/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:

[Signature]
Major Professor
# TABLE OF CONTENTS

**CHAPTER 1 - INTRODUCTION** ............................................. 1  
1.1 Requirements Specifications ........................................... 1  
1.2 Approaches to Requirements Prototyping ............................. 4  
1.3 Petri Net Simulation Systems .......................................... 5  
1.4 Scope of the Implementation ......................................... 9  

**CHAPTER 2 - REQUIREMENTS** ........................................... 10  
2.1 The Translator .......................................................... 11  
2.2 PNL Intermediate Code ................................................. 26  
2.3 The Interpreter .......................................................... 29  
2.4 Miscellaneous Information .......................................... 32  

**CHAPTER 3 - DESIGN** .................................................. 34  
3.1 Implementation Hierarchy ............................................. 34  
3.2 The Main Module ....................................................... 34  
3.3 The Scanner ............................................................. 35  
3.4 The Translator .......................................................... 43  
3.5 The Interpreter .......................................................... 50  
3.6 Miscellaneous Information .......................................... 53  

**CHAPTER 4 - EXTENSIONS AND CONCLUSIONS** ....................... 54  
4.1 Extensions to this Implementation ................................ 54  
4.2 Conclusion .............................................................. 56  

REFERENCES ............................................................... 58  

**APPENDIX A - BNF GRAMMAR FOR ERA SPECIFICATIONS** ........... 60  
**APPENDIX B - BNF GRAMMAR FOR THIS IMPLEMENTATION** .......... 64  
**APPENDIX C - ERA SPECIFICATION FOR THE CHESS PROCESS** ..... 68  
**APPENDIX D - PETRI NET GRAPH OF THE CHESS PROCESS** ......... 75  
**APPENDIX E - PNL CODE FOR THE CHESS PROCESS** ................. 77  
**APPENDIX F - DIAGNOSTIC AND ERROR MESSAGES** ................ 81  
**APPENDIX G - RUNNING THE IMPLEMENTATION** ..................... 84  
**APPENDIX H - SOURCE CODE LISTINGS** .............................. 87  

- i -
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

Figure 1-1. Petri Net Graph Nodes.......................... 6
Figure 1-2. Petri Net Graph: Two Transitions in Conflict,... 7
Figure 1-3. PNL code corresponding to Fig. 1-2.............. 8
Figure 2-1. Data Flow Diagram of this Implementation....... 10
Figure 2-2. Entity Syntax.................................. 13
Figure 2-3. Example of a Demand Function Entity............ 17
Figure 2-4. Example of a Periodic Function Entity........... 18
Figure 2-5. Petri Net Graph of a Demand Function.......... 23
Figure 2-6. Petri Net Graph of a Periodic Function......... 24
Figure 2-7. PNL code for a Demand Function................. 28
Figure 2-8. PNL code for a Periodic Function............... 29
Figure 3-1. Hierarchy Diagram................................ 35
Figure 3-2. The Scanner's Data Structures..................... 41
Figure 3-3. The Scanner's Data Structures, revisited........ 48
Figure 3-4. The Translator's Data Structures............... 48
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].

- 1 -
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, Tew77, 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 Schess_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 [Ne183].

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](image)

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
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 [Nei82]. 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 [Nei83]. 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 [Kei76].
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 a la prototyping based upon the above Petri Net representation.

* UNIX is a trademark of AT&T Bell Laboratories.
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 [Nei82].

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.

![Diagram of Petri net with Entity line, relation line, and blank line]

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 \texttt{<i_o_data>} class are always delimited by the dollar sign \texttt{\$} 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 \texttt{\$board\_description\$} Input entity. \texttt{\$board\_description\$} is related to the \texttt{"Create\_special\_board" <function> entity, so \texttt{\$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 \texttt{\$status\$} Output entity. \texttt{\$status\$} is related to the \texttt{"Computer\_Move" <function> entity, so \texttt{\$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 \texttt{\$chess\_board\$} and \texttt{\$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 (i.e. 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

\[
\begin{align*}
\text{Activity : } & \text{Recreate_board} \\
\text{input} & : \text{stored_board} \\
\text{output} & : \text{Sname_game} \\
\text{output} & : \text{Sstatus} \\
\text{required_mode} & : \text{"START"} \\
\text{necessary_condition} & : \text{Sretireve} \\
\text{action} & : \text{"START"} \rightarrow \text{"NORMAL"}
\end{align*}
\]

**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: $Sname_game$).

- **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 (i.e. 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 (i.e. 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:

\[
\text{$\text{retrieve}$} : \text{*START*} \rightarrow \text{*NORMAL*}
\]

\[
\begin{array}{c}
\text{*START*} \\
\text{f(2-5)} \\
\text{*NORMAL*}
\end{array}
\]

\[
\begin{array}{c}
\text{$\text{status}$} \\
\text{f(2-5): $\text{chess\_board} = \text{Recreate\_board ( stored\_board )}$}
\end{array}
\]

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 \text{Recreate\_board} entity is annotated by the function \text{f(2-5)}, which is meant to be invoked when the transition is fired. This is because the Input\_output entities \text{stored\_board} and \text{chess\_board} are listed as input and output relations, respectively.

Referring back to the Periodic\_function entity in Figure 2-4, note that \text{Time\_warning\_2} has three input places and two output places. One of the input places is the Input entity \text{limit\_time}, which is the value of the input keyword used in debugging the implementation. The other input places are the required\_modes \text{*NORMAL*} and \text{*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.

\[ \text{*NORMAL*} \quad \text{f(2-6.a)} \quad \text{f(2-6.b)} \quad \text{*ILLEGAL*} \]

\[ \text{f(2-6.a): Time_out_2_-NORMAL- WHEN ( timer > 300 seconds )} \]
\[ \text{f(2-6.b): Time_out_2_-ILLEGAL- WHEN ( timer > 300 seconds )} \]

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 [Ne182, Ne183].

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.
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 (i.e. $statusS$ and $time_warningS$) 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.

- 28 -
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 formatting 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,
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 an 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:

\[ <\text{event}> : \text{old}_\text{Mode} \rightarrow \text{new}_\text{Mode} \]

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

- 39 -
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.
Function Table "FUNTAB"

/* Function Table fields: */
/* Periodic or Activity? */
/* necessary condition */
/* first input */
/* second input */
/* second output */
/* first required mode */
/* next mode */
/* Annotation */
/* not shown: stim & warn */

Symbol Table "SYMTAB"

- Recreate_board
- *NORMAL*
- *START*
- $chess_board$
- $status$
- $name_of_game$
- $stored_board$
- $retrieve$

/* Symbol Table fields: */
/* symbol name */
/* symbol class */
/* function table index */
/* not shown: comment */

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 (i.e. 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:

\[
\text{T_INIT} : \text{INIT OUTPUT TO ( *START* )}
\]

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:

$\text{name_of_game} : \text{PLACE OUTPUT TO (Recreate_board)}$

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*, SretrieveS, $name_of_game$)
INVOKES ( Schess_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 FUNITAB 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.

![Diagram of SYMTAB and FUNTAB tables]

"INVOKES ( $chess_board = Recreation of board ( $stored_board ) )"

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.

![Diagram of entity graph]

"INVOKES ( $chess_board = Recreation of board ( $stored_board ) )"

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*, $retrieveS$ and $name_of_gameS$ 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 (i.e.: 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.
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 [Nie82] 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,

[Boe84] B. W. Boehm, T. E. Gray and T. Seewaldt,
"Prototyping Versus Specifying: A Multiproject Experiment",
IEEE Transactions on Software Engineering,

[Bra82] M. Branstad and M. V. Zelkowicz,
"Special Issue on Rapid Prototyping",
ACM SIGSOFT Software Engineering Notes,

[Chr83] K. Christian,
The UNIX Operating System,

[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,

[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,

[Ke176] R. M. Keller,
"Formal Verification of Parallel Programs",
Communications of the ACM,

[Ker78] B. W. Kernighan and D. M. Ritchie,
The C Programming Language,

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

- 58 -


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.

<table>
<thead>
<tr>
<th>Type</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>Input_output</td>
</tr>
<tr>
<td>Activity</td>
<td>Periodic_function</td>
</tr>
<tr>
<td>Data</td>
<td>Constant</td>
</tr>
<tr>
<td>Comment</td>
<td></td>
</tr>
</tbody>
</table>

* 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> ::= 
   \n
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_B> 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>

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

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

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

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

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

<NL> ::= 
  '
' | '
' <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:

\[
\begin{align*}
\text{<WORD>} & ::= \text{<char>} \\
& \quad | \text{<char> <WORD>} \\
\text{<CAPITAL_WORD>} & ::= \text{<capital_letter> <WORD>} \\
& \quad | \text{<capital_letter>} \\
\text{<LOWER_CASE_WORD>} & ::= \text{<lower_case_char> <WORD>} \\
& \quad | \text{<lower_case_char>} \\
\text{<char>} & ::= \text{<capital_letter>} \\
& \quad | \text{<lower_case_char>} \\
& \quad | \text{<symbol>} \\
\text{<lower_case_char>} & ::= \text{a} | \text{b} | \ldots | \text{z} | 0 | 1 | \ldots | 9 \\
\text{<symbol>} & ::= \text{#} | \text{%} | \text{&} | ( | ) | ? | _ \\
\text{<capital_letter>} & ::= \text{A} | \text{B} | \ldots | \text{Z} \\
\end{align*}
\]

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 directly related to a command which the user of the chess game might enter. (As opposed to Input 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 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', '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 representation of the standard 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 `$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 unifiable.
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','conceded'} 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 )

- 75 -
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 ( Schess_board$ = Initialize_board ( ) )

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

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

- 78 -
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 ( $syntax_error$ )

- 80 -
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 "FUNCTAB"

----------------------------- fatal ERRORS in scan.c
ERROR ( "FUNCTION names begin with a capital letter" );
   for example: Recreate_board

ERROR ( "I_O_DATA names are delimited by 'S' 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 precede 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 'x' 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)" );
ddebug ( 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:

# Role
   - f P.t*
   P
Enter 't' for a Help message

* Enter one of the following:
  q - quit this session
  ( or ) - fire a transition
  a - change Interpreter options
  d - dump the Petri net
  ? - print this help message

* Enter an option, or '?', for help: t

Trace 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_Queue #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

t =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_Queue #NORMAL# : WHEN ( user input does not match syntax )

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

- 84 -
Enter the number of the transition to be fired: 0
FIRING menu item # 0:
Computer_Move : if $status$ in ('checkmate', 'concede') then *END*
INVOCKES ( $chess_board$ = Computer_Move ( $chess_board$ ) )

FIRING TERMINAL transitions:
T_TERM_fcomputer_move_message$ : INPUT FROM ( $computer_move_message$ )
T_TERM_fstatus$ : INPUT FROM ( $status$ )
> 0: Computer_Move : if $status$ in ('checkmate', 'concede') then *END*
INVOCKES ( $chess_board$ = 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*
INVOCKES ( $chess_board$ = Computer_Move ( $chess_board$ ) )

FIRING TERMINAL transitions:
T_TERM_fcomputer_move_message$ : INPUT FROM ( $computer_move_message$ )
T_TERM_fstatus$ : INPUT FROM ( $status$ )
> 0: Computer_Move : if $status$ in ('checkmate', 'concede') then *END*
INVOCKES ( $chess_board$ = 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*
INVOCKES ( $chess_board$ = Computer_Move ( $chess_board$ ) )

FIRING TERMINAL transitions:
T_TERM_fcomputer_move_message$ : INPUT FROM ( $computer_move_message$ )
T_TERM_fstatus$ : INPUT FROM ( $status$ )
> 0: Computer_Move : if $status$ in ('checkmate', 'concede') then *END*
INVOCKES ( $chess_board$ = 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*
INVOCKES ( $chess_board$ = Computer_Move ( $chess_board$ ) )

FIRING TERMINAL transitions:
T_TERM_fcomputer_move_message$ : INPUT FROM ( $computer_move_message$ )
T_TERM_fstatus$ : INPUT FROM ( $status$ )
> 0: Enter an option, or '?' for help; "false input"
Enter '?' for a Help message
Enter an option, or '?' for help.
Starvation defeat ON
Enter an option, or '?' for help:
Returning to Interpreter
>:

0: Computer_Move : if $status$ in ('checkmate', 'concede') then *END*
1: Store_board : INVOCKES ( Store_board ( $chess_board$ ) )
2: Validate_user_move : if $move_message$ in ('illegal_move') then *ILLEGAL*
INVOCKES ( Validate_user_move ( $chess_board$ ) )
3: Computer_Move : if $status$ in ('checkmate', 'concede') then *END*
INVOCKES ( $chess_board$ = Computer_Move ( $chess_board$ ) )
4: Update_board : INVOCKES ( $chess_board$ = Update_board ( $chess_board$ ) )
5: Display_board*$NORMAL* : INVOCKES ( Display_board ( $chess_board$ ) )
6: Stealmate : 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* : INVOCKES ( Display_board ( $chess_board$ ) )

FIRING TERMINAL transitions:
T_TERM_fboard_display$ : INPUT FROM ( $board_display$ )
>0
GOODbye,
#
The following is a sample run of the implementation:

```
rm -f P.t

Enter '?' for a Help message
>?
Enter one of the following:
q  - quit this session
<cr> - fire a transition
g  - 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 ( $retrieve_message$ )
```

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 ( $status$ )

Enter an option, or '?' for help: o
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 END
    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

Bye bye.

- 86 -
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_symfunc.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_printtab.i T_symtab.i T_funtab.i
SRCS = main.o scan.o translate.o interpret.o
      T_symtab.o T_symfun.o T_funtab.o T_print.o input.o debug.c
OBJS = main.o scan.o translate.o interpret.o

# ALL PROJECT SPECIFIC INFORMATION PRECEDES THIS LINE
# THE REMINDER OF THIS FILE IS PURE GENERIC Makefile
CFLAGS = -g
TMP = /usr/tmp/bcg.
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 -oacS > $(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 awk -F ' \"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 nonhole 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 kb kill & echo
# Commands to set up header dependencies automatically in Makefile:
# grep outputs lines from 1 or more C language files like this:
# foo.c: #include "bar.h"
# These lines (and NOT lines like "#include <stdio.h>") become:
# foo.o: bar.h
# These resulting lines are put at the end of Makefile.
# NOTE: use VPATH (defined in .profile) if header files are not in ".":
# VPATH = :/HOME/usr/include:/PROJ/include ; export VPATH

dependencies : .save
  .echo "Recreating Makefile dependencies..."
  .cp Makefile .save/OMakefile
  .echo '/** HEADER DEPENDENCIES FOLLOW/+1,$d\nw/ | ed Makefile
  .echo '/* ' >> Makefile
  .grep '.*include' $(SOURCES) $(HDRS) $(INCS) /dev/null |
    sed 's/\([^a-zA-Z0-9_][a-zA-Z0-9_]*/\)/\1/; t1/'
    sed 's/</[^>]*/\)/; t1/'
    sed 's/\([^a-zA-Z0-9_][a-zA-Z0-9_]*/\)/\1/; t1/'
  .sort >> Makefile
  .echo '# DO NOT PUT ANYTHING BELOW THIS LINE!' >> Makefile

# HEADER DEPENDENCIES FOLLOW:
# Tue Nov 20 16:58:38 MST 1984
T_functab.o: T_functab.h
T_print.o: T_printtab.h
T_print.o: T_cpubtab.h
T_print.o: T_symtab.h
T_print.o: T_tables.h
T_symfun.o: T_functab.h
T_symfun.o: T_symtab.h
T_symfun.o: T_tables.h
T_symtab.o: T_symtab.h
T_tables.o: T_tables.h
tables.o: debug.h
T_tables.h: files.h
T_tables.h: project.h
dbg.o: project.h
input.o: debug.h
input.o: project.h
interpret.o: T_functab.h
interpret.o: T_printtab.h
interpret.o: T_symtab.h
interpret.o: T_tables.h
dbg.o: debug.h
main.o: debug.h
main.o: files.h
main.o: project.h
scan.o: T_tables.h
can.o: debug.h
scan.o: project.h
translate.o: T_functab.h
translate.o: T_printtab.h
translate.o: T_symtab.h
translate.o: T_tables.h
dbg.o: debug.h

# 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 */
CMSC 680 - Master's Project, Summer 1984 */

/* AUTHOR: Brad C. Gaylord */
AT&T Information System Laboratories */
11900 North Pecos, Denver CO. 80234 */
dru2@bcg. OR 3160, (303)-598-1413 */

