Machine Learning, 37, 143–181 (1999)
© 1999 Kluwer Academic Publishers. Manufactured in The Netherlands.

Effective and Efficient Knowledge Base Refinement

LEONARDO CARBONARA leonardo.carbonara@bt.com
BT UK Markets, British Telecom, pp 411.7, 120 Holborn, London EC1N 2TE, UK

DEREK SLEEMAN dsleeman@csd.abdn.ac.uk
Department of Computing Science, King's College, University of Aberdeen, Aberdeen AB24 3UE, UK

Editor: Raymond J. Mooney

Abstract. This paper presents the STALKER knowledge base refinement system. Like its predecessor KRUST, STALKER proposes many alternative refinements to correct the classification of each wrongly classified example in the training set. However, there are two principal differences between KRUST and STALKER. Firstly, the range of misclassified examples handled by KRUST has been augmented by the introduction of inductive refinement operators. Secondly, STALKER's testing phase has been greatly speeded up by using a Truth Maintenance System (TMS). The resulting system is more effective than other refinement systems because it generates many alternative refinements. At the same time, STALKER is very efficient since KRUST's computationally expensive implementation and testing of refined knowledge bases has been replaced by a TMS-based simulator.

Keywords: knowledge base refinement, theory revision, knowledge acquisition, truth maintenance, dependency networks, expert systems

1. Introduction

Constructing an accurate representation of an expert's domain knowledge is a crucial aspect in the development of a Knowledge-Based System (KBS). This process is essentially composed of three phases:

- knowledge elicitation;
- knowledge representation;
- testing and subsequently refining the initial knowledge base.

These three activities are collectively referred to as the knowledge acquisition task. This process is traditionally carried out through protracted interaction between a domain specialist and a knowledge engineer, and is extremely difficult and time-consuming. Feigenbaum (1977) terms this the *bottleneck problem*. To alleviate this bottleneck, systems and techniques have been developed to make this process less painful for both the domain expert and knowledge engineer. The first tools for knowledge acquisition were mainly concerned with the automation of the knowledge elicitation and knowledge representation phases (Bairess, Porter, & Murray, 1989; Boose & Gaines, 1989). More recently, a number of Knowledge Base Refinement (KBR) systems (Craw & Sleeman, 1990; Ginsberg, 1988a; Mahoney &

Mooney, 1993; Ourston & Mooney, 1990; Politakis & Weiss, 1984; Richards & Mooney, 1995; Tangkitvanich & Shimura, 1992; Towell & Shavlik, 1992; Wilkins, 1990) have been developed.

The task of knowledge base refinement can be formalized as follows. Let KB_0 be a knowledge base, and let C be a set of cases for which the expert's answer is known. We assume that KB_0 is not a perfect knowledge base, and we expect that KB_0 will not completely reproduce the expert's judgement over C. The task of a refinement system is to discover, and possibly modify "faulty" rules in KB_0, so as to improve the empirical performance of the knowledge base, i.e., to make it capable of reproducing the expert's performance in more cases. In order to do this, the cases in C will be partitioned as two subsets: training cases will be used as source of evidence for generating refinements, and test cases will be employed to measure the effectiveness of the refinements. In keeping with accepted statistical practice, these two subsets should be disjoint. An important assumption is that KB_0 needs only minor "tweaking" rather than a major overhaul. We refer to this as the *tweaking* assumption. Under this assumption, refinements are preferred which attempt to improve the effectiveness of KB_0 while making the fewest changes to KB_0.

Since there are usually different ways to fix a particular fault, most refiners use extensive heuristics to choose a refinement for a given wrongly solved task. These heuristics are based on a series of assumptions (e.g., the nature of the knowledge to be refined, the probable causes of failure, etc.), and are normally encoded in the refinement algorithm itself. Hence, the resulting systems lack flexibility. Other systems perform a certain amount of search, and test alternative refinements to find the one that gives the greatest gain in empirical performance. Among the alternatives to heuristic refinement generation are FORTE (Richards & Mooney, 1995), SEEK2 (Ginsberg, 1988a), and KRUST (Craw & Sleeman, 1990), which constitutes the basis of the presented research. KRUST, which is outlined in Section 2, generates many possible refinements to correct a single incorrectly solved training example. Strictly speaking, KRUST also uses heuristics since it does not search the entire solution space for a given refinement problem. Nevertheless, KRUST searches a large portion of the solution space since for a given error it systematically generates all the alternative refinements which can be produced with its set of refinement operators. Such a method is of course less biased towards pre-defined assumptions, but, on the other hand, is obviously liable to problems of computational intractability. In fact, in KRUST all the alternative solutions must be implemented as KBs and tested against a set of test cases to select the best refined KB; these two operations are extremely expensive both in space and CPU time. KRUST tried to overcome this by employing a limited set of refinement operators, together with filters to discard redundant, contradictory, and unpromising refinements. However, refinement systems using broader sets of refinement operators can correct a larger class of errors than KRUST.

Given these premises, it is clear that a system able to generate alternative corrections, making use of an extensive set of refinement operators, while maintaining a reasonable degree of efficiency, would constitute an appealing and powerful alternative to heuristic refinement generation. These are the aims of the STALKER (Speeding up **T**he **A**lgorithm for **K**noweldg**E** base **R**efinement) system (Carbonara & Sleeman, 1996); in particular, STALKER is a hill-climbing, KBR system, which:

- broadens the class of errors handled by KRUST by using inductive refinement operators (i.e., operators based on an inductive learning technique);
- overcomes the computational problems inherent in generating multiple refinements by means of a technique based on Truth Maintenance that allows efficient testing of alternative (refined) KBs.

While inductive operators have been already employed by other refinement systems, the use of a Truth Maintenance mechanism to speed up the evaluation of refinements is unique to STALKER. Nevertheless, other refinement systems exist that use a TMS to help the *detection* of faults and the generation of refinements. For a discussion of these systems, see Section 5.

STALKER's basic cycle is as follows. Firstly, the initial KB and the training examples are converted into a Truth Maintenance System as described in Section 3.2. For each training case in the training set, the cause of failure for the current training case is detected and a set of alternative refinements is generated. Secondly, each of the refinements is efficiently tested by using the Truth Maintenance System, and scored according to its performance on the complete training set. If all the refinements generated fail to produce the desired effect, inductive operators are used to generate more radical corrections. The best refined KB is implemented, and the cycle is repeated for the next training case. The algorithm terminates when all the training cases have been processed. The final refined KB is then evaluated against an independent set of test cases.[1]

As can be seen, STALKER uses a single-example revision algorithm to refine a set of cases. Section 3 explains why this approach was adopted and why an example-covering revision algorithm was not considered to be appropriate.

STALKER has been tested on two real-world rule bases, one in molecular biology and one in plant pathology (see Section 4). Our results have been compared to those obtained for other symbolic and connectionist refinement systems, showing that STALKER can perform at comparable, or higher, levels than heuristic refinement methods in terms of the accuracy of the KB produced. STALKER proved to be 50 times faster than its predecessor KRUST on these tasks.

The rest of the paper is organized as follows. Section 2 describes the KBR system KRUST. In Section 3 STALKER is described in some detail with the help of an example. In Section 4 experimental results are presented and discussed. Section 5 compares STALKER with other refinement systems. Section 6 includes conclusions and suggestions for further work.

2. The KRUST system

KRUST (Craw & Sleeman, 1990) repairs backward-chaining propositional KBs with rule ordering as their conflict resolution strategy. KRUST assumes that rules are represented in Disjunctive Normal Form (a conjunction of antecedents and one consequent). Negated literals are not allowed. Literals are expressed as Attribute-Object-Value triples. In particular, KRUST provides for the refinement of three types of attributes:

Discrete attributes. A discrete structure is a finite set which has no ordering. Suppose the Attribute-Object-Value triple ⟨colour wine white⟩, which is an antecedent of rule R, is to be generalized. Suppose also that the value of attribute colour for the current training example is red. In the case of a discrete attribute, the antecedent is generalized by duplicating rule R, and changing antecedent ⟨colour wine white⟩ to ⟨colour wine red⟩ in the new rule. This is considered to be a generalization because, in so doing, all the examples which were previously satisfied by rule R still are, and in addition the current training example is also satisfied by a rule which differs from R only in the modified antecedent. Note that in this representation discrete antecedents cannot be directly specialized.

Linear attributes. A linear structure is a finite or infinite list. The ordering for the items is the "natural" ordering $x < y$ if x precedes y in the list. Some lists must be specified explicitly, others have an implicit successor function. Numeric attributes are an example of linear attributes taking their values from an infinite list. Antecedent ⟨price wine [5 7]⟩, for instance, would be modified by simply changing range [5 7] so that the value for the current training example is included in the range (in the case of generalization), or excluded from it (in the case of specialization). Given the antecedent ⟨body wine [light full]⟩ and the finite, ordered list [light medium full powerful], this antecedent could, for instance, be generalized to ⟨body wine [light powerful]⟩, or specialized to ⟨body wine [light medium]⟩.

Tree-structured attributes. A tree structure specifies a partial ordering $x < y$ if x is an ancestor of y in the tree. An antecedent for such an attribute can be generalized by replacing its value with an ancestor in the tree. Similarly, it can be specialized by replacing its value with a descendent. For instance, the attribute ⟨origin wine France⟩ can be generalized to ⟨origin wine Europe⟩, or specialized to ⟨origin wine Loire⟩.
As noted above, KRUST generates and manages multiple refinements to correct a single wrongly classified example. When refining a KB, there are usually many potential ways to fix an error. Consider the following example (bold indicates unsatisfied antecedents):

$$R1: A \leftarrow B, \boldsymbol{C}, D \quad R2: C \leftarrow \boldsymbol{E}, F \quad R3: E \leftarrow G, \boldsymbol{H} \quad R4: E \leftarrow \boldsymbol{L}, M$$

Suppose that B, D, F, G, H, L, M are operational literals (as opposed to C and E which are rule conclusions) and rule $R1$ does not fire because its antecedent C is not satisfied. In turn, rule $R2$, whose conclusion is C, does not fire since neither rules $R3$ or $R4$, the two rules whose conclusion is E, fire. Similarly, this fails to happen as antecedent H in rule $R3$ is not satisfied, and antecedent L in rule $R4$ is also not satisfied. It is thus possible to make rule $R1$ fire by: deleting C from $R1$; deleting E from $R2$; deleting H from $R3$; and deleting L from $R4$. We can group these four cases in three categories (see figure 1); namely, fixes to be made at the top-level of the KB (deleting C from $R1$), fixes at the intermediate-level (deleting E from $R2$) and fixes at the leaf-level (deleting H from $R3$; deleting L from $R4$). Some refinement systems prefer to perform changes at a fixed level in the KB and, if at that level alternative corrections are still possible, they choose the refinement to be implemented by means of some heuristics (information gain, statistical methods, etc.). By doing so, they reduce the possible combinatorial explosion of refinement generation, but are liable to miss solutions. As mentioned earlier, KRUST adopts a different approach and generates many

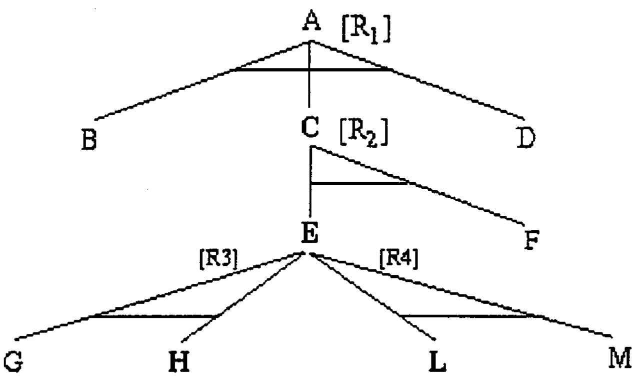


Figure 1. An AND-OR tree representation of rule chains.

alternative refined KBs that are then tested against a set of examples in order to choose the one(s) that performs best with a given set of cases. The system works as follows.

2.1. Rule classification

Given a single training example for which the expert and the KB conclusion disagree, the Rule Classifier tags endpoint rules which should be allowed to fire or prevented from firing. Endpoint rules are rules that conclude one of the categories recognized by the KBS. In the AND-OR tree representation shown in figure 1 endpoint rules are easily detected: their conclusion is the root of a tree. In particular, KRUST uses the following classification of rules:

- the *Error Causing* rule is the rule that is enabled (i.e., all its antecedents are satisfied), that wins the conflict resolution (this rule fires), and has a conclusion different from the expert's;
- *Potentially Error Causing* rules are all the other enabled rules whose conclusions are different from the expert's (they may interfere with proposed corrections);
- *No Fire* rules are target rules (i.e., endpoint rules whose conclusion is the same as the expert's conclusion) which have a higher priority under the conflict resolution strategy than the Error Causing, but are not enabled;
- *Can Fire* rules are enabled target rules which fail to fire because they do not win the conflict resolution;
- *No Can Fire* rules are target rules which fail to fire both because they are not enabled and they have not won the conflict resolution.

The rule classification algorithm can be summarised as follows. Given a KB, a misclassified example belonging to class A, and the proof or attempted proof of the example, the highest priority, satisfied endpoint rule is the Error Causing rule. Endpoint rules concluding A with higher priority than the Error Causing rule are classed as No Fire rules. Endpoint rules which appear with lower priority than the Error Causing rule may be Can Fire rules, No Can Fire rules, or Potentially Error Causing rules, depending on their conclusion and whether or not they are satisfied. Those rules concluding A are either satisfied, hence Can Fire rules, or not satisfied, hence No Can Fire rules. Satisfied rules which do not conclude A are Potentially Error Causing rules. All other rules are of no interest as input to the Refinement Generator.

