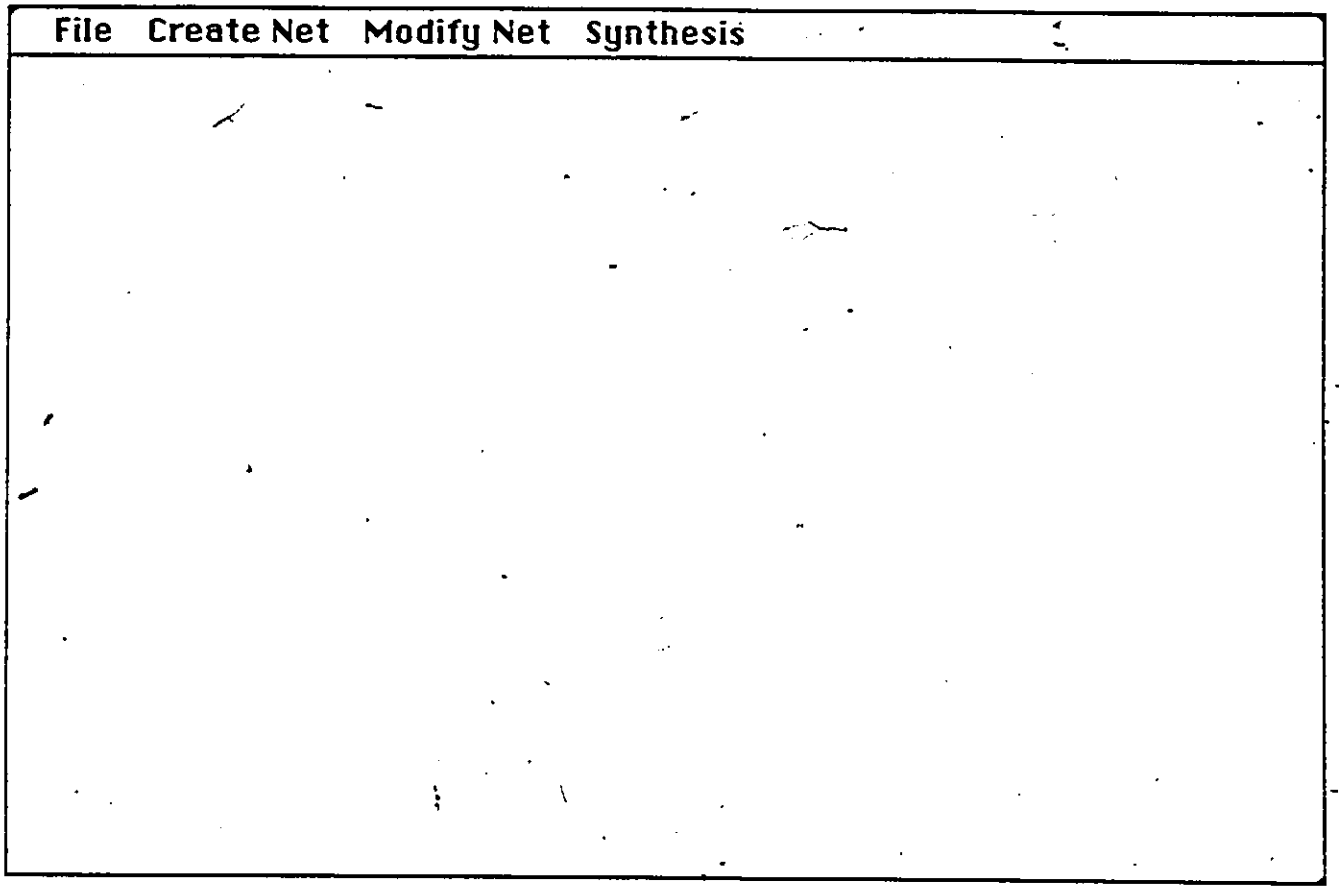


Figure A.4 The cleared screen and the menu for graphical design

A.4 Functions and User/System Interactions

This section describes in details the user-system interactions for each of the menus: File, Create Net, Modify Net and Synthesis. The user can create, modify Local Petri nets and call procedures for designing peer Petri nets.

A.4.1 Menu heading: File

Under this menu, the user has the options: start a new problem, work on an existing problem, save the data and save the drawing, and discontinue the execution of GSAPS.

Menu entries:

1. New
2. Open
3. Save: for GSAPS
4. Save: for MacPaint
5. Quit

* **New:** If the user wants to solve a new problem, the entry New has to be selected. The maximum number of transitions, external input places and external output places have to be entered . The screen is cleared and it is ready for the new local Petri net design. If a file has already been opened, the system asks whether it should be saved before starting a new one.

* **Open:** When this entry is selected, the window shown in Figure A.2 appears on the screen. If a file has already been opened, the system asks whether it should be saved before opening a new one. This entry gives the option of opening an existing Petri net file.The user chooses the file by clicking its name, and then the box OPEN.

* **Save: for GSAPS** When this entry is selected, all information and internally created data structures of the Petri net representing the local protocol entity will be stored in the disk for a future use. After choosing this entry from the menu File, the system prompts for the name to be given to this file. After selecting the name, the user should click the box SAVE. Figure A.5 shows how to save a file.

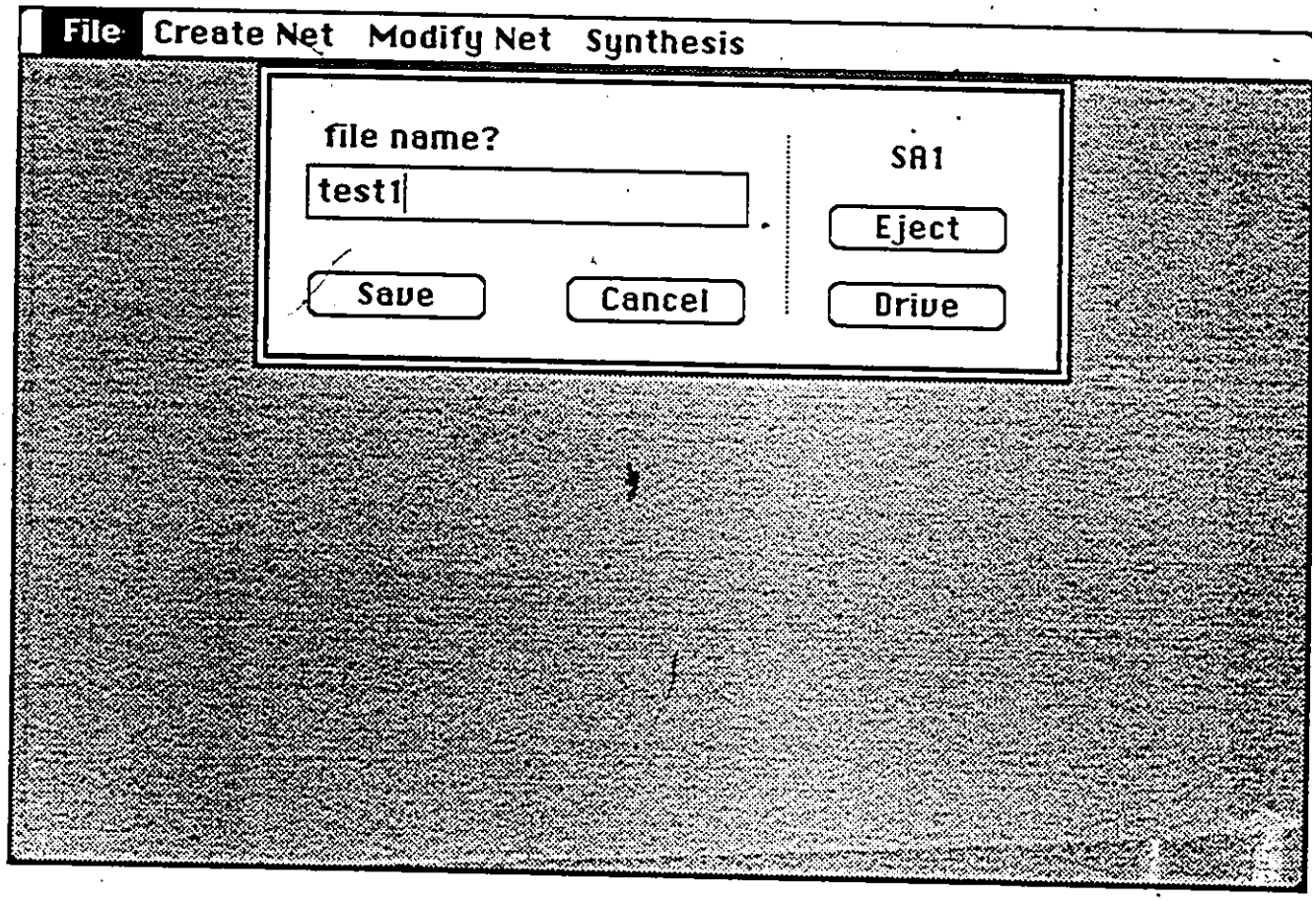


Figure A.5 Saving a file For GSAPS

* **Save: for MacPaint** This entry saves the graphics being displayed in

the screen in a file named "DRAWING". The MacPaint system should be used to open this file or to print it.