/* USED BY: all modules */
/* ACCESS FCTS: init(), fini() */

/* USES: TRANSULATE, INTERPRET */

/* DATA OWNED: */

*/

/* 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 */
#define "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();
83     /* initialize data structures and data files */
84     init ();
85     /* translate ERA spec into PNL entity graph */
86     translate ();
87     /* interpret the Petri net structure */
88     interpret ();
89     /* cause successful process termination */
90     fini ( SUCCESS );
91 } /* main () */
/*
  */
 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 Imp_init(), st_init(), pnl_init();
120    BEGIN ( "init" );
121    Imp_init ( IN_FILE );  /* open input files */
122    st_init ();          /* initialize symbol tables */
123    pnl_init ();         /* initialize PNL tables */
124    RETURN;
125  } /* init () */
\*  
\*  " fini ()  
\*  " Orderly shutdown of the system.  
\*  " This is the ONLY routine to cause process exit.  
\*  " once the process is initialized.  
\*  " INPUT:  
\*  " called by: main ()  
\*  " input parameters: status (SUCCESS OR FAILURE)  
\*  " OUTPUT:  
\*  " return type & value: integer return code to sh(1)  
\*  " GLOBAL:  
\*  " variables used: n/a  
\*  " variables changed: n/a  
\*  "  
\*  VOID 
\*  fini ( status )  
\*  int status;  
\*  {  
\*  VOID pr_tables(), Inp_fini(), Dbug_fini(), sync(), exit();  
\*  BEGIN ( "fini" );  
\*  pr_tables (); /* print tables */  
\*  Inp_fini (); /* close input files */  
\*  Dbug_fini (); /* close debugging */  
\*  sync (); /* write memory to disk */  
\*  if ( status eq SUCCESS ) (  
\*   exit ( status ); /* successful process: */  
\*  } else {  
\*    ABORT (); /* dump core for sdb */  
\*  }  
\*  } /* fini () */
/* INCLUDE files required by this module: */
#include <stdio.h>    /* for EOF */
#include <ctype.h>    /* for isupper() */
#include "project.h"  /* general definitions */
#include "ltables.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
/* To add new entries to valid_entity[] or valid_relation[], put the 
* new string at the end of the appropriate list, and add the 
* corresponding #define index. Increment the LAST_ENTRY #define. 
* Then find the switch statements which use the class and type 
* fields and add the code for the new cases. 
*/

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

/* the following #defines represent indices into valid_entity[] */
#define E_MODE _T_MODE /* 0 */
#define E_ACTIVITY 1
#define E_PERIODIC_FUNCTION 2
#define E_INPUT 3
#define E_OUTPUT 4
#define E_INPUT_OUTPUT 5
#define E_LAST_ENTRY 6

char *valid_entity[] = {
    "Mode",
    "Activity",
    "Periodic_function",
    "Input",
    "Output",
    "Input_output"
};

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

/* the following #defines represent indices into valid_relation[] */
#define R_NECESSARY_CONDITION 0
#define R_INPUT 1
#define R_OUTPUT 2
#define R_REQUIRED_MODE 3
#define R_OCCURRENCE 4
#define R_MEDIA 5
#define R_LAST_ENTRY 6

char *valid_relation[] = {
    "necessary_condition",
    "input",
    "output",
    "required_mode",
    "occurrence",
    "media"
};

BOOL ent_ok = ND; /* not an interesting entity yet... */
char *era_title [WORD_LEN]; /* title of this PROCESS */
char *era_line; /* points to current line */
114 /*
115 *  is_fun_name (),  is_ioc_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 - rns 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 capital letter" );
145     }
146 } /* is_fun_name() */

151 VOID
152 is_ioc_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 */
157         if ( (sym[0] NE ' ') AND (sym[strlen(sym)] NE ' ' ) )
158             debug( rname, D_ERROR, "Invalid name: %s", sym );
159         ERROR( "I/O/DATA names are delimited by ' ' signs" );
160     }
161 } /* is_ioc_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 */
173         if ( (sym[0] NE ' ') AND (sym[strlen(sym)] NE ' ' ) )
174             debug( rname, D_ERROR, "Invalid name: %s", sym );
175         ERROR( "MODE names are delimited by ' ' signs" );
176     }
177 } /* is_mode_name() */
/*
 * scan ()
 */

/* This scanner follows the approach suggested in Appendix B:
 * - First a PROCESS line must be identified.
 * - null lines & comments are removed
 * - the ordering of input lines is checked by line_syntax()
 * - scan() returns when a MODE_TABLE is encountered and processed.
 */

/* INPUT:
 * called by: translate()
 *
 * OUTPUT:
 * return type & value: VOID
 *
 * GLOBAL:
 * variables used: ent_ok
 * variables changed: era_line
 */

VOID

void scan ()

{
  NUM process_line();  /* read the era PROCESS line */
  char *get_line();    /* get an era input line */

  BOOL **line();       /* removes non-useful lines */
  NUM line_syntax();   /* check syntax of era lines */
  VOID mode_table();   /* process the mode_table */
  VOID st_orphan();    /* find syntax orphans */
  VOID st_invokes();   /* resolve annotations */

  char lhs [ WORD_LEN ];  /* left hand side of line */
  char rhs [ WORD_LEN ];  /* right hand side of line */
  NUM s_count;           /* number of scanf matches */
  NUM line_type;         /* current line type */

  BEGIN ( "scan" );

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

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

    /* break when the MODE_TABLE is encountered */
    if ( strcmp ( era_line, "MODE_TABLE" ) ) {
      line_type = MODE_TABLE;
      break;
    }

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

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

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

    debug ( rname, D_TRACE, \n );

   /* while */
  }

  if ( line_type != MODE_TABLE ) {
    ERROR ( "MODE_TABLE not encountered" );
  } else {
    mode_table ();        /* process the mode_table */
    st_orphan ( E_INPUT, E_OUTPUT ); /* find syntax orphans */
    st_invokes ( E_INPUT_OUTPUT );  /* resolve annotations */
  }

  RETURN;
}

/* scan () */
250 /*
251 * process_line()
252 * scan the first line in the era file for a PROCESS line.
253 * INPUT: called by: scan()
254 * OUTPUT: return type & value: line_type = AFTER_PROCESS_LINE
255 * GLOBAL: variables used: n/a
256 * variables changed: era_line, era_title
257 */

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 ( scanf ( era_line, "PROCESS : %s", era_title ) == 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() */
/*
 * line_syntax ()
 * Each line is classified as a BLANK_LINE, an ENTITY line or a
 * RELATION line. This code tries to ensure that the following
 * syntax is adhered to: BLANK_LINES precede ENTITY lines (which
 * begin with a capital letter in column 1), and ENTITY lines precede
 * one or more RELATION lines (lower-case & not beginning in column 1).
 * line_syntax () calls entity () to process ENTITY lines. Only those
 * entities which are of interest to this process (ent_ok == YES)
 * are examined in greater detail by relation ()..
 */

/* INPUT: called by: scan ()

/* OUTPUT:
 */
return type & value: line_type = current line's type

/* GLOBAL:
 variables used: ent_ok, prev_type
 variables changed: ent_ok

/*

line Syntax ( prev_type, line, lhs, rhs )
NUM prev_type;
char * line; /* previous line type */
char * lhs; /* left-hand side of line */
char * rhs; /* right-hand side of line */

{ BOOL entity(); /* identify interesting entities */
 VOID relation(); /* process relations of said entities */

NUM ret;

BEGIN ( "line_syntax" );

generate a hypothesis as to the identity of the line */
/* based upon column 1 and the previous line's type */
switch ( prev_type ) {
 case BLANK_LINE:
  "entity lines follow blank lines "
  if ( isupper ( line [0] ) ) {
    ret = ENTITY;
  } else {
    ERROR ( "illegal: <NL> <relation_line>" ) ;
  }
 break;

 case ENTITY:
 case RELATION:
  "relation lines follow these 2 kinds of line. "
  "entity lines follow neither kind of these. "
  if ( isupper ( line [0] ) ) {
    ERROR ( "Entity must follow blank line" ) ;
  } else {
    if ( ent_ok ) {
  "examine the relations "
    relation ( lhs, rhs );
  }
  ret = RELATION;
 break;

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

RETURN ( ret );

} /* line_syntax () */
*/
/*
 * entity()
 * if this is an interesting entity line, enter it into the symtab.
 * INPUT:
 * called by: scan()
 * OUTPUT:
 * return type & value: BOOL YES - this entity is useful
 * NO - this entity is not useful
 * GLOBAL:
 * variables used: valid_entity[]
 * variables changed: n/a
 */

BOOL entity ( lhs, rhs )
char " lhs;
char " rhs;

VOID st_entity(); /* adds entities to the symbol table */
NUM is_f_or_i(); /* T_FUNCTION or T_J_0_DATA? */

NUM class;
BEGIN ( "entity" );

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

  /* add entity name to symtab */
  st_entity ( rhs, class, is_f_or_i ( class ) );
  RETURN (YES);
}
/* for */
RETURN (NO);

} /* entity () */
/*
 * relation ()
 * is this an interesting relation line?
 * if so, populate the table corresponding to this relation class
 * (FUNCTION or I_O_DATA)
 * NOTE: no check is made for lowercase only in relation names;
 * different entities can't share any entity relation types
 *
 * INPUT:
 * called by: scan ()
 *
 * OUTPUT:
 * return type & value: VOID
 *
 * GLOBAL:
 * variables used: valid_relation[]
 * variables changed: n/a
 */

VOID relation ( lhs, rhs )
char * lhs;
char * rhs;
{ VOID st_rec_cond(), st_input(), st_output(), st_rec_mode();
  VOID st_note(), st_media(), st_warn();
  NUM is_p_or_a();

  register class;

BEGIN ("relation");

for (class=0; class < R_LAST_ENTRY; class++) {
    if (NOT eastr (lhs_valTd_relation[class])) {
        /* ignore this keyword: it's for someone else */
        continue;
    }

    switch (class) {
    case R_NECESSARY_CONDITION:
        if (is_D_or_a() NE E_ACTIVITY) {
            ERROR ("only used in Activity entities");
            st_nec_cond (rhs);
            break;
        }
    case R_INPUT:
        if not Input_output_type */
        if (is_D_or_a() EQ E_PERIODIC_FUNCTION) {
            EE_warn ("don't use 'input' relations in Periodic_functions");
            st_input (rhs);
            break;
        }
    case R_OUTPUT:
        st_output (rhs);
        break;
    case R_REQUIRED_MODE:
        st_req_mode (rhs);
        break;
    case R_OCCURRENCE:
        if (is_D_or_a() NE E_PERIODIC_FUNCTION) {
            ERROR ("only used in Periodic_function entities");
        }
        st_note (era_line);
        break;
    case R_MEDIA:
        /* caveat: no verification done on device type */
        /* as it relates to IO_DATA type */
        st_media (rhs);
        break;
        default:
        debug (rname, D_ERROR, "class=%d", class);
        ERROR ("internal error");
    }
    /* switch */

RETURN;
} /* for */

RETURN;
/* 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 */
524
525 VOID
526 mode_table ()
527 {
528  char * get_line(); /* get an era input line */
529  VOID st_Verify(); /* verify modes */
530  VOID st_l_mode(); /* store Initial_Mode */
531  VOID st_mult_modes(); /* clone duplicate Modes */
532  VOID st_transitions(); /* generate nxt_modes */
533  char lhs [ WORD_LEN ]; /* left hand side of line */
534  char io_val[ WORD_LEN ]; /* I/O_DATA value */
535  char mode1 [ WORD_LEN ]; /* mode: current mode */
536  char mode2 [ WORD_LEN ]; /* mode: next mode */
537  NUM s_count; /* number of sscanf matches */
538  NUM line_type; /* current line type */
539  NUM mode_state; /* which 1/2 of mode table? */
540  */
BEGIN ( "mode_table" );

mode_state = MODE_TABLE;

while ( era_line = get_line() ) {
  switch ( mode_state ) {
  case MODE_TABLE:
    lns[0] = mode1[0] = NULL;
    s_count = sscanf ( era_line, "%s : %s\n\n%ls. mode1\n\n/* remove comments and non-useful lines */
    if ( null_line ( rname, &line_type, s_count, lns ) ) {
      continue; /* with next era_line in the while loop */
    }
    if ( eqstr ( lns, valid_entity[E_MODE] ) ) {
      st_verify ( mode1 );
    } else if ( eqstr ( lns, "Initial_Mode\n\n    st_i_mode ( mode1 );
    /* all of the modes have been read */
    st_mult_modes ( );
    mode_state = MODE_INITIAL;
    break;
  }
  case MODE_INITIAL:
    lns[0] = NULL;
    s_count = sscanf ( era_line, "%s\n\n\n/* remove comments and non-useful lines */
    if ( null_line ( rname, &line_type, s_count, lns ) ) {
      continue; /* with next era_line in the while loop */
    }
    if ( eqstr ( lns, "Allowed_Mode_Transitions\n\n      /* process the last part of the MODE_TABLE */
      mode_state = MODE_TRANSITIONS;
      continue; /* with next era_line in the while loop */
    }
    break;
  case MODE_TRANSITIONS:
    lns[0] = io_val[0] = mode1[0] = mode2[0] = NULL;
    s_count = sscanf ( era_line, "%s : %s\n\n\n    if ( s_count EQ 1 ) {
      s_count = sscanf ( era_line, "%s = %s : %s\n\n\n      lns, io_val, model1, model2 );
    }
    /* remove comments and non-useful lines */
    if ( null_line ( rname, &line_type, s_count, lns ) ) {
      continue; /* with next era_line in the while loop */
    }
    st_transitions ( lns, io_val, model1, model2 );
    break;
      default:
      debug ( rname, D_ERROR, "mode_state=\%d", mode_state );
      ERROR ( "Internal error" );
    } /* switch */
  }
  /* while */
  RETURN;
} /* 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 * "op" 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 *              ND - process this line
630 *
631 * GLOBAL:
632 * variables used: n/a
633 * variables changed: line_type (in scan())
634 */
635
636 BOOL
637 null_line ( rname, line_type_addr, s_count, lhs )
638 char * rname;
639 NUM * line_type_addr;
640 NUM s_count;
641 char * lhs;
642 {
643     /* test for a blank line or "op" */
644     if (((s_count EQ EOF) OR eqstr (lhs, ".op") )
645         && ( "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
" );
651         return ( YES );
652     }
653 }
654 /
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     }
661 }
662 return ( ND );
663 }
664 } /* null_line () */
/*
 * is_f_or_i ( )
 * return one of the primary table type encodes
 * INPUT:
 * called by: TABLES module
 * OUTPUT:
 * return type & value: T_MODE - E_MODE & ERRORS in class
 * T_FUNCTION - Activities, Periodics
 * T_IO_DATA - Inputs, Outputs, Input_outputs
 * GLOBAL:
 * variables used: n/a
 * variables changed: n/a
 */

NUM is_f_or_i ( class )

NUM class; /* class of symtab entry */

{    
    NUM f_or_i:

    switch ( class ) {
        case E_MODE:
            f_or_i = T_MODE;
            break;

        case E_ACTIVITY:
            case E_PERIODIC_FUNCTION:
                f_or_i = T_FUNCTION;
                break;

        case E_INPUT:
        case E_OUTPUT:
            case E_INPUT_OUTPUT:
                f_or_i = T_IO_DATA;
                break;

        default:
            /* This strange error return is used to prevent */
            /* switch statements from blowing up in pr_entity(). */
            /* This is called from fini. Blow ups cause */
            /* switch defaults to call ERROR which calls fini. */
            /* This indirect recursion is not useful. */
            /* ERROR ("internal error"); */
            } /* switch */

    return ( f_or_i );

    } /* is_f_or_i ( ) */
/*
 pr_class ()

 * print the class of an entity stored in the symbol table.
 * class encodes are "owned" by the SCAN module

 * INPUT:
 * called by: pr_entity()

 * OUTPUT:
 * return type & value: VOID

 * GLOBAL:
 * variables used: n/a
 * variables changed: n/a
 */

VOID pr_class ( fp, class )
FILE "fp: /* file pointer for output */
NUM class: /* class of symtab entry */
{
BEGIN ( "pr_class" );

