The Automated Analysis of Rule-based Systems, Based on their Procedural Semantics.

Rick Evertsz
Scientia Ltd,
150 Brompton Road,
London, SW3 1HX,
United Kingdom.

Abstract

This paper describes a method of analysing rule-based systems, which models the *procedural semantics* of such languages. Through a process of 'abstract interpretation', the program, AbsPS, derives a description of the mapping between a rule base's inputs and outputs. In contrast to earlier approaches, AbsPS can analyse the effects of: conflict resolution, closed-world negation and the retraction of facts. This considerably reduces the size of the search space because, in the abstract domain, AbsPS takes advantage of the very same control information which guides the inference engine in the concrete domain. AbsPS can detect redundancies which would be missed if the procedural semantics were ignored. Furthermore, the abstract description of a rule base's input-output mapping can be used to prove that the rule base meets its specification.

1. Introduction

Much effort has been devoted to the problems of analysing forward-chaining rule-based systems with regard to improving their reliability and efficiency [cf. Suwa *et* al., 1982; Nguyen *et* al., 1985; Beauvieux, 1990]. To the extent that a rule-based system is a piece of software, it too can embody unintentional errors. Some of these errors merely reduce the efficiency of the rule base, whilst others may result in erroneous inferences. An example of the latter is the problem of contradictory rule subsets where, given an initial set of facts, two (possibly intersecting) groups of rules lead to contradictory conclusions (e.g. P A -IP). Examples of efficiency-reducing features include:

- redundant rules - where one or more rules are equivalent in terms of the states which match their antecedents, and the conclusions drawn in their consequents;
- subsumed rules - where a rule's antecedent matches a subset of those states matched by another, and the two rules have equivalent consequents;
- unreachable rules - where none of the initial stales can ever lead to the invocation of a particular rule (this can reduce efficiency by wasting computauonal effort on processing the rule's antecedent);
- dead-end rules - these are rules whose consequents do not affect the other rules in the system, and so have no part to play in generating a solution.

The program, CHECK [Nguyen *et al.,* 1985], tackles these problems by examining the antecedents of pairs of rules, but can also apply a more complex analysis based on the dependencies between the consequents and antecedents of rules. For example, if one of the clauses in the antecedent of a rule is not matched by any of the consequent clauses of the rule base, nor by any of the facts entered by the user, then it can never be instantiated. Though this pairwise analysis of rules is 'incomplete' in the sense that it cannot find *all* inconsistencies and redundancies, it is still of practical use - the very restrictions which preclude completeness greatly improve the tractability of the analysis. Another approach, embodied in KB-reducer [Ginsberg, 1988], performs a full analysis of the knowledge base using ATMS techniques - this method is 'complete'. In the worst case, this is computationally intractable, however, experience with typical knowledge bases suggests that this docs not happen in practice.

To date, those approaches which are 'complete' have embodied a number of assumptions which preclude their use in an important class of forward-chaining production system interpreters (or at least limit their effectiveness). They all assume monotonicity, i.e. the addition of new facts to the knowledge base does not invalidate previously deduced conclusions, and assertions can never be retracted. Secondly, the inference engine does not perform any conflict resolution - in KB-Reducer for example, each instantiated rule is fired exactly once, with no rules taking precedence over others [cf. also: Rousset, 1988; Meseguer, 1990]. CHECK too does not reason about the role played by conflict resolution.

The work described thusfar models the 'declarative semantics' of their respective inference engines. The trouble is that for many rule-based systems, the declarative semantics of the language are not equivalent to the procedural semantics. In order to reason effectively about knowledge bases expressed in such languages, one must accurately model their procedural semantics. For example, though one can spot some unreachable rules by virtue of the fact that no consequent clauses match the unreachable rule's antecedent, there may be rules whose antecedent clauses are matched, but which can never fire by virtue of the conflict resolution strategy employed. A complete analysis of such rule bases is only possible if one considers the inference engine's *control strategy.* In figure 1, if the conflict resolution strategy is based on rule specificity, then an analysis which is only based on the declarative semantics of the language would miss the fact that the rule Rl may not get a chance to fire when the assertion NO-WINGS(x) is present. If, as a result of conflict resolution, other rules in the rule base take over once R2 has fired, then Rl will be an unreachable rule for that class of inputs.

```
R1:    if BIRD(x) then FLIES(x).
R2:    if BIRD(x) & NO-WINGS(x) then RUNS(x).
```

Figure 1 - Specificity.

