Requirements Analysis Using Petri Nets

by

Bradley Colvin Gaylord

B. A. Western State College of Colorado, 1978

A Master's Report

submitted in partial fulfillment of the

requirements for the degree

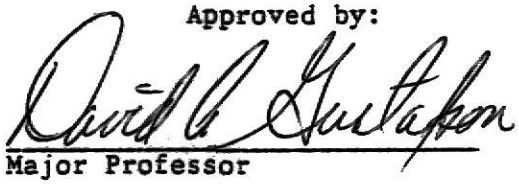
Master of Science

Department of Computer Science

Kansas State University
Manhattan, Kansas

1984

Approved by:



Major Professor

TABLE OF CONTENTS

THIS BOOK CONTAINS NUMEROUS PAGES WITH DIAGRAMS THAT ARE CROOKED COMPARED TO THE REST OF THE INFORMATION ON THE PAGE.

THIS IS AS RECEIVED FROM CUSTOMER.

LIST OF FIGURES

CHAPTER 1 - INTRODUCTION

This report describes the implementation of a tool which aids in the analysis of software requirements specifications. The tool translates a requirements specification into a Petri net model [Pet81], which is then interpreted to provide a prototype simulation of the specification. An analysis of aspects of the completeness and correctness can be performed by building such a prototype of the specification.

This chapter introduces the discussion of requirement specifications, the software life cycle and prototyping, and Petri net simulation systems. Chapter 2 describes the externally visible behavior of the implementation, as well as describing *what* transformations are applied to achieve this external behavior. Chapter 3 deals with *how* the transformations are applied, in terms of the data structures and algorithms used by the implementation. Chapter 4 deals with conclusions and extensions.

1.1 Requirements Specifications

Many software developments begin without a clear understanding of the details necessary to design the software product. Such projects frequently end up having to be "fixed" after they have been built, at a much higher cost than that which would have been incurred had requirements been specified in advance. This predicament has been extensively discussed in the literature [Mye78, Zel78].

A requirements specification is a document which describes *what* the software product will do, but not *how* it will do it [Boe76]. While there have been many proposals [Ros77, Tei77, Hen80, etc.], concerning the correct nature of requirements specifications, it is generally agreed that a specification should be the "black box" description of everything a software engineer needs to know to produce the software product, and no more.

1.1.1 *Entity-Relationship-Attribute Requirements Specifications*

The type of requirements specification which this implementation will be translating and interpreting is called an Entity-Relationship-Attribute (ERA) model. The ERA model uses a form based specification system, and bears a strong resemblance to the method used in the specification of the software requirements for the A-7E aircraft [Hen78].

Components of the software product are specified as "entities". Entities may be "related" to other entities. Entities also have "attributes" which characterize the component. Some entities specify such components as the *input*, *output* and internal *data* of the software product. Other entities are used to specify the *Activities* and *Periodic functions* of the software product. Examples of these classes of entities may be found in Appendix C (ie: the $chess_board$ and Computer_Move entities, respectively).

The syntax for ERA specifications may be found in Appendix A. This implementation does not utilize all of the information that may be expressed in a ERA specification accepted by this grammar. Some of the entities and their relationships/attributes are ignored by this implementation, because they are intended for other

implementations. A stricter BNF grammar which describes the input language which is acceptable to this implementation is given in Appendix B. The grammar of Appendix A will accept any input which the grammar of Appendix B will accept, with the exception of certain upper and lower case lexical conventions.

1.1.2 *The Software Life-cycle*

A requirements specification is the product of the initial phase of the software life-cycle. The life-cycle model of software development [Zel78, Boe76] proceeds in a "waterfall" fashion: requirements are developed for a software product, a design is developed from the requirements, and the code is written from the design. Associated with every phase is a product which is a prerequisite of subsequent phases. In any phase, one can backup to an earlier phase to rework the product of that earlier phase. The changes to that earlier phase will then cascade as subsequent phases are reworked.

It has been maintained that the pure software life-cycle model is not useful in all instances [McC82]. One view is that requirements are difficult to specify in advance for some application areas. Another is that flaws in the requirements can be carried through into the implementation phase, thus necessitating a costly reworking of the software product. To this end, prototyping has been suggested as an alternative.

1.1.3 *Prototyping*

Prototyping involves creating versions of parts of the software product in order to determine how to build the operational software product [Boe84]. A "pure" prototyping effort would forgo much of

the planning, and would spend a much greater proportion of the development time experimenting with various implementations, many of which would be thrown away.

There are many advantages to prototyping, including always having something that works (however inefficiently). Additionally, the user interface developed in a prototyping approach is usually superior to that of a "specified" software product, because users who are unsure of their needs can experiment with several prototype versions. Disadvantages also exist with a "pure" prototyping effort. Because there is less planning, more time has to be spent on finding errors which might not have occurred had there been a precise specification.

Research on prototyping [Bra82] does not exclude software life-cycle concepts. A number of approaches involve "executing" a requirements specification. This prototyping of requirements allows for a determination of the completeness and correctness of a specification.

1.2 Approaches to Requirements Prototyping

Extensive work has been done in the area of executable requirements specifications [Zav82, Bra82]. The formulation of a requirements specification into a language with operational semantics allows the requirements specification to become a prototype [Smo82].

A requirements specification is termed "executable" because the specification becomes a prototype of the system being specified. The prototype may be interpreted to provide an analysis of the system. Because ERA requirements specifications are expressed

using a precise syntax, ERA specifications can be "executed".

Requirements prototyping facilitates the development of the specification, because users can interact with a model of the software product and can discover deficiencies in the requirements early in the life-cycle. An additional benefit of requirements which are in a machine executable form is that products of subsequent phases may be generated automatically [Kla82]. A similar situation is seen in the generation of valid control structure code in Pl/I from a design specification [Nel83].

1.3 Petri Net Simulation Systems

Petri nets are tools which are used to model systems [Pet81, Tan81]. Typically, the systems being modeled exhibit concurrency. Components of such systems interact, and must be synchronized. Synchronization and other considerations, such as deadlock and resource starvation, are modeled well by Petri nets.

1.3.1 *Petri Net Structures*

A Petri net has four parts [Pet81]: a set of *places* P, a set of *transitions* T, an *input* function I, and an *output* function O. The input and output functions relate transitions and places. The input function I is a mapping from a transition t to a collection of places $I(t)$, known as the *input places* of the transition. The output function O maps a transition t to a collection of places $O(t)$ known as the *output places* of the transition.

The structure of a Petri net is defined by its places, transitions, its input function and its output function. The next section discusses an equivalent definition of Petri nets which introduces

arcs and tokens.

1.3.2 *Petri Net Graphs*

A Petri net graph represents a Petri net structure as a bipartite directed multigraph [Jor76]. It is bipartite because there are two types of nodes, called "places" and "transitions", with the pictorial representation shown in Figure 1-1. It is directed because the arcs point from one type of node to the other type of node. It is a multigraph because nodes can have many arcs pointing to them and away from them.

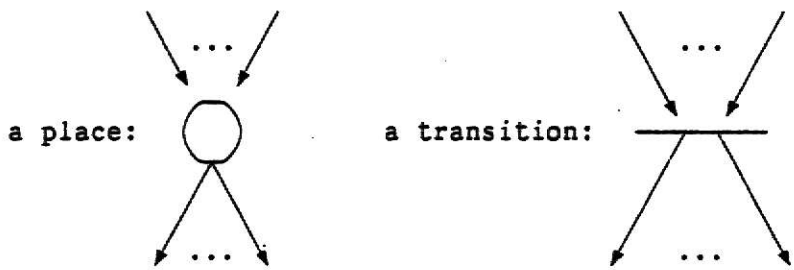


Figure 1-1. Petri Net Graph Nodes

Places may contain tokens, and the distribution of tokens within the graph may be altered by "firing" a transition. When all places connected to a transition by directed arcs pointing to that transition contain a token, the transition is "enabled" for firing. The firing of a transition is an atomic event which removes one token from the "input places" associated with each "input" arc pointing to the transition and puts one token into each of the "output places" associated with every "output" arc pointing away from the transition.

If there are several enabled transitions in a graph, it is indeterminate as to which transition will fire first. They may

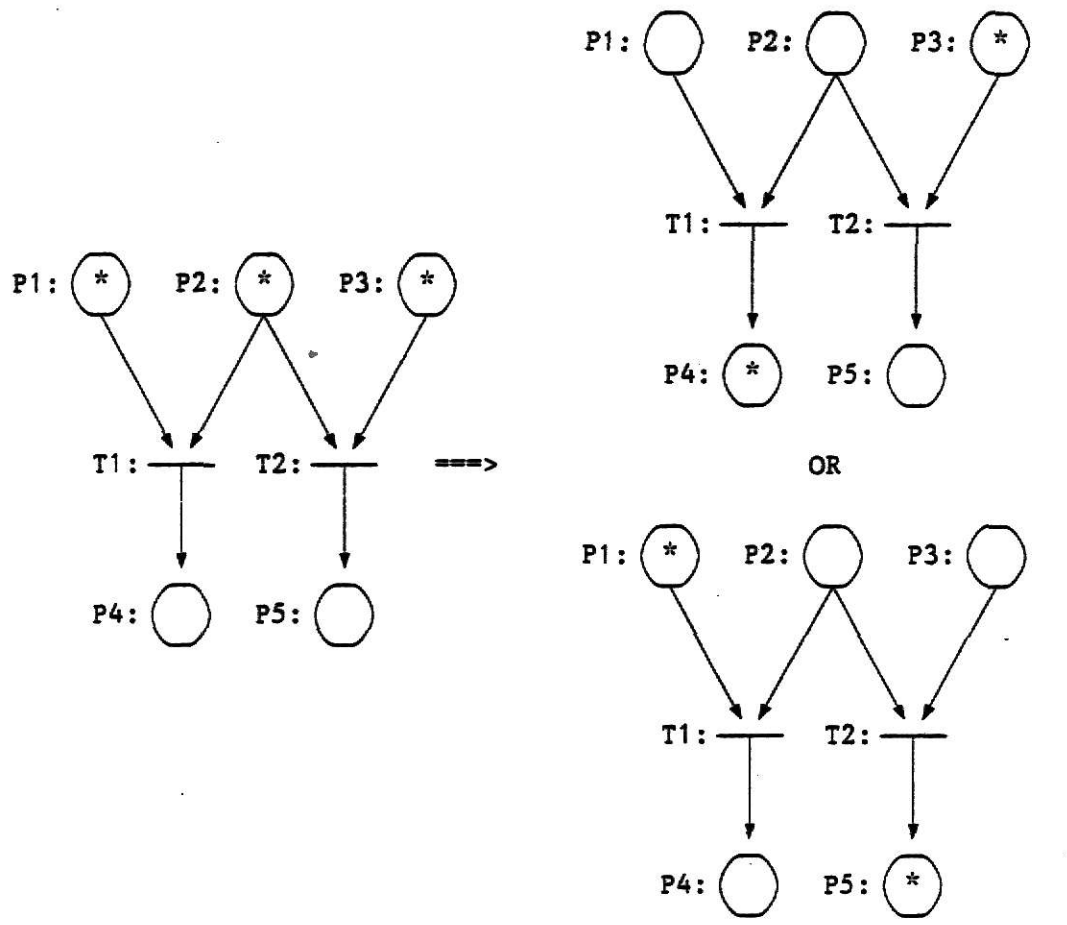


Figure 1-2. Petri Net Graph: Two Transitions in Conflict

fire "in parallel", or if two events share an input place which has only one token, a "conflict" occurs in which one and only one of the enabled transitions may be fired. Figure 1-2 shows a Petri net graph which is in conflict. Transitions T1 and T2 are both enabled, and it is indeterminate as to which transition will fire. If T1 fires, a token will be taken from places P1 and P2, and a token will be put into place P4. Likewise, with the firing of T2, a token will be taken from P2 and P3, and a token will be put into P5.

1.3.3 *Petri Net Language*

A textual representation of Petri net graphs has been developed [Nel82]. Every node in the graph has a corresponding statement in the Petri Net Language (PNL).

```
T0 : INIT   OUTPUT TO ( P1, P2, P3 )
P1 : PLACE  OUTPUT TO ( T1 )
T1 : TRANS  OUTPUT TO ( P4 )
P2 : PLACE  OUTPUT TO ( T1, T2 )
T2 : TRANS  OUTPUT TO ( P5 )
P3 : PLACE  OUTPUT TO ( T2 )
P4 : PLACE  OUTPUT TO ( T3 )
P5 : PLACE  OUTPUT TO ( T3 )
T3 : TERM
```

Figure 1-3. PNL code corresponding to Fig. 1-2

An illustration of the PNL program which represents the Petri net graph of Figure 1-2 is shown in Figure 1-3. Each line of the program has the node name, a colon, the node type and a list of nodes that a token will be directed to.

Certain annotations were added to Petri nets in order to make the simulation more useful [Nel83]. The annotations used by this implementation include a specification of conditional flow and an ability to invoke a procedure as a side effect of a transition. Although this implementation does not use the "inhibitor arcs" of PNL, it should be noted that a Petri net model that incorporates inhibitor arcs is equivalent to a Turing machine, and thus can model any process [Pet81]. Another notation which has been added for this implementation is a "WHEN" construct which conditionally allows a transition to become firable [Kel76].

1.4 Scope of the Implementation

The UNIX* implementation associated with this report provides for many of the features discussed above. The two processes in this implementation are a translator and an interpreter. The translator compiles an ERA requirements specification into a language which represents an augmented Petri Net representation of certain portions of the ERA specification. The interpreter produces a dynamic analysis ala prototyping based upon the above Petri Net representation.

---

* UNIX is a trademark of AT&T Bell Laboratories.

CHAPTER 2 - REQUIREMENTS

The implementation includes two primary processes: the "translator" and the "interpreter", as shown in Figure 2-1. This chapter describes the externally visible behavior of these processes, as well as describing the transformations which are applied to achieve this external behavior.

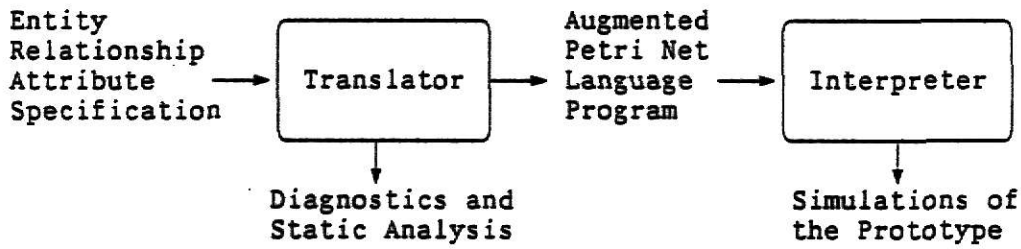


Figure 2-1. Data Flow Diagram of this Implementation

The translator converts an Entity-Relationship-Attribute (ERA) requirements specification into an augmented Petri Net language (PNL) representation of the specified system. The interpreter analyzes the PNL intermediate code representation produced by the translator and provides a prototyping of the system being specified.

The next three sections discuss the requirements specifications for the translator process, the PNL intermediate code which both processes use, and the interpreter process. Particular attention is given to the nature of the Requirements Specification used as input to the translator (with respect to the Petri net which is formed from the specification), because this directly affects the formation of the PNL intermediate code used by the interpreter.

2.1 The Translator

The translator scans a file in the ERA format and creates an internal Petri net representation of the specification. The ERA file contains entities and modes, which have grammar productions describing them. These entities and modes are mapped to Petri net places and transitions, by assigning semantic meanings to the grammar productions which describe the entities and modes.

The translator creates a file containing statements in PNL. These statements directly correspond to the internal Petri net representation. The PNL file is readable by ordinary text editors, and may be visually examined to discover problem areas in the requirements specification. The PNL file is used as input for the interpreter, or any other tool which is designed to accept PNL [Nel82].

There are two classes of messages which can be produced by the error handling capability of the translator: fatal "error" messages and non-fatal "diagnostic" messages. For instance, if the translator encounters syntax errors in the scanning of the ERA input file, fatal error messages are produced and the translation process terminates. The translator also issues non-fatal diagnostic messages during the static analysis phase. During the static analysis of the requirements specification, conditions such as deadlock and starvation are detected in the internal Petri net representation, and a diagnostic message is produced. The messages are explained in detail in Appendix F.

2.1.1 *ERA Input to the Translator*

The ERA file contains a "PROCESS" line, a group of entities, and a MODE_TABLE. The PROCESS line gives a name to the requirements specification for a specific software product. The entities specify the data and functions of the software product. The MODE_TABLE lists the mode transitions, which correspond to global state information of the software product.

In the discussions which follow, examples of the ERA format will be taken from the specification of the chess process in Appendix C. The BNF grammar describing the ERA input language may be found in Appendix B.

The ERA structures of primary interest are the entities and modes. The entities are subdivided into two classes. The first class of entities specifies the data and the second class of entities specifies the functions. Modes are disjoint sets of system states.

There are three major grammar productions which correspond to the primary ERA structures of interest. They are the <i_o_data>, <function> and <mode_table> productions. These productions correspond to the two classes of entities, and the class of modes, respectively.

The <i_o_data> production derives entities involving the input, output and internal storage of data items. These entities will generally be mapped to "places" in the Petri net representation of the specification.

The <function> production derives entities involving Demand and Periodic Functions. They will be mapped to "transitions" in the

Petri net representation.

The <mode_table> production derives the last section of the specification. Modes are mapped to places in the Petri net representation of the specification.

2.1.1.1 *Entity Format:*

Entities begin with an Entity line, giving the name of the entity. This line is followed by relation lines, representing relationships and attributes associated with that entity. Entities are separated by blank lines. This ordering is expressed in Figure 2-2.

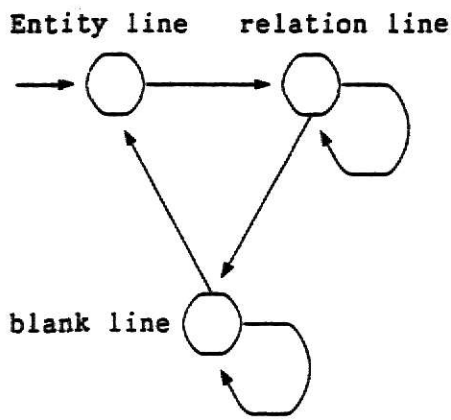


Figure 2-2. Entity Syntax

Every line in an entity is of the form "keyword : value". Each Entity line begins with a capitalized "entity keyword" in column 1. Relation lines are indented with spaces; "relation keywords" begin with a lower case character.

While the BNF for this input language is stated precisely in Appendix B, the following subsections discuss the semantics of the productions (for the three primary types of entity) that the translator recognizes. The relationships and attributes recognized

for each type of entity are also discussed.

2.1.1.2 *The <i_o_data> Class of Entities:*

The <i_o_data> entities form a class of entities which describes data associated with the software product. In the context of the chess specification, the data is either input from a keyboard, output to a crt, or internal data on secondary storage.

For the purposes of this implementation, the only attribute of interest in this class is "media". The media attribute associates a device with the entity, and is displayed by the interpreter when the occurrence of an input or output event is simulated. Other attributes, such as "structure", are ignored because the syntax describing their possible values is imprecise, and the information is meant for other implementations.

For the purposes of this implementation, relations are undefined for the <i_o_data> class of entity. Relations between functions and data are *only* expressed in the definition of <function> entities.

Entities in the <i_o_data> class are mapped to Petri net places, and <function> entities are mapped to Petri net transitions. <i_o_data> entities serve as the input and output places of the <function> transitions. For instance, a function may be related to two input entities and three output entities. The corresponding transition will not be enabled until both input places contain a token; at that time, the three output places will each receive a token. <i_o_data> entities can be related to one or more "Function" entity types.

The names of entities in the <i_o_data> class are always delimited by the dollar sign '$' and are preceded by one of the keywords "Input", "Output" or "Input_output":

- **Input keyword:** When the media is a keyboard, the Input entity consists of data expected to be entered by the user of this product. This entity will be translated into an input place of the transitions it is related to. An example from the chess specification would be the "$board_description$" Input entity. $board_description$ is related to the "Create_special_board" <function> entity, so $board_description$ would become an input place of the Create_special_board transition.

- **Output keyword:** When the media is a crt, the Output entity is usually data meant to be displayed to the user of the product. This entity will be translated into an output place of the transitions it is related to. An example from the chess specification would be the "$status$" Output entity. $status$ is related to the "Computer_Move" <function> entity, so $status$ would become an output place of the Computer_Move transition.

- **Input_output keyword:** Input_output entities specify the stored data of the software product. The media for Input_output entities is usually internal or secondary_storage. Examples would be $chess_board$ and $stored_board$ respectively. Input_output entities are not translated into places. Instead, they are translated into 'function statements" meant to be invoked when the corresponding transition fires. If

such an entity is related to a function entity via the "input" keyword, then the Input_output entity name becomes an argument to the function to be invoked. If it is related via the "output" keyword, it becomes the return value of the function invocation.

2.1.1.3 *The <function> Class of Entities:*

The <function> entities form the other class of entities, describing functions of the software product. In a typical ERA specification, functions are never related to one another directly. Instead they are related to data entities and modes. This definition of "relations" between functions and data insures that the Petri net graph will be bipartite.

Entities in the <function> class are mapped to Petri net transitions, because they represent *events* [Pet81]. <i_o_data> entities and modes are mapped to Petri net places, because they represent the *condition* of the system.

The <i_o_data> entities and modes related to the functions in the next two Figures are all defined in Appendix C. The "Recreate_board" and "Time_warning_2" examples are meant to be used for illustrative purposes and do not form a part of the chess specification in Appendix C.

The class of <function> entities is divided into Demand and Periodic functions. Demand functions are generally invoked by an event (ie: a user request), while Periodic functions occur at regular intervals based upon a predetermined cycle. The names of Demand and Periodic functions always begin with a capital letter,

and are denoted by the keywords "Activity" and "Periodic_function", respectively:

- **Activity keyword:** An Activity is an entity which specifies a Demand function. An analogous concept in the UNIX environment would be a user-callable program. Figure 2-3 shows a Demand function called "Recreate_board". A Demand function is

```
Activity : Recreate_board
   input                          : $stored_board$
   input                          : $name_of_game$
   output                         : $status$
   output                         : $chess_board$
   required_mode                  : *START*
   necessary_condition            : $retrieve$
   action                         : *START* -> *NORMAL*
```

Figure 2-3. Example of a Demand Function Entity

initiated by an event which is external to the software product. For example, a user might type the command "retrieve chess.game" on the keyboard, and thus initiate the Recreate_board Activity.

Activities use the "necessary_condition" keyword, and are related to Input entities. A "necessary_condition" is the relation of a command (ie: the Input entity $retrieve$) to a Demand function. When the command is issued, the Activity becomes executable, and the corresponding transition becomes firable. The Activity can be related to other Input entities which specify further information (ie: $name_of_game$).

- **Periodic_function keyword:** A Periodic_function is an entity which (curiously enough) specifies a Periodic function. An analogous concept in the UNIX environment would be a daemon

process. Figure 2-4 shows a Periodic function called "Time_warning_2". A Periodic function is initiated by the occurrence of some logical condition which is based upon some internal cyclic measure. For example, the Periodic_function Time_warning_2 is initiated upon the occurrence of the condition of "timer > 300 seconds".

```
Periodic_function : Time_warning_2
   required_mode        : *NORMAL*
   required_mode        : *ILLEGAL*
   occurrence           : whenever timer > 300 seconds
   input                : $limit_time$
   output               : $time_warning$
```

Figure 2-4. Example of a Periodic Function Entity

Periodic_functions use the "occurrence" keyword, and are not related to Input entities. Note that all of the Periodic_functions in Appendix C have a pseudo-input driver, which is used by the implementation for debugging purposes (like $limit_time$ in the example). An "occurrence" is a condition which causes a Periodic_function to become executable, and the corresponding transition to become firable.

For the purposes of this implementation, the class of <function> entities has five relations of interest. They are input, output, required_mode, necessary_condition, and occurrence. All other relations and attributes are ignored for this class of entity; they are meant to be used with other implementations. Activities are generally invoked by the use of "required_mode" Input entities and Periodic_functions are invoked at regular intervals by the use of the "occurrence" condition.

Since functions are mapped to transitions, some relations are mapped to input places and output places of the transition. Other relations result in an annotation of the Petri net.

- **Input places** of each transition are used to model Input entities (ie: from the user's keyboard) and a mode. Input entities can be related to a function by the "input" keyword, to specify data which is required to perform the function. An example would be the Input entity "$name_of_game$"; this input is required before a game can be retrieved by Recreate_board. Input entities can also be related to a function by the "necessary_condition" keyword, which is the ERA method of specifying an input used to initiate an Activity function. An example would be the Input entity $retrieve$, which initiates the Recreate_board Activity. A mode must be related to a function by the "required_mode" keyword.

- **Output places** of each transition are used to model Output entities (ie: to the user's crt) and a mode. Output entities can be related to a function by the "output" keyword, to specify data meant to be displayed by the function. An example would be the Output entity "$status$" which is printed when the Recreate_board Activity is executed. The execution of a function may or may not result in a mode change. In either case, a mode is mapped to an output place of the transition.

- **Annotations** to the transition are used to model Input_output entities and to express the value of the "occurrence" keyword. An Input_output entity can be related to a function by one or

both of the "input" and "output" keywords. These entities are mapped into "function statements", which are meant to be invoked when the transition fires. This is how transformations on internal data are specified in the Petri net model. The "occurrence" keyword is used to express a logical condition, which must be valid in order for a Periodic_function to be initiated.

An entity may have more than one keyword of the same type. When this happens for the "input" and "necessary_condition" keywords, there is a corresponding one-for-one increase in the number of input places to the transition. There is a similar increase in the number of output places when a multiplicity of "output" keywords exists in an entity. For example, "Computer_Move" has three instances of the output keyword. One of these relations is an Input_output entity which will result in an annotation; the other two relations will be mapped to output places of the "Computer_Move" transition. For multiple instances of the "required_mode" keyword, see the next section.

2.1.1.4 *The MODE_TABLE*

Modes provide for a form of sequencing by means of global state transitions. Modes are grouped into classes. A system can be in more than one mode at one time, but there are exclusion relations among the modes which specify the allowable mode combinations. For example, the A-7E aircraft has several mode classes. The aircraft can be in the *DIG* "navigation" mode, and at the same time, the aircraft can be in the "weapon delivery" modes of *A/A Guns* and *Manrip* separately or together [Hen78].

Because modes are related to functions, modes are mapped to places.

The MODE_TABLE is the last section of the ERA specification. It consists of three parts: a list of valid mode names, an Initial_Mode, and a list of Allowed_Mode_Transitions.

The list of valid mode names can be used to verify that all modes related to entities are defined. A mode name is a word composed of capital letters and delimited by asterisks. Because entities are related to modes via the action, assertion and required_mode keywords, there is a built in redundancy which allows for the verification of the consistency of the MODE_TABLE.

The Initial_Mode is used by the interpreter as the place which receives a token from the firing of the initial transition.

The notation of an Allowed_Mode_Transition is:

```
<event>                    :   old_Mode -> new_Mode
```

where <event> can be the name of an Input or Output entity. The Input or Output entity may also "contain" a value. Examples of Allowed_Mode_Transitions are:

```
$syntax_error$             :   *NORMAL* -> *ILLEGAL*
$status$ = 'checkmate'     :   *NORMAL* -> *END*
```

While a mode change generally occurs in conjunction with the firing of a transition, it should be stressed that mode changes are caused by the occurrence of input and output, and not solely by the firing of some transition. A similar concept is utilized in the A-7E specifications [Hen80], via the use of output-driven transition tables.

In terms of the Petri net model, if the mode table contains a mode change related to the occurrence of a function, then the "old_Mode" is mapped to an input place of the function transition and the "new_Mode" is mapped to an output place. Note that the old_Mode should correspond to the value of the required_mode keyword. If the mode table does not contain a mode change related to the occurrence of a function, then both the input and output places of the function transition map to the value of the required_mode keyword. This temporary consumption of a mode token is done to prevent two transition sequences which are firable in the same mode state from firing concurrently.

If a function has more than one required_mode, then separate copies of the function entity are cloned for each of the required_modes. This is done so that the mode value is preserved through the transition, when there is no change in mode value.

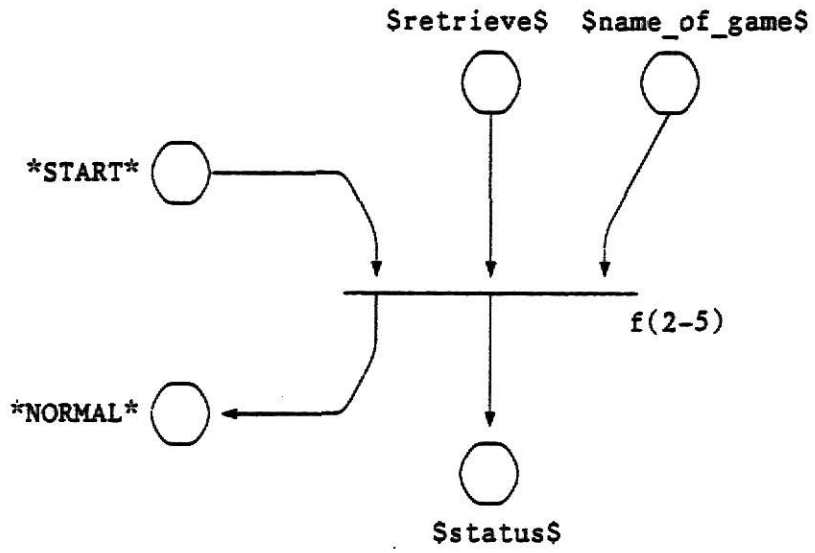
2.1.2 *Mapping ERA onto a Petri Net Graph Model*

This section graphically displays the Petri net transitions corresponding to Figures 2-3 and 2-4. The examples illustrate the annotations and the use of multiple input and output places.

Referring back to the Activity entity in Figure 2-3, note that Recreate_board has three input places and two output places. Two of the input places are Input entities: $name_of_game$ and $retrieve$, (the values of the input and necessary_condition keywords, respectively). The other input place is the required_mode *START*. One of the output places is the output relation $status$. The other output place is the *NORMAL* mode. The action keyword is ignored; the mode change is a result of the

following entry in the MODE_TABLE:

```
$retrieve$                    : *START* -> *NORMAL*
```



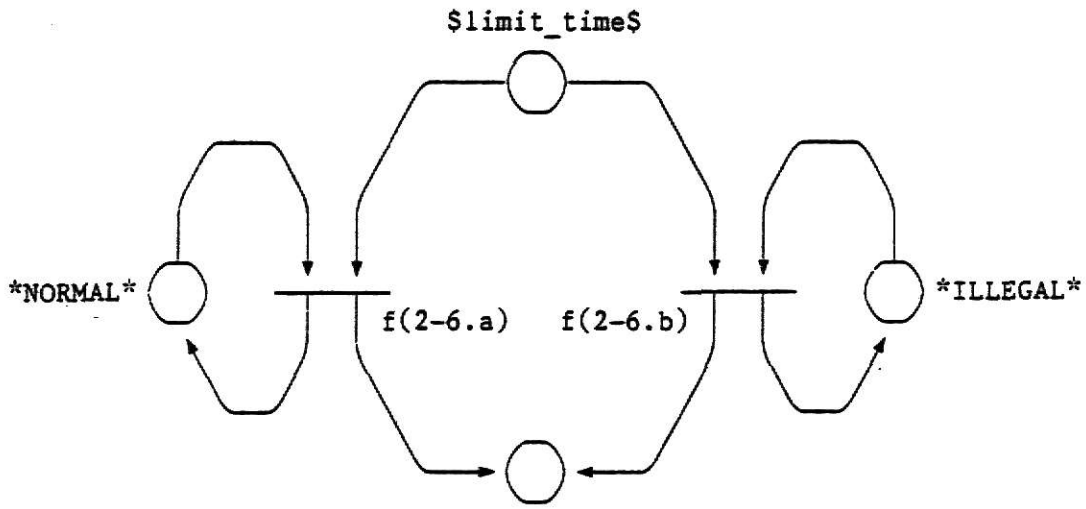
f(2-5): $chess_board$ = Recreate_board ($stored_board$)

Figure 2-5. Petri Net Graph of a Demand Function

The Petri net graph fragment corresponding to the Demand function is shown in Figure 2-5, where the Recreate_board entity is annotated by the function f(2-5), which is meant to be invoked when the transition is fired. This is because the Input_output entities $stored_board$ and $chess_board$ are listed as input and output relations, respectively.

Referring back to the Periodic_function entity in Figure 2-4, note that Time_warning_2 has three input places and two output places. One of the input places is the Input entity $limit_time$, which is the value of the input keyword used in debugging the implementation. The other input places are the required_modes *NORMAL* and *ILLEGAL*. As discussed in the MODE_TABLE section above, there must be a separate entity for every required_mode.

Consequently, Time_warning_2 will clone into the two entities called Time_warning_2_*NORMAL* and Time_warning_2_*ILLEGAL*. Because there is no mode change related to this entity, the Periodic function must preserve the value of the mode through the transition. The Petri net graph in Figure 2-6 shows that the state of the mode is preserved because of the duplication of this transition. The two new transitions share the input place associated with the input relation $limit_time$, as well as the output place associated with output relation $time_warning$. In summary, the net result of executing this Periodic function is consistent, independent of the mode.



f(2-6.a): Time_out_2_*NORMAL* WHEN (timer > 300 seconds)
f(2-6.b): Time_out_2_*ILLEGAL* WHEN (timer > 300 seconds)

Figure 2-6. Petri Net Graph of a Periodic Function

The Petri net graph fragment corresponding to this Periodic function is shown in Figure 2-6, where the Time_warning_2 entities are annotated by f(2-6.a) and f(2-6.b). The WHEN condition is the same for both transitions; it must be valid before the transition

may be fired. The WHEN condition is nothing more than the text following the ERA reserved word "whenever" in the occurrence relation. This implementation will not make use of this facility; a pseudo-input command (ie: $limit_time$) will be used to simulate the calling of the Periodic function.

2.1.3 *Translator Diagnostics and Static Analysis*

In addition to the primary output of the translator, (the PNL code), the translator outputs a number of error and diagnostic messages. Some of these messages correspond to fatal conditions which cause the process to terminate immediately after printing an "error" message. Less severe conditions are detected by a static analysis of the Petri net; the translation proceeds with the production of the PNL code, after printing a "diagnostic" message. All fatal error messages and non-fatal diagnostic messages are detailed in Appendix F.

Fatal errors are most frequently found while reading the ERA input. Examples of such fatal errors include a syntax failure in the ERA file or the referencing of an entity which is not defined. Other fatal internal errors are indicated by the printing of the message "Memory fault", for such conditions as exhaustion of an internal table or other unexpected internal faults.

The diagnostics which are produced by static analysis are related to such Petri net properties as reachability, starvation and deadlock. Other conditions which are related to problems caused by the use of the Petri net model are also flagged. In particular, when two transitions share an Input place (ie: $name_of_game$), removal of the token from this place can cause anomalous behavior

in the simulation. Note that, while PNL intermediate code is produced, the interpreter may not produce a satisfactory simulation of the specification when static analysis indicates that a problem area exists.

Static analysis of the Petri net can provide insight into problem areas in the requirements specification. While static analysis of Petri nets was not the primary focus of this implementation, many of the conditions are quite easy to detect, given the correct data structure design.

2.2 PNL Intermediate Code

PNL intermediate code is a character string language which directly represents Petri net graphs, such as those generated from an ERA specification. The syntax of the language is described elsewhere [Nel82, Nel83].

The subset of the PNL language used in this implementation has four statements. The two PNL statements "TRANS" and "PLACE" correspond to Petri net transitions and places. The other two PNL statements are "INIT" and "TERM", which correspond to initial transitions with no input places and terminal transitions with no output places.

There are five modifiers to the above statements.

The most common modifier is "OUTPUT TO", which lists the output places of a transition, (as well as the output transitions of a place). OUTPUT TO is used to modify the INIT, TRANS and PLACE statements. The "INPUT FROM" modifier provides the complementary information of the OUTPUT TO modifier and is used to modify the

TRANS and TERM statements.

The other three modifiers are used as annotations on the TRANS statement only. They are the "OUTPUT IF", "INVOKES" and "WHEN" modifiers. The OUTPUT IF modifier conditionally places a token in one of two mode places. The INVOKES modifier supplies a function which is to be invoked upon the firing of the associated transition. The WHEN modifier supplies a conditional expression which must be satisfied before the transition may fire.

It should be noted that two features are not a part of the original definition of PNL. The first feature involves the use of the WHEN modifier, which gives the ability to wait for the satisfaction of some condition. The second feature involves the use of "variables". This implementation uses variables to represent Input_output entities. These variables are used as the parameters and results of the functions in instances of the INVOKES modifier. The variables are also used in instances of the WHEN condition for the storage of internal values (ie: "timer" > 300 seconds). Both the WHEN modifier and variables are described elsewhere in the literature [Kel76].

The OUTPUT IF, WHEN and INVOKES annotations are not currently analyzed by the interpreter. The annotations are simply displayed to aid the user in the simulation of the requirements specification prototype.

