LOGICAL DEVELOPMENT OF A PETRI NET
DEADLOCK ANALYSIS PROGRAM

Daniel E. Cooke
Computer Science Department
University of Texas El Paso

ABSTRACT

If a well-structured concurrent program is represented as a Petri net, it is possible to analyze the network for deadlock. This paper defines program liveness for a well-structured concurrent program represented as a Petri net, presents a Prolog program which will analyze the Petri net for deadlock, and proves that the program truly implements the specification of program liveness. A well-structured concurrent program is one in which levels of nesting follow structured programming tenets.

1.0 INTRODUCTION

The significance of this paper is in its introduction of a Prolog program capable of detecting proneness to deadlock in Petri nets. The program is capable of discovering deadlock proneness in a class of Petri nets which is defined intuitively. Based upon this class of Petri nets, a specification of "program liveness" is given and the Prolog program is proven to be correct w.r.t. the specification through the use of program proof techniques from the area of logic programming. We present:

(1) basic definitions of the Petri net,
(2) an intuitive definition of the class of Petri nets of interest,
(3) formal specifications of program liveness, and
(4) a derivation of a Prolog program to inspect an arbitrary network for program liveness.

2.0 BASIC DEFINITIONS

This paper assumes some prior knowledge of Petri nets, predicate calculus, and Prolog. A reader unfamiliar with Petri nets is referred to [2, 5, 6]. A reader unfamiliar with concurrency and deadlock detection is referred to [1,3]. What follows is a basic set of definitions for Petri nets.

A Petri net is a five-tuple: $N=(P, T, F, H, M_0)$ where:

- $P$: is a finite set of places;
- $T$: is a finite set of transitions;
- $F: P \times T \rightarrow P$;
- $H: T \times P \rightarrow P$; and
- $M_0$: a multiset of places which forms the initial state of the network.

We introduce the vectors of sets $SF_t$ and $SH_t$ according to the following:

$SF_t = \{ p \mid F(p, t) = p \}$ and

$SH_t = \{ p \mid H(t, p) = p \}$ where $t \in T$ and $p \in P$.

Note that $M$ is a vector of multisets, $M_0$, $M_1$, ..., $M_m$. The vector $M$ provides an execution history of the associated Petri net. A given state in the execution history, $M_i$, gives the name of each place containing a token in the $i$th execution state. If a place contains $n$ tokens at state $i$ its name will appear $n$ times in $M_i$. Therefore, when $n=0$, the place's name does not appear in $M_i$.

This paper provides a definition of program liveness, presents a Prolog procedure set which evaluates a Petri net for program liveness, and verifies that the program fulfills the definition of program liveness.

3.0 THE CLASS OF PETRI NETS

The class of Petri nets considered are safe and conservative (but not strictly conservative). [5,6] These Petri nets are intended to model concurrent programs. An assumption underlying the results discussed here is that the programs to be inspected are well-structured. This requires a special relationship between the instructions which allocate and deallocate some resource $R$ and nearby...
loops. In particular, the instructions allocating and deallocating a resource \( R \) must be contained completely inside a loop and/or completely surround a loop. If a program is well-structured, there is a symmetrical relationship between allocations and deallocations.

The final requirement for a program to be well-structured is that for every fork in execution there is a matching join. These constructs in the concurrent programming environment are realized by a set of tasks encapsulated in a procedure. When the procedure is invoked, there is a fork in execution and each encapsulated task begins execution. The join occurs when all tasks complete execution and a return from the encapsulating procedure occurs.

If a program is well-structured, it is possible to extract control flow loops from its associated Petri net model for the purpose of inspection for a proneness for deadlock. Therefore, the program to be developed in this paper will inspect parallel structures which do not contain control flow loops. It is possible to distinguish control flow loops from the loops which allocate/deallocate some resource. In our modeling, a control flow loop involves a connecting transition (i.e., the transition has the loop’s beginning place in its outdegree and the loop’s ending place in its indegree), and the resource allocation/deallocation loop involves a connecting place (i.e., the resource place has the loop’s beginning transition in its outdegree and the loop’s ending transition in its indegree).