* **Quit:** This entry is selected to discontinue the execution of GSAPS. If a file is being displayed on the screen, the system asks whether it should be saved before quitting. Figure A.6 shows the screen after the user selects the quit entry from the menu File. The MacIntosh Pascal menu bar

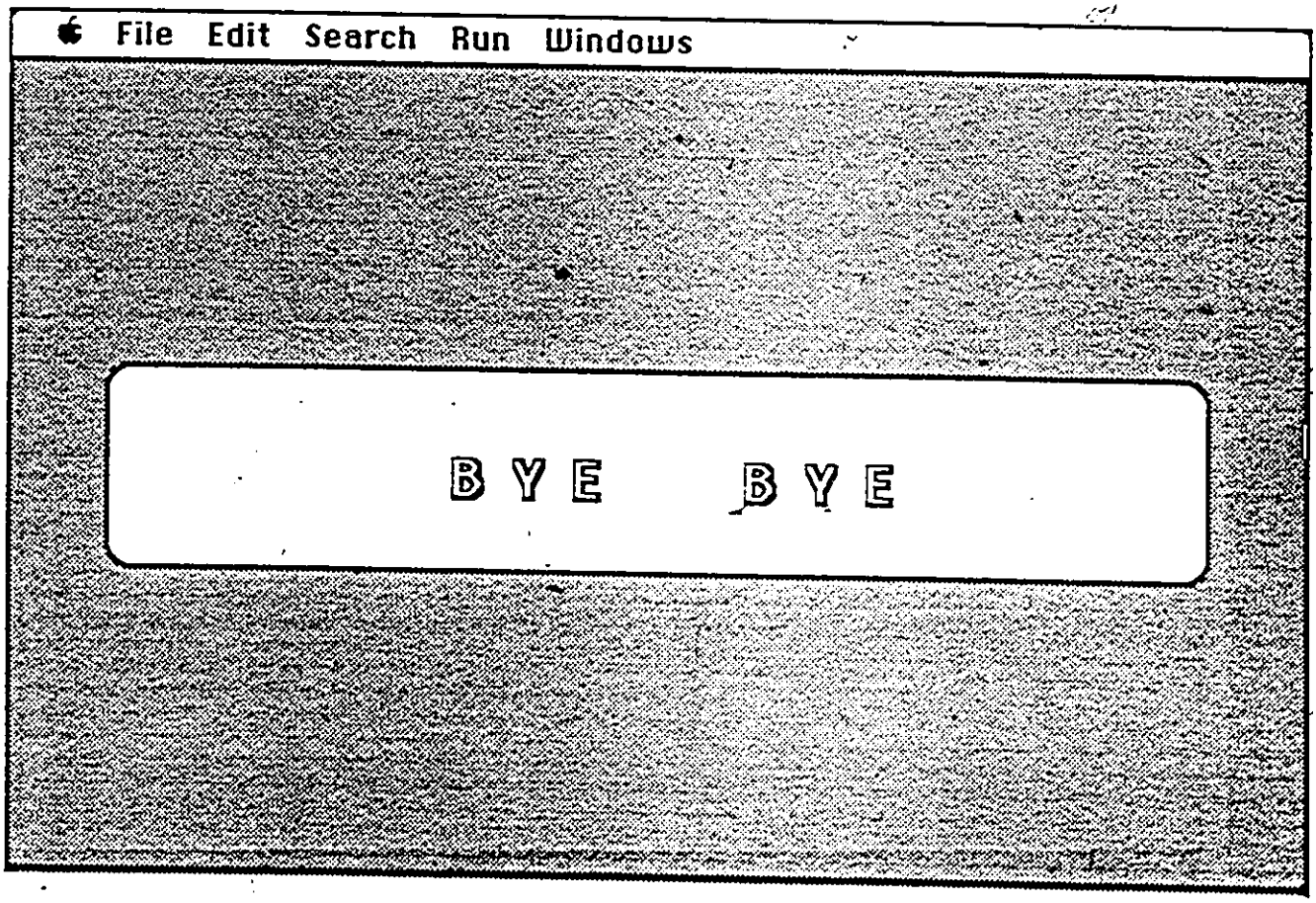


Figure A.6 The quiting message of GSAPS

A.4.2 Menu heading: Create Net

This menu contains the options of creating places, transitions, arcs, names and tokens. The graphics of these items are displayed on the screen.

Menu entries:

1. Int-place
2. Ext-input-place
3. Ext-output-place
4. Transition
5. Arc
6. Name
7. Token

* **Int-place:** This entry is used to create the internal places for the Petri net representing the local protocol entity. These places can be used as input or output places to the transitions in Petri net.

After selecting this entry, each time the user clicks the mouse a blank circle will be drawn on the screen at the location where the arrow is pointing to. The user can draw as many internal places as he wants without exceeding the specified maximum number which is 50 places. Figure A.7 shows the screen after the user creates some internal places.

* **Ext-input-place:** Each time the user clicks in the screen a black circle appears. These external input places are used only as inputs for the transitions in the local Petri net model. Figure A.8 shows the screen after creating some external input places.

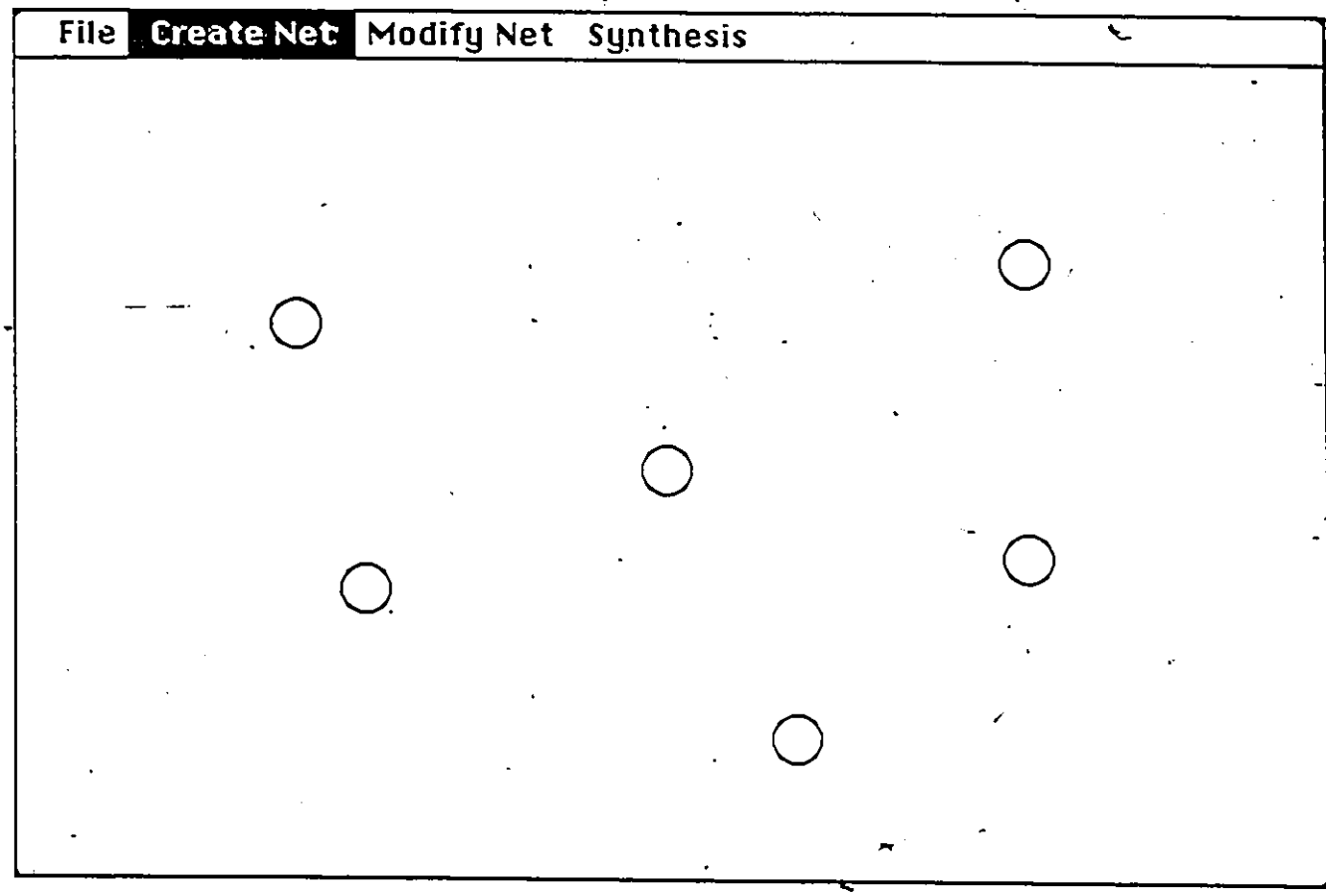


Figure A.7 An example of internal places

* **Ext-output-place:** Similarly, external output places are created. they are black circles with a small wight circle in the centre. (See Figure A.9)

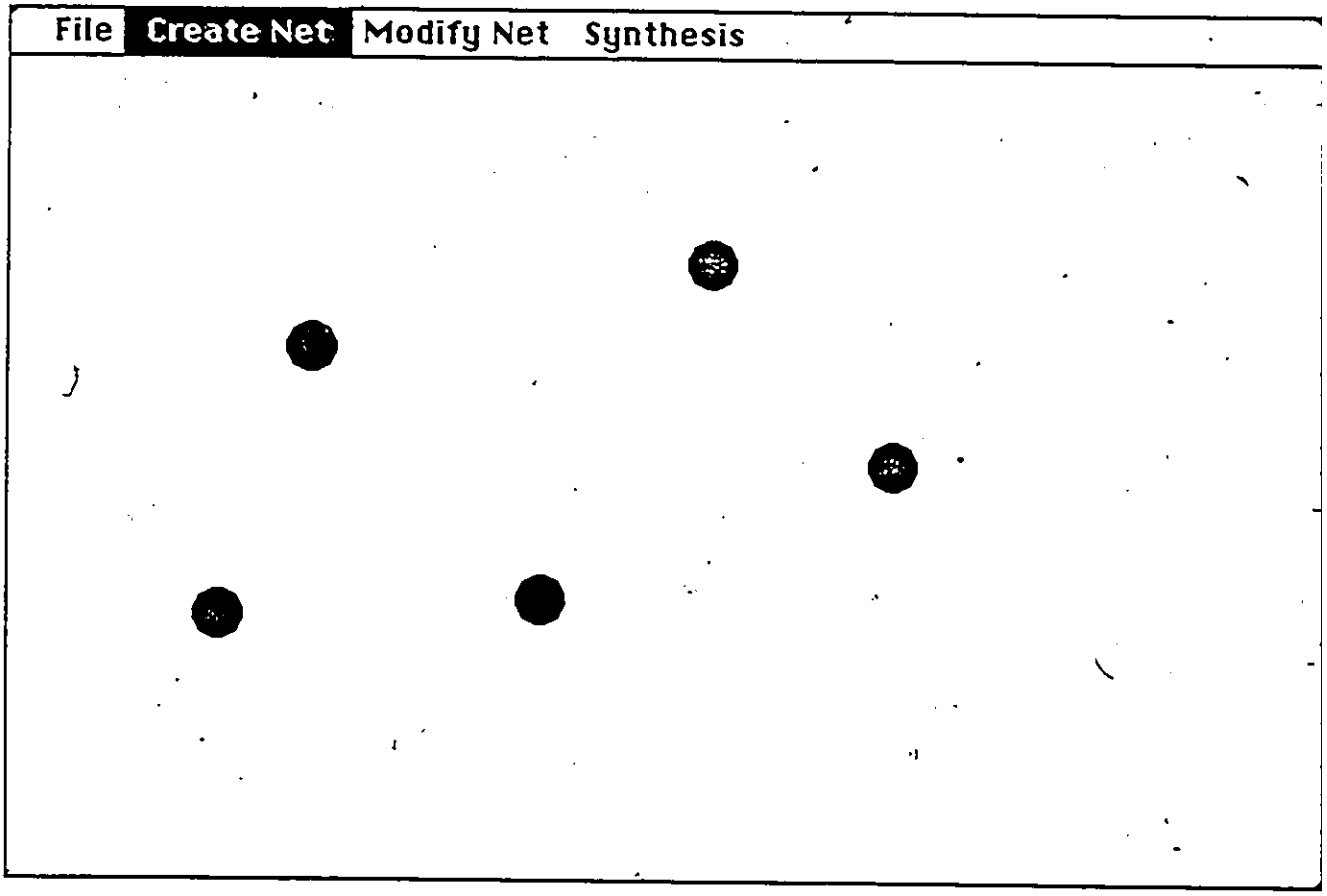


Figure A.8 An example of external input places

* **Transition:** When the user selects this item from the menu Create Net, each time he clicks on the screen, a black bar will appear. Figure A.10 shows the screen after creating some transitions to the local Petri net.