This paper describes a general-purpose method of reasoning about non-monotonic, forward-chaining rule bases which rely on *deterministic* conflict resolution for flow of control. This class of language is important, encompassing as it does a large proportion of the forward-chaining rule interpreters in use today (e.g. OPS5, ART, KEE). Our approach is based on the notion of 'abstract interpretation', where computation is performed in an abstract domain rather than the normal concrete one. Our implementation, AbsPS, operates on abstract data (i.e. data containing uninstantiated variables) and thereby accurately models the behaviour of the rule base on the set of concrete data subsumed by the abstract domain. An exhaustive search of the rule base, using abstract data, yields a complete and sound description of the space of possible behaviours of that rule base, provided that there arc no cycles.

We have used AbsPS to find examples of redundant rules, dead-end rules and unreachable rules which would have been missed had AbsPS not reasoned about the role of conflict resolution and the effects of retracted assertions. More importantly, AbsPS has been used to prove that a rule base meets its specification. This is accomplished by comparing the results of AbsPS' analysis with a formal specification of the mapping between the rule base's inputs and outputs. In the final section of this paper, we outline further uses to which AbsPS can be put.

2. The Abstract Interpretation of Production Systems

A rule base plus its inference engine can be viewed as a partial function. The 'domain' of this function is the set of possible inputs which the rule base is defined to accept. The 'range' is the set of final databases which the rule base can generate from the input domain. In order to characterise this partial function we have developed a method which, given an abstract description of the input domain, generates a description of the set of final databases which can be generated, and refer to this as the rule base's 'I/O mapping' (input-output mapping). In generating the I/O mapping, a full analysis is made of all possible routes through the rule base, and this enables us to identify, for example, rules which do not contribute to the output of the rule base.

AbsPS was originally developed as part of PG, a program which takes pairs of rule bases and generates inputs which discriminate between them. The algorithm is described in detail in Evertsz [1990] and is complete for rule bases which do not contain cycles. AbsPS is given a rule base and an abstract specification of the set of inputs which the rule base is designed to handle. This specification differs from normal initial fact bases in that *ground* elements which can vary are replaced with uninstantiated variables. Each variable is associated with a domain description which represents the set of permissible values for that variable. Figure 2 shows an example of an input specification together with two concrete instances, one of which satisfies the specification, the other of which does not because the integer, 27, is out of range.

```
Input Specification:
  GOAL(ASSESS_WEIGHT(x)), HEIGHT(x.h).
  c:PRIMATE(x) A c:INTEGER(h) A h>50 A h<100.
Positive Instance:
  GOAL(ASSESS_WEIGHT(CHIMP)),
  HEIGHT(CHIMP.76).
Negative Instance:
  GOAL(ASSESS_WEIGHT(MANDRILL)),
  HEIGHT(MANDRILL,27).
```

Figure 2 - An input specification.

The input specification contains two facts (predicates: GOAL and HEIGHT); in addition there is a set of constraints on the variables which define the variables' domains (in these examples constraints are prefixed with "c:" or are either ">" or "<").

Because Working Memory (WM) contains abstract descriptions of facts, the rules are matched using unification rather than the one-way pattern matching of normal forward-chaining PSs. The actions of AbsPS mirror those of a PS working on concrete data, however, all paths are explored. As each rule is unified and executed down a given path, new constraints on the variables are generated as a side-effect of unification. This environment is carried down the path from cycle to cycle and is *local* to that path. When a path is exhausted, the local environment constitutes a description of the set of final databases which can be generated down that path. For example, after applying the rule base to the input specification of figure 2, the description in figure 3 might have been produced.

```
Output Description:
  WEIGHT(x,h/2),
  where c:PRIMATE(x) A c:INTEGER(h) A h>75 A h<85.
```

Figure 3 - One possible final database.