Two examples follow, in which the PNL code associated with previous examples is shown.

```
T_INIT           : INIT  OUTPUT TO ( *START* )
*START*          : PLACE OUTPUT TO ( Recreate_board, ... )
*NORMAL*         : PLACE OUTPUT TO ( ... )
$retrieve$       : PLACE OUTPUT TO ( Recreate_board )
$name_of_game$   : PLACE OUTPUT TO ( Recreate_board )

Recreate_board   : TRANS OUTPUT TO ( *NORMAL*, $status$ )
                   INPUT FROM ( *START*, $retrieve$, $name_of_game$ )
                   INVOKES ( $chess_board$ =
                             Recreate_board ( $stored_board$ ) )

$status$         : PLACE OUTPUT TO ( T_TERM_$status$ )
T_TERM_$status$  : TERM  INPUT  FROM ( $status$ )
```

Figure 2-7. PNL code for a Demand Function

Figure 2-7 illustrates the PNL code generated for the Demand function modeled by the Petri net in Figure 2-5. There are a number of items of interest in this figure. The transition INVOKES a function with an Input_output entity as its input and a different Input_output entity as its output. A mode change from *START* to *NORMAL* occurs. Note that the ellipses in the OUTPUT TO lists for these modes suggest that the modes are input places for transitions not shown.

In both of these examples, Output entities (ie: $status$ and $time_warning$) serve as the input places to a terminal transition, denoted by a TERM statement. The TERM statement will remove tokens from the Output entity place, to simulate the printing of the Output entity on the crt. The label of the TERM statement will be the string "T_TERM_" with a suffix of the entity name.

Figure 2-8 illustrates the PNL code generated for the Periodic function modeled by the Petri net in Figure 2-6. In this example, both of the transitions are annotated with the same WHEN modifier. This was done to preserve the mode's value through the transition.

```
*NORMAL*                : PLACE OUTPUT TO ( Time_out_2_*NORMAL*, ... )
*ILLEGAL*               : PLACE OUTPUT TO ( Time_out_2_*ILLEGAL*, ... )
$limit_time$            : PLACE OUTPUT TO ( Time_out_2_*NORMAL*,
                                            Time_out_2_*ILLEGAL* )

Time_out_2_*NORMAL*     : TRANS OUTPUT TO ( *NORMAL*, $time_warning$ )
                          INPUT FROM ( *NORMAL*, $limit_time$ )
                          WHEN ( timer > 300 seconds )

Time_out_2_*ILLEGAL*    : TRANS OUTPUT TO ( *ILLEGAL*, $time_warning$ )
                          INPUT FROM ( *ILLEGAL*, $limit_time$ )
                          WHEN ( timer > 300 seconds )

$time_warning$          : PLACE OUTPUT TO ( T_TERM_$time_warning$ )
T_TERM_$time_warning$   : TERM  INPUT  FROM ( $time_warning$ )
```

Figure 2-8. PNL code for a Periodic Function

Diagnostic warnings and comments may be inserted in the PNL intermediate code by the translator. The diagnostic warnings are the result of the static analysis phase of translation. The comments are automatically inserted by the translator to make the PNL intermediate code more readable. They are denoted by a '-' and a '#' in column 1 respectively, and are documented in Appendix F.

2.3 The Interpreter

The interpreter simulates the Petri net represented by the file of PNL instructions. This simulation provides a prototyping of the ERA specification.

The simulation is accomplished by firing any firable transitions and printing the names of the output places of those transitions. The normal output of the simulation is a sequential list of entity names associated with input and output places, showing the external behavior of the prototype. This firing trace of the input and output places of the transitions will simulate a trace of the external behavior of the software product, as the software product

has been specified. It is hoped that deficiencies in the specification will be become apparent through this simulation.

When more than one firable transition is available, the interpreter will fire the first firable transition it encounters.

The user of the interpreter is expected to thoroughly test combinations of transition firings. This includes those transitions which represent Periodic_functions. A condition specified by the "occurrence" relation in a Periodic_function (and represented by the WHEN modifier) can become true at any time. The interpreter makes no attempt to analyze the WHEN condition for validity. Instead, a pseudo-input command is associated with Periodic_function transitions. While the interpreter is not aware of any difference between Periodic_functions and Activities, the WHEN annotation is displayed in the menu to indicate to the user that a Periodic_function transition is potentially firable. Thus, the interpreter leaves the testing of the condition specified in the WHEN modifier to the user.

When a function is "executed" by means of an INVOKES modifier, the only discernible result is the printing of the function's name and parameters. The user of the interpreter is expected to comprehend the result of executing a function which operates on Input_output entities.

When there are no firable transitions, transitions which *could* become firable are considered.

The interpreter uses a menu driven system. The menu will display a list of these potentially firable transitions, including any

annotations to those transitions. (The annotations printed in the menu are the WHEN and INVOKES modifiers.) These transitions correspond to the functions which require a user request as a "necessary condition" prior to becoming firable. The user selects the desired transition from the menu, and a token is placed in the input place which represents the appropriate necessary condition. The interpreter then resumes firing the net.

The interpreter can provide indications of problems such as deadlock and starvation. The first condition occurs when there are no firable transitions. The second occurs when one section of the Petri net monopolizes the firings without allowing other sections of the net to become firable.

A number of options are available to the user of the interpreter.

An option to trace the transitions as they are fired is available, to debug the requirements specification. The output of this trace adds the Activity and Periodic_function transition names to the output list. These names are interleaved with the inputs and outputs that occur, so that the ordering of events can be seen, relative to those inputs and outputs.

Another option exists, which allows the user to "single_step" through the firing of the net. A menu of all firable transitions and *potentially* firable transitions is displayed. This allows the user to fully select the order of transitions firings. A variety of firing orderings can be tried, which allows the user to simulate the random concurrent nature of the Petri net. This option is particularly useful if one transition is "starving" the net.

There are also options to aid in debugging the implementation itself. For instance, the ability exists to print out the current state of the Petri net at any time. This option displays the placement of tokens and the firable transitions in the net.

2.4 Miscellaneous Information

All software for this implementation is to be written in the C language [Ker78] on the UNIX Operating System [Chr83]. The software must be executable on "version V" UNIX (Bell Laboratories internal standard), as well as the "4.2 BSD" UNIX used at Kansas State University.

A number of standard UNIX commands will be utilized in this implementation. Make(1) will be used to describe the dependencies between the files that comprise this implementation. Lint(1) will be used to eliminate incorrect and wasteful code.

The software will utilize the "debug" facility. This facility allows for the selective printing of debugging statements during the execution of the processes. The "trace level", which specifies the type and quantity of debugging statements, is fully selectable at run time. No recompilation of the software is required to select trace levels.

The ERA requirements specifications listed in the appendices of this report have actually been tested as input to the implementation. The implementation silently ignores nroff commands (ie: ".bp" for pagination) which have been inserted in the specifications files for formating purposes. Furthermore, all

input and output files used by the programs that comprise this implementation shall be readable by ordinary text editors.

CHAPTER 3 - DESIGN

The implementation contains three modules which perform the processes of Translation and Interpretation described in the previous chapter. The "scanner" module lexically scans an ERA requirements specification. The "translator" module generates a file containing PNL code. The "interpreter" module simulates the Petri net described by the PNL code in order to provide a prototype of the requirements specification. This chapter describes the algorithms and data structures used by these modules.

3.1 Implementation Hierarchy

Each module has a particular task it *performs* upon data structures "owned" by that module. In general, other modules must use access routines which are provided by the module owning a data structure, if an action needs to be performed involving that data.

Figure 3-1 shows a hierarchy diagram for the implementation. The lowest level of the diagram represents data files and data structures. The tasks which use the data are shown in the levels above the data.

3.2 The Main Module

The main module handles the initialization, control and termination of the process. The main module uses two primary modules: the translator and the interpreter.

The translator uses the scanner module to build the symbol table. With the symbol table, the translator constructs an entity graph,

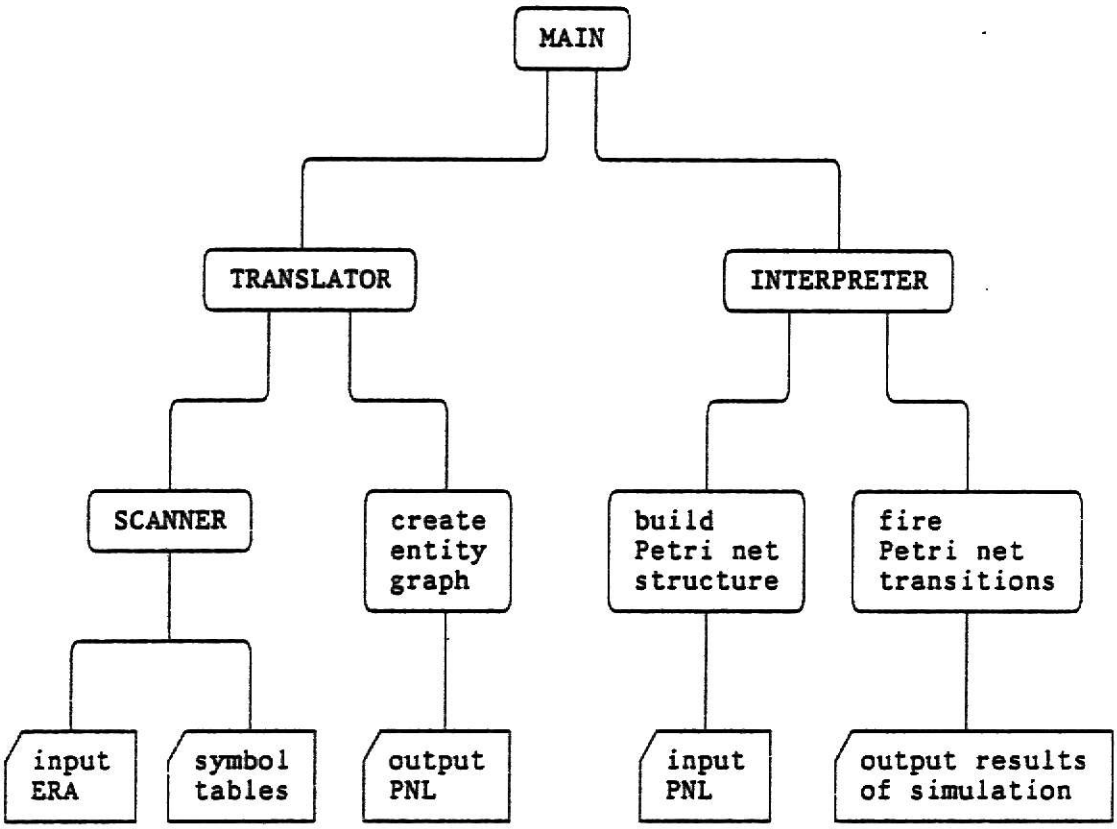


Figure 3-1. Hierarchy Diagram

which expresses the Petri net relationships between entities, and then builds a file of PNL instructions corresponding to this graph. The interpreter fires the Petri net structure which is produced by the translator. Each of the following sections discusses one of these modules, emphasizing the data structures "owned" by that module, and the algorithms used by the module to maintain those data structures.

3.3 The Scanner

The scanner reads entities from the ERA file and stores them in an internal data structure format suitable for use by the translator. A Symbol table is maintained as well as a table for functions.

The scanner in this implementation is analogous to the "lexical scanner" used in many compilers [Aho78]. While it reads the input in one pass, it differs in that it does not pass tokens to a parser; the Symbol table is built in such a way as to allow the translator to produce intermediate code without the need for a parse tree.

It should be noted that, while the input language is suitable for recursive descent parsing, the grammar in Appendix B is ambiguous. Because of this, the language is not LL(1).

The scanner is capable of handling forward references in its one pass through the input text. This implies that the entities can occur in any order in the ERA file.

3.3.1 *The Scanner's Algorithm*

The scanner module reads the entities and MODE_TABLE found in the ERA specification file. If this input satisfies the syntax for ERA specifications, expressed in Appendix B, the entities and modes are entered into a Symbol table.

Function entities have "relations" which express interrelationships with other entities. Some relations are expressed via indices to the Symbol table entries which correspond to those other entities. Other relations can result in the generation of an "annotation" to the function. Annotations express the PNL concepts of conditional transition firings and function invocations, as well as comments and warnings.

3.3.1.1 *Scanning the ERA file - Entities*

The scanner processes the entity section of the ERA file first, removing the PROCESS line and comments. As the entities are read, the ordering of the input lines is checked. There are three types of lines which naturally occur in an entity: "blank" lines, "Entity" lines and "relation" lines. The lexical format of words within these lines is also verified.

Some entities are of interest to this implementation, and some are meant for other purposes. If the entity matches the syntax and the entity keyword is found in array of valid entity names, then the entity *is processed.* A similar algorithm is used to select relations which are of interest, within valid entities.

As each valid entity is consumed from the input ERA file, its values are put directly into two tables. Information concerning valid entities is stored in a "Symbol table" called SYMTAB. This table is sufficient for the storage of all entities except function entities. Information about valid relation values found within functions is put in a "Function table" called FUNTAB.

Some of the fields in the FUNTAB table contain the values of relations. These values are stored as indices to Input, Output and Mode names in the SYMTAB table. Other fields are derived from information in the relations. For instance, when a Periodic_function entity is encountered, the value of its "occurrence" keyword is used to build the WHEN annotation. As shown in Figures 2-4 and 2-6, the WHEN annotation is the text following the word "whenever" in the occurrence string.

The scanner must wait to fill in some of the information in the FUNTAB table. The building of the INVOKES annotation must wait until all Input_output entities have been read. Likewise, the value of the function's "next mode" will not be filled in until the MODE_TABLE has been read.

As was mentioned in Chapter 2, if a function has more than one required_mode, the entity must be "cloned" so that each of the new entities has only one required_mode. For example, in the Periodic_function "Input_Check", the value of required_mode is the reserved word "every_mode". This entity will be cloned into four entries in the SYMTAB table, because there are four modes in the ERA specification of Appendix C. Each of the new entries will have the name of Input_Check with a suffix of the corresponding mode name. The division of an entity like Input_Check must be delayed until the list of mode names in the MODE_TABLE has been processed.

3.3.1.2 *Scanning the ERA file - Modes*

The MODE_TABLE is the last section of the ERA file, and it is divided into three parts.

The first part is a list of valid mode names, which is used to perform a consistency check on the rest of the file. The list of valid names is also used to clone function entities which have a required_mode of "every_mode".

The second part is the value of the "Initial_Mode". The scanner saves a index to the SYMTAB table entry which is given as the value of the Initial_Mode. This value will be used by the translator to build the INIT statement.

The third part is the list of "Allowed_Mode_Transitions" which the scanner uses to fill in the "next mode" information in the FUNTAB table. The MODE_TABLE displays mode change information in relation to input and output "<events>" thusly:

<event> : old_Mode -> new_Mode

For every entry in the MODE_TABLE, the functions in the SYMTAB table must be scanned for references to the <event> which causes a mode change entry in the MODE_TABLE. If such a reference is found, and if the required_mode of the function is the same as "old_Mode", then the next mode value in the FUNTAB table will be "new_Mode". If no mode change occurs for a function, the next mode is the same as the required mode.

3.3.1.3 *Completing the Derivation of the Scanner's Tables*

When all of the information has been read from the ERA input file, the symbol tables are checked for internal consistency. The primary task is to locate entities which are referenced, but never defined. These "orphan" entities are given default characteristics, so that processing may continue. Diagnostic messages are issued when such inconsistencies are due to the omission of an entity from the ERA file.

When all of the entities have been consumed, the scanner builds the INVOKES annotation for those functions which are related to Input_output entities. As discussed in Chapter 2, if such an entity occurs as an input relation of a function, that entity becomes the argument of an INVOKES annotation. If such an entity occurs as an output relation, then that entity becomes the result of the INVOKES annotation. The INVOKES annotation is shown in

functional notation:

INVOKES (result = Function_entity_name (arguments))

An example is shown in Figure 2-5. The building of the INVOKES annotations is done in one pass through the SYMTAB table. Another pass through the SYMTAB table is required to remove all references to Input_output entries, because they have no further use.

3.3.2 *The Scanner's Data Structures*

Figure 3-2 displays an example of the two tables which are of primary interest to the scanner module: the SYMTAB table and the FUNTAB table. This example shows the entries related to the scanning of the Recreate_board function in Figure 2-3. The FUNTAB table is shown at a point where all of the values of input and output relations have been fully defined, but annotation and MODE_TABLE processing is not yet complete, as indicated by the values of '?'.

The SYMTAB table contains the name of the entity, a comment, the entity class, and an index to a FUNTAB entry. The name of the entity is a string. The comment field (not shown) is a PNL comment which will be emitted by the translator. The entity class is defined to be one of the following:

F - Function
I - Input
O - Output
I_o - Input_output
M - Mode

The FUNTAB index is used for function entities only, and is an index to a FUNTAB table entry which gives further information about the function.

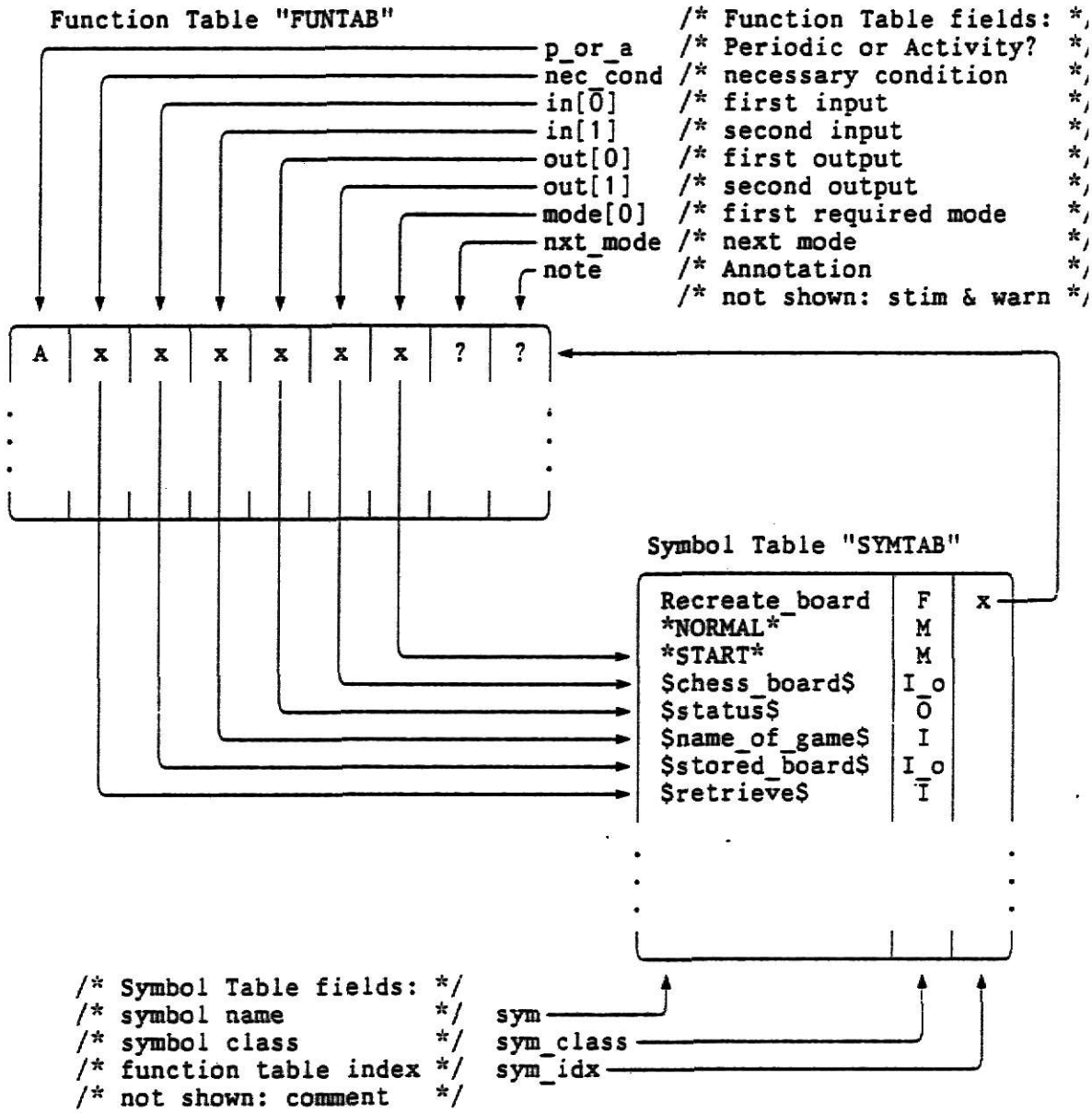


Figure 3-2. The Scanner's Data Structures

In the figure, fields which contain pointers to entries in another table are denoted by an "x" and an arrow pointing to that entry. All pointers are implemented via the use of indices into tables, to avoid the delays that address exceptions can cause in the development cycle.

The FUNTAB table has the fields that are enumerated at the top of Figure 3-2. Some of the fields directly correspond to relations in the entity, and thus contain SYMTAB indices. The "p_or_a" field is a flag indicating whether a function is a Periodic_function or an Activity. The "in", "out" and "mode" fields are implemented as arrays, because an entity can have more than one of these relations. The "note", "stim" and "warn" fields are strings containing various annotations to the function.

As entity names are encountered, either by definition or reference, the SYMTAB table is searched for that name. If it is not found, it is added to the tables.

3.3.3 *The Scanner - Error Handling*

The scanner module must be able to detect more error conditions than any of the other modules. This is because the scanner obtains its input from a file which has been manually built. The fatal errors which can be detected are caused by syntactic or semantic errors in the input, or are the result of limitations in the internal symbol tables.

Syntactic errors can be detected when the ERA input does not follow the grammar specified in Appendix B. A syntax error is generated when the ordering of the sections within the file is incorrect. For instance, the MODE_TABLE must follow the entities. A syntax error is also generated when the ordering of lines within a section, or the structure of words within a line, is incorrect. For example, the lines of an entity must occur in a certain order, and each line must have a "left-hand-side" and a "right-hand-side" which are separated by a colon. The left- and right-hand-sides

must also follow certain lexical conventions (ie: modes are delimited by '*' characters).

Fatal errors are also generated when semantic inconsistencies are detected. These inconsistencies usually involve the use of an improper relation in an entity definition. For instance, the "necessary_condition" keyword is only valid within Activities and the "occurrence" keyword is only valid within Periodic_functions.

Other fatal errors are related to the SYMTAB and FUNTAB tables. The number of entries in these tables and the size of fields in each entry are both limited. When the limits are exceeded, the process terminates.

The scanner also has the ability to detect certain non-fatal errors. When this happens, a diagnostic warning message is generated and processing continues. For instance, a diagnostic warning is issued for Periodic_functions which have an "input" relation, because Periodic_functions are not supposed to accept input from the keyboard. The message which is generated is a PNL warning, which is stored in the FUNTAB entry for a function. This PNL warning will be emitted by the translator when the PNL code for the associated transition is generated.

3.4 The Translator

The translator module calls the scanner module, which reads the ERA file, and builds the SYMTAB and FUNTAB tables. The translator then builds an "entity graph", which represents the ERA specification as a Petri net. The entity graph is used to generate an output file containing a PNL program.

3.4.1 *The Translator's Algorithm*

The translator makes one pass through the SYMTAB table to generate the entity graph from function entities. The entity graph is a doubly-linked list representation of a Petri net, which expresses the relationships between transitions and places. In the entity graph, transitions represent ERA functions and places represent Input/Output related entities and modes.

The translator uses the entity graph to produce an output file of PNL instructions. After the INIT transition is written to the file, the translator makes three passes through the entity graph to generate the rest of the PNL output file. One pass is required to produce each of the PLACE, TRANS and TERM sections in the output file.

3.4.1.1 *The Translator - Building the Entity Graph*

The entity graph contains a table for transitions (TRANS) and a table for places (PLACES). The TRANS table maintains lists of input and output places, for each transition. While this linkage is sufficient to assure that the data structure is a Petri net structure, the PLACES table also maintains a list of output transitions, for every place. This double-linkage simplifies the translator's algorithm for printing the "OUTPUT TO" list for PLACE statements, as well as the interpreter's algorithm for firing the Petri net.

The graph is built by searching the SYMTAB table for functions. When a function is found, an entry is added to the TRANS table. The function will have relations which are indices to other entities in SYMTAB; these are added to the lists of input and

output places for that transition. If the function has relations which are annotations, then these are copied directly into fields in the transition entry.

The lists of input and output places for a transition are stored in the TRANS table entry as indices to the PLACES table. Input and output places are added to the PLACES table when they are first referenced by a function. The PLACES table entry will also contain an index to the TRANS table, for input places to transitions (to provide for double-linkage).

When all transitions have been added to the TRANS table, a search for unreachable nodes is performed. Any entity in the SYMTAB table which has not yet been entered into the PLACES or TRANS tables is considered to be unreachable. Such entities are added to the entity graph with the PNL warning that the corresponding entity graph node will never be accessed by any combination of firings.

When the TRANS and PLACES tables have been completely built, the translator is ready to generate the output file of PNL code.

3.4.1.2 *The Translator - The Initial Transition*

The INIT transition is the first instruction to be put in the file containing the PNL intermediate code:

```
T_INIT          : INIT  OUTPUT TO ( *START* )
```

The INIT statement is a special transition with no input places. Its only output place is the value of Initial_Mode, which was saved by the scanner during the processing of the MODE_TABLE. Firing the INIT transition will clear the tokens from the net, and place one token in the Initial_Mode output place. This will cause the net to

become activated.

3.4.1.3 *The Translator - Pass 1 - Places*

Once the special INITial transition has been put in the output file, the place statements must be generated. An example of a place statement is:

```
$name_of_game$ : PLACE OUTPUT TO ( Recreate_board )
```

One pass is made through the PLACES table to generate the place statements. Entries in the PLACES table which have a comment annotation will have that comment printed before the place statement is emitted. The only modifier to the place statement is "OUTPUT TO", which supplies a list of transitions that can remove a token from that place.

While a place can have tokens placed in it by a transition, no "INPUT FROM" list is generated for the place statement. This is because the list would be empty (or trivial) for the majority of places. Frequently, input places of transitions will receive their tokens from the virtual transition of "Keyboard". These input places would have an empty INPUT FROM list, because the Keyboard transition is simulated within the interpreter. The Keyboard transition statement is not emitted to the output file.

3.4.1.4 *The Translator - Pass 2 - Function Transitions*

Transition statements are generated after the place statements have been put in the file of PNL intermediate code. An example of a transition statement is:

```
Recreate_board : TRANS OUTPUT TO ( *NORMAL*, $status$ )
                 INPUT FROM ( *START*, $retrieve$, $name_of_game$ )
                 INVOKES ( $chess_board$ =
                           Recreate_board ( $stored_board$ ) )
```

One pass is made through the TRANS table to generate the transition statements. Comment or warning annotations about a transition will be printed before a transition statement is emitted. Transition statements can be adorned with any of the five modifiers of PNL statements discussed in the previous chapter.

3.4.1.5 *The Translator - Pass 3 - Terminal Transitions*

The TERM statement is the other special transition, and is the fourth kind of instruction to be put in the file. An example is:

```
T_TERM_$status$ : TERM  INPUT FROM ( $status$ )
```

One pass through the TRANS table is required to generate all of the terminal transitions.

These transitions have no output places. For every Output entity, there is a unique terminal transition statement. The Output entity is the only input place of the TERM transition. The label of the TERM statement is the string "T_TERM" with the suffix of the Output entity name. Firing the TERM transition will cause tokens to be removed from the OUTPUT place, which will simulate the printing of the output on a crt.

See Figures 2-7 and 2-8 for related examples of PNL statements.

3.4.2 *The Translator's Data Structures*

Figure 3-3 displays the SYMTAB and FUNTAB entries representing the Recreate_board example once again. These entries have now been completely processed by the scanner, and are ready to be used by the translator to create the entity graph. The entries in the

FUNTAB and SYMTAB tables are essentially the same as those shown in Figure 3-2. Note that the next mode and annotation information is now complete. Since the annotation is completed, the Input_output entities $chess_board$ and $stored_board$ are no longer used.

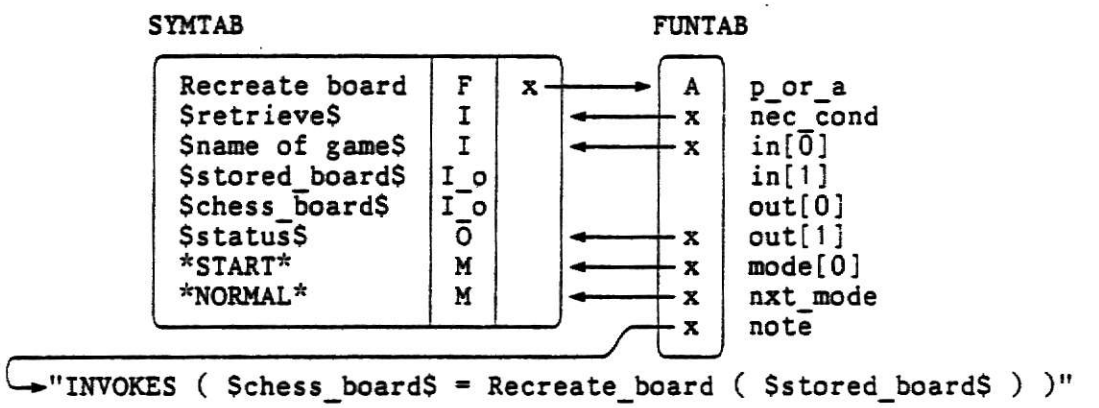


Figure 3-3. The Scanner's Data Structures, revisited

Figure 3-4 displays an entry in the entity graph corresponding to the SYMTAB and FUNTAB table entries shown in Figure 3-3. The entity graph is composed of two tables called PLACES and TRANS, which show the relationships between places and transitions.

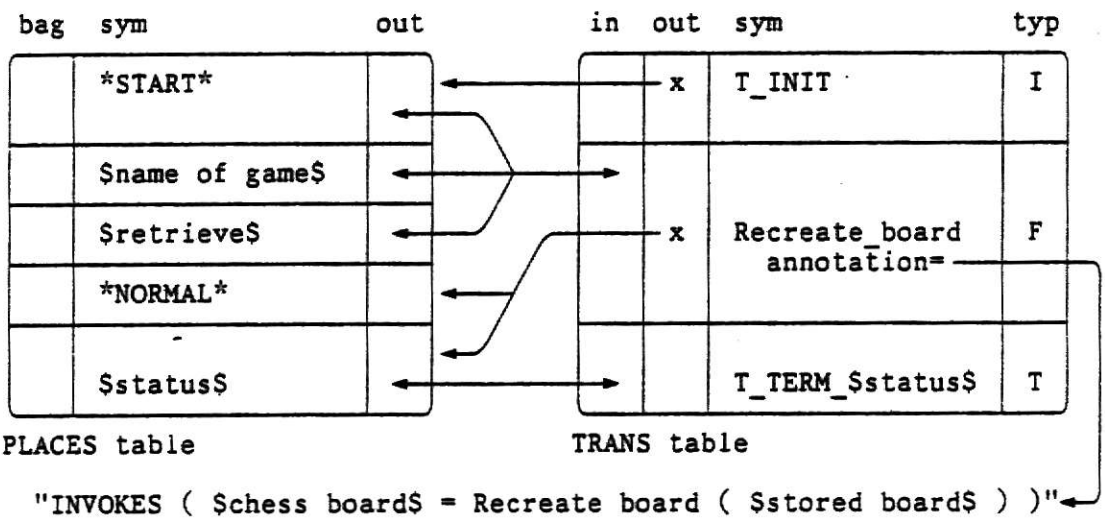


Figure 3-4 The Translator's Data Structures

In these figures, lines with arrows at both ends denote entities in two tables which point at each other; the two entities are said to be doubly-linked. As before, an "x" connected to an arrow which points to the "edge" of an table entry is a singly-linked index to that entry. Lines which terminate in more than one arrow are implemented as an array of indices; the "in" field in the TRANS table is an example.

The PLACES table has fields labeled "bag", "out" and "sym". The bag field is where the number of tokens in a place will be stored. If a place is an input place of a transition, then the "out" field of its corresponding PLACES table entry will be an index to the TRANS table entry. The two tables are doubly-linked between PLACES.out and TRANS.in. The sym field is a string containing the place name. There is also a comment field (not shown).

The TRANS table has fields labeled "in", "out", "sym" and "typ". The in field is a doubly-linked pointer to input places of the transition. The out field is a singly-linked pointer to output places of the transition. The sym field is a string containing the transition name. The transition typ is defined to be either an Initial (I), Function (F) or Terminal (T) transition. The I and T transition types are special cases of the general F type transition. There are also fields (not shown) which can contain annotations, mode transition stimuli, comments and warnings.

The example in Figure 3-4 shows that the token flow proceeds from the T_INIT transition to the *START* place, and then from the *START*, $retrieve$ and $name_of_game$ places to the Recreate_board transition. Recreate_board puts a token into the *NORMAL* and

$status$ places, and the $status$ place outputs a token to the T_TERM_$status$ terminal transition. Putting a token in T_TERM_$status$ simulates printing the $status$ message on the crt.

3.4.3 *The Translator - Error Handling*

The translator does not need to perform as much defensive error checking as the scanner does. The SYMTAB and FUNTAB tables, which the translator uses to build the entity graph, are already free of inconsistencies. The only fatal errors generated by the translator are due to limitations in the size of the entity graph as it is being built.

The translator has the ability to detect certain non-fatal errors as the entity graph is being built. This is called "static analysis" of the Petri net structure. For instance, when a node in the entity graph is found to be unreachable, a diagnostic warning is generated. This PNL warning message will be emitted by the translator when the PNL code for the associated node is generated.

3.5 **The Interpreter**

The interpreter module uses the entity graph which was created by the translator module. The entity graph is a Petri net structure which must be initialized and then interactively fired, guided by menu-driven input from the user.

3.5.1 *The Interpreter's Algorithm*

The net is initialized by clearing all places of tokens. When all bags are empty, the T_INIT transition is fired. Firing this transition will cause a token to be placed in the bag of the *START* place.

The main loop of the interpreter is then entered. The following activities will be performed until the process is terminated.

- The user is given a prompt. At this point, the user can ask for instructions, modify the interpreter's options or depress carriage return to continue with this iteration.

- The Trans table is searched for firable transitions. A firable transition is one which has a token in the bag of each of its input Place table entries. If the trace mode is enabled, a menu of these transitions is displayed, and the user can select the transition to be fired. If trace mode is not enabled, the first transition which is firable is fired.

- If no firable transitions exist, a menu of transitions which are potentially firable is displayed. These transitions will become firable if a token is placed in the bag of a Place table entry which is an input place of the transition.

- After firing the transition, the Trans table is searched for firable terminal transitions. A terminal transition is firable if its input place has a token. Since its input place is an Output entity, the name of the entity is displayed when the transition fires.

When the user elects to cease the firing of the net, statistics concerning the simulation are printed. In particular, the number of transitions which were fired, and the number of tokens added and removed from the net are displayed in a summary format.

3.5.2 *The Interpreter's Data Structures*

The interpreter uses the entity graph which was created by the translator. In addition to the PLACES and TRANS tables, the interpreter maintains the MENU array, which contains indices to the TRANS table. The MENU array is a list of transitions which are enabled, or which could become enabled if input occurs. When tracing is enabled, the user is presented with a list of the transitions pointed to by the MENU array.

When a transition is selected for firing, a token is removed from the bag of each of the transition's input places and a token is added to the bag of each of the transition's output places.

3.5.3 *The Interpreter - Error Handling*

The interpreter needs to perform defensive error checking on the user's input as the main loop of the interpreter is executed. The entity graph which the interpreter fires as a Petri net structure is free of inconsistencies, because of the way in which it was built.

The interpreter has the ability to detect certain non-fatal errors as the entity graph is being fired. This is called "dynamic analysis" of the Petri net structure. For example, when a node (ie: Computer_Move) in the entity graph is found to be constantly enabled, a diagnostic warning is generated. The diagnostic will suggest that the node is starving the rest of the net.

3.6 Miscellaneous Information

The init() routine in the main module calls the initialization routine for each module. Likewise, the fini() routine in the main module is called to perform the "cleanup" routines for each module when the process is finished. The opening of files and initialization of data structures is the responsibility of each module, as well as the closing of files and other termination activities.

Very little error recovery is performed in this implementation. The modules are able to generate diagnostic messages, and continue processing, when certain irregular conditions are detected. However, the majority of "unexpected events" are considered to be fatal errors. Unexpected events include those described in the error handling sections above, as well as internal errors such as invalid cases in switch statements and table exhaustion. When such an unexpected event occurs, the typical response is a call to the "ERROR()" macro. This routine prints the fatal error string which is its argument and calls fini(), which performs a thorough process shutdown. All error and diagnostic messages are explained in Appendix F.

It should be noted that the data structures used are limited, and that they could be made more general by the addition of a table management module.

CHAPTER 4 - EXTENSIONS AND CONCLUSIONS

4.1 Extensions to this Implementation

A number of extensions could be made to improve the quality and power of this implementation. Extensions of this work include making the implementation more robust, expansions to the implementation, and utilization of some of the powerful constructs seen in variations on specification models discussed in the literature. The direction that these extensions ultimately take suggests the development of a unified Software Engineering Environment which would aid in the management of the phases of the prototyping software life-cycle.

