Simulation of Petri Nets in Rule-Based Expert System Shell McESE

F. Franek and I. Bruha

Dept of Computer Science and Systems, McMaster University
Hamilton, Ont., Canada, L8S4K1
Email: {franya | Bruha}@mcmaster.ca

Abstract

There exist various tools for knowledge representation and modelling in artificial intelligence. We have designed and built a software tool called *McESE* (*McMaster Expert System Environment*) that comprises three such representations: production systems, neural nets, and Petri nets. The first two tools are almost complementary in their strengths and weaknesses: neural nets are very good in noisy-data processing but weak in high-level reasoning, production systems exhibit reasonably good performance in high-level reasoning but are weak in handling imprecise and uncertain data. Petri nets, another widely used paradigm, are useful in modelling concurrent processes and simulation of discrete processes. Consequently, some researchers are trying to combine these tools in order to utilize the strength of each system.

This paper describes the way of simulating Petri nets in the expert system shell McESE.

1. Motivation

McESE (*McMaster Expert System Environment*) [F], [FB1] is an interactive environment for design, creation, and execution of backward chaining rule based expert systems. The main objectives of the project focused on two aspects: extension of regular languages so they be able to deal with rule bases and use them (at this time two particular extensions are at use, Common-Lisp extension and C extension), and a way to deal with uncertainty.

The first objective is reached by providing the language being extended by a set of built-in functions dealing with the rule bases. In case of CommonLisp these functions themselves written in C are ported to CommonLisp via its C interface and so the user of McESE-CommonLisp has no way of distinguishing them from the built-in functions of CommonLisp and that is why we can call it an extension of CommonLisp. In case of C there was no problem to add these functions to the set of functions used in the interactive environment of McESE-C.

The other objective is reached by the particular design of McESE rules utilizing weights, threshold directives, and CVPF's (*Certainty Value Propagation Function*). McESE rules have the following syntax:

R: T_1 & T_2 & .... & T_n $==F==>$ T

$T_1,\ldots,T_n$ are left-hand side terms of the rule R and T is the right-hand side term of the rule R. A term has the form:

weight * *predicate* [*op cvalue*]

where *weight* is an explicit certainty value, *predicate* is a possibly negated (by ~ or -) predicate possibly with variables, and *op cvalue* is the threshold directive (*op* can either be >, >=, <, or <=, and *cvalue* is an explicit certainty value). If the weight is omitted it is assumed to be 1 by default. The threshold directive can also be omitted. The certainty values can be either integers or reals (in any range). The default is reals in the range 0..1 .

The value of a term depends on the current value of the predicate for the particular instantiation of its variables; if the threshold directive is used, the value becomes 0 (if the current value of the predicate does not satisfy the directive), or 1 (if it does). The resulting value of the term is then the value of the predicate modified by the threshold directive and multiplied by the weight.

A rule is eligible to fire if the value of all its left hand side terms is non-0. The firing of McESE rule consists of assigning the new certainty value to the predicate of the right hand side term (for the given instantiation of variables). The value is computed by the CVPF F based on the values of terms $T_1,...,T_n$. In simplified terms the certainty of the left-hand side terms determines the certainty of the right-hand side predicate. There are several built-in CVPF the user can use (*min*, *max*, *average*, *weighted average*), or the user can provide his/her own custom-made CVPF's. If the CVPF is omitted in a rule, *min* is used by default. This approach allows, for instance, with the default certainty values to create expert systems with fuzzy logic used to deal with uncertainty.

With McESE, as with any rule-based expert system, the problem which of the eligible rules should be fired arises. This is dealt with by what is commonly known as *conflict resolution*. McESE provides the user with three predefined conflict resolution strategies: *min* (where one of the rules leading to the minimal certainty value is fired), *max* (where one of the rules leading to the maximal certainty value is fired), and *rand* (where a randomly chosen rule is fired). The user has an option to use his/her own conflict resolution strategy.

To be able to make McESE to a useful tool for modelling and analysis of general systems, we were faced with the need to extend McESE to include neural nets and Petri nets. In [FB2] we described how neural nets can be simulated within McESE. Though the simulation was possible, it turned out that for any practical purposes the neural nets needed tended to be rather big and their execution in simulated form was kind of slow. Thus the new version of McESE has built-in means to define and train neural nets [K]. Of course only a few types of neural nets are available directly, thus for any other type the user would have to revert to the simulation approach as described in [FB2].

