

WhyNot: Debugging Failed Queries in Large Knowledge Bases

Hans Chalupsky and Thomas A. Russ

Information Sciences Institute
University of Southern California
4676 Admiralty Way, Marina del Rey, CA 90292

Abstract

When a query to a knowledge-based system fails and returns “unknown”, users are confronted with a problem: Is relevant knowledge missing or incorrect? Is there a problem with the inference engine? Was the query ill-conceived? Finding the culprit in a large and complex knowledge base can be a hard and laborious task for knowledge engineers and might be impossible for non-expert users. To support such situations we developed a new tool called “WhyNot” as part of the PowerLoom knowledge representation and reasoning system. To debug a failed query, WhyNot tries to generate a small set of *plausible partial proofs* that can guide the user to what knowledge might have been missing, or where the system might have failed to make a relevant inference. A first version of the system has been deployed to help debug queries to a version of the Cyc knowledge base containing over 1,000,000 facts and over 35,000 rules.

Introduction

Every knowledge representation system deserving of its name has a reasoning component to allow the derivation of statements that are not directly asserted in the knowledge base but instead follow from the facts and rules that are asserted. A common approach is to use the language of some symbolic logic L as the knowledge representation (KR) language and an implementation of a proof procedure for L as the reasoning (R) engine. This results in what is usually called a knowledge representation and reasoning (KR&R) system (such as Loom, PowerLoom, Classic, Cyc, SNePS, etc.). For example, in a KR&R system based on first-order predicate logic, one can represent facts and rules like the following (we will use KIF notation (Genesereth 1991) throughout this paper):

```
(person fred)
(citizen-of fred germany)
(national-language-of germany german)

(forall (?p ?c ?l)
  (= > (and (person ?p)
            (citizen-of ?p ?c)
            (national-language-of ?c ?l))
       (speaks-language ?p ?l)))
```

When asked about the truth of (speaks-language fred german), the KR&R system can use logical inference and

Copyright © 2002, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

answer “true”. When asked about (speaks-language fred french), however, the system has to answer “unknown”, since it does not have enough information to answer this question. Here and throughout the rest of this paper we make an *open world assumption*, that is, what is in a knowledge base is assumed to be an incomplete model of the world.

The larger the knowledge base grows, the more questions the system can answer; however, that set is usually small compared to the set of interesting questions within its domain that it cannot answer. Therefore, a very common query response is “unknown.” While this is expected, it can be frustrating for knowledge engineers as well as non-expert users, in particular, if the system *should have known* the answer. The problem is exacerbated by the fact that the system cannot provide a good explanation for such failures beyond that everything it tried to derive an answer failed. Standard explanation techniques for logical reasoning rely on the availability or constructibility of a proof that can then be rendered and explained to a user (Chester 1976; McGuinness 1996). In this case, these techniques do not apply, since no successful proof was found among the possibly very many that were tried. Debugging such a failure manually can be very difficult and tedious.

In this paper we describe a novel query debugging tool called WhyNot, that is designed to help a human user to analyze *why* a query could *not* be answered successfully by the underlying KR&R system. It does so by trying to generate *plausible partial proofs* that approximate the correct (or expected) proof that would answer the query. The gaps in such a partial proof can then be scrutinized by the user to see whether they constitute missing knowledge or some other deficiency of the KB or query. A first version of WhyNot has been implemented and used successfully with a large Cyc knowledge base containing over a 1,000,000 facts and over 35,000 rules.

The Problem

KR&R systems can commonly answer two types of queries: (1) true/false questions such as “is it true that (speaks-language fred german)”, and (2) Wh-questions such as “retrieve all ?x such that (speaks-language fred ?x).” We say that a true/false query fails, if the system can neither prove the queried fact to be true nor to be false and returns “unknown.” A Wh-query fails if the system fails to generate

a complete set of answers relative to the expectations of the user or using application. In this paper we are primarily concerned with failed true/false questions. Our techniques apply to Wh-queries as well, but the openness of Wh-queries introduces an extra level of difficulty; however, any one missing value can be analyzed by a corresponding true/false query.

Here are common reasons why queries fail:

Missing knowledge such as relevant facts or rules that are missing from the KB.

Limitations such as using an incomplete reasoner or an undecidable logic, resource exhaustion such as search depth cutoffs or timeouts, etc.

User misconceptions where the user chooses vocabulary missing from the KB, or asks a query that does not actually express what s/he intended to ask, or if the queried fact contradicts what is in the KB.

Bugs in the KB such as incorrect facts or rules or bugs in the inference engine or surrounding software.

A query may fail because one or more of these failure modes is to blame. Some failure modes such as using undefined vocabulary or exhaustion of inference engine resources (timeouts) can be detected by the KR&R system and communicated to the user. Others will simply lead to silent failure. All failures including exhaustion of allotted inference resources can present difficult problems to solve. The reason for this difficulty is that in such a case the KR&R system has explored a potentially very large search space to generate an answer, but everything it tried to derive an answer failed.

Standard explanation techniques to explain results of logical inference cannot be used to help a user with this problem, since they all require a data structure representing a derivation or proof for the answer to a query. Such proof structures are either generated automatically by the inference engine when the query is answered, regenerated during a rerun of the query in “explanation mode”, or generated independently from how it was derived originally by the core inference engine. Which method is chosen depends on the underlying logic and performance optimizations of the inference engine. See (McGuinness 1996) for an example where proofs are generated for the sole purpose of producing explanations. Once a proof has been found, it can be rendered into an explanation for the user.