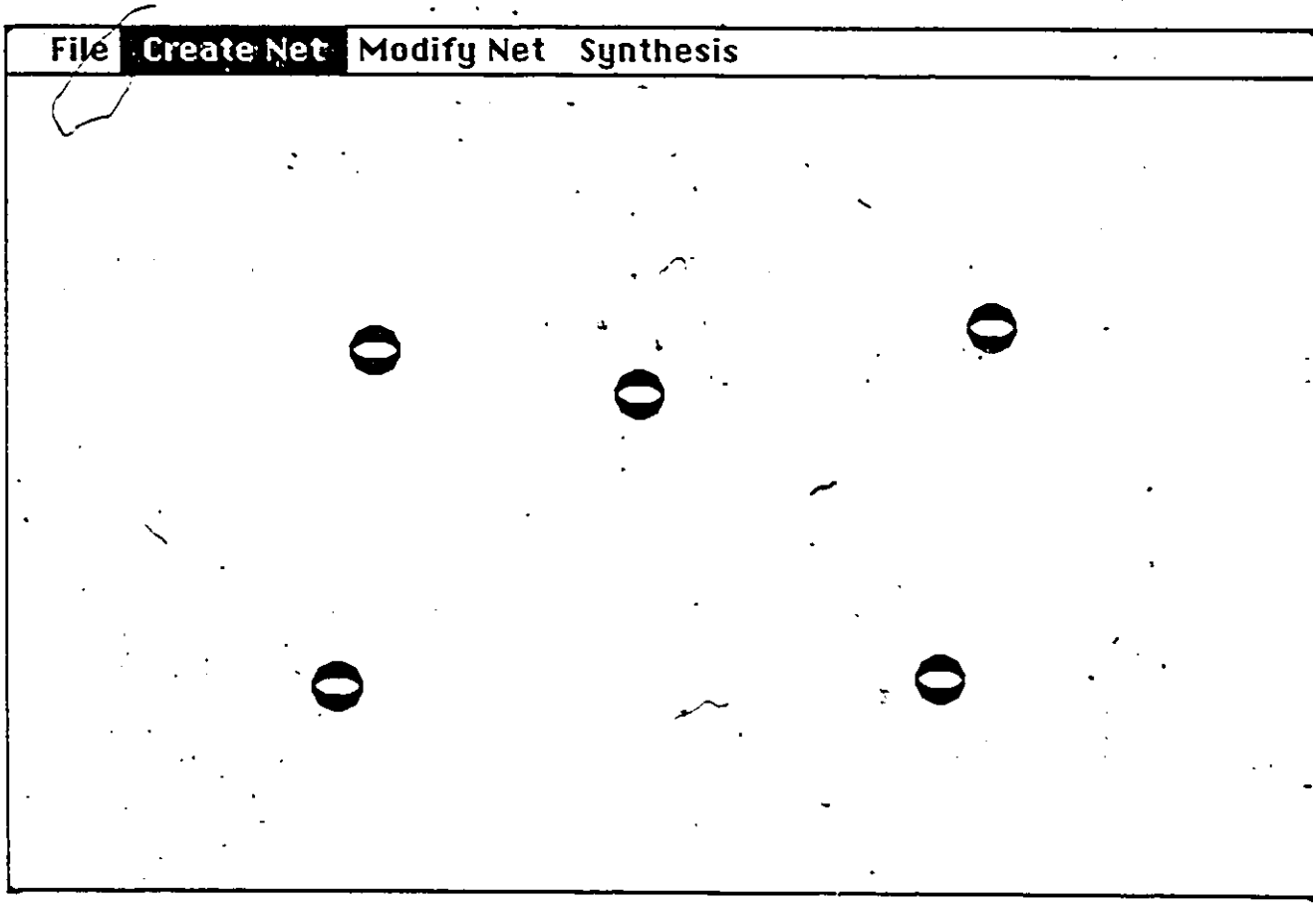


Figure A.9 An example of external output places

* **Arc:** This entry is used to connect internal places, external input places and external output places to transitions in the local Petri net. Each transition is allowed to have only one internal place and one external input place as inputs, and one internal place as output or one internal place as input, and one internal place, one external output place as outputs. To add an arc from an internal place P to a transition T the user has to click first the place P, then click the transition T. (See Figure A.11)

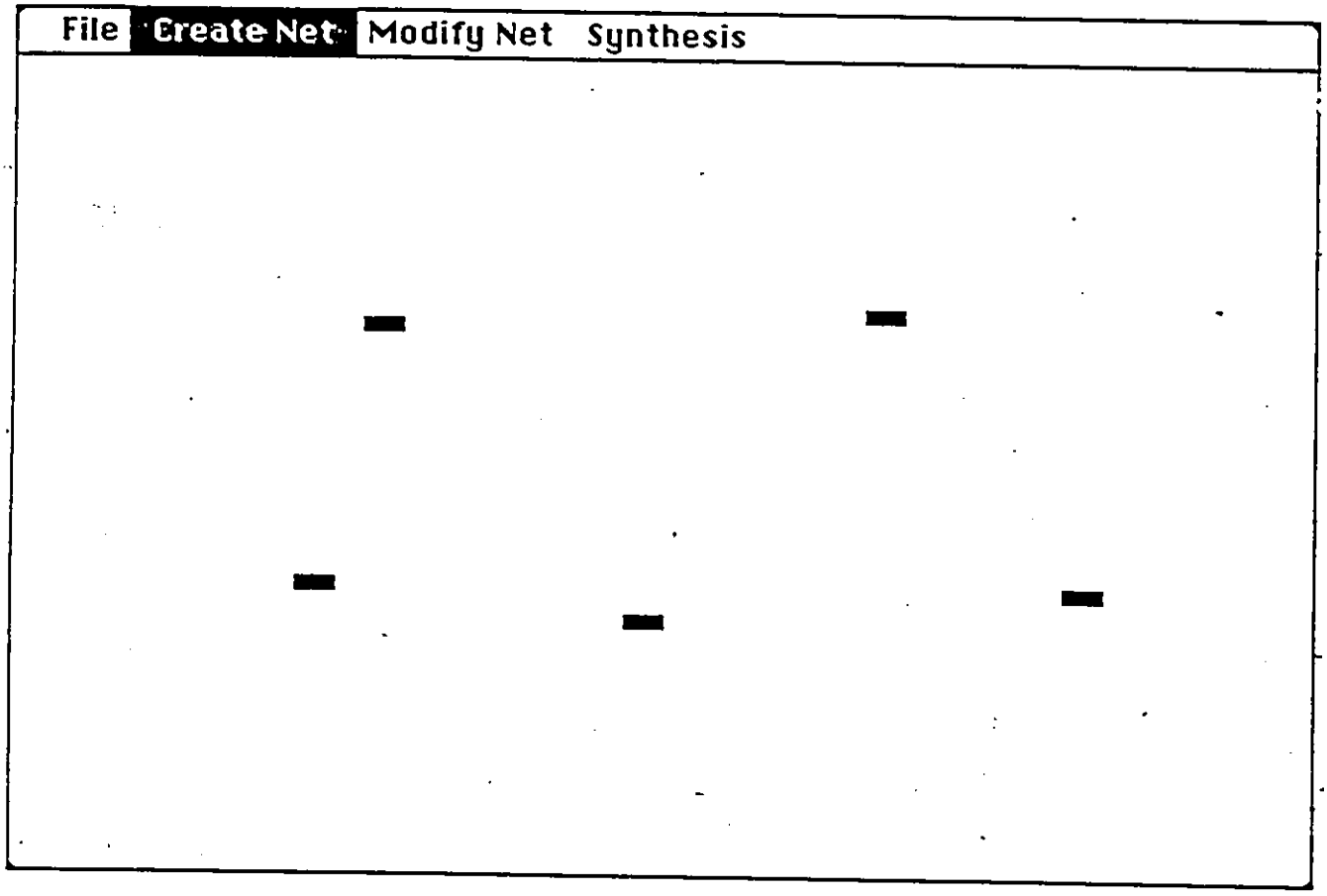


Figure A.10 An example of transitions

* **Name:** This entry is used to assign names to the internal places, external input places, external output places and transitions. After selecting this entry, the user clicks the items he wants to give them names, once at a time, then a small window will appear on the top of the screen asking to enter the name for that item, followed by a hit RETURN (Figure A.12). A

user may quit the naming session by selecting any other entry. But, the small window will not desappear unless the user selects the Draw local entry from the menu Modify Net or selects any other entry from the menu File.