Since Petri nets come in many varieties and forms, and for many practical applications they do not tend to be as big as practical neural nets, rather than extending McESE yet more to deal with Petri nets, we decided to provide the user with the means to use the functionality of McESE (with some small modifications) to define, execute, and analyze Petri nets.

This paper deals with the ways Petri nets can be defined, executed, and analyzed within the framework of McESE. Due to the briefness of this exposition we cannot go into very technical details (that are unavoidable for implementation), but we hope that the reader gains clear ideas about the concept of how Petri nets can be simulated in McESE, the strong and weak points of our approach, and the usability of such a system. Since neither of the authors works in the area of pure or applied Petri nets research, we do not think that we can add anything significant to this field. What we are offering here is a software tool that can be conveniently used to design, execute, and analyze Petri nets.

2. Simulation of general Petri nets

The first task in creating a Petri net is to describe its topology. The net consists of *places* and *transitions* with oriented arcs going from the input places of a transition to the transition and from the transition to the output places of the transition.

We use the syntax of McESE rules to describe the topology of the Petri net being simulated. Each rule represents a single (unique) transition of the net, the left-hand side constant predicates (i.e. with no variables) denotes both the input and output places of the transition. The unique right-hand side constant predicate denotes the transition.

Let us illustrate this approach on a small example. Consider the transition T of a Petri net (Fig. 1.). The corresponding McESE rule describing the topology of the transition T may look thing like this (as we shall see later we will have to refine the notation, for we need to capture more than just the topology of a transition, but for the illustration at this point it suffices):

R: $2{*}P_1 \& P_2 \& 2{*}P_3$ & $\sim P_1$ & $3{*}\sim P_4$ & $\sim P_5$ ==> T

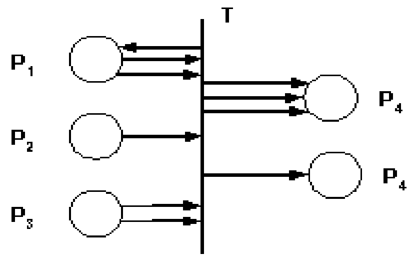

Fig.1 A sample Petri net.

The information carried by the syntax of the rule *R* does really describe the topology of the transition *T*, since the non-negated predicates of the left-hand side of *R*, i.e., P_1, P_2, and P_3, define the input places of *T* , while the negated predicates, i.e. P_1, P_4, and P_5, define the output places of *T*. The weights associated with the predicates (recall that when omitted the default value is 1) define the number of arcs going to or from the places. In particular, there are 2 arcs going from P_1 to *T* and one arc going from *T* to P_1. Thus, we can see that in such a manner we can describe any transition of a general Petri net.

Recall that a transition is *enabled* if every input place of the transition posses at least as many tokens as there are arcs going from that place to the transition. If a transition is enabled it may be chosen (based on some other criteria as we shall discuss later) to fire. If a transition fires the tokens are moved along the arcs from the input places to the output places. There are some conceptual restrictions needed: all tokens are deemed equivalent so it does not matter which token is moved where and the act of firing is considered an atomic event that takes no time from the point of view of the execution of the Petri net.

Thus, a McESE rule representing a transition of a general Petri net must capture not only the topology of the transition but also the condition describing when the transition is enabled. For example, the rule *R* described above must be refined to the following form using threshold directives:

R: $2{*}P_1$[>=2] & P_2 & $2{*}P_3$[>=2] & $\sim P_1$[>=0] & $3{*}\sim P_4$[>=0] & $\sim P_5$[>=0] ==> T

The tokens, or to be more precise, the number of tokens a place holds, are represented by its certainty value. Hence, first we have to tell the McESE compiler what is the range and type of certainty values by McESE compiler directives:

#cv type integer
#cv range [1..infinity]

In fact, as the default type of certainty values is *real* and the default range is 0..1, the rule *R* would not compile in the way it was originally presented.

Let us discuss the new form of the rule *R* to explain the significance of the use of the threshold directives. As we already mentioned, the certainty value of a place represents the number of tokens the place holds. Thus, for instance, the threshold directive [>=2] used with P_1 will evaluate to 1 if and only if the current value of P_1 is >= 2 (i.e. P_1 holds 2 or more tokens); in other words $2{*}P_1$[>=2] will evaluate to 1 if and only if P_1 holds at least 2 tokens. The threshold directives [>=0] used with the output places will always evaluate to 1 no matter what the current value of the output place is. Thus, if all the values of the left-hand side expressions come to non-0, the transition is enabled, otherwise it is not. In this way the rule captures both the topology of the transition and the condition that makes the transition enabled.

