Taking Advantages of Temporal Redundancy in High Level Petri Nets Implementations

J. A. Bañares, P. R. Muro-Medrano and J. L. Villarroel

Departamento de Ingeniería Eléctrica e Informática
UNIVERSIDAD DE ZARAGOZA
Maria de Luna 3, Zaragoza 50015, Spain**

Abstract. The aim of this paper is to present a software implementation technique for High Level Petri Nets. The proposed technique, implemented for a specialized version of HLPN called KRON, is interpreted and centralized. The approach makes use of the similarities between the inference engine of a rule based system and the interpretation mechanism of a HLPN. It performs an adaptation of the RETE matching algorithm to deal with HLPN implementations. As in RETE, the main objective is to exploit the data temporal redundancy, with this purpose, a RETE-like data structure is implemented. Additionally, our approach benefits from the partition of working memory facilitated by the HLPN. These peculiarities allow the generation of simpler data structures than the ones in more general production systems such as OPS5.
Keywords: Higher-level net models, Rule based systems, Petri net implementation, Matching algorithms.

1 Introduction

A Petri Net software implementation is a computer program created to simulate the firing of transitions following the theoretical behavioral rules imposed by the Petri net semantics. Petri nets implementation techniques can be classified in [CSV86]: *centralized* and *decentralized* implementations. In a centralized implementation, the full net is executed by just one sequential task, commonly called *token player*. Pieces of code associated to transitions (code can be associated to transitions to provide operational capabilities) can be executed as parallel tasks to guarantee the concurrence expressed by the net. In a decentralized implementation, the overall net is decomposed into subnets (normally, sequential subnets) and a task is created to execute each subnet. The other main classification for software implementations of Petri nets distinguishes *interpreted* and *compiled* implementations. An implementation is interpreted if the net structure and the marking are codified as data structures. These data structures are used by one or more tasks called interpreters to make the net evolve. The interpreters do not depend on the implemented net. A compiled implementation is based on the generation of one or more tasks whose control flows reproduce the net structure. No data structures are needed in compiled implementations.

** This work has been supported in part by project ROB91-0949 from the Comisión Interministerial de Ciencia y Tecnología of Spain and project IT-10/91 from the Diputación General de Aragón. J. A. Bañares is FPU fellow from the Ministerio de Educación y Ciencia of Spain.

The software implementation of High Level Petri Nets (HLPN) has not received much attention so far. Its most representative approaches are briefly stated in the following. [BM86] presents a decentralized and compiled implementation of a subclass of HLPN (PROT nets). A PROT net is partially unfolded and decomposed into sequential processes that are implemented as a set of communicating Ada tasks. Other representative approach has been stablished in [CSV86], where a centralized and interpreted implementation of Colored Petri Nets is proposed. One of the main ideas of this work is the extension of representing place concept to HLPN. Another centralized and interpreted implementation of HLPN is proposed in [Har87], in this case, several interpreters compete to make the net evolve. The net is codified in just one data structure. A highly decentralized implementation is proposed in [BBM89], each place and transition are implemented as an OCCAM process and the implementation runs on a transputer based architecture. [VB90], the closest related work, will be explained later.

The aim of this paper is to present a software implementation approach for HLPN developed for the KRON knowledge representation language ([MM90] and [Vil90]). The proposed technique is interpreted and, in principle, centralized. However, the basis of this technique can also be used in decentralized implementations. This technique is based on the similarities between the inference engine of a rule based system and the interpretation mechanism of a HLPN. In both, efficiency is an important consideration since rule based systems and HLPN based systems, may be expected to exhibit high performance in interactive or real-time domains. The proposed HLPN implementation technique makes use of an adaptation of the RETE matching algorithm [For82] which is used in the OPS5 [BFKM85] rule based language. As in RETE, the main idea is to exploit temporal data redundancies (comming from the marking that is not changed during transition firing).

The adequacy of the RETE match algorithm for software implementations of HLPN is getting greater interest within the research groups working on PN/AI. Thus, [BE86] and [DB88] adopt OPS5 as the implementation language whereas [Vil90] and [VB90] provide a more in-depth knowledge about the use of the OPS5 matching strategy (RETE) to deal with HLPN software implementations. The paper of Valette and col. provides a pretty nice conceptual and pedagogical view for the specific use of RETE for the matching process. The approach presented here goes further in these ideas, which have been refined with the feedback of our implementation experience. We propose a strategy to specialize the RETE match algorithm to be more suitable for HLPN software implementations. A specific implementation is proposed and some aspects, as the reduction of test tree, are treated with more deep. Some details, which are interesting from the software point of view (mainly related with data structures), are also considered.

The rest of the paper is structured as follows. In section 2 rule based systems and its interpretation mechanism are explained. Additionally, the matching pro-

cess and the RETE match algorithm are illustrated in more detail. In section 3 a special class of high level Petri nets (KRON nets) and its relationship with rule based systems are briefly presented. Software issues involved in the KRON interpreter and the RETE influences are considered in section 4. The advantages and limitations of the proposed approach are analyzed in section 5. Finally, section 6 collects some conclusions of the work.