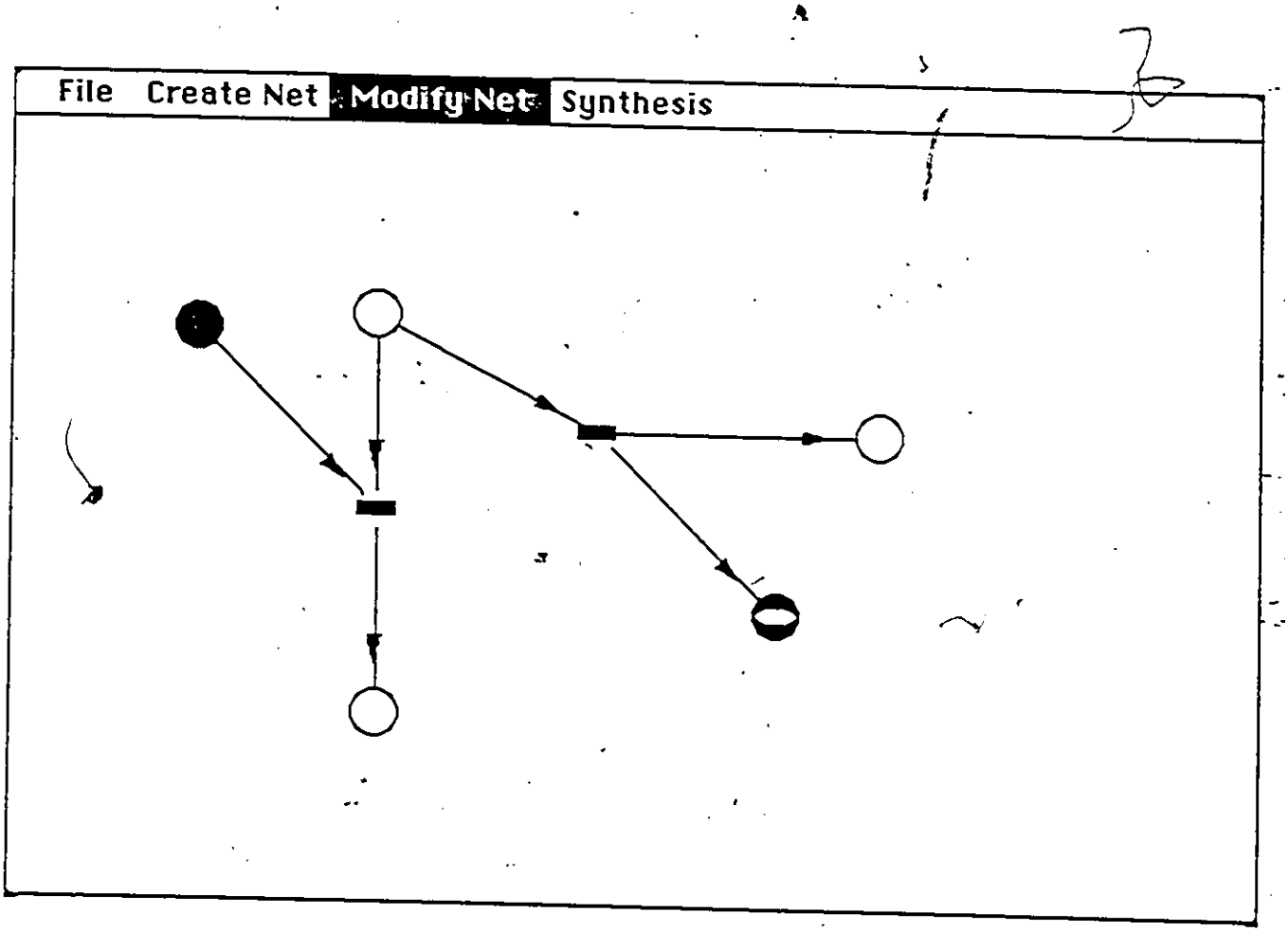


Figure A.11 An example of arcs connecting places and transitions

* **Token:** This entry is used to put tokens in internal places. After selecting this entry the user clicks the internal place he wants to put a token on it. When the small window is displayed on the top of the screen, the user specifies how many tokens goes into that place followed by a hit RETURN. A 0 is displayed in places with no token, when the user asks to see how many tokens are there in each internal place by selecting the entry Display token in the menu Modify Net.

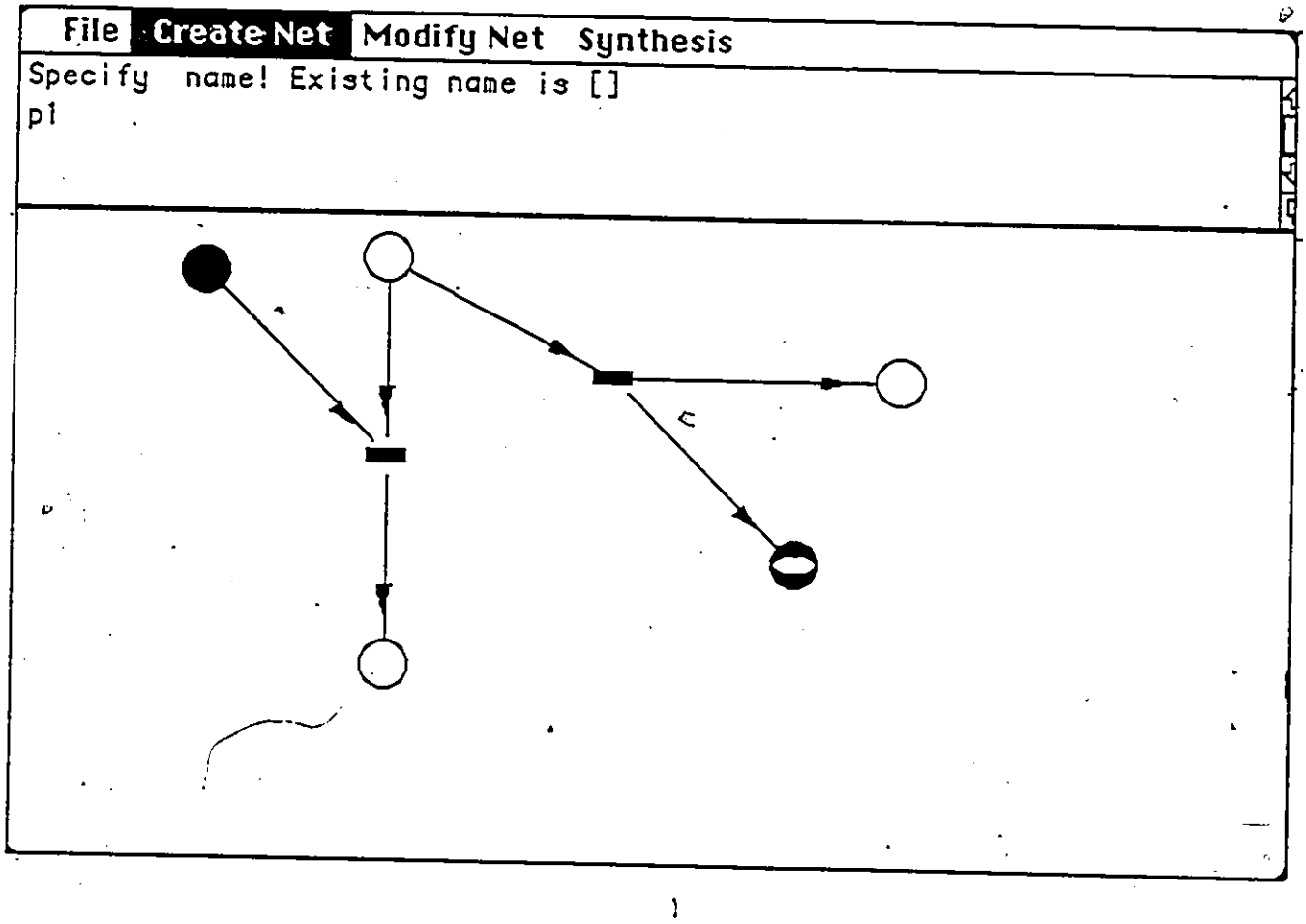


Figure A.12 A window for reading names of places and transitions

A.4.3 Menu heading: Modify Net

This menu contains the options for modification the local Petri net, such as deleting some places, transitions, arcs or moving these items to other locations in the screen. Other functions are: displaying the graphics for the local Petri net graphics with or without names of the items and the number of tokens in each internal place.

Menu entries:

1. Delete node
2. Delete arc
3. Move node
4. Draw local
5. Display or hide name
6. Display or hide token

* **Delete node:** This entry deletes internal places, external input places, external output places and transitions. The user has just to click the node he wants to delete and everything else is taking care of it by GSAPS. The deleted node is disappeared from the screen and from the data structure of that Petri net.

* **Delete arc:** If by mistake, the user creates a wrong arc in connecting transitions to places and vice verca, he still has the chance to correct by choosing the Delete arc entry from the Modify net menu. An arc from a

transition t and a place p, is deleted by first clicking the transition t, then clicking the place p.

* **Move node:** If the graph displayed on the screen does not look nice because of the position of its nodes, the user still have the chance to move those places to the positions he wants by choosing the entry Move node, clicking, moving and releasing the node in the right position.

* **Draw local:** This entry displays on the screen the graph of the Petri net representing the local protocol entity. It is needed specially after adding names or tokens and a part of the graphics is hided by the small window displayed on the top of the screen created by the selection of these entries.

* **Display or hide name:** This entry displays on the screen the names of places and transitions if they were hidden and hides them if they were displayed.

* **Display or hide token:** This entry displays on the screen the number of tokens in each internal place if they were hidden and hides them if they were displayed.

A.4.4 Menu heading: Synthesis

This menu contains the options of applying the rules for designing the peer Petri net from the given local Petri net, then the graphics of the former one or both graphics can be displayed on the screen by selecting the corresponding entries from the menu.

Menu Entries:

1. Apply Rule R1
2. Apply Rule R2
3. Draw peer
4. Draw both

* **Apply Rule R1:** This entry is used to start the internal design of the Petri net representing the peer protocol entity by calling the procedures for applying STEP II of the presented algorithm in chapter 3 of this work. STEP I of the algorithm was done automatically by GSAPS at the same time the user is adding arcs between transitions and places. All informations are stored into local matrices. After selecting Apply Rule R1, the peer matrices are initialized.

* **Apply Rule R2:** This entry is called to continue the internal design of the Petri net representing the peer protocol entity by executing the procedures for applying STEP III of the algorithm presented in chapter 3. After selecting the Apply Rule R2 entry the synthesis of the peer protocol

entity is completed and its Petri net model can be displayed using the entry Print peer from the same menu.

* **Draw peer:** After selecting Apply Rule R1, the user can choose the entry Draw Peer to see the partial result of the design process by displaying the partial peer Petri net model. the user can see the complete peer Petri net graphic on the screen by selecting both entries: Apply Rule R1 and then Apply Rule R2 before selecting Draw peer.

* **Draw both:** This entry displays both graphics: the local and peer Petri nets on the same screen. These graphics can be saved using the entry Save Drawing from the menu File. The saved file can be opened using the MacPaint system.