The description of h's domain has been refined as a result of the rules applied down this path. Figure 4 shows a single rule which would have this effect.

```
if   GOAL(ASSESS_WEIGHT(x)) & HEIGHT(x,h)
   & c:PRIMATE(x) & c:INTEGER(h) & h>75 & h<85
then WEIGHT(x,h/2) & retract(GOAL(ASSESS_WEIGHT(x))
   & retract(HEIGHT(x,h)).
```

Figure 4 - The rule R3.

The process of unifying the antecedent of this rule with WM generates the extra constraints on h. Note that because AbsPS maintains an abstract description of WM, it is also able to model the effects of retraction.

During abstract interpretation, invalid paths are pruned from the search space. For example, a rule whose antecedent contains the constraint h=75 would not be instantiated down a path in which the rule, R3, has already fired. This is because the firing of R3 constrains the value of h, in that local environment, to be between 75 and 85 exclusive. Such pruning of instantiations, whose domain variables do not intersect with the environment collected down the current path, drastically reduces the size of the search space, and is vital if one is to reduce the combinatorial explosion which can result when executing a rule base on abstract rather than concrete data.

The scheme described thusfar does not handle conflict resolution and so will generate many invalid I/O mappings. The paths which lead to these invalid mappings can be pruned early on by considering the effects of conflict resolution. Ignoring conflict resolution not only generates invalid I/O mappings, it also greatly increases the size of the search space because the rule base normally *relies* on conflict resolution for flow of control. We now describe a general method of characterising the role played by conflict resolution.

2.1. Abstract Conflict Resolution

An inference engine which employs conflict resolution will have an ordered set of conflict resolution principles which are applied one at a time until the conflicting set of instantiations is reduced to a singleton. These same principles can be used to filter the set of abstract instantiations generated on each cycle. However, during abstract interpretation it does not suffice to choose one instantiation for expansion and follow that path only; this is because each instantiation is likely to cover different subsets of the abstract domain. AbsPS must characterise these subsets and expand those instantiations with non-empty domains.

The key to characterising these domain subsets lies in generating *exclusion clauses* which describe the conditions under which those instantiations which appear to lose out during conflict resolution would actually be able to fire. Given a winning instantiation of the rule Rj, the other instantiations can only fire if R_i is not instantiated - i.e. if one of the constraints generated during the process of unifying R_i with WM is violated. If the constraints on Rj are $(C_1 \wedge ... \wedge C_n)$ and those on a losing rule, R_j, are $(C_1 \wedge C_2 \wedge C_4)$, then the exclusion clause which characterises the conditions under which it could not be instantiated is as shown in figure 5.

$$\neg((C_1 \wedge ... \wedge C_n) - (C_1 \wedge C_2 \wedge C_4))$$

gives: $\neg(C_3 \wedge C_5 \wedge ... \wedge C_n)$

which is: $\neg C_3 \vee \neg C_5 \vee ... \vee \neg C_n$

Figure 5 - Generating an exclusion clause.

This exclusion clause is added to the environments of the remaining instantiations. For some instantiations this will lead to a contradiction - such instantiations are discarded from the analysis because, down this path, there are no concrete instances which would enable them to be instantiated when Rj is not. The process of generating the exclusion clause is repeated on the remaining instantiations until no instantiations remain. Each instantiation, together with its associated exclusion clause, forms a new path emanating from the current state.

The simple example in figure 6 illustrates how conflict resolution should be handled in an abstract domain. If the inference engine incorporates a preference for rules which are more specific, then in the concrete domain R5 will be preferred in situations where both it and R4 are instantiated. However, there will be instances of x which enable R4 to fire, because R5 is not satisfied by that value of x. The exclusion clause, generated by negating the extra constraint in R5 describes this set of instances: xe {PENGUIN,OSTRICH}.

R4:	if BIRD(x) then FLIES(x).
R5:	if BIRD(x) & xe {PENGUIN,OSTRICH} then WALKS(x).

Figure 6 - R5 is more specific than R4.

