DEMONIZING PRODUCTION SYSTEMS

Giuliano Pacini and Franco Tunni

Dipartimento di Informatica
Universita di Pisa
Italy

ABSTRACT

The paper describes a language based on the paradigm of Production Systems. The novel aspects of the proposal are the possibility of augmenting Production Systems with demons able to monitor the visit of the search space, and the integration of Production Systems with Abstract Data Types and traditional functional programming.

1. Introduction

Traditional algorithm programming languages have shown inadequate to solve problems which naturally require a certain amount of reasoning.

Currently two approaches seem to be promising: Logic Programming /1/ and Production Systems /?.,3/. Both of them allow to organize the solution to a problem as a proof finding activity. A problem is essentially specified by two components: a collection of facts (data base, data memory) and a collection of inference rules (rewriting rules, productions). The solution is a sequence of applications of inference rules which transforms the data base in such a way that the goal is satisfied.

This paper describes a production system executor (DPSP for Demonized Production System Processor) built upon the following paradigms:

1. Extending production systems to include modules where search control and conflict resolution can be programmed.
2. Applying to production systems recent ideas developed in the realm of algorithmic languages research, especially the abstract data type approach.
3. Integrating Production Systems with a functional programming language in order to easily specify typical algorithmic tasks.

* This work was partly supported by CNR-PFI under contract N. 81.02053.97.

DPSP allows to add to a set of productions a set of demons, which watch over the problem space and control the evolution of the problem solving process.

The main objective of the design is to map the problem space into the architecture of production systems, while preserving their most-appealing aspect, i.e. their being so close to the actual nature of the problems they try to solve. As a consequence any direct annotation of productions aimed at improving the search of the solution seems doomed to betray the basic nature of the approach. The proposed solution is then to leave productions uncluttered and to add, possibly later, certainly in a separate module, demons which incorporate the search strategy.

The paper is organized as follows: section 2 discusses the basic ideas of the language. Section 3 gives a user view description of the language DPSP while section 4 provides a few examples. Conclusions are devoted to compare the authors' proposal with other solutions and to the analysis of possible developments of the project.

2. Demons and the problem space: basic ideas

A demonized production system consists of four principal components:

1. a data base
2. a set of productions <C0NDf,ACTi>
3. a goal condition G
4. a set of demons

By "problem space", associated to a production system, we mean a tree whose nodes are labelled by instances of the data base and whose arcs are labelled by productions. If a is an arc, labelled by <C0ND,ACT> , going from n1, with data base d1, to n2, with data base d2, then

COND(d1)=true and d2=ACT(d1). Informally, the problem space is a, possibly infinite, finite branching tree recording all possible evolutions of a production system given an initial state of the data base. Dy "search space" we mean a finite prefix of the problem space. At any point of the computation of a production system the search space records the part of problem space which has already been explored.

Rephrasing /4/ the process of searching is described as the repeated execution of the cycle:

1. Select a node in the search space (search strategy); select a production applicable to the corresponding data base state (conflict resolution);
2. Apply the production to the data base state; add the new state to the search space;
3. Decide if the new state is a goal state; decide to quit;

By default, DPSP implements a conflict resolution based on the declaration order- of productions and a depth first search strategy.

The first three components of a DPS constitute the classic composition of a production system /5,3/. The fourth one specifies the search strategy and the conflict resolution strategy. Demons are the entities authorized to browse into the problem space. Demons can implement conflict resolution selecting the production to apply or they can select a node of the search space, thus allowing to realize a search strategy different from depth-first. As a matter of fact, if productions are considered to be a set of inference rules, i.e. a level of reasoning, demons constitute a level of meta-reasoning, i.e. functions able to structure the deductive process.

The system has been designed and implemented to allow incremental programming in the following sense: a production system may be prepared and run without any demonization. in this case DPSP executes it using the default control strategy. The first runs can then be used to acquire knowledge about better strategies. Demons embodying more accurate strategies can then be added without disturbing the original production system and the process can obviously be iterated.