2 Rule based systems components and rule interpreter mechanism

Rule based systems can be decomposed into three parts: rule memory, working memory and inference engine. The rule memory contains declarative information based in precondition/consequence sentences. The working memory (data memory) contains dynamic data that is compared with the precondition part of the rules. The individual elements of the working memory are referred to as the working memory elements. The rule interpretation mechanism is materialized by the inference engine. The inference engine executes what is called a recognize-act cycle [BFKM85] which is itself composed by three main activities:

1. Match: Performs the comparison of the dynamically changing working memory elements to the precondition part of the rules. If a precondition is satisfied, the rule is included in the conflict set (the set of executable rules for the present working memory state). Currently, the match phase problem is often solved by the RETE match algorithm [For82].
2. Select: Selects one rule from the conflict set. Two main strategies are usually used depending on the recency of individual condition elements and the specificity of the precondition (LEX and MEA strategies).
3. Act: Executes the selected rule according to its consequence part.

The inference engine cycles until no rules are satisfied or the system is explicitly halted.

2.1 Matching process

The match phase of recognize-act cycle consumes the most time of the rule interpreter mechanism (using conventional approaches the interpreter can spend more than 90% of this time). To have a better understanding of the computational complexity involved in this phase we will have a look at the different steps of the matching process in rule based systems.

Rule preconditions are composed by condition elements. In a condition element, an attribute name is followed by an expression specifying restrictions on its value. If the expression specifies a constant value, the condition element will match only the working memory elements with that value. The expression can also be a variable, in this case it will match every working memory element having that attribute. For every match, a binding between the variable and the

attribute value of the matched element is created (the variable is bound to the value it matches). When a specific variable can be bound to the same value for each occurrence in the precondition part of the rule, we say that a consistent binding has been found [BFKM85].

From the computational point of view, the set of variables of the rule can be seen as a pattern that must be specified in a consistent way according with the restrictions imposed by the preconditions. The calculation of these patterns can be made in a step by step manner. At the first level, the possible bindings related to the first and second preconditions, are establishing resulting in two sets of patterns partially specified. Next, the consistency of each of the patterns of the first set must be contrasted for consistency against each one of the second set. When two patterns are consistent, a new pattern is generated that will hold the bindings or specifications of both (in general, more specific than its predecessors). Next, the possible bindings with respect the third condition element must be computed and its consistency must be contrasted against the set of patterns computed in the previous level. The result will be again another set of patterns still more specifics. This process goes on until restrictions of the last precondition are integrated and the set of consistent bindings represented by completely filled patterns are obtained. The calculation strategy is graphically illustrated by the binary tree shown in figure 1.

This general framework is instanciated for different match algorithms. Their main differences are the calculations they make each cycle and the information stored to remember previous calculations. In straightforward implementations there are many redundancies in the matching process which decrease efficiency. This execution efficiency problem led to the development of the RETE algorithm [For82], the TREAT algorithm [Mir86] and other related algorithms (see [Pas92] or [SPA92] for review). In the following paragraphs we explain RETE, a well-recognized algorithm which exploits these redundancies, and then we will see how these ideas can be used to improve the efficiency in high level Petri nets software implementations.

2.2 The RETE algoritm

The RETE match algorithm [For82], originally implemented in the OPS5 rule-based programming tool [BFKM85] avoids the brute force approach taking advantage of *temporal redundancy* (persistence of information in the working memory across the recognize-act cycle is called temporal redundancy). This is accomplished by matching only the changed data elements against the rules rather than repeatedly matching the rules against all the data. With this purpose, the information about previous matchings is recorded in a graph structure called *network*.

The network is composed by a global *test tree*, common for all rules, and a *join tree* specific for each rule. The test tree is composed by *one-input* nodes, each of them representing a test over some attribute. A path between the root node and a leaf node represents the sequence of tests required by a rule precondition. The consistent bindings are established following the computation strategy explained

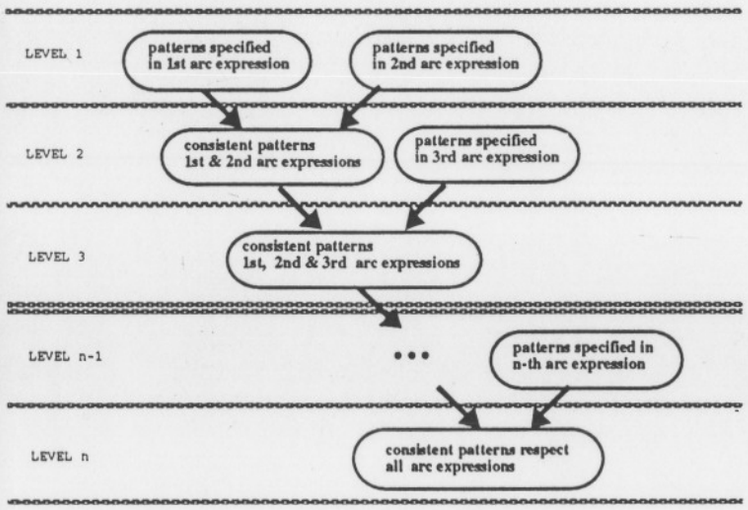


Fig. 1. Graphical view of the general computation process used during the match phase.

above (see figure 1). The information of this process is represented by the join tree associated to the rule (join tree nodes are called *two-input* nodes).