The environment carried down the path emanating from R4 would include this extra constraint; as a result any rules which refer to the fact BIRD(x) would not be instantiated if they include constraints which exclude all of the members of x's domain. For example, the rule R6 (figure 7) when unified with BIRD(x), would fail to be instantiated because this would conflict with x's current environment, xe (PENGUIN,OSTRICH}.

R6: if BIRD(PENGUIN) then SWTMS(PENGUIN).

Figure 7 - Cannot be instantiated after R4.

We now conclude this section on the modelling of the procedural semantics of rule bases, by describing the special handling of negation.

2.2. Negation and the Closed-world Assumption

The class of inference engine which allows the retraction of assertions and employs conflict resolution, normally embodies another non-monotonic feature: the 'closed-world assumption' - if a fact is not known to be true, then it is assumed to be false. Current approaches to the analysis of rule bases do not incorporate this assumption - an assertion is only false if it is explicitly negated. In languages such as OPS5, a rule containing a negated antecedent clause can only be instantiated if there are no matching WM elements. These semantics are modelled in the abstract domain by building what is in effect an exclusion clause for negated elements; this specifies the *minimal* conditions which have to be satisfied for the negated clause to *fail* to unify with any elements in WM. This would be u=a for R7 unified with the WM shown in figure 8.

R7:	if PARENT(x.y) & PARENT(y,z) & -MALE(x) then GRANDMOTHER(x,z).
WM:	PARENT(u,v), PARENT(v,w), MALE(a).

Exclusion Clause: u=a.

Figure 8-Building an exclusion clause.

2.3. Maintaining Environmental Consistency

During abstract interpretation, many paths are excluded because their environments are inconsistent. An inconsistent environment can be found during unification, when generating an exclusion clause and when dealing with negated patterns. Some of these environments are rejected by the unification algorithm even before they are generated, but the others need further analysis. The goal of this analysis is to spot environments which contain a contradiction - such environments can never be satisfied by any combination of concrete domain values and so should be discarded so as to avoid wasted computational effort.

This goal suggests that a refutation-based theorem prover would be well suited to the job. AbsPS incorporates a resolution theorem prover because this is one type of refutation system. It also employs the set of support strategy (this divides the current set of clauses into those that derive from the negated theorem, termed the 'set of support', and those that do not). The set of support strategy requires that every resolution involve at least one clause from the set of support, and thereby improves performance by restricting the set of potentially resolvable clause pairs. Now, if we know that some subset of the negated theorem is true, then that subset can be added to the set of axioms. This reduction, in the number of clauses in the negated theorem, further improves the beneficial effects of the set of support strategy. The incremental nature of PCs theorem-proving tasks, allows it to take advantage of just such a negated-theorem-dividing strategy. As each Abstract Instantiation fires, the set of constraints either gets larger, or stays the same size. So, if the set of constraints is currently, C_i, then on the next cycle, j, it will be Cj, where Cj = C_i u Cnew (Cnew being the new constraints, local to the fired instantiation). On the face of it, Cj should form the negated theorem, to be passed to the theorem prover. However, we already know that C_i is consistent, as it was checked on the previous cycle. Thus, the clauses in Ci can be viewed as axioms. The negated theorem need only consist of the clauses in C_{new}; those in C_i can legitimately be added to the set of axioms.

This augmentation, of the algorithm is only worthwhile if one is using a theorem prover which incorporates the set of support refinement (or at least a similar division between axioms and potentially false clause sets).

2.4. Summary

In practice, the symbolic constraints on the domains of variables, generated as a side effect of modelling the procedural semantics of the language, yield a large reduction in the number of paths which must be explored. This is because the rule base will inevitably have been designed with this procedural model in mind; i.e. it uses the procedural semantics to *control* the deductive process, and AbsPS makes use of this very same control information to guide its analysis of the rule base.

3. Applications of Abstract Interpretation

Abstract interpretation is not needed to identify 'redundant rules' - the pairwise comparison of Nguyen *et al.* [1985] suffices even for languages with differing procedural and declarative semantics. Subsumed rules can be detected without performing abstract interpretation, however, they cannot be removed without further analysis. Many unreachable and dead-end rules can only be detected on the basis of the procedural semantics of the language.