When a query fails, however, there is no relevant proof that could be explained to the user. Instead, the traditional approach to solve such a problem is to manually examine – if available – inference traces or failure nodes of search trees to see where something might have gone wrong. Since queries in logic-based systems can easily generate very large search trees due to the many different ways in which answers can be derived, this process can require a lot of analytical skills, expertise and time. Such problems are difficult for knowledge engineers, and often unsolvable for non-expert users.

To enable the use of standard proof-based explanation techniques to help with query failures, we need to generate some kind of proof despite the fact that there is not enough information in the KB to support a proof. The WhyNot tool

accomplishes that by generating *plausible partial proofs*. A partial proof is a proof with some of its preconditions unsatisfied (or gaps). If the proof is plausible and close to the correct or expected proof, these gaps can pinpoint missing knowledge that when added to the KB will allow the query to succeed.

Generating plausible partial proofs within large knowledge bases poses two difficult challenges: (1) determining what it means for a proof to be *plausible* among the many possible ones, and (2) *performance*, i.e., finding plausible partial proofs without looking at too many options. How these challenges are tackled by WhyNot is described in the remainder of the paper.

The current version of the WhyNot tool primarily concentrates on the detection of missing facts. Other failures such as missing rules, incorrect knowledge, etc. might also be made easier to diagnose in some cases, but no specific support for these failure modes is available yet. For the application scenario of knowledge base development by users who are not knowledge engineers, being able to suggest potentially missing facts is an important step for helping them debug failed queries and to extend the KB.

Plausible Partial Proofs

The WhyNot tool supports the analysis of query failures by trying to generate *ranked plausible partial proofs*. Intuitively, a plausible partial proof approximates the correct proof the system would find had it all the correct knowledge and capabilities. It is an approximation, since it might not completely match the structure of the correct proof and will not have all its preconditions satisfied. We claim that if we can find a good enough approximation of the correct proof, it will allow the user to determine what pieces of knowledge are missing or what other failures might have occurred that prevented the query to succeed. By generating multiple partial proofs ranked by plausibility, we can provide explanation alternatives and enable the user to focus on the most plausible explanations among a potentially very large set of possible ones.

A *partial* proof for a queried fact q is any proof

$$P : \{p_1, \dots, p_n\} \vdash q$$

where one or more of the premises p_i are not known to be true. A *plausible* partial proof is one where the unknown premises p_i could “reasonably” be assumed to be true. This is of course a semantic notion that can ultimately only be decided by the user.

A *good* plausible proof is one that is well entrenched in the knowledge base by applying relevant facts and rules, and that maximizes the ratio of known to unknown premises. Note that for any failed query q a minimal plausible partial proof is

$$Q : \{q\} \vdash q$$

This is akin to saying “if I had known that q is true I could have told you that q is true.” This is only a good partial proof if there are no rules available to conclude q or if we cannot adequately satisfy the preconditions of any applicable rules.

Generating plausible partial proofs is a form of abductive reasoning. In abduction, q would be something that was

observed (e.g., the symptoms of a disease), and the set of p_i whose truth values were not a-priori known would be the abduced explanation for q (e.g., the set of diseases that would explain the symptoms). In a standard abductive reasoner the abduced hypotheses are non-monotonic inferences that are added to the theory. In our case, they are used to generate hypothetical answers to the failed query q , for example, “if p_i and p_j were true then q would be true”. This therefore is related to the work on hypothetical question answering by Burhans and Shapiro (1999).

Judging the plausibility of a partial proof is a similar problem to deciding between alternative explanations in abductive inference. In the abduction literature, many different criteria have been proposed, for example, explanations should be minimal, basic, consistent, most specific, least specific, least cost, etc. The best choice depends very much on the problem domain. For WhyNot we want a criterion that reduces the number of partial proofs for a particular query to an amount that is amenable to manual inspection without being too discriminating. For example, the strongest plausibility criterion we can apply within a logic-based framework is to require that the set of unknown premises be logically consistent with the rest of the knowledge base. However, large knowledge bases, in particular if they are under development, are often not consistent or correct; therefore, we do not want to be too aggressive about weeding out inconsistent premises, since they might indicate other problems in the KB.

Example

Before we describe the plausibility scoring and proof generation scheme, we want to give an example how the WhyNot tool can be used to analyze the failure of a query. Consider the following small knowledge base fragment:

```
(and (person fred) (person phil) (person susi))
(parent-of fred phil)
(national-language-of usa english)
(national-language-of france french)
(national-language-of germany german)
(national-language-of canada english)
(national-language-of canada french)

(forall (?p ?c ?l)
  (=) (and (person ?p)
           (citizen-of ?p ?c)
           (national-language-of ?c ?l))
       (speaks-language ?p ?l)))

(forall (?p ?c ?l)
  (=) (and (person ?p)
           (birth-place-of ?p ?c)
           (national-language-of ?c ?l))
       (native-language-of ?p ?l)))

(forall (?p ?l)
  (=) (and (person ?p)
           (native-language-of ?p ?l))
       (speaks-language ?p ?l)))

(forall (?p ?l)
  (=) (exists (?f)
        (and (parent-of ?p ?f)
              (native-language-of ?f ?l))
            (speaks-language ?p ?l))))
```