With respect to the example above, for instance, $R1$ is the only endpoint rule, and is classified as a No Fire rule. The other rules ($R2$, $R3$ and $R4$) are dynamically inspected by the system during the refinement generation phase described below.

2.2. Refinement generation

Once the rules have been classified, the Refinement Generator produces a set of alternative refinements that aim at preventing the Error Causing rule (and the Potentially Error Causing rules) from firing and enabling the No Fire, Can Fire and No Can Fire rules to fire. Refinement operators can be divided into two main categories. Generalization operators try to make a conclusion in the KB easier to satisfy. Conversely, specialization operators attempt to make a conclusion more difficult to be reached. KRUST's generalization and specialization operators can in turn be sub-divided into three classes as shown in Table 1.

The Antecedent Change and Rule Change operators are the basic refinement operators which are common to nearly all symbolic refinement systems. In KRUST, generalization can be performed by increasing the range for an attribute (Antecedent Generalization), by retracting an antecedent from a rule (Antecedent Deletion), or by adding a new rule. KRUST does not use induction, hence its rule addition operator very simply creates a new rule whose conclusion is the expert's conclusion and whose body is the conjunction of all the facts in the current training example. A KB can be specialized by decreasing the range for an attribute (Antecedent Specialization), or by retracting a rule (Rule Deletion). Again, as the system is not provided with inductive operators, it cannot add new antecedents to an existing rule. In addition to these basic operators, KRUST also uses Rule Priority Change operators which are possible with the conflict-resolution strategy used by the system, namely rule order. These later operators can change the priority of a rule by altering its position in the KB.

These individual alterations are then combined by the Refinement Generator to produce refinements with the desired effect. The algorithm used by the Refinement Generator to assemble refinements is sketched in Table 2.

Note that when the Refinement Generator builds the revisions for a given rule, e.g. to enable a No Fire rule, not only will it produce changes that generalize the premise of that

Table 1. KRUST's refinement operators.

	Antecedent change	Rule change	Rule priority change
Generalization	Antecedent generalization Antecedent deletion	Rule addition	Rule priority increase
Specialization	Antecedent specialization	Rule deletion	Rule priority decrease

Table 2. The Refinement Generator.

- For each No Fire rule build refinements which enable the No Fire rule;
- Build a refinement changing the Error Causing rule's conclusion to A;
- For each Can Fire rule:
    - Build a refinement which increases the priority of the Can Fire rule above the Error Causing rule;
    - Build refinements which:
        - Increase the priority of the current Can Fire rule above the highest priority Can Fire rule,
        - Disable the Error Causing rule or decrease its priority below the Can Fire rule,
        - Disable or decrease the priority of all Potentially Error Causing rules with priorities between the Error Causing rule and the new priority Can Fire rule;
- For each No Can Fire rule:
    - Build refinements which enable the No Can Fire rule (these refinements transform the No Can Fire rule into a Can Fire rule);
    - merge them with refinements which treat the No Can Fire rule as a Can Fire rule (see previous bullet);
    - Build a refinement which assembles a new rule with the conjunction of known facts for the training case as its premise and A as its conclusion.

rule, but it will also recursively climb the derivation chain departing from that rule, and generate revisions that generalize all the rules on the chain. In this way, changes at all levels in the theory are produced.

Each individual change produced by the Refinement Generator is represented by KRUST with a frame-like structure specifying the name of the rule to be changed, the change type, and the place of the change (e.g., ⟨rule_name: $R2$; change: decrease priority; place: $R4$⟩). A refinement is a conjunction of such changes. The output of the Refinement Generator is the list of alternative refinements produced for a given training example.

2.3. *Filtering, implementation, and evaluation of refinements*

Before the refinements produced by the Refinement Generator are implemented, the First Filtering Phase removes refinements which are redundant, conflict with each other, or involve rules which are thought to be correct. (Two refinements are conflicting if they propose conflicting corrections to the same rule, e.g., one refinement suggests generalizing a rule and another refinements specializing it). The remaining refinements are then implemented by the Rule Changing Mechanism. The set of refined KBs is then tested against the current training example and a set of priority examples, i.e. important cases that must be answered correctly. KBs that do not pass this Second Filtering phase are rejected. Finally, the Judgement Module tests the remaining KBs against a larger set of examples, and ranks them according to their performance. The KB with the best score is recommended to the user.

In the next section, KRUST's successor, STALKER, is described with the help of a working example.

3. Overview of STALKER

In the previous section an overview of the KRUST system has been given. As it has been pointed out, KRUST's most distinguishing feature is its ability to generate a complete set of alternative refinements (modulo the given set of refinement operators) to fix the failure of a single training example. This approach allows KRUST to focus its search for the exact cause of failure of each training example, and therefore to generate very accurate corrections. Moreover, considering a broad range of possible alternatives to correct a single fault increases the chances of finding a "good" refinement among those produced. On the other hand, in KRUST all the potential solutions must be implemented as KBs and run on the test set in order to choose the one that performs best. Therefore, the undeniable advantages offered by multiple refinement generation are counter-balanced by a very expensive evaluation phase. KRUST tried to overcome this potential disadvantage by employing a limited set of refinement operators. As opposed to other refinement systems, in fact, KRUST is not provided with operators using an inductive learning mechanism (such as ID3, for instance) to create new rules or add new antecedents to existing rules. Filters were also used to discard redundant and contradictory refinements. While the use of filters effectively helped to contain the number of refinements without losing any solution, limiting the set of operators had the obvious effect of reducing the class of errors the system was able to cope with. In fact, KRUST has no means to deal with refinements that, although fixing the current error, make other training examples fail, which were previously correctly classified. These new failing examples are called *new inconsistencies* (also referred to as *new conflicts*).

At the other end of the spectrum, we find "batch" refinement systems like EITHER (Ourston & Mooney, 1990), which process all the training examples in a single phase, and limit the computational costs of multiple refinement generation by preferring the smallest number of fixes that correct the largest number of examples. However, these systems explore a smaller portion of the search space, and can overlook promising solutions.

Somewhere in between KRUST's multiple-refinement-generation approach and EITHER's batch approach lies a class of algorithms, like FORTE (Richards & Mooney, 1995) and SEEK2 (Ginsberg, 1988a), which use the evidence collected from all the examples to identify the portions of the KB that are most likely to be in error and generate multiple repairs that correct as many examples as possible. This last method has the clear advantage that the system can exploit the information coming from all the examples to identify the most promising corrections. However, the most *promising* corrections are not necessarily the most *effective*. Consider the following example. Suppose that generalizing antecedent '$k < 3$' in a rule can potentially correct the classification of 5 examples. These examples will have different values for attribute k, e.g., $3, 7, 5, 9, 6$, respectively. To correct all 5 examples the antecedent should be generalized to the maximum value for attribute k in the 5 examples considered, i.e. the antecedent should become '$k < 10$'. Suppose, though, that the 'correct' antecedent is '$k < 7$', because, if the antecedent is generalized further, negative examples are 'captured' by the rule where the generalized antecedent appears. Assume also that there are alternative revisions to correct the two remaining examples with $k = 7$ and $k = 9$. This example is not rare, but rather the norm in many real-world domains. Systems generating repairs based on the evidence collected from multiple examples cannot effectively cope with this situation. They are bound to over-generalize the KB, and then try to recover from the over-compensation introduced by using induction. On the other hand, for a system processing one example at a time, this does not constitute a problem. Such a system would in fact generalize the antecedent until new inconsistencies start being generated. At that point, alternative refinements which do not generate new conflicts would presumably be preferred. Only if there were no such alternatives would induction be invoked. Hence, processing one example at a time gives more control on the effects of the refinements generated, and allows more accurate corrections to be implemented.

For the above reasons, we have decided to adopt KRUST's multiple refinement generation approach in the design of STALKER, the system presented in this paper. To overcome KRUST's limitations, STALKER:

- uses inductive refinement operators based on ID3 to recover from new conflicts generated by other refinements;
- speeds up the testing of alternative refinements by using a Truth Maintenance System.

3.1. The basic refinement algorithm

STALKER is a KBR system that corrects faults in propositional rule bases. It is assumed that:

- the knowledge bases to be refined are composed of production rules;
- the inference engine of the knowledge-based system performs a backward-chaining, depth-first search;
- the conflict-resolution strategy adopted by the inference engine is rule-order. (Note that rules with certainty factors could be easily accommodated by STALKER.)

STALKER's basic refinement cycle for each single training example selects a refinement from a set of alternatives which:

- fixes the current training example;
- does not result in new inconsistencies (i.e., does not make previously correctly classified examples fail);
- maximizes the performance of the refined knowledge base against the training set.

The system, which is sketched in figure 2, works as follows. Firstly, the original KB and the sets of training cases are converted into a Truth Maintenance System (TMS). The TMS-version of the KB is used by the Rule Classifier and Refinement Generator to query the KB, and by the Tester Simulator to perform efficient testing of the refinements. After the original KB and the examples have been converted into the TMS, for each training example the following steps are performed:

1. The relevant rules are tagged by the Rule Classifier.
2. A set of alternative refinements for the current training example is produced by the Refinement Generator. (Note that both the Rule Classifier and the Refinement Generator work as in KRUST; see Section 2.)

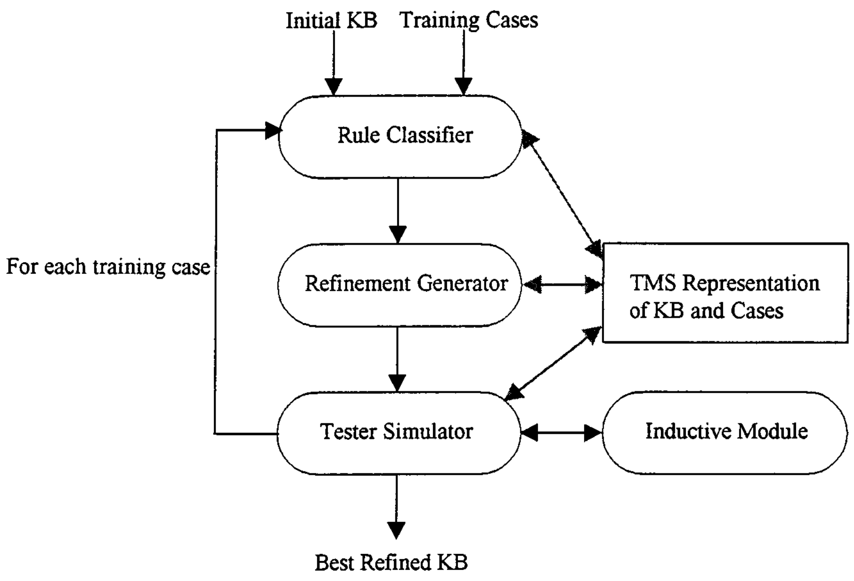


Figure 2. STALKER architecture: black arrows indicate the data and control flows between the system's components; grey arrows denote queries to the TMS module.

3. The refinements are then passed to the Tester Simulator (see Section 3.2) that implements them on both the KB and the TMS. The TMS is then used to efficiently test the refinements against the whole set of training cases, and orders the refinements according to their performance. This global overview gives the system better direction than if refinements were scored according to their performance on the single training case. In this respect, our approach is similar to PTR's and FORTE's (see Section 5 for a description of these systems).
4. If the first refinement in the ordered set R causes any new inconsistencies, the Inductive Module (see Section 3.3) is invoked and a new set of refinements R' is generated to eliminate the inconsistencies. The new refinements are then passed to the Tester Simulator for testing and ranking, and the best refinement is selected. If refinements in R' are not able to correct the fault without introducing further inconsistencies, the process is repeated for the remaining refinements in R. If these attempts also fail, no refinement is selected and the whole process is then repeated for the next training case (go to step 1).
5. When all the training cases have been processed, the final refined KB is returned.

In the rest of the section STALKER's most innovative components, the Tester Simulator and the Inductive Module are described in some detail.

3.2. The Tester Simulator

The basic idea underlying the Tester Simulator is that refinement operations do not involve the whole KB, but only a small portion of it. Hence, most of the testing done by KRUST's tester is redundant because only a few training examples will be affected by a given refinement. Unfortunately, in a standard KBS it is not trivial, if possible at all, to determine which training examples will be influenced by a refinement since the relationships between examples and rules are not explicit. This problem can be overcome by converting the rule base into a dependency graph (or network) which maintains explicit links between rules and examples. This can be achieved by using a Truth Maintenance System (TMS) (Doyle, 1979; McAllester, 1982; de Kleer, 1986), which is a general mechanism for caching, updating and analysing the inferences performed by a problem solver. To implement the Tester Simulator we modified the ATMS described by Forbus and de Kleer (1993).

At the very high level, the Tester Simulator can be described as follows:

```
TESTER-SIMULATOR (refinements, examples, TMS )
    CLASSIFY-CASES (examples, TMS);
    For each refinement r in refinements:
        TMS-CHANGING-MECHANISM (r, TMS);
        score = COMPUTE-REFINEMENT-SCORE (examples);
        ref-list = ordered-insert (⟨r, score⟩, ref-list);
    Return ref-list.
```

The Tester Simulator first of all records how each example is classified by the TMS version of the current KB (CLASSIFY-CASES). Then, each refinement is implemented on the network (TMS-CHANGING-MECHANISM). The revised network is tested against the examples and the number of correctly classified examples is recorded in *score* (COMPUTE-REFINEMENT-SCORE). The output of the algorithm is the list of tested refinements ordered according to their score.

In the following section our approach is illustrated with the help of an example.

3.2.1. The cup theory example. Consider the following theory:

R1: drinking_vessel → stable, liftable, open-vessel.
R2: stable → has-bottom, flat-bottom.
R3: liftable → graspable, lightweight.
[R4: graspable → has-handle.]
R5: graspable → (width small), (material styrofoam).
R6: graspable → (width small), (material ceramic).
R7: open-vessel → has-concavity, upward-pointing-concavity.