The McESE rules in the form we just described are still not fully suitable to our goal of simulating the simple Petri nets. We have to be able to execute such a net, or put it more simply, we have to have means to fire a particular transition. At this point we cannot just simply rely on the evaluation (firing) mechanism of McESE's inference engine, since it can only modify the certainty value of the right hand side predicates. Nevertheless to stay within the philosophy of McESE and its operation, we provide yet another predefined CVPF with a specific name *trans* that actually modifies the certainty values of the left--hand side predicates according to the movement of tokens as prescribed by the rules of firing for general Petri nets. In particular, the rule R looks in its proper form as follows:

R: $2*P_1[>=2]$ & P_2 & $2*P_3[>=2]$ & $\sim P_1[>=0]$ & $3*\sim P_4[>=0]$ & $\sim P_5[>=0]$ =*trans*=> T

If and when the rule R is fired, the value of P_1 is decremented by 2 (based on the weight associated with P_1), the value of P_2 is decremented by 1, and the value of P_3 is also decremented by 2. Then the value of P_1 is incremented by 1, the value of P_4 is incremented by 3, and the value of P_5 is incremented by 1 as well. In this fashion, the tokens have been distributed according to the topology of the transition.

After a general Petri net has its topology fully defined and before it may start executing we need to make the initial *marking*, i.e. to place tokens to places we desire.

The mechanism of McESE in fact allows us to have any number of markings and engage them at any time we desire. To do so we have to define a special constant predicate *TOKEN*. It is going to be a level-0 predicate (see [FB1]), and hence it has to have a function associated with it. So we have to define such a function (if we are using McESE-CommonLisp the function must be defined in CommonLisp, if using McESE-C, it must be defined in C). Because *TOKEN* is a constant predicate, the corresponding function *TOKEN* has no arguments and we make sure that it always returns 1. Now we are ready to define a marking in a very simple way using the ordinary McESE mechanism. For example,

2**TOKEN* ==> P_1
3**TOKEN* ==> P_2
TOKEN ==> P_3
P_1 & P_2 & P_3 => $Marking_1$

describes a possible initial marking that relies on the fact that the initial values of all predicates are set to 0. The execution of McESE inference engine activated by *eval*($Marking_1$) (see [FB1]) before we have started execution of the Petri net will cause the value of P_1 be set to 2, the value of P_2 be set to 3, the value of P_3 be set to 1, regardless of their current values.

Sometimes we may want to reset the marking. We have to add rules of the type

TOKEN ==> ~*Q*

that set the value of *Q* to 0. Such a marking then can be executed at any time and will set the values of all places to either zero or the number of tokens desired. In this fashion, the rule base can contain more than one marking.

The last set of rules we have to add are the rules that make the execution of the net possible. Let $T_1,...,T_n$ be all transitions of the net being simulated.

T_1 ==> *Net*
T_2 ==> *Net*
...
T_n ==> *Net*

where *Net* is a constant predicate.

A small loop of the form (here presented in a pseudo-code)

```
while (value of Net != 0)
      eval(Net, rand)
```

will cause the inference engine to evaluate *Net*, to do so it has to evaluate the transitions $T_1,...,T_n$, so from the transitions that are enabled it randomly chooses one and fires it. As long as this can go on it does. The process stops when no transition is enabled and can fire in which case the value of *Net* is set to 0 by the inference engine. As we mentioned before, *rand* is a conflict resolution strategy built in McESE. If the user desires a different way of choosing which among enabled transitions is to fire, a custom made function can be used with the *eval* function.

To summarize what we have discussed up to now: we have means to describe the topology of a general Petri net, to set/reset its marking, to fire a single transition (by *eval*(*T*, .)), or to execute the whole net.

3. Analysis of Petri nets

The fact that we can fire a single transition allows the user to investigate in run-time if that particular transition is live with respect to the given marking.

This brings us to the other important topic concerning Petri nets, their analysis.

At this stage of McESE project no static analysis is yet possible. But the fact that the Petri net is described by a McESE rule set that is compiled to a linked data structure gives us means to perform such static analysis concerning many important properties of Petri nets in the near future. In particular we plan to add computation of *reachability tree* [P] for a given net. That would allow static analysis of safeness, k-boundedness, and conservation.

Nevertheless, even at this stage McESE can still be a valuable tool for analysis of Petri nets.

