MODELING MULTITHREADED APPLICATIONS USING PETRI NETS

KRISHNA M. KAVI
ALIREZA MOSHTAGHI

AND
DENG-JYI CHEN

Since most modern computing systems contain multiple processing elements, applications are relying on multithreaded programming techniques that allow a program to execute multiple tasks concurrently to take advance of the processing capabilities. Multithreaded programs are more difficult to design and test because of the non-deterministic execution orders and synchronization among the threads. Different approaches can be used to test Multithreaded Applications. In our approach we use Petri nets to represent the key elements of interactions among threads to identify potential problems such as race conditions, lost signals and deadlocks. A tool called *C2Petri* has been developed which converts C-Pthreads programs to the equivalent Petri net model. This tool helps verification of Pthread-based programs. At present the tool has limited capabilities and we hope to expand the capabilities of our tool in the near future.

Key Words. Pthreads, Petri nets, Race Conditions, Deadlocks, Lost Signals

MODELING MULTITHREADED APPLICATIONS USING PETRI NETS

1. INTRODUCTION

In this paper we present an approach and a tool that converts the complex behavior (particularly the synchronization behavior) of multithreaded programs written in C, using the Pthreads libraries to Petri nets for the purpose of analysis and testing of multithreaded programs. Our goal is to show that the complex behavior of the multithreaded applications may be expressed graphically as Petri nets, which can then be analyzed to locate faults in the program. Such an higher level abstraction facilitates for the identification of key program segments that must be thoroughly exercised to identify errors.

Concurrent systems are inherently nondeterministic. For example, given the same input, we can execute a concurrent program several times and get several different correct answers, or get the same answer in three different orders. The correctness of concurrent programs is defined by the Sequential Consistency model of Lamport [8] which states the behavior of a concurrent should be the same as if the program is run sequentially by interleaving the instructions from the concurrent components. However, since many correct orders are permitted by the definition, this makes concurrent systems hard to test. Concurrent software is vulnerable to a special set of faults resulting from: *Improper Synchronization*, *Deadlock*, *Starvation*, *Lost Signals*, and *Race Conditions*. All of these errors are in addition to the types of errors one would find in a sequential program. The words *error* and *fault* are used interchangeably in this paper. There are a number of strategies for testing concurrent applications. Each strategy considers a specified model for the Program Under Test (PUT). These models reflect certain characteristics of PUT, which can be analyzed to pinpoint faults in the program. The more comprehensive the model is, the more faults that can be detected. But fault detection is only one part of the problem; the model must provide the means of relating the faults to the source code. Our primary goal in this research is to make a model of PUT, such that it is abstract, correct, descriptive, and enable mapping back to the Program Under Test.

The *Pthreads library* can be used to implement multithreaded programs with C and C++ languages. Like all other concurrent programs, the C programs using Pthreads library suffer from synchronization faults caused by inappropriate programming practices. More recently tools are becoming available that can be used to detect synchronization errors (see for example, Visual Threads [12], LockLint [14], Flavors [4], ThreadMon [2]). These tools however, are limited in their scope and/or capabilities. They are either designed for a specific multithreaded library (e.g., Solaris Threads[14]); designed to detect only specific types of concurrency conditions (e.g., race conditions [12]); or designed primarily to monitor program's execution for the purpose of improving its performance [2]. In this paper, Petri nets are used to represent multithreaded C-programs. Our emphasis is on the detection of all synchronization errors. Although we demonstrate our research using only Pthreads in this paper, our approach of using Petri Nets is applicable for other multithreaded libraries and languages.

For the purpose of this paper we consider three major synchronization errors.

a**). Race Conditions**. In multithreaded programs concurrent threads interact with each other through shared memory. Improper access to shared data can cause data races. Data races are easy problems to introduce: simply accessing a variable without first acquiring an appropriate lock can cause a data race. Data races are generally very difficult to detect. Symptoms manifest only if two threads happen to access the improperly protected data at nearly the same time. Thus a program with data race may run correctly for months without showing any signs of the problem.

It is extremely difficult to exhaustively test all concurrent states of a program for even a simple multithreaded program; so conventional testing and debugging are not adequate defenses against data races. Data race is especially important when the critical section of the code consists of non-atomic operations.

b). **Deadlocks.** Every computer science student is familiar with deadlock situations from courses on operating systems[13]. A deadlock may occur when multiple processes share resources and the operations within the processes require access to more than one of those resources. This means that each process will need to get a lock for each of the resources before performing its operation. If different processes use a common set of resources, but the order in which they acquire the locks is inconsistent, there is a potential for a deadlock.

c). **Lost signals**. In multithreaded systems, threads can also interact by using signals to communicate the occurrence of certain events or conditions. Typically one or more threads wait on a condition; some other thread signals the waiting threads when the condition is met. In many multithreaded languages and libraries, signals are not saved. In other words, if a signal arrives before any thread executed a “wait” on that signal, the signal is lost, causing the waiting thread to be blocked indefinitely. Although this type of an error manifests immediately (in term of blocked threads), the cause (i.e., lost signal) is very difficult to locate. None of the existing tools address lost signals.

1.1 Brief overview of Petri nets.

Petri net is a formal and graphically appealing model, which is appropriate for modeling concurrency. Petri nets have been researched since the beginning of the 1960's, when Carl Adam Petri defined the model. The reader referred to Murata’s paper [9] for a comprehensive survey of Petri nets. A Petri net is a bipartite directed graph, together with an initial state called the initial marking, M_0. The two kinds of nodes are called places and transitions; where arcs are either from a place to a transition or from a transition to a place. In graphical representation, places are shown as circles and transition as bars. Arcs are labeled with weights (positive integers), where a *j*-weighted arc can be interpreted as the set of *j* parallel arcs. Labels for unit weight are usually omitted. A marking (state) assigns to each place a nonnegative integer. If a marking assigns to place *p* a nonnegative integer *k*, we say that *p* is marked with *k* tokens. Pictorially, we place *k* black dots (tokens) in place *p*. A marking is denoted by *M*, an *m*-vector, where *m* is the total number of places. The *p*th component of *M*, denoted by *M(p)* is the number of tokens in place *p* in marking *M*. A formal definition of Petri net is given in Figure 1.