3.1. Rule Base Redundancy

Perhaps surprisingly, 'subsumed rules' cannot be removed without doing full abstract interpretation. This is because of the role of conflict resolution. The more specific (i.e. subsumed) rule, though apparently redundant, may only be there to ensure that its behaviour is executed in cases where another more specific rule might have seized control from the *subsuming* rule. Therefore, it cannot be dispensed with without altering the behaviour of the rule base. AbsPS correctly identifies such instances, only flagging subsumed rules which are truly redundant (i.e. their increased specificity has no effect on the choice of path during conflict resolution).

AbsPS is also able to locate 'unreachable rules'. These are rules which did not fire during the abstract interpretation of the rule base. Some of these unreachable rules would have been missed by systems which ignore conflict resolution, and can be very hard to detect by eye because of the subtle interactions between these rules, conflict resolution and the other rules in the conflict set.

If some subset of the rule base is only involved in paths which do not contribute to a solution, then all of the rules in that subset can be safely removed. Again, there arc many instances where such dead-end subsets can only be detected by considering conflict resolution, because an analysis based only on the declarative dependencies between rules will erroneously conclude that the subset contributes to a solution.

AbsPS does not automatically remove redundant rules but highlights the problems for the user to inspect. This is because the result may be a side effect of a 'bug' in the rule base. Conflict resolution can introduce subtle bugs which are hard for the user to detect. To illustrate, a rule R_i may never get a chance to fire because, for all possible initial fact bases, some other rule Rj always takes precedence. This might well be a bug; for example, the user may have omitted to include an extra clause in R_i's antecedent which ensures that R_i over-rules Rj under certain circumstances.

3.2. The Verification of Rule Bases

As rule-based systems are increasingly being considered for safety-critical applications, it has become important that one be able to verify that the system is 'correct' with respect to its specification. Because of the procedural semantics of many inference engines, this is not an easy task.

For a given input specification, AbsPS generates a description of the set of final databases computed by the rule base. This can be compared with the formal specification of the rule base. In AbsPS, this is not done automatically, rather, we compare the I/O mapping with the formal specification by hand to see if they concur. This is relatively easy because the I/O mapping has abstracted out all of the difficult-to-reason-about aspects of the rule base (i.e. pattern matching, conflict resolution and flow of control) and represents a declarative statement of what the rule base computes. Figure 9 shows a subset of the I/O mapping for the rule base from which R3 (figure 4) was derived.

```
Input Specification:
  GOAl(ASSESS_WEIGHT(x)), HEIGHT(x,h).
  c:INTEGER(h) A h>50 A h<100.

Output Description:
  WEIGHT(x,h/2),
    where  c.INTEGER(h) A h>75 A h<85.
  WEIGHT(x,h/3),
    where  c:INTEGER(h) A h<76.
  WEIGHT(x,h*2),
    where  -c:PRIMATE(x)  A c:INTEGER(h)
           A h>50 A h<100.
```

Figure 9 - Subset of an I/O mapping.

Given the simple declarative nature of AbsPS' I/O mappings, we are confident that the process of comparing it with a formal specification can be automated.

3.3. Rule Base Chunking

Abstract interpretation can be used to analyse a rule base into useful groups. For example, if the rules within some subset of the rule base are only used in sequence without any other rules interrupting, then that sequence of rules can be collapsed into a single rule. Abstract interpretation yields the required information and could form the basis of a rule-chunking mechanism which improves efficiency.

Large, flat rule bases can be difficult to maintain and debug [cf. R1/XCON, McDermott, 1981]. One remedy is to group rules automatically into functional sub-groups. These modules can be viewed as self-contained wholes which communicate with other modules via restricted channels. Jacob and Froscher [1988] have developed a rule-clustering algorithm to tackle this problem. The algorithm endeavours to cluster rules so as to minimise inter-group coupling and maximise intra-group cohesiveness by examining the dependencies between the rules. Though applied to OPS5, this algorithm ignores the procedural semantics of the language and so may generate some 'loose' groupings. We are now applying AbsPS to this problem [Evertsz and Motta, 1991], however, it is too early to tell whether the increased precision is worthwhile.