If we ask the system whether (speaks-language fred german) is true, it returns “unknown”, since at this point all that is known about Fred is that he is a person and has Phil as a parent. If we now run the WhyNot tool on the failed query

we get the following two explanations sorted by score (the plausibility score can be in the range of 1.0 for strictly true to -1.0 for strictly false):

Explanation #1 score=0.708:

```
1 (SPEAKS-LANGUAGE FRED GERMAN)
  is partially true by Modus Ponens
  with substitution ?p/FRED, ?l/GERMAN, ?f/PHIL
  since 1.1 ! (forall (?p ?l)
               (<= (SPEAKS-LANGUAGE ?p ?l)
                   (exists (?f)
                     (and (PARENT-OF ?p ?f)
                           (NATIVE-LANGUAGE-OF ?f ?l))))))
  and 1.2 ! (PARENT-OF FRED PHIL)
  and 1.3 (NATIVE-LANGUAGE-OF PHIL GERMAN)

1.3 (NATIVE-LANGUAGE-OF PHIL GERMAN)
  is partially true by Modus Ponens
  with substitution ?p/PHIL, ?l/GERMAN, ?c/GERMANY
  since 1.3.1 ! (forall (?p ?l)
                (<= (NATIVE-LANGUAGE-OF ?p ?l)
                    (exists (?c)
                      (and (PERSON ?p)
                            (BIRTH-PLACE-OF ?p ?c)
                            (NATIONAL-LANGUAGE-OF ?c ?l))))))
  and 1.3.2 ! (PERSON PHIL)
  and 1.3.3 ? (BIRTH-PLACE-OF PHIL GERMANY)
  and 1.3.4 ! (NATIONAL-LANGUAGE-OF GERMANY GERMAN)
```

Explanation #2 score=0.556:

```
2 (SPEAKS-LANGUAGE FRED GERMAN)
  is partially true by Modus Ponens
  with substitution ?p/FRED, ?l/GERMAN, ?c/GERMANY
  since 2.1 ! (forall (?p ?l)
               (<= (SPEAKS-LANGUAGE ?p ?l)
                   (exists (?c)
                     (and (PERSON ?p)
                           (CITIZEN-OF ?p ?c)
                           (NATIONAL-LANGUAGE-OF ?c ?l))))))
  and 2.2 ! (PERSON FRED)
  and 2.3 ? (CITIZEN-OF FRED GERMANY)
  and 2.4 ! (NATIONAL-LANGUAGE-OF GERMANY GERMAN)
```

In each explanation propositions that were found to be directly asserted in the KB are marked with ‘!’ and unknown propositions are marked with ‘?’. Propositions without a mark such as 1.3 are supported by a further reasoning step. Each explanation describes a partial proof for the failed query. If all its unknown leafs marked with ‘?’ would be true, the query would follow.

Despite the differences in score, both explanations above identify only a single missing fact that would allow the derivation of the expected answer. The first explanation scores higher because a larger percentage of the total number of ground facts needed in the proof tree are present. Since the plausibility scoring scheme will always be imperfect and since the system has no way to decide *a priori* which of those missing facts is more likely, we present multiple explanations for the user to consider.

By default, the WhyNot tool suppresses explanations with an absolute score of less than 0.3. If we lower the threshold to 0.0 we get the following additional explanation:

Explanation #3 score=0.208:

```
3 (SPEAKS-LANGUAGE FRED GERMAN)
  is partially true by Modus Ponens
  with substitution ?p/FRED, ?l/GERMAN, ?f/SUSI
  since 3.1 ! (forall (?p ?l)
               (<= (SPEAKS-LANGUAGE ?p ?l)
                   (exists (?f)
                     (and (PARENT-OF ?p ?f)
                           (NATIVE-LANGUAGE-OF ?f ?l))))))
  and 3.2 ? (PARENT-OF FRED SUSI)
  and 3.3 (NATIVE-LANGUAGE-OF SUSI GERMAN)

3.3 (NATIVE-LANGUAGE-OF SUSI GERMAN)
  is partially true by Modus Ponens
```

```

with substitution ?p/SUSI, ?l/GERMAN, ?c/GERMANY
since 3.3.1 ! (forall (?p ?l)
  (<= (NATIVE-LANGUAGE-OF ?p ?l)
    (exists (?c)
      (and (PERSON ?p)
            (BIRTH-PLACE-OF ?p ?c)
            (NATIONAL-LANGUAGE-OF ?c ?l))))))
and 3.3.2 ! (NATIONAL-LANGUAGE-OF GERMANY GERMAN)
and 3.3.3 ? (BIRTH-PLACE-OF SUSI GERMANY)
and 3.3.4 ! (PERSON SUSI)