Figure 1: Formal definition of Petri net

A Petri net is a 5-tuple, $PN = (P,T,F,W,M_0)$ where:

$P = \{p_1, p_1, \dots, p_m\}$	is	a finite set of places
$T = \{t_1, t_1, \dots, t_1\}$	is	a finite set of transitions
$F \subseteq (P \times T) \cup (T \times P)$	is	a set of arcs (flow relation)
$W: F \rightarrow \{1,2, \dots\}$	is	a weight function
$M_{0:}\ P \rightarrow \{1,2, \dots\}$	is	the initial marking
$P \cap T = \emptyset$ *and* $P \cup T \neq \emptyset$		

A Petri net structure $N = (P,T,F,W)$ without any specific initial marking is denoted by *N*.
A Petri net with the given initial marking is denoted by (N, M_0).

The behavior of many concurrent systems can be described in terms of system states and their changes. In order to simulate the dynamic behavior of a system, a state or marking in a Petri net is changed according to the following transition (firing) rules [9]:

1. A transition *t* is said to be enabled if each input place *p* of *t* is marked with at least *w(p,t)* tokens, where *w(p,t)* is the weight of the arc from *p* to *t*.
2. An enabled transition may or may not fire (depending on whether or not the event actually took place).
3. A firing of an enabled transition *t* removes *w(p,t)* tokens from each input place *p* of *t* and adds *w(t,p)* tokens to each output place *p* of t, where *w(t,p)* is the weight of the arc from *t* to *p*.

2. MODELING PTHREAD CONSTRUCTS

We limit our scope in this paper to model C programs using Pthread libraries. However, Petri nets can be used to model other multithreaded libraries and languages. In order to model the Pthread library, the components of this library are classified into two categories: the Pthreads control variables and the Pthread-functions.

2.1 Pthread Control Variables

The Pthread library has three essential control variable types: *mutexes*, *condition variables*, and *threads*. The Petri net models of these variable types are shown in Figure 2. These models are easier to understand when they are used in complete programs.

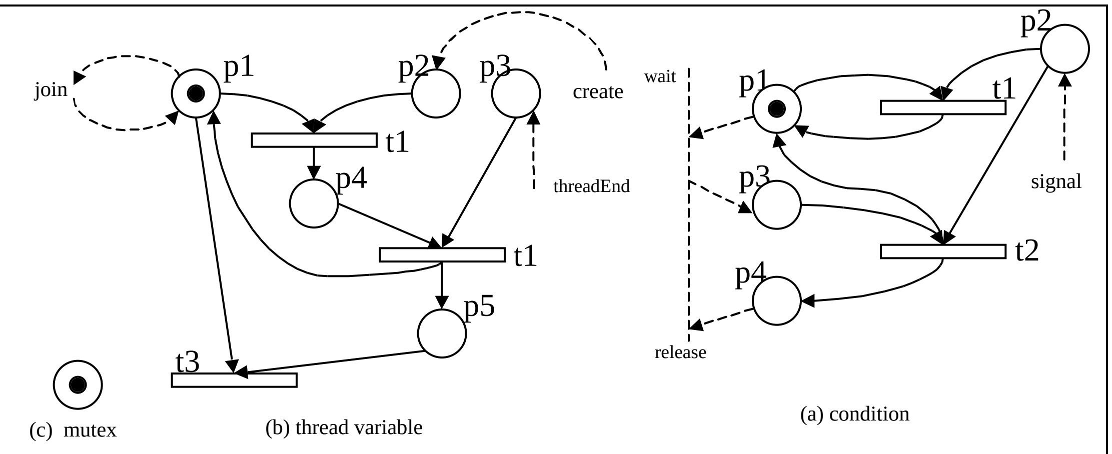

Figure 2: Petri net models of (a) *condition variables*, (b) *thread* and (c) *mutex*

2.1.1. Mutex variables are the simplest, they are declared with the abstract data type *Pthread_mutex_t*. A mutex is used to implement the mutual exclusion in a concurrent program. A place with an initial token can model a mutex variable; *Pthread_mutex_lock()* causes the token to be removed from the place and *Pthread_mutex_unlock()* places a token back in the place. When the token is unavailable to a thread executing mutex_lock, the thread will block. Although it is possible to model FIFO ordering on releasing blocked threads, we will assume a non-deterministic release of blocked threads, relying on the conflict presented by Petri net models.

2.1.2. Condition variables provide the means for signaling among threads. They are declared with the abstract data type *Pthread_cond_t* and are used by Pthread functions *Pthread_cond_signal()* and *Pthread_cond_wait()*. Condition variables are modeled with two transitions (*t1* and *t2*) and four places (*p1, p2*, *p3* and *p4*). The mechanism of this model is better understood when the behavior of *Pthread_cond_wait()* and *Pthread_cond_signal()* are explained.

2.1.3. A thread variable is declared using *Pthread_t*. The Petri Net model of a thread variable must hold the information about the thread's creation and termination. Five places and three transitions in Figure 2(b) (*p1, p2, p3, p4, p5, t1, t2* and *t3*) model the behavior of thread variables. Thread variables are used by Pthread-functions: *Pthread_create()* and *Pthread_join()* which are explained below.

It should be noted the number of places and transitions used to model the key Pthread control variables indicates the potential state space (or markings) complexity. The maximum number of states in our case is 2^p, where p is the number of places, although in most cases the number of states is much smaller because of only finite number of transitions that are enabled in any marking. In addition, since our tool permits analyzing program segments, the complexity can be more easily manageable.