3. The language DPSP

DPSP embeds production systems into a programming language suitable for defining data types and operations on them. The programming style is functional and the specific syntax resembles LISP.
A DPS has the following components:

1. type definitions;
2. function definitions;
3. data base definition;
4. goal definition;
5. a set of productions;
6. a set of demons.

The following subsections describe each of the components separately.

3.1. Type definitions

Type definitions introduce new abstract data types into the system. As in all other languages equipped with abstract data types mechanisms /6,7,8/ type modules export a set, of operations which characterize the data type. The remaining part of the data type module contains the implementation.

In practice, since DPSP is built upon Magmalisp /9/ which is a dialect of Lisp, the internal representation is a list structure and the operations are functions operating on it.

3.2. Function Definitions

Function definitions are basically Lisp functions which implement either more complex operations on the data types or operations on the data base.

3.3. Data base definition

The data base definition is simply a collection of declarations of variables.

3.4. Productions

A production has the- form L- COND,ACT . The label L identifies uniquely the production and it is used in the triggering mechanism of demons as well as for interaction with the user, i.e. tracing, recording of the history the solution and so on.

The condition COND is a predicate on the state of the data base. The action ACT is a transformation of the data base state. More

precisely it is a set of assignments on the variables of the data base.

3.5. The search space

before discussing demons it is necessary to define the search space as it is in DPSP.

The search space is a pair: a finite tree and a node. Each node of the tree is in turn a pair: a data base instance and a set. of production names, i.e. the productions which are applicable to the data base instance of the node but which have not been yet applied.

Each arc of the tree is labelled by a production label, i.e. the production which caused the transition from the father to the son. The node associated to the tree is a handle to the tree itself and it is the "current node".

Furthermore, each node of the search tree can be associated with control variables. Control variables can be inspected ana updated only by demons and they serve to maintain state information of the node in order to implement sophisticated search strategies.

Demons can inspect the search space via operations on trees, i.e.:

```
sons : node — > list-of-nodes
father : node — > node
leaves : — > list-of-nodes
nodes : — > list-of-nodes
root?: node — > bool
prods: node —>list-of-production label
inprod: node — >production label
```

All the previous operations have the search space as an implicit parameter. If no argument is specified for the operations expecting a node the current one is assumed. Prods returns the list of still applicable productions associated to a node. Inprod returns the label of the production which caused the transition to the node.

The addition of nodes to the tree occurs on applying a production to a node. A new node is created as a son and it becomes the current one. Nodes may be dropped via command "kill". Kill returns the father of the killed node.

Demons can evaluate expressions in a particular node with the form "in < node-returning-expr> value-of <expr>".

3.6. demons

Demons are similar to productions in that they are pairs <TRIGGER,CH01CEFUNCTI0N>.

Triggers are boolean expressions. They may involve functions which inspect the search tree looking at data base states, production labels and control information. If a production label occurs in a trigger it evaluates true or false if it does or does not occur in the applicable production list of the current node.

The choice function of a demon may return either a node or a production label. In the former case the demon chooses the node to be expanded next (search strategy). In the latter one, the demon suggests a conflict resolution strategy.

A "demonization" is a list of control variable declarations and a list of demons. When the demonization is invoked by the interpreter, triggers are evaluated in a sequential order. If a trigger is verified the associated function is computed and its result is returned to the interpreter. If no trigger is verified, a default demon (see next section) is applied.

3.7. The execution cycle

The execution cycle of DPSP is the following:

a: compute the goal condition for the current node. If it is satisfied halt.
b: compute the applicable production list for the current node and bind it to the node.
c: apply the demonization. Let r be the result.
d: if r is a node, set r as the current node and repeat step c.
if r is a production label, delete the production label from the current applicable production list, apply the production, i.e. generate a son of the current node, label the arc with r, set the newly created node as the current one, and repeat step a.

