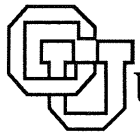
Trace Driven Petri Net Simulation

Gary J. Nutt

CU-CS-569-91 December 1991



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Trace Driven Petri Net Simulation

Gary J. Nutt

CU-CS-569-91 December 1991

University of Colorado at Boulder

Technical Report CU-CS-569-91
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

Trace Driven Petri Net Simulation

Gary J. Nutt

December 1991

Abstract

Petri nets have been used as a medium for constructing simulation models of software and computer systems for several years. Loads in Petri net models are typically described by stochastic mechanisms that describe the external loading on the model and the flow of tokens through the model.

In traditional simulation modeling, stochastic representations of resource loading are criticized because of their failure to accurately represent individual resource requests and, more importantly, their failure to correlate sets of resource requests that result from realistic requirements on the system.

Trace driven simulation is a technique for explicitly representing resource loading in a model. In this paper, we describe a technique for combining trace driven simulation techniques with Petri net simulation. Our example focuses on a particular application area – the execution of parallel programs on parallel computer systems. However, the method for combining traces with Petri net token flow is applicable to arbitrary Petri net simulation.

Contents

1 Introduction

Petri nets have been used as a medium for constructing simulation models of computer systems for several years. The Petri net places can describe resources in the system with the relationship between the use of different resources being represented by arcs and transitions. An instance of the execution of a model is represented by the flow of tokens through the graph. Thus tokens represent the flow of control and/or data among components in the model, i.e. if the graph represents resources, then the software execution is reflected in the pattern of token flow. This view of Petri net models leads naturally to data flow models, with data flowing from processing unit to processing unit.

Petri nets may also represent algorithm and software execution with a dual of the above: the Petri net places, transitions, and arcs can be a control flow diagram for a program, representing sequential execution of tasks (e.g., basic blocks), alternative control flow paths among tasks, and concurrency. In this case, the execution of the program/algorithm in a particular environment is represented by the rate and pattern of the flow of tokens through the graph. Thus the underlying resource utilization by the executing program can be implied from the token flow pattern. This use of Petri nets fits naturally with the study of algorithms and software.

Ferscha explicitly combines the two forms of model in his PRM nets [8]. A program-oriented model is complemented by places to represent resource availability, thus making computing resources explicit in the program representation. The basic idea behind the formation of our models is the same as for PRM models, although the trace driven modeling is unique to our work.

As simulation models, both approaches require that time be introduced to the model to represent the amount of time that resources are held. This requires that the basic Petri net firing and marking rules be enhanced. A function, f_i, is associated with each node, i; f_i may be used to define an amount of time, $f_i(x)$ (for an arbitrarily determined x) to "hold" a token on the corresponding node independent of the marking and firing rules. In some extensions, node i must be a place [19, 13], but in others it may be a transition or a place, e.g., see [14].

In many timed Petri net variants, f_i is a probability density function that can be sampled to yield a holding time [19, 13]. Other variants might allow more general usage of f_i, although the function may also be used as a stochastic mechanism for representing the time [3, 10, 11, 14].

In traditional simulation modeling, stochastic representations of resource loading are criticized because of their failure to accurately represent individual resource requests and, more importantly, their failure to correlate sets of resource requests that result from realistic requirements on the system.

Trace driven simulation (tds) is a technique for explicitly representing resource loading in a model. That is, an instance from a family of target systems is instrumented and its behavior is carefully observed. The observation is represented by recording a sequence of event occurrences in the execution of the instrumented system; thus, the event sequence – or *event trace* – describes the execution of the system. The event trace represents, among other things, the resource loading on the system. A tds is a simulation of other instances of the same family of target systems – instances that are similar to the instrumented instance, but differing in the amount of various resources available in the system. The tds uses the event trace to define the resource requests on the model; the model reacts to this event trace to simulate the behavior of the system with a different resource configuration but the same external load.

Tracing has been a fundamental performance evaluation methodology since the 1960s. Sherman et al. used traces of application programs to study their effect on operating systems [20]; Mattson, et al. used page reference traces to characterize the load of a single program on different page replacement algorithms [18]. By the late 1970s, Smith used traces to study the behavior of cache systems [22]. Currently traces are widely used to characterize memory references in studying the design of memory hierarchies in shared memory computer architectures (e.g., see [1, 6]).

In this paper, we describe a technique for combining trace driven simulation techniques with Petri net simulation. We are motivated to do this work by our desire to use Petri net models to represent the behavior of parallel programs on parallel computer systems. Therefore, our methodology focuses on parallel program and system models. However, the method for combining traces with Petri net token flow may be applicable to other Petri net simulation applications.

Since traces constrain the activity in the model, they provide an additional constraint to the firing conditions in the model, i.e., just as timing altered the basic marking and firing behavior of a net, traces must do the same, but using a mechanism other than a pure hold time function. The problems that we have had to overcome to use traces were to be able to correlate trace events with model components, then to define rules analogous to timing rules to introduce the trace effects to the firing behavior of the net.

We have solved the first problem by generating the model from the same source that defines the trace events. The second problem is addressed by altering the firing rules in an execution of the Petri net.

In the next section we outline our environment, assumptions, and general approach. Then we describe a tool used to instrument a subject parallel program and to generate abstract traces of events. The next section describes how tools are used to generate a Petri net model that has nodes corresponding to events that will be in the trace. Finally we describe how the trace is combined with the marking to schedule execution of the model.

2 The Approach

2.1 Building a Coordinated Model and Trace

The class of simulation problems on which we focus can be characterized as the study of behavior of parallel programs executing on parallel computer systems. In particular, we address a modeling environment in which one uses a Petri net simulation model of a particular configuration of a hardware configuration, operating system, and runtime system (called the *execution architecture* that can be simulated using loading conditions described by an event trace derived from the execution of a parallel program on a related execution architecture.

Using traces derived from the execution of a program on one configuration of a system to experiment with models of a related configuration are fundamental to trace-driven simulation. The accuracy of the trace model is reflected by the complexity of correlating the instrumentation, events, and instrumented and simulated execution architectures. "Good" trace-driven simulation studies implicitly resolve all of these correlations so that traces of realistic programs can be used to drive simulation models.

We believe that trace-driven simulation is a useful tool for studying the behavior of a parallel program within an execution architecture family, but that there is no systematic way to construct models so that they accurately represent the program load and different configurations within an execution architecture, i.e., so that they correlate the trace data from the instrumented execution architecture configuration with a modeled configuration.

The fundamental premise of our approach is that interpreted Petri nets can be used to represent the control flow characteristics in the parallel program, e.g., representing the best parallelism that could be obtained with an unbounded number of processors. Further one can obtain traces containing events to drive the execution – dictate the flow of tokens – of the Petri net for the instrumented configuration. Each path that a token traverses through the Petri net represents the control flow of a sequential unit of computation; thus the dynamic behavior of the Petri net model represents the behavior of an execution architecture for the particular program.

Suppose that we wish to model the behavior of a parallel program written in C using a threads library. The general tasks in our approach are as follows (represented pictorially in Figure 1).

1. Use the SPAE compiler [5] to translate the program into object code modules with embedded instrumentation. SPAE also identifies basic blocks of computation in the program.

2. The instrumented program is executed on a uniprocessor configuration of the target execution architecture (with an instruction set corresponding to the target execution architecture) to generate a trace of the logical parallelism among the basic blocks.