When a working memory element is added to the working memory, a pointer to the element is entered in the root of the test tree and propagated in case the test is successful. The working memory element pointers coming out of the test tree leaves are entered in the join tree. Then, the pointers are combined in tuples and stored in each two-input node. Each tuple collects the working memory elements allowing a consistent binding at the corresponding level. A tuple of the root node points to the working memory elements with which the associated rule can be applied.

On the other hand, when working memory element is removed, the corresponding pointer must be removed from the test tree root. The tuples having this pointer must be also removed from the two-input nodes. When many working memory elements match the same condition element, removing a working memory element is expansive. It takes a linear search to find the particular element to remove in the list of pointers for a condition.

3 High level Petri net/rule based model equivalence

Different versions of high level Petri nets have been proposed in the scientific literature (see the book of Jensen and Rozenberg [JR91]), we restrict ourselves here to the approach used in our software implementation, which is called KRON. KRON (Knowledge Representation Oriented Nets) is a frame based knowledge representation language for discrete event systems with concurrent behavior. It

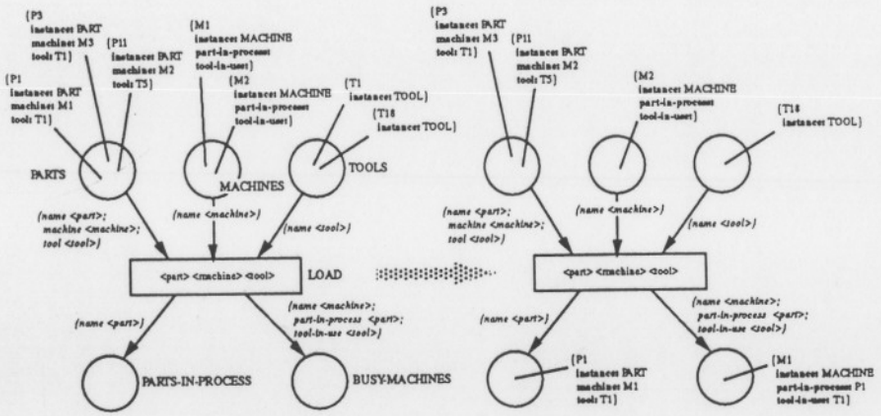


Fig. 2. A simple KRON net example.

belongs to the object oriented programming paradigm and incorporates a specialization of the color Petri net formalism for the representation of the dynamic aspects. The reader is referred to [MM90] and [Vil90] for more theoretical bases.

KRON provides a set of classes of specialized objects and primitives for the construction of a system model where the dynamic behavior is defined by an underlying HLPN that we will call in the sequel KRON nets. Given the scope of this paper, we will briefly introduce KRON through an illustrative example of a very simple KRON net, which is shown in figure 2. The net, composed by five places and one transition, is representing the load for a set of machines. The arcs are labeled by expressions that are lists of attribute-variable pairs. The set of all variables included in the arc expressions is associated to each transition. Each pair defines possible bindings between a variable and attribute values of tokens in the related place. For example, arc expression {`name <part> ; machine <machine> ; tool <tool>`} in transition `LOAD` allows three different bindings for variables `<part>`, `<machine>` and `<tool>` regarding to the actual marking:
`(<part> = P1; <machine> = M1; <tool> = T1)`
`(<part> = P3; <machine> = M3; <tool> = T1)` and
`(<part> = P11; <machine> = M2; <tool> = T5)`.

A transition is enabled if there exists a *consistent binding* for its associated variables. This means that all transition variables are bound, the bindings from all arc expressions are the same and the variable values verify the restrictions imposed by the predicate associate to the transition. Each of these consistent bindings defines a different *firing mode*. In the example shown in figure 2, there exists only one firing mode that is defined by the following consistent binding:
`(<part> = P1; <machine> = M1; <tool> = T1)`.

The similarities between Petri nets and rule based systems have been pointed out in several papers [BE86], [DB88], [VB90], [HLMMM91], [MMEV92]. A HLPN can be interpreted as a rule based system in which transitions have the role of

rules and tokens are the working memory elements. We are interested here in the aspects of this equivalence that are important for the interpretation mechanism.

Given the approach adopted in KRON, the relation between a KRON transition and a OPS5 rule follows in a very intuitive way. A transition of a KRON net can be easily interpreted as a rule with variables:

1. The precondition part of a rule-transition is composed by the conditions defined by the input labeled arcs. Each input arc expression can be assimilated to a *condition element* of a rule.
2. The consequence part collects the marking updates defined by both the input and the output labeled arcs.

To illustrate this equivalence, let us consider again transition LOAD from figure 2. This transition can be expressed as a OPS5 rule following the approach adopted in [BE86]:

```
(p LOAD
    (PART ↑name <name> ↑machine <machine> ↑tool <tool> ↑place PARTS)
    (MACHINE ↑name <machine> ↑place MACHINES)
    (TOOL ↑name <tool> ↑place TOOLS)
→
    (modify <part> ↑place PARTS-IN-PROCESS)
    (modify <machine> ↑part-in-process <part> ↑tool-in-use <tool>
        ↑place BUSY-MACHINES)
    (modify <tool> ↑place NIL))
```

In this approach, tokens (data elements) have a special attribute called (↑place) used to specify the place where the tokens are. The RETE test tree and join tree associated to the rule LOAD is shown in figure 3.

