THE UNIVERSITY OF
WARWICK

Original citation:
Alexander-Craig, I. D. (1987) SeRPenS - a production rule interpreter. University of Warwick. Department of Computer Science. (Department of Computer Science Research Report). (Unpublished) CS-RR-097

Permanent WRAP url:
http://wrap.warwick.ac.uk/60793

Copyright and reuse:
The Warwick Research Archive Portal (WRAP) makes this work by researchers of the University of Warwick available open access under the following conditions. Copyright © and all moral rights to the version of the paper presented here belong to the individual author(s) and/or other copyright owners. To the extent reasonable and practicable the material made available in WRAP has been checked for eligibility before being made available.

Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge. Provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way.

A note on versions:
The version presented in WRAP is the published version or, version of record, and may be cited as it appears here.For more information, please contact the WRAP Team at:
publications@warwick.ac.uk

warwickpublicationswrap
highlight your research

http://wrap.warwick.ac.uk/

Research report 97

SeRPenS - A PRODUCTION RULE INTERPRETER

Iain D Craig

(RR97)

Abstract

This paper describes the SeRPenS production rule interpreter. SeRPenS permits the description of the coded rule interpreter in the form of executable production rules. The interpreter description can be used to interpret the basic match-deliberate-act cycle in inspectable form, permitting a limited degree of introspection. The paper also describes ways in which planning can be incorporated. Although the system appears to have considerable power, it is limited by the fact that there is no general theory of action within which to describe the activity of the system. The limitations of SeRPenS are outlined.

Department of Computer Science
University of Warwick
Coventry
CV4 7AL, UK

March 1987

SeRPenS - A PRODUCTION RULE INTERPRETER

Iain D. Craig

Department of Computer Science,
University of Warwick,
Coventry CV4 7AL, UK.

ABSTRACT

This paper describes the SeRPenS production rule interpreter. SeRPenS permits the description of the coded rule interpreter in the form of executable production rules. The interpreter description can be used to interpret the basic match-deliberate-act cycle in inspectable form, permitting a limited degree of introspection. The paper also describes ways in which planning can be incorporated. Although the system appears to have considerable power, it is limited by the fact that there is no general theory of action within which to describe the activity of the system. The limitations of SeRPenS are outlined.

March 11, 1987

SeRPenS - A PRODUCTION RULE INTERPRETER

Iain D. Craig

Department of Computer Science,
University of Warwick,
Coventry CV4 7AL, UK.

1. Introduction

This paper describes and motivates SeRPenS[1], a production rule interpreter. The SeRPenS interpreter encourages users to combine meta- and object-rules (MRules and ORules, respectively) in an application. The rules in the system have the conventional forward chaining interpretation: when the antecedent is completely satisfied, the consequent is executed to cause changes in working memory. As is usual in forward chaining interpreters, a data structure called the Conflict Set (CS) is maintained on every cycle. The conflict set is used to hold the current set of rule instantiations so that one or more rules may be selected to fire causing the events on that cycle. MRules in SeRPenS are used to select appropriate ORules for firing and can operate on production memory as well as the conflict set. By operating on production memory, MRules can pre-select ORules for matching: this allows SeRPenS to behave in a *content-directed* fashion, similar to that described by Davis (Davis, 1980).]

SeRPenS is not, however, a conventional rule interpreter with meta-rules included more or less as an after thought. MRules are a fundamental component of the system and provide its more interesting features. It is a fundamental goal of the design of the interpreter that MRules be able to interpret all other rules. This kind of meta-level control has been examined by others (e.g., (van Harmelen, 1986) and (Reichgelt, 1987)). Other interpreters of this kind have not attempted to reproduce the interpreter as a sequence of productions and have not had, it would appear, the ability to interpret MRules at the meta-level. SeRPens, on the other hand, is designed explicitly to provide such extended interpretation. The result is an infinite tower of rule interpreters similar to that described by Smith (Smith, 1982) for 3-LISP, each with the ability to interpret the level below.

This paper is concerned with describing and motivating some of the more unusual features of SeRPenS. It is organised as follows. In the next section, the structure of the intepreter is presented. Section three is concerned with the role of MRules in the system and presents some variations on the basic interpretation MRules. Section four is concerned with a higher level discussion of the limitations of the SeRPenS interpreter and approach. The main limitations are felt to be the inadequacy of reasoning about action and the impoverished structure of the production representation.