PRINTF ( fp, "class : " );

switch ( class ) {
  case T_ORPHAN:
    PRINTF ( fp, "ORPHAN\n" );
    break;

  case E_MODE:
  case E_ACTIVITY:
  case E_PERIODIC_FUNCTION:
  case E_INPUT:
  case E_OUTPUT:
  case E_INPUT_OUTPUT:
    PRINTF ( fp, "%s\n", valid_entity[class] );
    break;

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

RETURN:

} /* pr_class () */

- 107 -
/* 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
dru2@bgc. DR 31260, (303) 598-1413

--------------------------------------------------------------------------

USED BY: main

ACCESS FCT'S: translate()

--------------------------------------------------------------------------

USES: TABLES

DATA OWNED: places[], trans[], pl_idx, tr_idx (see T_pn1tab[c])

--------------------------------------------------------------------------

.Gray files required by this module: */
#include "T_tables.i"     /* general table definitions */
#include "T_symtab.i"     /* symbol table */
#include "T_pn1tab.i"     /* function table */
#define extern             /* TRANSLATE owns T_pn1tab; */
#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: */
/*
 * translate ()
 * Translate the symbol table into place and transitions
 * and generate PNL code.
 *
 * INPUT:
 * called by:       main()
 *
 * OUTPUT:
 * return type & value: VOID
 *
 * GLOBAL:
 * variables used:    n/a
 * variables changed: n/a
 */

VOID translate ()
{
    VOID scan(), p_nodes(), p_pnl();
    BEGIN ("translate");
    /* fill the symbol and function tables */
    scan();
    /* create the Places and Transitions */
    p_nodes();
    /* output the PNL code */
    p_pnl();
    RETURN;
} /* translate () */
VOID p_nodes (
{
    VOID tr_out();
    NUM is_f_or_i(), p_place(), p_transition();
    NUM idx;     /* iterator */
    NUM p_idx;   /* place table index */
    NUM t_idx;   /* transition table index */

    BEGIN ( "p_nodes" );
    /* store the Place that the initial Transition outputs to */
    p_idx = p_place ( Init_Mode );
    /* store the initial Transition and it's output index */
    t_idx = p_transition ( NULL, P_INIT );
    tr_out ( t_idx, p_idx );
    /* create the rest of the Places and Transitions */
    for ( idx=0; idx<sym_idx; idx++ ) {
        if ( is_f_or_i[symtab[idx].sym_class] != T_FUNCTION ) {
            continue;
        }
        /* put the function in the transition table */
        (VOID) p_transition ( idx, P_FUNC );
    }
    RETURN;
} /* p_nodes () */
/*
 * p_pni ()
 * write the PNL code to an output file
 * INPUT:
 * called by: translate()
 * input parameters: n/a
 * OUTPUT:
 * return type & value: VOID
 * GLOBAL:
 * variables used: PNL_FILE
 * variables changed: n/a
 */

VOID
p_pni ()
{
    VOID perror();
    VOID pr_initial(), pr_places(), pr_transitions(), pr_terminals();
    FILE * Pnl_fp; /* PNL_FILE */
    BEGIN ( "p_pni" );

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

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

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

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

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

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

    RETURN;
}

/* p_pni () */
/*
  * p_place ()
  * Add a symbol to the places table if it is not already there.
  * Return the index to the symbol's entry in the table.
  *
  * INPUT:
  * called by: p_nodes(), p_transitions()
  * input parameters: symbol table index
  *
  * OUTPUT:
  * return type & value: current table index
  *
  * GLOBAL:
  * variables used: symtab[]
  * variables changed: places[], pl_idx

  */

NUM p_place ( st_idx )
NUM st_idx:
{
  NUM pl_find();
  NUM idx;
  BEGIN ( "p_place" ):

    if ( HAS_NO_VALUE ( idx=pl_find(syntab[st_idx].sym) ) ) {
      If ( pl_idx < PLACES_SIZE-1 ) {
        idx = ++pl_idx;
      } else {
        ERROR ( "places table overflow" );
    }

    /* put the symbol name in the places table */
    STRCpy ( places[idx].sym, syntab[st_idx].sym );

    if ( HAS_A_VALUE ( syntab[st_idx].comment[0] ) ) {
      STRCpy ( places[idx].comment, syntab[st_idx].comment );
    }

    RETURN ( idx );

} /* p_place () */
*/
243  * p_transition ()
244  *
245  * Add a symbol to the transitions table if it is not already there.
246  *
247  * Return the index to the symbol's entry in the transition table.
248  *
249  *
250  *
251  * INPUT:
252  * called by: p_nodes(), p_transitions()
253  *
254  * input parameters: symbol table index, and transition type
255  *
256  *
257  * OUTPUT:
258  * return type & value: current table index
259  *
260  *
261  **GLOBAL:
262  * variables used: symtab[], funtab[]
263  *
264  * variables changed: trans[], tr_idx, places[]
265 */
266
267 NUM p_transition ( st_idx, type )
268 NUM st_idx;
269 NUM type;
270 /* transition type: INIT, FUNC, TERM */
271 { void tr_in(), tr_out();
272
273 void tr_find();
274 NUM f_idx; /* function table index */
275 NUM t_idx, out_idx; /* transition table indices */
276 NUM i; /* iterator */
277 char sym [WORD_LEN]; /* transition name */
278
279 BEGIN ( "p_transition" );
280
281 switch ( type ) {
282 case P_INIT:
283     SPRINTF ( sym, "T_INIT" );
284     break;
285
286 case P_FUNC:
287     SPRINTF ( sym, "%%s", symtab[st_idx].sym,
288              HAS_A_VALUE (symtab[st_idx].sym_suffix[0])
289             ? symtab[st_idx].sym_suffix : NULL ) ;
290     break;
291
292 case P_TERM: /* should be OUTPUT name! */
293     SPRINTF ( sym, "T_TERM%%s", symtab[st_idx].sym );
294     break;
295
296 default:
297     debug ( rname, D_ERROR, "type=%%s", type );
298            ERROR ( "internal error" );
299        } /* switch */
300
301 if ( HAS_NO_VALUE ( t_idx=tr_find(sym)) ) {
302    if ( t_idx < PLACE_SIZE-1 ) {
303        t_idx = ++t_idx;
304    } else {
305        ERROR ( "trans table overflow" );
306    }
307    /* put the symbol name and type in the trans table */
308    strcpy ( trans[t_idx].sym, sym );
309    trans[t_idx].type = type;
310 }
311
312 if ( type NE P_FUNC ) {
313    RETURN ( t_idx );
314  }
" add the information found in the symbol & function tables ";

f_idx = symtab[st_idx].sym_idx;

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

for ( i=0; i<IO_MAX; i++ ) {
    if ( HAS_A_VALUE (funtab[f_idx].in[i]) ) {
        tr_in ( t_idx, p_place(funtab[f_idx].in[i]));
    }
    if ( HAS_A_VALUE (funtab[f_idx].out[i]) ) {
        tr_out ( t_idx, p_place(funtab[f_idx].out[i]));
        /* establish TERMINAL transition */
        /* for this output entity */
        out_idx = p_transition ( funtab[f_idx].out[i], P_TERM );
        tr_in ( out_idx, p_place(funtab[f_idx].out[i]));
    }
}

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

if ( HAS_A_VALUE (funtab[f_idx].nxt_mode) AND
     HAS_NO_VALUE (funtab[f_idx].stim[0]) ) {
    tr_out ( t_idx, p_place(funtab[f_idx].nxt_mode));
} else {
    tr_out ( t_idx, p_place(funtab[f_idx].mode[0]));
}

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

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

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

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

RETURN ( t_idx );

" 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 */
385
386 NUM pl_find ( sym )
387 char * sym;
388 {
389  NUM idx;
390  BEGIN ( "pl_find" );
391  for ( idx=0; idx<=pl_idx; idx++ ) {
392    if ( eqstr ( sym, places[idx].sym ) ) {
393      RETURN ( idx );
394    }
395  }
396  RETURN ( NO_VALUE );
397  } /* pl_find () */
398
400 NUM tr_find ( sym )
401 char * sym;
402 {
403  NUM idx;
404  BEGIN ( "tr_find" );
405  for ( idx=0; idx<=tr_idx; idx++ ) {
406    if ( eqstr ( sym, trans[idx].sym ) ) {
407      RETURN ( idx );
408    }
409  }
410  RETURN ( NO_VALUE );
411  } /* tr_find () */
/*
   p1_out()
   fill: the values for output places for a place
   INPUT:
   called by: tr_in()
   input parameters: indices to places[] and trans[]
   OUTPUT:
   return type & value: VOID
   GLOBAL:
   variables used: n/a
   variables changed: places[].out[]
 */

VOID
p1_out( p_idx, t_idx )
NUM p_idx;
NUM t_idx;
{
    NUM i;

    BEGIN ( "p1_out" ):
        for ( i=0; i<OUT_MAX; i++ ) {
            if ( places[p_idx].out[i].EQ t_idx ) {
                RETURN; /* already an output place */
            }

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

            if ( i.EQ_OUT_MAX-1 ) {
                /* all of the output pointers consumed */
                ERROR ( "max # of output indices = 25" );
            }
        }

    RETURN;

} /* p1_out() */

/*
 * tr_in()
 * fill the values for input places for a transition
 * INPUT:
 * called by: p_init(), p_transition()
 * input parameters: indices to tran[] and places[]
 * OUTPUT:
 * return type & value: VOID
 * GLOBAL:
 * variables used: n/a
 * variables changed: tran[].in[]
 */

VOID
tr_in( t_idx, p_idx )
NUM t_idx;
NUM p_idx;

{ VOID pl_out();

NUM i;

BEGIN ( "tr_in" );

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

  if ( HAS_NO_VALUE (tran[t_idx].in[i]) ) {
    tran[t_idx].in[i] = p_idx;
    break;
  }

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

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

  RETURN;

} /* tr_in() */
/** 
 * tr_out () 
 * fill the values for output places for a transition 
 * 
 * INPUT: 
 * called by: p_init(), p_transition() 
 * input parameters: indices to trans[] and places[] 
 * 
 * OUTPUT: 
 * return type & value: VOID 
 * 
 * GLOBAL: 
 * variables used: n/a 
 * variables changed: trans[].out[] 
 */ 

VOID tr_out ( t_idx, p_idx ) 
NUM t_idx; 
NUM p_idx; 
{
    NUM i;

    BEGIN ( "tr_out" );

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

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

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

    RETURN;
}

"/ tr_out ()"
/*
 * pl_add_bag (), pl_del_bag ()
 * add or delete tokens from the bag of a place
 *  
 * INPUT: 
 * called by: INTERPRET module
 *  
 * OUTPUT: 
 * return type & value: VOID 
 *  
 * GLOBAL: 
 * variables used: n/a 
 *  
 * variables changed: places[].bag 
 */

VOID
pl_add_bag ( p_idx )
NUM p_idx;
{
    BEGIN ( "pl_add_bag" );
    places[p_idx].bag++;
    RETURN;
} /* pl_add_bag () */

VOID
pl_del_bag ( p_idx )
NUM p_idx;
{
    BEGIN ( "pl_del_bag" );
    if ( places[p_idx].bag ) {
        places[p_idx].bag--;
    }
    RETURN;
} /* pl_del_bag () */
" pni_init ()
  initialize the places[] and trans[] tables.
*/

  INPUT:
  called by: init()

  OUTPUT:
  return type & value: VOID

  GLOBAL:
  variables used: n/a
  variables changed: places[], trans[], pl_idx, tr_idx
*/

VOID pni_init ()
{
  NUM i, j;

  BEGIN ( "pni_init" );

  /* initialize the places table */
  for ( i=0; i<PLACES_SIZE; i++ ) {
    places[i].sym[0] = NO_VALUE;
    places[i].sym[1] = NO_VALUE;
    for ( j=0; j<OUT_MAX; j++ ) {
      places[i].out[j] = NO_VALUE;
      places[i].comment[0] = NO_VALUE;
    }
  }

  /* initialize the transitions table */
  for ( i=0; i<TRANS_SIZE; i++ ) {
    trans[i].sym[0] = NO_VALUE;
    trans[i].sym[1] = NO_VALUE;
    for ( j=0; j<IN_MAX; j++ ) {
      trans[i].in[j] = NO_VALUE;
    }
    for ( j=0; j<OUT_MAX; j++ ) {
      trans[i].out[j] = NO_VALUE;
      trans[i].comment[0] = NO_VALUE;
      trans[i].stim[0] = NO_VALUE;
      trans[i].stim_nxt_mode = NO_VALUE;
      trans[i].note[0] = NO_VALUE;
      trans[i].warn[0] = NO_VALUE;
    }
    pl_idx = tr_idx = NO_VALUE;
  }
  RETURN;

} /* pni_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
 * CMPS 690 - Master's Project, Summer 1984
 * AUTHOR: Brad C. Gaylord
 * AT&T Information System Laboratories
 * 11500 North Pecos, Denver CO. 80234
 * drux2!bcg OR 3160. (303)-538-1413
 *
 * USES: TABLES
 * DATA OWNED: n/a
 */

#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 */

#else D_MODULE D_INTERPRET /* was defined in T_tables.i */
#include "debug.h" /* debug levels */

#define STARVATION 3 /* Starvation threshold */
#define MAX_MENU 10 /* max # of transitions 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;
/*
 * interpret ()
 * interpret the entity graph:
 * initialize the net
 * loop forever, firing any firable transitions
 * INPUT:
 * called by: main()
 * OUTPUT:
 * return type & value: VOID
 * GLOBAL:
 * variables used: n/a
 * variables changed: n/a
 */

VOID interpret ()
{
    VOID net_init();
    VOID net_process(), net_dump(), net_options(), net_statistics();
    char * gets();

    char * s_count;
    char line [ LINE_LEN ];

    BEGIN ( "interpret" );
    
    /* initialize the net */
    net_init ();

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

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

        if ( (s_count EQ NULL) OR eqstr ( line, "q" ) ) {
            PRINTF ( "Goodbye.\n" );
            break;
        } else if ( eqstr ( line, "" ) ) {
            net_process();
        } else if ( eqstr ( line, "o" ) ) {
            net_options ();
        } else if ( eqstr ( line, "d" ) ) {
            net_dump ( );
        } else if ( eqstr ( line, "?" ) ) {
            PRINTF ( "Enter one of the following:\n" );
            PRINTF ( "\tq - quit this session\n" );
            PRINTF ( "\tkcr - fire a transition\n" );
            PRINTF ( "\tc - change Interpreter options\n" );
            PRINTF ( "\tf - dump the Petri net\n" );
            PRINTF ( "\tr - print this help message\n" );
        } else {
            PRINTF ( "Enter '?' for a Help message\n" );
        }
    }

    net_statistics();

    RETURN;

} /* interpret () */
/**
132  *  
133  *  net_process ()
134  *  do the menu processing
135  *  
136  */
137  
138  *  INPUT:
139  *  called by: interpret()
140  *  
141  *  OUTPUT:
142  *  return type & value: VOID
143  *  
144  *  GLOBAL:
145  *  variables used: Tracing, Almost_enabled
146  *  variables changed: n/a
147  */
148
149 VOID net_process ()
150 {
151      VOID make_menu(), menu_choice(), fire();
152      NUM tran_enabled();
153
154      NUM tmp_trace;
155      NUM i;
156
157      BEGIN ( "net_process" );
158      /* look for enabled, or Almost_enabled transitions */
159      make_menu ();
160
161      if ( Tracing OR Almost_enabled ) {
162          menu_choice ();
163      } else {
164          fire ( menu[0] );
165      }
166
167      Almost_enabled = NOT_OK;
168      /* fire the terminal transitions */
169      PRINTF ( "FIRING TERMINAL transitions:\n" );
170      for ( i=0; i<#tr_idx; i++ ) {
171          if ( tran enabled ( i, P_TERM ) ) {
172              /* TERMINAL transitions are always printed */
173                  tmp_trace = Tracing;
174                  Tracing = YES;
175                  fire ( i );
176                  Tracing = tmp_trace;
177      }
178      }
179
180      RETURN;
181  } /* net_process () */
/*
 * make_menu ()
 * fill the menu array with firable transitions
 *
 * INPUT:
 * called by: net_process()
 *
 * OUTPUT:
 * return type & value: VOID
 *
 * GLOBAL:
 * variables used: trans[i], tr_idx
 * variables changed: menu[]
 */

void make_menu ()
{
    void net_dump();
    bool tran_enabled();

