Title
The P-NUT system : an environment for modeling and analyzing concurrent systems

Authors
Razouk, Rami R.
Morgan, E. Timothy

Publication Date
1985

Peer reviewed

The P-NUT System: An Environment for Modeling and Analyzing Concurrent Systems

Rami R. Razouk
E. Timothy Morgan

ABSTRACT

85-16

The availability of low-cost powerful processors has made distributed computer systems a reality. Currently, the major stumbling block in the design of these systems is the difficulty of designing and validating concurrent software which is to control and execute on the new processors. There is a need for new techniques and tools for modeling and evaluating designs of distributed computer systems during early stages of design. The Distributed Systems Project at UCI has been investigating Petri Net-based modeling techniques and has developed a suite of tools, named P-NUT, for constructing and analyzing complex Petri Net models. This paper describes the motivation behind the selection of the Petri Net model, and describes the tools which currently exist.

Technical Report #85-16

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717

July 1985

The P-NUT System: An Environment for Modeling and Analyzing Concurrent Systems

Rami R. Razouk
E. Timothy Morgan

Information and Computer Science Department
University of California, Irvine

ABSTRACT

The availability of low-cost powerful processors has made distributed computer systems a reality. Currently, the major stumbling block in the design of these systems is the difficulty of designing and validating concurrent software which is to control and execute on the new processors. There is a need for new techniques and tools for modeling and evaluating designs of distributed computer systems during early stages of design. The Distributed Systems Project at UCI has been investigating Petri Net-based modeling techniques and has developed a suite of tools, named P-NUT, for constructing and analyzing complex Petri Net models. This paper describes the motivation behind the selection of the Petri Net model, and describes the tools which currently exist.

Introduction

Recent advances in micro-electronics have sparked interest in the design of distributed systems. Distributed processing is seen as a means of achieving higher performance and greater reliability. The task of designing and implementing concurrent software which is to control (and execute on) these systems is a difficult and complex task. With the added complexity there is a greater need for models which can permit experimentation at early stages of the design process. These models must be

---

* This work has been supported in part by a MICRO grant co-sponsored by Hughes Aircraft Co. and the University of California, and by a grant from the National Science Foundation (grant no. DCR 84-06756).

supported by tools which can analyze them.

Petri Nets have long been touted as useful in modeling concurrent hardware/software. A variety of extensions have been proposed which support verification [Symons 80, Berthelot 82, Berthomieu 83] and performance evaluation [Ramchandani 74, Sifakis 77, Ramamoorthy 80, Zuberek 80, Molloy 82, Razouk and Phelps 84, Holliday and Vernon 85]. This dual use of the Petri Net model makes it a rare breed. Generally techniques which support verification (e.g. temporal logic, algebraic specifications) ignore timing and performance issues. Also, performance models (e.g. queueing networks) usually abstract away functionality and cannot be used for rigorous proofs of correctness. Yet in distributed systems issues of correctness and performance are so tightly coupled that it is difficult to deal with one without the other.

The Distributed Systems Project at UCI has focused on the use of Petri Nets to model and evaluate distributed systems. The research has produced a suite of tools, named P-NUT, which can be used to prove partial correctness and to evaluate performance. In Section 1 of this paper, the basic requirements for analyzing distributed systems are outlined. In Section 2, the Petri Net model on which this research is based is briefly described. Section 3 summarizes some of the principles which have guided the development of the tools which comprise the P-NUT system. Section 4 describes the tools which currently exist.

1. Analysis of Distributed Systems

Designers of distributed systems (software and hardware) face a complex and demanding task. Among the factors which contribute to the complexity of the task are:

1. Distributed systems include multiple processors which can act simultaneously. This "true concurrency" makes the design of distributed software more difficult than the design of concurrent software (multiple processes executing on a single

processor).

2. Distributed systems are often expected to continue operating in the face of processor failure and unreliable communication. These added requirements add to the complexity of the designs.

3. Time is an important factor in distributed systems. The existence of multiple processors, each with an essentially independent clock, makes synchronization difficult. Timing errors can lead to incorrect operation and/or to degraded performance.

In order to assist designers of distributed systems, new modeling and analysis techniques (supported by tools) must be developed. The focus of these techniques should be to provide the designer with a clear understanding of the "states" the systems can reach and how those states can be reached. The state-space of a system can be thought of as a graph where nodes represent system states and where edges represent events causing state transition. Unfortunately, the state-space of even the smallest of distributed systems is extremely large. The introduction of time into any model of such systems can make the state-space infinite. Any techniques which requires exhaustive enumeration of all the states is therefore expected to be of limited usefulness. Methods must be developed which allow designers to view this potentially infinite state-space in a more compact and understandable form. The Distributed Systems Project at UCI has been exploring two interrelated approaches to this problem. The first approach relies on grouping system states into classes of states, and exhaustively constructing graphs containing all classes of states. The second approach constructs a subset of the system states by traversing a single long path through the system state-space. This approach maintains all the details of each system state. Each of these approaches is briefly discussed below.