2.2. **Pthreads-functions**

Due to space limitations, in this paper we will describe only the Petri net models of the most relevant Pthread-functions. These functions capture the essential synchronization features of the Pthread library, and understanding the behavior of these functions is essential to recognize the role of the other Pthread-functions. In Pthread it is possible to change the behavior of a Pthread function by changing attributes for the various thread control variables, including *Pthread_thread_t,* Pt*hread_mutex_t, and Pthread_cond_t.* We will delay the description of our approach to modeling attributes to a later section and present the Petri Net models of relevant Pthread functions with default attribute values. More specifically, in this section we will describe Petri net models for *Pthread_create()*, *Pthread_mutex_lock()*, *Pthread_mutex_unlock()*, *Pthread_cond_wait()*, *Pthread_cond_signal()*, *Pthread_exit()*, and *Pthread_join()*. *Pthread_cond_broadcast()* is not described here, since it can be modeled as an extension *Pthread_cond_signal()*. Obviously Pthread-functions can only be understood when they are used in concurrent programs. For example, the operation *Pthread_create()* is better understood when it is used along with a *Pthread_join()*. Our goal in this research is to detect synchronization-related errors. For example, in an attempt to join with a thread that is yet to be created (or exited), the will cause an error condition but the thread is permitted to continue beyond the Pthread_join call. Our model will detect such cases to facilitate better design of Pthread programs.

2.3. Thread Creation and Termination

```
Pthread_create(Pthread_t *thread, Pthread_attr_t *attr, void *
                (*start_routine)(void *), void *arg);
Pthread_join(Pthread_t thread, void **status);
```

Figure 3 illustrates the interconnection between the Petri net models for *Pthread_create()*, *Pthread_join()* functions, and that of the *thread variable* of the thread being created. The left dotted box represents the Petri net model of *Pthread_join()*, and the right dotted box represents the model of *Pthread_create()* functions. The graph in between the two dotted boxes is the representation of the Pthreads variable.

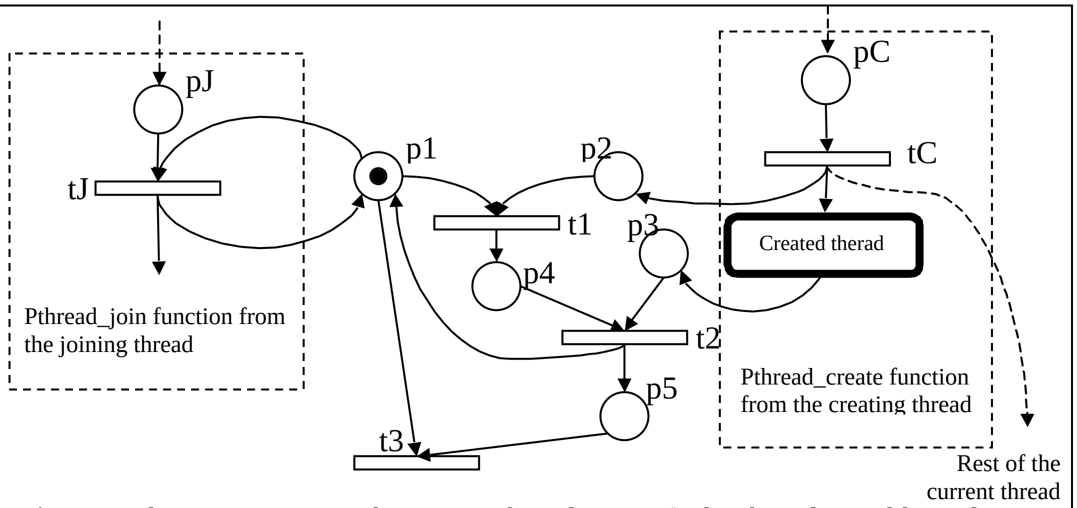
pJ
tJ
Pthread_join function from the joining thread
p1
p2
t1
p3
p4
t2
p5
t3
pC
tC
Created therad
Pthread_create function from the creating thread
Rest of the current thread

Figure 3 The interconnection between *Pthread_create()*, the *thread variable,* and *Pthread_join()*

As can be seen from Figure 3 *Pthread_create()* can be modeled using a single transition. In this model, the two outgoing solid arcs from transition *tC* illustrate the process of thread creation. The outgoing dotted arc from *tC* shows that the creating thread continues the execution. The complication of this model is primarily to handle the error behavior of *Pthread_join()* function. If a thread attempts to join with a thread that is not yet created, *Pthread_join()* will not block. It will receive an error code but allowed to continue its execution. Place *p1* models this behavior. If the thread being joined does not exist, there will be no token in place *p2*. This will disable transition *t1*, thus place *p1* will remain marked and transition *tJ* can consume this token and continue its execution. The token in p1 will be replenished to enable other joining threads. The behavior of *Pthread_join()* is quite different when the joined thread exists. In this case, *p2* is marked and *t1* will consume tokens from *p1*and *p2*, placing one token in *p4*, and disabling *tJ* from firing, thus blocking the joining thread.

Once the joined thread exits, a token will be placed in *p3*, enabling *t2*, which will place a token in *p1* and *p5*. Now the blocked thread can continue its execution. AS can be seen from this discussion the number of states needed to model Pthread_join is small. The semantics of the case when multiple threads attempt to join with a single thread differ from implementation to implementation. For example DEC-threads (the implementation of Pthreads library on DEC-Alpha) states that the semantics of such a situation leads to unpredictable behavior [3]. Our model provides a general framework for a variety of semantics associated with this situation. The unpredictability associated with DEC-threads is modeled by *t3*. As soon as *t2* fires, *t3* will be enabled and there will be a competition between *t3* and all of the *tJ* transitions of the joining threads. If *t3* wins the competition, the token in *p1* will be consumed disabling all joining threads, and no more joining threads will be able to continue. When modeling the case where only one thread joins with another, *t3* and *p5* must be eliminated.