3. Each basic block is mapped into a Petri net place; edges and transitions are added to reflect the control flow among basic blocks, generating a Petri net model of the program. (If the degree of

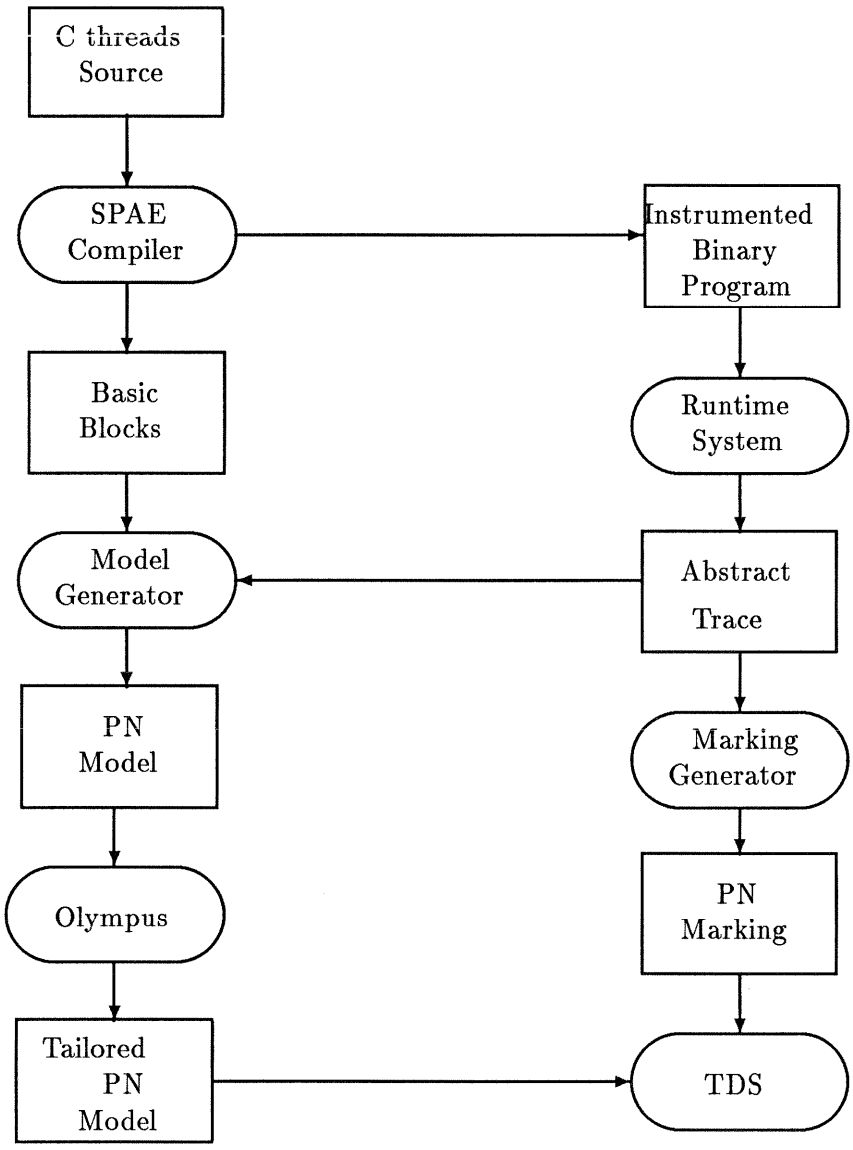


Figure 1: Trace Driven Petri Net Modeling

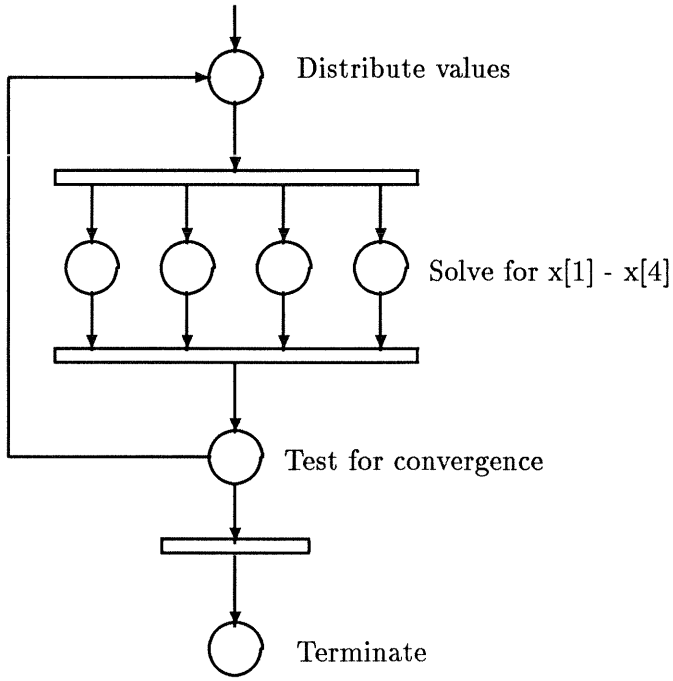


Figure 2: Represent SOR with a Petri Net

parallelism is determined at runtime, then part of the trace data is used to determine the degree of parallelism in the model.)

4. The trace data is correlated with basic blocks and is translated into a temporal history of markings in the Petri net.

5. Use Olympus [9] to generate a tailored version of the model such that places and transitions remain consistent with those identified in the trace.

6. The markings of transition firings drive the Petri net simulation model.

2.2 An Example: Successive Overrelaxation

In this subsection we provide a simplified example of how we use Petri nets to represent a computation and the execution architecture that supports it. Our example is the successive overrelaxation technique for solving an n x n linear system of n equations in n unknowns; we are interested in the execution of the program on an N processor system.

A successive overrelaxation (SOR) solver for a 4x4 linear system of equations can be represented by the interpreted Petri net (similar in character to *predicate/transition nets* (P/T nets) [10]) shown in Figure 2. Algorithmic iteration is represented in the model by the flow of tokens through the transitions and places shown in the figure.

The interpretation for predicates in the model is supplied by a procedure, e.g., Figure 3 is an interpretation for the predicate labeled "Test for convergence".

Petri nets do not explicitly represent the information flow among components of the model, e.g., among the individual solvers. However, we have used Petri net variants to describe such information flow, first in information control nets [3], then in bilogic precedence graphs [16] and distributed computation

```
predicate Test_for_convergence()
{
  <Obtain copies of n, A, b, and current approximation for x>
  error = 0.0;
  for(i=0; i < n; i++)
  {
    sum = 0.0;
    for (j=0; j < n; j++)
    {
      sum = sum + A[i][j] * x[j];
      error = error + (sum - b[i]);
    };
  };
  if(error < 0.0) error = -error;
  <Compute token flow out of this predicate as a function of error>
}
```

Figure 3: A Node Interpretation

precedence graphs [4]. These extensions rely on node interpretations in the net, similar to those employed n P/T nets.

Suppose that we had obtained a trace of event occurrences, where each event corresponded to a token moving onto or off of a place in Figure 2. That is, if "I(*event*)" represents the event that a token entered the place corresponding to *event* and "T(*event*)" represents a token leaving the place, the trace would have the form:

```
I(Distribute values)
T(Distribute values)
I(Solve for x[1])
I(Solve for x[2])
I(Solve for x[3])
I(Solve for x[4])
T(Solve for x[3])
T(Solve for x[2])
T(Solve for x[4])
T(Solve for x[1])
I(Test for convergence)
T(Test for convergence)
I(Distribute values)
T(Distribute values)
 ...
I(Test for convergence)
T(Test for convergence)
I(Terminate)
T(Terminate)
```

The trace will implicitly serialize all activity, e.g., the event that a token leaves the place labeled "Solve for x[3]" (also referred to as the *termination* of activity on the place) occurs before the termination of "Solve for x[2]" even though they may occur simultaneously.

If the trace were taken from an execution of the corresponding program on an execution architecture with a uniprocessor (with nonpreemptive scheduling), then the concurrency among solvers would have the form:

```
 ...
I(Solve for x[1])
T(Solve for x[1])
I(Solve for x[2])
T(Solve for x[2])
I(Solve for x[3])
T(Solve for x[3])
I(Solve for x[4])
T(Solve for x[4])
 ...
```

However, if the execution architecture for the trace has N = 3 CPUs, then three solvers can be active at one time, yielding traces of the form:

```
 ...
I(Solve for x[1])
I(Solve for x[2])
I(Solve for x[3])
T(Solve for x[2])
T(Solve for x[1])
T(Solve for x[3])
I(Solve for x[4])
T(Solve for x[4])
 ...
```

Notice that a correct trace will always initiate all four solvers, then terminate them prior to the occurrence of "I(Test for Convergence)" for a model of the 4 x 4 system, independent of N. Also notice that the pattern of token flow that enables the 4-input-1-output transition is determined by the trace.