    NUM i;
    BEGIN ( "make_menu" );
    /* clear menu array */
    for ( i=0; i<MAX_MENU; i++ ) {  
        menu[i] = NO_VALUE;
    }
    menu_idx = NO_VALUE;
    Almost_enabled = NOT_OK;
    /* look for enabled transitions */
    for ( i=0; i<tr_idx; i++ ) {  
        if ( tran_enabled ( i, P_FUNC ) ) {  
            /* add transition to menu */
            if ( ++menu_idx < MAX_MENU ) {  
                menu[menu_idx] = 1;
            } else {  
                ERROR ( "menu table overflow" );
            }
        }
    }
    if ( HAS_A_VALUE(menu_idx) AND NOT Starve_defeat ) {
        RETURN;
    }
    Almost_enabled = OK;
    /* look for almost enabled transitions */
    for ( i=0; i<tr_idx; i++ ) {  
        if ( tran_enabled ( i, P_FUNC ) ) {  
            /* add transition to menu */
            if ( ++menu_idx < MAX_MENU ) {  
                menu[menu_idx] = 1;
            } else {  
                ERROR ( "menu table overflow" );
            }
        }
    }
    if ( HAS_A_VALUE(menu_idx) ) {
        RETURN;
    }
    /* no menu? deadlock? */
    printf ( "WARNING: POTENTIAL DEADLOCK" );
    net_dump ();
    RETURN;
} /* make_menu () */
void menu_choice()
{
    void tran_print();

    num i, choice;
    char line [LINE_LEN];

    BEGIN ( "menu_choice" );

    i = NO_VALUE;

    for ( i=0; i<menu_idx; i++ ) {
        printf ( "%d: ", i );
        tran_print ( menu[i] );
    }

    printf ( "Enter the number of the transition to be fired: " );
    while ( gets ( line ) ) {
        if ( isdigit ( line[0] ) ) {
            choice = atoi ( line );
            if ( choice >= 0 && choice < menu_idx ) {
                printf ( "FIRING menu item #%d: \n", choice );
                fire ( menu[choice] );
                printf ( "\n" );
                return;
            }
        }
        printf ( "try again: " );
    }

    printf ( "EOF\n" );
    return;
}

/* menu_choice () */
/*
 * Fire
 * Fire a transition
 * INPUT:
 * called by: net_process()
 * OUTPUT:
 * return type & value: VOID
 * GLOBAL:
 * variables used: n/a
 * variables changed: hungry, last_trans, places[], bag
 */

VOID
fire ( t_idx )

NUM t_idx;

NUM tran_enabled();
VOID p1_add_bag(); p1_del_bag();

NUM i, p_idx;

BEGIN ( "fire" );

"" detect starvation of net ""
if ( trans[t_idx].type EQ P FUNC ) {  
  if (""last_trans EQ t_idx"") {  
    if ( ++hungry EQ STARVATION ) {  
      PRINTF ( "* Hungry Starvation!
""n"");  
      PRINTF ( "*Potential Net Starvation!
""n"");  
      PRINTF ( "*Turn Starvation Defeat ON!
""n"");  
      PRINTF ( "*Turn Starvation Defeat OFF!
""n"");  
    }
  } else {  
    last_trans = t_idx;  
    hungry = NULL;
  }
}

Almost_enabled = NOT_OK;

if ( NOT tran_enabled ( t_idx ) ) {
  ""almost enabled" we must supply tokens ""
  for ( i=0; i<IN_MAX; i++ ) {  
    p_idx = trans[t_idx].in[i];  
    if ( HAS_NO_VALUE (p_idx) ) {  
      Break;  ""end of input list ""
    }  
    if ( NOT places[p_idx].bag AND 
       places[p_idx].sym[0] EQ 'p' ) {  
      p1_add_bag ( p_idx );
    }
  }
}

"" remove tokens from the input places ""
for ( i=0; i<IN_MAX; i++ ) {  
  p_idx = trans[t_idx].in[i];  
  if ( HAS_NO_VALUE (p_idx) ) {  
    Break;  ""end of input list ""
  }
  p1_del_bag ( p_idx );
}

"" add tokens to the output places ""
for ( i=0; i<OUT_MAX; i++ ) {  
  p_idx = trans[t_idx].out[i];  
  if ( HAS_NO_VALUE (p_idx) ) {  
    Break;  ""end of input list ""
  }
  p1_add_bag ( p_idx );
}

"" stim and stim_nxt_mode!"" /*

- 126 -
if ( Tracing ) {
  tran_print ( t_idx );
}
RETURN;

/* fire () */
/*
 * net_dump()
 *
 * dump the Petri net
 *
 * INPUT:
 * called by: interpret()
 *
 * OUTPUT:
 * return type & value: VOID
 *
 * GLOBAL:
 * variables used: places[], trans[]
 * variables changed: n/a
 */

VOID net_dump()
{
    BOOL tran_enabled();
    VOID tran_print();

    NUM i;

    BEGIN ( "net_dump" );

    PRINTF ( "Dumping the Petri Net\n" );

    /* print the places with tokens in their bag */
    PRINTF ( "bag PLACE\n" );
    for ( i=0; i<np_idx; i++ ) {
        if ( NOT places[i].bag ) {
            continue;
        }
        PRINTF ( "%2d %s\n", places[i].bag, places[i].sym );
    }
    Almost_enabled = NOT_OK;

    PRINTF ( "enabled Function TRANSitions: " );
    for ( i=0; i<tr_idx; i++ ) {
        if ( tran_enabled ( i, P_FUNC ) ) {
            tran_print ( i );
        }
    }

    PRINTF ( "\n" );

    PRINTF ( "enabled TERMINal transitions: " );
    for ( i=0; i<tr_idx; i++ ) {
        if ( tran_enabled ( i, P_TERM ) ) {
            tran_print ( i );
        }
    }

    PRINTF ( "\n" );
    Almost_enabled = OK;

    PRINTF ( "almost enabled Function TRANSitions: " );
    for ( i=0; i<tr_idx; i++ ) {
        if ( tran_enabled ( i, P_FUNC ) ) {
            tran_print ( i );
        }
    }

    PRINTF ( "\n" );
    Almost_enabled = NOT_OK;

    RETURN;

} /* net_dump() */
void
net_options()
{
    char * s_count;
    char * line [ LINE_LEN ];

    BEGIN ( "net_options" );

    for EVER {
        PRINTF ( "Enter an option, or '?' for help: ");
        s_count = gets ( line );
        if ( s_count EQ NULL OR eqstr ( line, "q" ) ) {
            PRINTF ( "Returning to Interpreter\n" );
            break;
        } else if ( eqstr ( line, "t" ) ) {
            if ( Tracing EQ YES ) {
                PRINTF ( "Tracing turned OFF\n" );
                Tracing = NO;
            } else {
                PRINTF ( "Tracing turned ON\n" );
                Tracing = YES;
            }
        } else if ( eqstr ( line, "s" ) ) {
            if ( Starve_defeat EQ YES ) {
                PRINTF ( "Starvation defeat OFF\n" );
                Starve_defeat = NO;
            } else {
                PRINTF ( "Starvation defeat ON\n" );
                Starve_defeat = YES;
            }
        } else if ( eqstr ( line, "?" ) ) {
            PRINTF ( "Enter one of the following:\n" );
            PRINTF ( "\tq - return to Interpreter\n" );
            PRINTF ( "\tt - trace on/off\n" );
            PRINTF ( "\tls - starvation defeat on/off\n" );
            PRINTF ( "\tt - print this help message\n" );
            } else {
                PRINTF ( "Enter '?' for a Help message\n" );
        }
    }

    RETURN;
} /* net_options() */
/* 
 * net_statistics () 
 * 
 * print a summary of transition firings and conservation of tokens 
 * 
 * INPUT: 
 * called by: interpret() 
 * 
 * OUTPUT: 
 * return type & value: VOID 
 * 
 * GLOBAL: 
 * variables used: trans[] 
 * variables changed: n/a 
 */ 

VOID net_statistics () 
BEGIN ( "net_statistics" );
RETURN;

) /* net_statistics () */
 /************************************************************************* *
 * tran_enabled ()
 * Is the specified transition enabled?
 * INPUT:
 * called by: net_dump(), etc.
 * OUTPUT:
 * return type & value: BOOL: YES or NO
 * GLOBAL:
 * variables used: places[], trans[]
 * variables changed: n/a
 */

 BOOL tran_enabled ( t_idx, typ )
 NUM t_idx; /* index to trans[] */
 NUM typ; /* transition type */

 BEGIN ( "tran_enabled" );

 for ( i=0; i<IN_MAX; i++ )
   if ( trans[t_idx].type NE typ )
     RETURN ( NO );
   p_idx = trans[t_idx].in[i];
   if ( HAS_NO_VALUE ( p_idx ) )
     /* end of input list... success! */
     break;

 if ( NOT places[p_idx].bag )
   if ( Almost_enabled )
     if ( places[p_idx].sym[0] EQ '$' )
       continue;
   RETURN ( NO );

 RETURN ( YES );

} /* tran_enabled () */
 tran_print()
 * print one transition for a menu
 * INPUT:
   called by: net_dump(), etc.
 * OUTPUT:
   return type & value: VOID
 * GLOBAL:
   variables used: trans[]
   variables changed: n/a

VOID

trn_print (t_idx)

/* index to trans[] */

VOID pr_input(), pr_output();

BEGIN ("tran_print");

Pni_place = NO;

PRINTF ("%s : ", trans[t_idx].sym);

switch (trans[t_idx].type)
{
case P_INIT:
   /* PRINTF ("INIT\n") */
   printf (stdout, t_idx);
   break;

case P_FUNC:
   /* PRINTF ("TRANS
") ;
   PRINTF ("\t");
   * pr_output (stdout, t_idx);
   * printf (stdout, t_idx);
   */

   if (HAS_A_VALUE(trans[t_idx].stim[0]))
   {
      PRINTF ("%s\n", trans[t_idx].stim);
   }

   if (HAS_A_VALUE(trans[t_idx].note[0]))
   {
      PRINTF ("%s\n", trans[t_idx].note);
   }

   break;

case P_TERM:
   /* PRINTF ("TERM\n\t") */
   pr_input (stdout, t_idx);
   break;

default:
   debug (rname, D_ERROR, "trans[t_idx].type=", trans[t_idx].type);
   ERROR("internal error");
   /* switch */

RETURN;

} /* tran_print() */
/* net_init ()
   clear the net, and fire the initial transition
   INPUT:
      called by:       interpret()
   OUTPUT:
      return type & value: VOID
   GLOBAL:
      variables used: trans[0]
      variables changed: places[].bag
/

VOID net_init ()
{
    VOID pl_add_bag();
    NUM i;

    BEGIN ( "net_init" );
       /* clear the net */
    for ( i=0; i<PLACES_SIZE; ++ ) {
        places[i].bag = NULL;
    }
       /* fire the initial transition */
    pl_add_bag ( trans[P_INIT].out[P_INIT] );

    debug ( rname, D_TRACE, "place[%d].bag = %d",
            trans[P_INIT].out[P_INIT],
            places[trans[P_INIT].out[P_INIT]].bag );

    RETURN;
    } /* net_init () */
1 /* This sub-module deals with symtab exclusively */
2 #include "T_tables.i" /* see for MODULE DESCRIPTION */
3 #define extern /* T_sym1.c 'owns' T_symtab.i */
4 #include "T_symtab.i" /* symbol table */
"This routine is called when an ENTITY line is first encountered...
add entity names to the symbol table based upon entity class
and assign an index to the function table (for FUNCTIONs).
current.idx is set to the symbol table entry for this entity.
(Note that the entity name may already be in the table,
if a forward reference has occurred in a previous relation.)
if the entity is a FUNCTION, then funtab.p_or.a is assigned
the value of PERIODIC or ACTIVITY.

INPUT:
called by: entity ()
OUTPUT:
return type & value: VOID
GLOBAL:
variables used: current
variables changed: symtab[]
"/

VOID st_entity ( sym, class, f_or_i )
char *sym; /* entity name */
NUM class; /* entity class */
NUM f_or_i; /* T_FUNCTION or T_I_O_DATA */
{
    VOID st_current(), is_fun_name(), is_ioc_name();
    VOID st_idx(), st_class(), st_p_or_a();
    NUM st_sym(), st_fun();

    BEGIN ( "st_entity" );

    st_current ( st_sym (sym), f_or_i );

    switch ( f_or_i ) {
    case T_FUNCTION:
        is_fun_name ( rname, sym );
        debug ( rname, D_TRACE, "FUNCTION entity: %s", sym );
        st_idx();
        st_class ( current.idx, class );
        st_p_or_a ( class );
        break;

    case T_I_O_DATA:
        /* Input output causes annotation... */
        is_ioc_name ( rname, sym );
        debug ( rname, D_TRACE, "I_O_DATA entity: %s", sym );
        st_class ( current.idx, class );
        break;

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

    RETURN;

} /* st_entity () */
/ * st_sym () */
* add a symbol name to the symbol table if it is not already there. *
* return the index to the symbol's entry in the symbol table. *
* INPUT: *
* called by: st_entity(), T_functab.c, T_symfun.c *
* input parameters: sym name *
* OUTPUT: *
* return type & value: current symtab index *
* GLOBAL: *
* variables used: n/a *
* variables changed: sym_idx, symtab[] */

NUM st_sym ( sym )
char * sym;
{
NUM st_find();
NUM idx;

BEGIN ( "st_sym" );

if ( HAS_NO_VALUE ( idx=st_find(sym)) OR Cloning ) {
    /* entity is not in symtab: get next index */
    if ( sym_idx < SYMTAB_SIZE-1 ) {
        idx = ++sym_idx;
    } else {
        ERROR ( "symbol table overflow" );
    }

    /* put the symbol name in the symbol table */
    strcpy ( symtab[idx].sym, sym );
}

RETURN ( idx );

} /* st_sym () */
121 /*
122  "st_suffix ()
123  "store a symbol suffix.
124  " the suffix is a Mode name which is being appended to distinguish
125  " clones which come from multiple required_modes in an entity.
126  " INPUT:
127  " called by: st_mult_modes(), st_clone()
128  " OUTPUT:
129  " return type & value: VOID
130  " GLOBAL:
131  " variables used: current_idx
132  " variables changed: symtab[].sym_suffix
133 */
134
135 VOID
136 st_suffix ( mode )
137 NUM mode:
138 /* symbol name suffix */
139
140 BEGIN ( "st_suffix" );
141
142 if ( HAS_NO_VALUE ( symtab[current_idx].sym_suffix[0] ) ) {
143     SPRINTF ( symtab[current_idx].sym_Suffix, "_Xs",
144               symtab[mode].sym );
145 } else {
146     ERROR ( "max # of symbol suffixes = 1" );
147 }
148
149 RETURN;
150
151 /* st_suffix () */
/*
st_class()

store the class in the symbol table

called by: st_entity(), st_prohan(), st_Mode_list()

OUTPUT:
return type & value: VOID

GLOBAL:
variables used: n/a
variables changed: symtab().sym_class
*/

VOID

st_class ( s_idx, class )
NUM s_idx; /* index to symtab[] */
NUM class; /* type of entity */

BEGIN ( "st_class" );

if ( HAS_NO_VALUE ( symtab[s_idx].sym_class ) ) {
    symtab[s_idx].sym_class = class;
} else if ( symtab[s_idx].sym_class NE class ) {
    ERROR ( "max # of sym_class values = 1" );
}

RETURN;

} /* st_class () */
/*
  st_idx()
  store the function table index
  INPUT:
  called by: st_entity(), etc
  OUTPUT:
  return type & value: VOID
  GLOBAL:
  variables used: current_idx
  variables changed: symtab[i].sym_idx

  VOID
  st_idx ( s_idx )
  NUM s_idx;
  /* index to symtab[] */
  BEGIN ( "st_idx" ):
    if ( HAS_NO_VALUE (symtab[current.idx].sym_idx) ) {
      symtab[current.idx].sym_idx = s_idx;
    } else {
      ERROR ( "max # of sym_idx's = 1" );
    }
  RETURN;
  /* st_idx () */
" st_comment ()
  fill a PNL "comment" line
" INPUT:
  called by: TABLES, TRANSLATE
  input parameters: str - comment string
" OUTPUT:
  return type & value: VOID
" GLOBAL:
  variables used: current.idx
  variables changed: symtab[].comment
"/