2.4. Lock and Unlock Constructs

```
Pthread_mutex_lock(Pthread_mutex_t *mutex);
Pthread_mutex_unlock(Pthread_mutex_t *mutex);
```

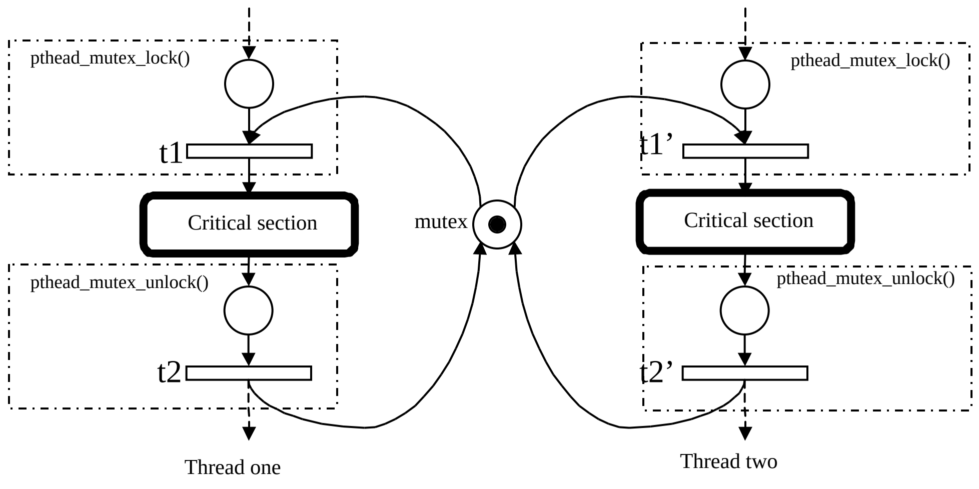

Figure 4 Petri net models *Pthread_mutex_lock()* and *Pthread_mutex_ulock()*

Pthread-functions *Pthread_mutex_lock()* and *Pthread_mutex_unlock()* provide the means for mutual exclusion in concurrent programs. Their only input argument is the mutex variable.

Figure 4 shows the interactions between these two functions using our Petri net model of a mutex variable. The conflict between *t1* and *t1'* to consume the token from *mutex* describes the mutual exclusion concept. The initial token in *mutex* allows only one of the threads to continue its execution. Once again, the number of states to represent the availability of a lack should not be an impediment to using Petri nets for modeling concurrent programs.

2.5. **Signal and Wait**

```
Pthread_cond_signal(Pthread_cond_t *cond);
Pthread_cond_wait(Pthread_cond_t *cond, Pthread_mutex_t *mutex);
```

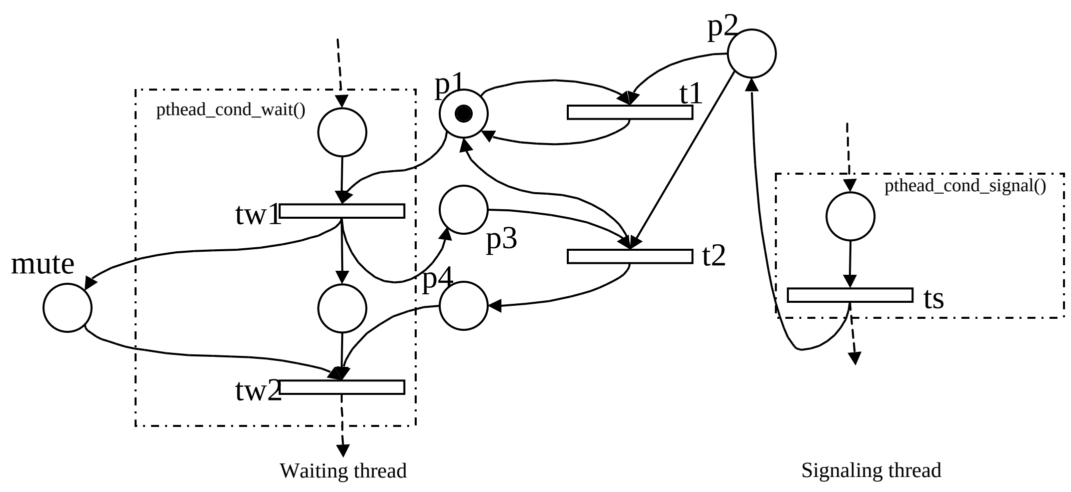

Figure 5: Petri net models *Pthread_cond_wait()* and *Pthread_cond_signal()*

Figure 5 illustrates how the mechanism of "signal" and "wait" in the Pthread library can be modeled using a Petri net. The left dotted-box shows the Petri net model of *Pthread_cond_wait()* and the right dotted-box that of *Pthread_cond_signal()*. As mentioned previously, if a signal is sent by a thread and no thread is waiting for that signal, that signal is lost. Thus a thread that executes a *Pthread_cond_wait()* right after the signal is lost, has to wait until the next signal arrives, and if not, will be deadlocked. In the model illustrated in Figure 5, the conflict between *t1* and *t2* for the token in *p2* distinguishes between the cases when the waiting thread executed *Pthread_cond_wait()* before the signal is sent or after the signal is lost. When the signaling thread executes *Pthread_cond_signal()*, *ts* places a token in *p2* and the execution of the thread continues. At this point, the conflict between *t1* and *t2* starts. If the waiting thread has already executed the *Pthread_cond_wait()* function, the token from *p1* will be consumed by *tw1* and a token will be placed in *p3*. Transition *t1* will lose the conflict with *t2* and *t2* will fire, placing a token in *p4*. This will prepare the situation for *tw2* to fire as soon as *mutex* is available. For the case when the waiting thread did not execute *Pthread_cond_wait()* when the signal arrived, *t1* will win the conflict with *t2* for the token in *p2.* When *t1* fires, the tokens from *p1 and p2* will be consumed. The loss of the token in *p2* models the loss of the signal. Although a bit more complex, the Petri net shown in Figure 5 represents manageable complexity in terms of the number of markings that are possible (less than 10).