Prior to analysis, control flow loops are extracted from the Petri net model. For static allocation/deallocation of resources or static message passing, this restriction does not present a problem.

4.0 CONCURRENT PROGRAM LIVENESS

In order to evaluate a Petri net for program liveness it will be necessary to have definitions for the execution of a network. The execution primitive of a Petri net involves firing some transition which is eligible for firing. Therefore, two definitions are required: one which defines a set, \( E \), of transitions eligible for firing at a given marking, \( M_i \):

\[
E = \{ t \mid M_i \supseteq SF_t \}
\]  

(1)

and one which fires a selected transition from the set of eligible transitions rendering marking \( M_{i+1} \) from \( M_i \):

\[
(\exists t \in E \rightarrow M_{i+1} = M_i - SF_t \bowtie SH_t)
\]  

(2)

where \( \bowtie \) is a concatenation operator capable of providing multiple occurrences of a place in the multiset, \( M_i \), and \( - \) is set difference defined on a multiset to remove a single occurrence of an element from \( M_i \) which also occurs in SF

We restate formulae 1 and 2 as formulae 3 and 4, respectively. The following formulae will, (1), provide specifications for defining the set \( E \) given a marking \( M \) and an initially null \( E \), and (2), provide for firing a transition \( t \) in a marking \( M_i \), to produce a successor marking \( M_{i+1} \):

\[
el(M,E) \iff (\forall t \in T \land M \supseteq SF_t \rightarrow E := E \cup \{t\})
\]  

(3)

\[
f(M_i,t,M_{i+1}) \iff bw(M_i,t,M_{i+1}) \land fw(M_i,t,M_{i+1})
\]  

(4a)

\[
bw(M_i,t,M_{i+1}) \iff \text{difference}(M_i,SF_t,M_{i+1})
\]  

(4b)

\[w(M_i,t,M_{i+1}) \iff \text{append}(M_i,SH_t,M_{i+1})
\]  

(4c)

Note that \( bw \) and \( fw \) are the backward and forward incidence functions, respectively. The references to \( \text{difference} \) and \( \text{append} \) are references to the standard definition of set difference and the standard Prolog \( \text{append} \) which is equivalent to the concatenation operator previously defined. For the sake of modularity, a specification that will fire a given transition in some marking \( M_i \) and determine the resulting set \( E \) follows:

\[
fe(M_i,t,M_{i+1},E) \iff f(M_i,t,M_{i+1}) \land el(M_{i+1},E)
\]  

(5)

The Petri net is used to model a machine. Therefore, the object of the Petri net is to continue firing transitions forever or until there are no remaining enabled transitions. Consider the

\[
E = \{ t \mid t \in T \land M_0 \supseteq SF_t \}, i=0;
\]

while \( E \neq \emptyset \)

(\( \emptyset \) stands for the null set) do

select some \( t \in E \);

\[
M_{i+1} = M_i - SF_t \bowtie SH_t;
\]

\[
E = \{ t \mid t \in T \land M_{i+1} \supseteq SF_t \};
\]

\( i:=i+1; \)

Note that in a well-structured concurrent program, we can map the control structure to a corresponding well-structured Petri net. The parallel structures of the resulting Petri nets are
guaranteed to have an initial fork tf and a matching final join tj. The subnet inside the outermost fork-join pair in a concurrent network is all we have to inspect for deadlock since deadlock requires parallel paths of execution. We designate $M_0$ to be $SF(tf)$ and $M_f$ to be $SH(tj)$. A focus on the subnet beginning with $SF(tf)$ and $SH(tj)$ allows for an operational definition of deadlock. If there exists a sequence of markings beginning at $M_0$ and ending at marking $M_l$ where $E=\emptyset$ and $M_l\neq M_f$, then we can conclude that deadlock occurred in the marking sequence of the parallel structure and the structure is deadlock-prone.

These observations provide us with the knowledge to intuitively understand program live-ness. If a program has a subset of orderings which deadlock (i.e., in which $E$ is null and we have not achieved a desired final marking), then the program is deadlock-prone. These are mutually exclusive characteristics. A program which meets the program liveness property does not have any execution orderings in which deadlock occurs.