On the other hand, HLPN provide some features that make its implementation more specific than the one in general rule-based systems. The main differences are related with the *working memory*. A rule based system has just one global working memory for all rules, whereas a HLPN has its working memory splited in places. The preconditions of a transition only must match against the tokens (data elements) of its input places. This fact allows a place to be seen as a working memory partition. The main effects on a RETE network produced by this partition can be established as follows:

- The root node is splited into several nodes, one for each input place, each one defining itself a local working memory.
- The test tree of the network is reduced because no class or place tests are needed. Consider the OPS5 rule of transition LOAD. Each precondition element of this rule has an implicit test over the token class and an explicit test over the place where the token is located. Place tests can be avoided making use of the HLPN structure because each condition element has implicitly associated an input arc. The class test (PART, MACHINE or TOOL) can also be avoided because each place has associated a set/class of possible tokens. All tokens in a place belong to the same class.

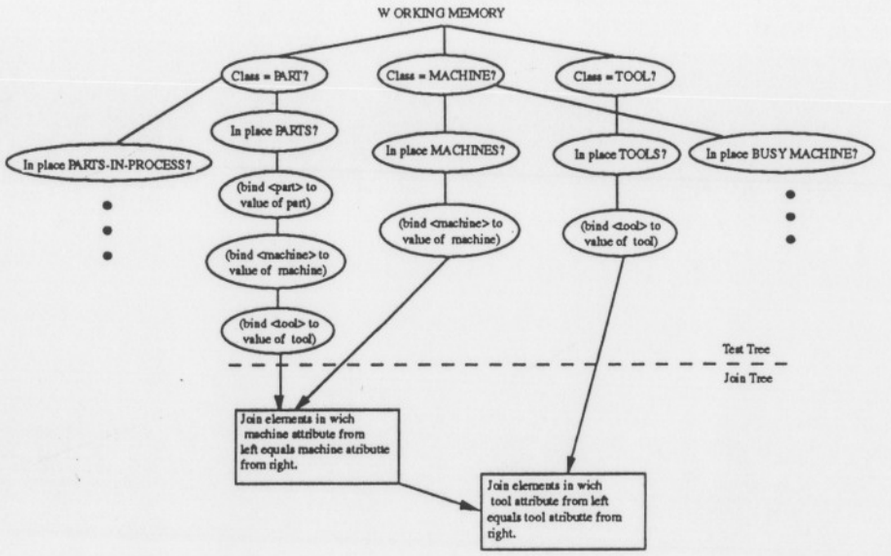


Fig. 3. RETE network generated from transition LOAD.

- It can not make use of the *structural similarity* [BFKM85]. The structural similarity allows the sharing of partial branches of the test tree by different, but similar, condition elements. Except in places that are shared by several transitions, the branches of the test tree come out from separate root nodes. This fact makes impossible for nodes to be shared.

If the structural similarity reminded are not used, the test tree obtained is a set of separated chains coming out from root nodes. Thus, each chain can be reduced to only one node representing all test and bindings of the corresponding condition element. It is graphically illustrated by the network shown in figure 4. It can also be compared with the RETE network in figure 3.

4 Interpretation mechanism in KRON

4.1 Matching phase

Our approach for the interpretation mechanism makes use of the temporal redundancy features pointed out in the RETE matching algorithm. With this purpose, a similar data structure to the RETE network is implemented. Additionally, our approach benefits from the partition of working memory provided by the HLPN that implies the generation of simpler networks than the ones in more general production systems such as OPS5.

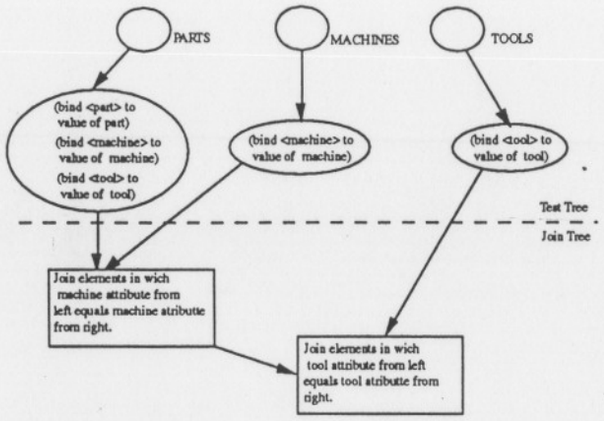


Fig. 4. Network generated from LOAD transition profiting by the HLPN structure.

In the KRON interpreter the tuple of variables associated to a transition is considered a pattern. Similarly to RETE, a tree is generated for each KRON transition. These trees are composed by two node classes:

1. Entry Nodes. Do not have predecessors and correspond to patterns with bindings generated by a single input arc. Each entry node has a link to the corresponding input place which is, in fact, a pointer to its local working memory. These nodes correspond to folded branches of the test tree in a RETE network.
2. Join Nodes. Correspond to consistent patterns with respect to several input arcs. These nodes are equivalent to the join nodes in a RETE network.

The tree is organized into levels:

- LEVEL 1: Two entry nodes corresponding to the patterns of the first and second input arcs (any order is accepted).
- LEVEL i (1< i < n): An entry node corresponding to the i+1 input arc and a join node created upon the nodes of level i-1.
- LEVEL n: A join node, which represents the consistent bindings of the transition.