2. The Interpreter

SeRPenS is written as a small number of Prolog clauses. It is so small that it could be implemented in a matter of hours. This simplicity is a deliberate design goal: because the interpreter is small and very simple, it can be represented with ease in the production rule formalism which it interprets. Rules have the conventional <condition,action> pair structure and are interpreted in an antecedent-driven manner. Condition clauses and actions are represented as Prolog clauses: this simplifies the structure of the interpreter without loss of generality and provides the more conventional user with the ability to interface SeRPenS to other Prolog systems and structures. This representation allows the rule matcher to be called from MRules with no complications.

The interpreter works by matching all rule conditions. Those rules whose antecedent has matched are placed in the *conflict set*. The conflict set performs the same function in SeRPenS as it does in, e.g., OPS5: it holds all instantiated rules and acts as the database upon which conflict resolution strategies operate.

[1] SeRPenS stands for Self-Reflective Production System.

SeRPenS contains no built-in conflict resolution strategies: much of the work in conflict resolution is performed by MRules, as will be seen. When the conflict set has been created, one or more rules are selected for firing by a conflict resolution strategy. The execution of these rules causes the contents of working memory to be altered and the main cycle repeats. Termination is caused by explicit termination rules which check for the presence of items in working memory.

In addition to maintaining the conflict set and managing the basic match-deliberate-act cycle, the interpreter places meta-information in working memory. The meta-information generated by the interpreter is concerned with rule matching. For example, while matching the antecedent of a rule, records of which clauses have been matched are added to memory. Each record is tagged with the number of the interpreter cycle so that no confusion can occur on subsequent cycles when the same clause is matched again. Also, when a rule's antecedent has been successfully matched, a record is added to working memory stating that the rule is eligible to enter the conflict set. Similar information is posted when the action of rule is being executed. The final class of information posted by the Prolog interpreter is a record of which rules it has selected to fire on each cycle.

The Prolog interpreter makes no use at all of this information. It is made available for other uses, in particular for use by MRules concerned with implementing CR strategies which make no use of the conflict set (that is, by MRules which perform rule selection on a more explicit basis by using local bindings to hold rule instantiations). It can also be used to speed up the process of antecedent matching. Firing records are used to determine the reliability of rules in given situations: if the interpreter is always presented with similar problems to solve, the firing counters can be used as a basis for pre-selecting rules which are likely to be useful to the solution process (this becomes particularly useful when the rulebase contains a significant number of rules).

SeRPenS makes no distinction between O and MRules. This is, again, a deliberate decision aimed at making the interpreter as simple as possible. Although the interpreter does not make such a distinction by default, it can be made to do so and a default rule precedence can be set up by adding appropriate elements to working memory and by adding a CR routine to enforce the prioritisation policy. In the default case, the interpreter will match all rules and put applicable instantiations into the conflict set. The distinction between O and MRules is based entirely upon rule contents. MRules contain elements which manipulate rules and system structures (firing records and the conflict set, for example): ORules deal only with problem specific entities[1].

MRules contain primitives for inspecting rules and for executing rule actions. They are also permitted to call the interpreter's matcher directly. In order to support these interpretive functions, set operations are also supplied. Set operations can be applied to triggered rules so that instantiations can be selected by filtering. The inspection primitives can operate either on rules in production memory or on instantiated rules held either in the conflict set or in MRule local bindings: this allows sets of rules mentioning certain objects or relations or with certain actions to be selected for execution. When MRules execute other rules, they may select the order in which the rules execute (which amounts to ordering the rule instantiations) or they may excute them in arbitrary order. It is also possible to execute rules using a combination of these two mechanisms. For tracing purposes, MRules can record the order in which they have executed rule instantiations.

Since MRules have access to those interpreter components relevant to rule interpretation, they can interpret all rules in the system *including themselves:* this is the most important point about the interpreter and is discussed more fully in the next section.

3. Meta-Rules

This section is concerned with exploring some of the ways in which MRules can be used in SeRPenS. Attention is focussed on the ways in which MRules can interpret the entire production system architecture. In particular, a set of basic interpretation MRules is presented and discussed. This section also addresses the problem identified by van Harmelen that the interpreters at the meta- and object-levels should be different because they are expected to perform different kinds of reasoning task: it will be argued that

[1] This is not the entire story, but there is insufficient room here to go into the matter in greater detail.

different interpreters may be incorporated in MRule sets in SeRPenS.