Exhaustive State Exploration:

As stated above, this approach is based on grouping states into classes. States are generally grouped by omitting some of the details of the system state. In order to more clearly understand the details which can be omited from a stated description it is possible to view a state as consisting of three components:

1. Control component: This is the portion of the system state which relates to the control state of each process (point of execution) and of each resource (busy, free,...).

2. Time component: This is the portion of the system state which describes the timing relationship between various hardware and software components.

3. Data component: This is the portion of the system state dealing with data, and data transformations.

The partitioning of a system state into these three components provides a convenient set of criteria for grouping states. In the P-NUT system, methods have been developed for grouping states according to the control component only (ignoring timing and data) or according to both the control and timing components (ignoring data). The first method allows the designer to focus strictly on control flow anomalies (e.g. deadlock, livelock), while the second adds timing thereby allowing the designer to investigate timing issues from both correctness and performance standpoints.

Path exploration:

The most commonly used method of analyzing systems with large state-spaces is to explore paths through the state-space. This technique is commonly referred to as simulation. One should not lose sight of the fact that path exploration and exhaustive state exploration are strongly related: path exploration attempts to reconstruct a subset of the complete state space by traversing one long path through the graph of system states; exhaustive state exploration attempts to reconstruct the complete state-

space (or a projection of that space) by exploring all paths out of every state.

The objective of the set of tools which are the subject of this paper is to provide designers with the ability to explore the system state-space in the ways described above. The model chosen as the basis of the analysis techniques and tools is the Petri net model. This selection was based on the large body of theoretical work which exists on Petri Nets, and based on the fact that it is the only model to date which has been effectively used for both correctness proofs and performance evaluation. The next section briefly introduces Petri Net models.

2. Petri Nets

The Petri Net model dates back to early work by Petri in the early 1960's [Peterson 81]. Since that time the model has evolved from a purely theoretical model of computation to a practical tool for design and analysis. A Petri Net consists of a set of places (represented by circles) modeling conditions, and a set of transitions (represented by bars) modeling events. A condition is said to hold if the corresponding place holds *tokens*. Arcs connecting places to transitions describe the conditions which must hold before an event occurs. Arcs connecting transitions to places describe the conditions which hold after an event has occurred. The occurrence of an event (a transition firing) removes tokens from input places (disabling the pre-conditions) and places tokens on the output places (enabling the post-conditions). Control dependencies, including parallelism, synchronization and resource sharing (the control component described in Section 2) can be easily modeled as described in [Agerwala 79].

The simple communication protocol shown in Figure 1 can be used to illustrate Petri Net models and to show the need for some of the extensions which have been adopted in the research. The model consists of a Sender, a Receiver, and a Transmission Medium (bi-directional). The Sender can be ready to send (place 1), can be waiting for an acknowledgement (place 4) or can be preparing a message for transmission (place 5). The Sender expects acknowledgements in place 6. The four possible events in the sender are sending a message (transition 2), timeout (transition

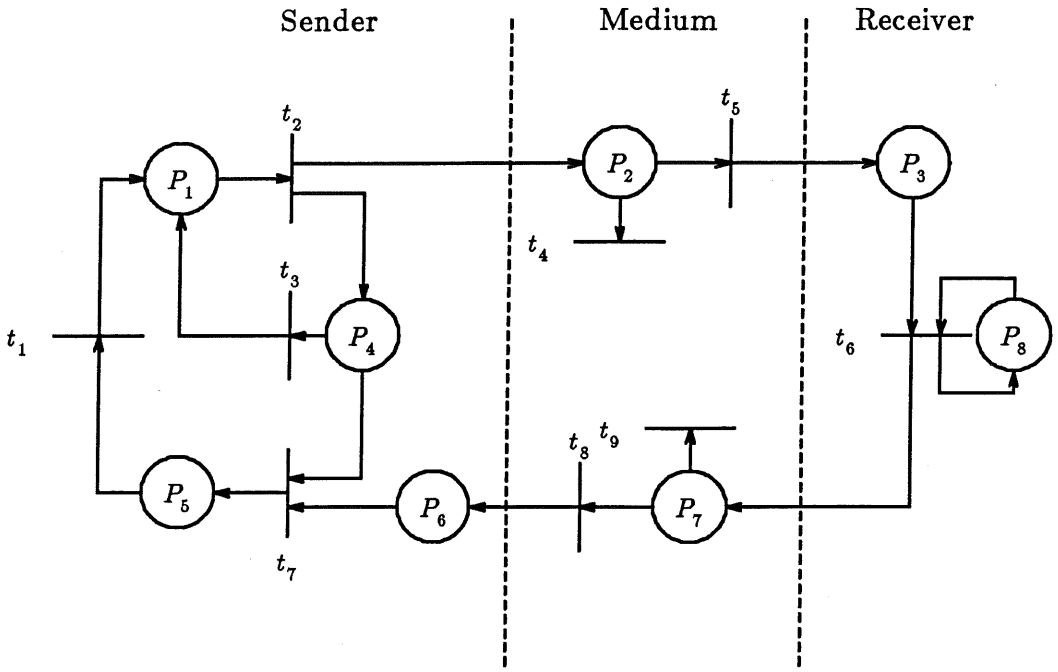

Figure 1. Model of Simple Protocol

3) successful receipt of acknowledgement (transition 7), and preparation of the next message to send (transition 1).