VOID st_comment ( str )
BEGIN "st_comment" ;
  \begin{verbatim}
    if ( HAS_NO_VALUE ( symtab[current.idx].comment[0] ) ) {
      \textbf{ERROR} ( "Max # of comment strings = 1" );
    } else {
      SPRINTF ( symtab[current.idx].comment, \"# %s:\", str );
    }
  \end{verbatim}
RETURN;
" st_comment () "/
/**
257  * st_find ()
258  *
259  */
260  */
261  /* return the index of a symbol in the symbol table
262  */
263  /*
264  * INPUT:
265  *
266  * called by: st_sym()
267  *
268  * input parameters: symbol name
269  *
270  */
271  /*
272  * GLOBAL:
273  *
274  * variables used: symtab[].sym
275  *
276  * variables changed: none
277  */
278 
279 /* NUM
280  st_find ( sym )
281  char * sym;
282  {
283      NUM idx;
284      BEGIN ( "st_find" );
285      for ( idx=0; idx<=sym_idx; idx++ ) {
286          if ( eqstr ( Sym, symtab[idx].sym ) ) {
287              RETURN ( idx );
288          }
289      }
290      RETURN ( NO_VALUE );
291  } /* st_find () */
" st_verify ()
296 " verify that a mode found in the first part of the MODE_TABLE
297 " has been referenced by some entity, and is not already in the
298 " Mode linked-list
299 " INPUT:
300 " called by: mode_table()
301 " input parameters: mode - mode to be verified
302 " OUTPUT:
303 " return type & value: VOID
304 " GLOBAL:
305 " variables used: symtab[]
306 " variables changed: symtab[]
307 "/
308 
309 void st_verify ( mode )
310 {
311  " begin
312  VOID is_mode_name(), st_current(), st_class(), st_idx();
313  NUM st_find(), st_Mode_list();
314  NUM mod_idx;
315  NUM id; /* st_mode to 'mode' */
316  /* iterator */
317  BEGIN ( "st_verify" );
318  /* is mode correct (lexically)? */
319  is_mode_name ( rname, mode );
320  "/" is 'mode' in the symbol table? */
321  if ( HAS_ND_VALUE ( mod_idx = st_find ( mode ) ) ) {
322    /* add 'mode' to symtab */
323    mod_idx = st_sym ( mode );
324    st_current ( mod_idx, T_MODE );
325    st_comment ( "No entity directly references this mode" );
326  }
327  "/" give this symbol some class */
328  st_class ( mod_idx, T_MODE );
329  "/" is this mode already in the Mode_list? */
330  id = st_Mode_list();
331  while ( HAS_ND_VALUE (symtab[id].sym_idx) ) {
332    id = symtab[id].sym_idx;
333    if ( id Eq mod_idx ) {
334      ERROR ( "Attempt to add duplicate to Mode_list" );
335      id = id;
336    }
337  }
338  "/" add the index to 'mode' to the end of the list */
339  st_current ( id, T_MODE );
340  st_idx ( mod_idx );
341  RETURN;
342 } /* st_verify () */
/*
 * st_Mode_list ()
 * return an index to the beginning of the Mode linked list.
 * INPUT:
 * called by: mode_table()
 * input parameters: n/a
 * OUTPUT:
 * return type & value: Mode_list - symtab index to MODE_LIST
 * GLOBAL:
 * variables used: MODE_LIST (#define)
 * variables changed: symtab[]
 */

#define

NUM
st_Mode_list()
{
    NUM st_find(), st_sym();
    VOID st_class();

    NUM Mode_list;

    BEGIN ( "st_Mode_list" );

    /* establish the mode list; does MODE_LIST exist? */
    if ( HAS_NO_VALUE ( Mode_list = st_find ( MODE_LIST ) ) ) {
        /* no; create the MODE_LIST list pointer */
        Mode_list = st_sym ( MODE_LIST );
        st_class ( Mode_list, T_MODE );
    }

    RETURN ( Mode_list );