4.1.1 *Making the Implementation More Robust*

There are a large number of areas within the implementation which could be made more robust. The ability to handle arbitrary numbers of entities, and relation lines within entities, could be implemented. The same is true of logical expressions of arbitrary complexity in the values of relations. In particular, a parser could be written which could compile an extended syntax of the Entity-Relationship-Attribute language, using a predictive parse table. The extended syntax could incorporate a large number of powerful features. Furthermore, the parser could be a generic tool which would output intermediate code useful to other tools in a Software Engineering Environment. An example might be a tool which automatically generated a design specification skeleton based upon the output of the translator.

4.1.2 *Expansions to the Implementation*

Rather than using a menu driven system in the interpreter, the simulation of the Petri net model could be done by firing a random sequence of enabled transitions. The random nature of the simulation would provide a more realistic test of the prototype system. Because the simulation would be left under the control of the interpreter, a much larger set of firing sequences could be observed.

As transitions were fired, a data base of firing sequences could be built, and patterns in those sequences could be identified. This would be useful for detecting scenarios which were characteristic of problem areas. For instance, the starvation caused by Computer_Move in Appendix C would be automatically flagged. It would also be useful to develop typical histories of firing sequences for use while testing the actual software product, once it is developed.

The interpreter could be extended to incorporate features currently used in debugging and simulation tools. A desirable feature would be the ability to set breakpoints on transitions, so as to examine the state of the system in the midst of a simulation. Also of interest would be the ability to alter the condition of the system "on the fly", while the simulation of the software product is in progress.

As an extension of the previous item, Periodic_functions could be modeled using a Petri net which is concurrent to the primary net. The interpreter would then take on the characteristics of a "multiple-process debugger". For instance, the timer that Time_out

refers to could be simulated as a separate net. Currently Periodic_functions must be manually initiated by issuing a "pseudo-input" command.

4.1.3 *Variations on the Models*

The ERA specification language could be extended to incorporate many of the techniques used in the specification of the A-7E aircraft [Hen80]. These include transition, event and condition tables.

There are a number of functionalities of the augmented PNL which were not utilized in this implementation. For instance, "inhibitor" arcs [Nel82] could be used to "disable" certain functions in the PNL code associated with an ERA specification. ("Inhibitor" arcs are directed from a place to a transition; when a token is in the place, the transition is not enabled.) Other extensions might include the use of "colored tokens" [Pet81] to represent the flow of various token types through the net, (where a token could be a Mode, an Input_output entity, etc.). Many other extensions to Petri Net Modeling systems exist in the literature.

4.2 Conclusion

This report discusses the implementation of a tool which analyzes software requirements which are in an "Entity-Relationship-Attribute" (ERA) model. The ERA software requirements are translated into an equivalent model based upon Petri nets. The Petri net model was chosen because of its ability to model concurrent activities (such as the interaction between users and software products) in an intuitively satisfying way. The Petri net

model is expressed textually via Petri Net Language (PNL). The PNL code which is generated is suitable for interpretive simulation. The simulation of the specification allows the requirements specifier and the end user to work together on a prototype of the system being specified.

Being able to prototype a specification generates a higher degree of confidence in the software product which will be developed. End users who are unsure of their needs can experiment with various combinations of user interfaces and system functionalities. In addition, areas where implicit misunderstandings exist between customers and developers can be brought into focus. Because a precise specification exists, disadvantages which are associated with rapid prototyping are minimized. By applying techniques similar to those illustrated in this report, the prototyping of specifications for requirements of software products should allow for a determination of aspects of the correctness and completeness of such specifications.

REFERENCES

[Aho78] A. V. Aho and J. D. Ullman,
Principles of Compiler Design,
Reading, Mass.: Addison-Wesley, 1981.

[Boe76] B. W. Boehm,
"Software Engineering",
IEEE Transactions on Computers,
vol C-25, no 12, pp 1226-1241, December 1976.

[Boe84] B. W. Boehm, T. E. Gray and T. Seewaldt,
"Prototyping Versus Specifying: A Multiproject Experiment",
IEEE Transactions on Software Engineering,
vol SE-10, no 3, pp 290-302, May 1984.

[Bra82] M. Branstad and M. V. Zelkowitz,
"Special Issue on Rapid Prototyping",
ACM SIGSOFT Software Engineering Notes,
vol 7, no 5, 185 pp, December 1982.

[Chr83] K. Christian,
The UNIX Operating System,
New York, NY: Wiley-Interscience, 1983.

[Hen78] K. L. Heninger, J. W. Kallander,
J. E. Shore and D. L. Parnas,
"Software Requirements for the A-7E Aircraft",
NRL Memorandum Report 3876,
Naval Research Laboratory,
Washington, D.C. 20375, 11/27/78.

[Hen80] K. L. Heninger,
"Specifying Software Requirements for Complex Systems:
New Techniques and their applications",
IEEE Transactions on Software Engineering,
vol SE-6, no 1, pp 2-13, January 1980.

[Jor76] H. F. Jordan and B. J. Smith,
"The Hardware-Software Interface",
Arbeitsberichte des Instituts fur Mathematische Maschinen und Datenverarbeitung,
Friedrich Alexander Universitat Erlangen Nurnberg,
vol 9, no 4, 260 pp, June 1976.

[Kel76] R. M. Keller,
"Formal Verification of Parallel Programs",
Communications of the ACM,
vol 19, no 7, pp 371-384, July 1976.

[Ker78] B. W. Kernighan and D. M. Ritchie,
The C Programming Language,
Englewood Cliffs, NJ: Prentice-Hall, 1978.

[McC82] D. D. McCracken and M. A. Jackson,
"Life Cycle Concept Considered Harmful",

ACM SIGSOFT Software Engineering Notes,
vol 7, no 2, pp 29-32, April 1982.

[Mye78] W. Myers,
"The Need for Software Engineering",
Computer,
vol 11, no 2, pp 12-25, February 1978.

[Nel82] R. A. Nelson, L. M. Haibt and P. B. Sheridan,
"Specification, Design, and Implementation
via Annotated Petri Nets",
IBM Research Report RC 9317 (#41041), 54 pages
IBM Thomas J. Watson Research Center,
Yorktown Heights, NY, 10598, 3/31/82.

[Nel83] R. A. Nelson, L. M. Haibt and P. B. Sheridan,
"Casting Petri Nets into Programs",
IEEE Transactions on Software Engineering,
vol SE-9, no 5, pp 590-602, September 1983.

[Ogi83] J. W. Ogilvie,
"Petri Net Simulation using Modula-2 Sets",
Journal of Pascal and ADA,
vol 2, no 5, pp 35-37, Sept./Oct. 1983.

[Pet81] J. L. Peterson,
Petri Net Theory and the Modeling of Systems,
Englewood Cliffs, NJ: Prentice-Hall, 1981.

[Ros77] D. T. Ross and K. E. Schoman, Jr.,
"Structured Analysis for Requirements Definition",
IEEE Transactions on Software Engineering,
vol SE-3, no 1, pp 6-15, January 1977.

[Tan81] A. S. Tanenbaum,
Computer Networks,
Englewood Cliffs, NJ: Prentice-Hall, 1981.

[Tei77] D. Teichroew and E. A. Hershey, III,
"PSL/PSA: A Computer-Aided Technique
for Structured Documentation and Analysis
of Information Processing Systems",
IEEE Transactions on Software Engineering,
vol SE-3, no 1, pp 41-48, January 1977.

[Zav82] P. Zave,
"An Operational Approach to Requirements
Specification for Embedded Systems",
IEEE Transactions on Software Engineering,
vol SE-8, no 3, pp 250-269, May 1982.

[Zel78] M. V. Zelkowitz,
"Perspectives on Software Engineering",
Computing Surveys,
vol 10, no 2, pp 197-216, June 1978.

APPENDIX A - BNF GRAMMAR FOR ERA SPECIFICATIONS

This is an attempt to formalize the syntax of the era specifications.

General Description:

The era specification will consist of a set of frames. The order of the frame is not fixed. Each frame will contain information about one entity. Each frame will start on a newline. The first line in the frame will contain the keyword that describes the type of the entity and the name of the entity. The first letter in the type is capitalized. The type and the name are separated by a colon. At least one blank line will separate each frame.

The information in a frame is generally in the form of relations between this entity and other entities. Some of the information is in the form of attributes. An attribute gives information about this entity without referring to other entities. The order of these relations/attributes is not fixed.

Each relation/attribute is specified by a keyword that specifies the relation/attribute and its value. The value is either the name of the entity that has that relation or a text description of the attribute value. A colon separates the keyword and its value. Each relation/attribute starts on a new line. If a relation/attribute continues on to another line, the continuation line starts with a blank field followed by a colon. Multiple occurrences of a relation/attribute is represented by multiple occurrences of the keyword.

Entity Types: All entity types will start with a capital letter.

Type	Input
Output	Input_output
Activity	Periodic_function
Data	Constant
Comment	

* additional entity types may be added at any time

Relations/Attributes All relation/attribute types will start with a lower case letter.

input	output
required_mode	necessary_condition
occurrence	assertion
action	structure
keywords	media
type	units
subpart_is	subpart_of
uses	comment

* additional relation/attribute types may be added at any time

These lists of types are not fixed. Additional types may be defined in the future.

Syntax Description:

```
<era_spec> ::=
        <era_title> <era_body> <mode_table>

<era_title> ::=
      . PROCESS : <text>

<era_body> ::=
        <frame> | <frame> <era_body>

<frame> ::=
        <NL> <NL> <frame_header> <frame_body>
      | <NL> <NL> Comment : <text_lines>

<frame_header> ::=
        <i_o_data_header> : <i_o_data_name>
      | <function_header> : <CAPITAL_WORD>

<i_o_data_header> ::=
        Type | Input | Output | Input_output | Data
      | Constant | <CAPITAL_WORD>

<function_header> ::=
        Activity | Periodic_function | <CAPITAL_WORD>

<frame_body> ::=
        <relation> | <relation> <frame_body>

<relation> ::=
        <NL_B> <relation_type> : <relation_value>

<relation_type> ::=
        keywords | input | output | required_mode
      | necessary_condition | occurrence | assertion
      | action | comment | media | structure | type
      | units | subpart_is | subpart_of | uses | <WORD>

<relation_value> ::=
        <text_lines> | <structure>

<structure> ::=
        <struct> | <struct> <NL_B> : <structure>

<struct> ::=
        <name> | <text> | <name> <structure> | <text> <structure>

<name> ::=
        <mode_name> | <i_o_data_name>

<i_o_data_name> ::=
        $ <WORD> $
```

```
<mode_name> ::=
        * <WORD> *

<mode_table> ::=
        <NL> <NL> MODE_TABLE <mode_list> <initial_mode> <transition_body>

<mode_list> ::=
        <mode> | <mode> <mode_list>

<mode> ::=
        <NL_B> Mode : <mode_name>

<initial_mode> ::=
        <NL> <NL_B> Initial_Mode : <mode_name>

<transition_body> ::=
        <NL> <NL_B> Allowed_Mode_Transitions : <transition_list>

<transition_list> ::=
        <transition> | <transition> <transition_list>

<transition> ::=
        <NL_B> <event> : <mode_name> -> <mode_name>

<event> ::=
        <i_o_data_name>
      | <i_o_data_name> = ' <text> '
      | <function_header>


<text_lines> ::=
        <text> | <text> <text_cont>

<text> ::=
        <WORD> | <WORD> <text>

<text_cont> ::=
        <NL_B> : <text> | <NL> : <text> <text_cont>

<NL> ::=
        '\n' | '\n' <NL>

<NL_B> ::=
        <NL> '   '
```

LEXICAL SCANNER INFORMATION:

Tokens used in the productions above may begin with <char> or one of the following characters: *$,':-={}
Blanks can delimit tokens as well.

The following tokens are important above:

```
        <WORD>          ::= <char> | <char> <WORD>
        <CAPITAL_WORD>  ::= <capital_letter> <WORD>

<char> ::=
        <lower_case_char> | <symbol>

<lower_case_char> ::=
        a | b | ...| z | 0 | 1 | ... | 9

<symbol> ::=
        # | % | & | ( | ) | ? | _

<capital_letter> ::=
        A | B | ... | Z
```

There exists a set of "reserved word" tokens which includes:
{keyboard,crt,internal,secondary_storage,NONE,every_mode}

APPENDIX B - BNF GRAMMAR FOR THIS IMPLEMENTATION

This BNF for Entity-Relationship-Attribute Requirements Specifications is similar to the BNF in the preceding appendix. This BNF is specific to the language used as input to this implementation. Differences from the preceding appendix are as follows:

- to promote readability in the report:
  <i_o_data_header> has become <i_o_data>
  <function_header> has become <function>

- these <i_o_data>s have been eliminated:
  Type, Data, Constant

- these <relation_type>s have been eliminated:
  keywords, occurrence, assertion, action, structure, type, units, subpart_is, subpart_of, uses

- All lines must have a "left-hand side" (lhs) and a "right-hand side" (rhs). Thus, the <text_lines> and <text_cont> productions have been eliminated. This implies that all the lines in a "Comment Frame" must have the "Comment" keyword as a left-hand side. The loss of these two productions is not significant to this implementation.

- <relation_value> changes from being <text> with some potential structure to being a single <WORD>.

- Entities must begin in column 1 with a capital letter.

- Relations must begin after column 1 with a lower case character.

- Both may have any kind of <char> following the first letter.

- <WORD> may contain any type of <char>.

- By convention, <mode>s are all capitals and <i_o_data_name>s are a capital followed by a lower case character or '_'

- NOTE that because of <function>, <i_o_data> and <relation_type>, this grammar is ambiguous.

Syntax Description:

```
<era_spec> ::=
        <era_title> <era_body> <mode_table>

<era_title> ::=
        PROCESS : <text>

<era_body> ::=
        <frame> | <frame> <era_body>

<frame> ::=
        <NL> <NL> <frame_header> <frame_body>
      | <NL> <NL> Comment : <text>

<frame_header> ::=
        <i_o_data> : <i_o_data_name>
      | <function> : <CAPITAL_WORD>

<i_o_data> ::=
        Input | Output | Input_output | <CAPITAL_WORD>

<function> ::=
        Activity | Periodic_function | <CAPITAL_WORD>

<frame_body> ::=
        <relation> | <relation> <frame_body>

<relation> ::=
        <NL_B> <relation_type> : <relation_value>

<relation_type> ::=
        input | output | required_mode | necessary_condition
      | comment | media | <LOWER_CASE_WORD>

<relation_value> ::=
        <name> | <WORD>

<name> ::=
        <mode_name> | <i_o_data_name>

<i_o_data_name> ::=
        $ <WORD> $
```

```
<mode_name> ::=
        * <WORD> *

<mode_table> ::=
        <NL> <NL> MODE_TABLE <mode_list> <initial_mode> <transition_body>

<mode_list> ::=
        <mode> | <mode> <mode_list>

<mode> ::=
        <NL_B> Mode : <mode_name>

<initial_mode> ::=
        <NL> <NL_B> Initial_Mode : <mode_name>

<transition_body> ::=
        <NL> <NL_B> Allowed_Mode_Transitions : <transition_list>

<transition_list> ::=
        <transition> | <transition> <transition_list>

<transition> ::=
        <NL_B> <event> : <mode_name> -> <mode_name>

<event> ::=
        <i_o_data_name>
      | <i_o_data_name> = ' <text> '
      | <function>


<text> ::=
        <WORD> | <WORD> <text>

<NL> ::=
        '\n' | '\n' <NL>

<NL_B> ::=
        <NL> '   '
```

LEXICAL SCANNER INFORMATION:

Tokens used in the productions above may begin with <char> or one of the following characters: *$,':-={}
Blanks can delimit tokens as well.

The following tokens are important above:

```
        <WORD>                  ::= <char>
                                  | <char> <WORD>
        <CAPITAL_WORD>          ::= <capital_letter> <WORD>
                                  | <capital_letter>
        <LOWER_CASE_WORD>       ::= <lower_case_char> <WORD>
                                  | <lower_case_char>

<char> ::=
        <capital_letter>
      | <lower_case_char>
      | <symbol>

<lower_case_char> ::=
        a | b | ...| z | 0 | 1 | ... | 9

<symbol> ::=
        # | % | & | ( | ) | ? | _

<capital_letter> ::=
        A | B | ... | Z
```

There exists a set of "reserved word" tokens which includes:
{keyboard,crt,internal,secondary_storage,NONE,every_mode}

APPENDIX C - ERA SPECIFICATION FOR THE CHESS PROCESS

```
PROCESS : Requirements specification for the chess program

Comment : as of 6/25/84 1:40 in ksu832:/usrb/we/era

Comment : comment on specification:
Comment : Not all of the activities necessary for this program to
Comment : be implemented are included in this description.  Some
Comment : activities are not included if their activities were
Comment : determined by the other activities.  The activity of
Comment : interpreting the user's command was not included.

Type : $piece$
   structure : a string from the set {Kr,Kk,Kb,K,Q,Qb,Qk,Qr,p}

Type : $rank$
   structure : a string from the set {1,2,...8}

Type : $position$
   structure : $piece$ $rank$

Type : $piece_position$
   structure : $piece$ ',' $position$

Type : $board_matrix$
   structure : array[1..8,1..8] of $piece$ OR ' '

Input : $board_description$
   media     : keyboard
   structure : 'white' set of $piece_position$
   structure : 'black' set of $piece_position$
   structure : 'end'

Input : $name_of_game$
   media     : keyboard
   structure : 1 to 20 alphanumeric characters

Input : $new_user_input$
   media     : keyboard
   structure : any string

Input_output : $stored_board$
   media     : secondary_storage
   structure : information to recreate the board configuration

Input_output : $chess_board$
   media     : internal
   structure : $board_matrix$
```

```
Comment : This page contains those Input entities which are
Comment : directly related to a command which the user of
Comment : the chess game might enter.  (As opposed to Input
Comment : data which is not a command, ie: $name_of_game$)

Input : $move$
   media     : keyboard
   structure : 'm' $position$ '-' $position$

Input : $display_board$
   media     : keyboard
   structure : 'display'

Input : $create$
   media     : keyboard
   structure : 'create'

Input : $concede$
   media     : keyboard
   structure : 'concede'

Input : $store$
   media     : keyboard
   structure : 'store' $name_of_game$

Input : $retrieve$
   media     : keyboard
   structure : 'retrieve' $name_of_game$

Comment : The remaining Input entities are 'pseudo commands'
Comment : intended to aid in manually exercising
Comment : Periodic_functions.  The entities were named by
Comment : switching the first and last words so as not to
Comment : cause name collisions with the Output entities

Input : $mate_stale$
   media     : keyboard
   structure : 'stalemate'

Input : $limit_time$
   media     : keyboard
   structure : 'time_limit'

Input : $out_time$
   media     : keyboard
   structure : 'time_out'

Input : $check_input$
   media     : keyboard
   structure : 'input_check'

Comment : 1 Input entity above is unused.
Comment : 1 Input entity is omitted.
```

```
Output : $status$
   media     : crt
   structure : string from the set {'your move','check',
   structure : 'checkmate','concede'}

Output : $board_display$
   media     : crt
   structure : visually oriented display of current chess board

Output : $syntax_error$
   media     : crt
   structure : <cr> 'illegal, try again'

Output : $store_message$
   media     : crt
   structure : 'board stored'
   structure : 'storage failed'

Output : $retrieve_message$
   media     : crt
   structure : $name_of_game$ 'retrieved'
   structure : 'retrieval failed'

Output : $stalemate$
   media     : crt
   structure : 'stalemate occurred'

Output : $time_warning$
   media     : crt
   structure : 'this is a warning - 5 minutes elapsed'

Output : $time_out$
   media     : crt
   structure : 'too much time - game over'

Output : $move_message$
   media     : crt
   structure : <cr>
   structure : 'illegal_move'

Output : $computer_move_message$
   media     : crt
   structure : 'computer moves from' $position$ 'to' $position$
```

```
Activity : Initialize_board
   keywords             : Standard_board,Initialize,Place_pieces
   input                : NONE
   output               : $chess_board$
   required_mode        : *START*
   necessary_condition  : $start$
   assertion            : The output board is a correct
   assertion            : representation of the standard
   assertion            : starting configuration for chess

Activity : Create_special_board
   keywords             : Assign_positions,Place_pieces
   input                : $board_description$
   output               : $chess_board$
   required_mode        : *START*
   necessary_condition  : $create$

Activity : Store_board
   keywords             : Store_game_status,Save_board
   input                : $name_of_game$
   input                : $chess_board$
   output               : $store_message$
   required_mode        : *NORMAL*
   necessary_condition  : $store$
   assertion            : game is stored in file '$name_of_game$'

Activity : Retrieve_board
   keywords             : Retrieve_board
   input                : $name_of_game$
   output               : $chess_board$
   output               : $retrieve_message$
   required_mode        : *START*
   necessary_condition  : $retrieve$
   assertion            : Retrieves game stored in file
   assertion            : '$name_of_game$' if successful

Comment : 1 Input item above is related to more than 1 other entity
```

```
Activity : Validate_user_move
   keywords              : Check_move,Check_m_status,Move_validation
   input                 : $chess_board$
   input                 : $move$
   output                : $move_message$
   required_mode         : *NORMAL*
   assertion             : If the move is illegal,
   assertion             : the mode changes to *ILLEGAL*

Activity : Computer_Move
   comment               : used to be Move
   keywords              : Select_move,Select_status
   input                 : $chess_board$
   output                : $chess_board$
   output                : $computer_move_message$
   output                : $status$
   required_mode         : *NORMAL*
   action                : mode may change to *END*
   action                : if $status$ = 'checkmate' OR 'concede'

Activity : Update_board
   keywords              : Update_position,Update_status
   input                 : $chess_board$
   input                 : $move$
   output                : $chess_board$
   required_mode         : *NORMAL*

Activity : Display_board
   keywords              : Display
   input                 : $chess_board$
   output                : $board_display$
   required_mode         : *NORMAL*
   required_mode         : *END*
   necessary_condition   : $display_board$

Comment : 1 Input item above is related to more than 1 other entity
```

```
Comment : for simplicity, pseudo Input entities exist to manually
Comment : exercise these Periodic_functions.

Periodic_function : Stalemate
   required_mode : *NORMAL*
   occurrence    : whenever a configuration is repeated 3 times
   input         : $mate_stale$
   output        : $stalemate$
   action        : change mode to *END*

Periodic_function : Time_Limit
   required_mode : *NORMAL*
   occurrence    : whenever the user response time > 5 minutes
   input         : $limit_time$
   output        : $time_warning$
   action        : NONE

Periodic_function : Time_Out
   required_mode : *NORMAL*
   occurrence    : whenever the user response time > 10 minutes
   input         : $out_time$
   output        : $time_out$
   action        : change mode to *END*

Periodic_function : Input_Check
   required_mode : every_mode
   occurrence    : whenever user input does not match syntax
   input         : $check_input$
   output        : $syntax_error$
   action        : change mode to *ILLEGAL*
```

```
MODE_TABLE
   Mode : *ILLEGAL*
   Mode : *NORMAL*
   Mode : *START*
   Mode : *END*

   Initial_Mode : *START*

   Allowed_Mode_Transitions :
   $create$                                  : *START* -> *NORMAL*
   $start$                                   : *START* -> *NORMAL*
   $retrieve$                                : *START* -> *NORMAL*

   $status$ = {'checkmate','concede'}        : *NORMAL* -> *END*
   $stalemate$                               : *NORMAL* -> *END*
   $time_out$                                : *NORMAL* -> *END*

   $move_message$ = {'illegal_move'}         : *NORMAL* -> *ILLEGAL*
   $syntax_error$                            : *NORMAL* -> *ILLEGAL*
   $syntax_error$                            : *START* -> *ILLEGAL*
   $syntax_error$                            : *END* -> *ILLEGAL*

   $new_user_input$                          : *ILLEGAL* -> *NORMAL*

   $'<cr>'$                                  : *END* -> *START*

Comment : 2 of the above transitions are unfirable.
Comment : 3 of the above transitions cause mode indeterminacy.
Comment : as specified here, *END* is not a terminal mode.
```

APPENDIX D - PETRI NET GRAPH OF THE CHESS PROCESS

The following transitions correspond to the graph on the next page:

1. T_INIT

2. Initialize_board
   INVOKES ($chess_board$ = Initialize_board ( ))

3. Create_special_board
   INVOKES ($chess_board$ = Create_special_board ( ))

4. Store_board
   INVOKES (Store_board ($chess_board$))

5. Retrieve_board
   INVOKES ($chess_board$ = Retrieve_board ( ))

6. Validate_user_move
   INVOKES (Validate_user_move ($chess_board$))
   if $move_message$ in {'illegal_move'} then *ILLEGAL*

7. Computer_Move
   INVOKES ($chess_board$ = Computer_Move ($chess_board$))
   if $status$ in {'checkmate','concede'} then *END*

8. Update_board
   INVOKES ($chess_board$ = Update_board ($chess_board$))

9. Display_board_*NORMAL*
   INVOKES (Display_board ($chess_board$))

10. Stalemate
    WHEN (a configuration is repeated 3 times)

11. Time_Limit
    WHEN (the user response time > 5 minutes)

12. Time_Out
    WHEN (the user response time > 10 minutes)

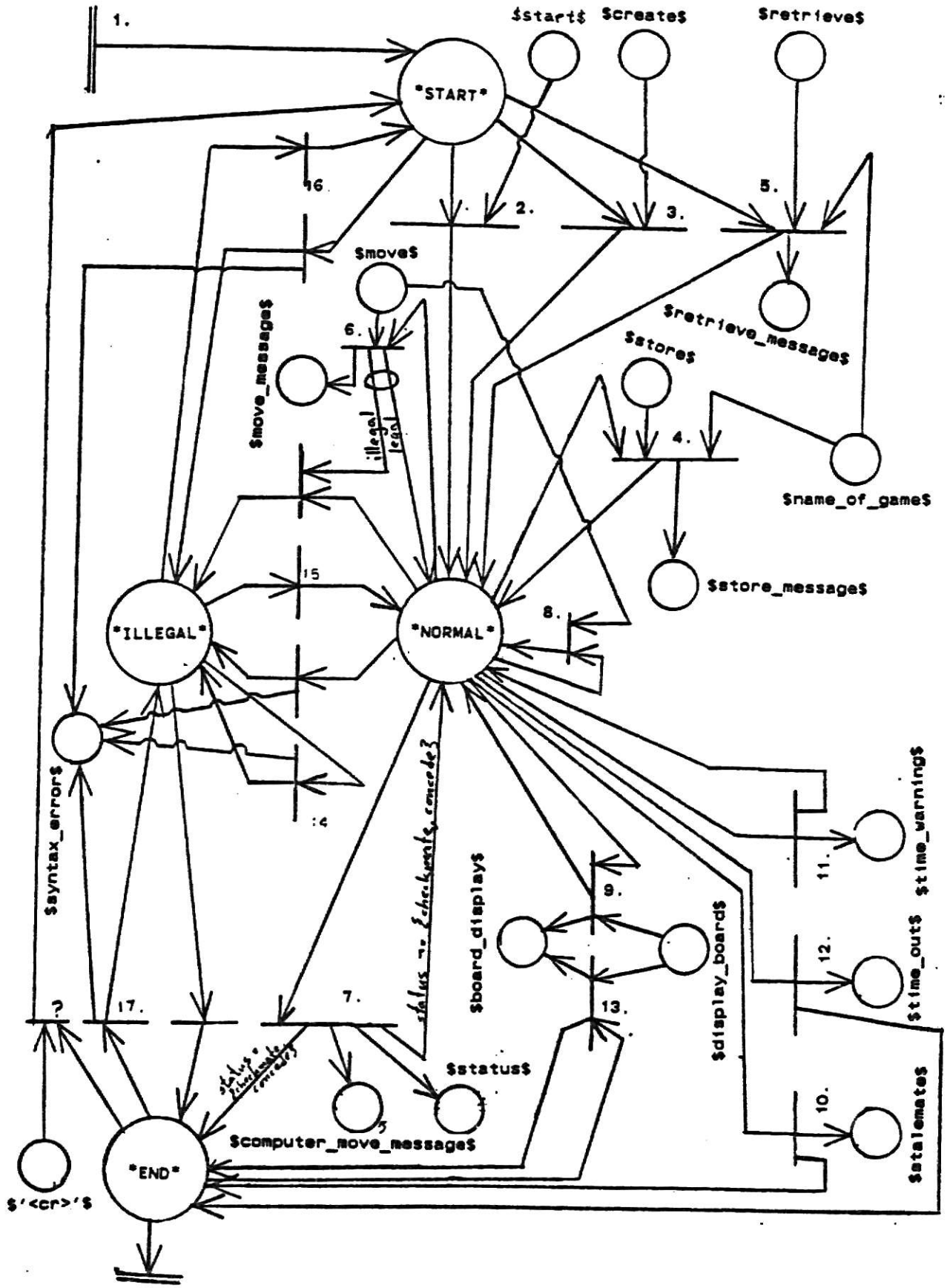
13. Display_board_*END*
    INVOKES (Display_board ($chess_board$))

14. Input_Check_*ILLEGAL*
    WHEN (user input does not match syntax)

15. Input_Check_*NORMAL*
    WHEN (user input does not match syntax)

16. Input_Check_*START*
    WHEN (user input does not match syntax)

17. Input_Check_*END*
    WHEN (user input does not match syntax)


1.
$start$
$create$
$retrieve$
"START"
16.
2.
3.
5.
$move$
$move_messages$
6.
illegal
legal
$retrieve_messages$
$store$
4.
$name_of_game$
$store_message$
15
"ILLEGAL"
"NORMAL"
8.
14
$syntax_errors$
$board_display$
9.
11.
$time_warnings$
12.
$time_out$
$display_boards$
13.
7.
17.
$status$
10.
$stalemates$
$computer_move_message$
"END"
$'<cr>'$

APPENDIX E - PNL CODE FOR THE CHESS PROCESS

```
# ------------------> INITial Transition:
T_INIT
    : INIT  OUTPUT TO ( *START* )


# ------------------> PLACEs:
*START*
    : PLACE OUTPUT TO ( Initialize_board, Create_special_board,
                        Retrieve_board, Input_Check_*START* )

# Entity undefined by user: default values assumed:
$start$
    : PLACE OUTPUT TO ( Initialize_board )

*NORMAL*
    : PLACE OUTPUT TO ( Store_board, Validate_user_move,
                        Computer_Move, Update_board,
                        Display_board_*NORMAL*, Stalemate,
                        Time_Limit, Time_Out,
                        Input_Check_*NORMAL* )

$create$
    : PLACE OUTPUT TO ( Create_special_board )

$board_description$
    : PLACE OUTPUT TO ( Create_special_board )

$store$
    : PLACE OUTPUT TO ( Store_board )

$name_of_game$
    : PLACE OUTPUT TO ( Store_board, Retrieve_board )

$store_message$
    : PLACE OUTPUT TO ( T_TERM_$store_message$ )

$retrieve$
    : PLACE OUTPUT TO ( Retrieve_board )

$retrieve_message$
    : PLACE OUTPUT TO ( T_TERM_$retrieve_message$ )

$move_message$
    : PLACE OUTPUT TO ( T_TERM_$move_message$ )

$move$
    : PLACE OUTPUT TO ( Validate_user_move, Update_board )

# No entity directly references this mode:
*ILLEGAL*
    : PLACE OUTPUT TO ( Input_Check_*ILLEGAL* )

$computer_move_message$
    : PLACE OUTPUT TO ( T_TERM_$computer_move_message$ )
```

```
$status$
    : PLACE OUTPUT TO ( T_TERM_$status$ )

*END*
    : PLACE OUTPUT TO ( Display_board_*END*, Input_Check_*END* )

$display_board$
    : PLACE OUTPUT TO ( Display_board_*NORMAL*,
                        Display_board_*END* )

$board_display$
    : PLACE OUTPUT TO ( T_TERM_$board_display$ )

$mate_stale$
    : PLACE OUTPUT TO ( Stalemate )

$stalemate$
    : PLACE OUTPUT TO ( T_TERM_$stalemate$ )

$limit_time$
    : PLACE OUTPUT TO ( Time_Limit )

$time_warning$
    : PLACE OUTPUT TO ( T_TERM_$time_warning$ )

$out_time$
    : PLACE OUTPUT TO ( Time_Out )

$time_out$
    : PLACE OUTPUT TO ( T_TERM_$time_out$ )

$check_input$
    : PLACE OUTPUT TO ( Input_Check_*ILLEGAL*,
                        Input_Check_*NORMAL*,
                        Input_Check_*START*,
                        Input_Check_*END* )

$syntax_error$
    : PLACE OUTPUT TO ( T_TERM_$syntax_error$ )


# ------------------> function TRANSitions:
- WARNING: value of 'necessary_condition' is undefined
Initialize_board
    : TRANS OUTPUT TO ( *NORMAL* )
            INPUT FROM ( $start$, *START* )
            INVOKES ( $chess_board$ = Initialize_board ( ) )

Create_special_board
    : TRANS OUTPUT TO ( *NORMAL* )
            INPUT FROM ( $create$, $board_description$, *START* )
            INVOKES ( $chess_board$ = Create_special_board ( ) )

Store_board
    : TRANS OUTPUT TO ( $store_message$, *NORMAL* )
            INPUT FROM ( $store$, $name_of_game$, *NORMAL* )
```

```
            INVOKES ( Store_board ( $chess_board$ ) )

Retrieve_board
    : TRANS OUTPUT TO ( $retrieve_message$, *NORMAL* )
            INPUT FROM ( $retrieve$, $name_of_game$, *START* )
            INVOKES ( $chess_board$ = Retrieve_board ( ) )

Validate_user_move
    : TRANS OUTPUT TO ( $move_message$, *NORMAL* )
            INPUT FROM ( $move$, *NORMAL* )
            INVOKES ( Validate_user_move ( $chess_board$ ) )
            if $move_message$ in {'illegal_move'} then *ILLEGAL*

Computer_Move
    : TRANS OUTPUT TO ( $computer_move_message$, $status$, *NORMAL* )
            INPUT FROM ( *NORMAL* )
            INVOKES
                ( $chess_board$ = Computer_Move ( $chess_board$ ) )
            if $status$ in {'checkmate','concede'} then *END*

Update_board
    : TRANS OUTPUT TO ( *NORMAL* )
            INPUT FROM ( $move$, *NORMAL* )
            INVOKES
                ( $chess_board$ = Update_board ( $chess_board$ ) )

Display_board_*NORMAL*
    : TRANS OUTPUT TO ( $board_display$, *NORMAL* )
            INPUT FROM ( $display_board$, *NORMAL* )
            INVOKES ( Display_board ( $chess_board$ ) )

- WARNING: don't use 'input' relations in Periodic_functions
Stalemate
    : TRANS OUTPUT TO ( $stalemate$, *END* )
            INPUT FROM ( $mate_stale$, *NORMAL* )
            WHEN ( a configuration is repeated 3 times )

- WARNING: don't use 'input' relations in Periodic_functions
Time_Limit
    : TRANS OUTPUT TO ( $time_warning$, *NORMAL* )
            INPUT FROM ( $limit_time$, *NORMAL* )
            WHEN ( the user response time > 5 minutes )

- WARNING: don't use 'input' relations in Periodic_functions
Time_Out
    : TRANS OUTPUT TO ( $time_out$, *END* )
            INPUT FROM ( $out_time$, *NORMAL* )
            WHEN ( the user response time > 10 minutes )

Display_board_*END*
    : TRANS OUTPUT TO ( $board_display$, *END* )
            INPUT FROM ( $display_board$, *END* )
            INVOKES ( Display_board ( $chess_board$ ) )

- WARNING: don't use 'input' relations in Periodic_functions
Input_Check_*ILLEGAL*
```

```
    : TRANS OUTPUT TO ( $syntax_error$, *ILLEGAL* )
            INPUT FROM ( $check_input$, *ILLEGAL* )
            WHEN ( user input does not match syntax )

- WARNING: don't use 'input' relations in Periodic_functions
Input_Check_*NORMAL*
    : TRANS OUTPUT TO ( $syntax_error$, *ILLEGAL* )
            INPUT FROM ( $check_input$, *NORMAL* )
            WHEN ( user input does not match syntax )

- WARNING: don't use 'input' relations in Periodic_functions
Input_Check_*START*
    : TRANS OUTPUT TO ( $syntax_error$, *ILLEGAL* )
            INPUT FROM ( $check_input$, *START* )
            WHEN ( user input does not match syntax )

- WARNING: don't use 'input' relations in Periodic_functions
Input_Check_*END*
    : TRANS OUTPUT TO ( $syntax_error$, *ILLEGAL* )
            INPUT FROM ( $check_input$, *END* )
            WHEN ( user input does not match syntax )


# ------------------> TERMinal Transitions:
T_TERM_$store_message$
    : TERM  INPUT FROM ( $store_message$ )

T_TERM_$retrieve_message$
    : TERM  INPUT FROM ( $retrieve_message$ )

T_TERM_$move_message$
    : TERM  INPUT FROM ( $move_message$ )

T_TERM_$computer_move_message$
    : TERM  INPUT FROM ( $computer_move_message$ )

T_TERM_$status$
    : TERM  INPUT FROM ( $status$ )

T_TERM_$board_display$
    : TERM  INPUT FROM ( $board_display$ )

T_TERM_$stalemate$
    : TERM  INPUT FROM ( $stalemate$ )

T_TERM_$time_warning$
    : TERM  INPUT FROM ( $time_warning$ )

T_TERM_$time_out$
    : TERM  INPUT FROM ( $time_out$ )

T_TERM_$syntax_error$
    : TERM  INPUT FROM ( $syntax_error$ )
```