The Receiver is always ready to receive (place 8) and expects messages in place 3. The only event possible in the Receiver is that a message is received and an acknowledgement is sent (transition 6).

The transmission medium (in each direction) can only be in one of two states: either it holds a message (a token in places 2 and 7) or it doesn't (no tokens). The only possible events are the successful transmission of a message/acknowledgement (transitions 5 and 8), or the loss of a message/acknowledgement (transitions 4 and 9).

This simple example illustrates the need for some extensions which have been adopted. To model processing and transmission delays, firing times are associated with each transition. Once a transition begins to fire, it continues to fire until the firing time elapses. To model timeouts, an enabling time is associated with each transition. Once a transition is enabled, it is not allowed to fire until the enabling time has passed. The transition must be enabled continually during that interval. In order to model probabilistic events, firing frequencies can be associated with transitions which contend with each other for tokens (modeling resources). The frequencies model the probability of events happening. These extensions can model the timing component of a computer system (as described in section 2).

The extensions discussed to this point can be used to construct abstract models of systems. They are not useful if more detail is needed. For example, lengths of messages cannot be modeled using the extensions listed above. In order to support more detailed model, the extensions which form predicate/action nets have been adopted [Diaz 82]. A user may specify predicates associated with transitions. These predicates model data-dependent factors which may influence flow of control. A user may also specify actions (in the form of small algorithms) which describe the data manipulation activities involved in each event. Section 4 gives brief examples of predicates and actions.

3. Objectives and Design Philosophy

The main goal behind the P-NUT system is to develop a collection of tools that a designer can "mix and match" in a variety of ways to achieve the overall objective of modeling and analyzing concurrent software/hardware systems. This goal is ambitious given that the P-NUT system is being developed in a university environment by graduate students working toward their Ph.D. degree. In order to make the goal achievable, care had to be taken in designing the overall structure within which the tools fit. The overall design philosophy of P-NUT can be summarize in the following two general rules:

1. The system is to be composed of small, highly specialized tools.

2. The tools interact by sharing a few "standard" representations. All tools should interface with these forms, extracting from them only the needed information.

Below, each of these points is elaborated.

Small, specialized tools

The approach of building P-NUT out of a large number of small and specialized tools was motivated by several factors. First, the small granularity of the tools permits concurrent development of tools. Secondly, the tools can be highly optimized to perform their function. Since many of the analyses being developed are computationally intensive, efficiency (both in time and space) is critical to the success of the project. Some significant gains have been achieved in the design of efficient analysis tools [Razouk and Hirschberg 85]. Finally, the resulting environment encourages innovation since the addition of tools requires little effort. There is no need to understand the inner workings of existing tools in order to develop new ones.

This approach also has some drawbacks. A user is faced with a large number of tools whose combined use is unclear. One remedy is to provide extensive documentation of the tools with detailed examples of how the tools were used collectively to achieve a goal. The system is currently aimed at sophisticated and knowledgeable users (at least in the Petri Net world) who are expected to discover new ways of combining the tools. Another drawback is that code re-use, while considered an excellent idea, is not actively supported. Currently, a large degree of code sharing is actually being done, but that is the result of the fact that the research group is rather small (six members).

Few "standard" representations

Given that the system is to be composed of many small tools, some standards had to be adopted for how the tools were to interface with one another. The tools have been designed to fit together using UNIX® "pipes". Each tool reads inputs from standard input and produce results on standard output. The user determines if the outputs are to be stored onto files, passed on to other tools, or both. This approach is particularly useful for non-interactive tools. Interactive tools expect input from standard input and from the user terminal.

The tools described below vary widely in their functionality. However, they all operate on Petri Nets, Reachability Graphs or Execution Traces. The *Petri Nets* can vary widely from standard "vanilla" Petri Nets to timed Petri Nets and even to fully-interpreted Petri nets. Regardless of the level of detail chosen by a designer, a standard representation of the Petri Net is shared by all the tools operating on them. Each tool extracts from the standard form the information it needs. For example, the reachability graph builder ignores timing and interpretation. A *Reachability Graph* represents a partial or a complete system state-space. As is the case with Petri Nets, different tools expect different types of graphs. For example, performance analysis tools expect timing information while other tools can operate on graphs which omit time. *Execution Traces* represent paths through the system state space.

As the system is currently in its infancy, all the chosen standard forms are textual, and even human-readable. Debugging is thereby simplified at the expense of some efficiency.

In addition to the two guiding principles outlined above, the tools were also designed to be portable. Since the intent of the research was to exchange techniques and tools with other Universities and with Industry, simple and portable implementations were deemed necessary.

UNIX is a registered trademark of the Bell System.

