Simulation of Petri Nets in Prolog: Modeling Dynamic System Behavior

By Girish Keshav Palshikar

Introduction

A Petri net is a popular visual notation, invented by Carl Adam Petri in his 1961 Ph.D. thesis, for modeling dynamic behavior of systems. Its rigorous mathematical basis enables systems modeled in Petri nets to have a precise and unambiguous meaning that can be executed (simulated) and tested. Today there are even techniques for analyzing properties (e.g., safety, liveness etc.) of Petri net system models.

Petri net models are particularly useful for systems whose behavior is characterized as dynamic, concurrent, asynchronous, distributed or stochastic; for example communication protocols, intelligent and flexible manufacturing systems, industrial control systems, office automation systems, business information systems, real-time embedded systems, hardware systems (e.g., multiprocessor memory subsystems) etc. The GrafCet international standard for process control systems is based on Petri nets.

Within software engineering, they express system architectures to analyze safety, performance etc. Petri nets are also a novel way of representing dynamic knowledge or internal behavior of many complex AI systems like software agents, Internet applications and robots12.

There are many tools available for drawing, simulation and analysis of Petri nets (www.daimi.aau.dk/PetriNets/tools and www.daimi.aau.dk/PetriNets/bibl) as well as classic references3. There are also regular conferences on their theory and applications.

In this article, we illustrate how symbolic logic programming can neatly represent and simulate Petri nets. We also demonstrate how to use Prolog for various types of analysis for one instance of a Petri net. Finally, we point out how to perform various transformations. The primary motivation for building a Petri net toolbox is that Prolog, being a high-level symbol manipulation language, is eminently suitable to succinctly and cleanly represent, analyze and simulate Petri nets. In addition, Prolog allows models of dynamic system behavior or domain knowledge-bases to be integrated with other components of knowledge-based systems e.g., rule-bases. See (Azema et al, 1984) for an early effort to use Prolog for Petri nets.

Dynamic Behaviour of Petri Nets

The following transition firing rule, which changes the marking (state) of a Petri net (see Syntax of Petri Nets sidebar), simulates the dynamic behavior of the system that the net represents:

- A transition t is 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.
- An enabled transition may or may not fire.
- A firing of an enabled transition t removes w(p,t) tokens from each input place p of t and adds w(t,q) tokens to each output place q of t, where w(t,q) is the weight of the arc from t to q.

Firing of an enabled transition changes the token distribution (marking) in a Petri net. A firing sequence is a sequence of transitions s = M 0, t1, M 1, t2, ... where each ti is a transition enabled at marking Mi-1 and firing of ti results in marking Mi; M 0 is the initial marking. A firing sequence is sometimes also written as t1, t2, ...

A transition without any input place is a source transition and one without any output place is a sink transition. A source transition is always enabled. Firing of a sink transition consumes tokens but does not produce any. A pair of a place p and transition t is called a self-loop if p is both an input and output place of t. A Petri net is pure if it does not contain any self-loops and it is ordinary if the weight for all its arcs is 1.

A Petri net is a state machine if each transition in the net has exactly one incoming arc and exactly one outgoing arc. It is a marked graph if each place in the net has exactly one incoming arc and exactly one outgoing arc. A place having two or more output transitions is called a conflict or a decision or a choice. Two transitions are concurrent if one transition may fire before, after, or in parallel with the other transition. A state machine can contain conflicts but not concurrent transitions; a marked graph can contain concurrent transitions but not conflicts.

Example 1: Consider an extremely simple client that repeatedly sends a request and waits for a reply. Figure 1(a) shows a Petri net representing the behavior of this client. A token in the place p1 (respectively, p2) indicates that the client is ready to send a request (waiting for the reply). Transition t1 (respectively, t2) denotes the event of sending a request (arrival of the reply). Initially, the transition t1 is enabled. Figure 1(b) shows the net after the firing of transition t1; now the transition t2 is enabled. This Petri net is pure, ordinary, a state machine and
does not contain any source or sink transitions.

**Example 2:** This example is a modification of Example 1. The client can rise up to 5 requests. The server can receive and simultaneously process up to 3 requests. The server can also reject a received request without processing it - for example, due to invalid data in the request. Figure 2 (a) shows the Petri net for this simple client-server system. Figure 2(b) shows the interpretation of the places and transitions. This Petri net is pure, ordinary, and does not contain any source or sink transitions but does contain a conflict at place $p_3$. At the end of the firing sequence $t_1$, the net contains concurrent transitions; e.g., $t_3$, $t_4$. 

**Prolog Representation of a Petri Net**

We use several Prolog facts to represent a Petri net. Each place $P$ and transition $T$ is represented as a singleton fact place($P$) and transition($T$) respectively. Each incoming arc from a place $P$ to a transition $T$ having a weight $W$ is represented as a fact arc($T$, in, $P$, $W$). Similarly, each outgoing arc from a transition $T$ to a place $P$ having a weight $W$ is represented as a fact arc($T$, out, $P$, $W$).

$M$ marking of a Petri net is represented by the fact marking($M$ marking($L$)) where $M$ marking($L$) is a list of tuples of the form (place, $k$) where $k$ denotes the number of tokens in a place.