2.6. **An Example**

To illustrate how these Petri net models can be combined together to represent a complete program, an example program and its Petri net model are provided in Figure 6. The C-Pthread program in this example implements a simple Producer/Consumer problem. The producer thread generates data items inside a loop and signals the consumer after each item is ready. The consumer waits for the signal from the Producer and consumes the item when the signal is received. This is a good example to illustrate the order of waiting for a signal and sending a signal. The model in Figure 6 is the output from our tool C2Petri. Each thread is enclosed in a box and the box is identified by a line number that corresponds to the line number in the C program containing the call to *Pthread_create()* function. This information is used to map errors back to source code. The source code line number is extracted by our tool so that the user can be provided with a means of locating the error in the source. Each of the Pthreads-control variables (mutexs, conditions, and threads) is also enclosed in separate boxes.

```
#include <Pthread.h>
int a;
int b;
Pthread_t th1;
Pthread_t th2;
Pthread_mutex_t mu;
Pthread_cond_t cond;

int main()
{
  Pthread_create(&th1,NULL,consumer,NULL);
  Pthread_create(&th2,NULL,producer,NULL);

  Pthread_join(th1,NULL);
  Pthread_join(th2,NULL);
}

int consumer()
{
  Pthread_mutex_lock(&mu);
  while(1)
    {
      Pthread_cond_wait(&cond,&mu);
      b=a;
    }
  Pthread_mutex_unlock(&mu);
}

int producer()
{
  while(moreDate())
    {
      Pthread_mutex_lock(&mu);
      a=getNewValueForA();
      Pthread_mutex_unlock(&mu);
      Pthread_cond_signal(&cond);
    }
}
```

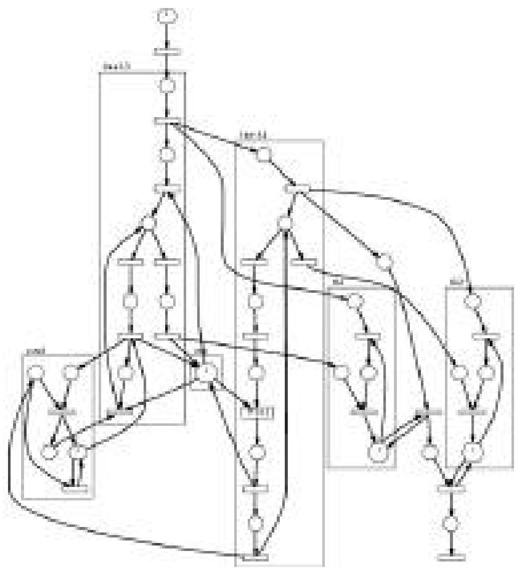

Figure 6: A producer/consumer program and the Petri net model

2.7. Other Pthreads Functions

So far we have illustrated how Petri nets can be used to model essential Pthread constructs with default attribute values. In this section will briefly outline how to model other attribute values. A complete description of Petri nets for each possible attribute value will consume too much space to be of value and the user is referred to [10] for details

2.7.1. Functions Related to Thread Creation

The following is a partial list of functions that can effect or be affected by attribute values associated with Pthread variables.

```
Pthread_create()
Pthread_attr_init()
Pthread_attr_setdetachstate()
Pthread_exit()
Pthread_join()
Pthread_detach()
```

The most important attribute of a thread is whether it is *detached* or *joinable*. The Petri net model shown in Figure 3 reflected the models for *Pthread_create()* and *Pthread_join()* using default attributes; in this case threads are joinable. If a thread attempts to join with a detached thread, however, an error condition is returned by *Pthread_join()* and the joining thread continues beyond the join. To model the behavior of a detached thread, the arcs from right dotted box to *p2* and *p3* in Figure 3 must be omitted.

So far we have not described how *Pthread_exit()* can be modeled. This function terminates the thread executing the exit function. The Petri net model of *Pthread_exit()* requires an arc from the point in the model where *Pthread_exit()* is called to the place *p3* in Figure 3. Figure 7 shows how a *Pthread_exit()* function can be modeled. Note that since the while loop in *func1()* has no contribution to the synchronization, its model is not shown completely. The figure also shows that the Petri net mode of Pthread_exit() is not complex.

```
#include <Pthread.h>
Pthread_t tid;

int main(){
  Pthread_create(&tid,NULL,func1,NULL);

  Pthread_join(tid);
}


void *func1(void *arg){
  if( arg == NULL)
    Pthread_exit(0);

  while(arg)
    {
      printf("this is the dumbest code I've ever wrote\n");
    }
}
```

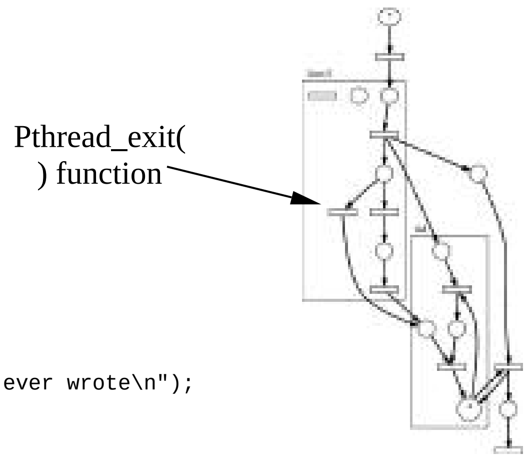

Figure 7: A program with *Pthread_exit()* function call

2.7.2. Functions Related to Mutexes

The following is a partial list of functions that can effect or be affected by attribute values associated with mutex variables.

```
Pthread_mutex_init()
Pthread_mutexattr_init()
Pthread_mutexattr_settype()
Pthread_mutex_lock()
   Pthread_mutex_unlock()
```

The Petri net models of *Pthread_mutex_lock()* and *Pthread_mutex_unlock()* described in previous sections (Figure 4) are based on default attributes of mutex variables. The behavior of *Pthread_mutex_lock()* and *Pthread_mutex_unlock()* for other attribute values (e.g., priority inheritance) depends on the implementation environment, requiring different models for each implementation. The Petri net models for representing priority inheritance is more complex and the complexity depends on the implementation. Since we are only interested in the abstract representation of abstract synchronization properties, the actual order of acquiring locks is not considered in this work.

3. C2PETRI TOOL DESIGN

The purpose of modeling a system is to capture and abstract as much useful information from that system as possible, in such a way as to permit analysis of the system. Modeling of a C program is no exception. For the purpose of our research not all aspects of a C program are captured in the Petri net model. We will only concentrate on the concurrency aspects of C programs using Pthreads. Here we use object oriented design techniques to describe how we identify key components of C programs using Pthreads. Once the components of a C program are recognized, its equivalent Petri net can be synthesized. Our approach is similar to that of [5].