APPENDIX F - DIAGNOSTIC AND ERROR MESSAGES

Fatal Errors and Diagnostic Warning messages in the SCAN MODULE

All of the following errors occur during the scanning of the ERA Requirements Specification file, and the storing of its contents in the Symbol table "SYMTAB" and Function table "FUNTAB"

```
-------------------------> fatal ERRORs in scan.c
ERROR ( "FUNCTION names begin with a capitol letter" );
        for example: Recreate_board

ERROR ( "I_O_DATA names are delimited by '$' signs");
        for example: $status$

ERROR ( "MODE names are delimited by '*' signs");
        for example: *START*

ERROR ( "failed to match '<lhs> : <rhs>'" );
        ALL lines of significance in an ERA
        specification must be of the form:
                "left_hand_side" : "right_hand_side"

ERROR ( "MODE_TABLE not encountered" );
        a MODE_TABLE section must occur between the
        list of entities and the end of the ERA
        specification file.

ERROR ( "No PROCESS line in specification" );
        a PROCESS line must be the first line in
        an ERA specification.

ERROR ( "illegal: <NL> <relation_line>" );
        it is an error to preced a relation line
        with a newline.

ERROR ( "Entity must follow blank line" );
        it is an error to not have a blank line
        preceding an entity.

ERROR ( "only used in Activity entities" );
        refers to the use of a "necessary_condition" keyword
        in Periodic_functions.

ERROR ( "only used in Periodic_function entities" );
        refers to the use of the "occurrence"
        keyword in Activities.

ERROR ( "illegal: <entity_line> <NL>" );
        it is an error for a newline to follow an
        entity line.  relations are expected to
        follow entity lines. .

ERROR ( "internal error" );
        usually caused by an invalid case in
        a switch statement
```

```
--------------------------> Diagnostics issued from scan.c
st_warn ( "don't use 'input' relations in Periodic_functions" );
        PNL WARNING: Periodic_functions are not
        supposed to take input from the keyboard.
```

Fatal Errors and Diagnostic Warning messages in the TRANSLATOR module

```
--------------------------> fatal ERRORs in translate.c
ERROR ( "can't open PNL_FILE for writing" );
        the PNL_FILE is where the translator
        puts PNL instructions.

ERROR ( "places table overflow" );
ERROR ( "trans table overflow" );
        indicates a part of the entity graph has been exhausted.

ERROR ( "max # of input indices = 25" );
ERROR ( "max # of output indices = 25" );
        a node in the entity graph exceeds the maximum number of
        input or output arcs.
        Recompile, after increasing INPUT_MAX or OUTPUT_MAX.
ERROR ( "symbol table overflow" );
        SYMTAB has become exhausted.

ERROR ( "max # of symbol suffixes = 1" );
ERROR ( "max # of sym_class values = 1" );
ERROR ( "max # of sym_idx's = 1" );
ERROR ( "Max # of comment strings = 1" );
        These messages list the maximum number of relations
        of a given type which are allowed in a function.

ERROR ( "Attempt to add duplicate to Mode_list" );
        A mode is listed more than once in the list of
        Modes, in the first part of the MODE_TABLE

ERROR ( "Initial_Mode isn't referenced by any entity" );
        Because of this, the net can not be activated.

ERROR ( "Initial_Mode isn't in the Mode_list" );
        The Mode_list must contain ALL of the Mode names.

--------------------------> Diagnostics issued from T_symtab.c
st_comment ( "No entity directly references this mode" );
        if a Mode occurs in the Mode_list, (in MODE_TABLE), but is
        not yet in SYMTAB, then it has not yet been referenced by
        any entity.

--------------------------> fatal ERRORs in T_symfun.c
ERROR ( "media: only valid for I_O_DATAs" );
        a media relation has been found in an entity other than
        an Input, Output or Input_output entity

ERROR ( "media not found in media table" );
        the value of the media keyword is not "crt", "keyboard", etc.
```

```
ERROR ( "max # of media values = 1" );
ERROR ( "max # of annotations = 1" );
        These messages list the maximum number of relations of a given
        type which are allowed in a function.

-------------------------> Diagnostics issued from T_symfun.c
st_comment ( "Entity undefined by user: default values assumed" );
st_warn ( "value of '%s' is undefined" );
        An "orphaned" entity is referenced, but never defined.  It is
        given default characteristics, so that processing may continue.
        A comment is placed in this default definition, and a warning
        is issued to the function which
        references this undefined orphan.

-------------------------> fatal ERRORs in T_funtab.c
ERROR ( "function table overflow" );
        FUNTAB has become exhausted

ERROR ( "max # of p_or_a values = 1" );
ERROR ( "Max # of 'necessary_condition' keywords = 1" );
ERROR ( "Max # of 'input' keywords = 3" );
ERROR ( "Max # of 'output' keywords = 3" );
ERROR ( "Max # of 'req_mode' keywords = 3" );
ERROR ( "max # of 'next_mode' keywords = 1" );
ERROR ( "Max # of annotations = 1" );
ERROR ( "Max # of warning strings = 1" );
        These messages list the maximum number of relations
        of a given type which are allowed in a function.

ERROR ( "necessary_condition: only valid for Activity" );
ERROR ( "input: only valid for FUNCTIONs" );
ERROR ( "output: only valid for FUNCTIONs" );
ERROR ( "required_mode: only valid for FUNCTIONs" );
ERROR ( "occurrence: only valid in Periodic_function" );
ERROR ( "warn: only valid for FUNCTIONs" );
        The indicated keyword can only occur in
        certain types of entity.

ERROR ( "occurrence : whenever 'expr'" );
        The occurrence relation has the given format.

-------------------------> fatal ERRORs in T_print.c
ERROR ( "First entry in trans[] must be P_INIT" );
        The first entry in the transition table (which is part of
        the entity graph) must be the value of Initial_Mode.

debug ( rname, D_ERROR, "can't open TABLES_FILE (see files.h)" );
debug ( rname, D_ERROR, "closing Tab_fp" );
        These error messages are related to opening and closing
        the file which contains an internal dump of tables.
```

APPENDIX G - RUNNING THE IMPLEMENTATION

```
$ # The following is a sample run of the implementation:
$ make
        rm -f P.t*
        P
Enter '?' for a Help message
> ?
Enter one of the following:
        q     - quit this session
        <cr>  - fire a transition
        o     - change Interpreter options
        d     - dump the Petri net
        ?     - print this help message
> o
Enter an option, or '?' for help: t
Tracing turned ON
Enter an option, or '?' for help: q
Returning to Interpreter
> 
0: Initialize_board : INVOKES ( $chess_board$ = Initialize_board ( ) )
1: Create_special_board : INVOKES ( $chess_board$ = Create_special_board ( ) )
2: Retrieve_board : INVOKES ( $chess_board$ = Retrieve_board ( ) )
3: Input_Check_*START* : WHEN ( user input does not match syntax )

Enter the number of the transition to be fired: 0
FIRING menu item # 0:
Initialize_board : INVOKES ( $chess_board$ = Initialize_board ( ) )

FIRING TERMinal transitions:
(none)
> d
Dumping the Petri Net
bag PLACE
 1   *NORMAL*
enabled Function TRANSitions:
Computer_Move : if $status$ in ('checkmate','concede') then *END*
        INVOKES ( $chess_board$ = Computer_Move ( $chess_board$ ) )

enabled TERMinal transitions:
(none)
almost enabled Function TRANSitions:
Store_board : INVOKES ( Store_board ( $chess_board$ ) )
Validate_user_move : if $move_message$ in ('illegal_move') then *ILLEGAL*
        INVOKES ( Validate_user_move ( $chess_board$ ) )
Computer_Move : if $status$ in ('checkmate','concede') then *END*
        INVOKES ( $chess_board$ = Computer_Move ( $chess_board$ ) )
Update_board : INVOKES ( $chess_board$ = Update_board ( $chess_board$ ) )
Display_board_*NORMAL* : INVOKES ( Display_board ( $chess_board$ ) )
Stalemate : WHEN ( a configuration is repeated 3 times )
Time_Limit : WHEN ( the user response time > 5 minutes )
Time_Out : WHEN ( the user response time > 10 minutes )
Input_Check_*NORMAL* : WHEN ( user input does not match syntax )

> 
0: Computer_Move : if $status$ in ('checkmate','concede') then *END*
        INVOKES ( $chess_board$ = Computer_Move ( $chess_board$ ) )
```

```
Enter the number of the transition to be fired: 0
FIRING menu item # 0:
Computer_Move : if $status$ in ('checkmate','concede') then *END*
        INVOKES ( $chess_board$ = Computer_Move ( $chess_board$ ) )

FIRING TERMinal transitions:
T_TERM_$computer_move_message$ : INPUT FROM ( $computer_move_message$ )
T_TERM_$status$ : INPUT FROM ( $status$ )
>
0: Computer_Move : if $status$ in ('checkmate','concede') then *END*
        INVOKES ( $chess_board$ = Computer_Move ( $chess_board$ ) )

Enter the number of the transition to be fired: 0
FIRING menu item # 0:
Computer_Move : if $status$ in ('checkmate','concede') then *END*
        INVOKES ( $chess_board$ = Computer_Move ( $chess_board$ ) )

FIRING TERMinal transitions:
T_TERM_$computer_move_message$ : INPUT FROM ( $computer_move_message$ )
T_TERM_$status$ : INPUT FROM ( $status$ )
>
0: Computer_Move : if $status$ in ('checkmate','concede') then *END*
        INVOKES ( $chess_board$ = Computer_Move ( $chess_board$ ) )

Enter the number of the transition to be fired: 0
FIRING menu item # 0:
Computer_Move : if $status$ in ('checkmate','concede') then *END*
        INVOKES ( $chess_board$ = Computer_Move ( $chess_board$ ) )

FIRING TERMinal transitions:
T_TERM_$computer_move_message$ : INPUT FROM ( $computer_move_message$ )
T_TERM_$status$ : INPUT FROM ( $status$ )
>
0: Computer_Move : if $status$ in ('checkmate','concede') then *END*
        INVOKES ( $chess_board$ = Computer_Move ( $chess_board$ ) )

Enter the number of the transition to be fired: 0
FIRING menu item # 0:
************************
Potential Net Starvation!
Turn Starvation Defeat ON
************************
Computer_Move : if $status$ in ('checkmate','concede') then *END*
        INVOKES ( $chess_board$ = Computer_Move ( $chess_board$ ) )

FIRING TERMinal transitions:
T_TERM_$computer_move_message$ : INPUT FROM ( $computer_move_message$ )
T_TERM_$status$ : INPUT FROM ( $status$ )
> o
Enter an option, or '?' for help: "false input"
Enter '?' for a Help message
Enter an option, or '?' for help: s
Starvation defeat ON
Enter an option, or '?' for help: q
Returning to Interpreter
>
0: Computer_Move : if $status$ in ('checkmate','concede') then *END*
        INVOKES ( $chess_board$ = Computer_Move ( $chess_board$ ) )
1: Store_board : INVOKES ( Store_board ( $chess_board$ ) )
2: Validate_user_move : if $move_message$ in ('illegal_move') then *ILLEGAL*
        INVOKES ( Validate_user_move ( $chess_board$ ) )
3: Computer_Move : if $status$ in ('checkmate','concede') then *END*
        INVOKES ( $chess_board$ = Computer_Move ( $chess_board$ ) )
4: Update_board : INVOKES ( $chess_board$ = Update_board ( $chess_board$ ) )
5: Display_board_*NORMAL* : INVOKES ( Display_board ( $chess_board$ ) )
6: Stalemate : WHEN ( a configuration is repeated 3 times )
7: Time_Limit : WHEN ( the user response time > 5 minutes )
8: Time_Out : WHEN ( the user response time > 10 minutes )
9: Input_Check_*NORMAL* : WHEN ( user input does not match syntax )

Enter the number of the transition to be fired: 5
FIRING menu item # 5:
Display_board_*NORMAL* : INVOKES ( Display_board ( $chess_board$ ) )

FIRING TERMinal transitions:
T_TERM_$board_display$ : INPUT FROM ( $board_display$ )
> q
Goodbye.
$
$
```

```
   :
$ # The following is a sample run of the implementation:
$ make
        rm -f P.t*
        P
Enter '?' for a Help message
> ?
Enter one of the following:
        q     - quit this session
        <cr> - fire a transition
        o     - change Interpreter options
        d     - dump the Petri net
        ?     - print this help message
>
0: Initialize_board : INVOKES ( $chess_board$ = Initialize_board ( ) )
1: Create_special_board : INVOKES ( $chess_board$ = Create_special_board ( ) )
2: Retrieve_board : INVOKES ( $chess_board$ = Retrieve_board ( ) )
3: Input_Check_*START* : WHEN ( user input does not match syntax )

Enter the number of the transition to be fired: 2
FIRING menu item # 2:

FIRING TERMinal transitions:
T_TERM_$retrieve_message$ : INPUT FROM ( $retrieve_message$ )
>
FIRING TERMinal transitions:
T_TERM_$computer_move_message$ : INPUT FROM ( $computer_move_message$ )
T_TERM_$status$ : INPUT FROM ( $status$ )
>
FIRING TERMinal transitions:
T_TERM_$computer_move_message$ : INPUT FROM ( $computer_move_message$ )
T_TERM_$status$ : INPUT FROM ( $status$ )
>
FIRING TERMinal transitions:
T_TERM_$computer_move_message$ : INPUT FROM ( $computer_move_message$ )
T_TERM_$status$ : INPUT FROM ( $status$ )
>
**************************
Potential Net Starvation!
Turn Starvation Defeat ON
**************************
FIRING TERMinal transitions:
T_TERM_$computer_move_message$ : INPUT FROM ( $computer_move_message$ )
T_TERM_$status$ : INPUT FROM ( $status$ )
> o
Enter an option, or '?' for help: s
Starvation defeat ON
Enter an option, or '?' for help: q
Returning to Interpreter
>
0: Computer_Move : if $status$ in {'checkmate','concede'} then *END*
        INVOKES ( $chess_board$ = Computer_Move ( $chess_board$ ) )
1: Store_board : INVOKES ( Store_board ( $chess_board$ ) )
2: Validate_user_move : if $move_message$ in {'illegal_move'} then *ILLEGAL*
        INVOKES ( Validate_user_move ( $chess_board$ ) )
3: Computer_Move : if $status$ in {'checkmate','concede'} then *END*
        INVOKES ( $chess_board$ = Computer_Move ( $chess_board$ ) )
4: Update_board : INVOKES ( $chess_board$ = Update_board ( $chess_board$ ) )
5: Display_board_*NORMAL* : INVOKES ( Display_board ( $chess_board$ ) )
6: Stalemate : WHEN ( a configuration is repeated 3 times )
7: Time_Limit : WHEN ( the user response time > 5 minutes )
8: Time_Out : WHEN ( the user response time > 10 minutes )
9: Input_Check_*NORMAL* : WHEN ( user input does not match syntax )

Enter the number of the transition to be fired: 1
FIRING menu item # 1:

FIRING TERMinal transitions:
T_TERM_$store_message$ : INPUT FROM ( $store_message$ )
> q
Goodbye.
$
```

APPENDIX H - SOURCE CODE LISTINGS

The following files are the source for this implementation.
Files ending in ".c" are the C-language source code.
Files ending in ".h" contain constant definitions.
Files ending in ".i" contain table declarations.

The source files are listed in this order:

1.	Makefile	- controls compilation and execution	
2.	main.c	- MAIN module:	uses TRANSLATE INTERPRET
3.	scan.c	- SCANNER module:	uses INPUT TABLES
4.	translate.c	- TRANSLATOR module:	uses SCANNER TABLES
5.	interpret.c	- INTERPRETER module:	uses TABLES
6.	T_symtab.c	- TABLES module:	(symbol table)
7.	T_symfun.c	- TABLES module:	(symbol/function tables)
8.	T_funtab.c	- TABLES module:	(function table)
9.	T_print.c	- TABLES module:	(print the tables)
10.	input.c	- INPUT module:	
11.	debug.c	- DEBUG module:	
12.	project.h	- common definitions:	used by all files
13.	files.h	- input/output file names	
14.	T_tables.h	- common definitions:	used by TABLES module
15.	debug.h	- debugging trace levels	
16.	T_tables.i	- common declarations:	used by TABLES module
17.	T_pnltab.i	- PNL tables:	used by TABLES module
18.	T_symtab.i	- symbol table	used by TABLES module
19.	T_funtab.i	- function table	used by TABLES module

```
EXEC = P
HDRS = project.h files.h T_tables.h debug.h
INCS = T_tables.i T_pnltab.i T_symtab.i T_funtab.i
SRCS = main.c scan.c translate.c interpret.c \
       T_symtab.c T_symfun.c T_funtab.c T_print.c input.c debug.c
OBJS = main.o scan.o translate.o interpret.o \
       T_symtab.o T_symfun.o T_funtab.o T_print.o input.o debug.o

# ALL PROJECT SPECIFIC INFORMATION PRECEDES THIS LINE
# THE REMAINDER OF THIS FILE IS PURE GENERIC Makefile

CFLAGS = -g
TMP    = /usr/tmp/bcg.`date "+%m.%d,%H"`

run : $(EXEC)
        rm -f $(EXEC).t"
        -$(EXEC)

$(EXEC) : tags $(OBJS)
        cc -o $(EXEC) $(OBJS)

save : .save dependencies
        cp $(SRCS) $(HDRS) $(INCS) Makefile .save
        make nusend FILE=$(TMP)

nusend :
        (cd ..; find P R -print | cpio -oacB > $(FILE))
        sync
        ls -l /usr/tmp/bcg.*
        if [ `uname -n` != "drux2" ];                           \
        then                                                    \
                nusend -d drux2 $(FILE);                        \
        fi
        if [ `uname -n` != "drux3" ];                           \
        then                                                    \
                nusend -d drux3 $(FILE);                        \
        fi
        if [ `uname -n` != "druor" ];                           \
        then                                                    \
                nusend -d druor $(FILE);                        \
        fi

stats :
        @wc $(SRCS) $(HDRS) $(INCS) Makefile > $(EXEC).stats
        @echo "\nYou have written" `cat tags|wc -l` "routines. in" \
                `cat $(SRCS) $(HDRS) $(INCS) M" | wc -l` "lines."    \
                | tee -a $(EXEC).stats

lint : $(EXEC).lint
        more $(EXEC).lint

$(EXEC).lint : $(SRCS) $(HDRS) $(INCS)
        lint $(SRCS) > $(EXEC).lint

ksuprint :
        xcl -y ksu -f nohole Makefile $(SRCS) $(HDRS) $(INCS)

print : stats lint save
        xcl -y ksu Makefile $(SRCS) $(HDRS) $(INCS) $(EXEC).[tpl]"

print.more : stats $(EXEC).lint
        cflow $(SRCS) > $(EXEC).cflow
        -cxref -c -t $(SRCS) $(HDRS) $(INCS) > $(EXEC).cxref
        xcl -y ksu $(EXEC)."

tags : $(SRCS)
        ctags $(SRCS) &

clean :
        rm -f ".o $(EXEC) core

.save :
        -mkdir .save .bak1 .bak2

sane :
        stty  sane  erase \b  kill @  echoe
```

```
 78  # Commands to set up header dependencies automatically in Makefile:
 79  # grep outputs lines from 1 or more c language files like this:
 80  #       foo.c:#include "bar.h"
 81  # These lines (and NOT lines like "#include <stdio.h>") become:
 82  #       foo.o: bar.h
 83  # These resulting lines are put at the end of Makefile.
 84  # NOTE: use VPATH (defined in .profile) if header files are not in ".":
 85  #       VPATH = :$HOME/usr/include:$PROJ/include        ; export VPATH

 87  dependencies : .save
 88          @echo "\tRecreating Makefile dependencies..."
 89          @cp Makefile .save/oMakefile
 90          @ecno '/^# HEADER DEPENDENCIES FOLLOW/+1,$$d\nw' | ed - Makefile
 91          @ecno '# '`date` >> Makefile
 92          @grep '^#include' $(SRCS) $(HDRS) $(INCS) /dev/null      \
 93                  | sed 's/:[^"]*"\([^"]*\)"./: \t\1/'               \
 94                  | sed '/<.*>/d'                                   \
 95                  | sed 's/\.c/.o/'                                 \
 96                  | sort >> Makefile
 97          @echo "# DO NOT PUT ANYTHING BELOW THIS LINE!" >> Makefile

 99  # HEADER DEPENDENCIES FOLLOW:
100  # Tue Nov 20 16:58:38 MST 1984
101  T_funtab.o:     T_funtab.i
102  T_funtab.o:     T_tables.i
103  T_print.o:      T_funtab.i
104  T_print.o:      T_pnltab.i
105  T_print.o:      T_symtab.i
106  T_print.o:      T_tables.i
107  T_symfun.o:     T_funtab.i
108  T_symfun.o:     T_symtab.i
109  T_symfun.o:     T_tables.i
110  T_symtab.o:     T_symtab.i
111  T_symtab.o:     T_tables.i
112  T_tables.i:     T_tables.h
113  T_tables.i:     debug.h
114  T_tables.i:     files.h
115  T_tables.i:     project.h
116  debug.o:        project.h
117  input.o:        debug.h
118  input.o:        project.h
119  interpret.o:    T_funtab.i
120  interpret.o:    T_pnltab.i
121  interpret.o:    T_symtab.i
122  interpret.o:    T_tables.i
123  interpret.o:    debug.h
124  main.o:         debug.h
125  main.o:         files.h
126  main.o:         project.h
127  scan.o:         T_tables.h
128  scan.o:         debug.h
129  scan.o:         project.h
130  translate.o:    T_funtab.i
131  translate.o:    T_pnltab.i
132  translate.o:    T_symtab.i
133  translate.o:    T_tables.i
134  translate.o:    debug.h
135  # DO NOT PUT ANYTHING BELOW THIS LINE!
```

```
/*************************************************************
 *
 * MODULE:       MAIN
 *
 * Translate Entity-Relationship-Attribute Requirements Specifications
 *           into Petri Net Language Programs
 *                and Interpret the Resulting Prototype
 *
 * PROJECT:      Requirements Analysis using Petri Nets
 *               CMPSC 690 - Master's Project, Summer 1984
 *
 * AUTHOR:       Brad C. Gaylord
 *               AT&T Information System Laboratories
 *               11900 North Pecos, Denver CO.  80234
 *               drux2!bcg, DR 31L60,  (303)-538-1413
 *
 *************************************************************
 *
 * USED BY:      all modules
 *
 * ACCESS FCTS:  init(), fini()
 *
 *************************************************************
 *
 * USES:         TRANSLATE, INTERPRET
 *
 * DATA OWNED:   n/a
 *
 *************************************************************
 */

/* INCLUDE files required by this module: */
#include "project.h"                    /* general definitions      */
#include "files.h"                      /* file names               */

#define  D_MODULE       D_MAIN          /* primary module trace level */
#include "debug.h"                      /* debug levels             */

/* global variables: */
char * program = "P";                   /* used by DEBUG module     */


/* PREFIXES used in #defines in this implementation:
 *              D_      print encodes for tracing with debug()
 *              E_      entities
 *              R_      relations
 *              T_      tables (ie symbol tables)
 */

/*
 *  PREFIXES used in definitions (in MODULES) in this implementation
 *              Dbug_   debug.c
 *              Inp_    input.c
 *              st_     routines which store fields in tables.c
 */
```

```
60  /*
61   * main ()
62   *
63   * This is the first routine to be executed.
64   *
65   * This routine uses     - TRANSLATE module via translate()
66   *                       - INTERPRET module via interpret()
67   *
68   * INPUT:
69   *       called by:              exec(2) via sh(1)
70   *       input parameters:       n/a
71   *
72   * OUTPUT:
73   *       return type & value:    fini() causes process exit
74   *
75   * GLOBAL:
76   *       variables used:         n/a
77   *       variables changed:      n/a
78   */

80  main ()
81  {
82          VOID init(), translate(), interpret(), fini();

84          /* initialize data structures and data files */
85          init ();

87          /* translate ERA spec into PNL entity graph */
88          translate ();

90          /* interpret the Petri net structure */
91          interpret ();

93          /* cause successful process termination */
94          fini ( SUCCESS );

96  }  /* main () */
```

```
 99  /*
100   * init ()
101   *
102   * orderly initialization of this system
103   *
104   * INPUT:
105   *       called by:              main ()
106   *       input parameters:       none
107   *
108   * OUTPUT:
109   *       return type & value:    n/a
110   *
111   * GLOBAL:
112   *       variables used:         n/a
113   *       variables changed:      n/a
114   */

116   VOID
117  init ()
118  {
119           VOID Inp_init(), st_init(), pnl_init();

121           BEGIN ( "init" );

123           Inp_init ( IN_FILE );          /* open input files          */

125           st_init ();                    /* initialize symbol tables */

127           pnl_init ();                   /* initialize PNL tables     */

129           RETURN;

131  }  /* init () */
```

```
134  /*
135   * fini ()
136   *
137   * Orderly shutdown of the system.
138   *
139   * This is the ONLY routine to cause process exit,
140   * once the process is initialized.
141   *
142   * INPUT:
143   *      called by:              main ()
144   *      input parameters:       status (SUCCESS OR FAILURE)
145   *
146   * OUTPUT:
147   *      return type & value:    integer return code to sh(1)
148   *
149   * GLOBAL:
150   *      variables used:         n/a
151   *      variables changed:      n/a
152   */

154   VOID
155  fini ( status )
156   int status;
157  {
158          VOID pr_tables(), Inp_fini(), Dbug_fini(), sync(), exit();

160          BEGIN ( "fini" );

162          pr_tables ();                   /* print tables         */

164          Inp_fini ();                    /* close input files    */

166          Dbug_fini ();                   /* close debugging      */

168          sync ();                        /* write memory to disk */

170          if ( status EQ SUCCESS )  {
171                  exit ( status );        /* successful process!  */
172          }  else  {
173                  ABORT ();               /* dump core for sdb    */
174          }

176  }  /* fini () */
```

```
/*************************************************************
 *
 * MODULE:       SCAN
 *
 * Lexical scanner which fills the symbol/function table with "tokens".
 *
 * PROJECT:      Requirements Analysis using Petri Nets
 *               CMPSC 690 - Master's Project, Summer 1984
 *
 * AUTHOR:       Brad C. Gaylord
 *               AT&T Information System Laboratories
 *               11900 North Pecos, Denver CO.  80234
 *               drux2!bcg, DR 31L60.  (303)-538-1413
 *
 *************************************************************
 *
 * USED BY:      TRANSLATE
 *
 * ACCESS FCTS:  scan()
 *               is_fun_name(), is_iod_name(), is_mode_name()
 *               is_f_or_i(), pr_class()
 *
 *************************************************************
 *
 * USES:         TABLES, INPUT
 *
 * DATA OWNED:   valid_entity[], valid_relation[]
 *
 *************************************************************
 */

/* INCLUDE files required by this module: */
#include <stdio.h>                      /* for EOF                  */
#include <ctype.h>                      /* for isupper ()           */
#include "project.h"                    /* general definitions      */
#include "T_tables.h"                   /* symbol tables, etc       */

#define  D_MODULE       D_SCAN          /* primary module trace level */
#include "debug.h"                      /* debug trace levels       */

/* the significant types of lines in an input file; see scan () */
#define AFTER_PROCESS_LINE      0
#define BLANK_LINE              1
#define ENTITY                  2
#define RELATION                3
#define MODE_TABLE              4
#define MODE_INITIAL            5
#define MODE_TRANSITIONS        6
```

```
52  /*
53   * To add new entries to valid_entity[] or valid_relation[], put the
54   * new string at the end of the appropriate list, and add the
55   * corresponding #define index.  Increment the LAST_ENTRY #define.
56   * Then find the switch statements which use the class and type
57   * fields and add the code for the new cases.
58   */



62  /*
63   * entities always begin in column 1 with a capital letter
64   */

66  /* the following #defines represent indices into valid_entity[] */
67  #define E_MODE                  T_MODE          /* 0 */
68  #define E_ACTIVITY              1
69  #define E_PERIODIC_FUNCTION     2
70  #define E_INPUT                 3
71  #define E_OUTPUT                4
72  #define E_INPUT_OUTPUT          5
73  #define E_LAST_ENTRY            6

75  char * valid_entity [] = {
76          "Mode",
77          "Activity",
78          "Periodic_function",
79          "Input",
80          "Output",
81          "Input_output"
82  };



86  /*
87   * relations are lowercase words which are the first token on a line
88   * after at least one blank
89   */

91  /* the following #defines represent indices into valid_relation[] */
92  #define R_NECESSARY_CONDITION   0
93  #define R_INPUT                 1
94  #define R_OUTPUT                2
95  #define R_REQUIRED_MODE         3
96  #define R_OCCURRENCE            4
97  #define R_MEDIA                 5
98  #define R_LAST_ENTRY            6

100 char *  valid_relation [] = {
101         "necessary_condition",
102         "input",
103         "output",
104         "required_mode",
105         "occurrence",
106         "media"
107 };


110 BOOL    ent_ok = NO;            /* not an interesting entity yet...  */
111 char    era_title [WORD_LEN];   /* title of this PROCESS             */
112 char *  era_line;               /* points to current line            */
```

```
114  /*
115   * is_fun_name (), is_iod_name (), is_mode_name ()
116   *
117   * routines to guarantee lexical conventions for the names of:
118   *       - Function entities
119   *       - I/O & Data entities
120   *       - Modes
121   *
122   * INPUT:
123   *       called by:              routines in the TABLES module
124   *       input parameters:       rname - calling routine's name
125   *                               sym   - rhs of an era line
126   *
127   * OUTPUT:
128   *       return type & value:    VOID
129   *
130   * GLOBAL:
131   *       variables used:         n/a
132   *       variables changed:      n/a
133   */



137   VOID
138  is_fun_name ( rname, sym )
139   char * rname;                                /* calling routine's name  */
140   char * sym;                                  /* is sym an era function? */
141  {
142          if ( NOT isupper ( sym[0] ) )  {
143                  debug ( rname, D_ERROR, "Invalid name: %s", sym );
144                  ERROR ( "FUNCTION names begin with a capitol letter" );
145          }

147  }  /* is_fun_name () */



151   VOID
152  is_iod_name ( rname, sym )
153   char * rname;                                /* calling routine's name  */
154   char * sym;                                  /* is sym an era I/O/DATA? */
155  {
156          /* caveat: should check for all lowercase letters in sym */

158          if ( (sym[0] NE '$') AND (sym[strlen(sym)] NE '$' ) )  {
159                  debug ( rname, D_ERROR, "Invalid name: %s", sym );
160                  ERROR ( "I_O_DATA names are delimited by '$' signs");
161          }

163  }  /* is_iod_name () */



167   VOID
168  is_mode_name ( rname, sym )
169   char * rname;                                /* calling routine's name  */
170   char * sym;                                  /* is sym an era Mode? */
171  {
172          /* caveat: should check for all capital letters in sym */

174          if ( (sym[0] NE '*') AND (sym[strlen(sym)] NE '*' ) )  {
175                  debug ( rname, D_ERROR, "Invalid name: %s", sym );
176                  ERROR ( "MODE names are delimited by '*' signs");
177          }

179  }  /* is_mode_name () */
```

```
182  /*
183   * scan ()
184   *
185   * This scanner follows the approach suggested in Appendix B:
186   *       - First a PROCESS line must be identified.
187   *       - null lines & comments are removed
188   *       - the ordering of input lines is checked by line_syntax()
189   *       - scan() returns when a MODE_TABLE is encountered and processed.
190   *
191   * INPUT:
192   *       called by:                translate()
193   *
194   * OUTPUT:
195   *       return type & value:      VOID
196   *
197   * GLOBAL:
198   *       variables used:           ent_ok
199   *       variables changed:        era_line
200   */

202   VOID
203  scan ()
204  {
205     NUM   process_line();          /* read the era PROCESS line     */
206     char *get_line();              /* get an era input line         */
207     BOOL  null_line();             /* removes non-useful lines      */
208     NUM   line_syntax();           /* check syntax of era lines     */
209     VOID  mode_table();            /* process the mode_table        */
210     VOID  st_orphan();             /* find symtab orphans           */
211     VOID  st_invokes();            /* resolve annotations           */

213     char  lhs [ WORD_LEN ];        /* left hand side of line        */
214     char  rhs [ WORD_LEN ];        /* right hand side of line       */
215     NUM   s_count;                 /* number of sscanf matches      */
216     NUM   line_type;               /* current line type             */

218     BEGIN ( "scan" );

220     /* read the era PROCESS line */
221     line_type = process_line ();

223     /* process the entities between <era_title> and <mode_table> */
224     while ( era_line = get_line () )  {
225           s_count = sscanf ( era_line, "%s : %s", lhs, rhs );

227           /* break when the MODE_TABLE is encountered */
228           if ( eqstr ( era_line, "MODE_TABLE" ) )  {
229                 line_type = MODE_TABLE;
230                 break;
231           }

233           /* remove comments and non-useful lines */
234           if ( null_line ( rname, &line_type, s_count, lhs ) )  {
235                 continue;
236           }

238           if ( s_count < 2 )  {
239                 ERROR ( "failed to match '<lhs> : <rhs>'" );
240           }

242           /* analyze the gross ordering syntax of this input line */
243           line_type = line_syntax ( line_type, era_line, lhs, rhs );

245           debug ( rname, D_TRACE, "\n" );
246     } /* while */

248     if ( line_type NE MODE_TABLE )  {
249           ERROR ( "MODE_TABLE not encountered" );
250     }  else  {
251           mode_table ();                    /* process the mode_table */
252           st_orphan ( E_INPUT, E_OUTPUT); /* find symtab orphans     */
253           st_invokes ( E_INPUT_OUTPUT );  /* resolve annotations     */
254     }

256     RETURN;
257  }  /* scan () */
```

```
259  /*
260   * process_line ()
261   *
262   * scan the first line in the era file for a PROCESS line.
263   *
264   * INPUT:
265   *       called by:              scan()
266   *
267   * OUTPUT:
268   *       return type & value:    line_type = AFTER_PROCESS_LINE
269   *
270   * GLOBAL:
271   *       variables used:         n/a
272   *       variables changed:      era_line, era_title
273   */

275   NUM
276  process_line ()
277  {
278          char * get_line();

280          BEGIN ( "process_line" );

282          /* read the <era_title> PROCESS line */
283          era_line = get_line ();

285          /* scan for correct format of PROCESS line */
286          if ( sscanf ( era_line, "PROCESS : %s", era_title ) EQ 1 )  {
287                  debug ( rname, D_TRACE, "PROCESS : %s\n", era_title );
288          }  else  {
289                  ERROR ( "No PROCESS line in specification" );
290          }

292          RETURN ( AFTER_PROCESS_LINE );

294  }  /* process_line () */
```

```
296  /*
297   * line_syntax ()
298   *
299   * Each line is classified as a BLANK_LINE, an ENTITY line or a
300   * RELATION line.  This code tries to ensure that the following
301   * syntax is adhered to: BLANK_LINEs precede ENTITY lines (which
302   * begin with a capital letter in column 1), and ENTITY lines precede
303   * one or more RELATION lines (lower-case & not beginning in column 1).
304   *
305   * line_syntax () calls entity () to process ENTITY lines.  Only those
306   * entities which are of interest to this process (ent_ok == YES)
307   * are examined in greater detail by relation ().
308   *
309   * INPUT:
310   *      called by:              scan()
311   *
312   * OUTPUT:
313   *      return type & value:    line_type - current line's type
314   *
315   * GLOBAL:
316   *      variables used:         ent_ok, prev_type
317   *      variables changed:      ent_ok
318   */

320  NUM
321 line_syntax ( prev_type, line, lhs, rhs )
322  NUM prev_type;                          /* previous line type       */
323  char * line;                            /* ERA input line           */
324  char * lhs;                             /* left-hand side of line   */
325  char * rhs;                             /* right-hand side of line  */
326 {
327         BOOL entity();          /* identify interesting entities */
328         VOID relation();        /* process relations of said entities*/

330         NUM ret;

332         BEGIN ( "line_syntax" );

334         /* generate a hypothesis as to the identity of the line */
335         /* based upon column 1 and the previous line's type     */
336         switch ( prev_type )  {
337         case BLANK_LINE:
338                 /* entity lines follow blank lines */
339                 if ( isupper ( line [0] ) )  {
340                         ent_ok = entity ( lhs, rhs );
341                         ret = ENTITY;
342                 }  else  {
343                         ERROR ( "illegal: <NL> <relation_line>" );
344                 }
345                 break;

347         case ENTITY:
348         case RELATION:
349                 /* relation lines follow these 2 kinds of line. */
350                 /* entity lines follow neither kind of these.   */
351                 if ( isupper ( line [0] ) )  {
352                         ERROR ( "Entity must follow blank line" );
353                 }  else  {
354                         if ( ent_ok )  {
355                                 /* examine the relations */
356                                 relation ( lhs, rhs );
357                         }
358                         ret = RELATION;
359                 }
360                 break;

362         default:
363                 debug ( rname, D_ERROR, "prev_type=%d", prev_type );
364                 ERROR ( "internal error" );
365         }  /* switch */

367         RETURN ( ret );

369 }  /* line_syntax () */
```

```
371  /*
372   * entity ()
373   *
374   * if this is an interesting entity line, enter it into the symtab.
375   *
376   * INPUT:
377   *       called by:              scan ()
378   *
379   * OUTPUT:
380   *       return type & value:    BOOL YES - this entity is useful
381   *                                    NO  - this entity is not useful
382   *
383   * GLOBAL:
384   *       variables used:         valid_entity[]
385   *       variables changed:      n/a
386   */

388   BOOL
389  entity ( lhs, rhs )
390   char * lhs;                              /* left-hand side of line  */
391   char * rhs;                              /* right-hand side of line */
392  {
393          VOID st_entity();          /* adds entities to the symbol table */
394          NUM  is_f_or_i();          /* T_FUNCTION or T_I_O_DATA?         */

396          NUM  class;

398          BEGIN ( "entity" );

400          for ( class=0; class < E_LAST_ENTRY; class++)  {
401                  if ( NOT eqstr ( lhs, valid_entity [class] ) )  {
402                          /* ignore this keyword: it's for someone else */
403                          continue;
404                  }

406                  /* add entity name to symtab */
407                  st_entity ( rhs, class, is_f_or_i ( class ) );

409                  RETURN (YES);

411          }  /* for */

413          RETURN (NO);

415  }  /* entity () */
```

```
418  /*
419   * relation ()
420   *
421   * is this an interesting relation line?
422   * if so, populate the table corresponding to this relation class
423   *      (FUNCTION or I_O_DATA)
424   *
425   * NOTE: no check is made for lowercase only in relation names;
426   *       different entities can't share any entity relation types
427   *
428   * INPUT:
429   *      called by:              scan ()
430   *
431   * OUTPUT:
432   *      return type & value:    VOID
433   *
434   * GLOBAL:
435   *      variables used:         valid_relation[]
436   *      variables changed:      n/a
437   */

439  VOID
440  relation ( lhs, rhs )
441   char * lhs;
442   char * rhs;
443  {
444      VOID st_nec_cond(), st_input(), st_output(), st_req_mode();
445      VOID st_note(), st_media(), st_warn();
446      NUM  is_p_or_a();

448      register    class;
```

```
450       BEGIN ( "relation" );

452       for ( class=0; class < R_LAST_ENTRY; class++ )  {
453           if ( NOT eqstr ( lhs, valid_relation [class] ) )  {
454                   /* ignore this keyword; it's for someone else */
455                   continue;
456           }

458           switch ( class )  {
459           case R_NECESSARY_CONDITION:
460                   if ( is_p_or_a() NE E_ACTIVITY )  {
461                           ERROR ( "only used in Activity entities" );
462                   }
463                   st_nec_cond ( rhs );
464                   break;

466           case R_INPUT:
467                   /* if not Input_output type */
468                   if ( is_p_or_a() EQ E_PERIODIC_FUNCTION )  {
469                           st_warn ( "don't use 'input' relations in Periodic_functi
    ions" );
470                   }
471                   st_input ( rhs );
472                   break;

474           case R_OUTPUT:
475                   st_output ( rhs );
476                   break;

478           case R_REQUIRED_MODE:
479                   st_req_mode ( rhs );
480                   break;

482           case R_OCCURRENCE:
483                   if ( is_p_or_a() NE E_PERIODIC_FUNCTION )  {
484                           ERROR ( "only used in Periodic_function entities" );
485                   }
486                   st_note ( era_line );
487                   break;

489           case R_MEDIA:
490                   /* caveat: no verification done on device type */
491                   /*         as it relates to IO_DATA type       */
492                   st_media ( rhs );
493                   break;

495           default:
496                   debug ( rname, D_ERROR, "class=%d", class );
497                   ERROR ( "internal error" );
498           }  /* switch */

500           RETURN;
501       }  /* for */

503       RETURN;

505   }  /* relation () */
```

```
507  /*
508   * mode_table ()
509   *
510   * process the sections in the MODE_TABLE - the list of Modes
511   *                                       - the Initial_Mode
512   *                                       - Allowed_Mode_Transitions
513   *
514   * INPUT:
515   *      called by:              scan ()
516   *
517   * OUTPUT:
518   *      return type & value:    VOID
519   *
520   * GLOBAL:
521   *      variables used:         n/a
522   *      variables changed:      era_line
523   */

525   VOID
526  mode_table ()
527  {
528      char * get_line();                  /* get an era input line     */
529      VOID   st_verify();                 /* verify modes              */
530      VOID   st_i_mode();                 /* store Initial_Mode        */
531      VOID   st_mult_modes();             /* clone duplicate Modes     */
532      VOID   st_transitions();            /* generate nxt_modes        */

534      char   lhs [ WORD_LEN ];            /* left hand side of line    */
535      char   io_val  [ WORD_LEN ];        /* I_O_DATA value            */
536      char   mode1 [ WORD_LEN ];          /* mode:  current mode       */
537      char   mode2 [ WORD_LEN ];          /* mode:  next mode          */
538      NUM    s_count;                     /* number of sscanf matches  */
539      NUM    line_type;                   /* current line type         */
540      NUM    mode_state;                  /* which 1/2 of mode table?  */
```

```
542       BEGIN ( "mode_table" );

544       mode_state = MODE_TABLE;

546       while ( era_line = get_line() )  {
547           switch ( mode_state )  {
548           case MODE_TABLE:
549                   lhs[0] = mode1[0] = NULL;
550                   s_count = sscanf ( era_line, "%s : %s", lhs, mode1 );

552                   /* remove comments and non-useful lines */
553                   if ( null_line ( rname, &line_type, s_count, lhs ) )  {
554                           continue; /* with next era_line in the while loop */
555                   }

557                   if ( eqstr ( lhs, valid_entity[E_MODE] ) )  {
558                           st_verify ( mode1 );
559                   }  else if ( eqstr ( lhs, "Initial_Mode" ) )  {
560                           /* store the initial mode */
561                           st_i_mode ( mode1 );

563                           /* all of the modes have been read */
564                           st_mult_modes ();

566                           mode_state = MODE_INITIAL;
567                   }
568                   break;

570           case MODE_INITIAL:
571                   lhs[0] = NULL;
572                   s_count = sscanf ( era_line, "%s :", lhs );

574                   /* remove comments and non-useful lines */
575                   if ( null_line ( rname, &line_type, s_count, lhs ) )  {
576                           continue; /* with next era_line in the while loop */
577                   }

579                   if ( eqstr ( lhs, "Allowed_Mode_Transitions" ) )  {
580                           /* process the last part of the MODE_TABLE */
581                           mode_state = MODE_TRANSITIONS;
582                           continue; /* with next era_line in the while loop */
583                   }
584                   break;

586           case MODE_TRANSITIONS:
587                   lhs[0] = io_val[0] = mode1[0] = mode2[0] = NULL;
588                   s_count = sscanf ( era_line, "%s : %s -> %s",
589                                      lhs, mode1, mode2 );
590                   if ( s_count EQ 1 )  {
591                           s_count = sscanf ( era_line, "%s = %s : %s -> %s",
592                                              lhs, io_val, mode1, mode2 );
593                   }

595                   /* remove comments and non-useful lines */
596                   if ( null_line ( rname, &line_type, s_count, lhs ) )  {
597                           continue; /* with next era_line in the while loop */
598                   }

600                   st_transitions ( lhs, io_val, mode1, mode2 );
601                   break;

603           default:
604                   debug ( rname, D_ERROR, "mode_state=%d", mode_state );
605                   ERROR ( "internal error" );
606           } /* switch */


609       } /* while */

611       RETURN;

613   } /* mode_table () */
```

```
615  /*
616   * null_line ()
617   *
618   * The following will be transparently removed wherever they occur:
619   *      Comments         lines beginning with "Comment" or "comment"
620   *      Blank lines      lines with nothing on them
621   *      ".bp" lines      these allow for pagination in 'nroff' appendices
622   *
623   * INPUT:
624   *      called by:               scan(), mode_table()
625   *      input parameters:        various info about current era line
626   *
627   * OUTPUT:
628   *      return type & value:     YES      - ignore this line
629   *                               NO       - process this line
630   *
631   * GLOBAL:
632   *      variables used:          n/a
633   *      variables changed:       line_type (in scan())
634   */

636   BOOL
637  null_line ( rname, line_type_addr, s_count, lhs )
638   char * rname;                              /* calling routine's name    */
639   NUM  * line_type_addr;                     /* previous line's type      */
640   NUM    s_count;                            /* number of tokens in line  */
641   char * lhs;                                /* left-hand side of line    */
642  {
643          /* test for a blank line or ".bp" */
644          if ( ( s_count EQ EOF ) OR eqstr ( lhs, ".bp" ) )  {
645                  if ( *line_type_addr EQ ENTITY )  {
646                          /* a relation should follow <entity_line> */
647                          ERROR ( "illegal: <entity_line> <NL>" );
648                  }  else  {
649                          *line_type_addr = BLANK_LINE;
650                          debug ( rname, D_TRACE, "BLANK_LINE\n" );
651                          return ( YES );
652                  }
653          }

655          /* exhaust all comment lines */
656          if ( eqstr ( lhs, "Comment" ) OR
657               eqstr ( lhs, "comment" ) )  {
658                  debug ( rname, D_TRACE, "%s ignored\n", lhs );
659                  return ( YES );
660          }

662          return ( NO );

664  }  /* null_line () */
```

```
666  /*
667   * is_f_or_i ()
668   *
669   * return one of the primary table type encodes
670   *
671   *.INPUT:
672   *      called by:              TABLES module
673   *
674   * OUTPUT:
675   *      return type & value:    T_MODE     - E_MODE & ERRORs in class
676   *                              T_FUNCTION - Activities, Periodics
677   *                              T_I_O_DATA - Inputs, Outputs, Input_outputs
678   *
679   * GLOBAL:
680   *      variables used:         n/a
681   *      variables changed:      n/a
682   */

684   NUM
685  is_f_or_i ( class )
686   NUM     class;                          /* class of symtab entry */
687  {
688           NUM f_or_i;

690           switch ( class )  {
691           case E_MODE:
692                   f_or_i = T_MODE;
693                   break;

695           case E_ACTIVITY:
696           case E_PERIODIC_FUNCTION:
697                   f_or_i = T_FUNCTION;
698                   break;

700           case E_INPUT:
701           case E_OUTPUT:
702           case E_INPUT_OUTPUT:
703                   f_or_i = T_I_O_DATA;
704                   break;

706           default:
707                   /* This strange error return is used to prevent      */
708                   /* switch statements from blowing up in pr_entity(). */
709                   /* which is called from fini.  Blow ups cause        */
710                   /* switch defaults to call ERROR which calls fini.   */
711                   /* This indirect recursion is not useful.            */
712                   f_or_i = T_MODE;
713                   /* ERROR ( "internal error" ); */
714           }  /* switch */

716           return ( f_or_i );

718  }  /* is_f_or_i () */
```

```
720  /*
721   * pr_class ()
722   *
723   * print the class of an entity stored in the symbol table.
724   *
725   * class encodes are "owned" by the SCAN module
726   *
727   * INPUT:
728   *      called by:              pr_entity()
729   *
730   * OUTPUT:
731   *      return type & value:    VOID
732   *
733   * GLOBAL:
734   *      variables used:         n/a
735   *      variables changed:      n/a
736   */

738   VOID
739  pr_class ( fp, class )
740   FILE * fp;                              /* file pointer for output */
741   NUM    class;                           /* class of symtab entry   */
742  {
743          BEGIN ( "pr_class" );

745          FPRINTF ( fp, "class        : " );

747          switch ( class )  {
748          case T_ORPHAN:
749                  FPRINTF ( fp, "ORPHAN\n" );
750                  break;

752          case E_MODE:
753          case E_ACTIVITY:
754          case E_PERIODIC_FUNCTION:
755          case E_INPUT:
756          case E_OUTPUT:
757          case E_INPUT_OUTPUT:
758                  FPRINTF ( fp, "%s\n", valid_entity[class] );
759                  break;

761          default:
762                  debug ( rname, D_ERROR, "class=%d", class );
763                  /* ERROR ( "internal error" ); */
764          }  /* switch */

766          RETURN;

768  }  /* pr_class () */
```

```
/*************************************************************
 *
 * MODULE:       TRANSLATE
 *
 * Creator of the Entity Graph (a Petri Net Structure)
 *
 * PNL code generator
 *
 * PROJECT:      Requirements Analysis using Petri Nets
 *               CMPSC 690 - Master's Project, Summer 1984
 *
 * AUTHOR:       Brad C. Gaylord
 *               AT&T Information System Laboratories
 *               11900 North Pecos, Denver CO,  80234
 *               drux2!bcg, DR 31L60,  (303)-538-1413
 *
 *************************************************************
 *
 * USED BY:      main
 *
 * ACCESS FCTS:  translate()
 *
 *************************************************************
 *
 * USES:         TABLES
 *
 * DATA OWNED:   places[], trans[], pl_idx, tr_idx  (see T_pnltab.[ci])
 *
 *************************************************************
 */


/* INCLUDE files required by this module: */
#include "T_tables.i"                   /* general table definitions  */
#include "T_symtab.i"                   /* symbol table               */
#include "T_funtab.i"                   /* function table             */

#define  extern                         /* TRANSLATE owns T_pnltab.i  */
#include "T_pnltab.i"                   /* PNL tables                 */


#undef   D_MODULE                       /* was defined in T_tables.i  */
#define  D_MODULE       D_TRANSLATE     /* primary module trace level */
#include "debug.h"                      /* debug levels               */


/* DEFINEs local to this module: */


/* global variables "local" to this module: */
```

```
54  /*
55   * translate ()
56   *
57   * Translate the symbol table into place and transitions
58   * and generate PNL code.
59   *
60   * INPUT:
61   *      called by:              main()
62   *
63   * OUTPUT:
64   *      return type & value:    VOID
65   *
66   * GLOBAL:
67   *      variables used:         n/a
68   *      variables changed:      n/a
69   */

71   VOID
72  translate ()
73  {
74          VOID scan(), p_nodes(), p_pnl();

76          BEGIN ( "translate" );

78          /* fill the symbol and function tables */
79          scan ();

81          /* create the Places and Transitions */
82          p_nodes ();

84          /* output the PNL code  */
85          p_pnl ();

87          RETURN;

89  }  /* translate () */
```

```
 91  /*
 92   * p_nodes ()
 93   *
 94   * create the Place and Transition nodes from the T_FUNCTION entities
 95   *
 96   * INPUT:
 97   *      called by:              translate()
 98   *      input parameters:       n/a
 99   *
100   * OUTPUT:
101   *      return type & value:    VOID
102   *
103   * GLOBAL:
104   *      variables used:         Init_Mode, sym_idx, symtab[]
105   *      variables changed:      n/a
106   */

108  VOID
109  p_nodes ()
110  {
111          VOID tr_out();
112          NUM  is_f_or_i(), p_place(), p_transition();

114          NUM  idx;                       /* iterator             */
115          NUM  p_idx;                     /* place table index    */
116          NUM  t_idx;                     /* transition table index */

118          BEGIN ( "p_nodes" );

120          /* store the Place that the initial Transition outputs to */
121          p_idx = p_place ( Init_Mode );

123          /* store the initial Transition and it's output index */
124          t_idx = p_transition ( NULL, P_INIT );
125          tr_out ( t_idx, p_idx );

127          /* create the rest of the Places and Transitions */
128          for ( idx=0; idx<=sym_idx; idx++ )  {
129                  if ( is_f_or_i(symtab[idx].sym_class) NE T_FUNCTION )  {
130                          continue;                       /* ??? */
131                  }

133                  /* put the function in the transition table */
134                  (VOID) p_transition ( idx, P_FUNC );
135          }

137          RETURN;

139  }  /* p_nodes () */
```

```
141  /*
142   * p_pnl ()
143   *
144   * write the PNL code to an output file
145   *
146   * INPUT:
147   *       called by:              translate()
148   *       input parameters:       n/a
149   *
150   * OUTPUT:
151   *       return type & value:    VOID
152   *
153   * GLOBAL:
154   *       variables used:         PNL_FILE
155   *       variables changed:      n/a
156   */

158   VOID
159  p_pnl ()
160  {
161          VOID   perror();
162          VOID   pr_initial(), pr_places(), pr_transitions(), pr_terminals();

164          FILE * Pnl_fp;                          /* PNL_FILE */

166          BEGIN ( "p_pnl" );

168          /* open the PNL file */
169          if ( ( Pnl_fp = fopen ( PNL_FILE, "w" ) ) EQ NULL )  {
170                  perror ( "translate: can't open Pnl_fp" );
171                  ERROR ( "can't open PNL_FILE for writing" );
172          }

174          /* print the Initial Transition */
175          pr_initial ( Pnl_fp, P_INIT );

177          /* print the Places */
178          pr_places ( Pnl_fp );

180          /* print the Function Transitions */
181          pr_transitions ( Pnl_fp, P_FUNC );

183          /* print the Terminal Transitions */
184          pr_terminals ( Pnl_fp, P_TERM );

186          /* close the PNL file */
187          if ( fclose ( Pnl_fp ) EQ EOF )  {
188                  perror ( "translate: can't close Pnl_fp" );
189          }

191          RETURN;

193  }  /* p_pnl () */
```

```
195  /*
196   * p_place ()
197   *
198   * Add a symbol to the places table if it is not already there.
199   *
200   * Return the index to the symbol's entry in the table.
201   *
202   * INPUT:
203   *      called by:              p_nodes(), p_transitions()
204   *      input parameters:       symbol table index
205   *
206   * OUTPUT:
207   *      return type & value:    current table index
208   *
209   * GLOBAL:
210   *      variables used:         symtab[]
211   *      variables changed:      places[], pl_idx
212   */

214   NUM
215  p_place ( st_idx )
216   NUM st_idx;
217  {
218          NUM pl_find();

220          NUM idx;

222          BEGIN ( "p_place" );

224          if ( HAS_NO_VALUE (idx=pl_find(symtab[st_idx].sym)) )  {
225                  if ( pl_idx < PLACES_SIZE-1 )  {
226                          idx = ++pl_idx;
227                  }  else  {
228                          ERROR ( "places table overflow" );
229                  }

231                  /* put the symbol name in the places table */
232                  STRCPY ( places[idx].sym, symtab[st_idx].sym );

234                  if ( HAS_A_VALUE (symtab[st_idx].comment[0]) )  {
235                          STRCPY ( places[idx].comment, symtab[st_idx].comment );
236                  }
237          }

239          RETURN ( idx );

241  }  /* p_place () */
```

```
243  /*
244   * p_transition ()
245   *
246   * Add a symbol to the transitions table if it is not already there.
247   *
248   * Return the index to the symbol's entry in the transition table.
249   *
250   * INPUT:
251   *      called by:              p_nodes(), p_transitions()
252   *      input parameters:       symbol table index, and transition type
253   *
254   * OUTPUT:
255   *      return type & value:    current table index
256   *
257   * GLOBAL:
258   *      variables used:         symtab[], funtab[]
259   *      variables cnanged:      trans[], tr_idx, places[]
260   */

262  NUM
263  p_transition ( st_idx, type )
264   NUM st_idx;
265   NUM type;                      /* transition type: INIT, FUNC, TERM */
266  {
267          VOID tr_in(), tr_out();
268          NUM  tr_find();

270          NUM  f_idx;            /* function table index      */
271          NUM  t_idx, out_idx;   /* transition table indices  */
272          NUM  i;                /* iterator                  */
273          char sym [WORD_LEN];   /* transition name           */

275          BEGIN ( "p_transition" );

277          switch ( type )  {
278          case P_INIT:
279                  SPRINTF ( sym, "T_INIT" );
280                  break;

282          case P_FUNC:
283                  SPRINTF ( sym, "%s%s", symtab[st_idx].sym,
284                          HAS_A_VALUE (symtab[st_idx].sym_suffix[0])
285                          ? symtab[st_idx].sym_suffix : NULL );
286                  break;

288          case P_TERM:
289                  /* should be OUTPUT name! */
290                  SPRINTF ( sym, "T_TERM_%s", symtab[st_idx].sym );
291                  break;

293          default:
294                  debug ( rname, D_ERROR, "type=%d", type );
295                  ERROR ( "internal error" );
296          }  /* switch */

298          if ( HAS_NO_VALUE (t_idx=tr_find(sym)) )  {
299                  if ( tr_idx < PLACES_SIZE-1 )  {
300                          t_idx = ++tr_idx;
301                  }  else  {
302                          ERROR ( "trans table overflow" );
303                  }

305                  /* put the symbol name and type in the trans table */
306                  STRCPY ( trans[t_idx].sym, sym );
307                  trans[t_idx].type = type;
308          }

310          if ( type NE P_FUNC )  {
311                  RETURN ( t_idx );
312          }
```

```
314         /* add the information found in the symbol & function tables */

316         f_idx = symtab[st_idx].sym_idx;

318         if ( HAS_A_VALUE (funtab[f_idx].nec_cond) )  {
319                 tr_in ( t_idx, p_place(funtab[f_idx].nec_cond) );
320         }

322         for ( i=0; i<IO_MAX; i++ )  {
323                 if ( HAS_A_VALUE (funtab[f_idx].in[i]) )  {
324                         tr_in ( t_idx, p_place(funtab[f_idx].in[i]) );
325                 }
326                 if ( HAS_A_VALUE (funtab[f_idx].out[i]) )  {
327                         tr_out ( t_idx, p_place(funtab[f_idx].out[i]) );
328                         /* establish TERMinal transition */
329                         /* for this output entity        */
330                         out_idx = p_transition ( funtab[f_idx].out[i], P_TERM );
331                         tr_in ( out_idx, p_place(funtab[f_idx].out[i]) );
332                 }
333         }

335         if ( HAS_A_VALUE (funtab[f_idx].mode[0]) )  {
336                 tr_in ( t_idx, p_place(funtab[f_idx].mode[0]) );
337         }

339         if ( HAS_A_VALUE (funtab[f_idx].nxt_mode) AND
340              HAS_NO_VALUE (funtab[f_idx].stim[0]) )  {
341                 tr_out ( t_idx, p_place(funtab[f_idx].nxt_mode) );
342         }  else  {
343                 tr_out ( t_idx, p_place(funtab[f_idx].mode[0]) );
344         }

346         if ( HAS_A_VALUE (symtab[st_idx].comment[0]) )  {
347                 STRCPY ( trans[t_idx].comment, symtab[st_idx].comment );
348         }

350         if ( HAS_A_VALUE (funtab[f_idx].stim[0]) )  {
351                 STRCPY ( trans[t_idx].stim, funtab[f_idx].stim );
352                 trans[t_idx].stim_nxt_mode = p_place(funtab[f_idx].nxt_mode);
353         }

355         if ( HAS_A_VALUE (funtab[f_idx].note[0]) )  {
356                 STRCPY ( trans[t_idx].note, funtab[f_idx].note );
357         }

359         if ( HAS_A_VALUE (funtab[f_idx].warn[0]) )  {
360                 STRCPY ( trans[t_idx].warn, funtab[f_idx].warn );
361         }

363         RETURN ( t_idx );

365 }  /* p_transition () */
```

```
368  /*
369   * pl_find (), tr_find ()
370   *
371   * return the index of a symbol in the places or transitions table
372   *
373   * INPUT:
374   *       called by:              p_place(), p_transition()
375   *       input parameters:       symbol name
376   *
377   * OUTPUT:
378   *       return type & value:    NUM idx: index of sym in places or trans
379   *                                        (NO_VALUE if not found)
380   *
381   * GLOBAL:
382   *       variables used:         pl_idx, places[], tr_idx, trans[]
383   *       variables changed:      n/a
384   */

386   NUM
387  pl_find ( sym )
388   char * sym;
389  {
390          NUM idx;

392          BEGIN ( "pl_find" );

394          for ( idx=0; idx<=pl_idx; idx++ )  {
395                  if ( eqstr ( sym, places[idx].sym ) )  {
396                          RETURN ( idx );
397                  }
398          }

400          RETURN ( NO_VALUE );

402  }  /* pl_find () */


406   NUM
407  tr_find ( sym )
408   char * sym;
409  {
410          NUM idx;

412          BEGIN ( "tr_find" );

414          for ( idx=0; idx<=tr_idx; idx++ )  {
415                  if ( eqstr ( sym, trans[idx].sym ) )  {
416                          RETURN ( idx );
417                  }
418          }

420          RETURN ( NO_VALUE );

422  }  /* tr_find () */
```

```
425  /*
426   * pl_out ()
427   *
428   * fill the values for output places for a place
429   *
430   * INPUT:
431   *       called by:              tr_in()
432   *       input parameters:       indices to places[] and trans[]
433   *
434   * OUTPUT:
435   *       return type & value:    VOID
436   *
437   * GLOBAL:
438   *       variables used:         n/a
439   *       variables changed:      places[].out[]
440   */

442   VOID
443  pl_out ( p_idx, t_idx )
444   NUM p_idx;
445   NUM t_idx;
446  {
447           NUM i;

449           BEGIN ( "pl_out" );

451           for ( i=0; i<OUT_MAX; i++ )  {
452                   if ( places[p_idx].out[i] EQ t_idx )  {
453                           RETURN;          /* already an output place */
454                   }

456                   if ( HAS_NO_VALUE (places[p_idx].out[i]) )  {
457                           places[p_idx].out[i] = t_idx;
458                           break;
459                   }

461                   if ( i EQ OUT_MAX-1 )  {
462                           /* all of the output pointers consumed */
463                           ERROR ( "max # of output indices = 25" );
464                   }
465           }

467           RETURN;

469  }  /* pl_out () */
```

```
471  /*
472   * tr_in ()
473   *
474   * fill the values for input places for a transition
475   *
476   * INPUT:
477   *      called by:              p_init(), p_transition()
478   *      input parameters:       indices to trans[] and places[]
479   *
480   * OUTPUT:
481   *      return type & value:    VOID
482   *
483   * GLOBAL:
484   *      variables used:         n/a
485   *      variables changed:      trans[].in[]
486   */

488  VOID
489 tr_in ( t_idx, p_idx )
490  NUM t_idx;
491  NUM p_idx;
492 {
493         VOID pl_out();

495         NUM  i;

497         BEGIN ( "tr_in" );

499         for ( i=0; i<IN_MAX; i++ )  {
500                 if ( trans[t_idx].in[i] EQ p_idx )  {
501                         RETURN;         /* already an input place */
502                 }

504                 if ( HAS_NO_VALUE (trans[t_idx].in[i]) )  {
505                         trans[t_idx].in[i] = p_idx;
506                         break;
507                 }

509                 if ( i EQ IN_MAX-1 )  {
510                         /* all of the input pointers consumed */
511                         ERROR ( "max # of input indices = 25" );
512                 }
513         }

515         /* establish double linkage */
516         pl_out ( p_idx, t_idx );

518         RETURN;

520 }  /* tr_in () */
```

```
522  /*
523   * tr_out ()
524   *
525   * fill the values for output places for a transition
526   *
527   * INPUT:
528   *       called by:              p_init(), p_transition()
529   *       input parameters:       indices to trans[] and places[]
530   *
531   * OUTPUT:
532   *       return type & value:    VOID
533   *
534   * GLOBAL:
535   *       variables used:         n/a
536   *       variables changed:      trans[].out[]
537   */

539   VOID
540  tr_out ( t_idx, p_idx )
541   NUM t_idx;
542   NUM p_idx;
543  {
544          NUM i;

546          BEGIN ( "tr_out" );

548          for ( i=0; i<OUT_MAX; i++ )  {
549                  if ( trans[t_idx].out[i] EQ p_idx )  {
550                          RETURN;          /* already an output place */
551                  }

553                  if ( HAS_NO_VALUE (trans[t_idx].out[i]) )  {
554                          trans[t_idx].out[i] = p_idx;
555                          break;
556                  }

558                  if ( i EQ OUT_MAX-1 )  {
559                          /* all of the output pointers consumed */
560                          ERROR ( "max # of output indices = 25" );
561                  }
562          }

564          RETURN;

566  }  /* tr_out () */
```

```
569  /*
570   * pl_add_pag (), pl_del_bag ()
571   *
572   * add or delete tokens from the bag of a place
573   *
574   * INPUT:
575   *       called by:                INTERPRET module
576   *
577   * OUTPUT:
578   *       return type & value:      VOID
579   *
580   * GLOBAL:
581   *       variables used:           n/a
582   *       variables changed:        places[].bag
583   */

585   VOID
586  pl_add_bag ( p_idx )
587   NUM p_idx;
588  {
589           BEGIN ( "pl_add_bag" );

591           places[p_idx].bag++;

593           RETURN;

595  }  /* pl_add_bag () */



599   VOID
600  pl_del_bag ( p_idx )
601   NUM p_idx;
602  {
603           BEGIN ( "pl_del_bag" );

605           if ( places[p_idx].bag )  {
606                   places[p_idx].bag--;
607           }

609           RETURN;

611  }  /* pl_del_bag () */
```

```
613  /*
614   * pnl_init ()
615   *
616   * initialize the places[] and trans[] tables.
617   *
618   * INPUT:
619   *      called by:                  init()
620   *
621   * OUTPUT:
622   *      return type & value:        VOID
623   *
624   * GLOBAL:
625   *      variables used:             n/a
626   *      variables changed:          places[], trans[], pl_idx, tr_idx
627   */

629   VOID
630  pnl_init ()
631  {
632          NUM i, j;

634          BEGIN ( "pnl_init" );

636          /* initialize the places table */
637          for ( i=0; i<PLACES_SIZE; i++ )  {
638                  places[i].sym[0]          = NO_VALUE;
639                  places[i].bag             = NO_VALUE;
640                  for ( j=0; j<OUT_MAX; j++ )  {
641                          places[i].out[j] = NO_VALUE;
642                  places[i].comment[0]      = NO_VALUE;
643                  }
644          }

646          /* initialize the transitions table */
647          for ( i=0; i<TRANS_SIZE; i++ )  {
648                  trans[i].sym[0]           = NO_VALUE;
649                  trans[i].type             = NO_VALUE;
650                  for ( j=0; j<IN_MAX; j++ )  {
651                          trans[i].in[j]    = NO_VALUE;
652                  }
653                  for ( j=0; j<OUT_MAX; j++ )  {
654                          trans[i].out[j]   = NO_VALUE;
655                  trans[i].comment[0]       = NO_VALUE;
656                  trans[i].stim[0]          = NO_VALUE;
657                  trans[i].stim_nxt_mode    = NO_VALUE;
658                  trans[i].note[0]          = NO_VALUE;
659                  trans[i].warn[0]          = NO_VALUE;
660                  }
661          }

663          pl_idx = tr_idx = NO_VALUE;
664          RETURN;

666  }  /* pnl_init () */
```

```
/*************************************************************
 *
 * MODULE:       INTERPRET
 *
 * Interpreter of the Entity Graph (a Petri Net Structure).
 *
 * The interpretation provides a simulation of the ERA prototype.
 *
 * PROJECT:      Requirements Analysis using Petri Nets
 *               CMPSC 690 - Master's Project, Summer 1984
 *
 * AUTHOR:       Brad C. Gaylord
 *               AT&T Information System Laboratories
 *               11900 North Pecos, Denver CO,  80234
 *               drux2!bcg, DR 31L60,  (303)-538-1413
 *
 *************************************************************
 *
 * USED BY:      main
 *
 * ACCESS FCTS: interpret()
 *
 *************************************************************
 *
 * USES:         TABLES
 *
 * DATA OWNED:   n/a
 *
 *************************************************************
 */


/* INCLUDE files required by this module: */
#include "T_tables.i"                   /* general table definitions  */
#include "T_symtab.i"                   /* symbol table               */
#include "T_funtab.i"                   /* function table             */
#include "T_pnltab.i"                   /* PNL tables                 */


#undef   D_MODULE                       /* was defined in T_tables.i  */
#define  D_MODULE       D_INTERPRET     /* primary module trace level */
#include "debug.h"                      /* debug levels               */


/* DEFINEs local to this module: */
#define STARVATION      3               /* Starvation threshold       */
#define MAX_MENU        10              /* max # of transtions in menu*/
#define NOT_OK          0               /* Almost_enabled is not ok   */
#define OK              1               /* Almost_enabled is ok       */

extern NUM Pnl_place;

/* global variables "local" to this module: */
NUM Almost_enabled;
NUM Tracing       = NO;
NUM Starve_defeat = NO;
NUM hungry        = NULL;
NUM last_trans    = NO_VALUE;

NUM menu[MAX_MENU];
NUM menu_idx;
```

```
 65  /*
 66   * interpret ()
 67   *
 68   * interpret the entity graph:
 69   *      initialize the net
 70   *      loop forever, firing any firable transitions
 71   *
 72   * INPUT:
 73   *      called by:              main()
 74   *
 75   * OUTPUT:
 76   *      return type & value:    VOID
 77   *
 78   * GLOBAL:
 79   *      variables used:         n/a
 80   *      variables changed:      n/a
 81   */

 83  VOID
 84  interpret ()
 85  {
 86          VOID   net_init();
 87          VOID   net_process(), net_dump(), net_options(), net_statistics();
 88          char * gets();

 90          char * s_count;
 91          char   line [ LINE_LEN ];

 93          BEGIN ( "interpret" );

 95          /* initialize the net */
 96          net_init ();

 98          /* command line processing */
 99          PRINTF ( "Enter '?' for a Help message\n" );

101          for EVER  {
102                  PRINTF ( "> " );                /* prompt */
103                  s_count = gets ( line );

105                  if ( (s_count EQ NULL) OR eqstr ( line, "q" ) )  {
106                          PRINTF ( "Goodbye.\n" );
107                          break;
108                  }  else if ( eqstr ( line, "" ) )  {
109                          net_process ();
110                  }  else if ( eqstr ( line, "o" ) )  {
111                          net_options ();
112                  }  else if ( eqstr ( line, "d" ) )  {
113                          net_dump ();
114                  }  else if ( eqstr ( line, "?" ) )  {
115                          PRINTF ( "Enter one of the following:\n" );
116                          PRINTF ( "\tq    - quit this session\n" );
117                          PRINTF ( "\t<cr> - fire a transition\n" );
118                          PRINTF ( "\to    - change Interpreter options\n" );
119                          PRINTF ( "\td    - dump the Petri net\n" );
120                          PRINTF ( "\t?    - print this help message\n" );
121                  }  else  {
122                          PRINTF ( "Enter '?' for a Help message\n" );
123                  }
124          }

126          net_statistics();

128          RETURN;

130  }  /* interpret () */
```

```
132  /*
133   * net_process ()
134   *
135   * do the menu processing
136   *
137   * INPUT:
138   *       called by:              interpret()
139   *
140   * OUTPUT:
141   *       return type & value:    VOID
142   *
143   * GLOBAL:
144   *       variables used:         Tracing, Almost_enabled
145   *       variables changed:      n/a
146   */

148   VOID
149  net_process ()
150  {
151          VOID make_menu(), menu_choice(), fire();
152          NUM  tran_enabled();

154          NUM  tmp_trace;
155          NUM  i;

157          BEGIN ( "net_process" );

159          /* look for enabled, or Almost_enabled transitions */
160          make_menu ();

162          if ( Tracing OR Almost_enabled )  {
163                  menu_choice ();
164          }  else  {
165                  fire ( menu[0] );
166          }

168          Almost_enabled = NOT_OK;

170          /* fire the terminal transitions */
171          PRINTF ( "FIRING TERMinal transitions:\n" );
172          for ( i=0; i<=tr_idx; i++ )  {
173                  if ( tran_enabled ( i, P_TERM ) )  {
174                          /* TERMinal transitions are always printed */
175                          tmp_trace = Tracing;
176                          Tracing = YES;
177                          fire ( i );
178                          Tracing = tmp_trace;
179                  }
180          }

182          RETURN;

184  }  /* net_process () */
```

```
187  /*
188   * make_menu ()
189   *
190   * fill the menu array with firable transitions
191   *
192   * INPUT:
193   *       called by:              net_process()
194   *
195   * OUTPUT:
196   *       return type & value:    VOID
197   *
198   * GLOBAL:
199   *       variables used:         trans[], tr_idx
200   *       variables changed:      menu[]
201   */

203   VOID
204  make_menu ()
205  {
206          VOID net_dump();
207          BOOL tran_enabled();

209          NUM i;

211          BEGIN ( "make_menu" );

213          /* clear menu array */
214          for ( i=0; i<MAX_MENU; i++ )  {
215                  menu[i] = NO_VALUE;
216          }
217          menu_idx = NO_VALUE;

219          Almost_enabled = NOT_OK;

221          /* look for enabled transitions */
222          for ( i=0; i<=tr_idx; i++ )  {
223                  if ( tran_enabled ( i, P_FUNC )  )  {
224                          /* add transition to menu */
225                          if ( ++menu_idx < MAX_MENU )  {
226                                  menu[menu_idx] = i;
227                          }  else  {
228                                  ERROR ( "menu table overflow" );
229                          }
230                  }
231          }

233          if ( HAS_A_VALUE(menu_idx) AND NOT Starve_defeat )  {
234                  RETURN;
235          }

237          Almost_enabled = OK;

239          /* look for almost enabled transitions */
240          for ( i=0; i<=tr_idx; i++ )  {
241                  if ( tran_enabled ( i, P_FUNC )  )  {
242                          /* add transition to menu */
243                          if ( ++menu_idx < MAX_MENU )  {
244                                  menu[menu_idx] = i;
245                          }  else  {
246                                  ERROR ( "menu table overflow" );
247                          }
248                  }
249          }

251          if ( HAS_A_VALUE(menu_idx) )  {
252                  RETURN;
253          }

255          /* no menu?  deadlock? */
256          PRINTF ( "WARNING: POTENTIAL DEADLOCK" );
257          net_dump ();

259          RETURN;

261  }  /* make_menu () */
```

```
264  /*
265   * menu_choice ()
266   *
267   * print the menu, so the user can choose a transition.
268   *
269   * INPUT:
270   *       called by:              net_process()
271   *
272   * OUTPUT:
273   *       return type & value:    VOID
274   *
275   * GLOBAL:
276   *       variables used:         menu[]
277   *       variables changed:      n/a
278   */

280  VOID
281  menu_choice ()
282  {
283          VOID tran_print();

285          NUM i, choice;
286          char line [LINE_LEN];

288          BEGIN ( "menu_choice" );

290          i = NO_VALUE;

292          for ( i=0; i<=menu_idx; i++ )  {
293                  PRINTF ( "%d: ", i );
294                  tran_print ( menu[i] );
295          }

297          PRINTF ( "\nEnter the number of the transition to be fired: " );
298          while ( gets ( line ) )  {
299                  if ( isdigit ( line[0] ) )  {
300                          choice = atoi ( line );
301                          if ( choice >= 0  AND  choice <= menu_idx )  {
302                                  PRINTF ( "FIRING menu item # %d:\n", choice );
303                                  fire ( menu[choice] );
304                                  PRINTF ( "\n" );
305                                  RETURN;
306                          }
307                  }
308                  PRINTF ( "try again: " );
309          }

311          PRINTF ( "EOF\n" );
312          RETURN;

314  }  /* menu_choice () */
```

```
316  /*
317   * fire ()
318   *
319   * fire a transition
320   *
321   * INPUT:
322   *       called by:              net_process()
323   *
324   * OUTPUT:
325   *       return type & value:    VOID
326   *
327   * GLOBAL:
328   *       variables used:         n/a
329   *       variables changed:      hungry, last_trans, places[].bag
330   */

332   VOID
333  fire ( t_idx )
334   NUM t_idx;                                  /* index to trans[] table */
335  {
336          NUM  tran_enabled();
337          VOID pl_add_bag(), pl_del_bag();

339          NUM  i, p_idx;

341          BEGIN ( "fire" );

343          /* detect starvation of net */
344          if ( trans[t_idx].type EQ P_FUNC )  {
345                  if ( last_trans EQ t_idx )  {
346                          if ( ++hungry EQ STARVATION )  {
347                                  PRINTF ( "*************************\n");
348                                  PRINTF ( "Potential Net Starvation!\n");
349                                  PRINTF ( "Turn Starvation Defeat ON\n");
350                                  PRINTF ( "*************************\n");
351                          }
352                  }  else  {
353                          last_trans = t_idx;
354                          hungry = NULL;
355                  }
356          }
357          Almost_enabled = NOT_OK;

359          if ( NOT tran_enabled ( t_idx ) )  {
360                  /* almost enabled - we must supply tokens */
361                  for ( i=0; i<IN_MAX; i++ )  {
362                          p_idx = trans[t_idx].in[i];
363                          if ( HAS_NO_VALUE (p_idx) )  {
364                                  break;  /* end of input list */
365                          }
366                          if ( NOT places[p_idx].bag AND
367                                   places[p_idx].sym[0] EQ '$' )  {
368                                  pl_add_bag ( p_idx );
369                          }
370                  }
371          }

373          /* remove tokens from the input places */
374          for ( i=0; i<IN_MAX; i++ )  {
375                  p_idx = trans[t_idx].in[i];
376                  if ( HAS_NO_VALUE (p_idx) )  {
377                          break;  /* end of input list */
378                  }
379                  pl_del_bag ( p_idx );
380          }

382          /* add tokens to the output places */
383          for ( i=0; i<OUT_MAX; i++ )  {
384                  p_idx = trans[t_idx].out[i];
385                  if ( HAS_NO_VALUE (p_idx) )  {
386                          break;  /* end of input list */
387                  }
388                  pl_add_bag ( p_idx );
389          }

391          /* stim and stim_nxt_mode!!! */
```

```
393             if ( Tracing )  {
394                     tran_print ( t_idx );
395             }

397             RETURN;

399     }  /* fire () */
```

```
401  /*
402   * net_dump ()
403   *
404   * dump the Petri net
405   *
406   * INPUT:
407   *       called by:                interpret()
408   *
409   * OUTPUT:
410   *       return type & value:      VOID
411   *
412   * GLOBAL:
413   *       variables used:           places[], trans[]
414   *       variables changed:        n/a
415   */

417   VOID
418  net_dump ()
419  {
420          BOOL tran_enabled();
421          VOID tran_print();

423          NUM  i;

425          BEGIN ( "net_dump" );

427          PRINTF ( "Dumping the Petri Net\n" );

429          /* print the places with tokens in their bag */
430          PRINTF ( "bag PLACE\n" );
431          for ( i=0; i<=pl_idx; i++ )  {
432                  if ( NOT places[i].bag )  {
433                          continue;
434                  }
435                  PRINTF ( "%2d   %s\n", places[i].bag, places[i].sym );
436          }

438          Almost_enabled = NOT_OK;

440          PRINTF ( "enabled Function TRANSitions: " );
441          for ( i=0; i<=tr_idx; i++ )  {
442                  if ( tran_enabled ( i, P_FUNC ) )  {
443                          tran_print ( i );
444                  }
445          }
446          PRINTF ( "\n" );

448          PRINTF ( "enabled TERMinal transitions: " );
449          for ( i=0; i<=tr_idx; i++ )  {
450                  if ( tran_enabled ( i, P_TERM ) )  {
451                          tran_print ( i );
452                  }
453          }
454          PRINTF ( "\n" );

456          Almost_enabled = OK;

458          PRINTF ( "almost enabled Function TRANSitions: " );
459          for ( i=0; i<=tr_idx; i++ )  {
460                  if ( tran_enabled ( i, P_FUNC ) )  {
461                          tran_print ( i );
462                  }
463          }
464          PRINTF ( "\n" );

466          Almost_enabled = NOT_OK;

468          RETURN;

470  }  /* net_dump () */
```

```
472  /*
473   * net_options ()
474   *
475   * change the options setting for the interpreter
476   *
477   * INPUT:
478   *      called by:              interpret()
479   *
480   * OUTPUT:
481   *      return type & value:    VOID
482   *
483   * GLOBAL:
484   *      variables used:         n/a
485   *      variables changed:      Starve_defeat, Tracing
486   */

488   VOID
489  net_options ()
490  {
491          char * gets();

493          char * s_count;
494          char   line [ LINE_LEN ];

496          BEGIN ( "net_options" );

498          for EVER  {
499                  PRINTF ( "Enter an option, or '?' for help: " );
500                  s_count = gets ( line );

502                  if ( s_count EQ NULL OR eqstr ( line, "q" ) )  {
503                          PRINTF ( "Returning to Interpreter\n" );
504                          break;
505                  }  else if ( eqstr ( line, "t" ) )  {
506                          if ( Tracing EQ YES)  {
507                                  PRINTF ( "Tracing turned OFF\n" );
508                                  Tracing = NO;
509                          }  else  {
510                                  PRINTF ( "Tracing turned ON\n" );
511                                  Tracing = YES;
512                          }
513                  }  else if ( eqstr ( line, "s" ) )  {
514                          if ( Starve_defeat EQ YES)  {
515                                  PRINTF ( "Starvation defeat OFF\n" );
516                                  Starve_defeat = NO;
517                          }  else  {
518                                  PRINTF ( "Starvation defeat ON\n" );
519                                  Starve_defeat = YES;
520                          }
521                  }  else if ( eqstr ( line, "?" ) )  {
522                          PRINTF ( "Enter one of the following:\n" );
523                          PRINTF ( "\tq - return to Interpreter\n" );
524                          PRINTF ( "\tt - trace on/off\n" );
525                          PRINTF ( "\ts - starvation defeat on/off\n" );
526                          PRINTF ( "\t? - print this help message\n" );
527                  }  else  {
528                          PRINTF ( "Enter '?' for a Help message\n" );
529                  }
530          }

532          RETURN;

534  }  /* net_options () */
```

```
537  /*
538   * net_statistics ()
539   *
540   * print a summary of transition firings and conservation of tokens
541   *
542   * INPUT:
543   *       called by:              interpret()
544   *
545   * OUTPUT:
546   *       return type & value:    VOID
547   *
548   * GLOBAL:
549   *       variables used:         trans[]
550   *       variables changed:      n/a
551   */

553   VOID
554  net_statistics ()
555  {
556           BEGIN ( "net_statistics" );

558           RETURN;

560  }  /* net_statistics () */
```

```
562  /*
563   * tran_enabled ()
564   *
565   * Is the specified transition enabled?
566   *
567   * INPUT:
568   *      called by:              net_dump(), etc.
569   *
570   * OUTPUT:
571   *      return type & value:    BOOL: YES or NO
572   *
573   * GLOBAL:
574   *      variables used:         places[], trans[]
575   *      variables changed:      n/a
576   */

578   BOOL
579  tran_enabled ( t_idx, typ )
580   NUM t_idx;                                  /* index to trans[] */
581   NUM typ;                                    /* transition type  */
582  {
583          NUM  i;
584          NUM  p_idx;

586          BEGIN ( "tran_enabled" );

588          for ( i=0; i<IN_MAX; i++ )  {
589                  if ( trans[t_idx].type NE typ )  {
590                          RETURN ( NO );
591                  }
592                  p_idx = trans[t_idx].in[i];
593                  if ( HAS_NO_VALUE (p_idx) )  {
594                          /* end of input list... success! */
595                          break;
596                  }
597                  if ( NOT places[p_idx].bag )  {
598                          if ( Almost_enabled )  {
599                                  if ( places[p_idx].sym[0] EQ '$' )  {
600                                          continue;
601                                  }
602                          }
603                          RETURN ( NO );
604                  }
605          }

607          RETURN ( YES );

609  }  /* tran_enabled () */
```

```
611  /*
612   * tran_print ()
613   *
614   * print one transition for a menu
615   *
616   * INPUT:
617   *       called by:              net_dump(), etc.
618   *
619   * OUTPUT:
620   *       return type & value:    VOID
621   *
622   * GLOBAL:
623   *       variables used:         trans[]
624   *       variables changed:      n/a
625   */

627   VOID
628  tran_print ( t_idx )
629   NUM t_idx;                                  /* index to trans[] */
630  {
631          VOID pr_input(), pr_output();

633          BEGIN ( "tran_print" );

635          Pnl_place = NO;

637          PRINTF ( "%s : ", trans[t_idx].sym );

639          switch ( trans[t_idx].type )  {
640          case P_INIT:
641                  /* PRINTF ( "INIT\n\t" ); */
642                  pr_output ( stdout, t_idx ):
643                  break;

645          case P_FUNC:
646                  /* PRINTF ( "TRANS\n" );
647                   * PRINTF ( "\t" );
648                   * pr_output ( stdout, t_idx );
649                   * PRINTF ( "\t" );
650                   * pr_input  ( stdout, t_idx );
651                   */

653                  if ( HAS_A_VALUE(trans[t_idx].stim[0]) )  {
654                          PRINTF ( "%s\n", trans[t_idx].stim );
655                  }
656                  if ( HAS_A_VALUE(trans[t_idx].note[0]) )  {
657                          PRINTF ( "%s\n", trans[t_idx].note );
658                  }
659                  break;

661          case P_TERM:
662                  /* PRINTF ( "TERM\n\t" ); */
663                  pr_input  ( stdout, t_idx );
664                  break;

666          default:
667                  debug ( rname, D_ERROR, "trans[t_idx].type=%d", trans[t_idx].typ
     e );
668                  ERROR ( "internal error" );
669          }  /* switch */

671          RETURN;

673  }  /* tran_print () */
```

```
675  /*
676   * net_init ()
677   *
678   * clear the net, and fire the initial transition
679   *
680   * INPUT:
681   *      called by:              interpret()
682   *
683   * OUTPUT:
684   *      return type & value:    VOID
685   *
686   * GLOBAL:
687   *      variables used:         trans[0]
688   *      variables changed:      places[].bag
689   */

691   VOID
692  net_init ()
693  {
694          VOID pl_add_bag();

696          NUM  i;

698          BEGIN ( "net_init" );

700          /* clear the net */
701          for ( i=0; i<PLACES_SIZE; i++ )  {
702                  places[i].bag = NULL;
703          }

705          /* fire the initial transition */
706          pl_add_bag ( trans[P_INIT].out[P_INIT] );

708          debug ( rname, D_TRACE, "place[%d].bag = %d",
709                  trans[P_INIT].out[P_INIT],
710                  places[trans[P_INIT].out[P_INIT]].bag );

712          RETURN;

714  }  /* net_init () */
```

```
1  /* This sub-module deals with symtab exclusively */

3  #include "T_tables.i"                    /* see for MODULE DESCRIPTION */

5  #define  extern                          /* T_sym1.c "owns" T_symtab.i */
6  #include "T_symtab.i"                    /* symbol table               */
```

```
 9  /*
10   * st_entity ()
11   *
12   * This routine is called when an ENTITY line is first encountered...
13   *
14   * add entity names to the symbol table based upon entity class
15   * and assign an index to the function table (for FUNCTIONs).
16   *
17   * current.idx is set to the symbol table entry for this entity.
18   *
19   * (NOTE that the entity name may already be in the table,
20   * if a forward reference has occurred in a previous relation.)
21   *
22   * if the entity is a FUNCTION, then funtab.p_or_a is assigned
23   * the value of PERIODIC or ACTIVITY.
24   *
25   * INPUT:
26   *      called by:              entity ()
27   *
28   * OUTPUT:
29   *      return type & value:    VOID
30   *
31   * GLOBAL:
32   *      variables used:         current
33   *      variables changed:      symtab[]
34   */

36  VOID
37  st_entity ( sym, class, f_or_i )
38  char * sym;                                 /* entity name                  */
39  NUM class;                                  /* entity class                 */
40  NUM f_or_i;                                 /* T_FUNCTION or T_I_O_DATA     */
41  {
42          VOID st_current(), is_fun_name(), is_iod_name();
43          VOID st_idx(), st_class(), st_p_or_a();
44          NUM  st_sym(), st_fun();

46          BEGIN ( "st_entity" );

48          st_current ( st_sym (sym), f_or_i );

50          switch ( f_or_i )  {
51          case T_FUNCTION:
52                  is_fun_name ( rname, sym );
53                  debug ( rname, D_TRACE, "FUNCTION entity: %s", sym );
54                  st_idx ( st_fun () );
55                  st_class ( current.idx, class );
56                  st_p_or_a ( class );
57                  break;

59          case T_I_O_DATA:
60                  /* Input_output causes annotation... */
61                  is_iod_name ( rname, sym );
62                  debug ( rname, D_TRACE, "I_O_DATA entity: %s", sym );
63                  st_class ( current.idx, class );
64                  break;

66          default:
67                  debug ( rname, D_ERROR, "class=%d", class );
68                  ERROR ( "internal error" );
69          }  /* switch */

71          RETURN;

73  }  /* st_entity () */
```

```
 75  /*
 76   * st_sym ()
 77   *
 78   * add a symbol name to the symbol table if it is not already there.
 79   *
 80   * return the index to the symbol's entry in the symbol table.
 81   *
 82   * INPUT:
 83   *      called by:              st_entity(), T_funtab.c, T_symfun.c
 84   *      input parameters:       sym name
 85   *
 86   * OUTPUT:
 87   *      return type & value:    current symtab index
 88   *
 89   * GLOBAL:
 90   *      variables used:         n/a
 91   *      variables changed:      sym_idx, symtab[]
 92   */

 94  NUM
 95 st_sym ( sym )
 96  char * sym;
 97 {
 98          NUM st_find();

100          NUM idx;

102          BEGIN ( "st_sym" );

104          if ( HAS_NO_VALUE (idx=st_find(sym)) OR Cloning )  {
105                  /* entity is not in symtab: get next index */
106                  if ( sym_idx < SYMTAB_SIZE-1 )  {
107                          idx = ++sym_idx;
108                  }  else  {
109                          ERROR ( "symbol table overflow" );
110                  }

112                  /* put the symbol name in the symbol table */
113                  STRCPY ( symtab[idx].sym, sym );
114          }

116          RETURN ( idx );

118 }  /* st_sym () */
```

```
121  /*
122   * st_suffix ()
123   *
124   * store a symbol suffix.
125   *
126   * the suffix is a Mode name which is being appended to distinguish
127   * clones which come from multiple required_modes in an entity.
128   *
129   * INPUT:
130   *       called by:              st_mult_modes(), st_clone()
131   *
132   * OUTPUT:
133   *       return type & value:    VOID
134   *
135   * GLOBAL:
136   *       variables used:         current.idx
137   *       variables changed:      symtab[].sym_suffix
138   */

140   VOID
141  st_suffix ( mode )
142   NUM mode;                                  /* symbol name suffix */
143  {
144          BEGIN ( "st_suffix" );

146          if ( HAS_NO_VALUE (symtab[current.idx].sym_suffix[0]) )  {
147                  SPRINTF ( symtab[current.idx].sym_suffix, "_%s",
148                          symtab[mode].sym );
149          }  else  {
150                  ERROR ( "max # of symbol suffixes = 1" );
151          }

153          RETURN;

155  }  /* st_suffix () */
```

```
157  /*
158   * st_class ()
159   *
160   * store the class in the symbol table
161   *
162   * INPUT:
163   *       called by:              st_entity(), st_orphan(), st_Mode_list()
164   *
165   * OUTPUT:
166   *       return type & value:    VOID
167   *
168   * GLOBAL:
169   *       variables used:         n/a
170   *       variables changed:      symtab[].sym_class
171   */

173   VOID
174  st_class ( s_idx, class )
175   NUM s_idx;                              /* index to symtab[] */
176   NUM class;                              /* type of entity    */
177  {
178          BEGIN ( "st_class" );

180          if ( HAS_NO_VALUE (symtab[s_idx].sym_class) )  {
181                  symtab[s_idx].sym_class = class;
182          }  else if ( symtab[s_idx].sym_class NE class )  {
183                  ERROR ( "max # of sym_class values = 1" );
184          }

186          RETURN;

188  }  /* st_class () */
```

```
191  /*
192   * st_idx ()
193   *
194   * store the function table index
195   *
196   * INPUT:
197   *       called by:              st_entity(), etc
198   *
199   * OUTPUT:
200   *       return type & value:    VOID
201   *
202   * GLOBAL:
203   *       variables used:         current.idx
204   *       variables changed:      symtab[].sym_idx
205   */

207   VOID
208  st_idx ( s_idx )
209   NUM s_idx;                                  /* index to symtab[] */
210  {
211           BEGIN ( "st_idx" );

213           if ( HAS_NO_VALUE (symtab[current.idx].sym_idx) )  {
214                symtab[current.idx].sym_idx = s_idx;
215           }  else  {
216                ERROR ( "max # of sym_idx's = 1" );
217           }

219           RETURN;

221  }  /* st_idx () */
```

```
223  /*
224   * st_comment ()
225   *
226   * fill a PNL "comment" line
227   *
228   * INPUT:
229   *       called by:              TABLES, TRANSLATE
230   *       input parameters:       str     - comment string
231   *
232   * OUTPUT:
233   *       return type & value:    VOID
234   *
235   * GLOBAL:
236   *       variables used:         current.idx
237   *       variables changed:      symtab[].comment
238   */

240   VOID
241  st_comment ( str )
242   char * str;
243  {
244           BEGIN ( "st_comment" );

246           if ( HAS_NO_VALUE (symtab[current.idx].comment[0]) )  {
247                SPRINTF ( symtab[current.idx].comment, "# %s:", str );
248           }  else  {
249                ERROR ( "Max # of comment strings = 1" );
250           }

252           RETURN;