This KB is a version of the cup theory (Winston et al., 1983), and classifies an object as a drinking vessel if it satisfies the requirements of being stable, liftable, and an open vessel. We have "corrupted" the original theory by removing a rule for the concept graspable ($R4$). Suppose moreover our training set is composed of six examples ($E1$ to $E6$), three positive ($E1$, $E2$, $E3$) and three negative ($E4$, $E5$, $E6$), which are consistent with the original theory; the examples are shown in Table 3.

The knowledge base and the examples can be represented by means of a dependency graph as shown in figure 3. In this diagram, rule conclusions and antecedents are represented as graph nodes. Rule conclusions (drinking_vessel, stable, liftable, open-vessel, graspable)

Table 3. The cup theory examples.

	$E1$	$E2$	$E3$	$E4$	$E5$	$E6$
has-concavity	√	√	√	√	√	√
upward-pointing-concavity	√	√	√	√	√	√
has-bottom	√	√	√	√	√	√
flat-bottom	√	√	√	√	√	√
lightweight	√	√	√	√	√	√
has-handle	—	√	√	—	—	—
material	styrofoam	styrofoam	ceramic	—	styrofoam	styrofoam
colour	red	red	white	grey	red	blue
width	small	medium	large	small	medium	large
volume	8	16	8	8	16	16
shape	hemisphere	hemisphere	cylinder	cylinder	hemisphere	hemisphere
class	drinking-vessel	drinking-vessel	drinking-vessel	negative	negative	negative

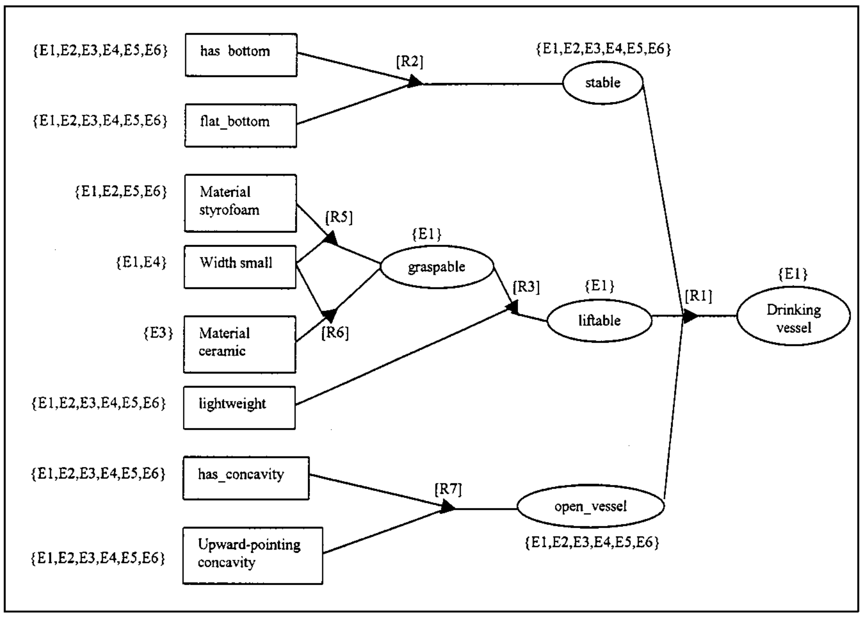


Figure 3. Network representation of the corrupted cup theory.

are shown as ellipses, while leaves, i.e. rule antecedents which are also example features (has_bottom, flat_bottom, has_handle, styrofoam, ceramic, (width small), lightweight, has_concavity, upward-pointing-concavity) are represented as rectangles. Rules are shown as arcs converging at arrows (justifications).

For the sake of clarity arrows have been tagged with the rule they correspond to (in square brackets). In addition, each node has a label (shown in braces) which specifies the examples which satisfy the conditions associated with that particular node. The ordering of the rules is simulated on the network by assigning each justification a numerical index corresponding to the position of the rule in the KB (e.g., the justification corresponding to rule $R1$ is assigned index 1). Justifications with a lower index have higher priority.

The mechanism for assigning labels in our TMS works as follows. Initially, each leaf node in the graph is assigned a label composed of the examples in which the feature represented by the node appears. For instance, the label of leaf node (material ceramic) is $\{E3\}$ since the feature ceramic only appears in example $E3$. A propagation algorithm is then used to compute the labels of the remaining nodes. A justification's label (justifications are represented in the graph as black arrows) is obtained by computing the *intersection* of the labels of its antecedents. Ellipse nodes, which represent rule conclusions, have their label computed by forming the *union* of the labels of the justifications converging at the node.

Node graspable, for example, is the conclusion of rules $R5$ and $R6$. $R5$'s contribution to node graspable's label is the intersection the labels of $R5$'s antecedents: $\{E1, E2, E5, E6\} \cap \{E1, E4\} = \{E1\}$. In turn, $R6$'s contribution is: $\{E1, E4\} \cap \{E3\} = \{\}$. Therefore, node graspable's label is: $\{E1\} \cup \{\} = \{E1\}$.

Given this representation, the task of determining whether a rule satisfies an example can be performed with a simple element-of test checking to see whether the example belongs to the label of the node corresponding to the conclusion of the rule.

When the network is modified to simulate a refinement (e.g., a rule is deleted, or a new antecedent is added to an existing rule), the propagation algorithm updates only the labels of the nodes which have been affected by the refinement. Therefore, the amount of computation performed by the propagation algorithm is much smaller than KRUST's tester, which, after each refinement, must run the revised knowledge base on every example.

Let's now go back to our example. Above, we gave the original theory modified by removing rule $R4$. The modified theory when represented as a network is given in figure 3; this shows that example $E1$ is still correctly classified by the corrupted theory. In fact, for this example rule $R5$ is used to prove the concept graspable. Examples $E2$ and $E3$, on the other hand are now not provable. Let's suppose our current training example is $E2$. The Rule Classifier will classify $R1$ as a No Fire rule. (Note that, since in this theory there is only one endpoint rule ($R1$), no Error Causing rule is detected.) Given STALKER's refinement operators, four alternative refinements will be generated to make $R1$ fire:

Refinement 1: delete antecedent liftable from the body of $R1$; or
Refinement 2: delete antecedent graspable from the body of $R3$; or
Refinement 3: generalize antecedent (width small) in rule $R5$; or
Refinement 4: (generalize antecedent (width small) in rule $R6$ AND generalize antecedent (material ceramic) in rule $R6$).

As can be seen, while the first refinement acts directly upon rule $R1$, the other three refinements suggest changes at a lower level in the rule derivation chain. In fact, they propose to modify rule $R3$, which is invoked by $R1$ to prove the statement liftable, as well

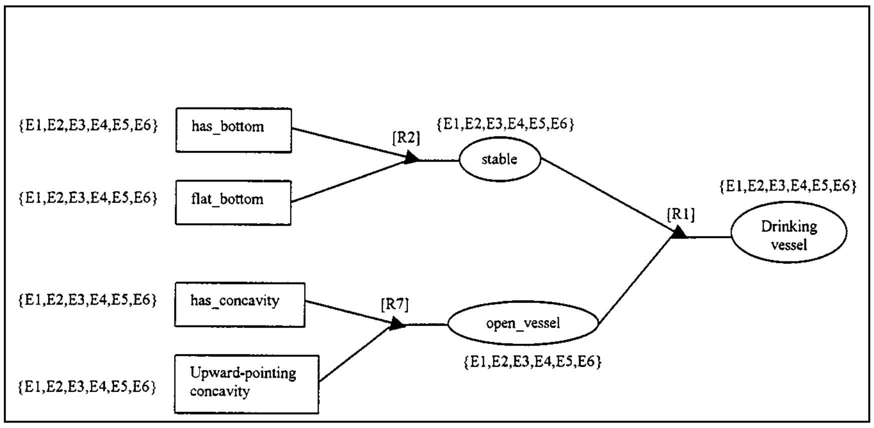


Figure 4. The corrupted cup theory after the first refinement.

as $R5$ and $R6$, which are the two alternative rules that can be invoked by $R3$ to prove the statement graspable. These refinements can be easily implemented on the graph. Deleting liftable from $R1$ implies the retraction of the corresponding node and of all the rules of which liftable is a consequence (see figure 4). After this operation, the labels of the nodes connected with the retracted node (in this case drinking_vessel) are updated. Note that the labels of the other nodes (stable and open_vessel) remain untouched because they are not affected by the refinement; one can imagine label propagation as a "one-way wave" that travels from the deleted node in the direction indicated by the arrows.

As can be seen in figure 4, the refinement does correct the classification of the current training example $E2$, and of example $E3$ as well, but introduces new inconsistencies[2] since also the three negative examples ($E4$, $E5$ and $E6$), which were previously classified correctly, are now classified as drinking vessels. Hence this refinement has score $\langle 3\,\{E4\ E5\ E6\}\rangle$, meaning that the refinement correctly classifies three training examples, but causes new inconsistencies $\{E4\ E5\ E6\}$. In general the score is $\langle n$ {new inconsistencies created by the refinement}$\rangle$, where n is the number of correctly classified examples. (If the refinement did not cause any inconsistencies, the empty set would follow n.) Note that the second refinement produces exactly the same result. Let's consider now the third refinement. We know that $R5$ fails for example $E2$ because antecedent (width small) is not satisfied. In fact, the feature width for example $E2$ has the value medium. To generalize $R5$, STALKER creates a new rule identical to $R5$ except that antecedent (width small) is replaced with (width [small medium]). This refinement and the resulting network are represented in figure 5. Note that the new rule is called $R5a$ and replaces $R5$. This refinement has score $\langle 4\,\{E3\ E5\}\rangle$ since the revised network classifies correctly $E1$, $E2$, $E4$ and $E6$, but mis-classifies $E3$ and $E5$. This refinement generates one new inconsistency, $E5$, as this is the only example that was correctly classified by the initial (corrupted) theory which becomes misclassified after the implementation of the refinement. Although mis-classified, $E3$ is not a *new* inconsistency since it was *already* misclassified by the initial theory.

The fourth refinement yields exactly the same result as the third refinement. The final output of the Tester Simulator is therefore:

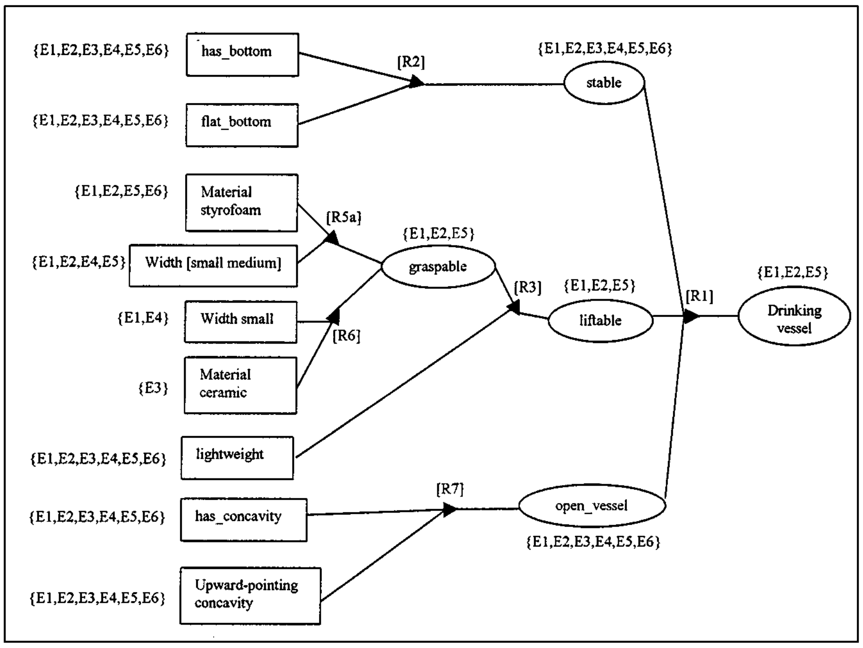


Figure 5. The corrupted cup theory after the third refinement.

Refinement 3: $\langle 4\{E5\}\rangle$
Refinement 4: $\langle 4\{E5\}\rangle$
Refinement 1: $\langle 3\{E4\ E5\ E6\}\rangle$
Refinement 2: $\langle 3\{E4\ E5\ E6\}\rangle$

Since all the refinements create new inconsistencies, STALKER will pass each of the refinements to the Inductive Module (see next section), starting from the one with the highest score. If, as in this case, some refinements have the same score, one is randomly selected. The Inductive Module will try to amend the refinement by eliminating the inconsistencies generated. As soon as a refinement is found which removes the inconsistencies, this is implemented on the network and this sub-cycle is completed; the next training case is then processed, as discussed in Section 3.1.

3.3. The Inductive Module

The Inductive Module is invoked when all the refinements generated for a given training example cause new inconsistencies to arise. In this case, STALKER tries to eliminate these inconsistencies by using an inductive algorithm, namely ID3 (Quinlan, 1986), to learn new rules, or new antecedents to be added to existing rules. The rationale behind this strategy is the following. STALKER's standard refinements are designed to revise a knowledge just enough to achieve the desired effect. If these changes over-generalize or over-specialize the KB, this means that “tuning” the information contained in the KB is not sufficient to discriminate completely between the given examples. Hence, new antecedents for existing rules or new rules must be learnt from the examples themselves. STALKER uses induction in a way similar to EITHER (Ourston & Mooney, 1990). Also in EITHER induction is invoked when other refinement techniques fail to yield the desired results. Nonetheless, as discussed in Section 3.1, we claim that STALKER’s multiple refinement generation combined with its hill-climbing strategy ensures a more parsimonious use of induction since the system has more chances to find, among the many possible refinements generated, one that corrects the current fault without causing new inconsistencies.