3 Trace Generation

,

The nature of the events that are captured in a trace reflects the characteristics of the behavior in which the observer is interested. For example, if one were studying the performance of the cache system, then an event might correspond to a memory read or write operation. Alternatively, one might capture the sequence of operating system or library calls made by the program if one were studying the operating or runtime system.

The resulting event trace can be analyzed to infer the behavior of the program on the given execution architecture, or it can be used to represent execution of the original program on a simulation of a distinct execution architecture.

Detailed event traces of running programs (e.g., traces of memory references) may produce voluminous observation data. Therefore, a popular technique for avoiding the need for such data storage is to analyze the event trace sequence and then to characterize the behavior with probability distribution functions. Such functions capture the aggregate behavior of the program in the context of a particular execution architecture, but it is difficult to represent causal relationships that might exist among individual events using distributions. Early page reference stream traces quicky identified the inappropriateness of aggregate representations for comparing page replacement algorithms. However, synthetic programs that interconnect patterns of calls on different distributions to represent the behavior of real programs can be useful for performance studies of CPU and device utilization.

3.1 Recording Traces

The SPAE tool in the PEET testbed [5] provides us with a mechanism to record event traces for driving Petri net models. The tool provides a means for correlating the trace with model components by using it to define the trace and the base model for a particular program.

SPAE extends the ideas used in AE (due to Larus [12]) to recognize and trace *symbolic, parallel abstract event* occurrences. Abstract events are analagous to events that a hardware logic analyzer can be programmed to recognize: in the case of a logic analyzer, the events are generally some logical combination of signals that exist at any given time; in the case of software abstract events, the combination may be some serial string of occurrences within a particular context.

Recognizing abstract event occurrences by scanning event occurrences is an undecidable problem (analagous to recognizing a program by recording and analyzing its program counter locus). However, the trace collector can recognize many abstract events if they are marked at compile time.

AE is a modified C compiler that reads a C source program and ultimately produces a condensed set of abstract events from an execution of the program, and another program that can read the abstract events then produce a trace of concrete event occurrences in the execution. SPAE is intended to apply AE to multithreaded program traces, and to abstract nondeterministic and execution architecture independent events out of the trace.

In particular, the SPAE compiler reads a file containing a C program, say program.c, and produces an AE schema file called program.sma and an object file, program.o, with the compiled code with instrumentation. The program.o file, when linked with AE procedures, will produce a file of significant events as determined by the trace information to be collected, e.g., basic block entry and exit. The program.sma file describes how the trace information collected by the execution of program.o can be used to regenerate a full trace; before program.sma can be used, it must be translated into a C program by the AEC schema compiler, then combined with an arbitrary program that consumes the trace data.

AE abstract events ordinarily relate to machine instruction execution, and not necessarily to high level abstract machine execution, e.g., a system call. In SPAE, there is an assumption that programs are written on top of a parallel programming platform such as the Argonne Fortran macros or a C thread package. These libraries are instrumented to mark the occurrence of thread events such as creation, locking, and so on. This results in the injection of an abstract event in place of a thread primitive call.

3.2 Expanding Abstract Events

The trace data collected by SPAE records abstract events as opposed to concrete events. Each abstraction is performed by the instrumentation software to allow a uniprocessor to represent logical concurrency specified by the thread package. An operation such as **fork** induces an abstract event in the trace stream; the event specifies that a new *context* has been created, i.e., a new stream of concurrent operation has been introduced. Trace information is correlated with its context, so that the abstract trace incorporates blocks of context traces.

The other form of abstract event is one which is resolved by conditions that exist in a specific execution architecture. For example, a context may obtain a spinlock only after many test-and-set operations on the lock. The abstract trace will include only the event of obtaining the lock, and none of the unsuccessful attempts.

Thus, the abstract trace cannot be used to drive any simulation until each abstract event is bound to some real execution architecture – by simulating the action of the execution architecture. (The simulation may be complex, as in [7], or simple as in the SPAE documentation [21].)

For the Petri net application, each place corresponds to a basic block in the original program. Therefore, a meaningful trace might include events for the placement/removal of a token on/from a place. Each **fork** event notes the creation of a new thread; the thread library instrumentation also leaves information in the trace file to identify which thread is active (producing events in the trace) at any given point in the trace. Thus, the abstract trace explicitly contains a description of all active threads and their state of activity; the threads can be bound to an execution architecture with a specific number, N, of processors by simulating the thread scheduling algorithm with the specified number. This

defines the sequence of place initiation/termination events in the event trace for the computation on each of the N processors (independent of the number of threads of computation).

We choose not to bind execution times into the trace, saving only the temporal sequence. Thus, the trace does not include timestamps, only ordering of event occurences for token movement in the Petri net. Finally, as we discuss in Section 5, the trace file need only specify the events that correspond to the entry of a basic block.

4 Generating the Model

A Petri net represents the control flow in the execution of the program, but does not represent performance until some metrics are introduced to the model. By controlling the movement of tokens through the Petri net using timing and trace information, the control flow model represents the execution of the program on a specific execution architecture. Notice that the nature of program's demands on the execution architecture can be represented by the structure of the Petri net while the sequence of activity is represented by the timing and trace information and the token flow in the net. The performance of a program for a specific set of data on a specific execution architecture can be modeled by combining basic timing information, the temporal trace, and the Petri net; we return to discuss timing in the next section.

Basic Petri nets are nondeterministic: in Figure 2, there is nondeterminism in the token flow out of the place labeled "Test for convergence" since a token may flow back through the iterative portion of the net, or flow through the transition to the "Terminate" place (a similar situation occurs in the aforementioned place labeled "Loop if i less than n"). Of course the interpretation can resolve this nondeterminism in Predicate/Transition nets and other interpreted Petri nets. However, we use the trace information to resolve such cases.

We also use SPAE (with other tools) to generate a canonical Petri net model – it would also be possible to generate a colored Petri net or a P/T net, if it served our purposes. (Ferscha uses a similar techique to derive the program portion of his PRM nets [8].) The canonical model typically has more detail that we might want for considering the behavior of the program as we vary the number of processors, or it may have a representation of parallelism that is not amenable to performance studies. We use PN-Olympus [9, 15] to manually transform the model to one that meets the analysts specific needs. The model can then be used for trace driven simulation studies.

4.1 Deriving the Base Model

Static parsing and parallism detection algorithms can be used to contruct a basic Petri net from a source program with suitable parallel constructs. However, in other cases the degree of parallelism is not determined until runtime, when dynamically-determined values establish the degree of parallelism. For dynamically-determined parallelism, the program must be observed during execution to detect the degree of parallelism.