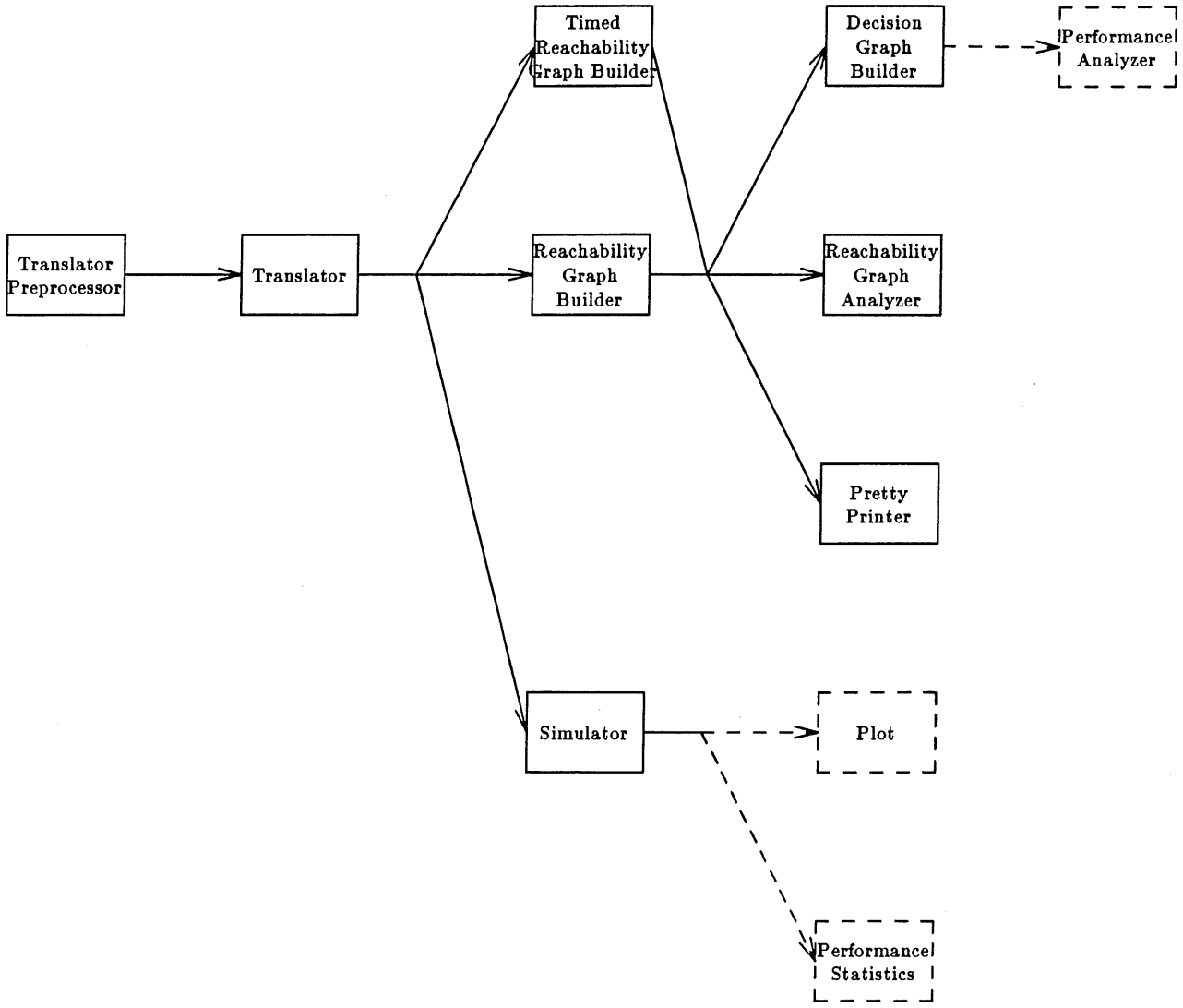

Figure 2. The P-NUT System

4. The P-NUT Tools

Figure 2 shows the tools available in the P-NUT system. This section describes each of the tools and presents brief examples of their use.

Translator

The Petri Net translator accepts textual representations of Petri nets and transforms them into standard form processable by other tools. A simple Petri Net can be represented as a set of transitions of the form:

input places -> output places

where each place is named and can be followed by the number of required tokens (in parentheses). Transitions can also be named. Figure 3 shows the textual description of the Petri Net in Figure 1, with the places given more descriptive names.

```
/* Sender **/
t1: ack_received -> ready_to_send
t2: ready_to_send -> message, wait_for_ack
t3: wait_for_ack -> ready_to_send
t7: wait_for_ack, received_ack -> ack_received

/* Sender to Receiver Medium */
t4: message ->
t5: message -> received_message

/* Receiver to Sender Medium */
t8: ack -> received_ack
t9: ack ->

/* Receiver */
t6: wait_for_message, received_message -> wait_for_message, ack
```

Figure 3. Textual Description of Simple Protocol.

The Petri Net Translator also supports the following extensions to Petri Nets:

1. *Timing.* Each transition in a Petri Net can have a triple associated with it (see Figure 4). The first number is the enabling delay. The second number is the firing time (processing delay). The third number is the relative firing frequency which indicates how often this transition fires compared to other conflicting transitions (transitions which share input places).