The Inductive Module is supplied with the refinement causing the new inconsistencies, the new inconsistencies themselves, i.e., the set of newly misclassified examples, and in some cases the remaining examples. The refinement is then analysed to detect the cause of the new inconsistencies, and appropriate action is taken to try to recover from the over-compensations generated. Below, for each of the four primitive refinement operations,[3] we explain what should be done if an additional inconsistency arises (a), and indicate the examples that will be used by the inductive algorithm (b):

1a. *Antecedent generalization and antecedent deletion.* In these two cases the rule might have been over-generalized. The Inductive Module tries to recover from this situation by generating two alternative refinements:

- adding new antecedents, learnt by induction, to the over-general rule;
- assembling a new rule with the learnt antecedents.

1b. The examples used for induction are the current training example and the misclassified examples.

2a. *Antecedent specialization and rule deletion*. The rule (or the KB in case of rule deletion) might have been over-specialized. Therefore, the original rule is restored and a new attempt at specializing it is made by adding new antecedents generated by induction.

2b. The examples used for induction are those misclassified and the training examples belonging to the class of the specialized rule.

3. *Rule priority increase*. Increasing the priority of a rule can make the rule classify examples belonging to other classes. It is therefore an over-generalization and is treated as described in point 1.

4a. *Rule priority decrease*. In this case new inconsistencies can arise because examples belonging to the rule whose priority has been decreased are classified by rules which now have higher priority. Hence, new antecedents must be added to these high-priority over-general rules.

4b. In this case the examples used by induction are the new inconsistencies and the examples for the over-general rules.

Note that in general a refinement can be composed of a number of applications of the “primitive” refinement operators listed above (1–4). For instance, a refinement might involve generalizing a rule and decreasing the priority of another rule. Each of these operations may be the source of a number of inconsistencies. When the Inductive Module is presented with the inconsistencies generated by a certain refinement, the first step is associating each inconsistency with the primitive refinement operation which caused it. Subsequently, the appropriate inductive operators are applied as described above. By detecting the exact causes of new inconsistencies, this approach is able to minimize the use of induction, and propose specific solutions for each particular type of problem. (Using this analysis it is also possible to reduce the number of changes considered.) Let us now complete the example introduced in Section 3.2.1.

3.3.1. The cup theory example. As explained in Section 3.2.1, the first refinement considered by the Inductive Module is Refinement 3 (generalize antecedent (width small) in *R5*). Firstly, the Inductive Module analyses the refinement to make a hypothesis about the possible cause of failure. Since Refinement 3 was generated to enable a No Fire rule, the only possible source of inconsistencies is an over-generalization. Hence, the Inductive Module tries to learn a discrimination between the current training example $E2$ and the misclassified example $E5$ by invoking ID3 on these two examples. ID3 discovers that the only discriminating feature between the two examples is has-handle. This feature is therefore used to build two alternative refinements:

- Refinement 3 is augmented by adding “add antecedent has-handle to $R5$”. The new refinement (Refinement 5) is therefore:

  generalize antecedent (width small) in rule $R5$ AND
  add antecedent has-handle to $R5$

- Refinement 3 is rejected and a new refinement (Refinement 6) is created:

  add the new rule “graspable $\leftarrow$ has-handle” above $R5$.

These two refinements are then passed to the Tester Simulator for evaluation. Figure 6 shows the implementation of the first refinement on the network. This refinement eliminates the inconsistency since $E5$ is no longer misclassified by the amended network, but causes the misclassification of $E1$, which was previously correctly classified.

The score of Refinement 5 is therefore $\langle 4\,\{E1\}\rangle$ as $E1$ is a new inconsistency and $E3$ is still not correctly classified.[4] The second refinement is shown in figure 7. This refinement not only eliminates the inconsistency, but also fixes example $E3$, so that all six training examples are now correctly classified. (Its score is $\langle 6\,\{\,\}\rangle$.) Hence, this refinement is selected and presented to the user.

4. Experimental results

This section presents the results of an empirical evaluation of STALKER.

4.1. Experiments with the DNA and soybean domains

We have tested our system with two real-word domains, namely the Promoter Recognition and the Soybean Disease domains. The first of these, a domain for recognizing promoters

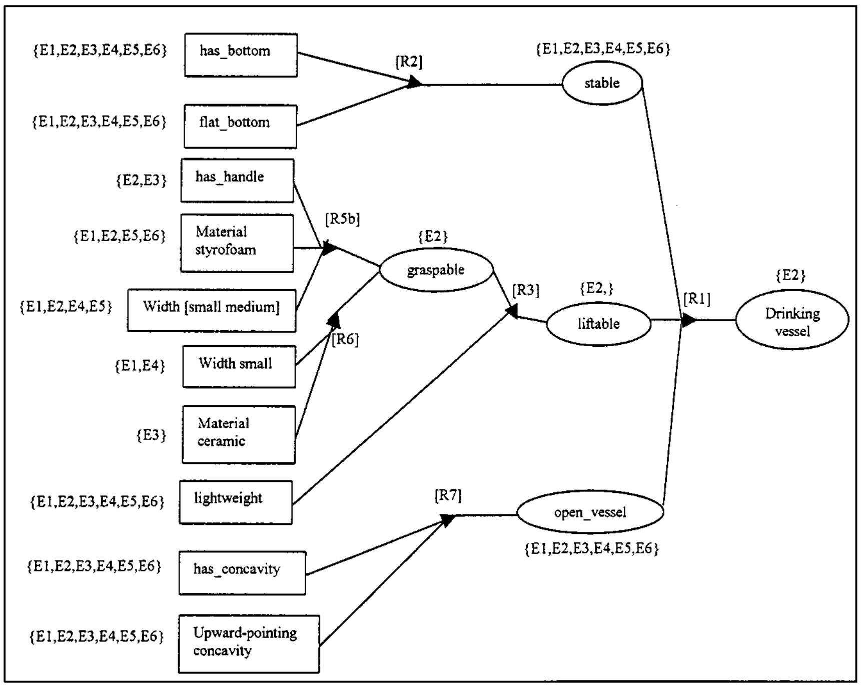


Figure 6. The cup theory after the first inductive refinement.

in DNA sequences, constitutes a single category theory. The second diagnoses soybean diseases and is a multiple category theory. We compared STALKER with its predecessor KRUST and other state-of-the-art refinement systems (see Section 5 for descriptions of these systems).

4.1.1. Accuracy results in the DNA domain. The original Promoter Recognition theory (Towell, Shavlik, & Noordewier, 1990) contains 11 rules with a total of 76 propositional symbols. This theory recognizes promoters in strings of DNA nucleotides. A promoter is a genetic region which initiates the first step in the expression of an adjacent gene. The examples used in the tests consisted of 53 positive and 53 negative examples assembled from the biological literature. The initial theory classified none of the positive examples and all of the negative examples correctly, thus indicating that the initial theory was over-specific (and did not contain any over-general statements). STALKER and the other systems were trained on this domain with a maximum of 90 examples, using the remaining 16 examples as test examples. The results for EITHER, NEITHER, and RAPTURE were taken from Baffes and Mooney (1993). The results for PTR were taken from Koppel, Ronen, and Segre (1994). To make a meaningful comparison, STALKER's results were averaged over 25 independent trials as it was done for the other systems. Figure 8 shows the testing accuracy for the five systems. As can be seen, STALKER's performance is comparable to RAPTURE's, NEITHER's and PTR's on this domain. Note that KRUST's results have not

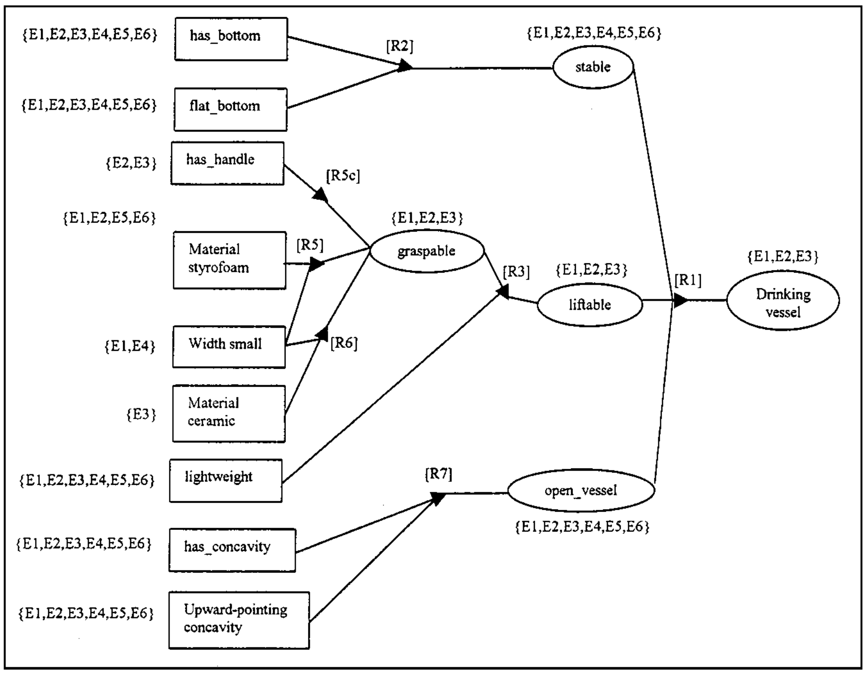


Figure 7. The cup theory after the second inductive refinement.

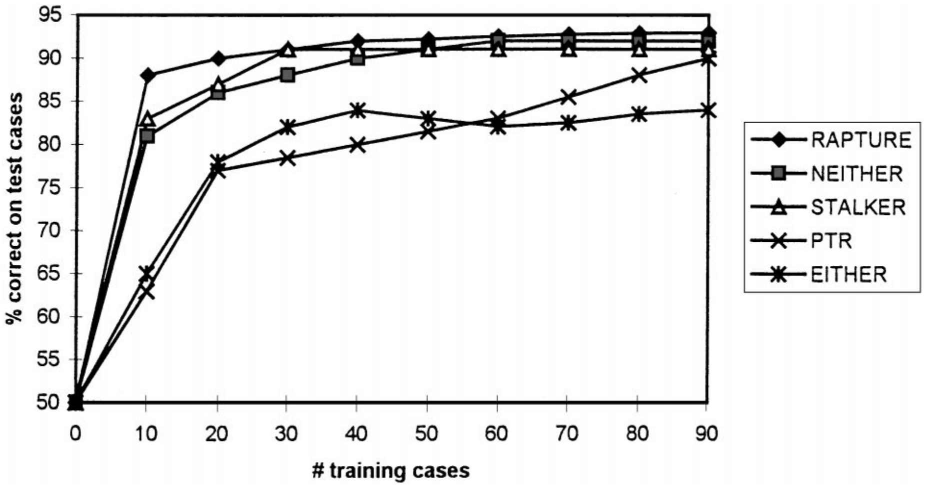


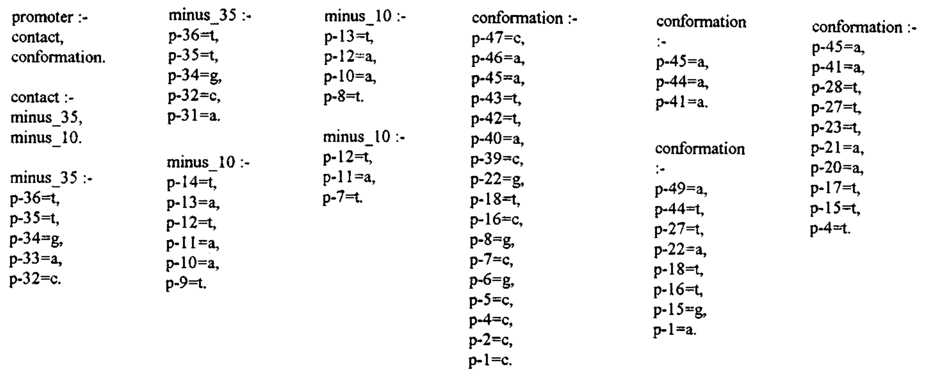
Figure 8. DNA domain testing accuracy.

been included in the graph as they are the same as STALKER's. This was due to the fact that STALKER did not use its new inductive operators to refine this domain, hence obtaining the same accuracy as KRUST. This is an important result since it confirms the hypothesis that in some cases generation of alternative refinements, without invoking induction, can yield very good results. As we shall see later, the only difference in the performance of STALKER and KRUST for this domain was the execution time, much smaller for STALKER due to its use of the TMS.

The Promoter Recognition domain has been thought to possess the M-of-N property, that is, most of its rules should fire when some number (M) of its antecedents are true. This is due to properties of the DNA strands themselves. In such strands, there are several potential sites at which hydrogen bonds can form between DNA and a protein. When enough of these bonds form, promoter activity can occur. Of the system considered, NEITHER and RAPTURE output "numerical" rules, i.e., rules that fire when a given number of antecedents are satisfied. In fact, NEITHER explicitly handles M-of-N rules, while RAPTURE revises probabilistic rules using certainty factors. Koppel, Ronen, and Segre (1994) showed that the original Promoter theory is very accurate provided that its rules are given a numerical interpretation, and they thus concluded that the success of systems like RAPTURE and NEITHER for this domain is not a consequence of learning from examples, but rather a side effect of the use of numerical rules. STALKER, which, on the other hand, has no special provisions to cope with numerical rules achieved better results than any of the other purely symbolic methods considered here.