The default demonization implements a conflict resolution based on the declaration order of productions and a depth first-backtracking search strategy. More precisely the default demonization is defined as:

```
(NOT (NULL PRODS)) (FIRST PRODS);
(NULL PRODS) (KILL);
```

4. Examples

The following examples aim at offering the flavor of the construction of DPSP programs.

4.1. Computing the change

The problem is to form the change for a customer paying a bill. The initial data base contains information about the bill, the amount handed by the customer and the amount of available coins. The syntax is, hopefully, self explanatory. Where comments are needed they are enclosed in curly brackets.

```
 type C01N=(CENT,NICKEL, DIME.QUARTER.DOLLAR);
{COIN is an enumerated type; the syntax is Pascal-like}
 type COUNTER = (0 .. MAXINTEGER);
 type DECMONEY = (0 .. MAXINTEGER);
 type MONEY-
   exports
     ZERO: MONEY;
     ADDC0IN(C01N,MONEY): MONEY;
     DEC(MONEY): DECMONEY;
     READ: MONEY
(READ reads values of type money as a list of integer-coin pairs}
     implementation

{the implementation section contains Lisp functions implementing the exported operations};
(the following section declares the data base}
   var CENTS,NICKELS,DIMES,
       QUARTERS,DOLLARS: C0UNTER=(READ);
   var BILL: DECM0NEY=(READ);
   var CASH: MONEY=(READ);
   var CHANGE: M0NEY=(ZER0);
{follows the goal declaration}
   goal (EQ(DEC CASH) (PLUS(DEC CHANGE)BILL))

   productions
   PI -
     (AND
         (GREATER (DIFFERENCE
                     (DEC CASH)
                     (PLUS(DEC CHANGE)BILL)
                  100)
         (GREATER DOLLARS 0))
      ((ASSIGN DOLLARS (SUB1 DOLLARS))
       (ASSIGN CHANGE (ADDCOIN CHANGE' DOLLAR)));
   P2 - . . .
{P2 etc. are similar to PI, with P2 labelling the production for quarters, P3 the production for dimes and so forth}

{A simple demonization may be added in order to maintain a comparable number of dimes and quarters}
      demonization
         (AND (NOT P1)P2 P3)
         (C0ND((GREATER DIMES QUARTERS) 'P3)
               ('TRUE 'P2))
```

4.2. A general best-first strategy

The second example is a possible demonization which realizes a best-first strategy. Suppose you have a function which evaluates the likelyhood of success given a data base state. A possible strategy is implemented by the following cycle:

1- when in a node, apply all applicable productions generating new sons;
2- select the best of the sons;
3- move to the selected son.

This strategy is implemented by the following demonization:

```
      demonization
      var ACTIVE: B00L='FALSE;
{variable ACTIVE is a control variable; when a new node is created, ACTIVE with its initial value is associated to it}
      (NOT ACTIVE) ((ASSIGN ACTIVE 'TRUE)
                    (FATHER));
{upon creation a node sets itself active and lets the father continue the generation of sons}
      (AND ACTIVE(NOT(NULL (PRODS))))
           (FIRST(PRODS));
{continuing the generation of sons}
      (AND ACTIVE(NULL PRODS)(NOT(NULL(SONS))))
           (... returns the best son ...);
```

It is worth noting that if a failure occurs, i.e. no productions are applicable in a selected node, the default demon implies a backtracking to the father and the demonization implies the selection of the second best son and so on.

5. Conclusions

Computing in a production system environment produces a sequence of search spaces. Then a possible general definition is that controlling Production Systems means limiting the possible produced sequences of search spaces. In this light it is evident that control decision must in general rely on the examination of the actual search space, in order to decide what kind of evolution the search space itself can undergo.

This approach, which is not explicitly present in other proposals /3,10,11,1/ is the

most characterizing aspect of our work. Indeed demons can operate examining the search space in order to control both conflict resolution and search strategy.