3.1 Deriving the Petri net Model

We use a component-based approach to combine Petri net models for specific (and relevant) C program constructs. This approach simplifies the derivation of Petri nets and permits analysis of sub-systems more easily since the state space of the entire multithreaded program can be very large. To better understand our approach, consider the class diagram in Figure 8. A typical C-program is composed of *Functions*. A function has only one main *Block*.

A Block can be composed of many (sub-) blocks, and each (sub-) block may be an *IfElse* block, a *Loop* block or a *FunctionCall*. A switch statement can be modeled using several IfElse statements. These blocks are connected together to represent the control flow (or the order of execution) of the C program statements. Consider the program and associated blocks recognized by our system as shown in Figure 9. This example program has three functions `func1()` and `func2()` and `main()`.

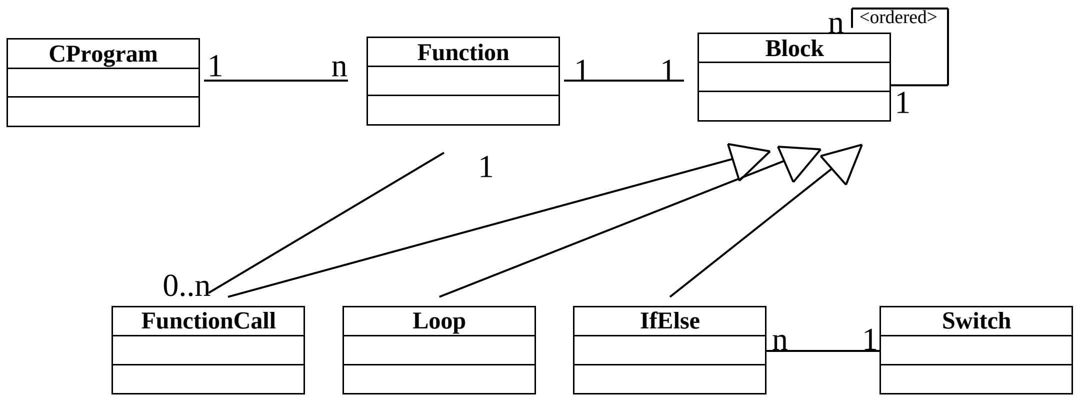

Figure 8: Class Diagram; a model for C programs

```
int main(void){
  int i,a;

  scanf("%d",&a);
  for(i=0;i<50;i++){
    if(i<a){
      func1();
    }
    else{
      func2();
    }

  }
}

int func1(void){
  return 1;
}

int func2(void){
  return 2;
}
```

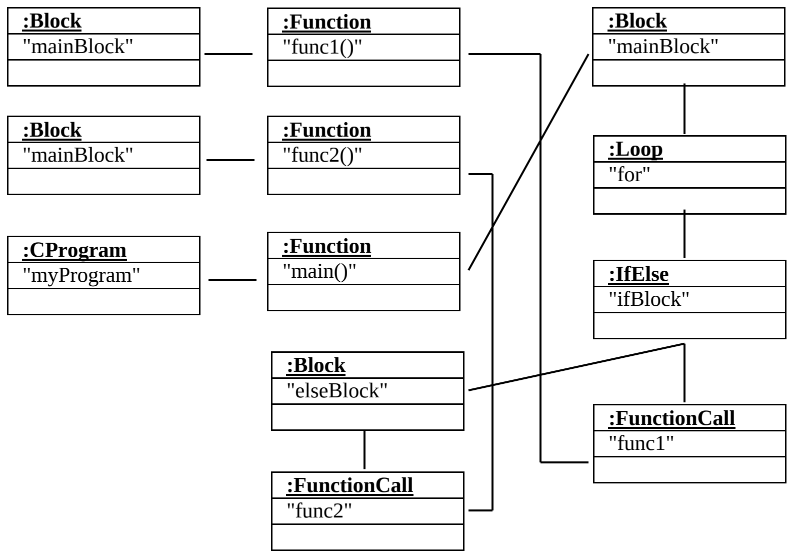

Figure 9: Object Diagram of the program on left; look at class diagram in Figure 8

3.2. Component Analysis

In order to generate a Petri net model of complete C programs, it is necessary to develop Petri net models for each relevant construct. In this paper recursive functions, pointers and arrays of thread types are not modeled. The Petri net models for these constructs require a dynamic creation of the Petri net, since the number of recursive calls or the address of the data item referenced by a pointer is known only at run time. It should be noted that since our tool attempts to model the dynamic interactions among threads, the theoretical correctness of the program can only be assured by the dynamic creation of Petri nets for representing recursion, pointers and arrays. However, it should also be pointed out that in most practical cases, one can observe the behavior by utilizing a limited number of recursive calls, array elements or pointer indirections to test multithreaded programs and gain confidence in the correctness of the program. Similarly, the interaction among a large number of threads (e.g., arrays of threads, a linked list of threads) can be tested by modeling the interactions among a small number (greater than 3) of threads. Our tool can be extended in manner similar to [1], to build Petri nets dynamically to model recursive calls and variable of number of threads.

