A Distributed Approach for Inferring Production Systems

Ching-Chi Hsu, Shao-Ming Wu and Jan-Jan Wu

Department of Computer Science and Information Engineering
National Taiwan University
Taipei, Taiwan, ROC

Abstract

Tools for building production systems encounter the problem of low performance, and many researchers are working on the improvements of performance of these tools. This paper proposes a distributed approach for inferring production systems. The resulting distributed production systems are expected to be built over distributed systems with broadcast capability, and production rules on different sites work in a cooperative way with only a few communications between them. Working memory on a local site is made visible to rules on remote sites. A tool for building distributed production systems, called DPS, has been implemented. DPS not only supports elegant constructs for expressing the capacity of distributed inference but also provides the facilities for building clusters of rules. With these facilities, DPS allows users to make the inference engine focus on a particular set of rules. This paper also describes the knowledge representation and other features about DPS.

I. Introduction

The development of tools for building rule-based systems has evolved in several directions. Consultation-based tools use the backward-chaining reasoning mechanism [1], together with questioning-and-answering facilities to achieve goals. Systems and tools with blackboard architecture [3,10,11] emphasize on the integration of complicated heuristic control and uncertain knowledge. There are also researches focusing on the architecture of forward-chaining reasoning mechanism, especially those derived from Rete algorithm [12,14].

The major advantages of production system tools such as OPS5 [5] and ORBS [4] are derived from the elegance, expressiveness and moderate complexity of their knowledge constructs. Users can create lists of attributes about concepts and a set of production rules each of which has condition elements to match against user-defined concepts, and actions to take when conditions are satisfied. The fact that condition elements of each rule join the relationship between concepts suggests more powerful expressiveness than other tools do. However, they pay for the expressive power of production rules. For example, OPS5 has to do more work on complex pattern-matching. Rete algorithm [6] for many-pattern /many-object pattern match problems was introduced to resolve the problem. Nevertheless, low performance is still a disadvantage of OPS5. So far, several approaches have been proposed to improve it, such as rewriting OPS5 into OPS83 [9] in C, improving the inference mechanism of OPS5 by eliminating unnecessary matching and supporting more powerful constructs for building production rules [12], and introducing parallel architectures [14, 7] to exploit the parallelism of pattern-matching and rule-firing.

In this paper, we propose a model for distributed processing of production systems. In this model, a production system, which can be constructed as a *virtual* integrated Rete network [6], is distributed over a distributed system or a local area network. Knowledge distribution is done by users. Subsystems on different sites are almost autonomous, and they can cooperate with each other. This greatly exploits the parallelism between subsystems since unrelated subsystems can run in parallel. A tool named DPS is constructed under this model. Moreover, we also support other features to improve the performance of production systems, including the split of Rete network on a single site into a cluster of subnets, object-based knowledge representation, and enhanced rule facilities.

II. Knowledge representation

The attraction of expressive power of knowledge representation in OPS5 has a great influence on the scheme of knowledge representation of DPS. We found that one of the reasons that make OPS5 popular comes from the semantic meanings of production rules. Since a rule consists of a condition part (LHS) and an action part (RHS), a rule can join the relationship between different concepts described in LHS, and then execute responding actions in RHS. The representative manner and complexity of such rules are at similar level of human thoughts, and hence facilitates the work of knowledge engineering.

In general, DPS has OPS5-like syntax of rules. In addition to the LHS and RHS, each rule of DPS may be given an attribute. When rules are interpreted, those with the same attribute are constructed as a sub-Rete-network which is used to narrow the inference space during execution of the systems. Moreover, users are allowed to use a prefix "?" or "? site-name" in a condition element to represent a condition element which matches against the working memory on remote sites. Such condition elements are called *remote condition elements*. This will be described later in more details.