Each place in the Petri net represents a basic block of computation, while arcs and transitions identify control flow among these subcomputations. Using the SPAE compiler, one can analyze the resulting schema file to derive the basic blocks that exist in the source program. For example, a C program with threads that solves an n x n linear system of equations using the SOR technique is shown below (the tools were incomplete at the time of this writing; the actual source code processed by SPAE is shown in Appendix A.):

```
#include   <stdio.h>

#define N   3
#include   "doall.h"

#define   MAX_N   4
```

```
/* Fake cthreads mutex_t type */
typedef struct m_t
{
   int   value;
} mutex_t;

mutex_t lock[MAX_N];
float A[MAX_N][MAX_N], b[MAX_N], x[MAX_N];

main(argc, argv)
int argc;
char *argv[];
{
   FILE *data_fp, *fopen();
   int i;
   int tmp;
   double double_eps;
   float error, check();
   float epsilon;
   char data_filename[128];

   double_eps = atof(argv[1]);
   epsilon = double_eps;
   strcpy(data_filename, argv[2]);

   printf("Opening %s ...\n", data_filename);
   data_fp = fopen(data_filename, "r");
   fscanf(data_fp, "%d", &tmp);
   for (i=0;i<N;i++)
      fscanf(data_fp,"%f%f%f", &A[i][0], &A[i][1], &A[i][2]);
   fscanf(data_fp,"%f %f %f", &b[0], &b[1], &b[2]);
   fscanf(data_fp,"%f %f %f", &x[0], &x[1], &x[2]);
   fclose(data_fp);

   DOALL (i, N);
      slave(i);
   ENDDOALL(i, N);

   error = 1000000.0;
   while(error > double_eps)
   {
      for (i=0; i<N; i++)
      {
         while(mutex_try_lock(lock[i]))
         {
            mutex_unlock(lock[i]);
         };
      };
      error = check(A, x, b, N);
      for (i=0; i<N; i++)
         mutex_unlock(lock[i]);
```

```
   };

   printf("Solution x[*] = %f %f %f\n", x[0], x[1], x[2]);
   printf("   With error factor = %f\n", error);
   exit(0);
}

float check(A, x, b, n)
float A[][4], x[], b[];
int n;
{
   int i, j;
   float sum;
   float error;

   error = 0.0;
   for(i=0; i<n; i++)
   {
      sum = 0.0;
      for (j=0; j<n; j++)
         sum = sum + A[i][j]*x[j];
      error = error + (sum - b[i]);
   };
   if(error<0.0) error = -error;
   return(error);
}

mutex_lock()
{
}

mutex_try_lock()
{
}

mutex_unlock()
{
}

slave(me)
int me;
{
   int j;

   for(;;)
   {
      mutex_lock(lock[me]);
      x[me] = b[me];
      for (j=0;j<N; j++)
         if(me!=j) x[me] = x[me] -  A[me][j]*x[j];
      x[me] = x[me]/A[me][me];
   };
```

}

The SPAE compiler produces a schema file (.sma) file and a set of instrumented object modules. The instrumented object modules can be executed on a serial machine, gathering dynamic trace information that is correlated with statically inferred information about the program that is saved in the .sma file. By processing the .sma, one can infer a Petri net of the form shown in Figures 4 5 (hand drawn in this example).

The raw Petri net reflects the basic blocks that the compiler finds in the subject program, where each place is labeled with the number of the corresponding basic blocks in the subject program. (For example, basic block 6 in the main program is the body of the **DOALL** loop, which calls **slave**). We have taken some freedom by representing procedure calls to "slave" and "check" by an ellipse with a start and stop node.

After the instrumented program has been executed, the instrumentation will have captured all dynamically-determined values that effect the degree of parallelism. For example, if N had been input data rather than a defined constant, the trace file can be analyzed to determine N. This value can be used in the simulation to control the number of iterations through basic block 6 in Figure 4, or to transform the model into a static structure similar to the one shown in Figure 2. Currently, we perform the transformation manually, although we plan to build a tool capable of dealing with complex programs that can perform such transformations algorithmically.

4.2 Tailoring the Model

There is considerable detail in the Petri net model – at least one place for each basic block of source code. Many of these basic blocks are of importance to the compiler, but not for considering parallelism in the code.

It is easy to recognize various control flow constructs in the Petri net (which is purely sequential, since the DOALL macros generate a **for** loop.). In general, the information from the trace file (generated when the instrumented program is executed) might define N, i.e., essentially defining the number of times that control flows through the loop of places 5, 6, and 7. The right side of Figure 6 represents this case for a fragment of the model in Figure 4. It is also possible that the value for N is defined at compile time (as is the case in the example source code), so it is conceptually possible to draw a Petri net which explicity specifies the parallelism among with replicated copies of places 6 and 7 as shown on the left of Figure 6.

The Petri net shown in Figure 4 can be modified using either the static or dynamic information to incorporate one of the two versions shown in Figure 6.

Figure 7 collapses several of the sequential loops into single places and illustrates a Petri net in which N is known at compile time. Figure 8 represents the case where N is defined at runtime.

In these cases, the tailored Petri net is derived from the original Petri net by some set of simple transformation, e.g., abstracting a single-entry-single-exit subgraph into a square node. We have performed these operations manually in the example, although it is clear that nearly the same transformations could be applied algorithmically.

The resulting transformed Petri net is equivalent to the original Petri net, provided that the definition of each square node is the corresponding SESX subnet that it replaced.

Transformations on a derived Petri net must result in a tailored model that is consistent with the trace information. Currently our transformations are manual; we plan to build other tools to ensure equivalence among the various transformations.

5 Trace Driven Petri Net Execution

We have used the Olympus modeling system (derived from earlier work on the Quinault system [17]) to implement a timed Petri net simulation facility, similar in functionality to Balbo's GreatSPN system [2], and others.