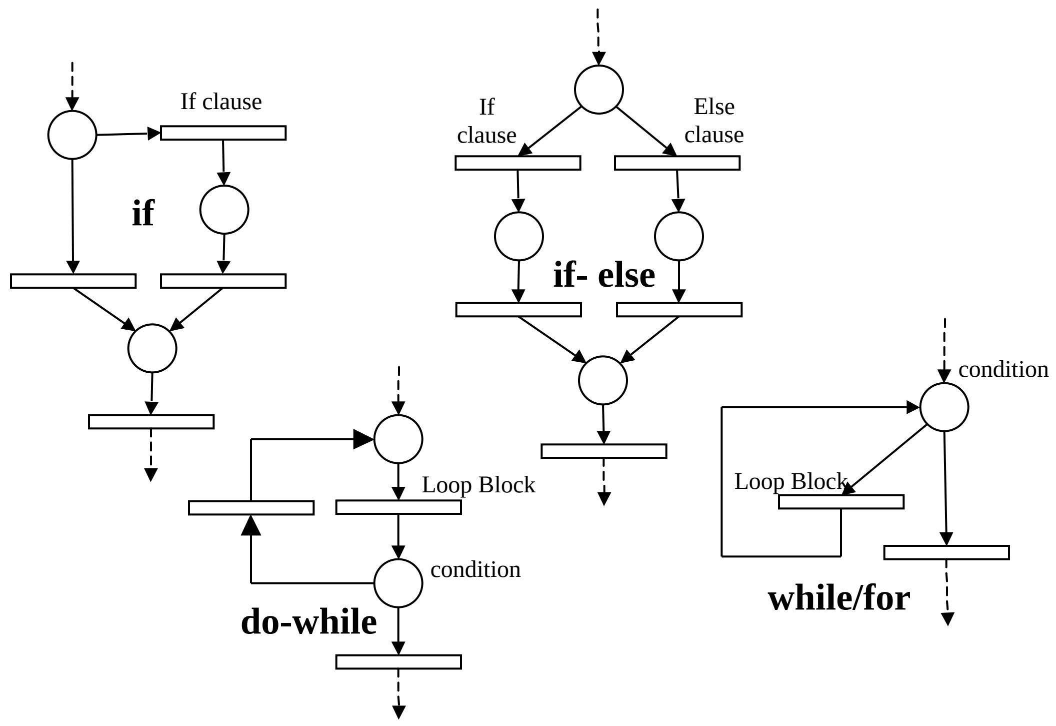

Figure 10: Petri net models of program constructs

In order to satisfy the bi-partite nature of Petri net, every component of a C program will be modeled starting with a place and terminating in a transition. The components can be composed by connecting the appropriate places and transitions. Figure 10 illustrates this idea.

3.3. Class Analysis

Figure 11 illustrates the Petri net model of the program shown in Figure 9.

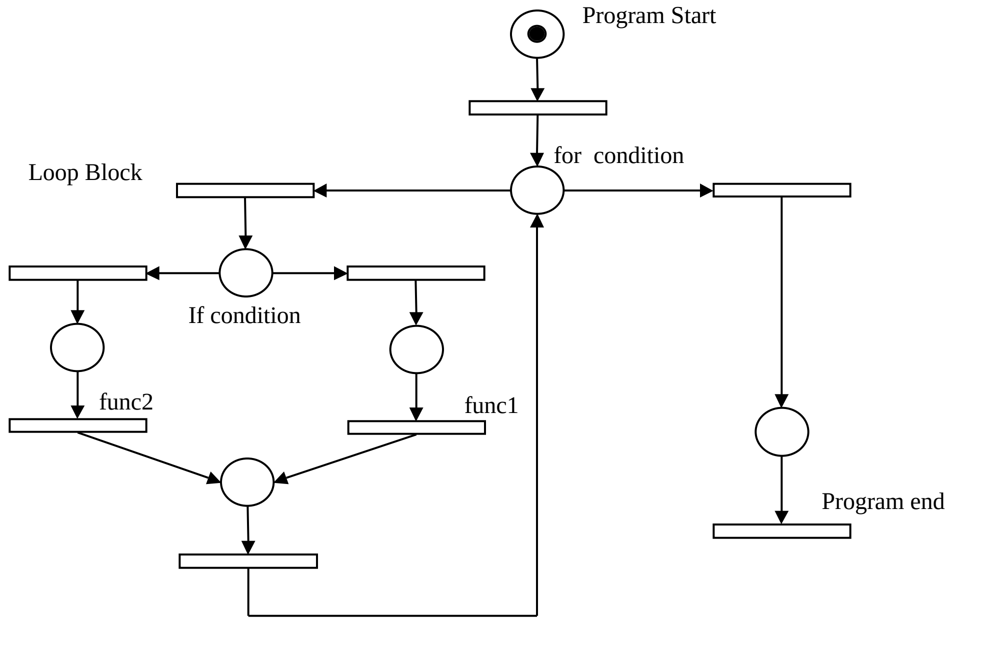

Figure 11: Petri net model of program in Figure 9

A token has been placed in the first place to represent the start of the program and enable the Petri net (that is, initial Marking). The last node in the Petri net is a sink transition to consume the token and terminate the execution. By comparing the Petri net model in Figure 11 and the object diagram in Figure 9, it is easy to recognize that the Petri net can be derived from the object diagram. For example, in the Petri net model, the loop embraces the *if-else* statement and in the object diagram, the *for* object has a link to the *if-else* statement object. Our tool C2Petri accepts a C program and generate the equivalent Petri net model, using the Petri net models for C program constructs described here (and in more detail in [10]). In our system, memory or access to program variables and assignment statements are not modeled. However we model control variables as described in Section 2 (e.g., mutex, condition variables, Pthread type), since they are essential to describe concurrency, synchronization and states of Pthreads.

3.4. Using C2Petri Tool.

We described the process of generating a Petri net model for a program. Now let us see how these Petri nets can be used to find defects in the program. The goal here is to find potential errors and display the path in the program that leads to the faulty situation. Assuming that we have the Petri net presentation of the program, and a marking that defines a faulty case, the set of markings leading to the faulty marking exhibits a path in the program that leads to potential synchronization fault in the program. This is often achieved by presenting the reachable markings of a Petri net as a graph called Markings graph G. There are a number of tools for analyzing Petri nets and generating all reachable markings. Such tools can be used to analyze Petri nets generated by our tool C2Petri. We can identify and trace program errors since we providing a mapping from faulty markings back to the original program using line numbers. The line number information for each block can be extracted during the compile time, and kept in the *block object* as one of its properties.

The Markings that signify lost signals are discuss previously. A marking *M* indicates a deadlock if and only if

- *M* is a reachable marking of *G* and
- *M* has no outgoing transitions.