DPS allows users to define object-based concepts [15], i.e., each concept is described as a list of attributes and is associated with any number of methods which are invoked by passing messages to an instance (working memory element, or wme) of the concept. The following is an example to illustrate the general form of knowledge representation in DPS.

```
(remote-site sensing-station)                    ; declare the remote sites
(object manager name location situation)         ; declare object "manager"
                ;whose attributes include "name", "location" and "situation"
(object store-house location material (status safe))  ; declare object
                ; "store-house", "safe " is default value of "status" attribute
(object material name stock property)            ; declare object "material"
```

```
(method manager emergency (name cause)  ; a method of "manager"
    ( LISP code for noticing and        ; A method can be invoked
        solving emergent events ))      ; by message-passing
            ; Here "emergency" is a message name, "name" and "cause"
            ; are arguments passed by the calling action.
(p setting-store-house-status           ; The third condition
    (store-house material: <x> status: safe) ; element matches against
    (material name: <x> property: volatile) ; the working memory on
? sensing-station                       ; remot site "sensing-
    (circumstance temperature-in-house: > 35)   ; station"
->
    (modify store-house
            status: dangerous
            cause: temperature-too-high)
    (focus emergent-ecent)              ; Make the inference
                                        ; engine focus on rules
      ::: sensing-events)               ; about emergent events.

(p emergency-manipulating
 <person> (manager
           name: <x>
           location: <y>
           situation: on-business)
    (store-house location: <y>
           status: dangerous
           cause: <z> )
    ->
    (send-message <person> emergency: <x> <z> )
                ; invoke the method associated with message
                ;"emergency" to solve emergent events
     ::: emergent-event )
```

The RHS of a rule allows an action to invoke a LISP procedure by message-passing. This procedure not only accepts arguments passed from calling actions, but also uses as variables the attributes in the matched wme. Compared with the "call" action in OPS5 which demands the special treatment of passed arguments, message-passing of DPS simplifies the use of procedure calls. The RHS of a rule also allows actions to control inference engine and manipulate working memory.

Rule Clustering

The ORBS production system [4] and blackboard systems such as Hearsay-II [3, 8], HASP [11] and AGE [10] allow rules to be grouped into related collections. There is generally a pointer that designates the current collection. Only rules from the current collection are allowed to match against working memory. There are several attractive language properties of rule clustering: it allows a complex space to be broken into more understandable pieces; it tends to simplify rule debugging because the debugging space can be narrowed to one rule cluster instead of the whole rule base; and it allows the system developer to reuse components (the rule clusters) found to be useful in the past.

In DPS, a rule can be given an attribute at the field indicated by the symbol ":::". Rules with the same attribute are grouped together. A subsystem on one site has a working memory and a number of rules grouped into a number of clusters, if needed. The inference engine may focus its inference space on one or more rule clusters. The switching of inference space among these clusters is guided by the selection of appropriate rule attribute. Fig.1 shows two rule clusters mentioned above and the working memory they share on the local site. When the rule setting-store-house is fired, the execution of (focus emergent-event) changes the inference space to the emergent-event cluster to manipulate emergent events.

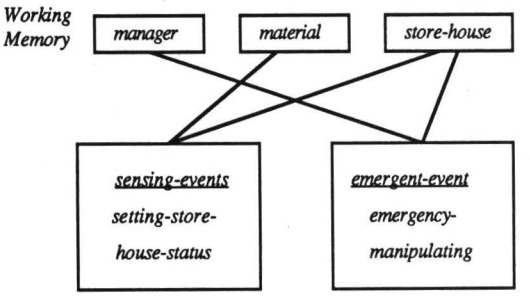

Fig. 1.Two rule clusters and the working memory they share

Furthermore, since systems like OPS5 and its various derivatives use the Rete algorithm for the pattern match function, part of the efficiency comes from sharing pattern matching test which are identical among all the rules that have the tests. However, the OPS5 Rete shares results of join tests only if the patterns are the same starting from the first one. Thus it is hard to get the benefits of shared tests. When a new working memory element is created, all the beta join nodes of satisfied tests should be modified. If later the working memory element is removed, these beta nodes should be modified all over again. It will be wasteful if very few of these rules are fired during the addition-then-deletion of the same working memory element.

Here, we allow the rules to be grouped into several collections, only rules from the current collection are allowed to match. In this way, the addition-then-deletion of the same woring memory element will not cause the beta-node modification in other clusters. This tends to improve the performance. An example is given later in section IV.