2. *Predicates.* Data variables can be used to simplify the Petri Net model. In

```
/* Sender **/
t1: ack_received -> (0, 1 ms, -) ready_to_send
t2: ready_to_send -> (0, 1 ms, -) message, wait_for_ack
t3: wait_for_ack -> (1 sec, 1 ms, 0) ready_to_send
t7: wait_for_ack, received_ack -> (0, 13.5 ms, 1) ack_received

/* Sender to Receiver Medium */
t4: message -> (0, 106.7, 5)
t5: message -> (0, 106.7 ms, 95) received_message

/* Receiver to Sender Medium */
t9: ack -> (0, 106.7 ms, 5)
t8: ack -> (0, 106.7 ms, 95) received_ack

/* Receiver */
t6: wait_for_message, received_message -> (0, 13.5 ms, -) wait_for_message, ack
```

Figure 4. Simple Protocol with Time.

order for the data variables to influence flow of control it is possible for a designer to add predicates to transitions. These predicates must be true before a transition can fire. The addition of predicates makes analysis for deadlocks more difficult since the Petri Net itself is an incomplete model of the control flow. Care must be taken in interpreting results of analyses which omit predicates since predicates can introduce undesirable states.

3. *Actions.* Data variables can be altered during the firing of transition. A transition can have a small program segment associated with it. This program segment is executed when the transition fires, thereby altering the values of shared variables.

Translator preprocessor

The translator described above is aided by a preprocessor which supports more compact textual representations of Petri Nets. These compact representations are particularly useful for Petri Nets whose structure is regular to the extent that connections between places and transitions can be described using some simple expressions. To best explain the concept, the dining philosopher problem is used as an example.

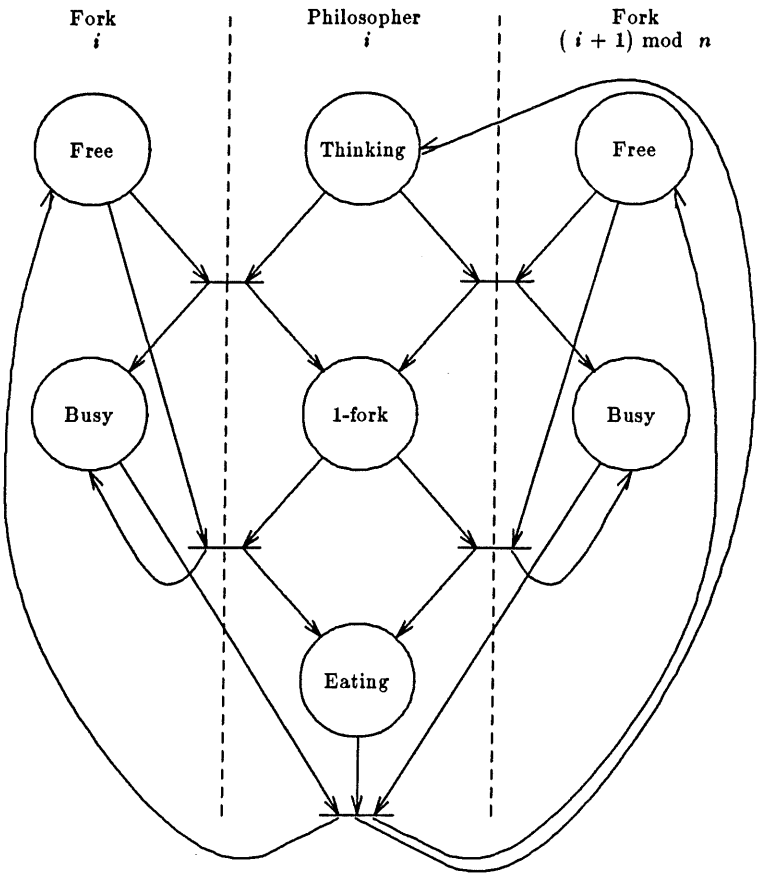

Figure 5a. Petri Net of a Dining Philosopher

In the dining philosophers problem we have n philosophers (usually five) separated by forks (or chop-sticks for lovers of Chinese food) seated around a table. Each philosopher can access forks to his/her right and left. This problem can be easily described by focusing on the behavior of the ith philosopher. This philosopher has access to forks i and (i+1) mod n. Figure 5a shows the Petri Net for the ith philosopher and figure 5b shows the textual description which the preprocessor supports. The preprocessor expands the net as shown in Figure 6. The expression appearing in angle brackets (<>) is the initial state of the net.

```
for n=3 {

array philosopher_thinking(n), philosopher_1_fork(n), fork_free(n)
array fork_busy(n)

for i=0 to n-1 {
:take_first_left_fork[i]: philosopher_thinking[i], fork_free[i] -> philosopher_1_fork[i],
                          fork_busy[i]
:take_first_right_fork[i]: philosopher_thinking[i], fork_free[(i+1) % n] -> philosopher_1_fork[i],
                          fork_busy[(i+1) % n]
:take_second_left_fork[i]: philosopher_1_fork[i], fork_free[i] -> philosopher_eating[i],
                          fork_busy[i]
:take_second_right_fork[i]: philosopher_1_fork[i], fork_free[(i+1) % n] -> philosopher_eating[i],
                          fork_busy[(i+1) % n]
:release_fork[i]: philosopher_eating[i], fork_busy[i],
      fork_busy[(i+1) % n] -> philosopher_thinking[i], fork_free[i],
      fork_free[(i+1) % n]

<philosopher_thinking[i], fork_free[i]>
}
}
```