4. Cyclical Rule Bases

Current approaches to analysing rule-based systems assume that there are no cycles. AbsPS is able to identify cycles, but is not able to reason about what they compute. During abstract interpretation, there are three major problems to be solved in analysing loops:

- Identifying the group of rules which forms the loop;
- Computing what changes on each loop iteration and deriving the termination condition for the loop;
- Representing the abstract behaviour of the loop so that other rules in the ruleset can manipulate the abstract sequence of WM changes effected by the loop.

Each of these problems is non-trivial, although identifying a loop is easier than the other two tasks. In most programming languages, identifying *explicit* loops is an easy problem. High-level languages provide structured looping constructs which are used to express iteration (e.g. Pascal's WHILE loop); thus, loops are made explicit by the language. Production systems do not incorporate syntactic conventions for flagging loops, rather, iteration occurs as a side-effect of the temporal aspects of the data in WM which force a particular control flow.

Abstract interpretation enables one to identify loops by modelling the temporal flow of concrete data items in terms of abstract ones. AbsPS does this by recording the fired instantiations as it goes along. If, down a given path, AbsPS is about to fire an instantiation of a rule which it fired earlier in the path, then the sequence of rules between those two points constitutes a loop. Note that this instantiation must be the one chosen for firing; merely being in the conflict set is not enough to constitute a loop.

Our definition of a loop is quite general. One can, however, think of other more restrictive definitions. For example, one could restrict our definition so that a repeating sequence of rules is only considered to be a loop if it processes items which it has produced on previous cycles. We argue that our chosen definition of 'loop' needs to be this general if we are to capture more obscure types of loop. For example, consider a rule, R_i, which on each cycle processes one of a sequence of elements in WM. This is equivalent to *generating* and processing them one at a time. It is unnecessarily restrictive to regard only the latter as a loop just because it processes values generated by itself on earlier cycles. Rj is just as cyclical, but processes the whole sequence *after* that sequence has been generated, rather than generating and processing the items one at a time.

Having identified a cyclical sequence of rules, beginning with R_i, it is not difficult to identify the termination condition of the loop - it terminates when one of the constraints of the rule is violated. AbsPS can already synthesise an expression which describes this condition - it does so when building an 'exclusion clause' during conflict resolution. The exclusion clause specifies the conditions under which a given rule could over-ride another which would otherwise have been selected during conflict resolution. The loop termination problem is a subset of the exclusion clause generation one. Rather than build a clause which expresses the conditions under which a rule would be prevented by another from firing, AbsPS builds a clause which expresses the conditions under which a rule could not fire, regardless of the other rules in the rule base - this is the termination condition.

Once the cycle has been identified and its termination condition generated, its output must be represented in such a way so as to enable the other rules to manipulate it. This temporal sequence of outputs could be represented as an aggregate data object, in the manner of Waters [1979]. This would enable AbsPS to manipulate it as a single entity. This functionality is still being developed and has yet to be implemented.

5. Tractability Issues

AbsPS' performance depends to a large extent on the characteristics of the rule base it is analysing. Because it explores all paths, it is the number and length of the paths which is the crucial feature, rather than the number of rules in the rule base. The number and length of paths is dependent, at least in part, on the size of the rule base, therefore rule base size is a rough guide to performance.

AbsPS has been used to analyse rule bases containing up to 50 rules. It takes on average 8 CPU minutes to analyse 20 rules and 45 CPU minutes to analyse 50 (Symbolics 3630). However, these timings are misleading because the program is implemented in a very naive fashion; the complete set of instantiations is recomputed on each cycle, for example. Many of the implementation techniques used in rule-based systems could be applied to abstract interpretation with little modification, including: the saving of instantiated rules and antecedent clauses from cycle to cycle (as in the Rete match algorithm, [Forgy, 1982]); the sharing of unification effort amongst equivalent antecedent clauses; and more efficient variable lookup [cf. Warren, 1977].

AbsPS' mean computational complexity is 0(N*R*P), where N is the number of terms in the input specification, R the number of rules in the rule base and P the number of paths which can be followed for each term in the input specification. By implementing a more efficient abstract interpreter this performance could be improved to 0(N*log(R)*P).