III. Distributed inference of production rules

In DPS, users at one site of a local area network can share the knowledge resident on remote sites, and production rules on different sites can interact with others according to the shared knowledge. DPS is expected to provide elegant syntax to express the distributed inference of production rules.

There are two major reasons why DPS supports the capacity of distributed computation. First, knowledge distribution can be done by users. Since a single rule is the grain-size of a predictable action sequence and knowledge distribution should be based on knowledge, researchers who worked on the exploitation of parallel production systems on parallel computers [7] have found that it is difficult to automatically distribute knowledge over processors. The distributed system approach, allowing users to distribute knowledge as they wish, can do better distribution of knowledge with the guideness of users. Hence, it exploits higher parallelism than the parallel approach on parallel computers. Second, with appropriate distribution of knowledge, systems written in DPS can solve distributed problems with two forms of cooperation between working sites: task-sharing and result-sharing [13]. Several nodes can participate in a single job and partial results between nodes can be shared to produce further results.

The distributed production systems built by DPS are expected to be divided into several autonomous subsystems, each of which resides on a single site and can proceed match-action cycles concurrently with each other. DPS

supposes only a few communications are needed among these subsystems. In order to exploit the capability of distributed inference easily, DPS supports several facilities as elegant as possible. As an example shown in Fig. 2, users can build and distribute rules on a nework which consists of sites X, Y, and Z.

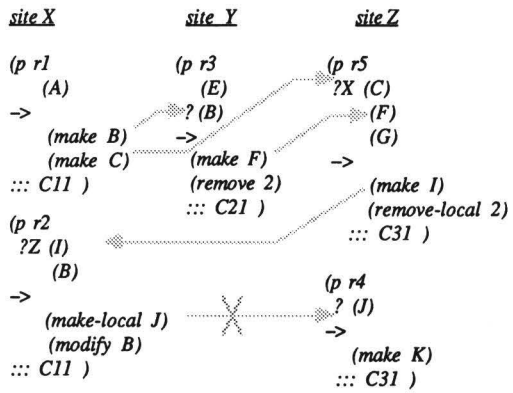

Fig. 2 Rules distributed over a network.

" ..... " means "visible to"

" ..... " means "invisible to"

The symbol "? " means the specified condition element is a remote condition element which matches against working memory on remote sites, and "? site " matches against the working memory on the specified remote site. The remove-local action is used to make the specified wme invisible to the local site; and if the removed wme was created on the same local site, then its effect is the same as that of *remove* action, i.e., that wme is invisible to all sites. The *make* action creates a new wme visible to all sites. The *make-local* action is similar to *make* action except that the visibility of the newly created wme is limited in the local site. It is used to hide a private working memory from other sites. In addition, the action part of a rule may also include a remote method invocation by sending a message to the matched remote wme. With these facilities, users can manage private and shared working memory and respond to remote situations.

IV. Implementation

In general, DPS interpreter acts in matching-action cycles like OPS5, but remote condition elements and additional tasks must be considered. The system architecture of DPS is shown in Fig.3. Working Memory (WM) collects data which production rules will match against. Visible Remote Working Memory contains data, received from remote sites, that satisfy remote condition elements of rules on the local site. Remote Condition Test is the set of condition elements extracted from remote rules which match against WM on the current site. Let's consider the example shown in Fig. 2. During the session of rule interpretation on site Y, the remote condition element *(B)* in rule *r3* is broadcast to remote sites. Site X will receive condition element *(B)* and then build it in Remote Condition Test. Whenever a wme that satisfies *(B)* is newly created on site X, it will be automatically transmitted to the Visible Remote WM on site Y. This saves communications during the execution of systems. WM Controller is responsible for maintaining the consistency of data between the WM on the local site and visible WMs on remote sites.