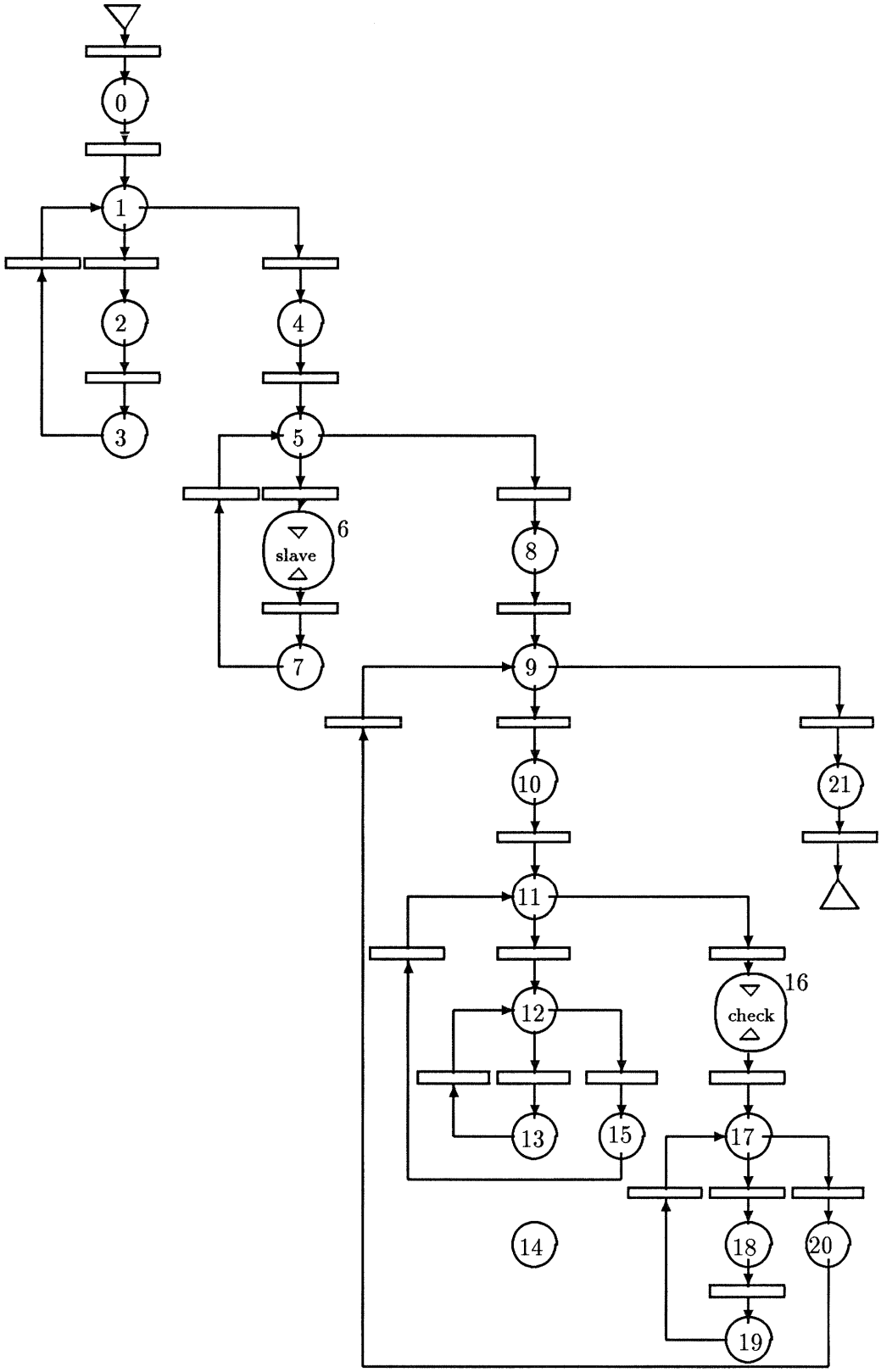


Figure 4: Petri Net of the main Procedure

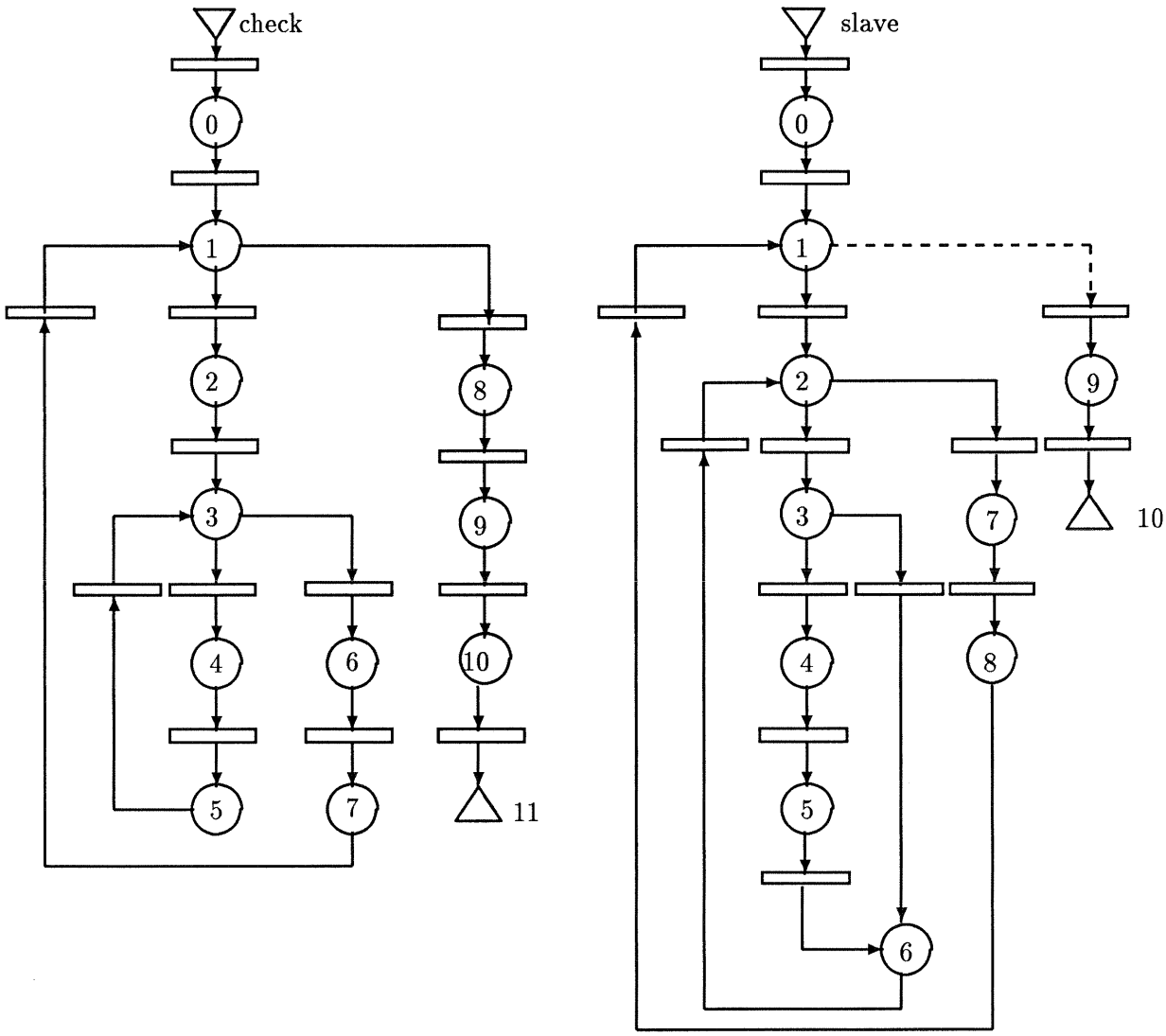


Figure 5: Petri Net of the check and slave Procedures

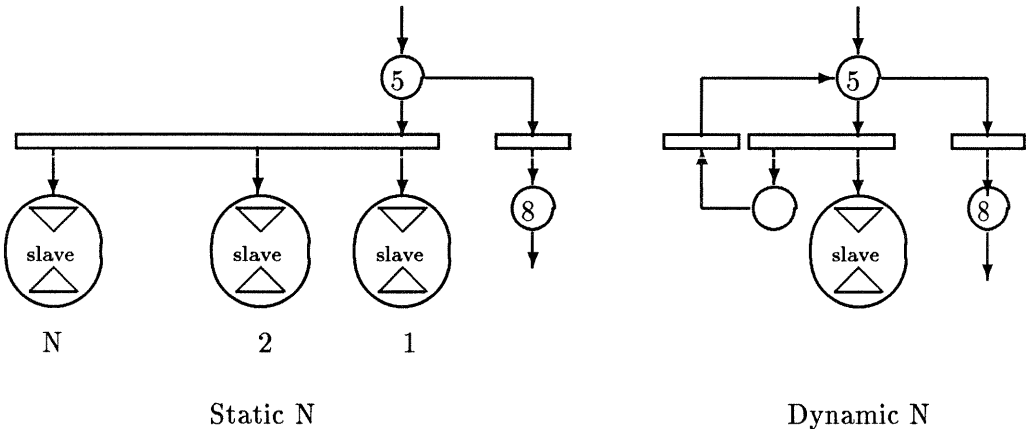


Figure 6: Dynamic vs Static N

In these systems, a variant of a Petri net defines a simulation model in which the time for node i to "execute" is defined by some function, f_i. In GreatSPN, f_i is determined by associating a distribution with place i in the net; when a token arrives at place i, a sample from the distribution function, f_i, determines the minimum length of time that the token will reside on the place before it participates in any enabling activity for downstream transitions. Note that even though the token may participate in enabling a downstream transition after the prescribed time, it is not removed from the place until the downstream transition actually fires. E-nets [14] and Quinauilt use a similar technique, although the sampled time may be computed by evaluating a function procedure rather than sampling a distribution; they also allow tokens to reside on arcs in the net in cases where the dwell time on a place has elapsed but the downstream transition is not enabled (this only occurs in the case where the transition has multiple input places).