For example, the user might want to know if the net he/she just designed is safe (or more generally if it is *k-bounded*). Although McESE cannot perform yet the analysis of the net and answer the question, it can do so in run-time. If the range of *cv* is defined [0..1], the inference engine in its debugging mode would alert the user if at any point of execution a certainty value of a place (i.e. the number of tokens) is out of range. In the same manner the *cv* range [0..k] can be used to test the net in run-time whether it is k-bounded.

Another of often desired properties of Petri nets is *conservation.* A Petri net is strictly conservative with respect to an initial marking if the number of tokens remains constant for all reachable markings. This idea can be generalized by having weights assigned to the places and requiring that the sum of the weights of the tokens (a token assumes the weight of the place it is residing at) remains constant. In either case McESE can provide run-time checking of either form of conservation.

4. Restricted Petri nets.

The usual restrictions considered are in fact restrictions on topology of transitions: ordinary Petri nets (no multiple arcs are allowed), nonreflexive Petri nets (no self loops are allowed, the input place of a transition cannot be its output place), and restricted Petri nets (nonreflexive ordinary Petri nets). As such they pose no problems for their simulation in McESE.

Petri [P] considered, what we could call now a restriction, a different rule for execution. A transition was enabled if in addition to the input places having enough tokens, the output places were empty. McESE allows for such a Petri net in an easy modification. Consider again the rule *R* in its last form as presented above. The threshold directives associated with output places had all form [>=0] to make the corresponding term always 1. If we change the threshold directi ves to [>=1] we can change the condition when the transition *T* is enabled. E.g., assume P_4 does not posses a token, i.e. its certainty value is 0. Then the value of $\sim P_4$ is 1, and so the value of $3*\sim P_4$ is 3 and the value of $3*\sim P_4$[>=1] is 1. On the other hand, if P_4 does posses token(s), its value is >= 1, hence the value of $\sim P_4$ is 0, and the value of $3*\sim P_4$[>=1] is 0. Thus, a transition in this from would be enabled only if all its output places had no tokens in them.

5. Conclusion

The paper describes the way of emulating Petri nets in McESE. This software tools [FB1], [F] was originally designed and built as an expert system shell. In order to utilize the strength various AI knowledge representations and thus the system a hybrid one, we firstly added an emulator of neural nets (multilayer perceptron) within the rule-based expert system shell [FB2].

Lately, we have incorporated an emulator of Petri nets into McESE, thus completing on edge of the "triangle" of various knowledge representations (Fig. 2). Emulating neural nets in terms of Petri nets can be found e.g. in [A]. [E] displays a method of building production rules by exploiting a multilayer perceptron as a classification oracle (the dotted arc in our triangle). Because Petri nets importance for theoretical computer science as well as for simulation of discrete processes in on the rise, we found a need to augment McESE to handle Petri nets in addition to production systems and neural nets (the bold arc). Consequently all three modelling paradigms may thus be exploited within one system McESE.

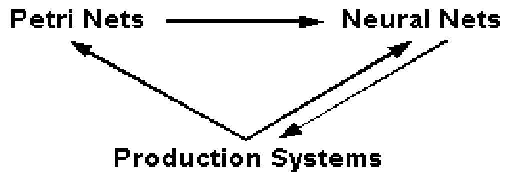

Fig. 2. A "triangle" of knowledge representations.

References

[A] P. Abellard et al., *A data flow Petri net approach to neural networks*, in [M], 1991

[E] M. Egmont-Petersen, *Homomorphic transformation from neural nets to rule bases*, in [M], 1991

[F] F. Franek, *McESE-FranzLISP: McMaster Expert System Extension of FranzLisp*, in: Computing and Information, North-Holland, 1989

[FB1] F. Franek, I. Bruha, *An environment for extending conventional programming languages to build expert system applications*, Proc. IASTED Conf. Expert Systems, Zurich, 1989

[FB2] F. Franek, I. Bruha, *Simulation of neural nets in expert system environment McESE*, in [M], 1991

[M] E. Mosekilde (ed.), *Modelling and Simulation*, Proc. European Simulation Conf, Copenhagen, 1991

[K] D.L. Knyf, *Incorporation and training of neural networks in McESE expert system*, M.Sc. Thesis, Dept. Computer Science ans Systems, McMaster univ, 1993

[P] J.L. Peterson, *Petri net theory and the modelling of systems*, Prentice-Hall, 1983