254  }  /* st_comment () */
```

```
256  /*
257   * st_find ()
258   *
259   * return the index of a symbol in the symbol table
260   *
261   * INPUT:
262   *       called by:              st_sym()
263   *       input parameters:       symbol name
264   *
265   * OUTPUT:
266   *       return type & value:    NUM idx;  index of sym in symtab
267   *                                         (NO_VALUE if not found)
268   *
269   * GLOBAL:
270   *       variables used:         symtab[].sym
271   *       variables changed:      none
272   */

274   NUM
275  st_find ( sym )
276   char * sym;
277  {
278       NUM idx;

280       BEGIN ( "st_find" );

282           for ( idx=0; idx<=sym_idx; idx++ )  {
283                   if ( eqstr ( sym, symtab[idx].sym ) )  {
284                           RETURN ( idx );
285                   }
286           }

288           RETURN ( NO_VALUE );

290  }  /* st_find () */
```

```
292  /*
293   * st_verify ()
294   *
295   * verify that a mode found in the first part of the MODE_TABLE
296   * has been referenced by some entity, and is not already in the
297   * Mode linked-list
298   *
299   * INPUT:
300   *       called by:              mode_table()
301   *       input parameters:       mode - mode to be verified
302   *
303   * OUTPUT:
304   *       return type & value:    VOID
305   *
306   * GLOBAL:
307   *       variables used:         symtab[]
308   *       variables changed:      symtab[]
309   */

311   VOID
312  st_verify ( mode )
313   char * mode;
314  {
315         VOID is_mode_name(), st_current(), st_class(), st_idx();
316         NUM  st_find(), st_Mode_list();

318         NUM  mod_idx;                   /* symtab idx to 'mode' */
319         NUM  idx;                       /* iterator */

321         BEGIN ( "st_verify" );

323         /* is mode correct (lexically)? */
324         is_mode_name ( rname, mode );

326         /* is 'mode' in the symbol table? */
327         if ( HAS_NO_VALUE ( mod_idx = st_find ( mode ) ) )  {
328                 /* add 'mode' to symtab */
329                 mod_idx = st_sym ( mode );
330                 st_current ( mod_idx, T_MODE );
331                 st_comment ( "No entity directly references this mode" );
332         }

334         /* give this symbol some class */
335         st_class ( mod_idx, T_MODE );

337         /* is this mode already in the Mode_list? */
338         idx = st_Mode_list ();
339         while ( HAS_A_VALUE(symtab[idx].sym_idx) )  {
340                 idx = symtab[idx].sym_idx;
341                 if ( idx EQ mod_idx )  {
342                         ERROR ( "Attempt to add duplicate to Mode_list" );
343                 }
344         }

346         /* add the index to 'mode' to the end of the list */
347         st_current ( idx, T_MODE );
348         st_idx ( mod_idx );

350         RETURN;

352  }  /* st_verify () */
```

```
355  /*
356   * st_Mode_list ()
357   *
358   * return an index to the beginning of the Mode linked list.
359   *
360   * INPUT:
361   *       called by:              mode_table()
362   *       input parameters:       n/a
363   *
364   * OUTPUT:
365   *       return type & value:    Mode_list - symtab index to MODE_LIST
366   *
367   * GLOBAL:
368   *       variables used:         MODE_LIST (#define)
369   *       variables changed:      symtab[]
370   */

372   NUM
373  st_Mode_list ()
374  {
375           NUM    st_find(), st_sym();
376           VOID   st_class();

378           NUM    Mode_list;

380           BEGIN ( "st_Mode_list" );

382           /* establish the mode list;  does MODE_LIST exist? */
383           if ( HAS_NO_VALUE ( Mode_list = st_find ( MODE_LIST ) ) )  {
384                   /* no;  create the MODE_LIST list pointer */
385                   Mode_list = st_sym ( MODE_LIST );
386                   st_class ( Mode_list, T_MODE );
387           }

389           RETURN ( Mode_list );

391  }  /* st_Mode_list () */
```

```
393  /*
394   * st_i_mode ()
395   *
396   * store the Initial_Mode
397   *
398   * INPUT:
399   *       called by:              mode_table()
400   *       input parameters:       mode - Initial_Mode
401   *
402   * OUTPUT:
403   *       return type & value:    VOID
404   *
405   * GLOBAL:
406   *       variables used:         n/a
407   *       variables changed:      Init_Mode, symtab[]
408   */

410   VOID
411  st_i_mode ( mode )
412   char * mode;
413  {
414          VOID is_mode_name();
415          NUM  st_find(), st_Mode_list();
416          NUM idx;

418          BEGIN ( "st_i_mode" );

420          /* is mode correct (lexically)? */
421          is_mode_name ( rname, mode );

423          /* has mode been referenced by any entity?  is it in symtab? */
424          if ( HAS_NO_VALUE ( Init_Mode = st_find ( mode ) ) )  {
425                  /* no.  Some entity should use this as an input place */
426                  ERROR ( "Initial_Mode isn't referenced by any entity" );
427          }

429          debug ( rname, D_TRACE, "Init_Mode=%s", symtab[Init_Mode].sym );

431          /* is Init_Mode in the Mode_list? */
432          idx = st_Mode_list ();
433          while ( HAS_A_VALUE(symtab[idx].sym_idx) )  {
434                  idx = symtab[idx].sym_idx;
435                  if ( idx EQ Init_Mode )  {
436                          RETURN;
437                  }
438          }

440          ERROR ( "Initial_Mode isn't in the Mode_list" );

442  }  /* st_i_mode () */
```

```
444  /*
445   * ft_idx (), ent_type ()
446   *
447   * return the function table index and the entity type, respectively
448   *
449   * INPUT:
450   *       called by:              FUNCTION sub-module
451   *       input parameters:       n/a
452   *
453   * OUTPUT:
454   *       return type & value:    ft_idx()   - function table index
455   *                               ent_type() - T_FUNCTION, T_I_O_DATA, etc
456   *
457   * GLOBAL:
458   *       variables used:         symtab[], current
459   *       variables changed:      n/a
460   */

462   NUM
463  ft_idx ()
464  {
465          return ( symtab[current.idx].sym_idx );

467  }  /* ft_idx () */


471   NUM
472  ent_type ()
473  {
474          return ( current.type );

476  }  /* ent_type () */
```

```
478  /*
479   * del_sym ()
480   *
481   * delete entries from the symbol table
482   *
483   * INPUT:
484   *      called by:              st_init(), st_mult_modes(), st_invokes()
485   *      input parameters:       idx     - table index
486   *
487   * OUTPUT:
488   *      return type & value:    VOID
489   *
490   * GLOBAL:
491   *      variables used:         n/a
492   *      variables changed:      symtab[]
493   */

495   VOID
496  del_sym ( idx )
497   NUM idx;
498  {
499          VOID    del_fun();
500          NUM     is_f_or_i();

502          if ( is_f_or_i ( symtab[idx].sym_class ) EQ T_FUNCTION )  {
503                  del_fun ( symtab[idx].sym_idx );
504          }

506          symtab[idx].sym [0]         = NO_VALUE;
507          symtab[idx].sym_suffix [0]  = NO_VALUE;
508          symtab[idx].sym_class       = NO_VALUE;
509          symtab[idx].comment [0]     = NO_VALUE;
510          symtab[idx].sym_idx         = NO_VALUE;

512  }  /* del_sym () */
```

```
1  #include "T_tables.i"                    /* see for MODULE DESCRIPTION */
2  #include "T_symtab.i"                    /* symbol table               */
3  #include "T_funtab.i"                    /* function table             */
```

```
 6  /*
 7   * st_media ()
 8   *
 9   * an array of media names is stored in the "media" table.
10   * for IO_DATA entities, an index into the "media" table is stored
11   * in the sym_idx field of the symbol table entry for the entity.
12   *
13   * INPUT:
14   *      called by:              relation()
15   *      input parameters:       rhs - value of 'media' keyword
16   *
17   * OUTPUT:
18   *      return type & value:    VOID
19   *
20   * GLOBAL:
21   *      variables used:         media[]
22   *      variables changed:      symtab[current.idx].sym_idx
23   */

25  VOID
26  st_media ( rhs )
27   char * rhs;
28  {
29          NUM idx;

31          BEGIN ( "st_media" );

33          if ( current.type NE T_I_O_DATA )  {
34                  ERROR ( "media: only valid for I_O_DATAs" );
35          }

37          for ( idx=0; idx < MEDIA_MAX; idx++ )  {
38                  if ( eqstr ( rhs, media[idx] ) )  {
39                          if ( HAS_NO_VALUE (symtab[current.idx].sym_idx) )  {
40                                  symtab[current.idx].sym_idx = idx;
41                          }  else  {
42                                  ERROR ( "max # of media values = 1" );
43                          }
44                          RETURN;
45                  }
46          }

48          ERROR ( "media not found in media table" );

50  }  /* st_media () */
```

```
 52  /*
 53   * st_mult_modes ()
 54   *
 55   * Clone any entities which have more than one mode
 56   * or a mode of "every_mode"
 57   *
 58   * INPUT:
 59   *      called by:              mode_table()
 60   *
 61   * OUTPUT:
 62   *      return type & value:    VOID
 63   *
 64   * GLOBAL:
 65   *      variables used:         symtab[], funtab[]
 66   *      variables changed:      symtab[], funtab[]
 67   */

 69   VOID
 70  st_mult_modes ()
 71  {
 72      VOID del_sym(), st_clone(), st_current(), st_suffix();
 73      NUM  is_f_or_i(), ft_idx(), st_Mode_list();

 75      NUM  s_idx;
 76      NUM  f_idx;
 77      NUM  m_idx;
 78      NUM  clone;

 80      BEGIN ( "st_mult_modes" );

 82      /* examine each of the entities;  look for multiple mode functions */
 83      for ( s_idx=0; s_idx<=sym_idx; s_idx++ )  {
 84          if ( is_f_or_i(symtab[s_idx].sym_class) NE T_FUNCTION )  {
 85              continue;
 86          }

 88          st_current ( s_idx, T_FUNCTION );
 89          f_idx = ft_idx ();
 90          clone = NO;

 92          /* caveat: code assumes only mode[0] can EQ "every_mode" */
 93          if ( funtab[f_idx].mode[0] EQ (m_idx=st_Mode_list()) )  {
 94              /* duplicate this entity for "every_mode" */
 95              while( HAS_A_VALUE(symtab[m_idx].sym_idx) )  {
 96                  m_idx = symtab[m_idx].sym_idx;
 97                  st_clone ( s_idx, m_idx );
 98              }
 99              del_sym ( s_idx );
100          }  else  {
101              /* do mode[1] or mode[2] have a value? */
102              for ( m_idx=1; m_idx<IO_MAX; m_idx++ )  {
103                  if ( HAS_A_VALUE (funtab[f_idx].mode[m_idx]) )  {
104                      st_clone ( s_idx, funtab[f_idx].mode[m_idx] );
105                      funtab[f_idx].mode[m_idx] = NO_VALUE;
106                      clone = YES;
107                  }
108              }

110              /* give the original entity a suffix of the 1st mode */
111              if ( clone EQ YES )  {
112                  st_current ( s_idx, T_FUNCTION );
113                  st_suffix ( funtab[f_idx].mode[0] );
114              }
115          }

117      }  /* for */

119      /* delete the list of modes */
120      del_sym ( st_Mode_list() );

122      RETURN;

124  }  /* st_mult_modes () */
```

```
127  /*
128   * st_clone ()
129   *
130   * Clone one entity
131   *
132   * INPUT:
133   *       called by:              st_mult_modes()
134   *
135   * OUTPUT:
136   *       return type & value:    VOID
137   *
138   * GLOBAL:
139   *       variables used:         Cloning
140   *       variables changed:      symtab[], funtab[]
141   */

143   VOID
144  st_clone ( s_idx, m_idx )
145   NUM s_idx;                     /* symtab entry to be cloned */
146   NUM m_idx;                     /* index to mode suffix      */
147  {
148         VOID st_current(), st_suffix(), st_class(), st_idx(), st_comment();
149         NUM  st_sym(), st_fun();

151         NUM  c_idx;              /* clone index    */
152         NUM  f_idx;              /* original index */
153         NUM  i;                  /* iterator       */

155         BEGIN ( "st_clone" );

157         Cloning = TRUE;

159         /* duplicate the symbol table information */
160         st_current ( st_sym(symtab[s_idx].sym), T_FUNCTION );
161         st_suffix ( m_idx );
162         st_class ( current_idx, symtab[s_idx].sym_class );
163         st_idx ( st_fun() );
164         if ( HAS_A_VALUE (symtab[s_idx].comment[0]) )  {
165                st_comment ( symtab[s_idx].comment );
166         }

168         /* duplicate the function table information. */
169         f_idx = symtab[s_idx].sym_idx;
170         c_idx = ft_idx ();
171         funtab[c_idx].p_or_a            = funtab[f_idx].p_or_a;
172         funtab[c_idx].nec_cond          = funtab[f_idx].nec_cond;
173         for ( i=0; i<IO_MAX; i++ )  {
174                funtab[c_idx].in[i]      = funtab[f_idx].in[i];
175                funtab[c_idx].out[i]     = funtab[f_idx].out[i];
176         }
177         funtab[c_idx].mode[0] = m_idx;
178         if ( HAS_A_VALUE (funtab[f_idx].note[0]) )  {
179                STRCPY ( funtab[c_idx].note, funtab[f_idx].note );
180         }
181         if ( HAS_A_VALUE (funtab[f_idx].warn[0]) )  {
182                STRCPY ( funtab[c_idx].warn, funtab[f_idx].warn );
183         }

185         Cloning = FALSE;

187         RETURN;

189  }  /* st_clone () */
```

```
191  /*
192   * st_transitions ()
193   *
194   * Generate next mode values from lines in the Allowed_Mode_Transitions
195   *
196   * INPUT:
197   *       called by:              mode_table()
198   *
199   * OUTPUT:
200   *       return type & value:    VOID
201   *
202   * GLOBAL:
203   *       variables used:         symtab[], funtab[]
204   *       variables changed:      funtab[].nxt_mode, funtab[].stim
205   */

207   VOID
208  st_transitions ( io_stim, io_val, r_mode, n_mode )
209   char * io_stim;                        /* I_O_DATA stimulus        */
210   char * io_val;                         /* optional io_stim value   */
211   char * r_mode;                         /* required mode            */
212   char * n_mode;                         /* next mode                */
213  {
214          VOID is_iod_name(), is_mode_name();
215          VOID st_current(), st_nxt_mode();

217          NUM  i_idx;                    /* symtab index to io_stim  */
218          NUM  r_idx;                    /* symtab index to r_mode   */
219          NUM  f_idx;                    /* function table index     */
220          NUM  i, idx;                   /* iterators                */
```

```
222         BEGIN ( "st_transitions" );

224         /* are the parameters correct lexically? */
225         is_iod_name ( rname, io_stim );
226         is_mode_name ( rname, r_mode );
227         is_mode_name ( rname, n_mode );

229         /* get the symtab indices to the parameters */
230         i_idx = st_sym ( io_stim );
231         r_idx = st_sym ( r_mode );

233         /* for entities with mode[0] EQ rmode, is io_stim a relation? */
234         for ( idx=0; idx<=sym_idx; idx++ )  {
235                 if ( is_f_or_i(symtab[idx].sym_class) NE T_FUNCTION )  {
236                         continue;
237                 }

239                 f_idx = symtab[idx].sym_idx;

241                 /* does this function have a required mode of r_mode? */
242                 if ( r_idx NE funtab[f_idx].mode[0] )  {
243                         continue;
244                 }

246                 st_current ( idx, T_FUNCTION );

248                 debug ( rname, D_TRACE, "symtab idx=%d: %s=%d, %s=%d",
249                         idx, io_stim, i_idx, r_mode, r_idx );

251                 if ( i_idx EQ funtab[f_idx].nec_cond )  {
252                         debug ( rname, D_TRACE, "nec_cond!" );
253                         st_nxt_mode ( n_mode, io_stim, io_val );
254                 }  else  {
255                         for ( i=0; i<IO_MAX; i++ )  {
256                                 if ((i_idx EQ funtab[f_idx].in[i])  OR
257                                     (i_idx EQ funtab[f_idx].out[i]))  {
258                                         debug ( rname, D_TRACE, "io=%d!", i );
259                                         st_nxt_mode ( n_mode, io_stim, io_val );
260                                 }
261                         }
262                 }
263         }

265         /* fill in nxt_mode for routines which didn't change mode */

267         RETURN;

269 }  /* st_transitions () */
```

```
271  /*
272   * st_orphan ()
273   *
274   * find orphans (without a class) in the symbol table,
275   * and flag references to those orphans.
276   *
277   * INPUT:
278   *       called by:              scan()
279   *       input parameters:       Input  - value of E_INPUT
280   *                               Output - value of E_OUTPUT
281   *                               (both are imported from the SCAN module)
282   *
283   * OUTPUT:
284   *       return type & value:    VOID
285   *
286   * GLOBAL:
287   *       variables used:         symtab[], funtab[]
288   *       variables changed:      funtab[].warn
289   *                               symtab[].media[], symtab[].comment
290   */

292   VOID
293  st_orphan ( Input, Output )
294   NUM Input;                                  /* value of E_INPUT (scan.c)  */
295   NUM Output;                                 /* value of E_OUTPUT (scan.c) */
296  {
297      VOID    st_class();                      /* store default sym_class    */
298      VOID    st_warn();                       /* store warning in funtab    */
299      VOID    st_current();                    /* store current idx & type   */
300      VOID    st_comment();                    /* store comment in entity    */
301      NUM     is_f_or_i();                     /* T_FUNCTION or T_I_O_DATA?  */

303      NUM     orphan;                          /* symtab index to orphan     */
304      NUM     r_idx;                           /* index related to orphan    */
305      NUM     f_idx;                           /* function table index       */
306      NUM     idx;                             /* index for in and out       */
307      char    str [ LINE_LEN ];                /* diagnostic string          */

309      char  * format = "value of '%s' is undefined";
310      char  * io_fmt = "value of '%s[%d]' is undefined";
311      char  * commnt = "Entity undefined by user: default values assumed";
```

```
313     BEGIN ( "st_orphan" );

315     for ( orphan=0; orphan<=sym_idx; orphan++ )  {
316         if ( HAS_NO_VALUE (symtab[orphan].sym[0]) OR
317              HAS_A_VALUE (symtab[orphan].sym_class) )  {
318             continue;
319         }

321         /* search for references to the orphan */
322         for ( r_idx=0; r_idx<=sym_idx; r_idx++ )  {
323                 if ( is_f_or_i(symtab[r_idx].sym_class) NE T_FUNCTION )  {
324                     continue;
325                 }

327                 f_idx = symtab[r_idx].sym_idx;

329                 /* necessary_condition */
330                 if ( funtab[f_idx].nec_cond EQ orphan )  {
331                     /* put a warning in the referencing entity */
332                     st_current ( r_idx, T_FUNCTION );
333                     SPRINTF ( str, format, "necessary_condition" );
334                     st_warn ( str );

336                     /* populate the entity with defaults */
337                     st_current ( orphan, T_I_O_DATA );
338                     st_class ( orphan, Input );
339                     st_media ( "keyboard" );
340                     st_comment ( commnt );
341                 }

343                 for ( idx=0; idx<IO_MAX; idx++ )  {
344                     /* in[0], in[1] and in[2] */
345                     if ( funtab[f_idx].in[idx] EQ orphan )  {
346                         /* put a warning in the referencing entity */
347                         st_current ( r_idx, T_FUNCTION );
348                         SPRINTF ( str, io_fmt, "in", idx );
349                         st_warn ( str );

351                         /* populate the entity with defaults */
352                         st_current ( orphan, T_I_O_DATA );
353                         st_class ( orphan, Input );
354                         st_media ( "keyboard" );
355                         st_comment ( commnt );
356                     }

358                     /* out[0], out[1] and out[2] */
359                     if ( funtab[f_idx].out[idx] EQ orphan )  {
360                         /* put a warning in the referencing entity */
361                         st_current ( r_idx, T_FUNCTION );
362                         SPRINTF ( str, io_fmt, "out", idx );
363                         st_warn ( str );

365                         /* populate the entity with defaults */
366                         st_current ( orphan, T_I_O_DATA );
367                         st_class ( orphan, Output );
368                         st_media ( "crt" );
369                         st_comment ( commnt );
370                     }

372                     /* mode[0], mode[1] and mode[2] */
373                     if ( funtab[f_idx].mode[idx] EQ orphan )  {
374                         /* put a warning in the referencing entity */
375                         st_current ( r_idx, T_FUNCTION );
376                         SPRINTF ( str, io_fmt, "mode", idx );

378                         /* populate the entity with defaults */
379                         st_current ( orphan, T_MODE );
380                         st_class ( orphan, T_MODE );
381                         st_comment ( commnt );
382                     }
383                 }  /* for idx: process the IO_MAX relations */
384         }  /* for r_idx: references to the orphan */
385     }   /* for orphan: look for index to an orphan */

387     RETURN;
388 }  /* st_orphan () */
```

```
390  /*
391   * st_invokes ()
392   *
393   * Create the annotations which will express the transformations that
394   * functions apply to Input_output entities.
395   *
396   * Delete all Input_output entities from the symbol table.
397   *
398   * INPUT:
399   *       called by:                scan ()
400   *       input parameters:         Input_output - value of E_INPUT_OUTPUT
401   *                                 (imported from the SCAN module)
402   *
403   * OUTPUT:
404   *       return type & value:      VOID
405   *
406   * GLOBAL:
407   *       variables used:           funtab[], symtab[]
408   *       variables changed:        funtab[].note
409   */

411   VOID
412  st_invokes ( Input_output )
413   NUM Input_output;
414  {
415      VOID del_sym();
416      NUM  is_f_or_i();

418      char note[LINE_LEN];          /* temporary annotation string      */
419      BOOL invocation;              /* copy temp annotation to note[]?  */
420      BOOL equals;                  /* insert an assignment sign (=)?   */
421      BOOL comma;                   /* insert a comma?                  */
422      NUM  f_idx;                   /* function table index             */
423      NUM  s_idx;                   /* symbol table index               */
424      NUM  i, idx;                  /* iterators                        */
```

```
426     BEGIN ( "st_invokes" );

428     for ( idx=0; idx<=sym_idx; idx++ )  {
429         if ( is_f_or_i(symtab[idx].sym_class) NE T_FUNCTION )  {
430                 /* annotations only apply to functions */
431                 continue;
432         }

434         f_idx = symtab[idx].sym_idx;
435         invocation = equals = NO;
436         note[0] = NULL;         /* clear the temporary annotation */

438         /* process the 'results' of the annotation function */
439         for ( comma=NO, i=0; i<IO_MAX; i++ )  {
440                 if ( HAS_NO_VALUE (s_idx=funtab[f_idx].out[i]) )  {
441                         continue;
442                 }
443                 if ( symtab[s_idx].sym_class EQ Input_output )  {
444                         if ( comma EQ YES )  {
445                                 STRCAT ( note, ", " );
446                         }
447                         STRCAT ( note, symtab[s_idx].sym );
448                         funtab[f_idx].out[i] = NO_VALUE;
449                         comma = equals = invocation = YES;
450                 }
451         }

453         /* now for the name of the annotation function */
454         if ( equals EQ YES )  {
455                 STRCAT ( note, " = " );
456         }
457         STRCAT ( note, symtab[idx].sym );
458         STRCAT ( note, " ( " );

460         /* process the 'arguments' of the annotation function */
461         for ( comma=NO, i=0; i<IO_MAX; i++ )  {
462                 if ( HAS_NO_VALUE (s_idx=funtab[f_idx].in[i]) )  {
463                         continue;
464                 }
465                 if ( symtab[s_idx].sym_class EQ Input_output )  {
466                         if ( comma EQ YES )  {
467                                 STRCAT ( note, ", " );
468                         }
469                         STRCAT ( note, symtab[s_idx].sym );
470                         funtab[f_idx].in[i] = NO_VALUE;
471                         comma = invocation = YES;
472                 }
473         }

475         /* finish the temporary annotation */
476         if ( comma EQ YES )  {
477                 STRCAT ( note, " " );
478         }
479         STRCAT ( note, ")" );

481         /* copy the temporary annotation into the function table */
482         if ( invocation EQ YES )  {
483                 if ( HAS_NO_VALUE (funtab[f_idx].note[0]) )  {
484                         SPRINTF ( funtab[f_idx].note, "INVOKES ( %s )", note);
485                 }  else  {
486                         ERROR ( "max # of annotations = 1" );
487                 }
488         }
489     }  /* for */

491     /* delete the uneeded Input_output entities from the symbol table */
492     for ( idx=0; idx<=sym_idx; idx++ )  {
493         if ( symtab[idx].sym_class EQ Input_output )  {
494                 del_sym ( idx );
495         }
496     }

498     RETURN;

500 }  /* st_invokes () */
```

```
503  /*
504   * st_init (), st_current ()
505   *
506   * initialize symbol table, function table and current entity info
507   *
508   * INPUT:
509   *       called by:              init()
510   *       input parameters:       n/a
511   *
512   * OUTPUT:
513   *       return type & value:    VOID
514   *
515   * GLOBAL:
516   *       variables used:         n/a
517   *       variables changed:      current
518   */

520   VOID
521  st_init ()
522  {
523          VOID del_sym(), del_fun(), st_current();
524          NUM idx;

526          BEGIN ( "st_init" );

528          /* initialize symbol table */
529          for ( idx=0; idx<SYMTAB_SIZE; idx++ )  {
530                  del_sym ( idx );
531          }

533          /* initialize function table */
534          for ( idx=0; idx<FUNTAB_SIZE; idx++ )  {
535                  del_fun ( idx );
536          }

538          /* initialize current entity information */
539          st_current ( NO_VALUE, NO_VALUE );

541          /* initialize table indices */
542          sym_idx = NO_VALUE;
543          fun_idx = NO_VALUE;

545          RETURN;

547  }  /* st_init () */


551   VOID
552  st_current ( i, t )
553   NUM i;                                  /* current.idx   */
554   NUM t;                                  /* current.type  */
555  {
556          BEGIN ( "st_current" );

558          current.idx  = i;
559          current.type = t;

561          RETURN;

563  }  /* st_current () */
```

```
1  #include "T_tables.i"                    /* see for MODULE DESCRIPTION */

3  #define  extern                          /* T_func.c "owns" T_funtab.i */
4  #include "T_funtab.i"                    /* function table             */

6  NUM ent_type();                          /* see T_sym1.c */
7  NUM st_sym();                            /* see T_sym1.c */
8  NUM ft_idx();                            /* see T_sym1.c */
```

```
10  /*
11   * st_fun ()
12   *
13   * return next index to function table
14   *
15   * INPUT:
16   *      called by:              st_entity ()
17   *
18   * OUTPUT:
19   *      return type & value:    current function table index
20   *
21   * GLOBAL:
22   *      variables used:         n/a
23   *      variables changed:      fun_idx
24   */

26   NUM
27  st_fun ()
28  {
29          NUM idx;

31          BEGIN ( "st_fun" );

33          if ( fun_idx < FUNTAB_SIZE-1 )  {
34                  idx = ++fun_idx;
35          }  else  {
36                  ERROR ( "function table overflow" );
37          }

39          RETURN ( idx );

41  }  /* st_fun () */
```

```
43  /*
44   * st_p_or_a ()
45   *
46   * for FUNCTION entities, set p_or_a to PERIODIC or ACTIVITY
47   *
48   * INPUT:
49   *      called by:              st_entity()
50   *
51   * OUTPUT:
52   *      return type & value:    VOID
53   *
54   * GLOBAL:
55   *      variables used:         n/a
56   *      variables changed:      funtab[].p_or_a
57   */

59   VOID
60  st_p_or_a ( class )
61   NUM class;                              /* class of symtab entry */
62  {
63          NUM   f_idx;                    /* function table index */

65          BEGIN ( "st_p_or_a" );

67          f_idx = ft_idx ();

69          if ( HAS_NO_VALUE (funtab[f_idx].p_or_a) )  {
70                  funtab[f_idx].p_or_a = class;
71          }  else  {
72                  ERROR ( "max # of p_or_a values = 1" );
73          }

75          RETURN;

77  }  /* st_p_or_a () */
```

```
 79  /*
 80   * st_nec_cond ()
 81   *
 82   * fill 'necessary_condition' relation value for current entity
 83   *
 84   * INPUT:
 85   *      called by:              relation()
 86   *      input parameters:       rhs - value of 'necessary_condition' keyword
 87   *
 88   * OUTPUT:
 89   *      return type & value:    VOID
 90   *
 91   * GLOBAL:
 92   *      variables used:         n/a
 93   *      variables changed:      funtab[].nec_cond
 94   */

 96   VOID
 97  st_nec_cond ( rhs )
 98   char * rhs;
 99  {
100          VOID is_iod_name();

102          NUM  s_idx;                     /* symbol table index   */
103          NUM  f_idx;                     /* function table index */

105          BEGIN ( "st_nec_cond" );

107          if ( ent_type() NE T_FUNCTION )  {
108                  ERROR ( "necessary_condition: only valid for Activity" );
109          }

111          is_iod_name ( rname, rhs );

113          s_idx = st_sym ( rhs );
114          f_idx = ft_idx ();

116          if ( HAS_NO_VALUE (funtab[f_idx].nec_cond) )  {
117                  funtab[f_idx].nec_cond = s_idx;
118          }  else  {
119                  ERROR ( "Max # of 'necessary_condition' keywords = 1" );
120          }

122          RETURN;

124  }  /* st_nec_cond () */
```

```
127  /*
128   * st_input ()
129   *
130   * fill 'input' relation values for current entity
131   *
132   * INPUT:
133   *       called by:              relation()
134   *       input parameters:       rhs - value of 'input' keyword
135   *
136   * OUTPUT:
137   *       return type & value:    VOID
138   *
139   * GLOBAL:
140   *       variables used:         n/a
141   *       variables changed:      funtab[].in[]
142   */

144   VOID
145  st_input ( rhs )
146   char * rhs;
147  {
148          VOID is_iod_name();

150          NUM  i;                          /* iterator */
151          NUM  s_idx;                      /* symbol table index   */
152          NUM  f_idx;                      /* function table index */

154          BEGIN ( "st_input" );

156          if ( ent_type() NE T_FUNCTION )  {
157                  ERROR ( "input: only valid for FUNCTIONs" );
158          }

160          /* caveat: should check for prior input[] entries */
161          if ( eqstr ( rhs, "NONE" ) )  {
162                  /* no input for this function */
163                  RETURN;
164          }

166          is_iod_name ( rname, rhs );

168          s_idx = st_sym ( rhs );
169          f_idx = ft_idx ();

171          /* put the input value in one of the three input variables */
172          for ( i=0; i<IO_MAX; i++ )  {
173                  if ( HAS_NO_VALUE (funtab[f_idx].in[i]) )  {
174                          funtab[f_idx].in[i] = s_idx;
175                          break;
176                  }

178                  if ( i EQ IO_MAX-1 )  {
179                          /* all 3 of the input variables have been filled */
180                          ERROR ( "Max # of 'input' keywords = 3" );
181                  }
182          }

184          RETURN;

186  }  /* st_input () */
```

```
189  /*
190   * st_output ()
191   *
192   * fill 'output' relation values for current entity
193   *
194   * INPUT:
195   *       called by:              relation()
196   *       input parameters:       rhs - value of 'output' keyword
197   *
198   * OUTPUT:
199   *       return type & value:    VOID
200   *
201   * GLOBAL:
202   *       variables used:         n/a
203   *       variables changed:      funtab[].out[]
204   */

206   VOID
207  st_output ( rhs )
208   char * rhs;
209  {
210          VOID is_iod_name();

212          NUM  i;                         /* iterator */
213          NUM  s_idx;                     /* symbol table index    */
214          NUM  f_idx;                     /* function table index */

216          BEGIN ( "st_output" );

218          is_iod_name ( rname, rhs );

220          s_idx = st_sym ( rhs );
221          f_idx = ft_idx ();

223          if ( ent_type() NE T_FUNCTION )  {
224                  ERROR ( "output: only valid for FUNCTIONs" );
225          }

227          /* put the output value in one of the three output variables */
228          for ( i=0; i<IO_MAX; i++ )  {
229                  if ( HAS_NO_VALUE (funtab[f_idx].out[i]) )  {
230                          funtab[f_idx].out[i] = s_idx;
231                          break;
232                  }

234                  if ( i EQ IO_MAX-1 )  {
235                          /* all 3 of the output variables have been filled */
236                          ERROR ( "Max # of 'output' keywords = 3" );
237                  }
238          }

240          RETURN;

242  }  /* st_output () */
```

```
245 /*
246  * st_req_mode ()
247  *
248  * fill 'required_mode' relation value for current entity
249  *
250  * INPUT:
251  *      called by:              relation()
252  *      input parameters:       rhs - value of 'required_mode' keyword
253  *
254  * OUTPUT:
255  *      return type & value:    VOID
256  *
257  * GLOBAL:
258  *      variables used:         n/a
259  *      variables changed:      funtab[].mode[]
260  */

262  VOID
263 st_req_mode ( rhs )
264  char * rhs;
265 {
266         VOID is_mode_name();

268         NUM i;                          /* iterator */
269         NUM s_idx;                      /* symbol table index   */
270         NUM f_idx;                      /* function table index */

272         BEGIN ( "st_req_mode" );

274         if ( ent_type() NE T_FUNCTION )  {
275                 ERROR ( "required_mode: only valid for FUNCTIONs" );
276         }

278         if ( eqstr ( rhs, "every_mode" ) )  {
279                 debug ( rname, D_TRACE, "duplicate for 'every_mode'" );
280         }  else  {
281                 is_mode_name ( rname, rhs );
282         }

284         s_idx = st_sym ( rhs );
285         f_idx = ft_idx ();

287         /* caveat: should check for prior 'every_mode' */
288         /* put the mode value in one of the three mode variables */
289         for ( i=0; i<IO_MAX; i++ )  {
290                 if ( HAS_NO_VALUE (funtab[f_idx].mode[i]) )  {
291                         funtab[f_idx].mode[i] = s_idx;
292                         break;
293                 }

295                 if ( i EQ IO_MAX-1 )  {
296                         /* all 3 of the mode variables have been filled */
297                         ERROR ( "Max # of 'req_mode' keywords = 3" );
298                 }
299         }

301         RETURN;

303 }  /* st_req_mode () */
```

```
306  /*
307   * st_nxt_mode.()
308   *
309   * fill 'next_mode' value for current entity
310   *
311   * INPUT:
312   *      called by:              st_transitions()
313   *
314   * OUTPUT:
315   *      return type & value:    VOID
316   *
317   * GLOBAL:
318   *      variables used:         n/a
319   *      variables changed:      funtab[].nst_mode
320   */

322   VOID
323  st_nxt_mode ( mode, io_stim, io_val )
324   char * mode;                              /* next mode                    */
325   char * io_stim;                           /* stim: mode[0] -> nxt_mode    */
326   char * io_val;                            /* set of io values             */
327  {
328          NUM  st_sym(), ft_idx();
329          VOID st_current();

331          NUM  s_idx, f_idx;

333          BEGIN ( "st_nxt_mode" );

335          is_mode_name ( rname, mode );

337          s_idx = st_sym ( mode );
338          f_idx = ft_idx ();

340          if ( HAS_NO_VALUE (funtab[f_idx].nxt_mode) )  {
341                  funtab[f_idx].nxt_mode = s_idx;
342                  if ( io_stim[0] AND io_val[0] )  {
343                          SPRINTF ( funtab[f_idx].stim,
344                                  "if %s in %s then ", io_stim, io_val );
345                  }
346          }  else  {
347                  ERROR ( "max # of 'next_mooe' keywords = 1" );
348          }

350          RETURN;

352  }  /* st_nxt_mode () */
```

```
354  /*
355   * st_note ()
356   *
357   * fill 'note' annotation for current Periodic_function entity
358   *
359   * INPUT:
360   *      called by:              relation()
361   *      input parameters:       era_line - string to the right of 'whenever'
362   *
363   * OUTPUT:
364   *      return type & value:    VOID
365   *
366   * GLOBAL:
367   *      variables used:         n/a
368   *      variables changed:      funtab[].note
369   */

371   VOID
372  st_note ( era_line )
373   char * era_line;
374  {
375          NUM  f_idx;                     /* function table index */
376          NUM  i;                         /* sscanf return value  */
377          char str[LINE_LEN];             /* occurrence string    */

379          BEGIN ( "st_note" );

381          if ( ent_type() NE T_FUNCTION )  {
382                  ERROR ( "occurrence: only valid in Periodic_function" );
383          }

385          /* clear the temporary occurrence string */
386          for ( i=0; i<LINE_LEN; i++ )  {
387                  str[i] = NULL;
388          }

390          /* obtain the string following "whenever" */
391          i = sscanf ( era_line, " occurrence : whenever %132c", str );
392          if ( i NE 1 )  {
393                  debug ( rname, D_ERROR, "era=%s", era_line );
394                  debug ( rname, D_ERROR, "sscanf=%d, expr=%s", i, str );
395                  ERROR ( "occurrence : whenever 'expr'" );
396          }

398          f_idx = ft_idx ();

400          if ( HAS_NO_VALUE (funtab[f_idx].note[0]) )  {
401                  SPRINTF ( funtab[f_idx].note, "WHEN ( %s )", str );
402                  debug ( rname, D_TRACE, funtab[f_idx].note );
403          }  else  {
404                  ERROR ( "Max # of annotations = 1" );
405          }

407          RETURN;

409  }  /* st_note () */
```

```
411  /*
412   * st_warn ()
413   *
414   * fill PNL "warning" line with a diagnostic message, when a
415   * potential error is found.
416   *
417   * INPUT:
418   *       called by:              relation()
419   *       input parameters:       str     - warning string
420   *
421   * OUTPUT:
422   *       return type & value:    VOID
423   *
424   * GLOBAL:
425   *       variables used:         n/a
426   *       variables changed:      funtab[].warn
427   */

429   VOID
430  st_warn ( str )
431   char * str;
432  {
433           NUM f_idx;                          /* function table index */

435           BEGIN ( "st_warn" );

437           if ( ent_type() NE T_FUNCTION )  {
438                   ERROR ( "warn: only valid for FUNCTIONs" );
439           }

441           f_idx = ft_idx ();

443           if ( HAS_NO_VALUE (funtab[f_idx].warn[0]) )  {
444                   SPRINTF ( funtab[f_idx].warn, "- WARNING: %s", str );
445           }  else  {
446                   ERROR ( "Max # of warning strings = 1" );
447           }

449           RETURN;

451  }  /* st_warn () */
```

```
453  /*
454   * is_p_or_a ()
455   *
456   * return p_or_a value for current entity
457   *
458   * INPUT:
459   *      called by:              relation()
460   *
461   * OUTPUT:
462   *      return type & value:    p_or_a  - if current entity is FUNCTION
463   *                              FAILURE - if current entity is not
464   *
465   * GLOBAL:
466   *      variables used:         funtab[].p_or_a
467   *      variables changed:      n/a
468   */

470   NUM
471  is_p_or_a ()
472  {
473          if ( ent_type() EQ T_FUNCTION    )  {
474                  return ( funtab[ft_idx ()].p_or_a );
475          }  else  {
476                  return ( FAILURE );
477          }

479  }  /* is_p_or_a () */
```

```
481  /*
482   * del_fun ()
483   *
484   * delete entries from the function table
485   *
486   * INPUT:
487   *       called by:              st_init(), del_sym()
488   *       input parameters:       idx     - table index
489   *
490   * OUTPUT:
491   *       return type & value:    VOID
492   *
493   * GLOBAL:
494   *       variables used:         n/a
495   *       variables changed:      funtab[]
496   */

498   VOID
499  del_fun ( idx )
500   NUM idx;
501  {
502          NUM i;

504          funtab[idx].p_or_a     = NO_VALUE;
505          funtab[idx].nec_cond   = NO_VALUE;
506          for ( i=0; i<IO_MAX; i++ )
507            funtab[idx].in[i]    = NO_VALUE;
508          for ( i=0; i<IO_MAX; i++ )
509            funtab[idx].out[i]   = NO_VALUE;
510          for ( i=0; i<IO_MAX; i++ )
511            funtab[idx].mode[i]  = NO_VALUE;
512          funtab[idx].nxt_mode   = NO_VALUE;
513          funtab[idx].stim [0]   = NO_VALUE;
514          funtab[idx].note [0]   = NO_VALUE;
515          funtab[idx].warn [0]   = NO_VALUE;

517  }  /* del_fun () */
```

```
 1  #include "T_tables.i"                    /* see for MODULE DESCRIPTION */
 2  #include "T_symtab.i"                    /* symbol table               */
 3  #include "T_funtab.i"                    /* function table             */
 4  #include "T_pnltab.i"                    /* PNL tables                 */

 6  /* macros to print the names of symbols;  see pr_tables() */
 7  #define PR_SYM(idx)     FPRINTF ( fp, "%s%s\n", symtab[idx].sym, \
 8   HAS_A_VALUE (symtab[idx].sym_suffix[0]) ? symtab[idx].sym_suffix : NULL );

10  #define NEWLINE         FPRINTF ( fp, "\n" );  Pnl_nl++
11  #define PR_NODE(sym)    FPRINTF ( fp, "%s", sym );

13  NUM  Pnl_nl    = NULL;                   /* PNL line counter    */
14  BOOL Pnl_place = NO;                     /* Place or Transition */
```

```
16  /* ********************* PNL TABLES ***************************** */

18  /*
19   * pr_initial ()
20   *
21   * emit the PNL INIT transition statement
22   *
23   * INPUT:
24   *      called by:              p_pnl()
25   *
26   * OUTPUT:
27   *      return type & value:    VOID
28   *
29   * GLOBAL:
30   *      variables used:         trans[]
31   *      variables changed:      n/a
32   */

34   VOID
35  pr_initial ( fp, type )
36   FILE * fp;                              /* PNL output file pointer */
37   NUM    type;                            /* transition type         */
38  {
39          VOID pr_output();

41          BEGIN ( "pr_initial" );

43          if ( trans[P_INIT].type NE type )  {
44                  ERROR ( "First entry in trans[] must be P_INIT" );
45          }

47          FPRINTF ( fp, "# ------------------> INITial Transition:" );
48          NEWLINE;

50          PR_NODE ( trans[P_INIT].sym );

52          NEWLINE;

54          FPRINTF ( fp, "    : INIT  " );

56          pr_output ( fp, P_INIT );

58          NEWLINE;

60          RETURN;

62  }  /* pr_initial () */
```

```
 65  /*
 66   * pr_places ()
 67   *
 68   * emit the PNL PLACES statements
 69   *
 70   * INPUT:
 71   *      called by:              p_pnl()
 72   *
 73   * OUTPUT:
 74   *      return type & value:    VOID
 75   *
 76   * GLOBAL:
 77   *      variables used:         places[]
 78   *      variables changed:      n/a
 79   */

 81   VOID
 82  pr_places ( fp )
 83   FILE * fp;                              /* PNL output file pointer */
 84  {
 85          VOID pr_output();

 87          NUM  idx;

 89          BEGIN ( "pr_places" );

 91          Pnl_place = YES;

 93          NEWLINE;
 94          FPRINTF ( fp, "# ------------------> PLACEs:" );
 95          NEWLINE;

 97          for ( idx=0; idx<=pl_idx; idx++ )  {
 98                  if ( HAS_A_VALUE (places[idx].comment[0]) )  {
 99                          FPRINTF ( fp, places[idx].comment );
100                          NEWLINE;
101                  }
102                  PR_NODE ( places[idx].sym );
103                  NEWLINE;
104                  FPRINTF ( fp, "     : PLACE " );
105                  pr_output ( fp, idx );
106                  NEWLINE;
107          }

109          Pnl_place = NO;

111          RETURN;

113  }  /* pr_places () */
```

```
116  /*
117   * pr_transitions ()
118   *
119   * emit the PNL function TRANSition statements
120   *
121   * INPUT:
122   *      called by:              p_pnl()
123   *
124   * OUTPUT:
125   *      return type & value:    VOID
126   *
127   * GLOBAL:
128   *      variables used:         trans[], places[]
129   *      variables changed:      n/a
130   */

132   VOID
133  pr_transitions ( fp, type )
134   FILE * fp;                              /* PNL output file pointer */
135   NUM    type;                            /* transition type         */
136  {
137          VOID pr_output(), pr_input();

139          NUM  idx;

141          BEGIN ( "pr_transitions" );

143          NEWLINE;
144          FPRINTF ( fp, "# ------------------> function TRANSitions:\n" );

146          for ( idx=0; idx<=tr_idx; idx++ )  {
147                  if ( trans[idx].type NE type )  {
148                          continue;
149                  }
150                  if ( HAS_A_VALUE (trans[idx].comment[0]) )  {
151                          FPRINTF ( fp, trans[idx].comment );
152                          NEWLINE;
153                  }
154                  if ( HAS_A_VALUE (trans[idx].warn[0]) )  {
155                          FPRINTF ( fp, trans[idx].warn );
156                          NEWLINE;
157                  }
158                  PR_NODE ( trans[idx].sym );
159                  NEWLINE;
160                  FPRINTF ( fp, "    : TRANS " );
161                  pr_output ( fp, idx );
162                  FPRINTF ( fp, "             " );
163                  pr_input ( fp, idx );
164                  if ( HAS_A_VALUE (trans[idx].note[0]) )  {
165                          FPRINTF ( fp, "                " );
166                          FPRINTF ( fp, trans[idx].note );
167                          NEWLINE;
168                  }
169                  if ( HAS_A_VALUE (trans[idx].stim[0]) )  {
170                          FPRINTF ( fp, "                " );
171                          FPRINTF ( fp, trans[idx].stim );
172                          PR_NODE ( places[trans[idx].stim_nxt_mode].sym );
173                          NEWLINE;
174                  }
175                  NEWLINE;
176          }

178          RETURN;

180  }  /* pr_transitions () */
```

```
183  /*
184   * pr_terminals ()
185   *
186   * emit the PNL TERMinal transition statement
187   *
188   * INPUT:
189   *       called by:              p_pnl()
190   *
191   * OUTPUT:
192   *       return type & value:    VOID
193   *
194   * GLOBAL:
195   *       variables used:         trans[]
196   *       variables changed:      n/a
197   */

199   VOID
200  pr_terminals ( fp, type )
201   FILE * fp;                              /* PNL output file pointer */
202   NUM    type;                            /* transition type         */
203  {
204         VOID pr_input();

206         NUM  idx;

208         BEGIN ( "pr_terminals" );

210         NEWLINE;
211         FPRINTF ( fp, "# ------------------> TERMinal Transitions:" );
212         NEWLINE;

214         for ( idx=0; idx<tr_idx; idx++ )  {
215                 if ( trans[idx].type NE type )  {
216                         continue;
217                 }
218                 PR_NODE ( trans[idx].sym );
219                 NEWLINE;
220                 FPRINTF ( fp, "     : TERM  " );
221                 pr_input ( fp, idx );
222                 NEWLINE;
223         }

225         RETURN;

227  }  /* pr_terminals () */
```

```
230  /*
231   * pr_output ()
232   *
233   * emit the list of nodes which are pointed to by output arcs
234   * from the current node
235   *
236   * INPUT:
237   *      called by:              pr_initial(), pr_places(), pr_transitions()
238   *
239   * OUTPUT:
240   *      return type & value:    VOID
241   *
242   * GLOBAL:
243   *      variables used:         places[], trans[]
244   *      variables changed:      n/a
245   */

247   VOID
248  pr_output ( fp, index )
249   FILE * fp;                                /* PNL output file pointer */
250   NUM    index;                             /* current node's index    */
251  {
252          NUM i, idx, comma;

254          BEGIN ( "pr_output" );

256          FPRINTF ( fp, "OUTPUT TO ( " );

258          comma = NO;
259          for ( i=0; i<OUT_MAX; i++ )  {
260                  if ( Pnl_place ? HAS_NO_VALUE(places[index].out[i]) :
261                                   HAS_NO_VALUE(trans[index].out[i]) )  {
262                          break;
263                  }

265                  if ( comma EQ YES )  {
266                          FPRINTF ( fp, ", " );
267                  }
268                  comma = YES;

270                  if ( Pnl_place )  {
271                          idx = places[index].out[i];
272                          PR_NODE ( trans[idx].sym );
273                  }  else  {
274                          idx = trans[index].out[i];
275                          PR_NODE ( places[idx].sym );
276                  }
277          }

279          FPRINTF ( fp, " )" );
280          NEWLINE;

282          RETURN;

284  }  /* pr_output () */
```

```
287  /*
288   * pr_input ()
289   *
290   * emit the list of nodes which are pointed to by input arcs
291   * from the current node
292   *
293   * INPUT:
294   *      called by:              pr_transitions(), pr_terminals()
295   *
296   * OUTPUT:
297   *      return type & value:    VOID
298   *
299   * GLOBAL:
300   *      variables used:         trans[], places[]
301   *      variables changed:      n/a
302   */

304   VOID
305  pr_input ( fp, index )
306   FILE * fp;                              /* PNL output file pointer */
307   NUM    index;                           /* current node's index    */
308  {
309          NUM i, idx, comma;

311          BEGIN ( "pr_input" );

313          FPRINTF ( fp, "INPUT FROM ( " );

315          comma = NO;
316          for ( i=0; i<IN_MAX; i++ )  {
317                  if ( HAS_NO_VALUE(trans[index].in[i]) )  {
318                          break;
319                  }

321                  if ( comma EQ YES )  {
322                          FPRINTF ( fp, ", " );
323                  }
324                  comma = YES;

326                  idx = trans[index].in[i];
327                  PR_NODE ( places[idx].sym );
328          }

330          FPRINTF ( fp, " )" );
331          NEWLINE;

333          RETURN;

335  }  /* pr_input () */
```

```
337  /*
338   * pr_Pplace ()
339   *
340   * print an entry in places[]
341   *
342   * INPUT:
343   *       called by:              pr_tables()
344   *
345   * OUTPUT:
346   *       return type & value:    VOID
347   *
348   * GLOBAL:
349   *       variables used:         places[], trans[]
350   *       variables changed:      n/a
351   */

353   VOID
354  pr_Pplace ( fp, idx )
355   FILE * fp;                              /* TABLES_FILE file pointer */
356   NUM    idx;                             /* index to places[]        */
357  {
358          NUM i;

360          BEGIN ( "pr_Pplace" );

362          if ( HAS_NO_VALUE (places[idx].sym[0]) )  {
363                  RETURN;
364          }

366          NEWLINE;

368          FPRINTF ( fp, "place[%2d] : ", idx );
369          PR_NODE ( places[idx].sym );
370          NEWLINE;

372          for ( i=0; i<IN_MAX; i++ )  {
373                  if ( HAS_A_VALUE (places[idx].out[i]) )  {
374                          FPRINTF ( fp, "out[%2d]    : %2d ", i, places[idx].out[i]
     );
375                          PR_NODE ( trans[places[idx].out[i]].sym );
376                          NEWLINE;
377                  }  else  {
378                          break;
379                  }
380          }
381          if ( HAS_A_VALUE (places[idx].comment[0]) )  {
382                  FPRINTF ( fp, "comment    : %s\n", places[idx].comment );
383          }

385          RETURN;

387  }  /* pr_Pplace () */
```

```
390  /*
391   * pr_Ptrans ()
392   *
393   * print an entry in trans[]
394   *
395   * INPUT:
396   *      called by:              pr_tables()
397   *
398   * OUTPUT:
399   *      return type & value:    VOID
400   *
401   * GLOBAL:
402   *      variables used:         trans[], places[]
403   *      variables changed:      n/a
404   */

406   VOID
407  pr_Ptrans ( fp, idx )
408   FILE * fp;                              /* TABLES_FILE file pointer */
409   NUM    idx;                             /* index to trans[]         */
410  {
411          NUM i;

413          BEGIN ( "pr_Ptrans" );


416          if ( HAS_NO_VALUE (trans[idx].sym[0]) )  {
417                  RETURN;
418          }

420          NEWLINE;

422          FPRINTF ( fp, "trans[%2d] : ", idx );
423          PR_NODE ( trans[idx].sym );
424          NEWLINE;

426          FPRINTF ( fp, "type      : %d\n", trans[idx].type );
427          for ( i=0; i<IN_MAX; i++ )  {
428                  if ( HAS_A_VALUE (trans[idx].in[i]) )  {
429                          FPRINTF ( fp, "in[%2d]    : %2d ", i, trans[idx].in[i] )'
    ;
430                          PR_NODE ( places[trans[idx].in[i]].sym );
431                          NEWLINE;
432                  }  else  {
433                          break;
434                  }
435          }
436          for ( i=0; i<OUT_MAX; i++ )  {
437                  if ( HAS_A_VALUE (trans[idx].out[i]) )  {
438                          FPRINTF ( fp, "out[%2d]   : %2d ", i, trans[idx].out[i] '
    );
439                          PR_NODE ( places[trans[idx].out[i]].sym );
440                          NEWLINE;
441                  }  else  {
442                          break;
443                  }
444          }
445          if ( HAS_A_VALUE (trans[idx].comment[0]) )  {
446                  FPRINTF ( fp, "comment   : %s\n", trans[idx].comment );
447          }
448          if ( HAS_A_VALUE (trans[idx].stim[0]) )  {
449                  FPRINTF ( fp, "stim      : %s ", trans[idx].stim );
450                  PR_NODE ( places[trans[idx].stim_nxt_mode].sym );
451                  NEWLINE;
452          }
453          if ( HAS_A_VALUE (trans[idx].note[0]) )  {
454                  FPRINTF ( fp, "note      : %s\n", trans[idx].note );
455          }
456          if ( HAS_A_VALUE (trans[idx].warn[0]) )  {
457                  FPRINTF ( fp, "warn      : %s\n", trans[idx].warn );
458          }

460          RETURN;

462  }  /* pr_Ptrans () */
```

```
464  /* ******************** SYMTAB & FUNTAB ************************** */

466  /*
467   * pr_entity ()
468   *
469   * print 1 entity in the symbol table
470   *
471   * INPUT:
472   *      called by:              pr_tables ()
473   *      input parameters:       fp      - FILE * for output
474   *                              idx     - symbol table index
475   *
476   * OUTPUT:
477   *      return type & value:    VOID
478   *
479   * GLOBAL:
480   *      variables used:         symtab[], funtab[]
481   *      variables changed:      n/a
482   */

484   VOID
485  pr_entity ( fp, idx )
486   FILE * fp;                                  /* TABLES_FILE file pointer */
487   NUM    idx;
488  {
489          VOID   pr_class();
490          NUM    is_f_or_i();

492          char * rname = "pr_entity";
493          NUM    i, sw, f_idx;

495          FPRINTF ( fp, "\nsymtab[%2d]   : ", idx );

497          if ( HAS_NO_VALUE (symtab[idx].sym[0]) )  {
498                  FPRINTF ( fp, ">>> NO_VALUE <<<\n" );
499                  return;
500          }  else  {
501                  PR_SYM ( idx );
502                  pr_class ( fp, symtab[idx].sym_class );

504                  /* print the PNL comment attached to this symbol */
505                  if ( HAS_A_VALUE (symtab[idx].comment[0]) )  {
506                          FPRINTF ( fp, "   comment   : " );
507                          FPRINTF ( fp, "%s\n", symtab[idx].comment );
508                  }
509          }

511          switch ( sw = is_f_or_i(symtab[idx].sym_class) )  {
512          case T_MODE:
513                  /* T_MODE is also the ERROR return from is_f_or_i() */
514                  break;

516          case T_FUNCTION:
517                  f_idx = symtab[idx].sym_idx;

519                  if ( HAS_A_VALUE (funtab[f_idx].nec_cond) )  {
520                          FPRINTF ( fp, "   nec_cond  : " );
521                          PR_SYM ( funtab[f_idx].nec_cond );
522                  }

524                  for ( i=0; i<IO_MAX; i++ )  {
525                          if ( HAS_A_VALUE (funtab[f_idx].in[i]) )  {
526                                  FPRINTF ( fp, "   in[%d]    : ", i );
527                                  PR_SYM ( funtab[f_idx].in[i] );
528                          }
529                  }

531                  for ( i=0; i<IO_MAX; i++ )  {
532                          if ( HAS_A_VALUE (funtab[f_idx].out[i]) )  {
533                                  FPRINTF ( fp, "   out[%d]   : ", i );
534                                  PR_SYM ( funtab[f_idx].out[i] );
535                          }
536                  }

538                  for ( i=0; i<IO_MAX; i++ )  {
539                          if ( HAS_A_VALUE (funtab[f_idx].mode[i]) )  {
540                                  FPRINTF ( fp, "   mode[%d]  : ", i );
```

```
541                                     PR_SYM ( funtab[f_idx].mode[i] );
542                             }
543                     }

545                     if ( HAS_A_VALUE (funtab[f_idx].nxt_mode) )  {
546                             FPRINTF ( fp, "   nxt_mode   : " );
547                             if ( HAS_A_VALUE (funtab[f_idx].stim[0]) )  {
548                                     FPRINTF ( fp, funtab[f_idx].stim );
549                             }
550                             PR_SYM ( funtab[f_idx].nxt_mode );
551                     }

553                     if ( HAS_A_VALUE (funtab[f_idx].note[0]) )  {
554                             FPRINTF ( fp, "   note       : " );
555                             FPRINTF ( fp, "%s\n", funtab[f_idx].note );
556                     }

558                     if ( HAS_A_VALUE (funtab[f_idx].warn[0]) )  {
559                             FPRINTF ( fp, "   warn       : " );
560                             FPRINTF ( fp, "%s\n", funtab[f_idx].warn );
561                     }
562                     break;

564             case T_I_O_DATA:
565                     FPRINTF ( fp, "   media      : %s\n",
566                               media[symtab[idx].sym_idx] );
567                     break;

569             default:
570                     /* this should NEVER be executed!!! */
571                     debug ( rname, D_ERROR, "switch=%d", sw );
572                     if ( HAS_A_VALUE (symtab[idx].sym[0]) )  {
573                             debug ( rname, D_ERROR, "sym[%d]=%s",
574                                     idx, symtab[idx].sym );
575                     }
576                     /* ERROR ( "internal switch error" ); */
577             }  /* switch */

579             return;

581     }  /* pr_entity () */
```

```
584  /*
585   * pr_mode_list ()
586   *
587   * print the Mode list (see mode_table())
588   *
589   * INPUT:
590   *       called by:              pr_tables()
591   *
592   * OUTPUT:
593   *       return type & value:    VOID
594   *
595   * GLOBAL:
596   *       variables used:         Init_Mode
597   *       variables changed:      n/a
598   */

600   VOID
601  pr_mode_list ( fp )
602   FILE * fp;                                  /* TABLES_FILE file pointer */
603  {
604          NUM st_Mode_list();
605          NUM idx;

607          BEGIN ( "pr_mode_list" );

609          FPRINTF ( fp, "\nInitial_Mode : %s\n", symtab[Init_Mode].sym );

611          /* print the head of the list */
612          idx = st_Mode_list ();
613          FPRINTF ( fp, "Mode_list     : %s", symtab[idx].sym );

615          while ( HAS_A_VALUE(symtab[idx].sym_idx) )  {
616                  idx = symtab[idx].sym_idx;
617                  FPRINTF ( fp, ", %s", symtab[idx].sym );
618          }

620          FPRINTF ( fp, "\n" );

622          RETURN;

624  }  /* pr_mode_list () */
```

```
627  /*
628   * pr_tables ()
629   *
630   * print the tables
631   *
632   * INPUT:
633   *       called by:              fini ()
634   *       input parameters:       n/a
635   *
636   * OUTPUT:
637   *       return type & value:    VOID
638   *
639   * GLOBAL:
640   *       variables used:         sym_idx
641   *       variables changed:      n/a
642   */

644   VOID
645  pr_tables ()
646  {
647          VOID pr_entity(), pr_mode_list(), perror();
648          VOID pr_Pplace(), pr_Ptrans();

650          NUM idx;
651          FILE * Tab_fp, * fopen();

653          BEGIN ( "pr_tables" );

655          if ( ( Tab_fp = fopen ( TABLES_FILE, "w" ) ) EQ NULL )  {
656                  perror ( rname );
657                  debug ( rname, D_ERROR, "can't open TABLES_FILE (see files.h)" )
     ;
658          }

660          for ( idx=0; idx<=sym_idx; idx++ )  {
661                  pr_entity ( Tab_fp, idx );
662          }

664          pr_mode_list ( Tab_fp );

666          FPRINTF ( Tab_fp, "\f" );
667          for ( idx=0; idx<PLACES_SIZE; idx++ )  {
668                  pr_Pplace ( Tab_fp, idx );
669          }

671          FPRINTF ( Tab_fp, "\f" );
672          for ( idx=0; idx<TRANS_SIZE; idx++ )  {
673                  pr_Ptrans ( Tab_fp, idx );
674          }

676          FFLUSH ( Tab_fp );

678          if ( fclose ( Tab_fp ) EQ EOF )  {
679                  perror ( rname );
680                  debug ( rname, D_ERROR, "closing Tab_fp" );
681          }

683          RETURN;

685  }  /* pr_tables () */
```

```
/*************************************************************
 *
 * MODULE:       INPUT
 *
 * contains access routines to get input from files
 *
 * PROJECT:      Requirements Analysis using Petri Nets
 *               CMPSC 690 - Master's Project, Summer 1984
 *
 * AUTHOR:       Brad C. Gaylord
 *               AT&T Information System Laboratories
 *               11900 North Pecos, Denver CO,  80234
 *               drux2!bcg, DR 31L60,  (303)-538-1413
 *
 *************************************************************
 *
 * USED BY:      SCAN module
 *
 * ACCESS FCTS: get_line(), fatal_error()
 *
 *************************************************************
 *
 * USES:         n/a
 *
 * DATA OWNED:   Inp_line[], Inp_count, Inp_fp
 *
 *************************************************************
 */

/* INCLUDE files required by this module: */
#include <stdio.h>
#include "project.h"                    /* general definitions        */

#define  D_MODULE       D_INPUT         /* primary module trace level */
#include "debug.h"                      /* debug levels               */

/* DEFINEs local to this module: */
/* fatal error string (see fatal_error) */
#define ERR_STR "ERROR found by %s():\n\tinput line %d: %s\n\toffending line fol
lows:\n'%s'\n\n"


/* global variables "local" to this module: */
FILE    * Inp_fp;                       /* input file pointer         */

char    Inp_line [ LINE_LEN ];          /* input buffer               */
NUM     Inp_count;                      /* line number                */
```

```
51  /*
52   * get_line ()
53   *
54   * This routine gets a single line from the input FILE pointer
55   * and returns a pointer to the buffer containing that line
56   *
57   * INPUT:
58   *      called by:              scan(), mode_table()
59   *      input parameters:       n/a
60   *
61   * OUTPUT:
62   *      return type & value:    (char *) Inp_line
63   *                              (char *) NULL           - at EOF
64   *
65   * GLOBAL:
66   *      variables used:         Inp_fp
67   *      variables changed:      Inp_line[], Inp_count
68   */

70   char *
71  get_line ()
72  {
73          char * fgets();
74          char * line;

76          BEGIN ( "get_line" );

78          /* increment the line number */
79          Inp_count ++;

81          line = fgets ( Inp_line, LINE_LEN, Inp_fp );

83          /* remove the newline at the end of Inp_line */
84          if ( line NE NULL )  {
85                  line [ strlen(Inp_line) - 1 ] = NULL;
86          }

88          debug ( rname, D_TRACE, "line %3d: %s", Inp_count, Inp_line );
89          RETURN ( line );

91  }  /* get_line () */
```

```
 93  /*
 94   * fatal_error ()
 95   *
 96   * print fatal error message and gracefully capitulate
 97   *
 98   * INPUT:
 99   *      called by:              ERROR() macro (debug.h)
100   *      input parameters:       rtn_name - calling routine's name
101   *                              str      - fatal error message
102   *
103   * OUTPUT:
104   *      return type & value:    process exit.
105   *
106   * GLOBAL:
107   *      variables used:         Inp_count, Inp_line
108   *      variables changed:      n/a
109   */

111  VOID
112 fatal_error ( rtn_name, str )
113  char * rtn_name;
114  char * str;
115 {
116         VOID fini();

118         BEGIN ( "fatal_error" );

120         /* display error message on terminal screen */
121         FPRINTF ( stderr, ERR_STR, rtn_name, Inp_count, str, Inp_line );

123         /* final trace message */
124         debug ( rtn_name, D_ERROR, ERR_STR, rtn_name, Inp_count, str, Inp_line);

126         /* terminate process cleanly */
127         fini ( FAILURE );

129 }  /* fatal_error () */
```

```
131  /*
132   * Inp_init ()
133   *
134   * This routine opens in_file for reading, and initializes variables
135   * related to input activities.
136   *
137   * INPUT:
138   *       called by:              init()
139   *       input parameters:       in_file - input file name
140   *
141   * OUTPUT:
142   *       return type & value:    n/a
143   *
144   * GLOBAL:
145   *       variables used:         n/a
146   *       variables changed:      Inp_fp, Inp_count
147   */

149   VOID
150  Inp_init ( in_file )
151   char * in_file;
152  {
153          VOID   exit(), perror();
154          FILE * fopen();

156          BEGIN ( "Inp_init" );

158          Inp_count = 0;

160          if ( ( Inp_fp = fopen ( in_file, "r" ) ) EQ NULL )  {
161                  perror ( "Inp_init: Inp_fp" );
162                  exit ( 1 );
163          }

165          RETURN;

167  }  /* Inp_init () */
```

```
170  /*
171   * Inp_fini ()
172   *
173   * closes Inp_fp
174   *
175   * INPUT:
176   *       called by:              fini()
177   *       input parameters:       n/a
178   *
179   * OUTPUT:
180   *       return type & value:    VOID
181   *
182   * GLOBAL:
183   *       variables used:         n/a
184   *       variables changed:      n/a
185   */

187   VOID
188  Inp_fini ()
189  {
190           VOID perror();

192           BEGIN ( "Inp_fini" );

194           if ( fclose(Inp_fp) EQ EOF )  {
195                   perror ( "Inp_fini: Inp_fp" );
196           }

198           RETURN;

200  }  /* Inp_fini () */
```

```
 1  /*
 2   * MODULE:       DEBUG
 3   *
 4   * If you don't have a source line oriented debugger, this is the
 5   * next best thing.  DEBUG allows you to put "printf" like statements
 6   * in your code, and then selectively turn any of the DEBUG statements
 7   * "on" and "off" between runs (without recompilation).
 8   *
 9   * First you put calls to debug() in your source.  debug() statements
10   * resemble calls to printf, in that a variable number of arguments
11   * are printed according to a format.  One of the parameters specifies
12   * the "trace level" of the debug() statement.  If that trace level
13   * is turned "on", the debug() statement will print a message.
14   * The trace levels are #defined symbolically:  for more details, see
15   * #include "debug.h"                  <---the parameters used by debug()
16   *
17   * You must have a global variable "program" declared above main() thusly:
18   *      char * program = "foo";
19   *
20   * These files must be in the directory where the program is run:
21   *
22   *      "i.foo" - if this file does NOT exist, debug() statements
23   *                do nothing.  Maximum cpu overhead is approx. 10%.
24   *
25   *              - if this file exists, it is checked for a list of
26   *                numeric "trace level" numbers (from 0 to LEVELS-1)
27   *                (ie: a line containing the numbers "20 29 30").
28   *                Only those debug() statements which specify one of
29   *                these trace levels will produce output.
30   *                NOTE: if i.foo contains the magic number "LEVELS",
31   *                then HOG tracing is enabled:  all debug()
32   *                statements are enabled and will produce output.
33   *                (see the #define for LEVELS in debug.c)
34   *
35   *      "o.foo" - if this file does NOT exist, the debug() output
36   *                is sent directly to the terminal.
37   *
38   *              - if it does exist, and contains a valid writeable
39   *                file name, the debug output is appended to the file.
40   *
41   *              - if it does exist, but is empty, the debug() output
42   *                is sent to the file "x.foo.12345".  The "12345"
43   *                is the process id (pid), and thus will be unique.
44   *                You end up with a LOT of "x.*" files after a while.
45   *
46   * ACCESS FCTS: debug (-)
47   *
48   * USED BY:      any routine
49   *      This module is suitable for linking with any other modules.
50   *      All internal data and functions are prefixed with "Dbug_",
51   *      so as to avoid collisions with declarations in other modules.
52   *
53   * USES:         n/a
54   *      This module isn't dependent upon other routines in this project.
55   *      Routines in other modules may use the definitions in "debug.h",
56   *      especially the trace levels, and BEGIN, RETURN and ERROR macros.
57   *      However, this module does not use the definitions in "debug.h".
58   *
59   * TO BE DONE:   - use an "rname stack" instead of rname parameter.
60   *                 BEGIN() will push rname onto stack.
61   *                 fatal_error() will printf stack.
62   *               - trace file will show the depth of the stack
63   *                 as well as the parameters and return values.
64   *
65   * ACKNOWLEDGMENTS:
66   *      Jim Laur originated the idea via his outlog() routine (RMATS II)
67   *
68   * PROJECT:      Requirements Analysis using Petri Nets
69   *               CMPSC 690 - Master's Project, Summer 1984
70   *
71   * AUTHOR:       Brad C. Gaylord, LABD
72   *               AT&T Information System Laboratories
73   *               11900 North Pecos, Denver CO,  80234
74   *               drux2!bcg, DR 31L60,  (303)-538-1413
75   *
76   */
```

```
78  /* INCLUDE files required by this module: */
79  #include <stdio.h>
80  #include <varargs.h>
81  #include <ctype.h>
82  #include "project.h"                    /* basic definitions          */


85  /* DEFINEs local to this module: */
86  #define LEVELS          100             /* max number of trace levels */
87  #define ON              YES             /* print trace at this level  */
88  #define OFF             NO              /* don't print at this level  */

90  #define PATH_SIZE       132             /* # of chars in path name    */
91  #define PATH_FORMAT     "%132s"         /* fmt of non "x." file name  */

93  /* These file names must be no more than 14 characters (UNIX default) */
94  #define IN_FORMAT       "i.%.12s"       /* "i." file: i.aaaaaaaaaaaa  */
95  #define OUT_FORMAT      "c.%.12s"       /* "o." file: o.aaaaaaaaaaaa  */
96  #define DBUG_FORMAT     "x.%.6s.%.5d"   /* "x." file: x.aaaaaa.nnnnn  */

98  #define TTY             "/dev/tty"      /* terminal's file name       */

100 /* external variables declared outside this module */
101 extern char * program;                  /* declared in main program   */
102 extern int    _doprnt();                /* varargs: can't use fprintf */

104 /* global variables "local" to this module: */
105 char   Dbug_on = 'n';                   /* is debug turned on yet?    */
106 char   Dbug_bad_fp = 'n';               /* if 'y' no write to Dbug_fp */
107 char   Dbug_level[LEVELS];              /* array of active levels     */

109 FILE * Dbug_fp = NULL;                  /* file pointer for debugging */
```

```
112  /*
113   * debug ()
114   *
115   * "printf(3)"-like debugging statements, which can be selectively
116   * turned on and off between runs (without recompilation).
117   *
118   * INPUT:
119   *      called by:              any routine
120   *      input parameters:       see parameter definitions
121   *
122   * OUTPUT:
123   *      return type & value:    VOID
124   *
125   * GLOBAL:
126   *      routines called:        Dbug_init(), debug(), _doprnt()
127   *      variables used:         Dbug_level[]
128   *      variables changed:      Dbug_bad_fp, Dbug_on
129   */

131  /*VARARGS3*/
132  VOID
133  debug ( name, level, format, va_alist )
134  char * name;                                /* calling routine name       */
135  int    level;                               /* trace level: [0,LEVELS-1]  */
136  char * format;                              /* printf(3)-like format      */
137  va_dcl                                      /* varargs declaration        */
138  {
139          va_list ap;                         /* varargs argument pointer   */

141          /* return immediately if debug initialization has already failed  */
142          if ( Dbug_bad_fp EQ 'y' )  {
143                  /* Dbug_init() failed previously */
144                  return;
145          }

147          /* if debug hasn't been turned on, do so */
148          if ( Dbug_on EQ 'n' )  {
149                  if ( Dbug_init() NE SUCCESS )  {
150                          /* initialization failure */
151                          Dbug_bad_fp = 'y';
152                          return;
153                  }
154                  Dbug_on = 'y';
155          }

157          /* return if debugging is not turned on for this trace level */
158          /* note that debugging is always turned on for FAILUREs (-1) */
159          if ( ( level >= 0 ) AND ( level < LEVELS ) )  {
160                  if ( NOT Dbug_level [level] )  {
161                          return;
162                  }
163          }

165          /* negative level codes are allowed (ie: FAILURE)     */
166          /* codes greater than LEVELS are flagged as an error */
167          if ( level > LEVELS )  {
168                  debug ( "debug", FAILURE, "%d > LEVELS: see debug.c", level );
169          }

171          /* a timestamp could be printed here, if desired */
172          FPRINTF ( Dbug_fp, "%s[%2d]: ", name, level );

174          /* this snippet of code is the heart of fprintf()  */
175          /* _doprnt() does the va_arg() analysis of ap      */
176          /* see /usr/src/lib/libc/port/print/fprintf.c      */
177          va_start(ap);                                  /* varargs */
178          (VOID) _doprnt(format, ap, Dbug_fp);           /* varargs */
179          va_end(ap);                                    /* varargs */

181          FPRINTF ( Dbug_fp, "\n" );
182          FFLUSH  ( Dbug_fp );

184          return;
185  }  /* debug () */
```

```
188  /*
189   * Dbug_init ()
190   *
191   *initalize logging by
192   *      - checking the trace levels in the i.PROGRAM file and
193   *      - opening an file for the trace outputs
194   *
195   * INPUT:
196   *      called by:              debug()
197   *      input parameters:       n/a
198   *
199   * OUTPUT:
200   *      return type & value:    SUCCESS - Dbug_fp is open
201   *                              FAILURE - Dbug_fp couldn't be opened
202   *
203   * GLOBAL:
204   *      routines called:        Dbug_header()
205   *      variables used:         program
206   *      variables changed:      Dbug_fp, Dbug_level[]
207   */


210  Dbug_init ()
211  {
212          VOID Dbug_header();

214          char file_name [PATH_SIZE + 1];         /* .i, .o & .x files */

216          FILE     *fp;
217          int      ret;
218          int      c;
219          register i;

221          /* initialize all levels to off */
222          for ( i = 0; i < LEVELS; i++ )  {
223                  Dbug_level[i] = OFF;
224          }

226          /* get name of input file containing trace levels */
227          SPRINTF ( file_name, IN_FORMAT, program );

229          /* try opening input file */
230          if ( (fp = fopen (file_name,"r")) NE NULL )  {
231                  /* read trace levels from input file */
232                  for EVER  {
233                          /* get the next number from input file */
234                          if ( fscanf (fp, "%d", &c) EQ EOF )  {
235                                  break;
236                          }

238                          /* check for normal tracing */
239                          if ( (c >= 0) AND (c < LEVELS) )  {
240                                  Dbug_level[c] = ON;
241                                  continue;
242                          }

244                          /* check for HOG tracing */
245                          if ( c EQ LEVELS )  {
246                                  for ( i=0; i<LEVELS; i++ )  {
247                                          Dbug_level[i] = ON;
248                                  }
249                                  break;
250                          }
251                  }
252                  FCLOSE ( fp );
253          }
254          else  {
255                  /* i.PROGRAM file is not readable */
256                  return (FAILURE);
257          }
```

```
259             /*
260              * The i.PROGRAM file exists,
261              * and the trace levels have been read.
262              * Now where does the output go?
263              *
264              *      if o.PROGRAM is readable
265              *              if o.PROGRAM contains a file name
266              *                      output is appended to that file
267              *              else output goes to x.PROGRAM.pid
268              *      else if o.PROGRAM doesn't exist,
269              *              output > /dev/tty (if it can be opened)
270              */

272             SPRINTF ( file_name, OUT_FORMAT, program );

274             /* is the o.PROGRAM file readable? */
275             if ( (fp = fopen ( file_name, "r" )) EQ NULL )  {
276                     /* the o.PROGRAM file isn't readable, so use tty */
277                     if ( (Dbug_fp = fopen ( TTY, "w" )) EQ NULL)  {
278                             /* we couldn't even open the tty to write to */
279                             return (FAILURE);
280                     }
281                     else  {
282                             /* print a header */
283                             Dbug_header ();

285                             /* all trace output goes the terminal */
286                             return (SUCCESS);
287                     }
288             }

290             /* does o.PROGRAM contain a file name? */
291             ret = SUCCESS;
292             if ( fscanf ( fp, PATH_FORMAT, file_name ) EQ 1 )  {
293                     /* use contents of "o." file, instead of "x." file */
294                     for (i = 0; (i < PATH_SIZE) AND (file_name[i] NE NULL); i++)  {
295                             if ( isprint ((int) file_name[i]) EQ FALSE )  {
296                                     /* unprintable char in file_name */
297                                     ret = FAILURE;
298                                     break;
299                             }
300                     }
301             }
302             else  {
303                     /* no file name string in the o.PROGRAM file */
304                     ret = FAILURE;
305             }
306             FCLOSE ( fp );

308             if ( ret EQ SUCCESS )  {
309                     /* open the trace output file */
310                     if ( (Dbug_fp = fopen ( file_name, "a" )) EQ NULL)  {
311                             /* couldn't open the file, try the x. default */
312                             ret = FAILURE;
313                     }
314             }

316             if ( ret NE SUCCESS )  {
317                     /* output the trace to the default "x." file */
318                     SPRINTF ( file_name, DBUG_FORMAT, program, getpid() );

320                     /* open the default ".x" trace output file */
321                     if ( (Dbug_fp = fopen ( file_name, "a" )) EQ NULL)  {
322                             return ( FAILURE );
323                     }
324             }

326             /* print a header */
327             Dbug_header ();

329             return ( SUCCESS );

331     }  /* Dbug_init () */
```

```
333  /*
334   * Dbug_header ()
335   *
336   * Output a trace header to the file, and return successfully.
337   * The output goes to Dbug_fp, which has been successfully opened.
338   *
339   * INPUT:
340   *      called by:              Dbug_init()
341   *      input parameters:       n/a
342   *
343   * OUTPUT:
344   *      return type & value:    VOID
345   *
346   * GLOBAL:
347   *      routines called:        time(), ctime()
348   *      variables used:         Dbug_fp
349   *      variables changed:      n/a
350   */

352  VOID
353  Dbug_header ()
354  {
355          long     clock, time();
356          char     *ctime();
357          register i;

359          /* print a date stamp */
360          clock = time ( (long *) 0 );
361          FPRINTF ( Dbug_fp, "--------> TRACE TIME: %s", ctime (&clock) );

363          /* display the trace levels */
364          FPRINTF ( Dbug_fp, "--------> TRACE LEVELS: ");
365          for ( i=0; i<LEVELS; i++ )  {
366                  if ( Dbug_level[i] )  {
367                          FPRINTF ( Dbug_fp, "%d ", i );
368                  }
369          }
370          FPRINTF ( Dbug_fp , "\n" );

372          FFLUSH ( Dbug_fp );

374          return;

376  }  /* Dbug_header () */
```

```
379  /*
380   * Dbug_fini ()
381   *
382   * close the file pointer associated with the trace output file
383   *
384   * INPUT:
385   *       called by:              fini()
386   *       input parameters:       n/a
387   *
388   * OUTPUT:
389   *       return type & value:    VOID
390   *
391   * GLOBAL:
392   *       variables used:         Dbug_fp_bad, Dbug_on
393   *       variables changed:      Dbug_fp
394   */

396   VOID
397  Dbug_fini ()
398  {
399           if ( (Dbug_bad_fp EQ 'y') OR (Dbug_on EQ 'n') )  {
400                   return;
401           }

403           if ( Dbug_fp NE NULL )  {
404                   FCLOSE ( Dbug_fp );
405           }

407           return;

409  }  /* Dbug_fini () */
```