} /* st_Mode_list () */
```c
393 /*
394 * st_i_mode()
395 * store the Initial_Mode
396 *
397 * INPUT:
398 * called by: mode_table()
399 * input parameters: mode = Initial_Mode
400 *
401 * OUTPUT:
402 * return type & value: VOID
403 *
404 * GLOBAL:
405 * variables used: n/a
406 * variables changed: Init_Mode, symtab[]
407 */
408
410 VOID st_i_mode ( mode )
411 char * mode;
412 {
413     VOID is_mode_name();
414     NUM st_find(), st_Mode_list();
415     NUM idx;
416
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 == 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() - _FUNCTION, _I_O_DATA, etc
456  *
457  * GLOBAL:
458  * variables used: symtabl[], current
459  * variables changed: n/a
460 */

462 NUM
463 _ft_idx ()
464 {
465    return ( symtabl[current.idx].sym_idx );
466 }
467 } /* _ft_idx () */

471 NUM
472 _ent_type ()
473 {
474    return ( current.type );
475 }
476 } /* _ent_type () */
478 /*
479  * del_sym ()
480  * delete entries from the symbol table
481  * INPUT:
482  * called by: st_init(), st_multi_modes(), st_invokes()
483  * input parameters: idx - table index
484  * OUTPUT:
485  * return type & value: VOID
486  * GLOBAL:
487  * variables used: n/a
488  * variables changed: symtab[]
489 */
490
491 VOID del_sym ( idx )
492 NUM idx;
493 {
494  VOID del_fun();
495  NUM is_f_or_l();
496  if ( is_f_or_l ( symtab[idx].sym_class ) == T_FUNCTION ) {
497    del_fun ( symtab[idx].sym_idx );
498  }
499
500 symtab[idx].sym[0] = NO_VALUE;
501 symtab[idx].sym_suffix[0] = NO_VALUE;
502 symtab[idx].sym_class = NO_VALUE;
503 symtab[idx].comment[0] = NO_VALUE;
504 symtab[idx].sym_idx = NO_VALUE;
505 */ del_sym () */
1 #include "T_tables.h"  /* see for MODULE DESCRIPTION */
2 #include "T_symtab.h"   /* symbol table */
3 #include "T_funtab.h"   /* function table */
/*
   st_media ()
   an array of media names is stored in the "media" table.
   for I/O_DATA entities, an index into the "media" table is stored
   in the sym_idx field of the symbol table entry for the entity.
   */

   /* INPUT:
      called by: relation()
      input parameters: rhs - value of 'media' keyword
   */

   /* OUTPUT:
      return type & value: VOID
   */

   /* GLOBAL:
      variables used: media[]
      variables changed: symtab[current.idx].sym_idx
   */

VOID
st_media ( rhs )
char * rhs;
{
    NUM idx:

    BEGIN ( "st_media" );

    if ( current.type NE T_I_O_DATA ) {
        ERROR ( "media: only valid for I_O_DATAs" );
    }

    for ( idx=0; idx < MEDIA_MAX; idx++ ) {
        if ( eqstr ( rhs, media[idx] ) ) {
            if ( HAS_NO_VALUE (symtab[current.idx].sym_idx) ) {
                symtab[current.idx].sym_idx = idx;
            } else {
                ERROR ( "max # of media values = 1" );
            }
        }
    }

    ERROR ( "media not found in media table" );

} /* st_media () */
/*
 * st_mult_modes()
 * Clone any entities which have more than one mode
 * or a mode of "every_mode"
 *
 * INPUT:
 * called by: mode_table()
 *
 * OUTPUT:
 * return type & value: VOID
 *
 * GLOBAL:
 * variables used: symtab[], funtab[]
 * variables changed: symtab[], funtab[]
 */

VOID st_mult_modes()
{
    VOID del_sym(), st_clone(), st_current(), st_suffix();
    NUM is_f_or_l(), ft_idx(), st_Mode_list();

    NUM s_idx;
    NUM f_idx;
    NUM m_idx;
    NUM clone;

    BEGIN ( "st_mult_modes" );

    /* examine each of the entities: look for multiple mode functions */
    for ( s_idx=0; s_idx<sym_idx; s_idx++ ) {
        if ( is_f_or_l(symtab[s_idx].sym_class) NE T_FUNCTION ) {
            continue;
        }

        st_current ( s_idx, T_FUNCTION );
        f_idx = ft_idx();
        clone = NO;

        /* caveat: code assumes only mode[0] can EO "every_mode" */
        if ( funtab[f_idx].mode[0] EO ( m_idx=st_Mode_list() ) ) {
            "duplicate this entity for "every_mode":"
            while( HAS_A_VALUE[funtab[m_idx].sym_idx] ) {
                m_idx = funtab[m_idx].sym_idx;
                st_clone ( s_idx, m_idx );
            }
            del_sym ( s_idx );
        } else {
            /* do mode[1] or mode[2] have a value? */
            for ( m_idx=1; m_idx<ID_MAX; m_idx++ ) {
                if ( HAS_A_VALUE[funtab[f_idx].mode[m_idx]] ) {
                    st_clone ( s_idx, funtab[f_idx].mode[m_idx] );
                    funtab[f_idx].mode[m_idx] = NO_VALUE;
                    clone = YES;
                }
            }

            /* give the original entity a suffix of the 1st mode */
            if ( clone EQ YES ) {  
                st_current ( s_idx, T_FUNCTION );
                st_suffix ( funtab[f_idx].mode[0] );
            }
        }
    }

    } /* for */

    /* delete the list of modes */
    del_sym ( st_Mode_list() );

    RETURN;

    } /* st_mult_modes() */
/
* st_clone()
* Clone one entity
* INPUT:
  * called by: st_mult_modes()
* OUTPUT:
  * return type & value: Void
* GLOBAL:
  * variables used: Cloning
  * variables changed: symtab[], funtab[]
/

VOID
st_clone(s_idx, m_idx)
NUM s_idx;
NUM m_idx;

{ VOID st_current(), st_suffix(), st_class(), st_idx(), st_comment();
  NUM st_sym(), st_fun();
  
  NUM c_idx; /* clone index */
  NUM f_idx; /* original index */
  NUM i; /* iterator */

  BEGIN ("st_clone");

  Cloning = TRUE;

  /* duplicate the symbol table information */
  st_current (st_sym(symtab[s_idx].sym), T_FUNCTION);
  st_suffix (m_idx);
  st_class (current_idx, symtab[s_idx].sym_class);
  st_idx (st_fun());
  if(HAS_A_VALUE(symtab[s_idx].comment[]) ) {
    st_comment (symtab[s_idx].comment[]);
  }

  /* duplicate the function table information */
  f_idx = symtab[s_idx].sym_idx;
  c_idx = ft_idx();
  funtab[c_idx].p_or_a = funtab[f_idx].p_or_a;
  funtab[c_idx].nec_cond = funtab[f_idx].nec_cond;
  for(i=0; i<ID_MAX; i++) {
    funtab[c_idx].in[i] = funtab[f_idx].in[i];
    funtab[c_idx].out[i] = funtab[f_idx].out[i];
  }
  funtab[c_idx].mode[0] = m_idx;
  if(HAS_A_VALUE(funtab[f_idx].note[0])) {
    STRCPY (funtab[c_idx].note, funtab[f_idx].note);
  }
  if(HAS_A_VALUE(funtab[f_idx].warn[0])) {
    STRCPY (funtab[c_idx].warn, funtab[f_idx].warn);
  }

  Cloning = FALSE;

  RETURN;
}

"st_clone()"
/*
  st_transitions()
  * Generate next mode values from lines in the Allowed_Mode_Transitions
  *
  * INPUT:
  * called by: mode_table()
  *
  * OUTPUT:
  * return type & value: VOID
  *
  * GLOBAL:
  * variables used: symtab[], funtab[]
  * variables changed: funtab[].nxt_mode, funtab[].stim
  */

VOID st_transitions ( io_stim, io_val, r_mode, n_mode )
char * io_stim;  /* I/O_DATA stimulus */
char * io_val;  /* optional io_stim value */
char * r_mode;  /* required mode */
char * n_mode;  /* next mode */
{
  VOID is_iod_name(), is_mode_name();
  VOID st_current(), st_next_mode();

  NUM i_idx;  /* symtab index to io_stim */
  NUM r_idx;  /* symtab index to r_mode */
  NUM f_idx;  /* function table index */
  NUM t_idx;  /* iterators */
BEGIN ( "st_transitions" );

/* are the parameters correct lexically? */
is_id_name ( rname, io_stim );
if ( is_mode_name ( rname, r_mode ) ) {
    is_id_name ( rname, n_mode );
    /* get the symtab indices to the parameters */
    int = st_sym ( io_stim );
r_idx = st_sym ( r_mode );
    /* for entities with mode[0] EQ rmode, is io_stim a relation? */
    for ( idx=0; idx<symtab[idx].sym_idx; idx++ ) {
        if ( is_f_or_T (symtab[idx].sym_class) NE T_FUNCTION ) {
            continue;
        } else {
            f_idx = symtab[idx].sym_idx;
        }
    }
    /* does this function have a required mode of r_mode? */
    if ( r_idx NE funtab[f_idx].mode[0] ) {
        continue;
    }
}

st_current ( idx, T_FUNCTION );

debug ( rname, D_TRACE, "symtab idx=%d: %s,%s, %s=%d", 
         idx, io_stim, f_idx, n_mode, io_val );

if ( f_idx EO funtab[f_idx].rec_cond ) {
    debug ( rname, D_TRACE, "rec_cond!" );

    st_nxt_mode ( n_mode, io_stim, io_val );
} else {
    for ( i=0; i<IO_MAX; i++ ) {
        if ( (f_idx EQ funtab[f_idx].in[i]) OR 
             (n_idx EQ funtab[f_idx].out[i]) ) {
            debug ( rname, D_TRACE, "io=%d", i );
            st_nxt_mode ( n_mode, io_stim, io_val );
        }
    }
}

/* fill in nxt_mode for routines which didn't change mode */
RETURN;

} /* 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
278 /* INPUT:
279 "called by:"
280 /*
281 "input parameters:"
282 /*
283 "OUTPUT:"
284 /*
285 /*
286 /* GLOBAL:
287 /*
288 /* variables used:"
289 /*
290 /* variables changed:"
291 /*
292 /*
293 VOID 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? */
302 NUM orphan; /* symtab index to orphan */
303 NUM r_idx; /* index related to orphan */
304 NUM f_idx; /* function table index */
305 NUM idx; /* index for in and out */
306 char str [ LINE_LEN ]; /* diagnostic string */
307
308 char * format = "value of '%s' is undefined";
309 char * io_fmt = "value of '%s[%d]' is undefined";
310 char * comment = "Entity undefined by user: default values assumed";
BEGIN ( "st_orphan" );
for ( orphan=0; orphan<sym_idx; orphan++ ) {
    if ( HAS_NO_VALUE ( symtab[orphan].sym[0] ) OR
        HAS_A_VALUE ( symtab[orphan].sym_class ) ) {
        continue;
    }
    /* search for references to the orphan */
    for ( r_idx=0; r_idx<sym_idx; r_idx++ ) {
        if ( is_T_or_i ( symtab[r_idx].sym_class ) NE T_FUNCTION ) {
            continue;
        }
        f_idx = symtab[r_idx].sym_idx;
    }
    /* necessary condition */
    if ( funtab[T_idx].nec_cond EQ orphan ) {
        /* put a warning in the referencing entity */
        st_current ( r_idx, T_FUNCTION );
        SPRINTF ( str, format, "necessary_condition" );
        st_warn ( str );

        /* populate the entity with defaults */
        st_current ( orphan, T_I_O_DATA );
        st_class ( orphan, Input );
        st_media ( "keyboard" );
        st_comment ( comment );
    }
    for ( idx=0; idx<IO_MAX; idx++ ) {
        /* in[0], in[1] and in[2]. */
        if ( funtab[f_idx].in[idx] EQ orphan ) {
            /* put a warning in the referencing entity */
            st_current ( r_idx, T_FUNCTION );
            SPRINTF ( str, io_fmt, "in", idx );
            st_warn ( str );

            /* populate the entity with defaults */
            st_current ( orphan, T_I_O_DATA );
            st_class ( orphan, Input );
            st_media ( "keyboard" );
            st_comment ( comment );
        }
        /* out[0], out[1] and out[2]. */
        if ( funtab[f_idx].out[idx] EQ orphan ) {
            /* put a warning in the referencing entity */
            st_current ( r_idx, T_FUNCTION );
            SPRINTF ( str, io_fmt, "out", idx );
            st_warn ( str );

            /* populate the entity with defaults */
            st_current ( orphan, T_I_O_DATA );
            st_class ( orphan, Output );
            st_media ( "crt" );
            st_comment ( comment );
        }
        /* mode[0], mode[1] and mode[2]. */
        if ( funtab[f_idx].mode[idx] EQ orphan ) {
            /* put a warning in the referencing entity */
            st_current ( r_idx, T_FUNCTION );
            SPRINTF ( str, io_fmt, "mode", idx );

            /* populate the entity with defaults */
            st_current ( orphan, T_MODE );
            st_class ( orphan, T_MODE );
            st_comment ( comment );
        }
    }
    /* for idx: process the IO_MAX relations */
    /* for r_idx: references to the orphan */
    /* for orphan: look for index to an orphan */
RETURN;
} /* st_orphan () */
/*
st_invokes()
Create the annotations which will express the transformations that
functions apply to Input_output entities.
Delete all Input_output entities from the symbol table.
*/

INPUT:
called by: scan()
input parameters: Input_output - value of E_INPUT_OUTPUT
(imported from the SCAN module)

OUTPUT:
return type & value: VOID

GLOBAL:
variables used: funtab[], symtab[]
variables changed: funtab[].note

VOID
st_invokes (Input_output)
NUM Input_output;
{
    VOID del_sym();
    NUM isitori();

    char note[LINE_LEN]; /* temporary annotation string */
    BOOL invocation; /* copy temp annotation to note[]? */
    BOOL equals; /* insert an assignment sign (=)? */
    BOOL comma; /* insert a comma? */
    NUM f_idx; /* function table index */
    NUM s_idx; /* symbol table index */
    NUM t_idx; /* iterators */
BEGIN ( "st_invokes" );
for ( idx=0; idx<=sym_idx; idx++ ) {
    if ( is_f_or_i(symtab[idx].sym_class) NE T_FUNCTION ) {
        /* annotations only apply to functions */
        continue;
    }
    f_idx = symtab[idx].sym_idx;
    invocation = equals = NO;
    note[0] = NULL;
    /* clear the temporary annotation */
    /* process the 'results' of the annotation function */
    for ( comma=NO, i=0; i<IO_MAX; i++ ) {
        if ( HAS_NO_VALUE (s_idx=funtab[f_idx].out[i]) ) {
            Continue;
            if ( symtab[s_idx].sym_class EQ Input_output ) {
                if ( comma Eq YES ) {
                    STRCAT ( note, ", " );
                    STRCAT ( note, symtab[s_idx].sym );
                    funtab[f_idx].out[i] = NO_VALUE;
                    comma = equals = invocation = YES;
                }
            }
            if ( equals Eq YES ) {
                STRCAT ( note, ", " );
            }
            STRCAT ( note, symtab[idx].sym );
            STRCAT ( note, "(" );
        }
        /* process the 'arguments' of the annotation function */
        for ( comma=NO, i=0; i<IO_MAX; i++ ) {
            if ( HAS_NO_VALUE (s_idx=funtab[f_idx].in[i]) ) {
                Continue;
                if ( symtab[s_idx].sym_class EQ Input_output ) {
                    if ( comma Eq YES ) {
                        STRCAT ( note, ", " );
                        STRCAT ( note, symtab[s_idx].sym );
                        funtab[f_idx].in[i] = NO_VALUE;
                        comma = invocation = YES;
                    }
                }
            }
        }
        /* finish the temporary annotation */
        if ( comma Eq YES ) {
            STRCAT ( note, ", " );
        }
        STRCAT ( note, ")" );
    }
    /* copy the temporary annotation into the function table */
    if ( invocation Eq YES ) {
        if ( HAS_NO_VALUE (funtab[f_idx].note[0]) ) {
            SPRINTF ( funtab[f_idx].note, "%s", note);
        } else {
            ERROR ( "max # of annotations = 1" );
        }
    }
    /* delete the unneeded Input_output entities from the symbol table */
    for ( idx=0; idx<=sym_idx; idx++ ) {
        if ( symtab[idx].sym_class EQ Input_output ) {
            del_sym ( idx );
        }
    }
}
RETURN;
/* st_invokes () */

- 156 -
503 /*
504    st_init (), st_current ()
505    initialize symbol table, function table and current entity info
506 */
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
521 VOID st_init ()
522 {
523     VOID del_sym (), del_fun (), st_current ();
524     NUM idx;
525     BEGIN ( "st_init" );
526     "initialize symbol table"
527     for ( idx=0; idx<SYMTAB_SIZE; idx++ ) {
528         del_sym ( idx );
529     }
530     "initialize function table"
531     for ( idx=0; idx<FUNTAB_SIZE; idx++ ) {
532         del_fun ( idx );
533     }
534     "initialize current entity information"
535     st_current ( NO_VALUE, NO_VALUE );
536     "initialize table indices"
537     sym_idx = NO_VALUE;
538     fun_idx = NO_VALUE;
539     RETURN;
540 } /* st_init () */
547
551 VOID st_current ( i, t )
552 NUM i; /* current.idx */
553 NUM t; /* current.type */
554 {
555     BEGIN ( "st_current" );
556     current.idx = i;
557     current.type = t;
558     RETURN;
559 } /* st_current () */
1 #include "T_tables.i"
   /* see for MODULE DESCRIPTION */
3 #define extern
4 #include "T_funtab.i"
   /* T_func.c "owns" T_funtab.i */
   /* function table */
6 NUM ent_type();
7 NUM st_sym();
8 NUM ft_idx();
   /* see T_sym1.c */
   /* see T_sym1.c */
   /* see T_sym1.c */


/ *
 * st_fun ()
 * return next index to function table
 * INPUT:
 * called by: st_entity ()
 * OUTPUT:
 * return type & value: current function table index
 * GLOBAL:
 * variables used: n/a
 * variables changed: fun_idx
 */

NUM st_fun ()
{
    NUM idx;
    BEGIN ( "st_fun" );
    if ( fun_idx < FUNTAB_SIZE-1 ) {
        idx = ++fun_idx;
    } else {
        ERROR ( "function table overflow" );
    }
    RETURN ( idx );
} /* st_fun () */
void st_p_or_a (class)
{
    NUM class; /* class of symtab entry */
    NUM f_idx; /* function table index */

    BEGIN ("st_p_or_a");

    f_idx = ft_idx();

    if (HAS_NO_VALUE (funtab[f_idx].p_or_a)) {
        funtab[f_idx].p_or_a = class;
    } else {
        ERROR ("max # of p_or_a values = 1");
    }

    RETURN;
} /* st_p_or_a() */
/**
 * st_nec_cond ()
 *
 * file 'necessary_condition' relation value for current entity
 *
 * INPUT:
 * called by: relation()
 * input parameters: rhs - value of 'necessary_condition' keyword
 * OUTPUT:
 * return type & value: VOID
 * GLOBAL:
 * variables used: n/a
 * variables changed: funtab[].nec_cond
 */

VOID
void st_nec_cond ( rhs )
{
    char * rhs;
    void is_lod_name();
    NUM s_idx; /* symbol table index */
    NUM f_idx; /* function table index */

    BEGIN ( "st_nec_cond" );

    if ( ent_type() != T_FUNCTION ) {
        ERROR ( "necessary_condition: only valid for Activity" );
    }

    is_lod_name ( rname, rhs );
    s_idx = st_sym ( rhs );
    f_idx = ft_idx ();

    if ( HAS_NO_VALUE ( funtab[f_idx].nec_cond ) ) {
        funtab[f_idx].nec_cond = s_idx;
    } else {
        ERROR ( "Max # of 'necessary_condition' keywords = 1" );
    }

    RETURN;
} /* st_nec_cond () */
void st_input(char * rhs)
{
    char * name = rhs;
    "iterator"
    NUM i;
    NUM s_idx;
    "symbol table index"
    NUM f_idx;
    "function table index"
    BEGIN ("st_input");
    NUM i;
    "caveat: should check for prior input[] entries"
    if (ent_type() != T_FUNCTION) {