Figure 5b. Compact representation of Dining Philosophers

```
array philosopher_thinking(3), philosopher_1_fork(3), fork_free(3)
array fork_busy(3)
:release_fork0: philosopher_eating0, fork_busy0, fork_busy1 -> philosopher_thinking0, fork_free0, fork_free1
:take_second_right_fork0: philosopher_1_fork0, fork_free1 -> philosopher_eating0, fork_busy1
:take_second_left_fork0: philosopher_1_fork0, fork_free0 -> philosopher_eating0, fork_busy0
:take_first_right_fork0: philosopher_thinking0, fork_free1 -> philosopher_1_fork0, fork_busy1
:take_first_left_fork0: philosopher_thinking0, fork_free0 -> philosopher_1_fork0, fork_busy0
:release_fork1: philosopher_eating1, fork_busy1, fork_busy2 -> philosopher_thinking1, fork_free1, fork_free2
:take_second_right_fork1: philosopher_1_fork1, fork_free2 -> philosopher_eating1, fork_busy2
:take_second_left_fork1: philosopher_1_fork1, fork_free1 -> philosopher_eating1, fork_busy1
:take_first_right_fork1: philosopher_thinking1, fork_free2 -> philosopher_1_fork1, fork_busy2
:take_first_left_fork1: philosopher_thinking1, fork_free1 -> philosopher_1_fork1, fork_busy1
:release_fork2: philosopher_eating2, fork_busy2, fork_busy0 -> philosopher_thinking2, fork_free2, fork_free0
:take_second_right_fork2: philosopher_1_fork2, fork_free0 -> philosopher_eating2, fork_busy0
:take_second_left_fork2: philosopher_1_fork2, fork_free2 -> philosopher_eating2, fork_busy2
:take_first_right_fork2: philosopher_thinking2, fork_free0 -> philosopher_1_fork2, fork_busy0
:take_first_left_fork2: philosopher_thinking2, fork_free2 -> philosopher_1_fork2, fork_busy2
<philosopher_thinking0, fork_free0>
<philosopher_thinking1, fork_free1>
<philosopher_thinking2, fork_free2>
```

Figure 6. Output of the Preprocessor

Reachability Graph Builder

One analysis method currently supported in P-NUT is the automated construction of reachability graphs. The reachability graph builder (RGB) operates on simple Petri Nets, ignoring timing and interpretation information. RGB produces a standard reachability graph which is suitable for processing by other tools such as the pretty-printer and the reachability graph analyzer.

RGB was designed with great care in order to maximize its efficiency. Reachability Graphs are known to grow exponentially (in general) with the number of places, transitions and tokens in a net. In some cases, reachability graphs can be infinite. In order to construct an efficient tool, several types of models were identified where efficiency could be gained by taking advantage of the designer's understanding of the problem. These cases are:

1. Bounded Graphs. If the designer knows that the reachability graph is bounded, then the time-consuming task of checking for potentially infinite graphs is eliminated. This yields large savings in time (and space since the arcs of the graph need not be stored). Should the designer guess incorrectly, the program enters an infinite loop. The designer can then abort the analysis and restart it without the boundedness assumption.

2. Bounded at less that 127. In the vast majority of Petri Net models, the maximum number of tokens in any place is a small integer. In such cases, the number of tokens in a place is stored in a single byte rather than a full word (2 or 4 bytes depending of the execution environment). This saving in space permits the generation of larger graphs. If the assumption is violated, the tools does NOT detect the error. The designer is advised to use the reachability graph analyzer to check for any states containing a place with 127 tokens. If any are found, the analysis should be repeated.

3. Bounded at 1 (safe). In the case where the net being analyzed is safe, a single bit is used to prepresent each place. Calculation of successors requires logical

operations (exclusive-OR) and can be done (in the case of a VAX implementation) 32 bits at a time (32 places). This form of analysis executes much faster than the others and requires significantly less space. For more details on the relative performance of each of these tools the reader is referred to [Razouk and Hirschberg 85]. If the safeness assumption is violated, the user is notified.

This approach of using the designer's understanding of the problem to aid in increasing the efficiency of the analysis has made it possible to analyze larger graphs than previously possible. The largest graph built to date contains nearly 20,000 states (9 dining philosophers) and, because it is safe, could be built in less than seven minutes of CPU time on a VAX 750 (less time than it took to format this paper).

Timed Reachability Graph Builder

One of the novel tools in P-NUT is one which constructs reachability graphs which incorporate time. This tools was inspired by work by Zuberek [Zuberek 80] as extended in [Razouk and Phelps 84]. The tool constructs a graph where each node contains a marking and a representation of the amount of time remaining before each transition can fire and before each firing transition can finish firing. arcs in the graph are weighted by the amount of time each state transition requires. Nodes with multiple successors have probabilities associated with each outgoing arc.

The output of this tool is a standard reachability graph which can be processed by other tools. A tools which compresses into a Decision Graph [Razouk and Phelps 84] has been built. A Decision Graph is a compacted version of the reachability graph which contains all performance related information needed to derive performance measures. A Performance Analyzer which processes these decision graphs is planned.