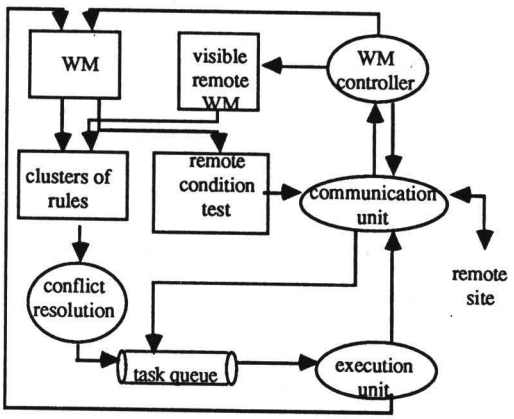

Fig. 3. system architecture of DPS on a single site

IV.1 Rule clustering

In DPS, there is an activity pointer which designates the current cluster. The initial value of the pointer is given by users at the beginning of execution. Each time when the RHS actions of a rule are executed, it may cause some variation in working memory. Since the working memory is shared by the clusters on the site, new working memory elements should be known to the noncurrent clusters. Here, we go around the problem with a simple and clear approach.

DPS gives each cluster a last-time-tag, which records the last working memory time tag for current cluster when switching to another cluster. When later this cluster is selected as the current cluster, it should match the working memory elements since the one whose time tag is immediately greater than last-time-tag. In this way, each cluster, when behaving as the current cluster, knows the variation since it switched to another cluster. Thus the addition-then-deletion of the same working memory element may affect only current cluster because noncurrent ones do not know the occurrence of addition-then-deletion event. All they can see are the elements exist in the working memory. For example, consider the two clusters in Fig.4.

```
cluster1                    cluster2

(p rule1                    (p rule5
 (B) (C)                        .....
->                          ->
 (make-local A))             (make-local D)
 (p rule2                    (make-local C))
   (C)(D)                   (p rule6
->                              ....
 (remove-local F))          ->
(p rule3                        (make-local B))
 (B)(D)                     (p rule7
->                              .....
 (make-local F))            ->
(p rule4                        (focus cluster1))
   (A)                      (p rule8
->                              (B)(A)
 ..... )                    ->
                             ..... )
```

Fig. 4. An example of addition-then-deletion of the same wme

If current cluster is cluster2 and at first rule5 is fired, (i.e. there exist working memory elements A and C that satisfies condition elements (A) and (C).) then new working memory elements D and C are added to working memory. Next, if rule6 is fired, working memory element B is added to working memory. Finally, rule7 is fired and cluster1 is selected to be the current cluster. The working memory so far is shown in Fig. 5(a).

In cluster1, rule1 and rule2 are satisfied and added to conflict set. According to the resolution strategy, rule1 is fired first and it adds element A to working memory. This causes an instantiation of rule4 to be added to conflict set. Next, rule3 is fired and element F is added to working memory. Finally, rule2 is fired, and element A is deleted. The instantiation of rule4 should be deleted from conflict set. Later when cluster2 is selected as the current cluster, all it can see is working memory element F (as in Fig. 5(b)). The addition-then-deletion of working memory element A has no effect on cluster2. This improves performance significantly when there are many identical condition elements among the clusters.

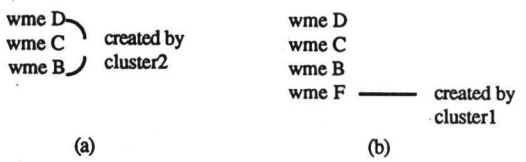

Fig. 5.(a) WM status after execution of cluster2
(b) WM status after execution of cluster1

IV.2 Distributed inference

The remote condition elements of rules are examined during the time of rule interpretation. DPS interpreter creates a special rule in the target site(s) for each remote condition element. The syntax of a special rule is:

```
(p <special-name>
  < the remote condition element>
    -->
  <send 1 <target-site>:<cluster>:<address of corresponding &mem node>
 ::: <remote-condition-test> )
```

Once the special rule is built, it can match against the working memory on remote sites. If the special rule is fired by a remote DPS system, then its RHS action is executed and the result is sent back to the local site. When received by the local DPS system, the matched data will be put into the Visible Remote WM and then copied to the specified &mem node. This makes Rete networks, which distributed over cooperating sites, form a *virtual* integrated Rete network since there exist *virtual channels* between special rules in target sites and &mem node in the local site; i.e., local rules sense changes on remote WM immediately after they occurred. For example, the rule, which contains a remote condition element, in site-A:

```
(p remote-condition-example
    (A)
    (B)
    ? site-B (C)
   --> .....)
```

results in the Rete networks shown in Fig. 6.