Reachability in a Petri net assumes an existing marking and a desired marking. If $M_f$ is reachable from $M_l$, then there exists a firing sequence taking us from $M_l$ to the desired marking $M_f$. For a program, program liveness will require that some final marking be reachable from any possible intermediate marking $M_i$, at which time $E$ should be null. Thus, we must inspect all possible firing sequences in order to determine program liveness for a given Petri net. Given these definitions, we now present a formal specification of what we call program liveness, $P_l$. In this definition, $M$ is an initial/intermediate marking, $F$ is a desirable final marking, and $E$ contains the transitions enabled due to marking $M$.

$$\begin{align*}
S1: \quad & pl(M,E,F) \iff \\
& (E=\emptyset \rightarrow M \not\subseteq F) \lor (\forall t (t \in E \rightarrow fo(M,t,F)))
\end{align*}$$

$S2$: $fo(M,t,F)$ iff

$$\begin{align*}
(\exists M')(\exists E')(fe(M,t,M',E') \land pl(M',E',F))
\end{align*}$$

$S3$: $t \in E$ iff

$$\begin{align*}
(\exists t' (E'=\{t' \mid E\}' \rightarrow (t=t' \lor t \in E)))
\end{align*}$$

Please note the use of the standard Prolog list construct, [ ]. A reference to [ ] is a reference to an empty list, a reference to [X,Y] allows access to the first element of a list through $X$ and access to the remainder of a list through $Y$. Further note that $S2^2$ references previously defined formula $fe(M,t,M,E)$, i.e., formula (5).

In what follows, our goal is to show that the following Prolog procedure set indeed implements the specification $S1$:

$$\begin{align*}
p(M,[[],F]):=&\text{subset}(F,M),
p(M,[T][I],F):=fo(M,T,F),
p(M,[T][E],F):=fo(M,T,F),pl(M,E,F).
\end{align*}$$

Since we are attempting to verify a program to inspect for program liveness, we will begin with the definition of $pl$ as a starting point. We will employ a method called definients transforms defined in [4] and augmented by rules of inference from symbolic logic. Each step in the process of derivation will be labeled $d$ followed by a natural number.

$$\begin{align*}
d1: \quad & (E\emptyset \rightarrow M \not\subseteq F) \lor (\forall t (t \in E \rightarrow fo(M,t,F)))
\end{align*}$$

In definients transforms, given a general definition $LHS \iff RHS$, we may replace a reference to the RHS by the corresponding LHS or a reference to a LHS by the corresponding RHS. In our first derivation step, we will replace the boldfaced reference $t \in E$ with its corresponding RHS from $S3$.

$$\begin{align*}
d2: \quad & (E\emptyset \rightarrow M \not\subseteq F) \lor (\forall t (X((\exists t' \exists E') (E=\{t' \mid E\}' \rightarrow (t=t' \lor t \in E))) \rightarrow fo(M,t,F)))
\end{align*}$$

Next, we will distribute $E=\{t' \mid E\}'$ using the rule $A \rightarrow B \lor C \Rightarrow A \rightarrow B \lor A \rightarrow C$:

$$\begin{align*}
d3: \quad & (E\emptyset \rightarrow M \not\subseteq F) \lor (\forall X((\exists E') (E=\{t' \mid E\}' \rightarrow t=t' \lor t \in E')) \rightarrow fo(M,t,F)))
\end{align*}$$

Next, we will distribute $fo(M,t,F)$ using the rule $A \land B \rightarrow C \Rightarrow A \rightarrow C \land B \rightarrow C$:

$$\begin{align*}
d4: \quad & (E\emptyset \rightarrow M \not\subseteq F) \lor (\forall X((\exists E') (E=\{t' \mid E\}' \rightarrow t=t' \lor t \in E')) \rightarrow fo(M,t,F)))
\end{align*}$$