Mon Nov 19 03:22:46 1984 project.h

```
 1  /*
 2   * project wide definitions
 3   *
 4   * Changing this file means large recompiles!  Reconsider!
 5   *
 6   * USED in      src/lib/Debug
 7   *              .../School/T/P
 8   */

10  typedef short   NUM;
11  typedef short   BOOL;

13  /**
14  *** These #defines are used to make stdio code read easier,
15  *** while keeping lint complaints to a minimum.
16  **/
17  #ifdef vax
18  #define VOID    void
19  #else
20  #define VOID    int                     /* void not defined for 8/32 */
21  #endif

23  #define ABORT           (VOID) abort
24  #define FCLOSE          (VOID) fclose
25  #define FFLUSH          (VOID) fflush
26  #define FPRINTF         (VOID) fprintf
27  #define PRINTF          (VOID) printf
28  #define SCANF           (VOID) scanf
29  #define SPRINTF         (VOID) sprintf
30  #define STRCAT          (VOID) strcat
31  #define STRCPY          (VOID) strcpy


34  #define EQ              ==
35  #define NE              !=
36  #define AND             &&
37  #define OR              ||
38  #define NOT             !

40  #define EVER            (;;)            /* for use with "for" loops */
41  #define YES             1
42  #define NO              0
43  #define TRUE            1
44  #define FALSE           0

46  #define SUCCESS         0
47  #define FAILURE         -1

49  #define eqstr( s1, s2 ) ( NOT strcmp ( s1, s2 ) )

51  #define LINE_LEN        132             /* max era line length */
52  #define WORD_LEN        40              /* max era word length */
```

```
1  /*
2   * This include file contains constants related to Input/Output
3   *
4   * USED BY: INPUT module, TABLES module, TRANSLATE module
5   */

7  #define IN_FILE          "P.era.input"          /* input file name   */
8  #define TABLES_FILE      "P.tables"             /* dump of tables    */
9  #define PNL_FILE         "P.pnl.output"         /* translator output */
```

```
1  #define T_ORPHAN                -1      /* see st_orphan()          */
2  #define T_MODE                  0       /* see E_MODE in scan.c     */
3  #define T_FUNCTION              1
4  #define T_I_O_DATA              2
```

```
/*
 * debug.h
 *
 * This file contains the numeric trace levels used by debug().
 *
 * A call to debug() in a module will have the following parameters:
 *      rname   - calling routines name
 *      level   - trace level
 *      format  - printf(3) style format
 *      "args"  - any number of arguments (as described by 'format')
 *
 * NOTE that "level" is a constant (ie: D_TRACE), which will be
 * an offset (ie: 5) from the module's "block" number (ie: D_SCAN).
 * This scheme is used so that a routine may be transparently
 * moved from one module to another.
 *
 * PROJECT:     CS 690, Master's Project, Summer 1984
 *              Requirements Analysis using Petri Nets
 *
 * AUTHOR:      Brad C. Gaylord, LABD
 *              AT&T Information System Laboratories
 *              11900 North Pecos, Denver CO,  80234
 *              drux2!bcg, DR 31L60,  (303)-538-1413
 *
 ************************************************************
 */


/* the D_ERROR trace level is ALWAYS printed.  see fatal_error() */
#define D_ERROR         FAILURE


/* BLOCK numbers - the primary module trace levels.
 *
 * Each module uses a block of up to 10 trace levels.
 *
 * If D_MODULE is not #defined before debug.h is #included,
 * the trace levels will default to the range of 0 to 9.
 * D_MODULE must be #defined so that BEGIN and RETURN can work.
 *
 * Feel free to add #defines for blocks 70, 80 and 90.  See LEVELS.
 * Do not use these constants in your calls to debug().
 */
#define D_NO_MODULE      0
#define D_MAIN          10
#define D_INPUT         20
#define D_SCAN          30
#define D_TABLES        40
#define D_TRANSLATE     50
#define D_INTERPRET     60

/* This definition for D_MODULE is the default block number */
#ifndef D_MODULE
#define D_MODULE        D_NO_MODULE
#endif

/* TRACE levels - the constants used in debug() calls
 *
 * The following serve as offsets from the primary module trace levels.
 * Feel free to add #defines for levels 6-8, if you need them.
 */
#define D_SUB           1 + D_MODULE    /* secondary subroutines  */
#define D_FUN           2 + D_MODULE    /* secondary functions    */
#define D_CRIT          3 + D_MODULE    /* critical trace level   */
#define D_IMP           4 + D_MODULE    /* important trace level  */
#define D_TRACE         5 + D_MODULE    /* ordinary trace level   */
#define D_TRIVIA        9 + D_MODULE    /* trivial details        */

/*
 *  These macros are called in each routine, to allow tracing
 *  of beginnings and returns of routines within a module.
 */
#define BEGIN( s )      char * rname = s; debug ( rname, D_MODULE, "Beginning" )
#define ERROR( s )      fatal_error ( rname, s )
#define RETURN          debug ( rname, D_MODULE, "Returned" ); return

VOID debug(), fatal_error();
```

```
 1  /*
 2   * MODULE:       TABLES
 3   *
 4   * manage the symbol table, the function table, etc
 5   *
 6   * all tables are searched linearly.
 7   * empty table entries are denoted by "NO_VALUE" values.
 8   *
 9   * if a table index HAS_NO_VALUE, then that table is empty.
10   *
11   * USED BY:      SCAN, TRANSLATE, INTERPRET modules
12   *
13   */


16  /* INCLUDE files required by this module: */
17  #include <stdio.h>
18  #include <ctype.h>
19  #include "project.h"                    /* general definitions          */
20  #include "T_tables.h"                   /* entity encodes, etc          */
21  #include "files.h"                      /* file names                   */

23  #define  D_MODULE       D_TABLES        /* primary module trace level */
24  #include "debug.h"                      /* debug levels                 */


27  /* DEFINES local to this module */
28  #define NO_VALUE        FAILURE
29  #define HAS_A_VALUE(n)  ( (NUM) (n) NE NO_VALUE )
30  #define HAS_NO_VALUE(n) ( (NUM) (n) EQ NO_VALUE )

32  /* external functions */
33  char * strcpy();
34  char * strcat();
```

```
/**
*** SUB_MODULE: T_pnltab.i
***
*** in MODULE:  TABLES
***
*** OWNED BY:   T_pnltab.c
***
*** USED BY:    pr_tables()
***
*** CONTAINS:   places[], trans[], pl_idx, tr_idx
**/

/* DEFINEs local to this sub-module: */
#define PLACES_SIZE      100     /* places table size                */
#define TRANS_SIZE       100     /* transitions table size           */
#define OUT_MAX           25     /* max # of outputs                 */
#define IN_MAX            25     /* max # of inputs                  */

#define  P_INIT            0     /* Initial Transition               */
#define  P_FUNC            1     /* functional Transition            */
#define  P_TERM            2     /* Terminal Transition              */


/* global variables "local" to this sub-module: */
extern
struct {                          /* PLACES TABLE                     */
        char sym    [WORD_LEN]; /* place name                       */
        NUM  bag;               /* # of tokens in a place           */
        NUM  out    [OUT_MAX];  /* output transitions               */
        char comment[LINE_LEN]; /* PNL "#" comment                  */
}  places   [PLACES_SIZE];      /* PLACES TABLE                     */

extern
struct {                          /* TRANSITIONS TABLE                */
        char sym    [WORD_LEN]; /* transition name                  */
        NUM  type;              /* type: P_INIT, P_FUNC, P_TERM     */
        NUM  in     [IN_MAX];   /* input transitions                */
        NUM  out    [OUT_MAX];  /* output transitions               */
        char comment[LINE_LEN]; /* PNL "#" comment                  */
        char stim   [LINE_LEN]; /* mode[0] -> nxt_mode iff stim     */
        char stim_nxt_mode;     /* if stim then output nxt_mode     */
        char note   [LINE_LEN]; /* annotation                       */
        char warn   [LINE_LEN]; /* PNL "-" warning                  */
}  trans [TRANS_SIZE];          /* TRANSITIONS TABLE                */

extern
NUM pl_idx;                       /* places table index               */

extern
NUM tr_idx;                       /* transtions table index           */
```

```
/**
*** SUB_MODULE: T_symtab.i
***
*** in MODULE:  TABLES
***
*** OWNED BY:   T_symtab.c
***
*** USED BY:    pr_tables()
***
*** CONTAINS:   symtab[], sym_idx, current, Init_Mode, Cloning, media[]
**/

/* DEFINEs local to this sub-module: */
#define SYMTAB_SIZE     100         /* symbol table size        */
#define  MODE_LIST      "every_mode" /* start of Mode_list      */


/* global variables "local" to this sub-module: */
extern
struct {                                    /* SYMBOL TABLE             */
        char sym [WORD_LEN];                /* symbol name              */
        char sym_suffix [WORD_LEN];         /* symbol name suffix       */
        NUM  sym_class;                     /* symbol class             */
        NUM  sym_idx;                       /* function table           */
        char comment[LINE_LEN];             /* PNL "#" comment          */
}  symtab   [SYMTAB_SIZE];                  /* SYMBOL TABLE             */

extern
struct {                            /* CURRENT ENTITY INFORMATION   */
        NUM  idx;                   /* symtab index                 */
        NUM  type;                  /* T_FUNCTION or T_I_O_DATA     */
}  current;                         /* CURRENT ENTITY INFORMATION   */

extern
NUM sym_idx;                        /* symbol table index           */

extern
NUM Init_Mode;                      /* initial Mode                 */

extern
BOOL Cloning;                       /* cloning on/off flag (st_sym) */

/* array of valid media names; used by T_symfun.c and T_print.c */
extern
char * media []
#ifdef extern                       /* Can not initialize externs:  */
                = {                 /* Allowed types of media       */
        "keyboard",
        "crt",
        "internal",
        "secondary_storage"
}
#endif
        ;                           /* end of media[] declaration   */

#define MEDIA_MAX       4           /* number of entries in media[] */
```

```
/**
*** SUB_MODULE: T_funtab.i
***
*** in MODULE:  TABLES
***
*** OWNED BY:   T_funtab.c
***
*** USED BY:    pr_tables()
***
*** CONTAINS:   funtab[], fun_idx
**/

/* DEFINEs local to this module: */
#define FUNTAB_SIZE     20      /* function table size          */
#define IO_MAX          3       /* max # of in/out relations    */

/* global variables */
extern
struct {                        /* FUNCTION TABLE               */
        NUM  p_or_a;            /* periodic or activity?        */
        NUM  nec_cond;          /* necessary condition          */
        NUM  in[IO_MAX];        /* inputs                       */
        NUM  out[IO_MAX];       /* outputs                      */
        NUM  mode[IO_MAX];      /* required modes               */
        NUM  nxt_mode;          /* next mode                    */
        char stim [LINE_LEN];   /* mode[0] -> nxt_mode iff stim */
        char note [LINE_LEN];   /* annotation                   */
        char warn [LINE_LEN];   /* PNL "-" warning              */
} funtab [FUNTAB_SIZE];         /* FUNCTION TABLE               */

extern
NUM fun_idx;                    /* function table index         */
```