If a firing rule causes an execution of "make-local" ("remove-local") action, the working memory element is added to (removed from) working memory , and may trigger another inference cycle. If the action is "make", which allow a new working memory element to be visible to remote sites, then the working memory element should be matched not only by the rules in current cluster but also by the special rules in remote-condition- test cluster on local site. This seems natural because the working memory elements created by "make" actions are visible globally and may be tested by some remote rules. The "remove" action removes working memory elements visible to remote sites, so it must be implemented in a special way.

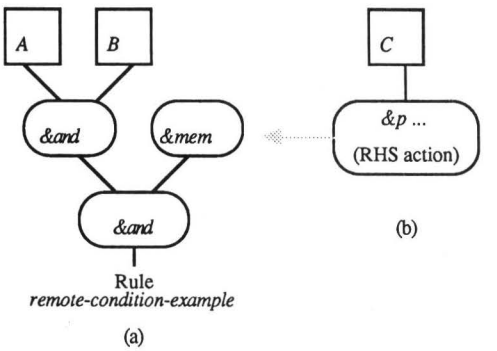

Fig. 6 (a) A cluster in site-A.
(b) The remote-condition-test cluster of site-B.

IV.3 Task scheduling

DPS system maintains a task queue on each site, which orders the tasks generated by the inference engine and the communication unit. We have four kinds of tasks. They are 1) remote-test rule building, 2) message passing, 3) remote deletion or modification of working memory element task, 4) RHS actions of selected rules.

When interpreting a rule containing remote condition elements, DPS interpreter creates a remote-test rule building task for each remote condition element and puts them in the task queue. When selected by the task scheduler, the task will be sent to the communication unit and then the remote sites, then executed by remote DPS interpreter. For the balance of DPS system, rules in the execution set resulting from conflict resolution are not evaluated immediately. They are grouped into a task, put into the task queue and rated by the task scheduler. They cannot be executed until selected by the scheduler. When the communication unit receives a request for building a special rule for remote condition element, the request is also treated as a task and must be put into the task queue. Rules in the remote-condition-test cluster are conditions received from remote DPS systems, along with a RHS action which sends the matched wme to a target site. If anyone of them is satisfied, its RHS action should be executed immediately instead of being put into the task queue, since the execution has no effect on the local working memory.

IV.4 Consistency maintenance

Let's consider the previously mentioned example in Fig. 7. After rule *r1* on *site X* has fired, it may happen that the condition parts of rule *r2* on *site X* and rule *r3* on *site Y* are both satisfied, and rule *r3* may *remove* wme *B* just before rule

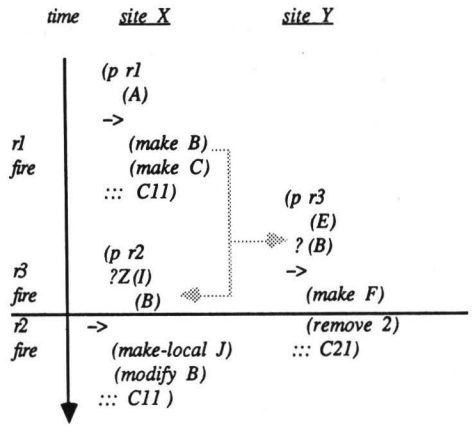

Fig. 7. Inconsistency occurs between site X and site Y

r2 fires. Here, inconsistency arises between working memory on *site X* and *site Y*. Other situations may result in similar inconsistency. DPS solves this problem by accessing the shared working memory elements (wmes) via *locks*. Working memory controller is responsible for the manipulation of locks of wmes according the following disciplines.