```

This last explanation scores lower, since two additional facts would need to be added to the KB for the question to be answered true. It was generated by postulating bindings for the variable *?f* in rule (3.1). In explanation 1 *?f* was bound by a direct assertion. Now, in addition to Susi, there could be many other persons who might be parents of Fred. Some of them would be implausible, such as in the following suppressed partial proof:

```

4 (SPEAKS-LANGUAGE FRED GERMAN)
is partially true by Modus Ponens
with substitution ?p/FRED, ?l/GERMAN, ?f/FRED
since 4.1 ! (forall (?p ?l)
  (<= (SPEAKS-LANGUAGE ?p ?l)
    (exists (?f)
      (and (PARENT-OF ?p ?f)
            (NATIVE-LANGUAGE-OF ?f ?l))))))
and 4.2 X (PARENT-OF FRED FRED)
and 4.3 (NATIVE-LANGUAGE-OF FRED GERMAN)

```

The explanation is suppressed, because Fred cannot be his own parent (clause 4.2), since parent-of is asserted to be ir-reflexive. In large and complex KBs, many of the dead ends pursued by the inference engine are nonsensical inferences such as the above. The primary function of the WhyNot module is to separate the wheat from the chaff and present a small number of good guesses for plausible partial inferences to the user. This allows the WhyNot module to reject the explanation on grounds of inconsistency, since (parent-of fred fred) is provably false using very simple inference. In general, however, we cannot rely on that completely, since the more ground a KB covers, the more room there is for unspecified semantic constraints; therefore, we always need a human user to make the final judgment.

Scoring Partial Proofs

A good plausible proof is one that maximizes the ratio of known to unknown premises while at the same time being well entrenched in the knowledge base. The assumed unknowns should also not be in direct conflict with other facts in the KB. We do not require full consistency, (1) since it is not computable, and (2) since it would be much too costly to try to even approximate. Whenever we accept an unknown premise, we therefore only check for its falsity via quick lookup or shallow reasoning strategies. To implement these plausibility heuristics we use the following score computation rules:

Atomic goals *p*: The score of an atomic proposition *p* is 1.0 if it was found strictly true, -1.0 if it was found strictly false or 0.0 if it is completely unknown. If it was inferred by a rule of inference such as Modus Ponens, its score is determined by the score combination function of that rule.

AND-introduction goals (and *p*₁ ... *p*_{*n*}): the score of a conjunction is computed as a weighted average of the scores of its conjuncts: let *s*(*p*_{*i*}) be the score of a conjunct and

w(*p*_{*i*}) be its weight. The resulting score is

$$\text{score} = \frac{\sum_{i=1}^n w(p_i) s(p_i)}{\sum_{j=1}^n w(p_j)}$$

If any of the *p*_{*i*} is found to be strictly false, the conjunction fails strictly and the resulting score is -1.

The weights *w*(*p*_{*i*}) are used to scale down the importance of certain classes of propositions. For example, a subgoal of the form (not (= ?*x* ?*y*)) has a very high likelihood to succeed for any pair of bindings. We therefore scale down the importance of the truth of such a subgoal by heuristically giving it a low weight of 0.1. Similarly, type constraints such as person in (and (person ?*x*) (person ?*y*) (father-of ?*x* ?*y*)) are given somewhat less importance, since in general they are more likely to be found true and they are often redundant. These are of course only simple heuristics in the absence of a more general scheme to determine the semantic contribution of a particular conjunct (see (Hobbs *et al.* 1993)[p.84] for some more discussion of this issue). Future versions of WhyNot might perform statistical analysis of the underlying KB to determine these weights automatically.

OR-introduction goals (or *p*₁ ... *p*_{*n*}): the score of a disjunction is the weighted score (*w*(*p*_{*i*})*s*(*p*_{*i*})) of the first disjunct (starting from the current choice point) whose score exceeds the current minimum cutoff. A higher-scoring disjunct can be found when we backtrack or generate the next partial proof.

Implication elimination (Modus Ponens) goals: when concluding *q* from *p* and ($\Rightarrow p q$) we compute *s*(*q*) as *s*(*p*)*d* where *d* is a degradation factor that increases with the depth of a proof. The rationale for the degradation factor is that the longer a partial reasoning chain is, the higher the likelihood that it is incorrect. For any unknown proposition *p* partial support from chaining through a rule is seen as support for its truth, but that support is counted less and less the deeper we are in the proof tree.

Generating Partial Proofs

The WhyNot partial proof generator is built into the inference engine of the PowerLoom KR&R system. PowerLoom uses a form of natural deduction to perform inference and it combines a forward and backward chaining reasoner to do its work. PowerLoom's inference engine is a "pragmatic" implementation. It is not a complete theorem prover for first-order logic, yet it has various reasoning services such as type level reasoning, relation subsumption, a relation classifier, selective closed world reasoning, etc. that go beyond the capabilities of a traditional first-order theorem prover. It also has a variety of controls for resource bounded inference to allow it to function with large knowledge bases.

The backward chainer uses depth-first search with chronological backtracking as its primary search strategy. Memoization caches successful and, if warranted, failed subgoals to avoid rederivation. The ordering of conjunctive goals is optimized by considering various costs such as unbound variables, goal fanout and whether a subgoal can be inferred by rules. Duplicate subgoals are detected and depth cutoffs and timeouts provide resource control. Once

a solution has been found, the next one can be generated by continuing from the last choice point.

The partial proof generator is part of PowerLoom’s backward chainer. Originally, it was built to overcome the general brittleness of logic-based KR&R systems by providing a partial match facility, i.e., to be able to find partial solutions to a query when no strict or full solutions can be found. The WhyNot tool extends this facility for the purposes of generating plausible partial proofs. The partial proof generator keeps trying to prove subgoals of a goal even if some other conjoined subgoals already failed. Once all subgoals have been attempted, instead of a truth value a combined score is computed to reflect the quality of the solution, and that score is then propagated to the parent goal.