Calculation is performed in an increasing direction of levels and, inside each level, from left to right. As in a RETE network the intermediate results of the matching process are stored and recalculation can be avoided meanwhile no changes in the allocation of the tokens of the places involved are made. To do that, the information about the set of patterns partially specified is stored in a special data structure. To provide fast accessibility each of these patterns is linked to its predecessors and successors. The patterns corresponding to the entry nodes do not have predecessors. In this case, a link is set between the pattern and the token that generated it. Figure 5 shows a graphical representation of

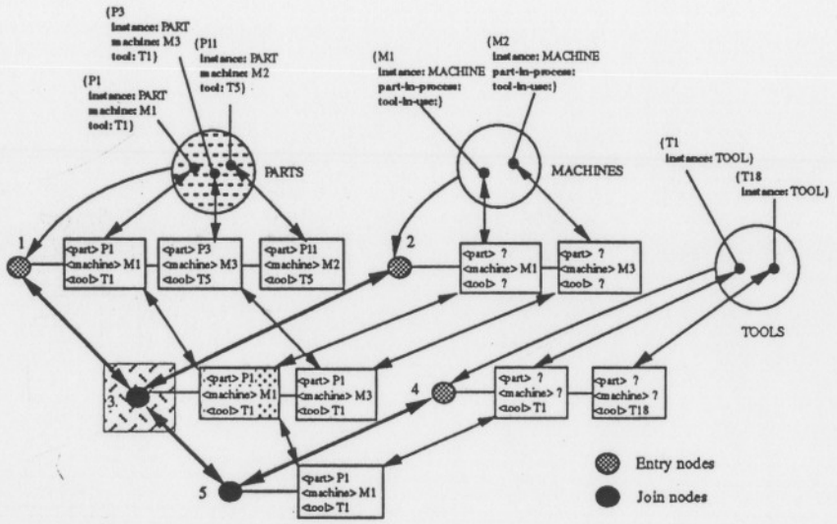
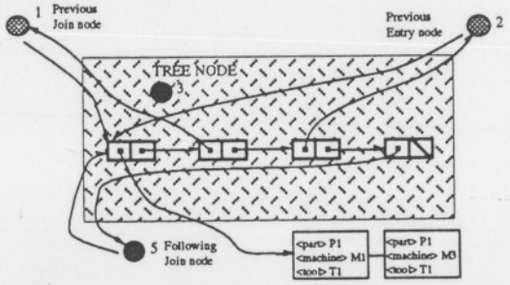


Fig. 5. Link structure for the LOAD network.

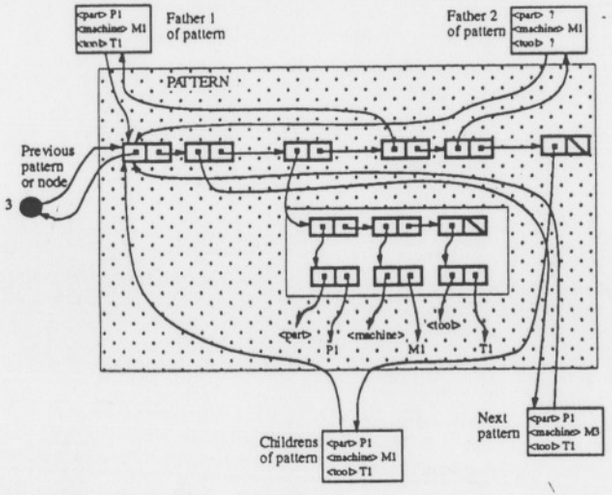
the network, this highly related data structure allows a fast tree update because it avoids computational searches.

Figure 6 shows more details about the link structure developed in our implementation. To facilitate software development, KRON has been implemented in KEE (Knowledge Engineering Environment from Intellicorp) which runs on top of Common Lisp. The interpreter itself has been implemented exclusively using Common Lisp primitives (Common Lisp is specially attractive to deal with pointers). The typical Lisp *cons cells* are used in the figure to describe more graphically the link structure. A cons cell is stood for a box with two pointers. The cons cells are linked by arrowlike pointers emerging from their right and left sides. Figure 6.a shows a prototypic tree node that is composed by a list of pointers: the first one points to the list of patterns, the rest are pointers to the previous and successor nodes. Figure 6.b shows the data structure for a pattern composed by: a list of pointers to the previous and following patterns at the same and different levels, another pointer points to an association list of pairs' variable-value (the tuple of partially specified variables of the transition). Finally, figure 6.c shows the data structure used in the place: the first element is a list of pointers to the entry nodes that pick up the patterns with bindings, the rest is a list with the links that are needed for each token in the place, such as the pointers to its generated patterns.