Though in the worst case the number of paths is combinatorially related to rule base size, AbsPS' performance is much better in practice. This is because it prunes paths a priori by taking advantage of the control information in the rule base. Indeed, rule base size is less of a problem if one partitions a rule base into self-contained modules with well-defined information flow between them; it would then be possible to analyse each module individually. Once a module had been analysed, it could be treated as a 'black box* and not analysed again until the user alters its contents. This methodology would be particularly valuable when incrementally developing a rule-based system - only those modules which have changed would have to be reanalysed. This is another compelling reason to develop an automatic rule-grouping algorithm.

6. Conclusions

We began by highlighting the problems which the procedural semantics of forward-chaining production systems cause for analysis. Previous approaches to the problem of analysing rule bases have only considered the language's declarative semantics - this is adequate so long as the procedural semantics are not an issue. We then described an algorithm for the abstract interpretation of production systems which models the effects of conflict resolution, closed-world negation and retraction.

This algorithm derives the I/O mapping for a rule base and can be used to identify common errors such as unreachable rules and dead-end paths. Because the algorithm abstracts out the intermediate procedural aspects of the rule base, its I/O mappings are well suited to formal verification. Though AbsPS cannot accurately reason about rule cycles, it can identify cycles and compute their termination conditions.

Abstract interpretation is a combinatorial problem, however, in practice it is tractable precisely because of its ability to make use of the control information which is implicit in the rule base. Grouping rules into self-contained functional units offers the prospect of interactive use of AbsPS if each group is small in size (of the order of 20 rules). Interestingly, abstract interpretation itself may offer a means by which such rule groups could be generated automatically.

Acknowledgements

Many thanks to Ian Assersohn and Stuart Watt for reading and commenting on earlier drafts, and to Enrico Motta for provocative discussions of this work. I am deeply indebted to Mark Elsom-Cook for his valuable input throughout this research.

References

[Beauvieux, 1990] A. Beauvieux. *A General Consistency (Checking and Restoring) Engine for Knowledge Bases.* ECAI90, Stockholm, pp77-82.

[Evertsz, 1990] R. Evertsz. *The Role of the Crucial Experiment in Student Modelling.* Doctoral Dissertation (and forthcoming CITE Report), IET, The Open University, U.K.

[Evertsz and Motta, 1990] R. Evertsz and E. Motta. - Work in Progress.

[Forgy, 1982] C.L. Forgy. *Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem.* Artificial Intelligence, 19, 1, pp 17-38.

[Ginsberg, 1988] A. Ginsberg. *Knowledge Base Reduction: A New Approach to Checking Knowledge Bases for Inconsistency and Redundancy.* AAAI88, St Paul.

[Jacob and Froscher, 1988] RJ.K. Jacob and J.N. Froscher. *Facilitating Change in Rule-based Systems,* in Hendler, J.A. (ed.), Expert Systems; The User Interface Ablex Publishing Corp., pp249-284.

[McDermott, 1981] J. McDermott *R1: The formative years.* AI Magazine, pp21-29.

[Meseguer, 1990] P. Meseguer. *A New Method to Checking Rule Bases for Inconsistency: A Petri Net Approach.* ECAI90, Stockholm, pp437-442.

[Nguyen *et* al., 1985] T.A. Nguyen, W.A. Perkins, T.J. Laffey, and D. Peeora. *Checking an expert system knowledge base for consistency and completeness.* Proceedings of 9th UCAI, pp375-378.

[Rousset, 1988] M.C. Rousset. *On the Consistency of Knowledge Bases: COVADIS System.* ECAI88, Munich.

[Suwa *et al.,* 1982] M. Suwa, A.C. Scott and E.H. Shortliffe. *An approach to verifying completeness and consistency in a rule-based expert system.* AI Magazine, Fall, pp16-21.

[Warren, 1977] D.H.D. Warren. *Logic programming and compiler writing.* DAI Research Report 44, University of Edinburgh.

[Waters, 1979] R.A. Waters. *A Method for Analyzing Loop Programs.* IEEE Transactions on Software Engineering, SE-5:3, May.