Pretty-Printer

One convenient tool for perusing reachability graphs is a pretty printer. This tools accepts reachability graphs as input and accepts a set of user-defined parameters to control the displaying of the graph. Normally, the pretty-printer displays the complete graph, starting from the initial state, as a tree. Only successor links are displayed, and the output is made as wide as the user's terminal. Each node is represented by a number. At the end of the display, the marking corresponding to each state is displayed next to the state number. Figure 7 shows a partial display of a reachability graph.

```
0->6->18->0
|  |  |
|  |  +->25->3->16->25
|  |  |        |  |
|  |  |        |  +->24->6
|  |  |        |
|  |  |        +->15->25
|  |  |        |  |
|  |  |        |  +->22
|  |  |        |
|  |  |        +->14->24
|  |  |        |  |
|  |  |        |  +->0
●
●
●

0. fork_free0,philosopher_thinking0,fork_free1,philosopher_thinking1,fork_free2,philosopher_thinking2
1. fork_busy0,philosopher_1_fork0,fork_free1,philosopher_thinking1,fork_free2,philosopher_thinking2
2. fork_free0,philosopher_1_fork0,fork_busy1,philosopher_thinking1,fork_free2,philosopher_thinking2
3. fork_free0,philosopher_thinking0,fork_busy1,philosopher_1_fork1,fork_free2,philosopher_thinking2
4. fork_free0,philosopher_thinking0,fork_free1,philosopher_1_fork1,fork_busy2,philosopher_thinking2
●
●
●
```

Figure 7. Sample Pretty-Printer output

The user-defined parameters allow a user to:

1. Display the graph backward. Starting from the initial state, predecessor links are traversed (rather than successor links).

2. Control the width of the display. If the output is to be stored on a file for later printing on a line printer, the designer can request the display to be geared to a wider display (e.g. 132 columns).

3. Control the starting point of the display. A state other than the initial state can be used as the starting point of the display.

4. Control the depth of the display. If the designer wishes to focus on part of the graph, he/she can request that only successors (or predecessor) which are reachable via paths of a certain length should be displayed. A depth of 0 causes only a single state to be displayed.

Currently the pretty-printer is targeted for the lowest common denominator for output devices. There is a clear need for more sophisticated graphical output.

Reachability Graph Analyzer

One of the most innovative tools in the P-NUT system is one which aids in analyzing reachability graphs. The need for this tool arises from the size of typical reachability graphs. Even if the graphs are finite, they are usually large. The efficiency of our reachability graph builders allows us to construct very large graphs which cannot be analyzed manually. The reachability graph analyzer was therefore built to permit automated analysis of these graphs to aid in verifying that key properties are satisfied.

The reachability graph analyzer (RGA) permits the user to define (in first-order predicate calculus) a set of properties relating to states, places, transitions and state transitions. These properties are then verified against the known set of reachable states. Since the user defines the properties to be verified, RGA is capable of verifying

general properties such as deadlock-freeness, as well as system-specific properties (e.g. the preservation of certain resources).

Because of the complexity of RGA, a full explanation of its capabilities are beyond the scope of this paper. The reader is referred to [Morgan and Razouk 85, Morgan 84] for further detail. For the purpose of this paper we will simply provide some brief examples related to the dining philosophers problem.

In order to verify that the dining philosophers cannot deadlock a user can ask is every state has at least one successor state (no terminal states). This question can be formulated as:

$$\text{forall s in S [nsucc(s)} > 0]$$

where S is a predefined set of all reachable states. In this case the tool responds with **false**. The user can then ask for the set of states which are deadlocks as follows:

$$\{\text{s in S} \mid \text{nsucc(s)} = 0\}$$

In this case there is only one state in the set. The user can further define functions which can be used to prove system specific properties. For example, the user can define a function which returns the number of philosophers eating as follows:

```
philosophers_eating (s) [count] ::= count := 0 \
    forall p in philosopher_eating [count := count + p(s); true] \
        count
```

In this case the local variable *count* is used to accumulate the total number of tokens in the set of places "philosopher_eating". The forall construct is used to loop. The user can then use this function to verify that the total number of philosophers eating at any time is less than or equal to the number of philosophers divided by the number of forks required to eat. This question can be formulated as:

forall s' in S [philosophers_eating(s') <= 5/2]

This question is specific to the case of five dining philosophers. The tool responds with **true.**

Since the tool provides a programming language, the designer can construct complex algorithms for analyzing the graphs. As more experience is gained using the tool, additional user-defined algorithms are added as built-ins. A recent extension to the tools also allows it to process timed reachability graphs. This opens the possibility of verifying properties of concurrent systems while taking timing assumptions into consideration.

Decision Graph Builder

As described earlier, the primary function of the Decision Graph Builder is to compress timed reachability graphs. The retained information consists of only nodes with multiple successors (modeling non-determinism or decision making). All other nodes and edges are collapsed with the information along the edges accumulated. For example, long sequential paths through a timed reachability graph are replaced by one edge labeled with the sum of all the delays along the original path.