In a system like SeRPenS, it is possible to execute the interpreter in one of three ways:

(i) execute MRules before ORules (essentially descending to the object level as and when necessary);

(ii) execute ORules before MRules (reflection occurs as a result of an explicit request on the part of an ORule), or

(iii) mix the execution of M and ORules in an arbitrary fashion.

For the remainder of this section, it will be assumed that the system enters with the interpreter selecting MRules before ORules. This assumption firmly places the locus of control within the set of MRules.

Initially, it can be assumed that one or more MRules is responsible for the interpretation of all other rules in the system. In effect, this assumption permits interpretive MRules to interpret *all* other rules (including MRules). Since M and ORules have exactly the same structure and are not tagged in any way, the basic control rules can be stated immediately. The first rule is concerned with interpreting the base cycle of the Prolog interpreter; the second is a termination rule.

MR1: **if** rules(R) & matches(R) & conflictSet(CS) **then** execute(CS).

This rule fetches all rules in the system (including, note, itself), matches them against working memory to create a conflict set and then executes all rules in the conflict set.

The second rule, MR2, is in fact a rule schema. The antecedent is specified in meta-syntax as a termination condition. Termination conditions will clearly vary with application.

MR2: **if** <*termination-condition*> **then** stop.

The action *stop* is a system-provided action which halts the execution of the basic interpreter. In the basic scheme of things, MR1 is repeatedly executed because MR2's antecedent is not satisfied until the execution of the system as a whole is complete and some goal state has been achieved.

The interesting point about the first of these rules, MR1, is that it is capable of interpreting all rules in the system *including itself.* This, of course, has some interesting and unfortunate consequences.One potentially unfortunate consequence is that it is possible for pathological cases to appear. The most obvious pathological case is that MR1 repeatedly executes itself causing an infinite number of reflective cycles to be built. The repeated execution of MR1 by itself also has the consequence that the rule instantiations in the conflict set may be executed an arbitrary number of times for it is possible that an instance of MR1 appears in the conflict set more than once on any cycle. It is, of course, possible to arrange for MR1 never to execute an instance of itself.

The interesting case becomes clear from considering the possibility that MR1 may engage in some filtering of the rules which it attempts to match. For each instance of MR1, it is possible to have a different set of fireable rules. When the control instance of MR1 executes its subordinate instances, rulesets with different properties will be executed: one clear way to do this is to permit second-order structures to appear in working memory elements and in rule antecedents.

The most interesting property of MR1 is that it interprets arbitrary MRules, and each MRule may interpret other rules. For example, the MRule MR3 prefilters the rules it finds in production memory to remove all rules which mention any of the meta-level operators:

MR3: **if** rules(R) & not(mentions-set(R,{meta-operator}),R1) & match(R1) & conflictSet(CS) **then** execute(CS),

where {meta-operator} is intended to represent the set of meta-level operators present in the system. This rule executes only those ORules which have matched the contents of working memory: it does not execute any MRules. MR3 can be interpreted by MR1 as an ordinary rule.

The point to note about these rules is, however, that they are quite unprincipled in their actions. They cause iterations of the basic cycle until some termination condition is satisfied. What they do not do is explore the search space in any particularly directed manner. It is assumed, one can argue, that ORules and other MRules have control of the search process: knowledge is applied to direct search in a fairly conventional manner.

One of the characterisitics of intelligent behaviour is the ability to function in a goal-directed manner. The interpreter provided by MR1 and MR2 does not respond to goals in any way, but merely executes all of the rules in production memory which have matched working memory contents. Goals can be added to the interpreter by the modification of MR1 to MR1':

```
MR1':if workingOnGoal(G) & rules(R) & mentions(R,G,action) & matches(R) & conflictSet(CS) then
        executeAction(CS).
```

This interpreter will select all those rules whose actions mention the goal for execution. In other words, all rules which conclude with this goal will be executed, thus satisfying the goal G. When running with this interpreter, there have to be rules which set goals for satisfaction and remove them from working memory once they have been satisfied. Such a rule would have the general form:

```
        if <some-state-in-working-memory> then setGoal(G).
```

It responds to a working memory state by posting a goal to be satisfied. What the working memory state is depends upon what is to be achieved: the state might be defined by the presence of certain meta-level assertions.