Appendix B

PROGRAM LISTING

```
(*****************************************************************************)
(*                                                                           *)
(* NAME :     GSAPS - a Graphical System for Automating a Proto-             *)
(*            col Synthesizer.                                               *)
(*                                                                           *)
(* FUNCTION   This is a software package for network protocol syn-           *)
(*            thesis. It gets its inputs the Petri net graphical repre-      *)
(*            sentation of a local protocol entity interactively. After      *)
(*            executing the steps of the synthesizer PNPS, the Petri         *)
(*            net graphical representation of its peer entity is cra-        *)
(*            ted. The graphical representations of both entities can        *)
(*            be displayed separately or simultanously.                      *)
(*                                                                           *)
(* Note:      As it has been mentioned in Section 5.2, the MacIntosh         *)
(*            Pascal interpreter does not accept long programs, this         *)
(*            file cannot be executed because of the inserted com-           *)
(*            ments. GSAPS itself does not contain any comments.             *)
(*                                                                           *)
(*****************************************************************************)

  program  graphsyn;

(*****************************************************************************)
(*                                                                           *)
(*  Some graphical functions and procedures are called by this program       *)
(*  from the available libraries quickdraw1 and quickdraw2.                  *)
(*                                                                           *)
(*****************************************************************************)

  uses
```

```
quickdraw1,  quickdraw2;
const
 max = 80; (*  the total maximum number of external input places,   *)
           (*  external output places and transitions should not    *)
           (* exceed 80                                             *)
TT = true;
FF = false;
type
ar = array[1..50, 1..max] of integer;
Ptr = ^LongInt;
Handle = ^Ptr;
WindowRecord = array[1..78] of integer;
WindowPtr = ^WindowRecord;
EventRecord = record
           What : Integer;
           Message : LongInt;
           When : LongInt;
           Where : LongInt;
           Modifiers : Integer
end;
str24 = string[24];

(*************************************************************************)
(*                                                                       *)
(*  This record net is used to keep track on the information about a Pe- *)
(* tri net. The fields of this record are explained as follows:          *)
(*                                                                       *)
(* p,t,m,j : number of internal places, transitions, external input places *)
(*         and external output places in the local Petri net, respecti- *)
(*         vely.                                                         *)
(*  maxp, maxt, MAXM, MAXJ : maximum number of the above elements*)
(*         respectively.                                                 *)
(* name:   array for storing the names of the above elements, respecti- *)
(*         vely.                                                         *)
(*  l : array for storing the coodinates of each element of the above    *)
(*         elements, respectively.                                       *)
(*  token : array for storing the distribution of tokens among the inter- *)
```

```
(*          nal places.                                                    *)
(*  LL, LLP : arrays for storing arcs from places to transitions and from  *)
(*          trnsitions to places, respectively.                            *)
(*                                                                         *)
(*************************************************************************)

    net = record
          p, t, m, j, maxp, maxt, MAXM, MAXJ : integer;
          name : array[-50..max] of string[16];
          l : array[-50..max] of point;
          token : array[-50..max] of integer;
          LL, LLP : ar;
    end;
    var
    nshow, tshow, netChanged, Done, inbar, first : boolean;
    Event : EventRecord;
    psize, tsize, a, b, ai, bj, delay, beta, segma : integer;
    alpha : real;
    oldmenu, newmenu : longint;
    whichwindow : windowptr;
    z, zp : net;
    c : point;
    r : rect;
    filevar : file of net;
    printer : text;
    OldMenuBar, filemenu, createmenu, modifynaMenu, parameterMenu :
    Handle;

(*************************************************************************)
(*                                                                       *)
(*  this procedure prints appropriate messages such as welcoming a       *)
(*  message. Messages length cannot exceed 24 characters.                *)
(*                                                                       *)
(*************************************************************************)

    procedure pmsg (P : point;
          m, con, ele : str24);
```

```
    var
        h, v, size : integer;
    begin
    if ele = 'p' then
    size := psize + 2;
    if ele = 't' then
    size := tsize;
    h := p.h - stringwidth(m) div 2;
    if con = 'n' then
    v := p.v + size + 8
    else if con = 'p' then
    v := p.v - size
    else if con = 'a' then
     v := p.v - size - 14;
    moveto(h, v);
    textsize(12);
    drawstring(m);
    end;

(*****************************************************************************)
(*                                                                           *)
(*    This procedure prepares a window on the screen having the speci-       *)
(*   fied dimensions for drawing purposes.                                   *)
(*                                                                           *)
(*****************************************************************************)

    procedure  setdrawing;
    begin
    hideall;
    setrect(R, 0, 20, 527, 357);
    setdrawingrect(R);
    showdrawing;
    end;

(*****************************************************************************)
(*                                                                           *)
(*   This procedure draws directed arcs from places and transitions or       *)
```

```
(*  from transitions to places.                                              *)
(*                                                                           *)
(*****************************************************************************)

    procedure fillarrow (f, t : point;
         radf, radt : integer);
    var
         x, y, angle : integer;
         r : rect;
    begin
    setrect(r, t.h - 12, t.v - 12, t.h + 12, t.v + 12);
    pttoangle(r, f, angle);
    y := round(radt * cos(angle * 0.0175));
    x := round(radt * sin(angle * 0.0175));
    f.v := f.v - round(radf * cos((angle - 180) * 0.0175));
    f.h := f.h + round(radf * sin((angle - 180) * 0.0175));
    moveto(f.h, f.v);
    lineto(t.h + x, t.v - y);
    setrect(r, t.h + 2 * x - 12, t.v - 2 * y - 12, t.h + 2 * x + 12, t.v - 2 * y +
    12);
    paintarc(r, angle - 15, 30);
    end;

(*****************************************************************************)
(*                                                                           *)
(*   This function prepares the dimensions of the places and transitions     *)
(*   in the local entity based on the passed coordinates of these items      *)
(*                                                                           *)
(*****************************************************************************)

    function prect (p : point;
         n : integer;
         k : real) : rect;
    var
         r : rect;
    begin
    if n = 1 then
```

```
    setrect(r, p.h - round(psize * k), p.v - psize, p.h + round(psize * k),
    p.v + psize)
    else if n = 2 then
    setrect(r, p.h - round(tsize * k), p.v - round(tsize * 0.4), p.h +
    round(tsize * k), p.v + round(tsize * 0.4));
    prect := r;
    end;

(*****************************************************************************)
(*                                                                           *)
(*   This function tests for the position of the mouse whether it is on the  *)
(*    menu bar or not. It returns the boolean value for the variable bar     *)
(*                                                                           *)
(*****************************************************************************)

    function testmouse (var bar : boolean) : boolean;
    begin
    testmouse := FF;
    bar := FF;
    repeat
    until BinLineF(SA970, S017F, @Event);
    if Event.what = 1 then
            begin
            testmouse := TT;
             if WinLineF(SA92C, Event.where, @whichwindow) = 1 then
            bar := TT;
            end;
    end;

(*****************************************************************************)
(*                                                                           *)
(*  This procedure prepares a small window for displaying the text of a      *)
(*  message or other text.                                                   *)
(*                                                                           *)
(*****************************************************************************)

    procedure  settext;
```

```
begin
setrect(r, 0, 20, 521, 80);
settextrect(r);
showtext;
end;

(*****************************************************************************)
(*                                                                           *)
(*  This procedure sets the maximum number of internal input places          *)
(*  to 50. It reads the maxmum number of transitions, external input         *)
(*  places and external output places, respectively. It clears all matri-    *)
(* ces of the new local entity.                                              *)
(*                                                                           *)
(*****************************************************************************)

procedure  emptynet;
var
        i, j : integer;
        initz : net;
        correct : boolean;
begin
correct := FF;
z := initz;
z.maxp := 50;
settext;
while not correct do
          begin
        writeln('Maximum number of transitions?');
        readln(z.maxt);
        writeln('Maximum number of external input places?');
        readln(z.MAXM);
        writeln('Maximum number of external output places?');
        readln(z.MAXJ);
        if (z.maxt + z.MAXM + z.MAXJ <= max) then
        correct := TT;
        end;
for i := 1 to 50 do
```

```
    for j := 1 to max do
            begin
            z.LL[i, j] := 0;
            z.LLP[i, j] := 0;
            end;
    end;

(*****************************************************************************)
(*                                                                           *)
(*  This function finds the index of an element and its type (internal       *)
(*  place, external input place, external output place or transition.        *)
(*                                                                           *)
(*****************************************************************************)

    function find (x, y, from, toindex, size : integer;
            var n : integer) : boolean;
    var
            continue : boolean;
    begin
    n := from;
    continue := TT;
    while (n <= toindex) and continue do
    if (sqr(x - z.l[n].h) + sqr(y - z.l[n].v) > size * size) then
    n := n + 1
    else
    continue := FF;
    find := not continue;
    end;

(*****************************************************************************)
(*                                                                           *)
(*  This function calls the function find to get the type of an element      *)
(*  and its position. Its gives each type a number to be used by other       *)
(*  procedures.                                                              *)
(*                                                                           *)
(*****************************************************************************)
```

```
function findtype (x, y : integer;
        var n : integer) : integer;
begin
if find(x, y, 1, z.t, tsize, n) then
findtype := 1
else if find(x, y, -z.p, -1, psize, n) then
findtype := 3
else if find(x, y, z.maxt + 1, z.maxt + z.m, psize, n) then
findtype := 7
else if find(x, y, z.maxt + z.MAXM + 1, z.maxt + z.MAXM + z.j, psize, n)
then
findtype := 15
else
findtype := 0;
end;

(*****************************************************************************)