Simulator

The reachability graph tools described above focus on exhaustive analysis. Each tools focuses on some limited aspect of system behavior in the hope of making the analysis manageable. It is still desirable to provide the designer with the capability of examining the models in their full details. This can be accomplished by limiting the state-exploration to some limited subset of all system states. For this purpose, a simulator has been built which exercises untimed, timed and interpreted Petri nets. The design of the simulator focuses on simplicity and efficiency. A standard form of simulation output has been developed and a set of output analysis tools (plotters and statistical packages) are planned. An "animator" is also planned to take advantage of

high-resolution bit-map graphics. This animator is intended to provide graphical representations of the operation of Petri Net models.

Conclusions

A set of useful and efficient modeling and analysis tools has been developed at UCI as part of the P-NUT system. The tools allow designers to construct Petri net models which have been extended to support timing and interpretation. These models can be exhaustively analyzed (with and without time) and the results of the analyses can be presented to the user in a flexible form. Simulation experiments can be used to traverse selected portions of the total system state-space. The tools have been used to verify some simple communication protocols (alternating-bit and X.21) and are currently being used to derive performance measures for Intel's 286 processor (a pipelined machine) and for a multiprocessor system.

Work is continuing on enhancements to the tools. Among the planned tools are a Graphics Editor, an Animator, a Performance Analyzer and a variety of Plotting and Performance Statistics tools. The work on P-NUT is, and will continue to be, in the public domain. The tools are highly portable and currently require a C compiler and and UNIX operating system. The tools execute directly on VAXes running UNIX 4.1, 4.2 and LOCUS.

Acknowledgements

The authors would like to acknowledge the member of the P-NUT group for their contributions to this research: James Fradkin, Charles Phelps, Richard Sidwell, and David Woo. The authors would also like to acknowledge the contributions of Dr. Daniel Hirschberg.

References

[Agerwala 79] Agerwala, T. "Putting Petri Nets to Work," *Computer*, December 1979, pp. 85-94.

[Berthelot 82] Berthelot, G. and Richard Terrat, "Petri Net Theory for the Correctness of Protocols," *Protocol Specification, Testing and Verification*, North Holland Pub. Co., (1982).

[Berthomieu 83] Berthomieu, B. and Menasche, M. "An Enumerative Approach for Analyzing Time Petri Nets," *Proceedings of the* 1983 *IFIP Congress*, Paris (Sept. 1983).

[Diaz 82] Diaz, M. "Modelling and analysis of communication and cooperation protocols using Petri Net based models" *Protocol Specification, Testing and Verification*, C. Sunshine (ed.) North-Holland Publishing Co., 1982.

[Holliday and Vernon 85] Holliday, M. and M. Vernon "A Generalized Timed Petri Net Model for Performance Analysis of Pipelined Architectures", To appear in the proceedings of the *International Workshop on Timed Petri Nets*, Torino Italy, July 1985.

[Molloy 82] Molloy, M., "Performance Modeling Using Stochastic Petri Nets," *IEEE Trans. on Computers*, Vol. C-31, pp. 913-917, Sept. 1982.

[Morgan 84] Morgan, E.T. "RGA Users Manual" Technical Rept. No. 243, Information and Computer Science Dept., University of California, Irvine, December 1984.

[Morgan and Razouk 85] Morgan, E.T, and R. R. Razouk, "Computer-Aided Analysis of Concurrent Systems," *Proceedings of the 5th International Workshop on Protocol Specification Verification and Testing*, Toulouse, FRANCE, June 1985.

[Peterson J. 81] Peterson, J., *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, Inc., Englewood Cliffs, N.J. (1981).

[Ramamoorthy 80] Ramamoorthy C.V. and G.S. Ho, "Performance Evaluation of Asynchronous Concurrency Systems using Petri Nets," *IEEE Transaction on Software Engineering*, SE-6, 5 (September 1980), 440-449.

[Ramchandani 74] Ramchandani, C. "Analysis of Asynchronous Concurrent Systems by Timed Petri Nets," Ph.D. Thesis, Project MAC Report No. MAC-TR-120, MIT (1974).

[Razouk and Phelps 84] Razouk, R.R. and C. Phelps "Performance Analysis Using Timed Petri Nets," *Proceedings of the 4th International Workshop on Protocol Specification, Testing, and Verification*, June 1984.

[Razouk and Hirschberg 85] Razouk, R.R. and D.S. Hirschberg "Tools for Efficient Analysis of Concurrent Software Systems" Technical Report No. 85-15, Information and Computer Science Dept., University of California, Irvine, June 1985.

[Sifakis 77] Sifakis, J. "Petri Nets for Performance Evaluation," *Measuring, Modeling and Evaluating Computer Systems*, Proceedings of the 3rd Symposium, IFIP Working Group 7.3, H. Beilner and E. Gelenbe (eds.), North Holland, 1977, pp. 75-93.

[Symons 80] Symons, F.J.W., "Verification of Communication Protocols using Numerical Petri Nets," *Australian Telecommunication Research*, 14,1 (1980) 34-38.

[Zuberek 80] Zuberek, W.M., "Timed Petri Nets and Preliminary Performance Evaluation," *7th Annual Symposium on Computer Architecture*, (1980), pp. 88-96.