        ERROR ("input: only valid for FUNCTIONS");
        }
    "no input for this function"
    if (eqstr(rhs,"NONE")) {
        RETURN;
        }
    is_ioc_name(name,rhs);
    s_idx = st_sym(rhs);
    f_idx = ft_idx();
    "put the input value in one of the three input variables"
    for (i=0;i<IO_MAX;i++)
        if (has_no_value(funtab[f_idx].in[i])) {
            funtab[f_idx].in[i] = s_idx;
            break;
            }
    "all 3 of the input variables have been filled"
    if (i == IO_MAX) {
        ERROR ("Max # of 'input' keywords = 3");
        }
    RETURN;
}
"st_input ()"
189 /*
190 * st_output ()
191 * fill 'output' relation values for current entity
192 *
193 * INPUT:
194 * called by:    relation()
195 * input parameters:  rhs - value of 'output' keyword
196 *
197 * OUTPUT:
198 * return type & value:  VOID
199 *
200 * GLOBAL:
201 * variables used:  n/a
202 * variables changed:  funtab[].out[]
203 */
204
205 VOID
206 st_output ( rhs )
207 char * rhs;
208 {
209     VOID is_iod_name();
210     NUM i;     /* iterator */
211     NUM s_idx;  /* symbol table index */
212     NUM f_idx;  /* function table index */
213
214     BEGIN ( "st_output" );
215     is_iod_name ( rname, rhs );
216     s_idx = st_sym ( rhs );
217     f_idx = ft_idx ( );
218
219     if ( ent_type() NE T_FUNCTION ) {
220         ERROR ( "output: only valid for FUNCTIONs" );
221     }
222
223     /* put the output value in one of the three output variables */
224     for ( i=0; i<IO_MAX; i++ ) {
225         if ( HAS_NO_VALUE ( funtab[f_idx].out[i] ) ) {
226             funtab[f_idx].out[i] = s_idx;
227             break;
228         }
229     }
230
231     if ( i=EQ IO_MAX-1 ) {
232         /* all 3 of the output variables have been filled */
233         ERROR ( "Max # of 'output' keywords = 3" );
234     }
235 }
236
237 RETURN:
238 }  /* st_output () */
/*
 */

"st_req_mode ()"

"fill 'required_mode' relation value for current entity"

"INPUT:
called by: relation()
input parameters: rhs - value of 'required_mode' keyword"

"OUTPUT:
return type & value: VOID"

"GLOBAL:
variables used: n/a
variables changed: funtab[].mode[]"

"/

VOID st_req_mode ( rhs )
char * rhs;

VOID is_mode_name();

NUM i;
NUM s_idx;
NUM f_idx;

BEGIN ( "st_req_mode" );

if ( ent_type() NE T_FUNCTION ) {ERROR ( "required_mode: only valid for FUNCTIONs" );}

if ( eqstr ( rhs, "every_mode" ) ) {
   debug ( rname, D_TRACE, "duplicate for 'every_mode'" );
   } else {
      is_mode_name ( rname, rhs );
      }

   s_idx = st_sym ( rhs );
   f_idx = f_t_idx ();

   /* caveat: should check for prior 'every_mode' */
   /* put the mode value in one of the three mode variables */
   for ( i=0; i<IO_MAX; i++ ) {
      if ( HAS_NO_VALUE ( funtab[f_idx].mode[i] ) ) {
         funtab[f_idx].mode[i] = s_idx;
         break;
      }
      }
      
   if ( 1 EQ IO_MAX-1 ) {
      "all 3 of the mode variables have been filled " /
      ERROR ( "Max # of 'req_mode' keywords = 3" );
   }

RETURN;

"/ st_req_mode () "}
/*
 * st_nxt_mode()
 * f:11 'next_mode' value for current entity
 * INPUT:
 * called by: st_transitions()
 * OUTPUT:
 * return type & value: VOID
 * GLOBAL:
 * variables used: n/a
 * variables changed: funtab[()].nst_mode */

VOID
st_nxt_mode ( mode, io_stim, io_val )
char *mode; /* next mode */
char *io_stim; /* stim: mode[0] -> nxt_mode */
char *io_val; /* set of io values */
{
    NUM st_sym(), st_idx();
    VOID st_current();
    NUM s_idx, f_idx:

    BEGIN ( "st_nxt_mode" );
    has_mode_name ( rname, mode );
    s_idx = st_sym ( mode );
    f_idx = ft_idx ();

    if ( HAS_NO_VALUE ( funtab[f_idx].nst_mode ) ) {
        funtab[f_idx].nst_mode = s_idx;
        if ( io_stim[0] AND io_val[0] ) {
            SPRINTF ( funtab[f_idx].stim,
                      "if %s in %s then \%, io_stim, io_val ");
        }
    } else {
        ERROR ( 'max # of 'next_mode' keywords = 1' );
    }

    RETURN:
    /* st_nxt_mode() */
```c
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 */
370
371 VOID st_note ( era_line )
372 char "era_line":
373 {
374     NUM f_idx;  /* function table index */
375     NUM i;    /* sscanf return value */
376     char str[LINELEN];    /* occurrence string */
377
378     BEGIN ( "st_note" );
379
380     if ( ent_type() NE T_FUNCTION ) {
381         ERROR ( "occurrence: only valid in Periodic_function" );
382     }
383
384     /* clear the temporary occurrence string */
385     for ( i=0; i<LINE_LEN; i++ ) {
386         str[i] = NULL;
387     }
388
389     /* obtain the string following "whenever" */
390     i = sscanf ( era_line, " occurrence: whenever %132c", str );
391     if ( i NE 1 ) {
392         debug ( rname, D_ERROR, "era=%s", era_line );
393         debug ( rname, D_ERROR, "sscanf=%d, expr=%s", i, str );
394         ERROR ( "occurrence: whenever 'expr'" );
395     }
396
397     f_idx = ft_idx();
398
399     if ( HAS_NO_VALUE ( funtab[f_idx].note[i] ) ) {
400         SPRINTF ( funtab[f_idx].note, "WHEN ( %s )", str );
401         debug ( rname, D_TRACE, funtab[f_idx].note );
402     } else {
403         ERROR ( "Max # of annotations = 1" );
404     }
405
406     RETURN;
407 }
408 /* st_note () */
```
" st_warn ()
" fill PNL "warning" line with a diagnostic message, when a
" potential error is found.

" INPUT:
" called by: relation()
" input parameters: str - warning string

" OUTPUT:
" return type & value: VOID

" GLOBAL:
" variables used: n/a
" variables changed: funtab[].warn

"/

VOID st_warn ( str )
char * str;
{
NUM f_idx;
    /* function table index */

BEGIN ( "st_warn" );

if ( ent_type() NE T_FUNCTION ) {
    ERROR ( "warn: only valid for FUNCTIONS" );
}

f_idx = ft_idx ();

if ( HAS_NO_VALUE (funtab[f_idx].warn[0]) ) {
    SPRINTF ( funtab[f_idx].warn, "- WARNING: %s", str );
} else {
    ERROR ( "Max # of warning strings = 1" );
}

RETURN;

/* st_warn () */
/**
 * is_p_or_a ()
 * return p_or_a value for current entity
 * INPUT:
 * called by: relation()
 * OUTPUT:
 * return type & value: p_or_a - if current entity is FUNCTION
 * FAILURE - if current entity is not
 * GLOBAL:
 * variables used: funtab[].p_or_a
 * variables changed: n/a
 */

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     }
478 } /* is_p_or_a () */
/* 
* del_fun ()
* delete entries from the function table
* INPUT:
* called by: st_init(), del_sym()
* input parameters: idx - table index
* OUTPUT:
* return type & value: VOID
* GLOBAL:
* variables used: n/a
* variables changed: funtab[]
*/

VOID del_fun ( idx )
NUM idx:
{
NUM i;

  funtab[idx].p_or_e = NO_VALUE;
  funtab[idx].nec Cond = NO_VALUE;
for ( i=0; i<ID_MAX; i++ )
  funtab[idx].in[i] = NO_VALUE;
for ( i=0; i<ID_MAX; i++ )
  funtab[idx].out[i] = NO_VALUE;
for ( i=0; i<ID_MAX; i++ )
  funtab[idx].mode[i] = NO_VALUE;
  funtab[idx].nxt_mode = NO_VALUE;
  funtab[idx].stim [i] = NO_VALUE;
  funtab[idx].note [i] = NO_VALUE;
  funtab[idx].warn [i] = NO_VALUE;
}
/* 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\n", Symtab[idx].sym, \n8     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 */
/*  ********************************************** PNL TABLES ********************************************** */

/*
  pr_initial()
  emit the PNL INIT transition statement

  INPUT:
  called by: p pnl()

  OUTPUT:
  return type & value: VOID

  GLOBAL:
  variables used: trans[]
  variables changed: n/a

*/

VOID pr_initial ( fp, type )
    FILE *fp;
    /* PNL output file pointer */
    NUM type;
    /* transition type */
{
    VOID pr_output();

    BEGIN ( "pr_initial" );
    if ( trans[P_INIT].type NE type ) {
        ERROR ( "First entry in trans[] must be P_INIT" );
    }

    FPRINTF ( fp, "#------------------> INITial Transition:" );
    NEWLINE;
    PR_NODE ( trans[P_INIT].sym );
    NEWLINE;
    FPRINTF ( fp, " : INIT " );
    pr_output ( fp, P_INIT );
    NEWLINE;
    .RETURN:
}

/* pr_initial() */
/*
 * pr_places ()
 * emit the PNL PLACES statements
 * INPUT:
 * called by: p pn()
 * OUTPUT:
 * return type & value: VOID
 * GLOBAL:
 * variables used: places[]
 * variables changed: n/a
 */

VOID pr_places ( fp )
FILE = fp; /* PNL output file pointer */

VOID pr_output();

NUM idx;

BEGIN ( "pr_places" );
Pnl_place = YES;

NEWLINE;
FPRT ( fp, "# ------------------------> PLACES:" );
NEWLINE:
for ( idx = 0; idx <= pl_idx; idx++ ) {
  if ( HAS_A_VALUE ( places[idx].comment[0] ) ) {
    FRT ( fp, places[idx].comment );
    NEWLINE:
    PR_NODE ( places[idx].sym );
    NEWLINE:
    FRT ( fp, " : PLACE " );
    pr_output ( fp, idx );
    NEWLINE:
  }
}
Pnl_place = NO;

RETURN:
/* 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 */
131
132 VOID pr_transitions ( fp, type )
133 FILE * fp;
134 /* PNL output file pointer */
135 NUM type;
136 /* transition type */
137 {
138     VOID pr_output(), pr_input();
139     NUM idx;
140     BEGIN ( "pr_transitions" );
141     NEWLINE;
142     FPRINTF ( fp, "\n" );
143     for ( idx=0; idx<tr_idx; idx++ ) {
144         if ( trans[idx].type NE type ) {
145             continue;
146         }
147         if ( HAS_A_VALUE ( trans[idx].comment[0] ) ) {
148             FPRINTF ( fp, trans[idx].comment );
149             NEWLINE;
150         }
151         if ( HAS_A_VALUE ( trans[idx].warn[0] ) ) {
152             FPRINTF ( fp, trans[idx].warn );
153             NEWLINE;
154         }
155         PR_NODE ( trans[idx].sym );
156         NEWLINE;
157         FPRINTF ( fp, " : TRANS " );
158         pr_output ( fp, idx );
159         FPRINTF ( fp, " ");
160         pr_input ( fp, idx );
161         if ( HAS_A_VALUE ( trans[idx].note[0] ) ) {
162             FPRINTF ( fp, " ");
163             FPRINTF ( fp, trans[idx].note );
164             NEWLINE;
165         }
166         if ( HAS_A_VALUE ( trans[idx].stim[0] ) ) {
167             FPRINTF ( fp, " ");
168             FPRINTF ( fp, trans[idx].stim );
169             PR_NODE ( places[trans[idx].stim_nxt_node].sym );
170             NEWLINE;
171         }
172     }
173     RETURN;
174 }
175 /* pr_transitions () */
void pr_terminals (fp, type)
{
    FILE *fp;
    num type;

    /* PNL output file pointer */
    /* transition type */

    VOID pr_input();

    num idx;

    BEGIN ("pr_terminals");

    EGRESS;
    FPRINTF (fp, "#------------------- TERMINAL Transitions: ");
    NEWLINE:

    for (idx=0; idx<tr_idx; idx++) {
        if (trans[idx].type NE type) {
            continue;
        }
        PR_NODE (trans[idx].sym);
        NEWLINE;
        FPRINTF (fp, " : TERM ");
        pr_input (fp, idx);
        NEWLINE;
    }

    RETURN;
}

/* pr_terminals () */
void pr_output ( fp, index )
{
    FILE *fp;
    NUM index;
    Num i, idx, comma;

    BEGIN ( "pr_output" );

    FPRINTF ( fp, "OUTPUT TO ( " );
    comma = NO;
    for ( i=0; i<OUT_MAX; i++ ) {
        if ( PnlPlace ? HAS_NO_VALUE(places[index].out[i]) : 
            HAS_NO_VALUE(trans[index].out[i]) ) {
            break;
        } 
        if ( comma EQ YES ) {
            FPRINTF ( fp, ", " );
        }
        comma = YES;
        if ( PnlPlace ) {
            idx = places[index].out[i];
            PR_NODE ( trans[idx].sym );
        } else {
            idx = trans[index].out[i];
            PR_NODE ( places[idx].sym );
        }
    }
    FPRINTF ( fp, ")" );
    NEWLINE;
    RETURN;
} /* pr_output () */
287 /*
288  * pr_input()
289  * emit the list of nodes which are pointed to by input arcs
290  * from the current node
291  *
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  *
302  */
303
304 void pr_input ( fp, index )
305 {
306     FILE * fp;
307     /* PNL output file pointer */
308     NUM index;
309     /* current node's index */
310     NUM i, idx, comma;
311
312     BEGIN ( "pr_input" );
313
314     fprintf ( fp, "INPUT FROM ( " );
315     
316     comma = NO;
317     for ( i=0; i<IN_MAX; i++ ) {
318         if ( HAS_NO_VALUE(trans[index].in[i]) ) {
319             break;
320         }
321         if ( comma EQ YES ) {
322             fprintf ( fp, ", " );
323         }
324         comma = YES;
325         idx = trans[index].in[i];
326         pr_node ( places[idx].sym );
327     }
328
329     fprintf ( fp, " )" );
330     newline;
331
332     return;
333 }
334 /* pr_input() */
337 /*
338 "pr_Pplace ()"
339 /* print an entry in places[]
340 */
341 /* INPUT: called by: pr_tables() */
342 /* OUTPUT: */
343 /* return type & value: VOID */
344 /* GLOBAL: */
345 /* variables used: places[], trans[] */
346 /* variables changed: n/a */
347 */
348
349 VOID pr_Pplace ( fp, idx )
350 FILE * fp;
351 /* TABLES_FILE file pointer */
352 NUM idx;
353 /* index to places[] */
354 {
355    NUM i;
356
357    BEGIN ( "pr_Pplace" );
358
359    if ( HAS_NO_VALUE (places[idx].sym[0]) ) {
360        RETURN;
361    }
362
363    NEWLINE;
364
365    FPRINTF ( fp, "place[%2d] = ", idx );
366    PR_NODE ( places[idx].sym );
367    NEWLINE;
368
369    for ( i=0; i<IN_MAX; i++ ) {
370        if ( HAS_A_VALUE (places[idx].out[i]) ) {
371            FPRINTF ( fp, "out[%2d] = %2d ", i, places[idx].out[i] );
372        } else {
373            PR_NODE ( trans[places[idx].out[i]].sym );
374        }
375
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    }
384
385    RETURN;
386}
387 */ "pr_Pplace ()" */
void pr_Ptrans ( fp, idx )

FILE * fp;
NUM idx:

/* 

pr_Ptrans ()

* print an entry in trans[]

* INPUT: 
called by: pr_tables()

* OUTPUT: 
return type & value: VOID

* GLOBAL:
variables used: trans[], places[]
variables changed: n/a

*/

BEGIN ( "pr_Ptrans" );

if ( HAS_NO_VALUE (trans[idx].sym[0]) ) {
    RETURN;
}

NEWLINE:

PRINTF ( fp, "trans[%2d] : ", idx );

PR_NODE ( trans[idx].sym );

NEWLINE:

PRINTF ( fp, "type : %d\n", trans[idx].type );

for ( i=0; i<IN_MAX; i++ ) {
    if ( HAS_A_VALUE (trans[idx].in[i]) ) {
        PRINTF ( fp, "in[%2d] : %d \", i, trans[idx].in[i] );
        PR_NODE ( places[trans[idx].in[i]].sym );
    }
    else {
        NEWLINE;
        break;
    }
}

for ( i=0; i<OUT_MAX; i++ ) {
    if ( HAS_A_VALUE (trans[idx].out[i]) ) {
        PRINTF ( fp, "out[%2d] : %d \", i, trans[idx].out[i] );
        PR_NODE ( places[trans[idx].out[i]].sym );
    }
    else {
        NEWLINE;
        break;
    }
}

if ( HAS_A_VALUE (trans[idx].comment[0]) ) {
    PRINTF ( fp, "comment : \%s\n", trans[idx].comment );
}

if ( HAS_A_VALUE (trans[idx].stain[0]) ) {
    PRINTF ( fp, "stain : \%s \", trans[idx].stain );
    PR_NODE ( places[trans[idx].stain.nxt_mode].sym );
}

NEWLINE:

if ( HAS_A_VALUE (trans[idx].note[0]) ) {
    PRINTF ( fp, "note : \%s\n", trans[idx].note );
}

if ( HAS_A_VALUE (trans[idx].warn[0]) ) {
    PRINTF ( fp, "warn : \%s\n", trans[idx].warn );
}

RETURN;

} /* pr_Ptrans () */
VOID pr_entity ( fp, idx )
FILE *fp;
/* TABLES_FILE file pointer */
NUM idx;

VOID pr_class();
NUM is_f_or_i();

char *rname = "pr_entity";
NUM t, sw, f_idx;

FPRINTF ( fp, "%nsymtab[%2d] : ", idx );
if ( HAS_NO_VALUE ( symtab[idx].sym[0] ) ) {
  FPRINTF ( fp, " >>> NO_VALUE <<<\n" );
  return;
} else {
  PR_SYM ( idx );
  pr_class ( fp, symtab[idx].sym_class );
  // print the PNL comment attached to this symbol */
  if ( HAS_A_VALUE ( symtab[idx].comment[0] ) ) {
    FPRINTF ( fp, " comment : ");
    FPRINTF ( fp, "%s\n", symtab[idx].comment );
  }
  switch ( sw = is_f_or_i(symtab[idx].sym_class ) ) {
  case T_MODE:
    /* T_MODE is also the ERROR return from is_f_or_i() */
    break;
  case T_FUNCTION:
    f_idx = symtab[idx].sym_idx;
    if ( HAS_A_VALUE ( funtab[f_idx].nec_cond ) ) {
      FPRINTF ( fp, " nec_cond : ");
      PR_SYM ( funtab[f_idx].nec_cond );
    }
    for ( i=0; i<10_MAX; i++ ) {
      if ( HAS_A_VALUE ( funtab[f_idx].in[i] ) ) {
        FPRINTF ( fp, " in[%d] : ", i);
        PR_SYM ( funtab[f_idx].in[i] );
      }
    }
    for ( i=0; i<10_MAX; i++ ) {
      if ( HAS_A_VALUE ( funtab[f_idx].out[i] ) ) {
        FPRINTF ( fp, " out[%d] : ", i);
        PR_SYM ( funtab[f_idx].out[i] );
      }
    }
    for ( i=0; i<10_MAX; i++ ) {
      if ( HAS_A_VALUE ( funtab[f_idx].mode[i] ) ) {
        FPRINTF ( fp, " mode[%d] : ", i);
        - 179 -
541   PR_SYM ( funtab[f_idx].mode[1] );
542
543 }
544
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 }
552
553 if ( HAS_A_VALUE (funtab[f_idx].note[0]) ) {
554   FPRINTF ( fp, "   note : " );
555   FPRINTF ( fp, "%s\n", funtab[f_idx].note );
556 }
557
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;
563
564 case T_I_O_DATA:
565   FPRINTF ( fp, "   media : %s\n",
566            media[symtab[idx].sym_idx] );
567   break;
568
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 */
578   return;
579
580 } /* 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;
603 /* "TABLES_FILE file pointer " */
604 {
605 NUM st_Mode_list();
606 NUM idx;
607 BEGIN ( "pr_mode_list" );
608 FPRINTF ( fp, "%Initial_Mode : %s\n", symtab[Init_Mode].sym );
609 /* print the head of the list */
610 idx = st_Mode_list ();
611 FPRINTF ( fp, "%Mode_list : %s", symtab[idx].sym );
612 while ( HAS_A_VALUE(symtab[idx].sym_idx) ) {
613 idx = symtab[idx].sym_idx;
614 FPRINTF ( fp, ". %s", symtab[idx].sym );
615 }
616 FPRINTF ( fp, \n );
617 RETURN;
618 */
620 ) /* pr_mode_list () */
627 /*
628 * pr_tables()
629 * print the tables
630 *
631 * INPUT:
632 * called by:     fini()
633 * input parameters:   n/a
634 *
635 * OUTPUT:
636 * return type & value: VOID
637 *
638 * GLOBAL:
639 * variables used:   sym_idx
640 * variables changed: n/a
641 */
642
643 VOID pr_tables()
644 {
645     VOID pr_entity(), pr_mode_list(), perror();
646     VOID pr_Pplace(), pr_Ptrans();
647
648     NUM idx;
649     FILE * Tab_fp = fopen(
650         "Tab_fp", "w"
651     ) ;
652     if ( ! ( Tab_fp = fopen( TABLES_FILE, "w" ) ) ) {  
653         perror ( rname );
654         debug ( rname, D_ERROR, "can't open TABLES_FILE (see files.h)" )
655     }  
656
657     for ( idx=0; idx<sym_idx; idx++ ) {