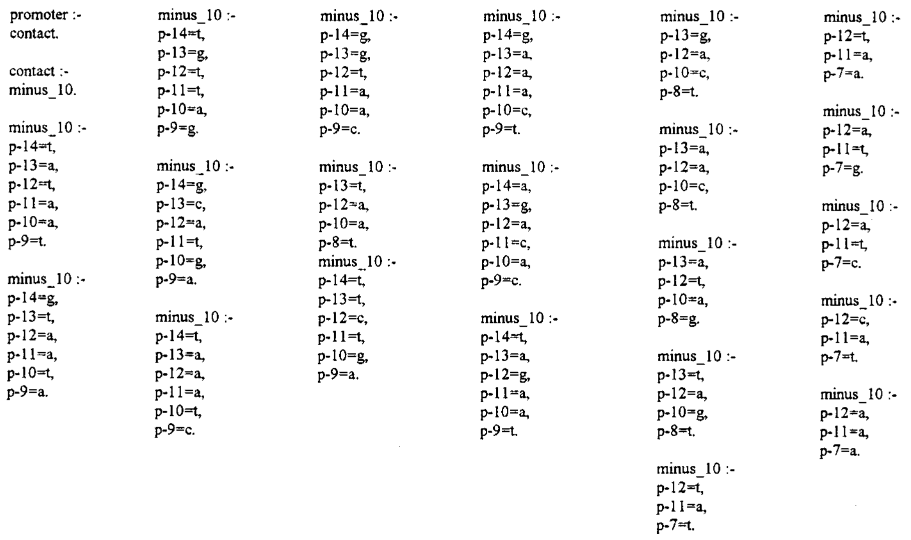
It is also interesting to note that Koppel, Ronen, and Segre (1994) explain PTR's improvement over EITHER as a consequence, among other reasons, of PTR repairing one flaw at a time; note this is also the strategy adopted by STALKER.

Another analogy between the two systems is the fact that, as a result of the single example revision approach, they always delete the extraneous literal *conformation* from the original theory (shown in figure 9), while EITHER occasionally fails to do so, particularly as the number of training examples increases. A typical refined theory produced by STALKER is shown in figure 10. As can be seen, STALKER deleted both literal *conformation* and literal



```
promoter :-
contact,
conformation.

contact :-
minus_35,
minus_10.

minus_35 :-
p-36=t,
p-35=t,
p-34=g,
p-33=a,
p-32=c.

minus_35 :-
p-36=t,
p-35=t,
p-34=g,
p-32=c,
p-31=a.

minus_10 :-
p-14=t,
p-13=a,
p-12=t,
p-11=a,
p-10=a,
p-9=t.

minus_10 :-
p-13=t,
p-12=a,
p-10=a,
p-8=t.

minus_10 :-
p-12=t,
p-11=a,
p-7=t.

conformation :-
p-47=c,
p-46=a,
p-45=a,
p-43=t,
p-42=t,
p-40=a,
p-39=c,
p-22=g,
p-18=t,
p-16=c,
p-8=g,
p-7=c,
p-6=g,
p-5=c,
p-4=c,
p-2=c,
p-1=c.

conformation
:-
p-45=a,
p-44=a,
p-41=a.

conformation
:-
p-49=a,
p-44=t,
p-27=t,
p-22=a,
p-18=t,
p-16=t,
p-15=g,
p-1=a.

conformation :-
p-45=a,
p-41=a,
p-28=t,
p-27=t,
p-23=t,
p-21=a,
p-20=a,
p-17=t,
p-15=t,
p-4=t.
```

Figure 9. The original promoter theory.



```
promoter :-
contact.

contact :-
minus_10.

minus_10 :-
p-14=t,
p-13=a,
p-12=t,
p-11=a,
p-10=a,
p-9=t.

minus_10 :-
p-14=g,
p-13=t,
p-12=a,
p-11=a,
p-10=t,
p-9=a.

minus_10 :-
p-14=t,
p-13=g,
p-12=t,
p-11=t,
p-10=a,
p-9=g.

minus_10 :-
p-14=g,
p-13=c,
p-12=a,
p-11=t,
p-10=g,
p-9=a.

minus_10 :-
p-14=t,
p-13=a,
p-12=a,
p-11=a,
p-10=t,
p-9=c.

minus_10 :-
p-14=g,
p-13=g,
p-12=t,
p-11=a,
p-10=a,
p-9=c.

minus_10 :-
p-13=t,
p-12=a,
p-10=a,
p-8=t.
minus_10 :-
p-14=t,
p-13=t,
p-12=c,
p-11=t,
p-10=g,
p-9=a.

minus_10 :-
p-14=g,
p-13=a,
p-12=a,
p-11=a,
p-10=c,
p-9=t.

minus_10 :-
p-14=a,
p-13=g,
p-12=a,
p-11=c,
p-10=a,
p-9=c.

minus_10 :-
p-14=t,
p-13=a,
p-12=g,
p-11=a,
p-10=a,
p-9=t.

minus_10 :-
p-13=g,
p-12=a,
p-10=c,
p-8=t.

minus_10 :-
p-13=a,
p-12=a,
p-10=c,
p-8=t.

minus_10 :-
p-13=a,
p-12=t,
p-10=a,
p-8=g.

minus_10 :-
p-13=t,
p-12=a,
p-10=g,
p-8=t.

minus_10 :-
p-12=t,
p-11=a,
p-7=t.

minus_10 :-
p-12=t,
p-11=a,
p-7=a.

minus_10 :-
p-12=a,
p-11=t,
p-7=g.

minus_10 :-
p-12=a,
p-11=t,
p-7=c.

minus_10 :-
p-12=c,
p-11=a,
p-7=t.

minus_10 :-
p-12=a,
p-11=a,
p-7=a.
```

Figure 10. A typical revised promoter theory generated by STALKER.

minus_35, while it revised the rules for literal *minus_10*. RAPTURE obtained a somewhat similar result as in its refined theory a higher certainty factor is given to rules concluding *minus_10* than to those for *minus_35*. EITHER, on the contrary, deleted literal *minus_10* and refined the rules for literal *minus_35*. We do not know whether a comparable outcome was produced by PTR as its refined theory is not reproduced in Koppel's paper. NEITHER, on the other hand, did not modify any of the rules for concepts *minus_10* and *minus_35*, except for the addition of the M-of-N condition specifying how many antecedents must be satisfied for a rule to fire. The fact that both STALKER and RAPTURE obtained better results than EITHER on this domain seems to indicate that revisions focusing on *minus_10* generate more accurate theories. The reason why STALKER created many new rules for this literal is that the system generalizes discrete attributes by creating a new rule identical to the original rule except for the value of the discrete attribute which has been generalized (e.g., antecedent $colour = red$ in a given rule is generalized by creating a new rule with antecedent $colour = blue$ instead).

It would be desirable to evaluate formally the distance between the original DNA theory and STALKER's revisions. Wogulis and Pazzani (1993) proposed a methodology to measure the syntactical distance between theories, but it is hardly applicable to any reasonably sized KB. Given a concept, let $C1$ be the set of rules defining the concept in one theory and $C2$ the set of revised rules for the same concept in the revised theory. Calculating Wogulis and Pazzani's measure involves computing the Cartesian product of the two sets (of size $|C1| \times |C2|$), and then the power set of the Cartesian product, $2^{|C1| \times |C2|}$. Each element in the power set is a possible mapping between the original and revised concept, and the distance (in terms of addition, deletions, and replacements of rule elements) between the clauses in the mapping must be calculated. In the case of the Promoter theory, for instance, STALKER refined the original 3 rules for concept minus_10 into a set of 20 rules. Hence, the Cartesian product of the sets of original and refined rules for this concept contains 60 elements and so the power set is composed of 2^{60} items!

4.1.2. Accuracy results in the soybean domain. The Soybean KB (Michalski & Chilausky, 1980) discriminates between 15 of 19 possible soybean diseases using examples that are described with 35 features. The original Soybean KB has a certainty-factor associating probabilistic weights with certain disease symptoms. For STALKER we used a version of the KB originally used to test EITHER, where the rules were translated to propositional Horn-clause form by deleting any symptoms from the theory that had a weight less than 0.8. The resulting KB contained 50 rules. This KB obtained a 12.3% classification performance compared with the accuracy of 73% reported in the original paper.

Figure 11 shows the testing accuracy for STALKER, KRUST, EITHER, and RAPTURE. The results for EITHER and RAPTURE were taken from Mahoney and Mooney (1993). (Note that EITHER has only been run with up to 100 training examples, and to our knowledge NEITHER and PTR have not been tested on this domain.) All the systems were evaluated using 75 test examples. As can be seen, STALKER, EITHER, and RAPTURE perform at comparable levels of accuracy. However, since RAPTURE was specifically designed to deal with certainty-factor KBs, it had the considerable advantage of using the original certainty-factor KB as its starting point. EITHER's results, on the other hand, were obtained in two different ways: with a standard tester, and using a partial-matching technique in order to approximate the behaviour of the original certainty-factor KB. This "flexible" tester accounts for two possible classification problems with the refined theory generated by EITHER. If an example is assigned to multiple categories, the flexible tester selects the category that makes the most use of the example's features. If an example is assigned to no category, the flexible tester chooses the category that comes closest to being

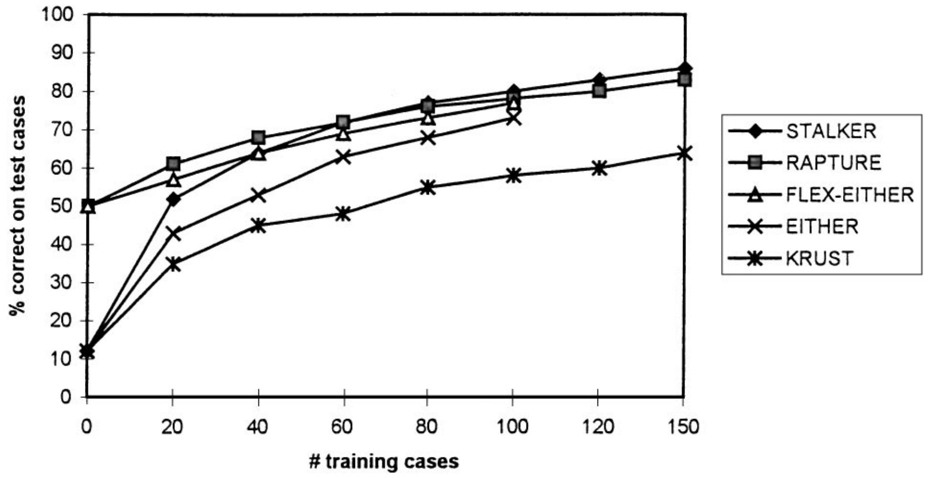


Figure 11. Soybean domain testing accuracy.

provable. With its standard tester, the average testing accuracy obtained by EITHER with 100 training examples was 72%.

As mentioned earlier, STALKER's current tester assumes rule ordering as the conflict-resolution strategy used by the KBS. Hence, if an example is assigned to multiple categories the category represented by the rule with the higher priority wins the conflict resolution. However, if an example is assigned to no category, no other attempt is made to classify it. Again, STALKER obtained very good results without being tailored to deal with the peculiarities of the domain.

In this domain KRUST performs much worse than the other systems. Most of the 19 diseases are relatively easy to diagnose. In fact, examining one or two key features is sufficient to classify the examples for these diseases. However, three diseases, Brown Spot, Alternaria Leaf Spot, and Frog Eye Leaf Spot, are very closely related. In fact, the features cited in the initial KB are *not* sufficient to discriminate between these categories. Hence, STALKER, EITHER and RAPTURE use induction extensively to learn new features; KRUST does not have this functionality and this explains its relatively poor performance in this domain.

Additionally, the initial knowledge base does not contain any rules for four of the diseases, so again induction is needed to learn completely new rules for these categories. Since STALKER augments KRUST's set of refinement operators by using inductive operators, this result gives us an indication of the part that induction can play in STALKER's performance. In this particular domain, it appears that the 33% accuracy gain obtained by STALKER over KRUST is due to the use of induction; see Section 4.3 for a more detailed discussion. (We do not include here the original and refined Soybean theories for lack of room.) As already pointed out, the original theory used by STALKER was composed of 50 rules. A typical refined theory comprised 90 rules. As in the Promoter domain, most of the new rules were variants of existing rules with generalized discrete attributes. If the rule representation used by the system were augmented by adding the set operator 'internal disjunction', refined theories could be noticeably reduced in size. Internal disjunction is usually denoted by 'in'. The expression '*colour in* {*red, blue*}' means that the value of attribute *colour* must be one of {*red, blue*}.

4.1.3. Time performance in the DNA and Soybean domains. Let us consider the timings of the systems being compared. Unlike STALKER and KRUST, which have been developed in the same environment, EITHER, NEITHER, and RAPTURE were tested on different platforms.[5] Hence, a direct comparison of the various systems' timings was not possible. Nevertheless, Tables 4 and 5 give a summary of timings for different numbers of cases. As discussed earlier, STALKER and KRUST obtained the same accuracy results on the Promoter Recognition domain since induction was not used. Hence, the real difference in the performance of the two systems lies in their execution times. Note that in these experiments we did not use a separate set of priority cases (see Section 2) to test KRUST. Instead, each refinement was tested against the whole training set. This not only coincides with the technique adopted by STALKER, but also corresponds to the method for using KRUST iteratively described by Craw and Hutton (1995).

As can be seen in Table 4, STALKER takes about 56 (cpu) seconds to process the entire training set, while KRUST needs almost 20 minutes to perform the same task. Hence,

Table 4. DNA domain training time (seconds).

	90 Training cases
STALKER	56
KRUST	1120
NEITHER	6
RAPTURE	20
EITHER	135

Table 5. Soybean domain training time (seconds).