We now simplify $(E=\{t' \mid E\}' \rightarrow t=t') \rightarrow fo(M,t,F))$. As a result of this step we can move $(\forall t)$ further inward:

$$\begin{align*}
d5: \quad & (E\emptyset \rightarrow M \not\subseteq F) \lor (\forall X((\exists t' \exists E') (E=\{t' \mid E\}' \rightarrow t \in E \rightarrow fo(M,t,F)))
\end{align*}$$

Note that $E'$ may be empty or not empty. Therefore, we can add the condition to check for $E'$ being empty prior to $(\forall t)$.
Next we apply \( A \rightarrow B \land C = A \rightarrow B \land A \rightarrow C \):

\[
(d_7): (E \rightarrow M \supset F) \lor (\exists t' \exists E \forall E[t' \mid E] \rightarrow \text{fo}(M,t',F) \land (\forall t \forall X E[t' \mid E] \rightarrow \text{fo}(M,t',F)))
\]

We know as a result of the previous steps, that in the final consequent, \( E' \) is not empty. Therefore we may apply definiens transform to the boldface subexpression to achieve \( \text{pl}(M,E',F) \):

\[
(d_8): (E \rightarrow M \supset F) \lor (\exists t' \exists E \forall E[t' \mid E] \rightarrow \text{fo}(M,t',F) \land (\forall t \forall X E[t' \mid E] \rightarrow \text{fo}(M,t',F) \land \text{pl}(M,E',F)))
\]

Dropping the existential quantifiers give us:

\[
(d_9): E \rightarrow M \supset F \lor E[t' \mid E] \rightarrow \text{fo}(M,t',F) \lor E[t' \mid E] \rightarrow \text{fo}(M,t',F) \land \text{pl}(M,E,F)
\]

After a few minor adjustments to convert to a Prolog-like syntax we achieve the following equivalent formula, wherein the antecedents of each subformula of \( d_9 \) are handled in the procedure head; the logical AND is indicated by the comma; and the disjunctions are provided through the three differing Prolog procedures:

\[
\begin{align*}
\text{pl}(M, [I,F]) & : \text{subset}(F,M). \\
\text{pl}(M, [T,E], F) & : \text{fo}(M, T, F) \land \text{pl}(M, E, F).
\end{align*}
\]

We will now complete the Prolog program by developing procedures for the remaining formulae in the specification. First we define:

\[
\text{fo}(M, T, M,N,E), \text{pl}(M,N,E,F).
\]

which corresponds to specification \( S2 \). The formula \( 5 \) appears as follows:

\[
\text{el}(M, [T,M,N,E]) -
\]

An attempt to derive \( \text{el} \) from formula (3) is left for the reader:

\[
\text{el}(M,[I,E,E]). \\
\text{el}(M,[H,R,I,E,N,W,E]) - t(H,S,F,SH) \lor \text{subset}(SF,M), \\
\text{append}(H,E,T,E) \lor \text{el}(M,R,T,E,N,W,E). \\
\]

The formulae (4.a-4.c) result in the following procedure sets:

\[
\text{fo}(M, OLD, T, M,N,E) - \\
\text{t}(T, S, F, SH), \text{backward}(M, OLD, SH, M, OLD T), \\
\text{forward}(M, OLD T, S, F, M, N,E).
\]

\[\text{backward}(M, OLD, SH, M, N,W,E) -
\text{append}(SH, M, OLD, SH, M, N,W,E).
\]

\[\text{forward}(M, OLD, SF, M, N,W,E) -
\text{diff}(M, OLD, SF, M, N,W,E).
\]

### 5.0 SUMMARY AND CONCLUSIONS

In this paper we developed a Petri net specification for program liveness. The specification assumes a well-structured concurrent program is represented from which loops can be extracted. The program was formally verified so that guarantees concerning its correctness w.r.t. the specification are maintained. Once produced, example executions of the program were presented. The significance of this work is that it introduces an important Prolog program for design analysis and it demonstrates a method for program development which leads to correct programs.

### ACKNOWLEDGEMENT

Research sponsored by the Air Force Office of Scientific Research (AFSC), under contract F49620-89-C-0074, and by NSF Grant No. CDA-9015006. The United States Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon.

### REFERENCES