. Before a rule, whose condition part contains remote condition elements, would fire, DPS has to request the write locks of matched remote wmes if these wmes are to be changed by the rule, and the read locks if these wmes are not to be changed.
. Read locks are shared, and many rules on different sites may concurrently read a wme. Write locks are exclusive, and no other rules may access a locked wme while its write locks is held by a rule.
. DPS requests locks for a rule only when the LHS of this rule is satisfied and its RHS is placed in the task queue.
. A rule can fire only after the involving locks are available, and all locks held by a rule are released until the end of execution of the rule.
. Write lock requests made to read-locked wme are queued until all read locks are released.
. WM controller serves lock requests with first-come-first-serve scheme.
. When the write lock of a wme is held by a rule, WM controller refuses all lock requests made to the same wme until the lock is released (since this wme is modified or no longer exists). Hence, a rule task in the task queue is canceled if one of lock requests made by it is refused.
. Modification of a shared wme is made visible to involving remote sites.

These disciplines guarantee the consistency of visible WM between cooperating sites.

V. Performance evaluation

Since DPS leaves the work of initial knowledge distribution to users, the performance of DPS is conceivably related to the way how users distribute their knowledge on cooperating sites. According to our experiments, if an overall production system is partitioned into n appropriate subsystems, each of which excutes almost autonomously on one site with only a few communications to/from other sites, then the performance would approximate n times higher than that of the original system that executes on a single site. However, if an overall system is partitioned into subsystems in such a way that rules on each site always match remote working memory elements, and hence rules on different site can only fire sequentially, then the performance would greatly degrade, even worse than that of the original system that executes on a single site. Fig.8 illustrates speedup and degeneration for the performance of DPS, where x-axis means the ratio of the total number of satisfied remote condition elements during execution to the total number of satisfied condition elements, and n means the number of subsystems. The curve marked 'n=1' is illustrated as comparison. The values of threshold ratios r1 and r2 for curve 'n=2' and 'n=3' are closely related to communication cost. The less the communications cost, the higher the ratios are; hence, higher speedup would be achieved.

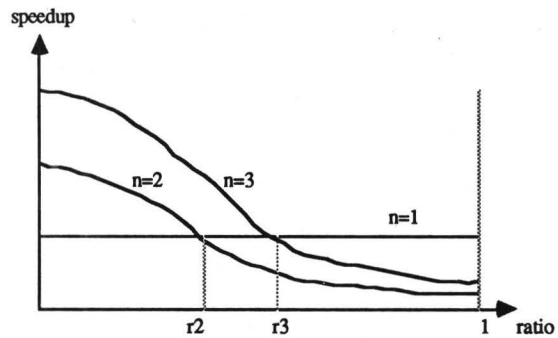

Fig. 8 performance speedup as a function of the ratio of the number of remote-matching to the number of total matching.

VI. Conclusion

In order to improve the performance of production systems, a number of reseaches focus on the parallel processing of production systems. In this paper, we propose an approach for building distributed production systems. We have attempted to show the development model used to build DPS programs and the environment that surrounds the language. One of the major objectives of DPS is to enhance performance of production systems. DPS supports elegant constructs for users to express the distributed inference of production rules. Rules on different sites work in a cooperative way and only a few communications are needed. Another objective of DPS is to propose a model for distributed inference. Since condition tests for remote condition elements are built on remote sites during interpretation session, communications needed in matching-action cycles are minimized. In addition, DPS also supports facilities to split inference engine on a local site into a number of subsets, each of which would focus on the inference about a particular event.

The prototype of DPS system is now implemented in Common Lisp on a HP LAN which connects three HP 9000/320 workstations. Due to the lack of efficient facilities to support interprocess communication, communication cost has influence on performance of this prototype. In current stage, DPS does not consider the ability of dynamic migration of knowledge. Users are required to divide the entire application into partitions, and spread them on appropriate sites on a distributed system. Inspite of this little inconvenience, DPS, with elegant constructs and a powerful distributed inference mechanism, is a useful tool for building distributed production systems.

References

[1] Buchnan,B.G. and Shortliffe, E.H., Eds, *Rule-Based Expert Systems* , Addison Wesley, Reading, MA, 1984.

[2] Ensor, J.R. and Gabbe, J.D. "Transactional Blackboard", Proc. of IJCAI, pp 340-344, Los Angles, 1985.