(*                                                                           *)
(*  This procedure displays the names of places and transitions if they      *)
(*  hidden and hides them if they are displayed.                             *)
(*                                                                           *)
(*****************************************************************************)

procedure  displayname;
var
        i, j : integer;
begin
textface([bold]);
if ashow then
textmode(srcbic);
for i := 1 to z.p do
pmsg(z.l[-i], z.name[-i], 'n', 'p');
for i := 1 to z.t do
pmsg(z.l[i], z.name[i], 'n', 't');
for i := z.maxt + 1 to z.maxt + z.m do
pmsg(z.l[i], z.name[i], 'n', 'p');
for i := z.maxt + z.MAXM + 1 to z.maxt + z.MAXM + z.j do
```

```
    pmsg(z.l[i], z.name[i], 'n', 'p');
    textmode(srcor);
    nshow := not nshow;
    end;

(*****************************************************************************)
(*                                                                           *)
(*  This procedure displays the number of tokens in each internal place      *)
(*  in the Petri net if hidden and hides it if displayed.                    *)
(*                                                                           *)
(*****************************************************************************)

    procedure  displaytoken;
    var
            i : integer;
    begin
    textface([bold]);
    if tshow then
    textmode(srcbic);
    for i := 1 to z.p do
            begin
            moveto(z.l[-i].h - 6, z.l[-i].v + 4);
            drawstring(stringof(z.token[-i]  :  1));
            end;
    textmode(srcor);
    tshow := not tshow;
    end;
(*****************************************************************************)
(*                                                                           *)
(*  This procedure includes three procedures for deleting internal pla-      *)
(*  ces, external places and transitions in the local entity, respectively.  *)
(*  The arcs attached to the deleted nodes are also deleted. A global        *)
(*  parameter is passed to this procedure to decide which node to delete     *)
(*                                                                           *)
(*****************************************************************************)

    procedure  deletenode;
```

```
var
        i, q, qq, a, ai, k : integer;

(*****************************************************************************)
(*                                                                           *)
(*  This procedure delete a column from the the input matrix and a           *)
(*  corresponding column from the output matrix in the local entity.         *)
(*  Other columns in location, name matrices are also deleted. This          *)
(*  means that an external place and its corresponding arcs in the local     *)
(* entity will be deleted.                                                   *)
(*                                                                           *)
(*****************************************************************************)

        procedure deletecolio (f, t : integer;
        var q, qq : integer;
            notj : boolean);
        var
        i, k : integer;
        begin
        penmode(patbic);
        eraseoval(prect(z.l[f], 1, 1));
        for k := 1 to z.t do
        if (z.LL[k, q] = 1) then
            begin
            z.LL[k, q] := 0;
            fillarrow(z.l[f], z.l[k], psize, tsize);
            end;
        for k := 1 to z.t do
        if (z.LLP[k, q] = 1) then
            begin
            z.LLP[k, q] := 0;
            fillarrow(z.l[k], z.l[f], tsize, psize);
            end;
        for k := f to t - 1 do
            begin
            z.l[k].h := z.l[k + 1].h;
            z.l[k].v := z.l[k + 1].v;
```

```
            z.name[k] := z.name[k + 1];
            z.token[k] := z.token[k + 1];
            end;
        for i := 1 to z.t do
            begin
            for k := q to qq - 1 do
                begin
                z.LL[i, k] := z.LL[i, k + 1];
                z.LLP[i, k] := z.LLP[i, k + 1];
                end;
            z.LL[i, qq] := 0;
            z.LLP[i, qq] := 0;
            end;
        z.l[t].h := 0;
        z.l[t].v := 0;
        z.name[t] := '';
        z.token[t] := 0;
        pennormal;
        end;

(*****************************************************************************)
(*                                                                           *)
(*  This procedure delete a row from the the input matrix and a corres-      *)
(*  ponding row from the output matrix in the local entity. Other rows       *)
(*  in location, name matrices are also deleted. This is mean, it deletes    *)
(*  a transition and its corresponding arcs in the local entity.             *)
(*                                                                           *)
(*****************************************************************************)

        procedure deleterow (f, t : integer;
            notj : boolean);
        var
            q, k, i : integer;
        begin
        eraserect(prect(z.l[ai], 2, 1));
        penmode(patbic);
        for k := 1 to z.p do
```

```
        begin
        if (z.LL[ai, k] = 1) then
            begin
            z.LL[ai, k] := 0;
            fillarrow(z.l[-k], z.l[ai], psize, tsize);
            end;
        if (z.LLP[ai, k] = 1) and notj then
            begin
            z.LLP[ai, k] := 0;
            fillarrow(z.l[ai], z.l[-k], tsize, psize);
            end;
        end;
    for k := z.maxp + 1 to z.maxp + z.m do
        begin
        q := k - z.maxp + z.maxt;
        if (z.LL[ai, k] = 1) then
            begin
            z.LL[ai, k] := 0;
            fillarrow(z.l[q], z.l[ai], psize, tsize);
            end;
        end;
    for k := z.maxp + z.MAXM + 1 to z.maxp + z.MAXM + z.j do
        begin
        q := k - z.maxp + z.maxt;
        if (z.LLP[ai, k] = 1) and notj then
            begin
            z.LLP[ai, k] := 0;
            fillarrow(z.l[ai], z.l[q], tsize, psize);
            end;
        end;
    for k := f to t - 1 do
        begin
        z.l[k] := z.l[k + 1];
        z.name[k] := z.name[k + 1];
        for i := 1 to (z.maxp + z.MAXM + z.j) do
        z.LL[k, i] := z.LL[k + 1, i];
        for i := 1 to (z.maxp + z.MAXM + z.j) do
```

```
        z.LLP[k, i] := z.LLP[k + 1, i];
        end;
    z.l[t].h := 0;
    z.l[t].v := 0;
    z.name[t] := '';
    for i := 1 to (z.maxp + z.MAXM + z.j) do
    z.LL[t, i] := 0;
    for i := 1 to (z.maxp + z.MAXM + z.j) do
    z.LLP[t, i] := 0;
    pennormal;
    end;
    begin
    getmouse(c.h, c.v);
    a := findtype(c.h, c.v, ai);
    netchanged := TT;
    if nshow then
    displayname;
    if tshow then
    displaytoken;
    case a of
    3 :
```

```
(*****************************************************************************)
(*                                                                           *)
(*  This case deletes a column from the the input matrix and a               *)
(*   corresponding column from the output matrix in the local entity.        *)
(*    Other columns in location, name matrices are also deleted. This        *)
(*   means that an internal place and its corresponding arcs in the local    *)
(*  entity will be deleted.                                                  *)
(*                                                                           *)
(*****************************************************************************)
```

```
        begin
        penmode(patbic);
        eraseoval(prect(z.l[ai], 1, 1));
        for k := 1 to z.t do
        if (z.LL[k, -ai] = 1) then
```

```
            begin
            z.LL[k, -ai] := 0;
            fillarrow(z.l[ai], z.l[k], psize, tsize);
            end;
        for k := 1 to z.t do
        if (z.LLP[k, -ai] = 1) then
            begin
            z.LLP[k, -ai] := 0;
            fillarrow(z.l[k], z.l[ai], tsize, psize);
            end;
        for k := ai downto -z.p + 1 do
            begin
            z.l[k].h := z.l[k - 1].h;
            z.l[k].v := z.l[k - 1].v;
            z.name[k] := z.name[k - 1];
            z.token[k] := z.token[k - 1];
            end;
        for i := 1 to z.t do
        for k := -ai to z.p - 1 do
            begin
            z.LL[i, k] := z.LL[i, k + 1];
            z.LLP[i, k] := z.LLP[i, k + 1];
            end;
        for i := 1 to z.t do
            begin
            z.LL[i, z.p] := 0;
            z.LLP[i, z.p] := 0;
            end;
        z.l[-z.p].h := 0;
        z.l[-z.p].v := 0;
        z.name[-z.p] := '';
        z.token[-z.p] := 0;
        pennormal;
        z.p := z.p - 1;
        end;
    1 :
        begin
```

```
                deleterow(ai, z.t, TT);
                z.t := z.t - 1;
                end;
        15 :
                begin
                q := ai - z.maxt + z.maxp;
                qq := z.maxp + z.MAXM + z.j;
                deletecolio(ai, z.maxt + z.MAXM + z.j, q, qq, TT);
                z.j := z.j - 1;
                end;
        7 :
                begin
                q := ai - z.maxt + z.maxp;
                qq := z.maxp + z.m;
                deletecolio(ai, z.maxt + z.m, q, qq, TT);
                z.m := z.m - 1;
                end;
        otherwise
        ;
        end;
end;
(*****************************************************************************)
(*                                                                           *)
(*   This procedure calls the procedure for deleting arcs betwee internal    *)
(* places and transitions.                                                   *)
(*                                                                           *)
(*****************************************************************************)

procedure erasep (f, t, ai : integer;
        notj : boolean);
var
        k : integer;
begin
for k := f to t do
        begin
        if (z.LL[k, -ai] = 1) then
        fillarrow(z.l[ai], z.l[k], psize, tsize);
```

```
            if (z.LLP[k, -ai] = 1) and notj then
            fillarrow(z.l[k], z.l[ai], tsize, psize);
            end;
    end;

(*****************************************************************************)
(*                                                                           *)
(*   This procedure erases the arcs between an external place and a tran-*)
(*  sition  once that external place is deleted.                             *)
(*                                                                           *)
(*****************************************************************************)

    procedure erasepio (f, t, ai : integer;
    var flag : integer;
            notj : boolean);
    var
            q, k : integer;
    begin
    if (flag = 1) then
    q := ai - z.maxt + z.maxp
    else if (flag = 2) then
    q := ai - z.maxt + z.maxp;
    for k := f to t do
            begin
            if (z.LL[k, q] = 1) then
            fillarrow(z.l[ai], z.l[k], psize, tsize);
            if (z.LLP[k, q] = 1) and notj then
            fillarrow(z.l[k], z.l[ai], tsize, psize);
            end;
    flag := 0;
    end;

(*****************************************************************************)
(*                                                                           *)
(*   This procedure erases all arcs between a transition and all types of    *)
(*  places once that transition is deleted.                                  *)
(*                                                                           *)
```

```
(*****************************************************************************)

    procedure eraset (f, t, ai : integer;
            pn : net;
            notj : boolean);
    var
            q, k : integer;
    begin
    for k := 1 to pn.p do
            begin
            if (pn.LL[ai, k] = 1) then
            fillarrow(pn.l[-k], pn.l[ai], psize, tsize);
            if (pn.LLP[ai, k] = 1) and notj then
            fillarrow(pn.l[ai], pn.l[-k], tsize, psize);
            end;
    for k := (pn.maxp + 1) to (pn.maxp + pn.m) do
            begin
            q := k - pn.maxp + pn.maxt;
            if (pn.LL[ai, k] = 1) then
            fillarrow(pn.l[q], pn.l[ai], psize, tsize);
            if (pn.LLP[ai, k] = 1) and notj then
            fillarrow(pn.l[ai], pn.l[q], tsize, psize);
            end;
    for k := (pn.maxp + pn.MAXM + 1) to (pn.maxp + pn.MAXM + pn.j) do
            begin
            q := k - pn.maxp + pn.maxt;
            if (pn.LL[ai, k] = 1) then
            fillarrow(pn.l[q], pn.l[ai], psize, tsize);
            if (pn.LLP[ai, k] = 1) and notj then
            fillarrow(pn.l[ai], pn.l[q], tsize, psize);
            end;
    end;

(*****************************************************************************)
(*                                                                           *)