658         pr_entity ( Tab_fp, idx );
659     }
660     pr_mode_list ( Tab_fp );
661
662     FPRINTF ( Tab_fp, "\f" );
663     for ( idx=0; idx<PLACES_SIZE; idx++ ) {
664         pr_Pplace ( Tab_fp, idx );
665     }
666     FPRINTF ( Tab_fp, "\f" );
667     for ( idx=0; idx<TRANS_SIZE; idx++ ) {
668         pr_Ptrans ( Tab_fp, idx );
669     }
670     FPRINTF ( Tab_fp, "\f" );
671
672     if ( fclose ( Tab_fp ) ) {  
673         perror ( rname );
674         debug ( rname, D_ERROR, "closing Tab_fp" )
675     }
676     RETURN;
677 } /* 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. OR 31600, (303)-538-1413

---

"USED BY: SCAN module

"ACCESS FCTS: get_line(), fatal_error()

---

"USES: \n
"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: "/
define ERR_STR "ERROR found by %s():\n\n\ntoffending line follows:\n\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 * line;
74     BEGIN ( "get_line" ):
75     /* increment the line number */
76     Inp_count ++;
77     line = fgets( Inp_line, LINE_LEN, Inp_fp );
78     /* remove the newline at the end of Inp_line */
79     if ( line NE NULL ) {
80         line[ strlen(Inp_line) - 1 ] = NULL;
81     }
82     debug ( rname, D_TRACE, "line %3d: %s", Inp_count, Inp_line );
83     RETURN ( line );
84 }
85 /* 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;  // For local variables.
114 char * str;
115 {
116     VOID fini();
117     BEGIN ( "fatal_error" );
118     /
119     /* display error message on terminal screen */
120     FPRINTF ( stderr, ERR_STR, rtn_name, Inp_count, str, Inp_line );
121     /* final trace message */
122     debug ( rtn_name, D_ERROR, ERR_STR, rtn_name, Inp_count, str, Inp_line);
123     /* terminate process cleanly */
124     fini ( FAILURE );
125 
126 
127 } /* fatal_error() */
/*
 *   Inp_init()
 *   This routine opens in_file for reading, and initializes variables
 *   related to input activities.
 *   called by: init()
 *   input parameters: in_file - input file name
 *   return type & value: n/a
 *   variables used: n/a
 *   variables changed: Inp_fp, Inp_count
 */

VOID
Inp_init ( in_file )
char * in_file;
{
    VOID exit(), perror();
    FILE * fopen();

    BEGIN ( "Inp_init" );

    Inp_count = 0;

    if (( Inp_fp = fopen ( in_file, "r" ) ) EQ NULL ) {
        perror ( "Inp_init: Inp_fp" );
        exit ( 1 );
    }

    RETURN;
}

/* Inp_init() */
/**
  171  * Inp_fini()
  172  * close Inp_fp
  175  * INPUT:
  176  * called by:       fini()
  177  * input parameters: n/a
  179  * OUTPUT:
  180  * return type & value: VOID
  182  * GLOBAL:
  183  * variables used:  n/a
  184  * variables changed: n/a
  */

VOID Inp_fini()
{
  VOID perror();
  BEGIN ( "Inp_fini" );
  if ( fclose(Inp_fp) == EOF ) {
    perror ( "Inp_fini: Inp_fp" );
  }
  RETURN;
} /* Inp_fini() */
/*
MODULE: DEBUG

If you don't have a source line oriented debugger, this is the next best thing. DEBUG allows you to put "printf" like statements in your code, and then selectively turn any of the DEBUG statements on and off between runs (without recompilation).

First you put calls to debug() in your source. debug() statements resemble calls to printf, in that a variable number of arguments are printed according to a format. One of the parameters specifies the "trace level" of the debug() statement. If that trace level is turned 'on', the debug() statement will print a message.
The trace levels are defined symbolically: for more details, see include "debug.h"<---the parameters used by debug()

You must have a global variable "program" declared above main() thusly: char * program = "foo";

These files must be in the directory where the program is run:

"1.foo" - if this file does NOT exist, debug() statements do nothing. Maximum cpu overhead is approx. 10%.

- if this file exists, it is checked for a list of numeric "trace level" numbers (from 0 to LEVELS-1) (i.e: a line containing the numbers "20 29 30"). Only those debug() statements which specify one of these trace levels will produce output.

NOTE: if "1.foo" contains the magic number "LEVELS", then HOG tracing is enabled: all debug() statements are enabled and will produce output. (see the #define for LEVELS in debug.c)

"o.foo" - if this file does NOT exist, the debug() output is sent directly to the terminal.

- if it does exist, and contains a valid writeable file name, the debug output is appended to the file.

- if it does exist, but is empty, the debug() output is sent to the file "x.foo.12345". The "12345" is the process id (pid), and thus will be unique.

You end up with a LOT of "x." files after a while.

ACCESS FCTS: debug ()

USED BY: any routine

This module is suitable for linking with any other modules.

All internal data and functions are prefixed with "Dbg_", so as to avoid collisions with declarations in other modules.

USES: n/a

This module isn't dependent upon other routines in this project. Routines in other modules may use the definitions in "debug.h", especially the trace levels, and BEGIN, RETURN and ERROR macros.

However, this module does not use the definitions in "debug.h".

TO BE DONE: - use an "rname stack" instead of rname parameter.
BEGIN() will push rname onto stack.

- trace file will show the depth of the stack as well as the parameters and return values.

ACKNOWLEDGMENTS:
Jim Leur originated the idea via his outlog() routine (RMATS II)

PROJECT: Requirements Analysis using Petri Nets
CMPS 690 - Master's Project, Summer 1984

AUTHOR: Brad C. Gaylord. LABD
AT&T Information System Laboratories
11900 North Pecos, Denver CO. 80234
drux@locog. DR 31L60. (303)-538-1413

*/
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 */
83
84 /* DEFINES local to this module: */
85 #define LEVELS 100 /* max number of trace levels */
86 #define ON YES /* print trace at this level */
87 #define OFF NO /* don't print at this level */
88
89 #define PATH_SIZE 132 /* # of chars in path name */
90 #define PATH_FORMAT "%132s" /* fmt of non "x." file name */
91
92 /* These file names must be no more than 14 characters (UNIX default) */
93 #define IN_FORMAT "i %.12s" /* "i." file: i.aaaaaaaaaaa */
94 #define OUT_FORMAT "c %.12s" /* "c." file: c.aaaaaaaaaaa */
95 #define DBUG_FORMAT "x %.6s%.5d" /* "x." file: x.aaaaaaaa.nnnnn */
96
97 #define TTY "/dev/tty" /* terminal's file name */
98
99 /* external variables declared outside this module */
100 extern char * program; /* declared in main program */
101 extern int _goprint(); /* varargs: can't use printf */
102
103 /* global variables "local" to this module: */
104 char Dbug_on = 'n'; /* is debug turned on yet? */
105 char Dbug_bad_fp = 'n'; /* if 'y' no write to Dbug_fp */
106 char Dbug_level[LEVELS]; /* array of active levels */
107
108 FILE * Dbug_fp = NULL; /* file pointer for debugging */
/*
  debug()

  "printf(3)"-like debugging statements, which can be selectively
  turned on and off between runs (without recompilation).

  INPUT:
    called by: any routine
    input parameters: see parameter definitions

  OUTPUT:
    return type & value: VOID

  GLOBAL:
    routines called: Debug_init(), debug(), __doprnt()
    variables used: Debug_level[]
    variables changed: Debug_bad_fp, Debug_on

  */

#define VARARGS3

int debug ( name, level, format, va_list )

  char *name; /* calling routine name */
  int level; /* trace level: [0, LEVELS-1] */
  char *format; /* printf(3)-like format */
  va_dcl /* varargs declaration */

  va_list ap; /* varargs argument pointer */

  /* return immediately if debug initialization has already failed */
  if ( Debug_bad_fp EQ 'y' ) {
    /* Debug_init() failed previously */
    return;
  }

  /* if debug hasn't been turned on, do so */
  if ( Debug_on EQ 'n' ) {
    if ( Debug_init() NE SUCCESS ) {
      /* initialization failure */
      Debug_bad_fp = 'y';
      return;
    }
    Debug_on = 'y';
  }

  /* return if debugging is not turned on for this trace level */
  /* note that debugging is always turned on for FAILURES (-1) */
  if ( ( level >= 0 ) AND ( level < LEVELS ) ) {
    if ( NOT Debug_level [level] ) {
      return;
    }
  }

  /* negative level codes are allowed (ie: FAILURE) */
  /* codes greater than LEVELS are flagged as an error */
  if ( level > LEVELS ) {
    debug ( "debug", FAILURE, "%d > LEVELS: see debug.c", level );
  }

  /* a timestamp could be printed here, if desired */
  FPRINTF ( Debug_fp, "%s[%2d]: ", name, level );

  /* this snippet of code is the heart of fprintf() */
  __doprnt() does the va_arg() analysis of ap
  See /usr/src/lib/libc/port/print/printf.c
  va_start(ap); /* varargs */
  (VOID) __doprnt(format, ap, Debug_fp); /* varargs */
  va_end(ap); /* varargs */

  FPRINTF ( Debug_fp, \"n\" );
  FFLUSH ( Debug_fp );

  return;

} /* debug() */
188 /*
189  * Dbug_init()
190  *
191  * "Initialize logging by
192  * - checking the trace levels in the .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 */
208
209 Dbug_init()
210 { 
211    VOID Dbug_header();
212
213    char file_name [PATH_SIZE + 1]; /* .i, .o & .x files */
214
215    FILE *fp;
216    int ret;
217    int c;
218    register i;
219
220    /* initialize all levels to off */
221    for ( i = 0; i < LEVELS; i++ )
222    {
223      Dbug_level[i] = OFF;
224    }
225
226    /* get name of input file containing trace levels */
227    SPRINTF ( file_name, IN_FORMAT, program );
228
229    /* try opening input file */
230    if ((fp = fopen (file_name, "r")) NE NULL )
231    {
232      /* read trace levels from input file */
233      for EVER {
234        if ( fscanf (fp, "%d", &c) EQ EOF )
235          break;
236      }
237
238      /* check for normal tracing */
239      if ( (c >= 0) AND (c < LEVELS) )
240      {
241        Dbug_level[c] = ON;
242        continue;
243      }
244    }
245
246    /* check for HOG tracing */
247    if ( c EQ LEVELS )
248    {
249      for ( i=0; i<LEVELS; i++ )
250        Dbug_level[i] = ON;
251      break;
252    }
253
254    } else {
255      /* .PROGRAM file is not readable */
256      return (FAILURE);
257    }
258
259 }
259  "/
260  " The .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  */
271
272  SPRINTF ( file_name, OUT_FORMAT, program );
273  "/" is the o.PROGRAM file readable? */
274  if ( ((fp = fopen ( file_name, "r" )) EQ NULL ) {  
275      /* the o.PROGRAM file isn't readable, so use tty */
276      if ( (Dbg_fp = fopen ( TTY, "w" )) EQ NULL ) {  
277          /* we couldn't even open the tty to write to */
278          return (FAILURE);
279      }
280     } else {  /* print a header */
281        Debug_header ( );
282  
283  /* all trace output goes the terminal */
284  return (SUCCESS);
285  }
286
287  }  /* does o.PROGRAM contain a file name? */
288  ret = SUCCESS:  
289  if ( fscanf ( fp, PATH_FORMAT, file_name ) EQ 1 ) {  
290      /* use contents of "o." file, instead of "x." file */
291      for ( i = 0; (i < PATH_SIZE) AND (file_name[i] NE NULL); i++ ) {  
292          if ( isprint ((int) file_name[i]) EQ FALSE ) {  
293              /* unprintable char in file_name */
294              ret = FAILURE;
295              break;
296         }
297       }
298  }
299  else {  /* no file name string in the o.PROGRAM file */
300      ret = FAILURE;
301  }
302  FCLOSE ( fp );
303
304  if ( ret EQ SUCCESS ) {  
305      /* open the trace output file */
306     if ( (Dbg_fp = fopen ( file_name, "a" )) EQ NULL ) {  
307         /* couldn't open the file, try the x. default */
308         ret = FAILURE;
309     }
310  }
311
312  if ( ret NE SUCCESS ) {  
313      /* output the trace to the default "x." file */
314      SPRINTF ( file_name, DEBUG_FORMAT, program, getpid() );
315  
316  /* open the default "x." trace output file */
317     if ( (Dbg_fp = fopen ( file_name, "a" )) EQ NULL ) {  
318        return ( FAILURE );
319     }
320  }
321
322  /* print a header */
323  Debug_header ( );
324
325  return ( SUCCESS );
326  }  /* Debug_init() */
" 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 */
351
352 VOID Dbug_header ()
353 {
354  long clock, time();
355  char *ctime();
356  register i;
357
358  /* print a date stamp */
359  clock = time ( (long *) 0 );
360  fprintf ( Dbug_fp, "------- TRACE TIME: %s", ctime (&clock ) );
361
362  /* display the trace levels */
363  fprintf ( Dbug_fp, "------- TRACE LEVELS: ");
364  for ( i=0; i<LEVELS; i++ ) {
365    if ( Dbug_level[i] ) {
366      fprintf ( Dbug_fp, "%d ", i );
367    }
368  }
369  fprintf ( Dbug_fp, "\n" );
370  fflush ( Dbug_fp );
371
372  return;
373
374 } /* Dbug_header () */
/ * Dbug_fini ()
  * close the file pointer associated with the trace output file
  * INPUT:
  * called by:          fini()
  * input parameters:  n/a
  * OUTPUT:
  * return type & value: VOID
  * GLOBAL:
  * variables used:    Dbug_fp_bad, Dbug_on
  * variables changed: Dbug_fp

void Dbug_fini()
{
    if ( ( Dbug_bad_fp EQ 'y' ) OR ( Dbug_on EQ 'n' ) ) {
        return;
    }
    if ( Dbug_fp NE NULL ) {
        FCLOSE ( Dbug_fp ) ;
    }
    return ;
} /* Dbug_fini () */
/*
  project wide definitions
  Changing this file means large recompiles! Reconsider!
  
  "USED in src/lib/Debug
  .../School/T/P"
 */

typedef short NUM;
typedef short BOOL;

/***
*** These defines are used to make static code read easier,
*** while keeping lint complaints to a minimum.
***/

#if def vax
#define VOID void
#else
#define VOID int
   /* void not defined for 8/22 */
#endif

#define ABORT (VOID) abort
#define FCLOSE (VOID) fclose
#define FFLUSH (VOID) fflush
#definePRINTF (VOID) printf
#define SCANF (VOID) scanf
#define SPRINTF (VOID) sprintf
#define STRCAT (VOID) strcat
#define STRCPY (VOID) strcpy

#define EQ 
#define NE !
#define AND &&
#define OR ||
#define NOT 

#define EVER (;;) /* for use with "for" loops */
#define YES 1
#define NO 0
#define TRUE 1
#define FALSE 0

#define SUCCESS 0
#define FAILURE -1

#define eqstr( s1, s2 ) ( NOT strcmp( s1, s2 ) )

#define LINE_LEN 132 /* max era line length */
#define WORD_LEN 40 /* max era word length */
/*
 * This include file contains constants related to Input/Output
 * USED BY: INPUT module, TABLES module, TRANSLATE module
 */

#define IN_FILE "P.era.input"           /* input file name */
#define TABLES_FILE "P.tables"           /* dump of tables */
#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
/*
 * This file contains the numeric trace levels used by debug().
 * A call to debug() in a module will have the following parameters:
 * name - calling routines name
 * level - trace level
 * format - printf() 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:beg, DR 31560, (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 */
#undef D_MODULE
#define D_MODULE D_NO_MODULE

/* 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();
"MODULE: TABLES

manage the symbol table, the function table, etc

all tables are searched linearly.

empty table entries are denoted by "NO_VALUE" values.

if a table index HAS_NO_VALUE, then that table is empty.

"USED BY: SCAN, TRANSLATE, INTERPRET modules"

" INCLUDE files required by this module: "/

#include <stdio.h>

#include "ctype.h"  /* general definitions */

#include "project.h"  /* entity encodes, etc */

#include "T_tables.h"  /* file names */

#define D_MODULE D_TABLES  /* primary module trace level */

#include "debug.h"  /* debug levels */

" DEFINES local to this module "/

#define NO_VALUE 0

#define HAS_A_VALUE(n) ((NUM) (n) NE NO_VALUE)

#define HAS_NO_VALUE(n) ((NUM) (n) EQ NO_VALUE)

" external functions "/

char * strcpy();

char * strcat();
1 /**
2 *** SUB_MODULE: T_pnltab.i
3 *** in MODULE: TABLES
4 ***
5 *** OWNED BY: T_pnltab.c
6 ***
7 *** USED BY: pr_tables()
8 ***
9 *** CONTAINS: places[], trans[i, pi_idx, tr_idx]
10 */
11
12 /* DEFINEs local to this sub-module: */
13 #define PLACES_SIZE 100 /* places table size */
14 #define TRANS_SIZE  100 /* transitions table size */
15 #define OUT_MAX    25  /* max # of outputs */
16 #define IN_MAX     25  /* max # of inputs */
17 #define P_INIT     0   /* Initial Transition */
18 #define P_FUNC     1   /* functional Transition */
19 #define P_TERM     2   /* Terminal Transition */
20
21 /* global variables "local" to this sub-module: */
22 extern
23 struct {
24    char sym [WORD_LEN]; /* places table */
25    NUM bag; /* # of tokens in a place */
26  } places [PLACES_SIZE]; /* PLACES TABLE */
27
28 extern
29 struct {
30    char sym [WORD_LEN]; /* transition name */
31    NUM type; /* types: P_INIT, P_FUNC, P_TERM */
32    NUM in [IN_MAX]; /* input transitions */
33    NUM out [OUT_MAX]; /* output transitions */
34    char comment [LINE_LEN]; /* PNL "*" comment */
35    char stim [LINE_LEN]; /* mode[0] -> nxt_mode iff stim */
36    char note [LINE_LEN]; /* annotation */
37    char warn [LINE_LEN]; /* PNL "--" warning */
38  } trans [TRANS_SIZE]; /* TRANSITIONS TABLE */
39
40 extern
41 NUM pi_idx; /* places table index */
42
43 extern
44 NUM tr_idx; /* transitions table index */
/*
 *** SUB_MODULE: T_symtab.c
 ***
 *** 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 *n* comment */

} symtab [SYMTAB_SIZE]; /* SYMBOL TABLE */

extern struct /* CURRENT ENTITY INFORMATION */
NUM idx; /* symtab index */
NUM type; /* T_FUNCTION or T_TYPEDEF */
NUM Init_Mode; /* initial Mode */

extern /* cloning on/off flag (st_sym) */

/* array of valid media names; used by T_symfun.c and T_print.c */
extern char "media []
#endif extern /* Can not initialize externs: */
#define MEDIA_MAX 4 /* number of entries in media[] */

- 201 -
/**
  *** 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 {
 [NUM p_or_a;       /* FUNCTION TABLE 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 [LINELEN]; /* mode[0] -> nxt_mode if stim */
  char note [LINELEN]; /* annotation */
  char warn [LINELEN]; /* 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.