	90 Training cases	150 Training cases
STALKER	100	200
KRUST	6800	10500
RAPTURE	1400	3000
EITHER	2250	—

STALKER was more than 20 times faster than KRUST in this domain. Note that NEITHER and EITHER work in batch mode, so they process the entire training set just once, and generate a set of refinements that correct all the training examples. Hence, the system that presents more similarities to STALKER's algorithm in the way training examples are processed is RAPTURE. In fact, in this connectionist system the network representation of the initial knowledge base is trained with one example at a time until all the training cases are processed. If after this first epoch the trained network has still not reached 100% accuracy on the training set, the process is repeated until no further improvement is achieved. Note that PTR's timings were not available.

Table 5 summarizes the timing results for the Soybean domain. In this domain, the difference between STALKER's and KRUST's performance is even more evident. STALKER takes on average 200 seconds to process 150 training examples, while KRUST needs almost 3 hours. Hence, in these particular trials STALKER is about 52 times faster than KRUST. (Above we have seen that STALKER was 20 times faster than KRUST in the experiments with the DNA domain.) It is suggested that the difference in the performance improvement obtained by STALKER in the two domains is due to the size of the problem. The DNA theory is composed of 11 rules, and the experiments were carried out using 90 training examples. The Soybean theory, on the other hand, is composed of 73 rules, and 150 training examples were used in each trial. As the size of the KB increases KRUST needs more time to solve a task, and some component of this time is directly proportional to the number of examples. Hence, the efficiency gain produced by STALKER's testing algorithm is more visible as the size of the KB and the number of examples increases.

Although, as explained above, a direct comparison between STALKER, RAPTURE and EITHER is not possible, some comments can be made on the *relative* difference of each system's timings in the two domains studied. As seen in Table 4, RAPTURE and

EITHER take respectively 20 and 135 seconds to process 90 training examples in the DNA experiments. On the other hand, for the same number of examples in the Soybean experiments their training time goes up to 1400 and 2250 seconds, respectively. Hence, for the Soybean domain RAPTURE takes 70 times the time needed for the DNA domain, while EITHER is 16 times slower. STALKER, on the contrary, takes about 56 seconds to process 90 training cases in the DNA domain, and 100 seconds to process the same number of cases in the Soybean domain.

Hence, although the Soybean KB is much bigger than the DNA KB, STALKER takes less than twice the time to process the same number of examples.[6] On the other hand, EITHER's time increases considerably as the number of rules in the KB increases, while the time taken by RAPTURE seems to be highly dependent on both the size of the KB and the number of examples in the domain. Therefore, STALKER's performance seems to deteriorate more slowly than the other systems' as the size of the knowledge base increases. Note that a worst-case analysis (Carbonara, 1996) concluded that the maximum theoretical complexity of the TMS propagation algorithms is of order n^3, where n is the number of cases represented in the dependency network. On the other hand, the experimental results presented in the next section suggest that the actual performance of the Tester Simulator is of order n^2.

4.2. Some more detailed timing experiments

STALKER's time performance is determined by:

- the time taken to generate the refinements for each training case;
- the time taken to test the refinements.

Multiple refinement generation is potentially exponential in the size of the KB. Nevertheless, for relatively 'shallow', approximately correct theories, this has not proven to be a significant bottleneck in practice.

Concerning the testing of alternative refinements, a theoretical analysis (Carbonara, 1996) suggested that the tester simulator's time performance is mainly determined by the total number n of examples represented in the TMS network. In particular, the worst-case complexity for the TMS algorithms when making a single change to the network is $(\mathsf{A}+\mathsf{D})\cdot\mathsf{B}\cdot\mathsf{C}\cdot n^3$, where A is the average number of antecedents in a justification, B is the average number of times a justification's consequence appears in some other justification's premise, C is the maximum length of justification chains, D is the average number of justifications which have the same consequence, and n is the number of examples represented in the network. However, this figure is likely to grossly over-estimate the actual cost of the TMS algorithms since the assumptions on which the analysis is based are far more pessimistic that any realistic situation. Further, this analysis indicates that the TMS system is not affected by the overall size of the KB. This is due to the fact that the TMS algorithm works locally on the parts of the network being altered. Hence, although, as the formula above shows, the size of the justifications and the depth of justification chains contribute to the algorithm's complexity, this is independent of the total number of rules represented in the network. Note that this is not the case for a standard inference engine, which potentially would have to match each example against each (endpoint) rule after a change is implemented on the KB. In this section, we discuss the results of some experiments performed to study empirically STALKER's computational requirements. The experiments were designed to:

- see how STALKER's time performance varies with increasing number of examples;
- test the hypothesis that the computational cost of STALKER's tester simulation algorithms is independent of the size of the KB (note again that, although refinement generation is heavily dependent on the size of the KB, we are here referring to the time performance of STALKER's Tester Simulator only);
- see how the number of refinements generated affects the system's time performance.

In the first set of experiments, the system was tested with increasing numbers of examples in both the DNA and the Soybean domain to see how the execution times varied. The results of these tests are presented in Tables 6 and 7. These tables show the number of training examples used in the experiments, and for each example set give details of the time taken to process the examples and the total number of refinements generated. All the values have been averaged over 20 trials. Note that the training examples for the various experiments were chosen so as to generate the desired number of refinements. The more incorrectly classified cases are included in the training set, the more refinements are produced. This explains why, in Table 6, different numbers of refinements were generated for two sets of 60 cases. The results for the DNA domain indicate that for small example sets the time taken by STALKER's algorithms is linear in the number of refinements generated. In fact, in Table 6 it can be seen that, varying the number of refinements generated from 23 to 225 (rows 1 and 3), the execution time also increases one order of magnitude. Similarly, doubling the

Table 6. Timings of experiments in the DNA domain.

Total number of training cases	Time (sec)	Total number of refinements
6	5	23
60	9	53
60	56	225

Table 7. Timings of experiments in the Soybean domain.

Total number of training cases	Time (sec)	Total number of refinements
6	1	8
60	11	51
225	200	245
600	1,200	402

number of refinements produced (from 23 to 53) causes the processing time to double too, irrespective of the fact that the number of examples represented in the dependency network was increased one order of magnitude. This confirms that in this case the processing time is solely dependent on the number of refinements produced. Unfortunately, the DNA dataset was not big enough (106 examples in total) to allow us to carry out more extensive tests.

Let us consider now the experiments in the Soybean domain. As can be seen in Table 7, with small sets of examples the time taken by STALKER increases linearly. In fact, when the number of examples increases from 6 to 60, the execution time also experiences a tenfold increase. This behaviour, though, is not maintained when the number of examples rises further, as increasing from 60 to 600 examples produces a 100-fold increase in execution time. Hence, the behaviour of the system in this range seems to be quadratic in the number of examples. This is also confirmed by the intermediary result with 225 examples. This overall empirical evidence substantiates this hypothesis, and gives a more realistic prediction of STALKER's performance, namely that STALKER's overall processing time is quadratic in the number of cases used.

The second point we set out to demonstrate is that STALKER's label propagation algorithms are independent of the size of the KB. Evidence on this point can be gathered by comparing the results in the second rows of Tables 6 and 7. In both these experiments the system was used with 60 examples and generated a similar number of refinements (53 for the DNA domain, and 51 for the Soybean domain). Hence, in these tests the only variable is the size of the KB (the Soybean KB is significantly bigger than the DNA KB). Nevertheless, STALKER's execution time in both domains is nearly the same (9 seconds for the DNA domain, and 11 seconds for the Soybean domain), suggesting that the performance of the system is not affected by the size of the KB.

We presented above experimental evidence supporting the hypothesis that the Tester Simulator's time performance is mainly determined by the number of examples processed. In fact, the Tester Simulator's algorithms seem to be quadratic in the number n of examples processed, and independent of the size of the KB. We now intend to study how the third major parameter in our problem, i.e., the number R of refinements produced for a given training set, contributes to STALKER's computational cost. Conceptually, the cost S of STALKER's algorithms can be described as a function of the two parameters n and R. In order to see the effect of R on S, it is sufficient to maintain n constant, and vary the number of refinements R. Provided refinement generation is effected in linear time, the cost of testing the refinements is expected to be proportional to the number of refinements generated.

Hence, we carried out a set of experiments (in both the DNA and the Soybean domains), where we kept the size of the training set constant, and varied the proportion of wrongly classified cases to produce increasing numbers of refinements. For the DNA domain, we employed a total of 60 training examples, while for the Soybean domain 225 examples were used. The results of these experiments are shown in figures 12 and 13. As can be seen from figure 12, in the DNA domain STALKER gives essentially a linear fit.

In the Soybean domain, STALKER's execution time first grows very slowly, then increases sharply after about half of the training cases are processed, and finally slows down

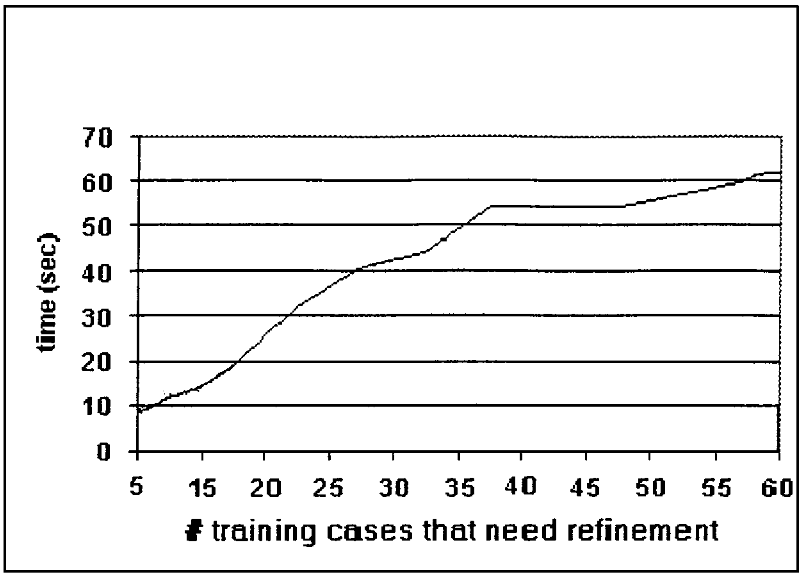


Figure 12. STALKER's time performance in the DNA domain with a total of 60 training examples.

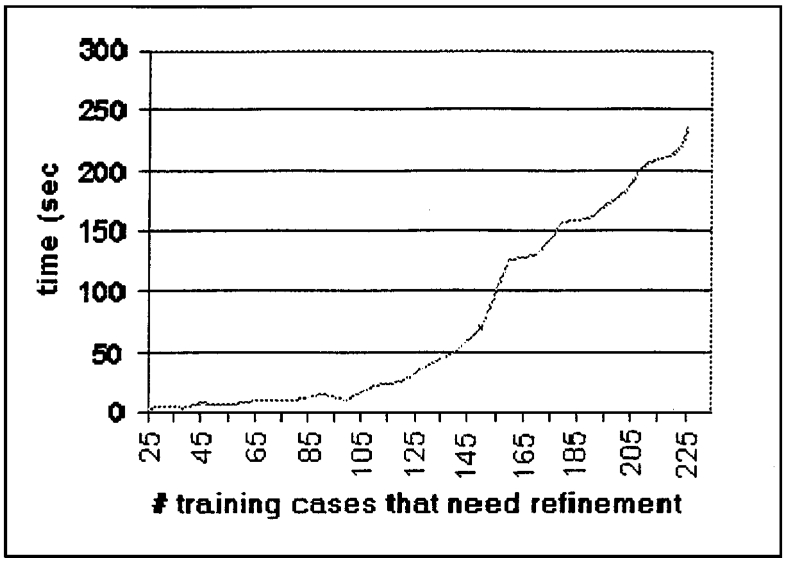


Figure 13. STALKER's time performance in the Soybean domain with a total of 225 examples.

again. The sharp rise is a consequence of the single-example refinement strategy used by the system combined with the way STALKER currently handles the generalisation of discrete attributes. When discrete attributes are refined, the number of rules in the knowledge base increases because, as explained in Section 4.1.1, refinements generalising discrete attributes will introduce new rules. As the number of rules increases, so does the amount of alternative refinements generated for subsequent training examples, and the time needed to generate and test them. In the future, we plan to obviate this problem by changing the way discrete attributes are treated. As already discussed at the end of Section 4.1.2, this can be achieved by introducing the set operator 'internal disjunction'. Note that inductive refinements may introduce new rules as well. However, STALKER's refinement strategy ensures that the number of inductive refinements is kept to a minimum.

To summarize, experimentally the time STALKER's Tester Simulator takes is:

- proportional to the square of the number of examples processed;
- linearly dependent on the number of refinements generated;
- independent of the size of the KB, i.e., the total number of rules does not influence the Tester Simulator's execution time.

4.3. The relevance of STALKER's refinement operators

In this section we study how the refinement operators used by STALKER affect the system's performance. Remember that in STALKER the set of refinement operators employed by KRUST[7] was augmented by adding inductive operators to create new rules and antecedents. We will now discuss which refinement operators have been used in the two domains examined, and discuss experiments designed to analyse the role played by each subset of operators. We start by illustrating the results obtained in the DNA domain. As already mentioned in Section 4.1, STALKER did not use inductive operators to refine the DNA KB, since for each training case STALKER was always able to generate at least one acceptable refinement which did not cause new inconsistencies. Additionally, in these experiments, no use was made of the operators which change the priority of rules. This is explained by the fact that the DNA domain is a single category domain. All the rules in the KB contribute to classifying promoter sites in DNA strands. If an example is not recognized as a promoter, it is assigned by default to the negative category. The Rule Priority Change operators are used to swap the position of (Potentially) Error Causing rules and (No) Can Fire rules to make the latter fire instead. Since there are no (Potentially) Error Causing rules in this KB, the Rule Priority Change operators are never invoked. Hence, in this particular case, generalization and specialization operators, coupled with multiple refinement generation, were sufficient to effectively revise the KB for all the examples presented.