So far, all interpretation rules have been presented independently. It is clear from the remarks made above that it is possible for a system to have many such rules, each applicable in different circumstances. This point directly addresses the point made by van Harmelen: in a system like SeRPenS, it is possible to maintain a set of rule interpreters each suited to a particular task and providing a particular set of interpreter services. The uniform interpreter (such as that provided by MRule MR1) permits any rule at any level to be matched and executed. Since the default interpreter is very weak and very general (providing little more than a basic control loop), stronger committments must be made elsewhere. Different interpreters can be made to respond to changes in the system state at different levels and at different times: it is, indeed, possible to allow more than one interpreter to be active at any one time -- this amounts to an implementation decision based on the requirements of the particular problem or task domain.

4. Limitations

SeRPenS has been shown to permit different rule interpretation mechanisms to be present in the system at the same time. SeRPenS also contains a rudimentary model of its own interpretive processes. The inclusion of MRules partially converts the procedural representation of production rules into a declarative representation by supporting inspection. These properties of the interpreter are certainly interesting and suggest that systems like SeRPenS have considerable amounts of power at their disposal. This power is coupled with a causal connection between the interpreter rules and the state of the interpreter: the interpreter executes rules which place items in working memory which determine the set of rules which can fire on the next cycle. This connection, it has to be admitted, is very weak and comparatively unstructured.

The major limitation, aside from representational problems, is the absence of a theory of action. In particular, it would be useful to be able to reason about the potential consequences of executing an action. The production model of problem solving states that the only way in which consequences can be derived is by executing the action and examining the state of working memory. This seems unsatisfactory in the present context. It is unsatisfactory because the meta-level features in SeRPenS are designed to permit the system to reason about its own actions. Although it is not possible to foresee the effects of all actions taken by an agent, it is possible to make an attempt to predict some of them. With the production architecture, it is possible, using meta-rules, to determine those rules which are potentially fireable after a change in working memory state. However, this seems to be a very limited view to take: it is too concerned, that is, with the architecture and not with the general problem of determining what the next action to be taken should be.

It was shown above how planning operators could be added to SeRPenS. This amounts to the addition of MRules which post and remove goals held in working memory. By introducing planning operators, it becomes possible to restrict the possible actions to be taken by the problem solver at any point. It is also possible to give a reason for why a particular action was chosen. The reason can be derived by examining the state which was current when the planning decision was made and the operator selection finally came into force.

Since SeRPenS is able to represent the planning process explicitly, it seems to be the case that the planning process can itelf be reasoned about: for example, if there is more than one operator to apply at any point, the system can reflect to the meta-level in an attempt to determine the best operator to apply. This, however, leads back to the original point: without an adequate account of action, there is no *general* way in which such decisions can be taken. Certainly, it is the case that domain-specific principles or heuristics can be employed to decide what to do next. These remarks should, of course, be taken in the context of a problem solving architecture which contains a theory and a model of its own basic behaviour.

5. Conclusions

This paper has presented SeRPenS, an introspective production rule interpreter. SeRPenS provides facilities for the use of meta-rules as the interpreters of other rules in the system. It also provides facilities to support the interpretation of rules which completely describe the operations of the Prolog interpreter. One strange fact which was discovered early on (January, 1984) was that only two levels of structure are required: the meta-level is able to interpret itself *as well as* the object level. This point gives added evidence to the hypothesis presented in (Smith, 1982) and in (Batali, 1983) that only two levels are required to support introspective behaviour.

The limitations of SeRPenS are quite clear, even though it has not been used extensively. SeRPenS is an unstructured system within which it is possible to do practically anything in a more or less principled way. Even when remaining in the production rule paradigm, the representational constraints imposed by the representation get in the way. In particular, the concept of a theory cannot be adequately represented in rules: the execution theory in SeRPenS is distributed across a number of MRules.

There is a deeper reason for criticising SeRPenS. This criticism comes at a different level: the description of actions in the system. As yet, there is no generally applicable theory upon which to base reasoning about actions, although some ideas were presented in section four.

SeRPenS satisfies one goal in the construction of introspective problem solving systems: its interpreter is sufficiently simple to permit its direct translation into the formalism it interprets.

REFERENCES

(Batali, 1983) J. Batali. Computational Introspection. AI Memo No. 701, AI Laboratory, MIT, 1983.

(Davis, 1980) R. Davis. Meta-Rules: Reasoning about Control. *AI Journal*, Vol. 15, pp. 179-222, 1980.

(Smith, 1982) B. Smith. Reflection and Semantics in a Procedural Language. Technical Report No. 272, AI Laboratory, MIT, 1982.

(van Harmelen, 1987) F. van Harmelen. A categorisation of meta-level architectures. Dept. of AI, Edinburgh University, 1987.