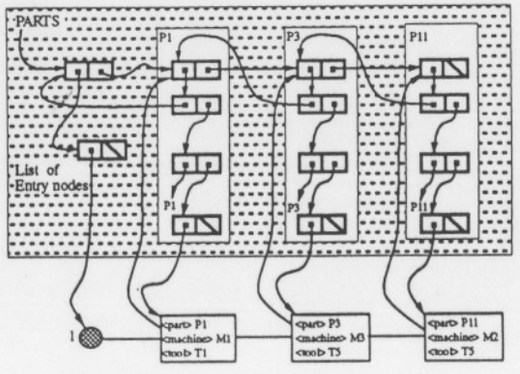
From the data structure point of view, there are two aspects that make differences between the RETE algorithm and the KRON interpreter. The first one is that partially specified patterns are stored instead of tuples of pointers as happens in the RETE algorithm. Another difference is related with the link



(a)



(b)



(c)

Levels of the link structure

structure between patterns that avoids searches in a node pattern list when an element must be eliminated.

The data structure created during the matching process is not modified until a change in the token distribution happens. Two cases must be distinguished:

1. **Deleting a token**. Carries on the elimination of the patterns specified by this token. The calculation trees of all descending transitions of the place containing the token are examined, and its successors' patterns deleted.
2. **Adding a token**. Originates new bindings of the transition variables. If the token has been added in the place corresponding to the i-th input arc, a new pattern or patterns are created in the entry node of the level i-1. The new pattern causes the verification of its consistency with respect the node of the same level. For each detected consistency a new pattern is created in the immediate following level. Actualization is propagated following this strategy.

4.2 Selection phase

Select one rule from the conflict set is accomplished in rule-based systems applying concepts such as recency, refraction or specificity (see strategies LEX and MEA from OPS5 language [BFKM85]). However, there are other important issues to be considered in implementing HLPN:

1. A HLPN can model concurrence. In each execution cycle, more than one enabled transition can be fired.
2. The strategies to solve conflicts depend on the application, and the resolution objectives can be different in each conflict.

In the KRON interpreter, transitions are grouped by conflicts, each one having its resolution strategy that we call control policy. During selection, all enabled transitions in a conflict are considered together, the conflict control policy is the responsible to provide a solution (transitions and firing modes must be chosen). The KRON interpreter offers a control policy library but the user can design new control policies according to the application domain.

In each execution cycle the set of all enabled transitions are known due to the incremental operation of the KRON interpreter [VB90]. It is an important difference from other centralized implementations, as the ones based on representing places [CSV86], where only a list of likely enabled transitions is known. This list can contain transitions that are not enabled. The consideration of these additional transitions makes the efficiency of the interpreter to decrease.

4.3 Firing phase

A transition firing with respect to a firing mode (pattern of the n level) carries on the following steps:

1. The calculation tree is traversed backwards from the consistent binding (firing mode) to find out which are the tokens used to generate it. Each of these

tokens is then removed from the input places and a tree update is performed in all the transitions sharing these input places.
2. The piece of code associated to the firing mode is executed.
3. The tokens specified by the output arcs of the transition are added to the output places and a tree updates is performed in all the transitions having these places as inputs.

5 Efficiency issues

The efficiency of a Petri net implementation can be measured by a function of the number of operations performed by transition firing. In the proposed implementation a transition firing carries on the elimination of the tokens produced by the firing mode and the addition of tokens in the output places. In these activities tree elemental operations are involved: creating a pattern, delete a pattern and make new pattern matching.

In a transition firing, the number of performed operations depends on two main factors: the number of tokens removed from the input places and the number of tokens added to the output places. The cost of adding a token in a place depends on the number of descending transitions, that is the number of trees to be updated. The operations involved in adding a token are: pattern creation and pattern matching. These operations depend on:

1. The number of input places. A large number of places means a deep join tree with many entry nodes. The average number of pattern matching operations grows with the deep of the join tree.
2. The patterns corresponding to entry nodes. The number of patterns in entry nodes depends on the number of tokens in places, the expressions labeling arcs and the cardinality of token attributes. Thus, if a token's attribute holds many values, one token may generate many patterns in an entry node. The number of pattern matching operations also increase if the number of patterns is bigger.
3. The arcs order. The interpretation mechanism evaluates the expressions labeling the arcs sequentially, stopping when there are not consistent matches with an initial sequence of arc expressions. Locating earlier the arc labeled with the most restrictive expression, or the one who has its place normally empty reduce the number of consistent matches that are passed down in the join tree. It makes sense to locate at the end the arcs whose matches are changed frequently (arcs whose places support frequently operations of adding and deleting tokens).

Pattern deletion is the operation involved on removing tokens from places. The number of these operations depends on the former factors in a similar way.

Unfortunately, the application of these ideas for improving efficiency and getting down the cost of calculate the firing modes are sometimes contradictory. For example, the principle of putting frequently changing marking places later may be in conflict with the principle of putting arcs with restrictive expressions earlier if an arc is both restrictive and matched by the tokens that are frequently put and deleted from their place.

For centralized implementations, the main emphasis has been put on the selective scanning of transitions. These implementations avoid the checking of all transitions, and the idea is to make a quick selection of a subset of transitions likely to be fired. In non incremental implementations, the number of operations in an execution cycle can be considered as: the number of created patterns and pattern matching operations coming from completely recomputing the firing modes of the fired transition and the transitions likely to be fired.

The adopted approach can be more efficient than non incremental implementation because avoids recalculations and considers only totally enabled transitions. However, there are situations where a non incremental technique may be more efficient as in the HLPN of figure 7. Let us show the number of operations by the firing of transition T_1 using both, the KRON interpreter approach and another non incremental technique. When transition T_1 is fired, token A_1 must be deleted from place P_1 and the same token must be added to place P_1 again.