In PN-Olympus, a Petri net is interpreted with the same technique as used in E-nets and Quinault:

```
time = 0;
while(any transition is firable)
{
  while(simulation events to process at current time)
  {
    switch(event type)
    {
    put token:
      Move token from input arc to output place(s);
      Execute interpretation;
      Schedule removal at current time + computed dwell time;
    remove token:
      Move token from place to output arc;
      Enable transition, if possible;
    };
  }
  Advance time;
}
```

A simulation event is one of the following actions: move a token onto a place, move a token onto an arc, or move a token onto a transition. Simulation events are enqueued whenever the system determines

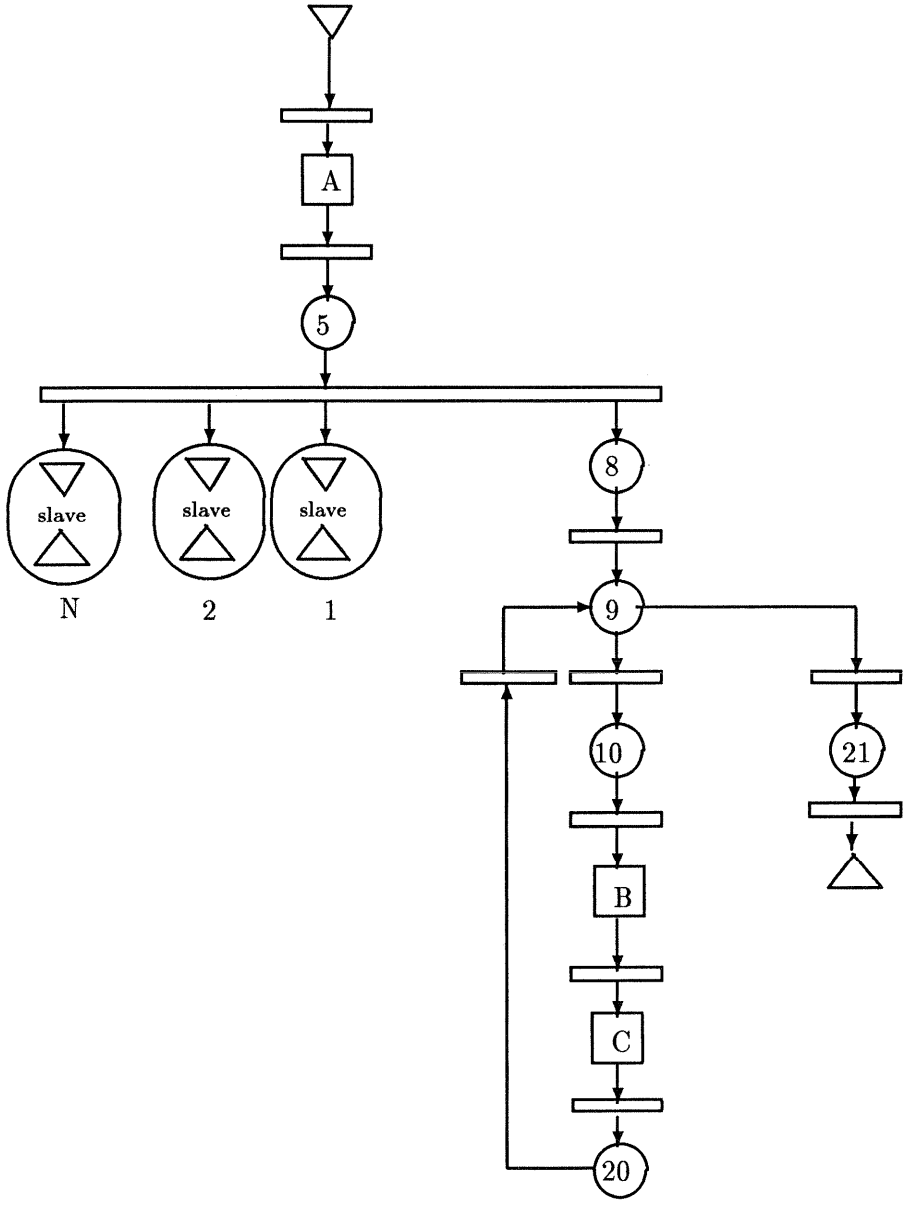


Figure 7: Abstracted Petri Net Using Static N

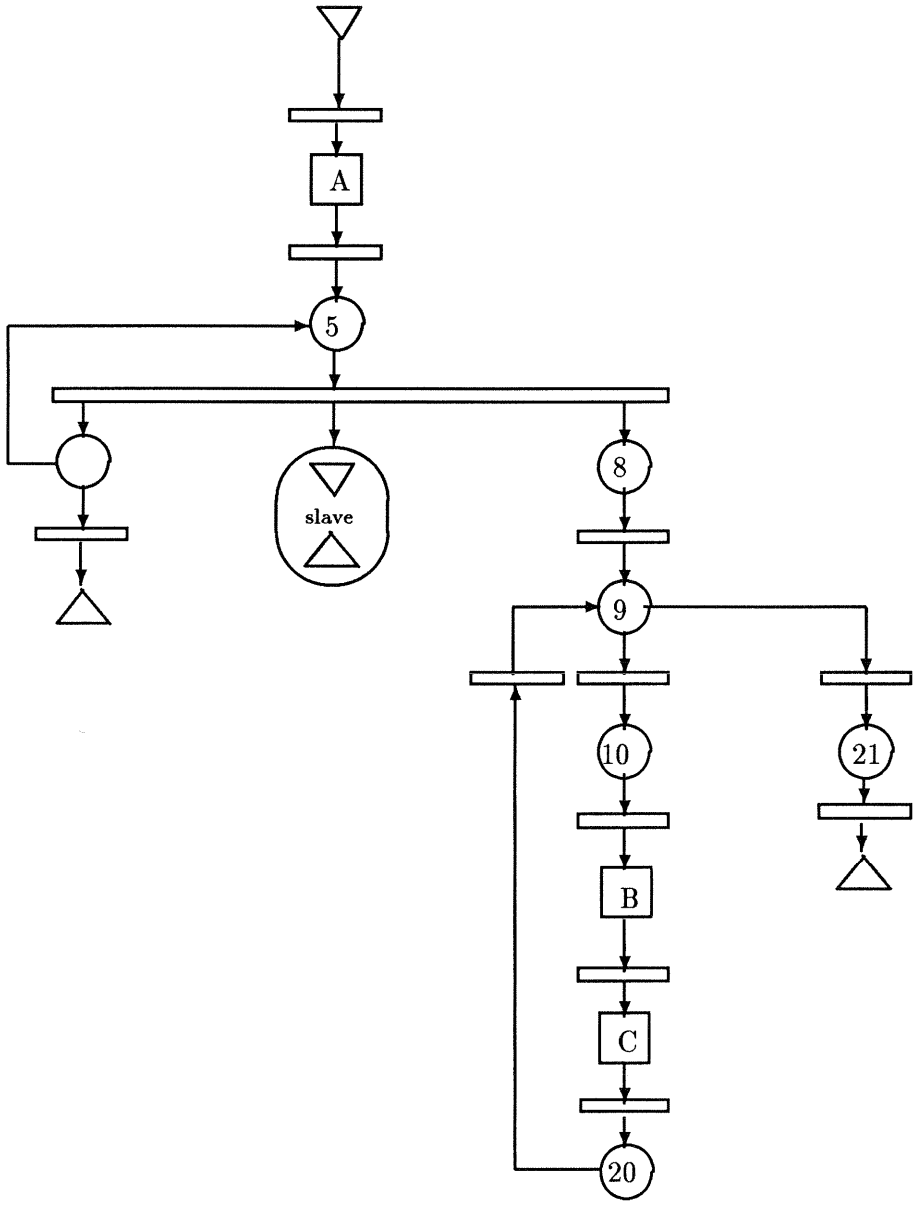


Figure 8: Abstracted Petri Net Using Dynamic N

that something should be scheduled to happen at some future time; this means that the time at which a token is to be removed from a place is computed when the token is put on the place and the departure simulation event is scheduled at that time.

Trace driven Petri nets cannot rely completely on the timing mechanism nor the place interpretation, since the temporal ordering of simulation event occurrences is determined by information held in the trace. Neither can they rely completely on the trace to control the firing of transitions, since the trace events have no timing information (only temporal relations); the trace may also call for a transiton to occur when the marking does not currently enable the transition.