To retain some pruning ability, a minimally required cutoff score is computed at each state in the proof tree and further subgoaling is avoided if it can be determined that the required cutoff score is unachievable. The top-level controller uses the cutoff mechanism to generate a set of better and better proofs by using the scores of previously found proofs as a minimal score cutoff. To avoid having high-scoring proofs found early mask out other slightly lesser scoring but equally plausible proofs, the controller accumulates at least N proofs before it uses the worst score of the current top- N proofs as the cutoff (currently, we use $N = 10$). While the WhyNot partial proof generator is running, justifications for (partially) satisfied subgoals are kept which results in a partial proof data structure for satisfied top-level goals. During normal operation of the inference engine, such justifications are not recorded for performance reasons.

Taming the Search

Generating plausible partial proofs is a form of abductive reasoning and even restricted forms of abduction are computationally intractable (cf., (Bylander *et al.* 1991; Selman & Levesque 1990)). In practice, generating partial proofs provides less opportunities to prune the search tree. For example, if we are trying to prove a conjunction as part of a strict proof, we can stop and backtrack as soon as we fail to prove one of the conjuncts. If we are looking for a partial proof we cannot do that, since the failed conjunct might be the missing knowledge we are trying to identify. We need to examine the other conjuncts to determine whether the remainder of the proof is plausible. For this reason, generating partial proofs requires different search control than generating regular proofs. Time limits are an important means to keep the effort spent under control. Additionally, the minimal cutoff score mechanism described above can reinstate some pruning opportunities, but by its nature partial proof search will be less focused. Particularly problematic can be generator subgoals as created by rules such as the following:

```
(forall (?p ?l)
  (= > (exists (?f)
    (and (person ?f)
      (parent-of ?p ?f)
      (native-language-of ?f ?l))))
  (speaks-language ?p ?l)))
```

When we are trying to prove the goal (speaks-language fred german) and backchain into the above rule, PowerLoom

tries to prove the existential by introducing a generator subgoal of the following form:

```
(and (parent-of fred ?f)
  (person ?f)
  (native-language-of ?f german))
```

The conjunctive goal optimizer reordered the subgoals to have the one that is expected to generate the least number of bindings for $?f$ as the first one. Once $?f$ is bound the remaining goals are harmless and can be solved by simple lookup or chaining. In the strict case, if the first subgoal fails to generate any binding for $?f$ the existential goal simply fails. If we are generating partial proofs the failed conjunct is skipped and the next conjunct is tried with $?f$ still unbound. This can constitute a problem, since the number of bindings generated for $?f$ by either of the remaining clauses might be much larger than what was estimated for the first conjunct. If we simply go on to the next clause, we would generate the set of all people in the KB which could be very large. As a first line of defense, we have to reoptimize the remaining clauses, since with $?f$ still unbound it might now be better to try the last clause first. If the remaining clauses are still too unconstrained, for example, if the language variable $?l$ was also unbound, we simply skip them and fail since the large number of unconnected bindings would probably not be useful for an explanation anyway.

Another search problem that surfaced when applying WhyNot to a very large KB such as Cyc was that not all applicable rules were tried due to query timeouts. In normal processing, this is not usually a problem, since aggressive pruning means rules often fail fairly early in their consideration. With the reduced pruning used by WhyNot, this caused certain rules not to be considered at all. We needed to change the WhyNot search strategy to “give each rule its equal chance.” Now whenever it tries to prove a subgoal p and there are n applicable rules, it allocates $\frac{1}{n}$ -th of the remaining available time to each rule branch. Time that wasn’t fully used by one of the rules is redistributed among the remaining ones. This time-slicing ensures that we never get stuck in one subproof without any time left to explore remaining alternatives.

Explanation Post-Processing

Before explanations are presented to the user, some post-processing is required: (1) to remove unnecessary detail and make them easier to understand, (2) to remove some idiosyncrasies introduced by the partial proof search, and (3) to give some advice on how to interpret the partial proof.

How to best present a proof to a non-expert user has not been the primary focus of our work. We rely on the fact that PowerLoom’s natural deduction inference can be translated fairly directly into explanations; however, some uninformative proof steps such as AND-introduction or auxiliary steps needed by the inference engine are suppressed.

A more interesting post-processing step is illustrated by the following example: suppose we ask (speaks-language fred french) in the example KB presented above. Just as before the answer will be “unknown”, but WhyNot will return a slightly different explanation for the failure:

Explanation #1 score=0.708:

```
1 (SPEAKS-LANGUAGE FRED FRENCH)
  is partially true by Modus Ponens
  with substitution ?p/FRED, ?l/FRENCH, ?f/PHIL
  since 1.1 ! (forall (?p ?l)
    (<= (SPEAKS-LANGUAGE ?p ?l)
      (exists (?f)
        (and (PARENT-OF ?p ?f)
              (NATIVE-LANGUAGE-OF ?f ?l))))))
  and 1.2 ! (PARENT-OF FRED PHIL)
  and 1.3 (NATIVE-LANGUAGE-OF PHIL FRENCH)

1.3 (NATIVE-LANGUAGE-OF PHIL FRENCH)
  is partially true by Modus Ponens
  with substitution ?p/PHIL, ?l/FRENCH, ?c/[FRANCE, CANADA]
  since 1.3.1 ! (forall (?p ?l)
    (<= (NATIVE-LANGUAGE-OF ?p ?l)
      (exists (?c)
        (and (PERSON ?p)
              (BIRTH-PLACE-OF ?p ?c)
              (NATIONAL-LANGUAGE-OF ?c ?l))))))
  and 1.3.2 ! (PERSON PHIL)
  and 1.3.3 ? (BIRTH-PLACE-OF PHIL ?c)
  and 1.3.4 ! (NATIONAL-LANGUAGE-OF ?c FRENCH)
```

The difference is that there were two countries whose national language was French. This resulted in two almost identical partial proofs that only differed in the binding for variable ?c in clauses 1.3.3 and 1.3.4. Instead of making an arbitrary choice or giving a series of almost identical explanations for each country whose national language is French, the WhyNot tool collapses those answers into one¹. The result is an intensional explanation which might be paraphrased as “if for some ?c such that (national-language-of ?c french) it were known that (birth-place-of phil ?c), then the query could be answered. A few example individuals that were actually found to make clause 1.3.4 true are annotated in a special syntax in the variable substitution. Once a generalization has occurred, the part of the inference tree responsible for generating similar bindings is cut off to force the generation of the next structurally different proof.

Future Work

One type of post-processing not yet attempted by WhyNot is to try to give advice how to best interpret a partial proof, since gaps in a partial proof can be the result of problems other than missing knowledge. For example, consider this query given to the Cyc knowledge base: “Can Anthrax lethally infect animals?” The query fails and WhyNot produces the explanation shown in Figure 1 (this explanation is rendered for a non-expert audience and uses Cyc’s natural language generation mechanism).

The partial proof found by WhyNot points to the relevant portion of the KB, but here we do not have a case of missing knowledge. Rule (1.1) is a type-level inheritance rule that says if some type of organism can lethally infect some type of host, then subtypes of the organism can infect subtypes of the host. In other words, if we know anthrax can infect mammals (fact 1.2), then anthrax can infect sheep. But instead of asking about sheep, the question was about anthrax infecting *animals*. What WhyNot could not determine in clause (1.4) was whether animal is a subtype of mammal.

¹In general, even if there were only one binding found for some variable, the choice might be arbitrary and an intensional explanation might be better; however, currently we only generate an intensional explanation if we find two or more similar bindings.

This is of course not missing knowledge, but the query was asked about too general a type. One hint to that could have been that mammals actually are a subtype of animal, and that these are usually proper subtype relationships; therefore, suggesting that animal is a subtype of mammal is unlikely to be correct. At this point, however, it is not clear how to reliably make such a determination which is why it is not yet done.

Related Work

Relatively little has been done in the area of analyzing failed inferences or queries in KR&R systems. The closest is McGuinness’ (1996) work on providing explanation facilities to description logics, in particular, explaining why some concept A does not subsume another concept B. Since most description logics are decidable, non-subsumption can be determined by having a complete reasoner fail on proving a subsumption relationship. To explain a non-subsumption to the user with “everything the system tried failed” is of course not a good option for the same reasons we outlined above. Since the logic is decidable, however, non-subsumption is constructively provable and explainable by generating a counter-example. McGuinness’ work differs from our case where we have an (at best) semi-decidable logic and an incomplete reasoner where the answer to a query really is unknown and not derivable as opposed to false.

Kaufmann (1992) describes a utility to inspect the failed proof output of the Boyer-Moore theorem prover. He claims that inspection of such output is crucial to the successful use of the prover. The utility allows the definition of checkpoints for situations where the prover is trying some fancy strategy, since this often indicates a processing has entered a dead end. This tool does not provide automatic analysis of a failed proof and is geared towards a technical audience.

Gal (1988) and Gaasterland (1992) describe cooperative strategies for query answering in deductive databases. One of their main concerns is to provide useful answers or allow query relaxation in cases where the user has misconceptions about the content of the database. These techniques handle some query failure modes not addressed by our tool and might be useful additions in future versions.

Finally, there is a large body of work describing how to best explain the reasoning results of expert systems, planners, theorem provers, logic programs, etc. to users (see for example, (Chester 1976; Wick & Thompson 1992; McGuinness 1996)). Some of these techniques could be applied to the output of our tool. Since we wished to focus our research on generating the plausible proofs we use a relatively simple strategy to describe the partial natural deduction proofs found by the system.