(*  This procedure calls the needed procedure for deleting an internal       *)
(*  place, an external place or a transition and taking care of all the      *)
```

```
(*  changes the corresponding fields.                                          *)
(*                                                                             *)
(*******************************************************************************)

   procedure deal (a, ai, n : integer;
           var flag : integer);
   var
           r : rect;
   begin
   case a of
   3 :
           begin
           frameoval(prect(z.l[ai],  1,  1));
           erasep(1, z.t, ai, TT);
           end:
   1 :
           begin
           paintrect(prect(z.l[ai],  2,  1));
           eraset(0, 0, ai, z, TT);
           end;
   7 :
           begin
           flag := 1;
           paintoval(prect(z.l[ai],  1,  1));
           erasepio(1, z.t, ai, flag, TT);
           end;
   15 :
           begin
           flag := 2;
           paintoval(prect(z.l[ai],  1,  1));
           eraseoval(prect(z.l[ai],  2,  1));
           erasepio(1, z.t, ai, flag, TT);
           end;
   otherwise
   ;
   end;
   end;
```

```
(*****************************************************************************)
(*                                                                           *)
(*  This procedure gives the option of moving places and transitions to      *)
(*  different positions on the screen. It also updates the length between    *)
(*  those places and transitions and redraws them.                           *)
(*                                                                           *)
(*****************************************************************************)

    procedure  movenode;
    var
            flag, a, ai : integer;
    begin
    if nshow then
    displayname;
    if tshow then
    displaytoken;
    flag := 0;
    penmode(patbic);
    getmouse(c.h,  c.v);
    a := findtype(c.h, c.v, ai);
    deal(a, ai, 1, flag);
    pennormal;
    while button do
    ;
    getmouse(c.h,  c.v);
    z.l[ai] := c;
    deal(a, ai, 2, flag);
    netchanged := TT;
    end;

(*****************************************************************************)
(*                                                                           *)
(*  This procedure redraws the graphs of the local entity, peer entity or    *)
(*  both entities depending on the passed parameter to it and by calling     *)
(*  the drawing procedures.                                                  *)
(*                                                                           *)
```

```
(*****************************************************************************)

procedure redraw (pn : net);
var
        i, j : integer;
begin
if beta = 1 then
        begin
        hideall;
        setdrawing;
        end;
for i := 1 to pn.p do
        begin
        pn.l[-i].h := round(pn.l[-i].h * alpha) + segma;
        frameoval(prect(pn.l[-i], 1, alpha));
        end;
for i := pn.maxt + 1 to pn.maxt + pn.m do
        begin
        pn.l[i].h := round(pn.l[i].h * alpha) + segma;
        paintoval(prect(pn.l[i], 1, alpha));
        end;
for i := pn.maxt + pn.MAXM + 1 to pn.maxt + pn.MAXM + pn.j do
        begin
        pn.l[i].h := round(pn.l[i].h * alpha) + segma;
        paintoval(prect(pn.l[i], 1, alpha));
        eraseoval(prect(pn.l[i], 2, alpha));
        end;
for i := 1 to pn.t do
        begin
        pn.l[i].h := round(pn.l[i].h * alpha) + segma;
        paintrect(prect(pn.l[i], 2, alpha));
        eraset(0, 0, i, pn, TT);
        end;
nshow := FF;
tshow := FF
end;
```

```
(*****************************************************************************)
(*                                                                           *)
(*  This procedure calls a corresponding procedure to draw a bar in the      *)
(*  given coordinates to represent a transition.                             *)
(*                                                                           *)
(*****************************************************************************)

   procedure drawt (p : point);
   begin
   if z.t < z.maxt then
           begin
           paintrect(prect(p, 2, 1));
           z.t := z.t + 1;
           z.l[z.t] := p;
           netchanged := TT;
           end
   else
   note(2500, 20, 10);
   end;

(*****************************************************************************)
(*                                                                           *)
(*  This procedure implements the second step of the synthesizer PNPS.       *)
(*  It initializes the input and output matrices of the peer entity by       *)
(*  using the matrices of the local entity.                                  *)
(*                                                                           *)
(*****************************************************************************)

   procedure ftrans (var dp, dc : ar);
   var
           i, j : integer;
   begin
   for i := 1 to z.t do
   for j := 1 to z.maxp do
           begin
           dp[i, j] := z.LL[i, j];
           dc[i, j] := z.LLP[i, j];
```

```
            end;
    for i := 1 to z.t do
    for j := (z.maxp + 1) to (z.maxp + z.MAXM + z.j) do
            begin
            dp[i, j] := z.LLP[i, j];
            dc[i, j] := z.LL[i, j];
            end;
    end;

(*****************************************************************************)
(*                                                                           *)
(*  This procedure adds a row at the bottom of the input place and a ro      *)
(*  at the bottom of the output place of the peer entity, each time is       *)
(* called.                                                                   *)
(*                                                                           *)
(*****************************************************************************)

    procedure addrow (var mc, l, x, nnn : integer;
            var dp, dc : ar);
    var
            i : integer;
    begin
    mc := mc + 1;
    for i := 1 to (z.p + z.m + z.j) do
            begin
            dp[mc, i] := 0;
            dc[mc, i] := 0;
            end;
    dp[mc, l] := 1;
    dp[mc, x] := 1;
    dc[mc, nnn] := 1;
    end;

(*****************************************************************************)
(*                                                                           *)
(*  This procedure implements the step III of the synthesizer PNPS. All      *)
(* conditions are checked one by one by passin through all the rows and      *)
```

```
(*   columns of the four matrices four both entities. This procedure calls *)
(*   the addrow procedure when all the conditions all satisfied.           *)
(*                                                                         *)
(*************************************************************************)

   procedure rules (m, n1, n2, n3 : integer;
           var mc : integer;
           var dp, dc : ar);
   label
           99, 100, 101, 102;
   var
           mmm, nnn, nn, l, oo, t1, t2, x, w : integer;
           i, j, ss, k, a, b, mm, n : integer;
           found1, found3, found4 : boolean;
   begin
   n := n1 + n2 + n3;
   mc := m;
   found1 := false;
   for i := 1 to m do
           begin
           for j := 1 to n1 do
                begin
                if (z.LL[i, j] = 1) then
                     begin
                     t1 := i;
                     nn := j;
                     end;
                if (z.LLP[i, j] = 1) then
                nnn := j;
                end;
           for j := (n1 + 1) to (n1 + n2) do
           if (z.LL[i, j] = 1) then
                begin
                w := j;
                found1 := true;
                end;
           if (found1 = false) then
```

```
goto 100
else
     begin
     found1 := false;
     for k := 1 to m do
     if (k <> i) then
          begin
          for j := 1 to n1 do
               begin
               if (z.LL[k, j] = 1) then
                    begin
                    t2 := k;
                    mm := j;
                    end;
               if (z.LLP[k, j] = 1) then
               l := j;
               end;
          for ss := (n1 + 1) to (n1 + n2) do
          if (z.LL[k, ss] = 1) then
          if (ss = w) then
               begin
               found3 := false:
               for a := 1 to mc do
                    begin
                    for j := 1 to n1 do
                         begin
                         if (dp[a, j] = 1) then
                         mmm := j;
                         if (dc[a, j] = 1) then
                         oo := j;
                         end;
                    if ((mm = mmm) and (nn = oo)) then
                    for j := (n1 + n2 + 1) to n do
                    if (dp[a, j] = 1) then
                         begin
                         x := j;
                         found3 := true;
```

```
                                goto 99;
                                end;
                            end;
99 :
                        if (found3 = true) then
                            begin
                            found3 := false;

                            for b := 1 to m do
                            if ((b <> t1) and (b <> t2)) then
                                begin
                                found4 := false;
                                if ((z.LL[b, l] <> 0) and (z.LLP[b, x] <> 0))
                                then goto 100
                                else
                                found4 := true;
                                end;
                            if (found4 = true) then
                                begin
                                addrow(mc, l, x, nnn, dp, dc);
                                c.h := (z.l[-l].h + z.l[-nnn].h) div 2;
                                c.v := (z.l[-l].v + z.l[-nnn].v) div 2;
                                zp.l[mc] := c;
                                end;
                            end;
                        end;
101 :
                    end;
                end;
100 :
            end;
    end;

(*****************************************************************************)
(*                                                                           *)
(*   This procedure calls for applying the Step II of the synthesizer PNPS *)
(*                                                                           *)
```

```
(*****************************************************************************)

   procedure  rule1;
   begin
   zp := z;
   ftrans(zp.LL,  zp.LLP);
   end;

(*****************************************************************************)
(*                                                                           *)
(*   This procedure calls for applaying Step III of the synthesizer PNPS     *)
(*                                                                           *)
(*****************************************************************************)

   procedure  rule2;
   begin
   rules(z.t, z.maxp, z.MAXM, z.j, zp.t, zp.LL, zp.LLP);
   end;

(*****************************************************************************)
(*                                                                           *)
(*   This procedure reads the number of tokens to be in an internal place *)
(*                                                                           *)
(*****************************************************************************)

   procedure  token;
   var
          a, ai, n : integer;
   begin
   getmouse(c.h,  c.v);
   a := findtype(c.h, c.v, ai);
   if a = 3 then
          begin
          invertrect(prect(z.l[ai],  1,  1));
          settext;
          writeln('Specify  the  number  of  tokens?');
          readln(n);
```

```
            if z.token[ai] + n >= 0 then
            z.token[ai] := n
            else
            z.token[ai] := 0;
            invertrect(prect(z.l[ai], 1, 1));
            netchanged := TT;
            end;
    end;

(*****************************************************************************)
(*                                                                           *)
(*  This procedure reads the names for internal places, external places      *)
(*  and transitions in the displayed net.                                    *)
(*                                                                           *)
(*****************************************************************************)

    procedure  npt;
     var
            a, ai : integer;
    begin
    getmouse(c.h, c.v);
    a := findtype(c.h, c.v, ai);
    if a <> 0 then
            begin
            invertrect(prect(z.l[ai], 1, 1));
            settext;
            writeln('Specify  name! Existing name is [', z.name[ai], '] ');
            readln(z.name[ai]);
            invertrect(prect(z.l[ai], 1, 1));