A marking-graph *G* is said to be deadlocking if and only if *G* has one or more deadlock states. The Pthread program in Figure 12 is a deadlocking program. Its equivalent Petri net and

```
#include <pthread.h>

pthread_t tid1;
pthread_t tid2;
pthread_mutex_t m,n;

int main(){
  pthread_create(&tid1,NULL,func1,NULL);
  pthread_create(&tid2,NULL,func2,NULL);
}

void *func1(void *arg){
  pthread_mutex_lock(&m);
  pthread_mutex_lock(&n);
  /*critical section*/
  pthread_mutex_unlock(&m);
  pthread_mutex_unlock(&n);
}

void *func2(void *arg){
  if(arg!=NULL){
    pthread_mutex_lock(&n);
    pthread_mutex_lock(&m);
    /*critical section*/
    pthread_mutex_unlock(&n);
    pthread_mutex_unlock(&m);
  }
}
```

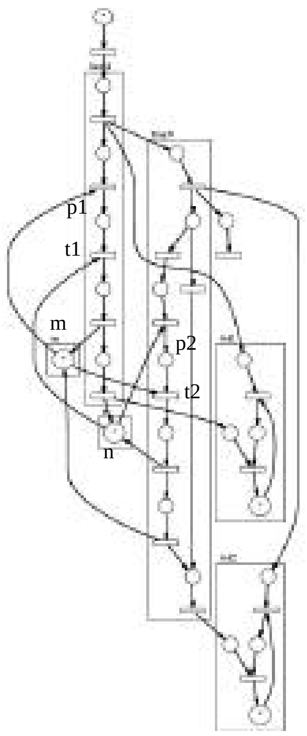

Figure 12: A deadlock program and its Petri-Nets model

the associated marking-graph are also shown in the same figure. Any marking with tokens in both *p1* and *p2* represents a deadlock. In such a case, the tokens in places *m* and *n* have been consumed previously, thus transitions *t1* and *t2* cannot fire because one of their input places does not have a token. This situation is very common when locking and unlocking of the mutexes are inconsistent among the threads. In this example, every path leads to a deadlock, but there are cases when only a few paths lead to deadlock. Note that since a program must terminate at some point, the terminating markings appear similar to a deadlock marking. However, once the Petri net is generated, we can easily distinguish the terminating markings from deadlock markings.

4.SUMMARY AND CONCLUSIONS

In this paper we presented how Petri nets can be used to model Pthread programs and how the Petri net can be used to detect synchronization errors. Our research is motivated by the lack a generic programming model and environment for developing multithreaded applications. Although we have concentrated on Pthread based applications, our approach can be used for other multithreaded libraries and languages. Since Petri nets have been studied extensively over the past 3 decades, and since there a number of analysis tools, multithreaded program development can benefit from using such tools for the purpose of analysis and testing.

We are in the process of extending the scope of our work to facilitate automatic translation of concurrent systems specified using formal models based on process algebras into Petri nets. Previously we developed one such translation from CSP to Petri nets [7]. The resulting Petri nets can then be analyzed and decomposed into subsystems. These subsystems can be translated into implementation in any multithreaded language. The work described in this paper aids in this last phase in generating program stubs for Pthread libaries.

Finally, although the major problem with Petri net based models is the size of the state space, for our purpose this need not be a major hindrance because, a) portions of programs can be analyzed and these analyses can be utilized in the overall system analysis, b) the interactions among a large number of threads can be extrapolated from the analysis of a small number of threads, and c) the dynamic program behavior such as recursion and pointers can be abstracted by analyzing a small number of recursions and pointer indirections.

5. REFERENCES.

[1] J. W. Anneck and M. Naedele: "Modeling Hierarchical and Recursive Structures Using Parametric Petri net" *Proceedings of High Performance Computing HPC'99*, San Diego, CA, U.S.A., 1999, pp. 445-452

[2] B. M. Cantrill and Thomas W. Doeppner Jr.: "Thread Mon, A Tool for Monitoring Multithreaded Program Performance", Department of Computer Science, Brown University Providence, RI 02912-1910.

[3] Dec-Alpha: *Pthreads Man Pages*, Dec-Alpha system users manual

[4] M. B. Dwyer and L. A. Clarke: “FLow Analysis for VERifying Specifications of Concurrent and Distributed Software”, _Technical Report 1999-52,_ University of Massachusetts, Amherst, Aug. 1999. ftp://ftp.cs.umass.edu/pub/techrept/techreport/1999/UM-CS-1999-052.ps.

[5] M. Heiner: “Petri net Based Software Validation Prospects and Limitations”, The International Computer Science Institute, University of California at Berkeley, _TR-92-022_, March 1992

[6] K. M. Kavi, B. P. Bukhles, and U. N. Bhat: “Isomorphism Between Petri net and Dataflow Graphs”, _IEEE Transactions on Software Engineering_, Vol. SE-13 No. 10, Nov. 1987.

[7] K.M. Kavi, F.T. Sheldon and S. Reed. "Specification and analysis of real-time systems using CSP and Petri nets", _International Journal of Software Engineering and Knowledge Engineering,_ (World Scientific Publishing Company) Vol. 6, No. 2, June 1996, pp 229-248.

[8] L. Lamport. “How to make multiprocessor computer that correctly executes multiprocess programs”, IEEE Tr. On Computers, September 1979, pp 241-248.

[9] T. Murata: “Petri Nets: Properties, Analysis and Applications”, _Proceedings of the IEEE,_ Vol.77, No.4, 1989, pp. 541-580

[10] A.R. Moshtaghi. “Modeling Multithreaded Programs Using Petri Nets”, MS Thesis, Dept of ECE, The University of Alabama in Huntsville, Huntsville, AL 35899, May 2001.

[11] J.P. Queille and J. Sifakis: *Iterative Methods for the Analysis of Petri Nets*, 1st European workshop on Applications and Theory of Petri Nets, Springer Verlag, Sept. 1982.

[12] S. Savage, Michael Burrows, Greg Nelson, and Patrick Sobalvarro: *Eracer: A Dynamic Data Race Detector for Multithreaded Programs*, ACM Trance on Comp Sys vol. 15 no. 4 Nov. 1997.

[13] A. Silberschatz, P. B. Galvin, and G. Gagne: *Applied Operating System Concepts, First Edition*, John Wiley & Sons, Inc, 2000

[14] SunSoft: *Lock Lint User's Guide*, SunSoft Manual, August 1994.

[15] A.Y. Zerhouni, E. Moudni, and A. Ferney: “On Finding Deadlocks and Traps in Petri net” _System Analysis -Modelling - Simulation (SAMS)_ 1999, pp. 495-507.