Using the KRON interpreter approach, on the one hand deleting A_1 from P_1 implies that 7 patterns must be removed from the tree attached to T_2, and another one from tree attached to T_1. On the other hand, adding A_1 in P_1 implies that 7 patterns must be created and 6 pattern matching operations must be performed in the tree attached to T_2 and 1 pattern is created in the tree attached to T_2.

Nevertheless, using a non incremental implementation such as *representing places* the number of operations is reduced. Let us take P_1 as representing place of T_1 and P_4 as representing place of T_2. In the place P_4 there are no tokens, therefore only the transition T_1 is in the list of likely enabled transitions. The completely recompute of firing modes of transition T_1 generates 2 patterns. The difference from KRON approach is obvious.

Since the RETE algorithm maintains state between cycles, KRON interpreter is efficient in situations where there are many tokens and most of them do not change on a cycle. On the other hand, the number of operations in a transition firing depends on the output transitions of the input and output places. That is, the average number of operations by transition firing depends on the average number of output transitions. There are two situations where the KRON interpreter does not take advantage of temporal redundancy: when the number of tokens that are involved in a transition firing is large with respect the total number of tokens; and when the average number of descending transitions from places is high. In both cases the number of operations required to update the stored information can be greater than the number required for a recalculation.

6 Conclusion.

This paper provides inside knowledge of a software implementation for HLPN. To be more specific we use a specialized version of HLPN called KRON. KRON

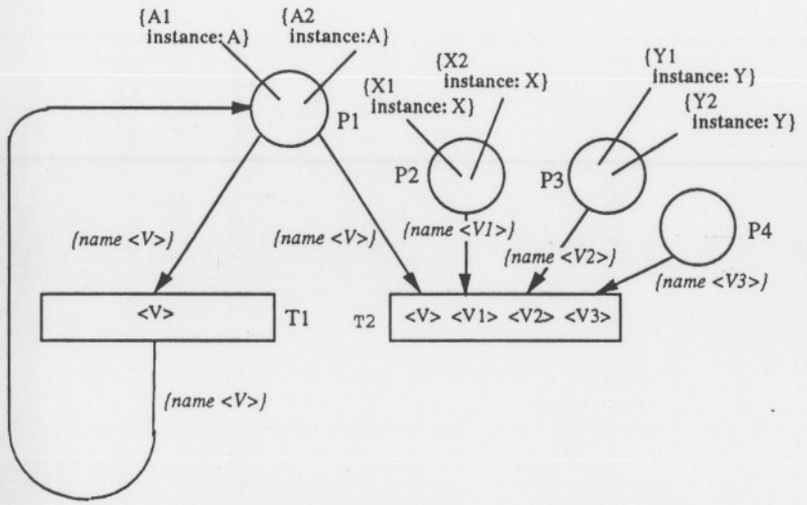


Fig. 7. A sample where the KRON interpreter shows lower efficiency.

(Knowledge Representation Oriented Nets) is a frame based knowledge representation language for discrete event systems. In this representation schema, the system dynamic behavior is represented by a HLPN that we call KRON net.

We follow an interpreted and centralized approach. This interpreter makes use of the similarities between the inference engine of a rule based system and the interpretation mechanism of a HLPN, specifically it uses an adaptation of the RETE matching algorithm used in the OPS5 rule based language. As in RETE, the main idea is to exploit temporal data redundancies.

A strategy to specialize the RETE match algorithm to be more suitable for HLPN software implementations has been proposed. Our approach for the interpretation mechanism makes use of a similar data structure to the RETE network. Additionally, our approach benefits from the partition of working memory in places provided by the HLPN that implies the generation of simpler networks than the ones in more general production systems such as OPS5

From the data structure point of view, there are two aspects that make differences between the RETE algorithm and the KRON interpreter. The first one is that partially specified patterns are stored instead of tuples of pointers as happens in the RETE algorithm. Another difference is related with the link structure between patterns that avoids searches in a node pattern list when an element must be eliminated.

An important difference from other centralized implementations is that the operation of the KRON interpreter is incremental. On each execution cycle, the set of all enabled transitions are known instead of a list of likely enabled transitions. The consideration of these not enabled transitions makes the efficiency of the interpreter to decrease.

The KRON interpreter has been designed to take advantage of temporal redundancy. However, there are two situations where the KRON interpreter fails in this objective: when the number of tokens that are involved in a transition firing is large with respect the total number of tokens; and when the average number of descending transitions from places is high. In both cases the number of operations required for update the stored information can be greater than the number required for a recalculation.

We can conclude that the implementation technique proposed is efficient in situations where there is a large marking and it is relatively stable. KRON has been implemented in a SUN workstation running KEE. The interpreter has been written using Common Lisp primitives, whereas KEE primitives have been use to access to the objects' information.

References

[BBM89] R. Esser B. Butler and R. Mattmann. A distributed simulator for high order petri nets. In *Proc. of International Conference on Applications and Theory of Petri Nets*, pages 22–34, Bonn, 1989.

[BE86] G. Bruno and A. Elia. Operational specification of process control systems: Execution of prot nets using ops5. In *Proc. of IFIC'86, Dublin*, 1986.