            netchanged := TT;
            end;
    end;

(*****************************************************************************)
(*                                                                           *)
(*  This procedure gives the options of retreiving a file from the disk to   *)
(*  display it on the screen or saves a displayed file on the disk and asks  *)
```

```
(*   to give a name to that file. Only local entities can be saved or retrei  *)
(*  ved from the disk by GSAPS.                                                *)
(*                                                                             *)
(*******************************************************************************)

   procedure disk (m : str24);
   var
         prompt, temp : str24;
   begin
   hideall;
   prompt := ' file name?';
   if m = 's' then
   temp  :=  newfilename(prompt)
   else
   temp  :=  oldfilename(prompt);
   if (temp <> '') and (m = 's') then
         begin
         open(filevar,  temp);
         filevar^ := z;
         put(filevar);
         close(filevar);
         end;
   if (temp <> '') and (m = 'g') then
         begin
         open(filevar,  temp);
         z := filevar^;
         close(filevar);
         alpha := 1;
         beta := 1;
         segma := 0;
         redraw(z);
         end;
   if (temp = '') and (m = 'g') then
         begin
         emptynet;
         setdrawing;
         end;
```

```
    netchanged := FF;
    end;

(*****************************************************************************)
(*                                                                           *)
(*  This is the main procedure in the system. It includes all the options    *)
(*  a user may have such as creating places, transitions and arcs; modi-     *)
(*  fying the local entity, generating the peer entity, displaying the       *)
(*  graphs of these entities and saving the local entity for GSAPS futu-     *)
(* re use or MacPaint fure use.                                              *)
(*                                                                           *)
(*****************************************************************************)

    procedure  HandleMenu;

(*****************************************************************************)
(*                                                                           *)
(* This procedure calls the corresponding procedure for drawing a circle    *)
(*  to represent a place. The location and position of that place are also   *)
(* saved.                                                                    *)
(*                                                                           *)
(*****************************************************************************)

    procedure drawp (p : point);
    begin
    if z.p < z.maxp then
            begin
            frameoval(prect(p,  1,  1));
            z.p := z.p + 1;
            z.l[-z.p].h := p.h;
            z.l[-z.p].v := p.v;
            netchanged := TT;
            end
    else
    note(2500,  20,  10);
    end;
```

```
(*****************************************************************************)
(*                                                                           *)
(*  This procedure calls a corresponding procedure for drawing a black       *)
(*  circle to represent an external input place in the local entity. The     *)
(*  location and position of that place are also saved.                      *)
(*                                                                           *)
(*****************************************************************************)

   procedure drawip (p : point);
   begin
   if z.m < z.MAXM then
           begin
           paintoval(prect(p, 1, 1));
           z.m := z.m + 1;
           z.l[z.maxt + z.m] := p;
           netchanged := TT;
           end
   else
   note(2500, 20, 10);
   end;

(*****************************************************************************)
(*                                                                           *)
(*  This procedure calls a corresponding procedure for drawing a black       *)
(*  circle with a wight bar inside to represent an external output place     *)
(*  in the local entity. The location and position of that place is also     *)
(* saved.                                                                    *)
(*                                                                           *)
(*****************************************************************************)

   procedure drawop (p : point);
   begin
   if z.j < z.MAXJ then
           begin
           paintoval(prect(p, 1, 1));
           eraseoval(prect(p, 2, 1));
           z.j := z.j + 1;
```

```
            z.l[z.maxt + z.MAXM + z.j] := p;
            netchanged := TT;
            end
    else
    note(2500, 20, 10);
    end;

(*****************************************************************************)
(*                                                                           *)
(*  This is the main procedure which calls other procedures to draw an       *)
(*  arc from a place to a transition or from a transition to a place. It      *)
(*  also has the option of deleting these arcs dependent on the value of     *)
(*  its boolean variable black.                                              *)
(*                                                                           *)
(*****************************************************************************)

    procedure arc (black : boolean);
    var
            aii, bjj : integer;
    begin
    if first then
            begin
            getmouse(c.h, c.v);
            a := findtype(c.h, c.v, ai);
            if a > 0 then
                 begin
                 first := FF;
                 note(1000, 10, 5);
                 end;
            end
    else
            begin
            getmouse(c.h, c.v);
            b := findtype(c.h, c.v, bj);
            if b > 0 then
                 begin
                 if not black then
```

```
penmode(patbic);
first := TT;
netchanged := TT;
case a + b of
4 :
if a <> 1 then
     begin
     fillarrow(z.l[ai], z.l[bj], psize, tsize);
     if black then
     z.LL[bj, -ai] := 1
     else
     z.LL[bj, -ai] := 0;
     end
else
     begin
     fillarrow(z.l[ai], z.l[bj], tsize, psize);
     if black then
     z.LLP[ai, -bj] := 1
     else
     z.LLP[ai, -bj] := 0;
     end;
8, 16 :
if (a = 1) then
     begin
     fillarrow(z.l[ai], z.l[bj], tsize, psize);
     bj := bj - z.maxt + z.maxp;
     if black then
     z.LLP[ai, bj] := 1
     else
     z.LLP[ai, bj] := 0;
     end
else
     begin
     fillarrow(z.l[ai], z.l[bj], psize, tsize);
     ai := ai - z.maxt + z.maxp;
     if black then
     z.LL[bj, ai] := 1
```

```
                    else
                    zLL[bj, ai] := 0;
                    end;
               otherwise
               ;
               end;
          pennormal;
          note(1000, 10, 8);
          note(1500, 10, 5);
          end;
          end;
     end;

(*****************************************************************************)
(*                                                                           *)
(*  Procedure Handle Menu starts at this point. Depending on the user        *)
(*  selection, corresponding procedures are called by Handle Menu.           *)
(*                                                                           *)
(*****************************************************************************)

          begin
          case WInLineF($A86A, newmenu) of
          100 :
          case WInLineF($A86B, newmenu) of
          1 :
               begin
               if netchanged then
               disk('s');
               emptynet;
               setdrawing;
               end;
          2 :
               begin
               if netchanged then
               disk('s');
               disk('g');
```

```
        end;
3 :
disk('s');
4 :
savedrawing('drawing');
5 :
        begin
        if netchanged then
        disk('s');
        done := TT;
        end;
end;
101 :
if not inbar then
case WInLineF(SA86B, newmenu) of
1 :
        begin
        getmouse(c.h, c.v);
        drawp(c);
        end;
2 :
        begin
        getmouse(c.h, c.v);
        drawip(c);
        end;
3 :
        begin
        getmouse(c.h, c.v);
        drawop(c);
        end;
4 :
        begin
        getmouse(c.h, c.v);
        drawt(c);
        end;
5 :
arc(TT);
```

```
        6 :
        npt;
        7 :
        token;
        end;
102 :
case WInlineF($A86B, newmenu) of
1 :
deletenode;
2 :
arc(FF);
3 :
movenode;
4 :
        begin
        alpha := 1;
        beta := 1;
        segma := 0;
        redraw(z);
        end;
5 :
displayname;
6 :
displaytoken;
end;
103 :
case WInlineF($A86B, newmenu) of
1 :
rule1;
2 :
rule2;
3 :
        begin
        alpha := 1;
        beta := 1;
        segma := 0;
        redraw(zp);
```

```
            end;
    4 :
            begin
            alpha := 0.5;
            beta := 2;
            setdrawing;
            segma := 0;
            redraw(z);
            segma := 264;
            redraw(zp);
            end;
    end;

    otherwise
    ;
    end;
    end;

(*****************************************************************************)
(*                                                                           *)
(*  This procedure creates all the menus in GSAPS and the possible           *)
(*  options the user will have. Five menus are created below:                *)
(*                                                                           *)
(*****************************************************************************)


    procedure  InitializedOK;
    begin
    FlushEvents($017F,  0);
    OldMenuBar  :=  Pointer(LInLineF($A93B));
    InLineP($A934);
    filemenu  :=  Pointer(LInLineF($A931,  100,  'File'));
    InLineP($A933, filemenu, 'New;Open;Save: for GSAPS;Save: for
            MacPaint;Quit');
    InLineP($A935,  filemenu,  0);
    createmenu  :=  Pointer(LInLineF($A931,  101,  'Create  Net'));
    InLineP($A933,  createmenu,
```

```
        'Int-place;Ext-input-place;Ext-output-place;Transition');
    InLineP($A933, createmenu, 'Arc;Name;Token');
    InLineP($A935, createmenu, 0);
    modifynaMenu := Pointer(LInLineF($A931, 102, 'Modify Net'));
    InLineP($A933, modifynaMenu, 'Delete node;Delete arc;Move node;Draw
          local');
    InLineP($A933, modifynaMenu, 'Display or hide name;Display or hide
          token');
    InLineP($A935, modifynaMenu, 0);
    parameterMenu := Pointer(LInLineF($A931, 103, 'Synthesis'));
    InLineP($A933, parameterMenu, 'Apply Rule R1;Apply Rule R2;Draw
          peer;Draw both');
    InLineP($A935, parameterMenu, 0);
    InLineP($A937);
    end;   (* Procedure handle menu ends *)

(*****************************************************************************)
(*                                                                           *)
(*  This procedure is used to print welcoming messages at the starting     *)
(* and ending sessions of GSAPS.                                             *)
(*                                                                           *)
(*****************************************************************************)


    procedure welcome (m : str24);
    begin
    setdrawing;
    fillrect(0, 0, 322, 512, Gray);
    setrect(r, 40, 120, 472, 200);
    fillroundrect(r, 20, 20, white);
    frameroundrect(r, 20, 20);
    textface([bold, shadow]);
    textsize(18);
    moveto(130, 170);
    drawstring(m);
    end;
```

```
(*****************************************************************************)
(*                                                                           *)
(*  The main program starts at this point. It initializes some variables     *)
(*  gets into a loop calling the procedure Handle menu until the user fi-    *)
(*  nishs his work and assigns the the value true to the variable done       *)
(*  by selecting the entry Quit from the File menu.                          *)
(*                                                                           *)
(*****************************************************************************)


begin   (* Main *)
done := FF;
first := TT;
oldmenu  :=  999999;
netchanged := FF;
nshow := FF;
tshow := FF;
psize := 10;
tsize := 8;
welcome('Welcome to    G S A P S');
for delay := 1 to 6000 do
;
InitializedOK;
disk('g');
repeat
if  testmouse(inbar)  then
        begin
        if inbar then
            begin
            first := TT;
            oldmenu  :=  LinLineF($A93D,  Event.where);
            end;
        newmenu  :=  oldmenu;
        Handlemenu;
        end;
until Done;
```

```
InLineP($A932,   parameterMenu);
InLineP($A93C,   OldMenuBar);
InLineP($A937);
welcome('     B Y E      B Y E ');
end.       (* main program ends    *)
```