The language for describing demons is essentially a conventional functional programming language enriched by primitive operations for inspecting the search space. The choice of a conventional programming language is primarily motivated by the fact that the search space is a complex data structure and, consequently, only a full blown programming language seems to be adequate for coding the inspection process.

As it is for the logic and the control part of algorithms in /1/ the distinction between productions on one hand and demons on the other is somewhat ambiguous. Indeed, one analysis of the problem might include in productions what another analysis include in demons. For instance, in the example 4.1 the demoruzation relies only on the data base current state. As such it could be embedded in productions.

Anyway, it is rather clear what cannot be included in productions. Productions cannot include control decisions which effectively rely on inspecting the search space. This is due to the fact that conditions in productions can deal only with the current data base state. This impossibility is not definitive, because data base states could be enriched by adding components for hiding some information about search spaces. However this method can lead to modify the data base nature betraying the spirit of the solution initially given to the problem.

Currently an experimental version of DPSP is implemented as an embedded language in Magmalisp, which in turn runs under VM-CMS. This way DPSP inherits the most interesting feature of the implementation of Magmalisp, i.e. the incremental state saving mechanism based on the notion of contexts /12/.

Planned developments for the system are a larger experimentation via the construction of simple expert systems to be used in an office automation project on one hand, and the study of the possibility of enriching the language with parallel features on the other hand. Parallelism can be added at least in two different ways:

- One is to allow parallel inspection of the problem space under control of demons.
- The other one is to have concurrent communicating production systems. Indeed guarded commands used in Communicating Sequential Processes /13/ share several interesting aspects with production systems, including the nondeterministic one.

REFERENCES

/1/ Kowalski H., "Algorithm = Logic + Control", Comm. ACM Vol. 22(7) pp. 424-436 (July 1979).

/2/ Davis R. and J.J. King, "An overview of Production systems", in Machine Intelligence 8: Machine Representation of Knowledge, ed. Elcock & Michie, Wiley & Sons, New York (1977).

/3/ Forgy C. and McDermott J., "OPS, a domain-independent production system language", pp. 933-939 in Proc. Fifth Int. J. Conf. on A.I., (1977).

/4/ Newell A., "Heasoning, Problem Solving, and Decision Processes: The Problem Space as a Fundamental Category", in Attention and Performance VIII, ed. Nickerson, R, Lawrence Erlbaum Associates, Hillsdale, NJ (1980).

/5/ Nillson N., Principles of Artificial Intelligence, TIOGA Books (I960).

/6/ Wulf W.A., L.R. London, and M. Shaw, "An introduction to the construction and verification of Alphard programs", IEEE Trans. on Soft. Eng. Vol. SE-2(4) pp. 252-256 (1976).

/7/ Liskov B., A. Snyder, R. Atkinson, and C. Shaffert, "Abstraction mechanisms in CLU", Comm. ACM Vol. 21(B) pp. 564-576 (1977).

/8/ Wegner P., "Programming with ADA: an introduction by means of graduate examples", SICPLAN Notices Vol. 14(12) pp. 1-47 (1979).

/9/ Montangero C., G. Pacini, and F. Turini, "Magmalisp: a Machine Language for Artificial Intelligence", pp. 5bb-561 in Proc. 4th Int. Joint Conf on A.I., (1975).

/10/ Georgeff M., "A Framework for Control in Production systems", pp. 328-334 in Proc. IJCAI-81, (1981).

/11/ Clark K.L. and F.G. McCabe, "IC-PR0L0G-language features", in Proc. of Logic Programming Workshop, (14-16 July 1980).

/12/ Montangero C., G. Pacini, M. Simi, and F. Turini, "Information management in context trees", Acta Informatica Vol. 10 pp. 85-94 (1978).

/13/ Hoare C.A.R., "Communicating Sequential Processes", Comm. ACM Vol. 21(8) pp. 666-677 (1978).