[BFKM85] L. Browston, R. Farrell, E. Kant, and N. Martin. *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming.* Adisson-Wesley, 1985.

[BM86] G. Bruno and G. Marchetto. Process-translatable petri nets for the rapid prototyping of process control systems. *IEEE transactions on Software Engineering*, 12(2):346–357, February 1986.

[CSV86] J.M. Colom, M. Silva, and J.L. Villarroel. On software implementation of petri nets and colored petri nets using high-level concurrent languages. In *Proc of 7th European Workshop on Application and Theory of Petri Nets*, pages 207–241, Oxford, July 1986.

[DB88] J. Duggan and J. Browne. Espnet: expert-system-based simulator of petri nets. *IEEE Proceedings*, 135(4):239–247, July 1988.

[For82] C. Forgy. A fast algorithm for many pattern / many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

[Har87] G. Hartung. Programming a closely coupled multiprocessor system with high level petri nets. In *Proc. of 8th European Workshop on Application and Theory of Petri Nets*, pages 489–508, June 1987.

[HLMMM91] G. Harhalakis, C.P. Lin, L. Mark, and P.R. Muro-Medrano. Information systems for integrated manufacturing (insim) - a design methodology. *International Journal of Computer Integrated Manufacturing*, 4(6), 1991.

[JR91] K. Jensen and G. Rozenberg, editors. *High-level Petri Nets.* Springer-Verlag, Berlin, 1991.

[Mir86] A. Miranker. *TREAT: A new and efficient match algorithm for AI production systems.* PhD thesis, Dep. Comput, Sci., Columbia University, 1986.

[MM90] P.R. Muro-Medrano. *Aplicación de Técnicas de Inteligencia Artificial al Diseño de Sistemas Informáticos de Control de Sistemas de Producción.*

PhD thesis, Dpto. de Ingeniería Eléctrica e Informática, University of Zaragoza, June 1990.

[MMEV92] P.R. Muro-Medrano, J. Ezpeleta, and J.L. Villarroel. *Aceptado en IMACS Transactions*, chapter Knowledge Based Manufacturing Modeling and Analysis by Integrating Petri Nets, 1992.

[Pas92] A. Pasik. A source-to-source transformation for increasing rule-based system paralellism. *IEEE Tran. on Knowledge and Data Engineering*, 4(4):336–343, August 1992.

[SPA92] M. Sartori, K. Passino, and P. Antsaklis. A multilayer perceptron solution to the match phase problem in rule-based artificial intelligence systems. *IEEE Tran. on Knowledge and Data Engineering*, 4(3):290–297, June 1992.

[VB90] R. Valette and B.: Bako. Software implementation of petri nets and compilation of rule-based systems. In *11th International Conference on Application and Theory of Petri Nets*, Paris, 1990.

[Vil90] J.L. Villarroel. *Integración Informática del Control de Sistemas Flexibles de Fabricación*. PhD thesis, Dpto. de Ingeniería Eléctrica e Informática, University of Zaragoza, September 1990.

A Subset of Lotos with the Comp of Pla/Transition-Ne

Michel Beau, Gregor v. Bochm

Abstract

In this paper, we define a set of Lotos that can be Transition-nets (P/T-nets). hat means that specifica can be translated into finite/T-nets and validated t techniques. An important ject of our work is that P/T-nets can be simulated our Lotos subset. It m we put on Lotos in order to tain finite nets are mini also conclude that our Lotosbset and P/T-nets have power. To the best of our lwledge, no such bidire has been published before.

Topics:

Relationships between net ory and other approacl

1. Introduction

In this paper, we define a bset of Basic Lotos [Bo modelled by finite Place/Tsition-nets (P/T-nets). tions in that Lotos subset cbe represented and tra and validated using P/T-nerification techniques. work is that we show thatnversely P/T-nets can subset. It means that the istraints we put on Lot nets are minimally restrict. We may also conclude P/T-nets have equivalent nputational power. To no such bidirectional transion scheme has been pu

The problem of modell process-oriented langu CCS and CSP like langua by Petri nets has been Cindio et al. [Cind 83], ano et al. [Dega 88], [Golt 84a, 84b, 88], NielseNiel 86], Olderog [Olde considered CCS or CSP, both. Lotos has been Leon [Marc 89], and Gar l and Sifakis [Gara 90

[1]This work was perforn within a research proj ifications funded by Bell-Nhern Research (BNR) Institute of Montréal (CR). Funding from the Na ing Research Council of Cda is also acknowledge

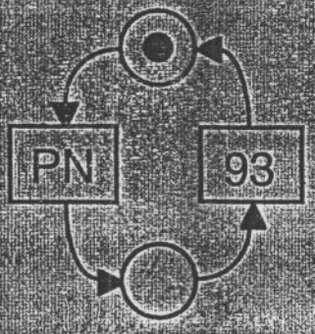
[2]First author's address: versité de Sherbrooke, Dé d'informatique, Sherbrooke (Qec), Canada, J1K 2R1. Se de Montréal, Département d'), C.P. 128, Succ. "A", M 3J7.

Lecture Notes in Computer Science 691

Marco Ajmone Marsan (Ed.)

Application and Theory of Petri Nets 1993

14th International Conference
Chicago, Illinois, USA, June 1993
Proceedings



Springer-Verlag