Traces could be translated to include unit timing information; however, we believe that the timing information should be determined by the simulation system rather than by the trace. We want the simulation system to support changing of model parameters at the user interface rather than by transforming the trace to reflect a new configuration. (This follows the rest of the philosophy for supporting model experimentation with Olympus.)

As we described in Section 3, the trace information from SPAE is bound to an N-processor architecture before it is used to drive the Petri net simulation. Even so, there may be n concurrent threads of control in the program, so the trace will be composed of up to n sequential traces, T_1, T_2, ..., T_n. But since there are N processors, the act of binding threads to processors will have explicitly ordered the events in the T_i into sets of N simultaneously active traces, T_{j_1}, T_{j_2}, ..., T_{j_N}, where events in these traces are independent. The simulator will receive the trace information on the N logical streams.

Each trace event represents the initiation of a basic block, i.e., that the upstream transition for the place fired. In the case where a place is one of several outputs from a transition, the transition must not fire in the simulation until there is a traced event corresponding to the placement of a token on *every* output of the transition. This will result in cases where a trace, T_{j_k}, specifies that a token be put on a place, but the upstream transition is not enabled. In these cases, the next trace event on the stream will refer to a different thread that has been multiplexed onto T_{j_k}. This represents a synchronization across trace streams.

The PN-Olympus token movement algorithm uses normal Petri net firing rules to cause a transition and to put tokens on places. When a token is put on a place, then the interpreter evaluates the timing function for the token and the place and schedules a departure at some time in the future. A token leaves a place after the simulated amount of dwell time has elapsed, i.e., when it is time for the departure event to occur. Thus the Olympus interpreter keeps a list of timestamped, pending simulation events (each corresponding to a pending departure).

The trace-driven Petri net simulator (TDPN-Olympus) is made ready to simulate a net by loading an initial marking, by determining the value for N, and by opening the N trace files, T_{j_1}, T_{j_2}, ... T_{j_N}. Simulation events are executed as follows:

```
time = 0;
while(any transition is firable)
{
  Read pending trace event, E[i], for each trace stream;
  while(There is any pending trace event, E[i])
  {
    Advance time if simulation event time is greater than current time;
    if(upstream transition is enabled)
    {
      switch(event type)
      {
      put token:
        Move token from input place(s) to output place(s);
        Execute interpretation;
        Schedule removal at current time + computed dwell time;
      remove token:
        Mark token as moveable;
```

```
        Pseudo enable transition, if possible;
      };
      Read new E[i];
    }
    else
      Quit;
  }
}
```

The information in the trace is used to define the order in which tokens are put on places, but not the order in which they depart places. Once a token is put on a place, the normal timing mechanism is used to compute the dwell time of the token. When it is time for the token to depart, it is removed from the place and put on the input of a transition node. In an ordinary Petri net interpretation, this would cause the transition to fire if the placement enabled it; in the trace driven simulator, there may be many enabled transitions that the interpreter does not fire at any given moment. The trace determines if all output places of the newly enabled transition should receive tokens, i.e., if the transition should fire; if so, the transition is fired, otherwise the transition is left enabled and the event from the trace is buffered.

The result is that transitions are fired in the order implied by the trace data, even though the dwell times of tokens on places is determined by the model rather than by the trace.

6 Conclusion

Trace driven simulation of Petri nets allows one to build models that drive the model with a realistic load. In our application, the base model and the trace are derived from a single source, allowing trivial changes to the base model to reflect the performance of a parallel program on a collection of related execution architectures where the differences among members is in the number of processors applied to the problem. It is also possible to experiment with various speeds of processors by manipulating dwell times in the model. We see this as being a valuable testbed for experimenting with execution architectures and for tuning parallel programs for a particular execution architecture family.

We have limited experience with the trace driven Petri net simulation system. At the time of this writing, we are only able to derive Petri nets and traces for modest examples since many of the requisite tools are not yet developed. In particular, the SPAE development must continue to the point that there is an instrumented version of the C library; then the model generator can create models of programs that reflect the full concurrency in a program. The trace generator is not implemented; our experiences have been based on manually created trace files. The most interesting extensions we see to the work are related to transformations of the base Petri net so that the traces can still drive it. Here it is necessary to find ways to be able to apply similar transformations to the model and the trace, while preserving the integrity of both.

References

[1] A. Agarwal, J. Hennessy, and M. Horowitz. Cache performance of operating system and multiprocessing workloads. *ACM Transactions on Computer Systems*, 6(4):393–431, November 1988.

[2] G. Balbo and A G. Chiola. Stochastic petri net simulation. In *1989 Winter Simulation Conference Proceedings*, pages 266–276, Washington, D. C., December 1989.

[3] Clarence A. Ellis and Gary J. Nutt. Office information systems and computer science. *ACM Computing Surveys*, 12(1):27–60, March 1980.

[4] Isabelle M. Demeure and Gary J. Nutt. Collected papers on visa and paradigm. Technical Report CU-CS-488-90, University of Colorado, Department of Computer Science, CB 430, August 1990.

[5] Dirk Grunwald, Gary Nutt, Anthony Sloane, David Wagner, William Waite, and Benjamin Zorn. A testbed for improving the performance of parallel programs and systems. Technical Report CU-CS-512-91, University of Colorado, Department of Computer Science, CB 430, January 1991.

[6] Susan Eggers, David Keppel, Eric Koldinger, and Henry Levy. Techniques for efficient inline tracing on a shared-memory multiprocessor. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Boulder, CO, May 1990.

[7] P. G. Farber. Analysis of a shared bus multiprocessor memory system using trace driven simulation. Master's thesis, University of Colorado, 1991.

[8] A. Ferscha. Performance oriented parallel program design in the capse environment. In *Proceedings of the Minnowbrook Workshop on Software Engineering for Parallel Computing*, July 1991.

[9] Gary J. Nutt, Adam Beguelin, Isabelle Demeure, Stephen Elliott, Jeff McWhirter, and Bruce Sanders. Olympus: An interactive simulation system. In *1989 Winter Simulation Conference Proceedings*, pages 601–611, 1989.

[10] Heinrich J. Genrich. Predicate/transition nets. *Petri Nets: Control Models and Their Properties, Advances in Petri Nets 1986, Part 1*, 1986.

[11] K.Jensen. Coloured petri nets. *Petri Nets: Control Models and Their Properties, Advances in Petri Nets 1986, Part 1*, pages 248–299, 1986.

[12] James R. Larus. Abstract execution: A technique for efficiently tracing programs. Technical report, University of Wisconsin-Madison, Computer Science Department, February 1990.

[13] Michael K. Molloy. Performance analysis using stochastic petri nets. *IEEE Transactions on Computers*, C-31(9):913–917, September 1982.

[14] Gary J. Nutt. *The Formulation and Application of Evaluation Nets.* PhD thesis, University of Washington, 1972.

[15] Gary J. Nutt. A simulation system architecture for graph models. *Advances in Petri Nets 1990*, 1990.

[16] Gary J. Nutt. Collected papers on olympus. Technical Report CU-CS-518-91, University of Colorado, Department of Computer Science, CB 430, February 1991.

[17] Gary J. Nutt and Paul A. Ricci. Quinault: An office environment simulator. *IEEE Computer*, 14(5):41–57, MAY 1981.

[18] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[19] C. Ramchandani. *Analysis of Asynchronous Concurrent Systems by Timed Petri Nets.* PhD thesis, Massachusetts Institute of Technology, 1974.

[20] S. Sherman and J. C. Browne. Trace driven modeling: Review and overview. In *Proceedings of Symposium on Simulation of Computer Systems*, pages 201–207, 1973.

[21] Anthony Sloane. Spae user's guide. Technical report, University of Colorado, Department of Computer Science, July 1991.

[22] Alan Jay Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

A SOR as a C Threads Program

This appendix illustrates the methodology by describing the detailed steps in processing an example C threads program that performs successive overrelaxation on a 4 x 4 linear system of equations.