Using WhyNot with Cyc

The PowerLoom WhyNot tool has been integrated into the Cyc tool suite to support the debugging of failed queries. This integration is part of DARPA’s Rapid Knowledge Formation program (RKF) — an effort to research and develop new techniques and technologies to widen the knowledge

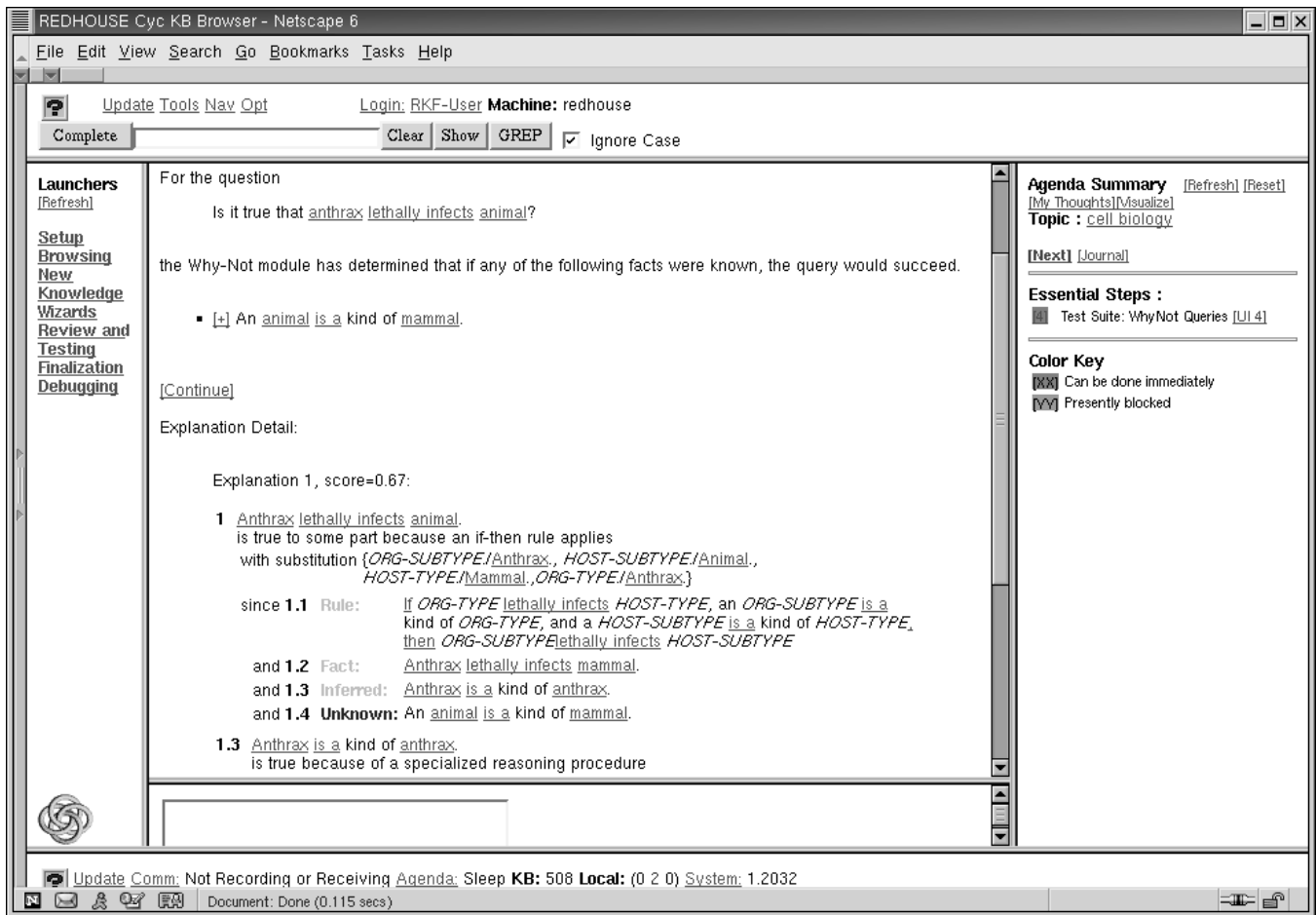


Figure 1: Showing a WhyNot Explanation in Cyc

acquisition bottleneck. The primary goal is to enable domain experts such as microbiologists who are not computer scientists to author competent knowledge bases of their field of expertise with no or only minimal help from knowledge engineers. Two teams, one led by Cycorp and one by SRI, are each developing a set of knowledge acquisition tools whose performance is repeatedly evaluated in the context of a challenge problem. Once this evaluation process is completed, we expect to be able to provide both quantitative data and qualitative feedback about the performance and usefulness of the WhyNot tool.

The tool developed by Cycorp is built around the large Cyc common sense knowledge base (Lenat 1995). The version of Cyc used within RKF contains over 1,000,000 facts and over 35,000 rules. One of the claims of Cycorp's approach is that the large amount of common sense knowledge available in Cyc makes it easier and more efficient to enter and verify new knowledge. Cycorp's tool also contains new, fairly sophisticated natural language parsing and generation components. All input from domain experts is given in natural language (in some cases constrained by templates), and any Cyc knowledge that is displayed is rendered in natural

language. The expert therefore never has to use the MELD KR language used internally by Cyc. The interface to Cyc is via a Web browser as shown in the screenshot in Figure 1.

WhyNot can be viewed as a Cyc Plug-in that can provide special reasoning services not available in Cyc. WhyNot runs as an external knowledge module connected to Cyc via the VirB³ blackboard developed by Cycorp. It is written in our own STELLA programming language (Chalupsky & MacGregor 1999) which can be translated into Lisp, C++ and Java. In this application we used the Java translation which can be shipped as a 2 Megabyte Java Jar file. A small amount of SubLisp code was also written to provide some extensions to the Cyc API. SubLisp is a programming language developed by Cycorp that, similar to STELLA, tries to preserve a Lisp-based development environment while allowing to release code in mainstream language such as C.