[3] Erman, L.D., Hayes-Roth, F., Lesserr, V.R. and Reddy, D.R., "The Hearsay-II Speech-Understanding System: Integrating knowledge to solve uncertainty", Computing Surveys, Vol.12, pp 213-253, 1980.

[4] Fickas, S. "Design Issues in a Rule-Based System", ACM SIGPLAN, Vol. 20, pp 208-215, 1985.

[5] Forgy, C., *OPS5 User's Manual* , Dep. of computer science, CMU, 1981.

[6] Forgy, C., "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem", Artificial Intelligence, Vol. 19, pp 17-37, 1982.

[7] Ishida, T. and Stolfo, S.J., "Towards the Parallel Execution of Rules in Production Systems Programs", Proc. of IEEE Internaional Conf. on Parallel Processing, pp 568-575, 1985.

[8] Lesser, V.R. and Corkill, D.D. "Functionally accurate cooperative distributed systems", IEEE Trans. on System, Man, Cybernetics, Vol. SMC-11, No.1, pp 81-96, Jan. 1981.

[9] Neiman,D. and Martin, J., "Rule-Based Programming in OPS83", AI Expert, Premier,1986, 54-65.

[10] Nii, H. P. and Aiello, N.,"AGE(attempt to generalize): A Knowledge-Based Program for Building Knowledge-Based Programs", Proc. of IJCAI, pp 645-655, Toyko, Japan, 1979.

[11] Nii, H. P., Feigenbaum, E. A., Anton, J. J. and Rockmore, A. J., "Signal-to-symbol transformation: HASP/SIAP case study", AI Magazine, Vol. 3, No. 2, pp 23-35, 1982.

[12] Schor, M. I., Daly, T.P., Lee, H. S. and Tibbitts, B.R., "Advances in Rete Pattern Matching", Proc. of AAAI, pp 226-234, Philadelphia, 1986.

[13] Smith, R.G. and Davis, R., "Frameworks for Cooperation in Distributed Problem Solving", IEEE Trans. on system, man, and cybernetics, Vol. SMC-11, No. 1, Jan. 1981.

[14] Stolfo, S.J. and Miranker, D.P., "DADO: A Parallel Processor for Expert Systems", Proc. of IEEE International Conf. on Parallel Processing, pp 74-82, 1984.

[15] Tocoro, M. and Ishikawa, Y., "An Object-Oriented Approach To Knowledge Systems", Proc. of the international Conf. on Fifth Generation Computer Systems, pp 623-631, Tokyo, Japan, 1984.

Appendix: Syntax specifications of DPS

```
system ::=
    [ remote-spec]
    object-declaration
    [ method-declaration ]
    production-rules
remote-spec ::=
    (remote-site  list-of-sitenames)
list-of-sitenames ::=
    sitename [ list-of-sitenames]
object-declaration ::=
    (object  ob-name list-of-attributes)
list-of-attributes ::=
    att-spec [ list-of-attributes ]
att-spec ::=
    attname: | (attname: default-value)
method-declaration ::=
    (method  ob-name methodname  ([ list-of-parameters ])
    (lisp code))
list-of-parameters ::=
    paraname [ list-of-parameters ]
production-rules ::=
    ( p  rulename
        LHS
    -->
        RHS
    ::: rule-attribute)
LHS ::=
    list-of-condition-elements
list-of-condition-elemetns ::=
    [ ce-var ] condition-element
    [ list-of-condition-elements ]
ce-var ::=
    < varname >
RHS ::=
    list-of-actions
list-of-actions ::=
    action [ list-of-actions ]
action ::=
    (make-optr new-wme) |
    (wm-optr matched-wme) |
    (sendmsg matched-wme message)
make-optr ::=
    make | make-local
wm-optr ::=
    remove | remove-local | modify
new-wme ::=
    number | ob-name [ list-of-attribute-values ]
matched-wme ::=
    pointer [ list-of-attribute-values ]
pointer ::=
    number | ce-var
message ::=
    msgname [ list-of-arguments ]
list-of-arguments ::=
    arg [ list-of-arguments ]
list-of-attribute-values ::=
    [ attname: ] value [ list-of-attribute-values ]
```