The experiments conducted in the Soybean domain allow one to get a better picture of the influence of each of the operator types on the accuracy of the revised KB. As opposed to the DNA domain, the Soybean domain is a multiple category domain. To refine the Soybean KB both Rule Priority Change operators and inductive operators were used. Table 8 shows the role of STALKER's refinement operators in this set of experiments. Note that the Rule Priority Change and Inductive operators could not be used independently from the Generalization and Specialization operators. In fact, in this domain STALKER always generated rule priority changes in conjunction with generalization operations (i.e., refinements designed to make No Can Fire rules fire), hence it was not possible to experiment with Rule Priority Change operators alone. Similarly, induction is only triggered if all the other refinements cause new inconsistencies, so it can only be used together with other operators.

Table 8. The role of STALKER's refinement operators in the Soybean experiments.

Generalization and specialization	Rule priority change	Induction	Accuracy (%)
√	√	√	86
√	—	√	81
√	√	—	64
√	—	—	55

The second row shows the results obtained when Rule Priority Changes are disabled. As can be seen, the resulting KB is on average 5% less accurate that the one obtained with the full set of refinement operators. On the other hand, using only Generalization, Specialization, and Rule Priority Change gives us an accuracy of 64%, i.e., the resulting KBs are on average 22% less accurate than the KBs obtained by also using induction. Finally, Generalization and Specialization alone produce an accuracy of only 55%

4.3.1. Suggested interpretations. From these results it appears that:

1. All the subsets of refinement operators contribute to the improvement of the initial Soybean KB.
2. The most sizeable contribution is produced by STALKER's new inductive operators. Their use allows the KB accuracy to rise from 64% (i.e., the accuracy obtained with generalization and specialization operators alone) to 86%.
3. Also Rule Priority Change operators play an important role in the refinement of the Soybean KB (increase from 55% to 64% with no induction, and from 81% to 86% with induction). Note that conceptually Rule Priority Change operators only perform operations that could be achieved by Generalization and Specialization operators alone. In fact, these operators aim at preventing (Potentially) Error Causing rules from firing by moving them below (No) Can Fire rules, and at making (No) Can Fire rules fire by moving them above the (Potentially) Error Causing rules. Although, in theory, the same result could be achieved by specializing the (Potentially) Error Causing rules, in the Soybean domain it seems that these refinements tend to over-specialize the (Potentially) Error Causing rules, hence causing new inconsistencies. Rule Priority Change operators, on the other hand, appear to be less prone to this problem. This result appears to be consistent both with and without induction. If induction is not used, Rule Priority Change operators improve the accuracy obtained with Generalization and Specialization operators by 9%. If induction is used, the further improvement achieved by Rule Priority Change operators is about 5%.

These points lead to a main consideration. It appears that the new inconsistencies generated by the Generalization and Specialization operators are more effectively restored when induction is used in conjunction with Rule Priority Change operators. This seems to confirm that the conflict-resolution strategy adopted by the inference engine is an important aspect which can be exploited to substantially improve the quality of refinements.

4.4. *The effect of presentation order on the accuracy of the refined KB*

In this final section we present the results of some experiments aimed at understanding how the order in which training examples are processed by STALKER affects the final outcome of the refinement process. We decided to order the examples in the training sets in three different ways:

- in the same order as the endpoint rules in the KB (i.e. if the endpoint rule for disease X appears before the rule for disease Y in the KB, then the examples for disease X appear before those for disease Y in the training set);
- in reverse order to the endpoint rules in the KB;
- in random order.

In the experiments with the DNA domain, our tests showed that the order in which examples are presented to STALKER does not have any discernible effect on the accuracy of the final refined KB. Indeed, the results obtained in all three experiments are not significantly different to those reported earlier in Section 4.1.

Let us consider again the experiments conducted in the Soybean domain. As explained in Section 4.3, the initial Soybean KB contains rules for 15 of the 19 disease classes. The experiments were carried out with 150 training examples and 75 test examples. The results, which have been averaged over 20 trials, are shown in Table 9. As can be seen, the results with reverse and random order are the same, and they differ by only one percent from those obtained with "direct" order. The small variation in accuracy is explained by the fact that, if examples are processed in reverse or random order, there are more possibilities that the refinements generated create new inconsistencies. In fact, if the examples corresponding to rules at the top of the KB are accommodated first, no subsequent changes to lower priority rules can affect their classification. On the other hand, if the examples corresponding to rules at the bottom of the KB are processed first, then any change to higher priority rules can in principle affect them (if the higher priority rules are over-generalized). A similar problem can occur when the order of the examples in the training set is random. Note that changing the order of the examples has an insignificant effect on the time taken to process the training set. STALKER changes the KB incrementally, examining one example at a time. If the order in which the examples are presented to the system is varied, overall a slightly different set of refinements will be generated since the refinements implemented for each example will interact in a different way. Nevertheless, the experiments showed that, irrespective of the order of the training examples, on average the total number of refinements produced remained the same, and consequently the processing time did not change either.

Table 9. Training examples order and KB accuracy in the Soybean domain.

"Direct" order	"Reverse" order	Random order
86%	85%	85%

To conclude, we can say positively that the order of training examples does not have any significant effect on STALKER's accuracy performance. When refining a single category KB this effect is almost null. With multiple-category KBs it is *slightly* preferable to process the examples in the same order as the endpoint rules, since, as the Soybean experiments suggest, this reduces the need for inductive refinements.

5. Related research

In several places in this paper, the contrast between batch, heuristic refinement generation and STALKER's single example, multiple refinement generation has been stressed. Among the advantages of the multiple refinement generation approach are:

- its greater ability to focus on the cause of failure of each training case, and propose ad hoc corrections; and
- a greater chance to find a "good" repair (i.e., a refinement which improves the accuracy of the KB without causing new inconsistencies), given alternative corrections.

Note that systems adopting batch, heuristic refinement generation can also perform some search. For instance, EITHER (Ourston & Mooney, 1990), a typical representative of the batch approach, firstly finds a minimal set of failure points, i.e. places in the KB that, if changed, would correct the classification of the entire training set; then, for each failure point the system accepts the first refinement that succeeds. Hence, in EITHER alternatives are only explored when a refinement fails to produce the desired effect.

A comparison between EITHER and STALKER on two different domains confirmed the superiority of the latter (see Section 4.1). Interestingly, in the Promoter Recognition domain, neither STALKER nor EITHER used induction, indicating that STALKER's superior performance was due to multiple refinement generation alone. In the other domain, the Soybean domain, induction played a crucial role in STALKER's performance, confirming the hypothesis that the introduction of inductive operators can allow the system to correct a broader class of errors. An enhanced version of EITHER, NEITHER (Baffes & Mooney, 1993) obtained similar results to STALKER's in the Promoter Recognition domain. This, though, was due to NEITHER's ability to revise theories having the M-of-N property (possessed by the Promoter Recognition domain). This in turn encouraged us to conclude that multiple refinement generation is also a very general and flexible approach since it allowed STALKER to perform at comparable levels to NEITHER, in spite of not having any special provision to cope with the particular features of the Promoter domain.

FORTE (Richards & Mooney, 1995) is probably the most similar system to KRUST and STALKER. FORTE employs a hill-climbing approach to revise theories iteratively. The current theory is tested on a training set of positive and negative examples. During this process, the theory is fully annotated by generating revision points, i.e., taking note of rules, antecedents and predicates that contributed to wrongly classifying the given examples. Revision points are then sorted by their potential, which is defined as the number of examples which caused the point to be marked. For each revision point, FORTE generates a set of possible revisions which are scored according to the accuracy gain they produce. This process terminates when the potential score of the next revision point in the sorted list is less than the score of the best revision found until then. The best revision is implemented and the whole process is triggered again until no revision improves the theory.

The main difference between STALKER and FORTE is that, while the former considers, at any time, a single failing example, the latter processes the whole set of training examples, and generates the revision points for all of them, sorted according to their potential. FORTE then produces all the possible refinements for the first revision point and if the accuracy gain of the best refinement is greater than the potential of the next revision point it implements the refinement and iterates. Hence, in general, at each iteration only a subset of all the possible refinements for all examples is actually generated and tested by FORTE. On the other hand, STALKER at each step only considers the revisions for a single example, but generates and tests all of them. Which of the two systems explores the biggest space is dependent on the characteristics of the particular domains. Another difference between the two systems is that FORTE refines First-order theories, and is therefore more suitable for the refinement of logic programs rather than expert system KBs. To date, FORTE has only been tested on relational domains, so a direct empirical comparison between FORTE and STALKER was not possible.

Another system generating alternative refinements is SEEK2 (Ginsberg, 1988a). The system first obtains a performance evaluation of the initial KB on the training cases. This consists primarily of an overall score, e.g. 75% of cases for a given category are classified correctly. The categories are then sorted in ascending order according to the score, starting from the category with the lowest rate of correctly classified examples. After identifying the rules which are involved in the wrong proof of the cases for the current category, the system computes a number of statistical properties for these rules, which are subsequently used to guide the refinement process. For instance, the function Gen(rule) computes the number of cases in which (a) a rule's conclusion should have been reached, but was not, (b) had the rule been satisfied the conclusion would have been reached, and (c) of all the rules for which the preceding clauses hold in this case, this one is the "closest to being satisfied". This statistical information is then used to compose heuristic meta-rules which suggest how to refine the rules. As several meta-rules can be fired by the statistical evidence collected, alternative refinements can be generated. Each refined KB is tested over all the cases and the system keeps a record of the refined KB that produced the maximum net performance gain. The procedure is reiterated for all categories, and the refinement that gives the greatest net gain in performance for all the categories is accepted.

Like FORTE, we believe SEEK2 is in general searching a different solution space to STALKER's. SEEK2 collects statistical evidence for rule generalization and specialization and then uses this evidence to heuristically generate refinements. Hence, not all the possible refinements (but only those suggested by the system's heuristics) are generated and tested. Moreover, SEEK2 does not have any operators to add new components to rules or new rules to the KB, hence the portion of the solution space it can explore is inherently smaller than STALKER's.

For a discussion of the advantages and disadvantages of FORTE and SEEK2's approach see the introductory paragraphs of Section 3.

Another approach to theory revision combines symbolic and connectionist techniques to refine propositional knowledge bases. KBANN (Towell & Shavlik, 1992) uses a knowledge base of hierarchically-structured rules which may be both incomplete and incorrect to form an artificial neural network. During the conversion of the rules into a neural network, all antecedents for a particular consequent literal become nodes that are linked into the input of the consequent node. Also, all features that are not mentioned in the rules are added into the network with low weighted links. All neighbouring layers are fully connected. This network is then trained with backpropagation, and the rules retranslated into symbolic form.

A similar system, RAPTURE (Mahoney & Mooney, 1993), maps certainty-factor rules into an equivalent neural network in which certainty factors become weights of connections between nodes of the network. A modified version of backpropagation is then used to refine certainty factors of existing rules, and ID3's information gain heuristic (Quinlan, 1986) is used to add new rules. In RAPTURE, the direct correspondence between weighted links and probabilistic rules removes any distinction between the symbolic and connectionist representations. Hence, there is no need to map the network back into rules. KBANN and RAPTURE obtained similar results on the Promoter domain. STALKER's opportunistic approach to suggesting possible refinements is somehow comparable to the incremental reward achieved by backpropagation in neural networks. It is therefore not surprising that STALKER obtained very similar accuracy to RAPTURE's on both the domains considered.

The idea, common to KBANN and RAPTURE, of taking a knowledge base in one representation and translating it into another representation which is more amenable to revision is also exploited by the TGCI system (Donoho & Rendell, 1995). In this system, a constructive induction algorithm is used to re-describe the training and testing examples by extracting new features based on the components of the initial theory. The constructed features need not be expressed in the same representational language as the initial theory. Finally, a standard inductive learning algorithm, C4.5, is applied to the re-described examples. An empirical comparison showed that TGCI outperformed KBANN and NEITHER on the Promoter domain. However, as the final theory is based on the new features extracted by the constructive induction algorithm, this system contravenes quite dramatically the minor tweaking assumption described in the introduction. In fact, it can be argued that, as TGCI uses the initial theory only to extract new features, it cannot be considered a refinement system, but rather an inductive algorithm exploiting some form of background knowledge to improve the learning process. A similar argument holds for systems like REGENT (Opitz & Shavlik, 1994) and DOGMA (Hekanaho, 1996) which use a genetic algorithm (GA) to search through the space of possible KBs. The initial theory is used to generate a population of KBs which is then evolved by the GA. This method allows the systems to explore a large solution space, but, as the mutation and crossover operators used by the GA make random modifications to the KBs, the final refined KB can be practically unrelated to the initial theory.

STALKER's most distinguishing feature is its use of a TMS to speed up the evaluation of alternative refinements. Different kinds of TMSs have already been used in KBS research for a number of different purposes. For instance, Ginsberg (1988b) used an Assumption-based TMS, the ATMS (de Kleer, 1986), to check KBs for inconsistency and redundancy, while Zlatareva (1994) employed a Contradiction-tolerant TMS (Popchev, Zlatareva, and

Mircheva, 1990) to develop a general framework for verification, validation and refinement of KBs. Both these approaches use the TMS version of the KB to detect anomalies in the KB, while the refinement process is controlled by the human expert who is responsible for actual modifications of the KB. STALKER, on the other hand, automatically generates the refinements, and uses the TMS to efficiently *test* them.