Figure 1. Petri net representation of a simple client.

**Syntax of Petri Nets**

- A Petri net is a 5-tuple $PN = (P, T, F, W, M_0)$ where $P = \{p_1, p_2, \ldots, p_m\}$ is a finite set of places $T = \{t_1, t_2, \ldots, t_n\}$ is a finite set of transitions, $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs (flow relation), $W : F \rightarrow \{0, 1, 2, 3, \ldots\}$ is a weight function, $M_0 : P \rightarrow \{0, 1, 2, 3, \ldots\}$ is the initial marking, $P \models T = f$ and $P \not\models T = \emptyset$. 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 $(N, M_0)$.

**Petri Net Representation**

Pictorially a Petri net structure is represented by a weighted, bipartite, directed graph consisting of two types of nodes: places and transitions. A place is drawn as a circle and a transition as a bar or box. An arc is represented by a directed edge from a place to a transition or from a transition to a place. The edge label is the arc weight (a positive integer). Conventionally, the edge label is not displayed for an arc whose weight is 1. A marking $M$ associates a non-negative integer with each place. If marking for a place $p$ is $k$, denoted $M(p) = k$, then the number of tokens is placed in the circle representing the place.

In a well-formed Petri net, there are no isolated places or transitions. There is at most one arc between a place and a transition. A place cannot be directly connected to a place and a transition cannot be directly connected to a transition. The weight of each arc must be a positive integer and the number of tokens at each place must be a non-negative integer.

**Program 1:** Prolog representation for some Petri nets given as the constant omega, then it means that the number of tokens in that place can grow without bound.

Program 1 displays the Prolog facts for the Petri net in Figure 1(a) and Figure 2(a).

**Simulation of Transition Firing in Prolog**

Program 2 displays a Prolog program for firing an enabled transition in a Petri net. This program also contains a number of useful auxiliary predicates for querying a Petri net. We assume that Prolog facts representing a Petri net (e.g., as in Figure 2(b)) were already consulted into the memory.

The predicate tokens_at(?Place, Tokens, MarkingL) returns the number of tokens at the given place as per the given marking. The predicate enabled_at(?TransitionName, MarkingL) succeeds if the given transition is enabled at the given marking. The predicate fire_at(?TransitionName, MarkingL) simulates the effect of firing the given transition. Essentially, it constructs the new marking from the given old marking, as per the transition-firing rule. This predicate fails on backtracking. In a marking, if the number of tokens in a place are

![Figure 2: Petri net representation of a simple client-server system.](image-url)
Querying and Classification of Petri Nets

Program 3: shows some predicates to classify and query a Petri net.

Predicate transitions_concurrent_at(T1, T2, M) true if given transitions are concurrent at the given marking. Predicates pure, ordinary, state_machine and marked_graph succeed if the given net is pure, ordinary, a state machine or a marked graph respectively.

Prolog facts in Program 1(a) represent a Petri net, which is pure, ordinary and a state machine. Prolog facts in Program 1(b) represent a net, which is pure, ordinary, and contains a conflict at place p3. After the firing sequence t1, t1, the Petri net contains concurrent transitions, e.g., t3, t4. This is verified below, assuming that Program 1(b) is already consulted.

Predicate transitions_concurrent_at(T1, T2, M) true if given transitions are concurrent at the given marking. Predicates pure, ordinary, state_machine and marked_graph succeed if the given net is pure, ordinary, a state machine or a marked graph respectively.

Program 3: Prolog predicates for querying and classification of a Petri net

Analysis of Petri Nets

Once a Petri net model for a system is constructed, it can be analyzed for several interesting properties such as reachability, boundedness, safety and liveness. There are several simple methods (which we will not cover) to analyze a given net for these properties.

The reachability problem is the problem of determining whether a given marking M is reachable from M0 in a given net N. In general, a marking may indicate a special scenario (e.g., a fault); so reachability analysis determines whether the system can ever be in an error or unsafe state.

A Petri net (N, M0) is said to be k-bounded or simply bounded if the number of tokens in each place does not exceed a finite number k for any marking reachable from M0. If the tokens represent some specific aspect of a real-life system, for instance a token represents a request in Example 2, so boundedness means the number of the requests accumulated without bounds at any place.

A 1-bounded net is called safe. A Petri net (N, M0) is said to be live if, from any marking reachable from M0, it is possible to ultimately fire any transition by progressing through some firing sequence, i.e., every transition is firable from any marking reachable from M0. Liveness is a strong property and indicates an absence of deadlocks. Several weaker notions of liveness have also been defined.

Manipulation of Petri Nets

Any simplifications can be applied to a Petri net without affecting its properties like liveness, safety and boundedness (Figure 3). It is easy to write Prolog predicates to implement such transformations.

Conclusions

This article has presented a simple toolbox of Prolog predicates to represent, simulate and analyze Petri nets. These programs can be enhanced to construct and simulate dynamic models of many different types of systems. Petri nets are also a novel way to represent dynamic knowledge or internal behavior of many AI systems. With enhancements, these programs can work with more powerful extensions of Petri nets such as predicate-transition nets, stochastic Petri nets or timed Petri nets.

References