When domain experts are adding knowledge or testing the knowledge they authored, they have to query the Cyc knowledge base. When a query fails and returns unknown as its result, they can ask the WhyNot module to analyze the failure. WhyNot requests are processed in the background and the user can perform other knowledge acquisition tasks while

waiting for a response.

When WhyNot receives such a request on the blackboard, it first translates the failed query expression from MELD into KIF and then starts to derive plausible partial proofs. Since the WhyNot module runs in a completely separate implementation outside of Cyc, it needs to get access to all the relevant knowledge in the Cyc KB. It does so by paging in knowledge from Cyc completely dynamically on demand as it is needed by the PowerLoom inference engine. To do that a knowledge paging mechanism is used that maps PowerLoom's KB indexing and access language onto that of a foreign knowledge store such as Cyc. The Cyc API provides a rich set of access and KB indexing functions that make this fairly efficient. The connection to the Cyc API is not through the blackboard but via a dedicated TCP/IP channel to get higher throughput. Once knowledge is paged in it is cached for efficiency. Updates on the Cyc side cause relevant portions of the paging cache to be flushed. In this application large portions of the Cyc KB can be considered read-only which makes this caching scheme feasible.

Each assertion shipped from Cyc to PowerLoom is translated from MELD to KIF and back again. This is not too difficult, since the languages are fairly similar. An initial translation is performed on the Cyc side, and any remaining translations are performed by a small set of OntoMorph rewrite rules (Chalupsky 2000). Once a WhyNot explanation has been generated, it is rendered into HTML and the referenced facts and rules are run through Cyc's natural language generator. The result is then shipped back to Cyc where it appears on the agenda and can be displayed by the user (see Figure 1).

Running partial inference dynamically against a Cyc KB proved to be quite a challenging task. Cyc contains relations such as `objectFoundInLocation` that have hundreds of rules associated with them. Proper search and resource control as described above is therefore extremely important. Similarly, since paging in knowledge from Cyc is somewhat slow, minimizing page-in was a priority. The resulting system is not blazingly fast, but the performance is acceptable. Generating the explanation for the query in Figure 1 takes about 30 seconds the first time around including the time to page in all the relevant knowledge. Subsequent calls run in less than 5 seconds. WhyNot queries run with a timeout of three minutes which seems to be a good compromise for the queries encountered so far. Since WhyNot queries are triggered by the user and would otherwise involve time-consuming manual debugging, the reported run times and timeouts seem to be acceptable.

Acknowledgements This research was supported by the Defense Advance Research Projects Agency under Air Force Research Laboratory contract F30602-00-C-0160.

References

Burhans, D., and Shapiro, S. 1999. Finding hypothetical answers with a resolution theorem prover. In *Papers from the 1999 AAI Fall Symposium on Question Answering Systems, Technical Report FS-99-02*, 32–38. Menlo Park, CA: AAAI Press.

Bylander, T.; Allemang, D.; Tanner, M.; and Josephson, J. 1991. The computational complexity of abduction. *Artificial Intelligence* 49(1–3):25–60.

Chalupsky, H., and MacGregor, R. 1999. STELLA – a Lisp-like language for symbolic programming with delivery in Common Lisp, C++ and Java. In *Proceedings of the 1999 Lisp User Group Meeting*. Berkeley, CA: Franz Inc.

Chalupsky, H. 2000. OntoMorph: a translation system for symbolic knowledge. In Cohn, A.; Giunchiglia, F.; and Selman, B., eds., *Principles of Knowledge Representation and Reasoning: Proceedings of the Seventh International Conference (KR2000)*. San Francisco, CA: Morgan Kaufmann.

Chester, D. 1976. The translation of formal proofs into English. *Artificial Intelligence* 7(3):261–278.

Gaasterland, T. 1992. Cooperative explanation in deductive databases. In *AAAI Spring Symposium on Cooperative Explanation*.

Gal, A. 1988. *Cooperative Responses in Deductive Databases*. Ph.D. Dissertation, University of Maryland, Department of Computer Science. CS-TR-2075.

Genesereth, M. 1991. Knowledge interchange format. In Allen, J.; Fikes, R.; and Sandewall, E., eds., *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning*, 599–600. San Mateo, CA, USA: Morgan Kaufmann Publishers.

Hobbs, J.; Stickel, M.; Appelt, D.; and Martin, P. 1993. Interpretation as abduction. *Artificial Intelligence* 63(1–2):69–142.

Kaufmann, M. 1992. An assistant for reading Nqthm proof output. Technical report 85, Computational Logic.

Lenat, D. 1995. CYC: A Large Scale Investment in Knowledge Infrastructure. *Communications of the ACM* 38(11):32–38.

McGuinness, D. 1996. *Explaining Reasoning in Description Logics*. Ph.d. dissertation, Department of Computer Science, Rutgers University, New Brunswick, New Jersey.

Selman, B., and Levesque, H. 1990. Abductive and default reasoning: A computational core. In Dietterich, T., and Swartout, W., eds., *Proceedings of the Eighth National Conference on Artificial Intelligence*, 343–348. Menlo Park, CA: AAAI Press.

Wick, M., and Thompson, W. 1992. Reconstructive expert system explanation. *Artificial Intelligence* 54(1–2):33–70.