Because of the early stage of developement of SPAE, it is neccessary to translate C threads calls into Fortran-like **DOALL** statements. For this example, the subject program was written with conditional compilation directives to reflect the C threads and the **DOALL** equivalent:

```
/*   This program can be compiled as a C threads program
 *   with the degree of parallelism, N, determined by the data file
 *   using
 *      cc sor.c
 *
 *   Or it may be compiled as a DOALL C program, suitable for
 *   processing by SPAE by using
 *      cc -DSPAE
 *   WARNING:  ALL VALIDITY CHECKS HAVE BEEN REMOVED IN THIS VERSION
 */

#include   <stdio.h>

#ifdef SPAE
#define N   3
#include   "doall.h"
#else
#include   <cthreads.h>
#endif

#define   MAX_N   4

/* Local function prototype */
#ifndef SPAE
void slave(int);
#endif

/* Shared memory among the threads */
#ifdef SPAE
/* Fake cthreads mutex_t type */
typedef struct m_t
{
   int   value;
} mutex_t;
#endif

mutex_t lock[MAX_N];
float A[MAX_N][MAX_N], b[MAX_N], x[MAX_N];

#ifndef SPAE
int N;
#endif

main(argc, argv)
int argc;
```

```
char *argv[];
{
   FILE *data_fp, *fopen();
   int i;
   int tmp;
   double double_eps;
   float error, check();
   float epsilon;
   char data_filename[128];

#ifndef SPAE
   cthread_t t_handle[MAX_N];
   cthread_t cthread_fork();
#endif

/* Get data from file; first, get the command line parameters ... */
   double_eps = atof(argv[1]);
   epsilon = double_eps;
   strcpy(data_filename, argv[2]);

/* ... then read the data file */
   printf("Opening %s ...\n", data_filename);
   data_fp = fopen(data_filename, "r");
   fscanf(data_fp, "%d", &tmp);
#ifndef SPAE
   N = tmp;
#endif
   for (i=0;i<N;i++)
      fscanf(data_fp,"%f%f%f", &A[i][0], &A[i][1], &A[i][2]);
   fscanf(data_fp,"%f %f %f", &b[0], &b[1], &b[2]);
   fscanf(data_fp,"%f %f %f", &x[0], &x[1], &x[2]);
   fclose(data_fp);


/* Create the N slaves */
#ifndef SPAE
/* This is the code for dynamically defined N (cthreads) */
   for (i=0; i<N; i++)
   {
   /* Initialize lock */
      lock[i] = mutex_alloc();
      t_handle[i] = cthread_fork(slave, i);
   };
#else
/* This is the code for statically defined N (DOALL) */
   DOALL (i, N);
      slave(i);
   ENDDOALL(i, N);
#endif

/* Solve the system */
   error = 1000000.0;
```

```
   while(error > double_eps)
   {
   /* Proceed only if all slaves are locked */
      for (i=0; i<N; i++)
      {
         while(mutex_try_lock(lock[i]))
         { /* wasn't locked  -- restore it*/
            mutex_unlock(lock[i]);
#ifndef SPAE
            cthread_yield();
#endif
         };
      };
      error = check(A, x, b, N);
      for (i=0; i<N; i++)
         mutex_unlock(lock[i]);
#ifndef SPAE
      cthread_yield();
#endif
   };

   printf("Solution x[*] = %f %f %f\n", x[0], x[1], x[2]);
   printf("   With error factor = %f\n", error);
   exit(0);
}


float check(A, x, b, n)
float A[][4], x[], b[];
int n;
{
   int i, j;
   float sum;
   float error;

   error = 0.0;
   for(i=0; i<n; i++)
   {
      sum = 0.0;
      for (j=0; j<n; j++)
         sum = sum + A[i][j]*x[j];
      error = error + (sum - b[i]);
   };
   if(error<0.0) error = -error;
   return(error);
}


#ifdef SPAE
/* Fake cthreads lock manipulation routines */

mutex_lock()
```

```
{
}


mutex_try_lock()
{
}


mutex_unlock()
{
}
#endif


/*--------- The Slave Thread Schema -----------*/
#ifndef SPAE
void slave(me)
#else
slave(me)
#endif
int me;
{
   int j;

#ifndef SPAE
   cthread_yield();
#endif
   for(;;)
   {
      mutex_lock(lock[me]);
      x[me] = b[me];
      for (j=0;j<N; j++)
         if(me!=j) x[me] = x[me] -  A[me][j]*x[j];
      x[me] = x[me]/A[me][me];
   };
}
```

DOALL and **ENDDOALL** are macros used to manipulate SPAE:

```
/*
 * doall.h
 * Macros for annotating programs for DOALL tracing.
 */

#define FORK_DOALL      0x00001000
#define JOIN_DOALL      0x00002000
#define START_DOALL_ITER   0x00004000
#define FINISH_DOALL_ITER   0x00008000

#define DOALL(i,N) \
   ae_off (); \
   ae_special_event (FORK_DOALL, N, 0); \
```

```
   for (i = 0; i < N; i++) { \
      ae_context_switch (i); \
      ae_special_event (START_DOALL_ITER, i, 0); \
      ae_on ()

#define ENDDOALL(i,N) \
      ae_off (); \
      ae_special_event (FINISH_DOALL_ITER, i, 0); \
   } \
   ae_special_event (JOIN_DOALL, N, 0); \
   ae_context_switch (0); \
        ae_on ()
}
```

And the source code that is actually processed by the SPAE compiler is shown below:

```
#include   <stdio.h>

#define N   3
#include   "doall.h"

#define   MAX_N   4

/* Fake cthreads mutex_t type */
typedef struct m_t
{
   int   value;
} mutex_t;

mutex_t lock[MAX_N];
float A[MAX_N][MAX_N], b[MAX_N], x[MAX_N];

main(argc, argv)
int argc;
char *argv[];
{
   FILE *data_fp, *fopen();
   int i;
   int tmp;
   double double_eps;
   float error, check();
   float epsilon;
   char data_filename[128];

   double_eps = atof(argv[1]);
   epsilon = double_eps;
   strcpy(data_filename, argv[2]);

   printf("Opening %s ...\n", data_filename);
   data_fp = fopen(data_filename, "r");
   fscanf(data_fp, "%d", &tmp);
   for (i=0;i<N;i++)
      fscanf(data_fp,"%f%f%f", &A[i][0], &A[i][1], &A[i][2]);
```

```
   fscanf(data_fp,"%f %f %f", &b[0], &b[1], &b[2]);
   fscanf(data_fp,"%f %f %f", &x[0], &x[1], &x[2]);
   fclose(data_fp);

   DOALL (i, N);
      slave(i);
   ENDDOALL(i, N);

   error = 1000000.0;
   while(error > double_eps)
   {
      for (i=0; i<N; i++)
      {
         while(mutex_try_lock(lock[i]))
         {
            mutex_unlock(lock[i]);
         };
      };
      error = check(A, x, b, N);
      for (i=0; i<N; i++)
         mutex_unlock(lock[i]);
   };

   printf("Solution x[*] = %f %f %f\n", x[0], x[1], x[2]);
   printf("   With error factor = %f\n", error);
   exit(0);
}

float check(A, x, b, n)
float A[][4], x[], b[];
int n;
{
   int i, j;
   float sum;
   float error;

   error = 0.0;
   for(i=0; i<n; i++)
   {
      sum = 0.0;
      for (j=0; j<n; j++)
         sum = sum + A[i][j]*x[j];
      error = error + (sum - b[i]);
   };
   if(error<0.0) error = -error;
   return(error);
}

mutex_lock()
{
}
```

```
mutex_try_lock()
{
}

mutex_unlock()
{
}

slave(me)
int me;
{
   int j;

   for(;;)
   {
      mutex_lock(lock[me]);
      x[me] = b[me];
      for (j=0;j<N; j++)
         if(me!=j) x[me] = x[me] -  A[me][j]*x[j];
      x[me] = x[me]/A[me][me];
   };
}
```