PTR (Koppel, Ronen, & Segre, 1994) is another refinement system which uses a graph representation of the domain theory. Each edge in the graph is assigned a weight which represents the user's degree of confidence that the edge does not need to be deleted to obtain the correct domain theory. In the absence of such information, values can be assigned by default. PTR processes training examples one at a time, in each case updating the weights associated with edges in accordance with the information contained in the examples. The more an edge contributes to the correct classification of an example, the more its weight is raised; and vice versa. If the weight of an edge drops below a pre-specified threshold, it is revised, where revisions can consist of deleting the edge, or adding children to a node. Nodes are divided into clause nodes representing the conditions of a rule, and proposition nodes representing the conclusion of a rule. Adding children to a clause node specialises it by adding conditions to its body, while adding edges to a proposition node generalises it by adding clauses to its definition. PTR uses ID3 to determine the sub-trees to be added to a graph. PTR obtained 90% accuracy on the Promoter domain, hence showing a comparable performance to STALKER on this domain. Others similarities between the two systems are the use of a graph representation of the domain theory (although PTR exploits this to generate the revisions, it does not use it to efficiently test the refinements), and the way they process training examples (one at a time). However, unlike STALKER, PTR does not have operators to revise ranges for numerical, list-structured, or tree-structured attributes, hence it is applicable to a smaller class of problems.

The First-order knowledge revision tool KRT (Wrobel, 1994) also performs reason maintenance when part of the knowledge base is changed. More specifically, derivation traces are used during specialization to compute *minimal removal sets*, i.e. minimal sets of clauses the removal of which would be sufficient to prevent the incorrect derivation. This is achieved by using the IM-2 inference engine (Emde, 1989), which is KRT's knowledge representation and maintenance component. However, IM-2 is not able to deal with assumptions as general truth maintenance systems and STALKER do. In IM-2, facts and rules which "belong together" can be organised into "worlds". Hence, multiple examples can presumably be represented by creating a number of worlds, one for each example, which share the same rules, while differing in their facts. It is not clear whether this mechanism can support the *efficient* representation and testing of multiple examples, particularly when large numbers of examples are considered, as this would involve the creation of an equally large number of worlds. Overall, IM-2's worlds seem to be better suited to representing multiple theories, a feature which, on the other hand, is not supported by STALKER.

6. Conclusions and future research

This paper has described and evaluated an approach to refining propositional rule bases that integrates symbolic learning and truth maintenance. The approach is implemented in a system called STALKER, which uses a TMS-based technique to efficiently test the set of alternative refinements generated for each incorrectly-solved training case.

Tested on a medium-sized domain, STALKER proved to be essentially 50 times faster than its predecessor, KRUST. An empirical comparison suggested that STALKER may be better suited to the refinement of large-size KBs than other state-of-the-art systems. This comparison also showed that STALKER performs at comparable, or higher, levels of accuracy with alternative state-of-the-art systems tested on the selected domains. Furthermore, unlike the other systems, STALKER did not need any special provisions, such as the M-of-N representation, to cope with the peculiarities of the domains considered. As already pointed out in Section 4.1.1, it has been shown that the original Promoter theory is very accurate provided that its rules are given a numerical interpretation, and therefore RAPTURE and NEITHER's performance on this domain is not a consequence of learning from examples, but rather a side effect of the use of numerical rules. In this domain, STALKER was the best of the 'non-numerical' algorithms considered. In the Soybean domain, STALKER obtained slightly better results than RAPTURE, in spite of the fact that this system, being specifically designed to deal with certainty-factor KBs, had the considerable advantage of using the original certainty-factor KB as its starting point. This is a very important result as it shows that STALKER's approach is successful in a wide range of different types of KBs.

STALKER's new inductive operators allowed the system to increase KRUST's accuracy performance on the Soybean domain by over 20%. On the other hand, the results obtained in the DNA domain demonstrated that, in some cases, multiple refinement generation can be a powerful technique on its own. More specific tests, in both the domains selected, clarified to what extent each class of refinement operators used by STALKER contributed to improving the accuracy of the KBs; further, these experiments demonstrated that in these domains STALKER's performance was independent of the order in which training examples were processed.

In the future we wish to investigate several ways to improve STALKER. Although the generation of alternative refinements gives more chances of finding good refinements, it also increases the computational load of the algorithm. Hence, we are interested in developing methods for constraining the search for possible refinements to further speed up our algorithm. We hope by means of Constraint Satisfaction techniques to detect and discard unpromising refinements without testing them. In particular, we plan to exploit the inherent constraints present in the examples to guide the selection of refinements.

Further, during the evaluation of the system we noted that, for a given error, often several alternative refinements are generated which yield the same testing accuracy. Currently, STALKER randomly selects one of these refinements. We plan to experiment with more sophisticated selection criteria to see how they affect the final refinements proposed. One solution could be to select refinements according to some minimality metric (e.g. syntactical simplicity). Alternatively, the rules in the initial theory could be weighted according to the domain expert's confidence in their correctness, and refinements changing these rules could be penalised. Another possibility would be to use a linguistically-based semantic bias similar to the one used in CLARUS (Brunk & Pazzani, 1995) to discard revisions which combine terms that are not meaningful when used together.

Finally, we would like to embed STALKER in a widely available expert system shell so as to make refinement part of the life-cycle of real-world knowledge based systems.[8] This would also imply adapting STALKER to cope with the conflict-resolution strategie(s) of the various expert system shells.

Acknowledgments

KRUST was implemented at the University of Aberdeen by Susan Craw. The STALKER work was supported by a grant from the University of Aberdeen. The authors wish to thank the anonymous reviewers for their invaluable comments and suggestions. Thanks also to James Mace, Huw Roberts, Simon White and Mark Winter for their comments on earlier drafts of this paper.

Notes

1. The test examples are actually loaded onto the TMS together with the training cases in order to minimise I/O. However, they are tagged as test cases and only used at the end of the refinement process to evaluate the final refined KB.
2. See the first paragraph in Section 3 for a definition of new inconsistency.
3. See Section 2, and Table 1, in particular, for a description of the refinement operators.
4. Remember that the reference for deciding new inconsistencies is the initial mis-classified theory, which is illustrated in figure 3.
5. STALKER and KRUST are implemented in Common Lisp and were run on a Sparc IPX. EITHER, NEITHER and RAPTURE are also implemented in Common Lisp, but were run on a Sparc 5.
6. Consider that the DNA domain is entirely over-specific, hence half of the training cases (the negative cases) do not need refining. Therefore, STALKER generated refinements for 45 of the 90 cases considered.
7. See Table 1 in Section 2.
8. This is also being done for STALKER's predecessor KRUST (Boswell, Craw, & Rowe, 1997).

References

Baffes, P. T., & Mooney, R. J. (1993). Symbolic revision of theories with M-of-N rules. *Proceedings of the Thirteenth International Joint Conference in Artificial Intelligence* (pp. 1135–1140). Chambery: France.

Bairess, R., Porter, B. W., & Murray, K. S. (1989). Supporting start-to-finish development of knowledge bases. *Machine Learning, 4,* 259–283.

Boose, J. H., & Gaines, B. R. (1989). Knowledge acquisition for knowledge-based systems: Notes on the state-of-the-art. *Machine Learning, 4,* 377–394.

Boswell, R., Craw, S., & Rowe, R. (1997). Knowledge refinement for a design system. *Proceedings of the European Knowledge Acquisition Workshop* (pp. 49–64). Springer.

Brunk, C., & Pazzani, M. (1995). A Linguistically-based semantic bias for theory revision. *Proceedings of the Twelth International Conference on Machine Learning*. Lake Tahoe: California.

Carbonara, L. (1996). Improving the effectiveness and the efficiency of knowledge base refinement. PhD Thesis, University of Aberdeen, Aberdeen: Scotland.

Carbonara, L., & Sleeman, D. H. (1996). Improving the efficiency of knowledge base refinement. *Proceedings of the Thirteenth International Conference on Machine Learning* (pp. 78–86). Bari: Italy.

Craw, S., & Hutton, P. (1995). Protein folding: Symbolic refinement competes with neural networks. *Proceedings of the Twelfth International Conference on Machine Learning* (pp. 133–141). Lake Tahoe: California.

Craw, S., & Sleeman, D. H. (1990). Automating the refinement of knowledge-based systems. In L. C. Aiello (Ed.), *Proceedings of the Ninth European Conference on Artificial Intelligence* (pp. 167–172). Stockholm: Sweden.

Craw, S., Sleeman, D. H., Boswell, R., & Carbonara, L. (1994). Is knowledge refinement different from theory revision? In S. Wrobel (Ed.), *Proceedings of the MLNet Familiarization Workshop on Theory Revision and Restructuring in Machine Learning* (at ECML-94, Catania, Italy). Arbeitspapiere der GMD, GMD, Pf. 1316, 53754 Sankt Augustin: Germany.

de Kleer, J. (1986). An assumption-based TMS. *Artificial Intelligence, 28*, 197–244.

Donoho, S. K., & Rendell, L. A. (1995). Rerepresenting and restructuring domain theories: A constructive induction approach. *Journal of Artificial Intelligence Research, 2*, 411–446.

Doyle, J. (1979). A truth maintenance system. *Artificial Intelligence, 12*, 231–272.

Emde, W. (1989). An inference engine for representing multiple theories. In K. Morik (Ed.), *Knowledge reporesentation and organization in machine learning*. Springer Verlag.

Feigenbaum, E. A. (1977). The art of artificial intelligence: Themes and case studies in knowledge engineering. *Proceedings of IJCAI-77* (pp. 1014–1029).

Forbus, K. D., & de Kleer, J. (1993). *Building problem solvers*. Cambridge, Massachusset: MIT Press.

Ginsberg, A. (1988a). *Automatic refinement of expert system knowledge bases*. San Mateo, California: Morgan Kaufmann.

Ginsberg, A. (1988b). Theory revision via prior operationalization. *Proceedings of the Seventh National Conference on Artificial Intelligence* (Vol. 2, pp. 590–595). Minneapolis, MN: Morgan Kaufmann.

Hekanaho, J. (1996). Background knowledge in GA-based concept learning. *Proceedings of the Thirteenth International Conference on Machine Learning* (pp. 234–242). Bari: Italy.

Koppel, M., Ronen, F., & Segre, A. M. (1994). Bias-driven revision of logical domain theories. *Journal of Artificial Intelligence Research, 1*, 159–208.

Mahoney, J. J., & Mooney, R. J. (1993). Combining connectionist and symbolic learning to refine certainty-factor rule-bases. *Connection Science, 5*, 339–364.

McAllester, D. (1982). *Reasoning utility package user's manual*. (AI Lab Report 667) Cambridge, MA: MIT.

Michalski, R. S., & Chilausky, S. (1980). Learning by being told and learning from examples: An experimental comparison of the two methods of knowledge acquisition in the context of developing an expert system for soybean disease diagnosis. *Journal of Policy Analysis and Information Systems, 4*(2), 126–161.

Opitz, D. W., & Shavlik, J. W. (1994). Using genetic search to refine knowledge-based neural networks. *Proceedings of the Eleventh International Conference on Machine Learning*. San Francisco, CA.

Ourston, D., & Mooney, R. (1990). Changing the rules: A comprehensive approach to theory refinement. *Proceedings of the Eighth National Conference on Artificial Intelligence* (pp. 815–820). Menlo Park, CA.

Politakis, P., & Weiss, S. (1984). Using empirical analysis to refine expert system knowledge bases. *Artificial Intelligence, 22*, 23–48.

Popchev, I., Zlatareva, N., & Mircheva, M. (1990). A truth maintenance theory: An alternative approach. *Proceedings of the Ninth European Conference on Artificial Intelligence* (pp. 509–514). Stockholm, Sweden: Pitman Publ.

Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning, 1*, 81–106.

Richards, B. L., & Mooney, R. J. (1995). Automated refinement of first-order Horn-clause domain theories. *Machine Learning, 19*, 95–131.

Tangkitvanich, S., & Shimura, M. (1992). Refining a relational theory with multiple faults in the concept and subconcepts. *Proceedings of the Ninth International Conference on Machine Learning* (pp. 436–444). Aberdeen, UK: Morgan Kaufmann.

Towell, G. G., & Shavlik, J. W. (1992). Interpretation of artificial neural networks: Mapping knowledge-based neural networks into rules. In R. Lippmann, J. Moody, & D. Touretzky (Eds.), *Advances in neural information processing systems* (Vol. 4). Morgan Kaufmann.

Towell, G. G., Shavlik, J. W., & Noordewier, M. O. (1990). Refinement of approximate domain theories by knowledge-based neural networks. *Proceedings of the Eight National Conference on Artificial Intelligence*. Boston, Massachusetts.

Wilkins, D. C. (1990). Knowledge base refinement as improving an incorrect and incomplete domain theory. In Y. Kodratoff & R. Michalski (Eds.), *Machine learning, an AI approach* (Vol. III). Morgan Kauffmann.

Winston, P. H., Binford, T. O., Katz, & Lowry, M. (1983). Learning physical descriptions from functional definitions, examples, and precedents. *Proceedings of the National Conference on Artificial Intelligence* (pp. 433–439). Washington, D.C.

Wogulis, J., & Pazzani, M. (1993). A methodology for evaluating theory revision systems. *Results with AUDREY II. Proceedings of the International Joint Conference on Artificial Intelligence*. Chambery, France.

Wrobel, S. (1994). Concept formation during interactive theory revision. *Machine Learning, 14*, 169–191.

Zlatareva, N. P. (1994). A framework for verification, validation, and refinement of knowledge bases: The VVR system. *International Journal of Intelligent Systems, 9*, 703–737.

Received April 24, 1997

Accepted March 17, 1999

Final manuscript March 17, 1999