Requirements Analysis Using Petri Nets

by

Bradley C. Gaylord

B. A. Western State College of Colorado, 1978

An Abstract of A Master's Report

submitted in partial fulfillment of the

requirements for the degree

Master of Science

Department of Computer Science

Kansas State University
Manhattan, Kansas

1984

REQUIREMENTS ANALYSIS USING PETRI NETS

by Brad Gaylord

AN ABSTRACT OF A MASTER'S REPORT

This paper suggests an extension to a current-generation requirements language (typically used to express a static data flow relationship via a "human-oriented" form-driven mechanism). This extension will allow a Petri Net analysis to be performed on the requirements specification.

An implementation is described which converts a requirements specification into an augmented textual variant of the Petri Net notation, and then interprets it to find indications of problem areas (such as deadlock and starvation). The implementation will involve the development of an augmented textual Petri Net language, a preprocessor to translate requirements specifications into the Petri Net language and an interpreter to analyze the augmented Petri requirements specification in order to generate reports. While the requirements specification is modified into a form suitable for "prototype execution" and dynamic quality analysis, the reports which are produced still retain their human-oriented aspect. The reports can be generated from various combinations of the viewpoints of control, data and global state transition flow.

The implementation could be extended to incorporate graphical input and output of the specifications, as well as the inclusion of AI techniques. Because of their general applicability, the dynamic analysis methods could be extended for use by similar tools in other phases of the life cycle.