Knowledge Base Reduction for Verifying Rule Bases Containing Equations

Keith Williamson, Mark Dahl
Boeing Computer Services
P.O. Box 24346, MS 7L-44
Seattle, WA 98124-0346
kew@atc.boeing.com

Abstract

The use of Knowledge Base Reduction to verify rule bases was first developed for propositional rule bases, and then extended to handle quantified variables in rules. However, these techniques did not reason about equations. This paper gives a formal specification of KB-Reducer3, an analysis tool for rule bases containing equations. The algorithm for KB-Reducer3 is described, and a brief discussion on its use and limitations is provided.

Introduction

The use of a Knowledge Base Reduction (KBR) approach to verifying rule bases was originally described in (Ginsberg 1988). This approach was first developed for propositional rule bases, and then extended to handle quantified variables in rules (Ginsberg & Williamson 1989). However, this latter extension did not reason about equations. As described in (Dahl & Williamson 1992), there was a need at Boeing to have this capability. This paper formalizes KB-Reducer3, an analysis tool for rule bases containing equations. We will begin by defining KBR knowledge bases and inference in those knowledge bases. Next we define consistency, redundancy, and completeness. We then present an algorithm that tries to determine if a KBR knowledge base is consistent, irredundant, and complete. Finally, we briefly describe the actual use of KB-Reducer3. For a more complete description of this use, the reader is referred to (Dahl & Williamson 1993).

Knowledge Bases

A familiarity with first order logic is assumed; see (Genesereth & Nilsson 1987), for example. Let P1, P2, P3, V1, V2, and V3 be disjoint sets of variable symbols. P1, P2, and P3 contain propositional variables, while V1, V2, and V3 contain non-propositional variables. Let Inputs = <P1,V1>, Intermediates = <P2,V2>, Outputs = <P3,V3>, and S = <Inputs, Intermediates, Outputs>. A **kbr-wff** is a well-formed formula of first-order logic with equality and arithmetic, but no quantifiers. A **kbr-wff over** S is a kbr-wff where variable symbols are taken from S. A **kbr-expression over** S is a function term where variable symbols are taken from (V1 ∪ V2 ∪ V3). A **kbr-rule over** S is a kbr-wff over S of one of the following forms:

$$Condition \rightarrow Var = Expr$$
$$Condition \rightarrow Lit$$

where Condition is a kbr-wff over S, Var is in (V2 ∪ V3), Expr is a kbr-expression over S, and Lit is a literal over (P2 ∪ P3). A **kbr-domain over** S is a two-tuple of the form <Var,{C1,C2,...,Cn}>, where Var is in (V2 ∪ V3), and each Ci is a constant symbol.

Let D be a set of kbr-domains over S, R be a conjunction of kbr-rules over S, C be a satisfiable conjunction of kbr-wffs over Inputs, and KB = < S, D, R, C >. We call KB a **KBR knowledge base**, D the **domain**, R the **rules**, and C the **constraints**. An **input-state** for KB is a logical interpretation of the symbols in Inputs. An input-state i is **valid** for KB iff i satisfies C. A **conclusion** for KB is a kbr-wff over Outputs of one of the two forms ($Var = Const$) or (Lit), where Var is in V3, Const is a constant, and Lit is a literal over P3. Let V be the set of all valid input-states for KB, and K be the set of all conclusions for KB. For any i in V and any k in K, let ($< KB, i > \vdash k$) denote the inference, under KB, of k from i. The inference mechanism associated with KBR knowledge bases has the following properties: (1) it uses back-chaining, (2) it uses monotonic reasoning, and (3) it uses standard logical semantics for negation (ie., it does not use "negation as failure" as Prolog does).

In the following example, KB = < S, D, R, C > is a knowledge base:

$$Inputs =< \{p1, p2, p3\}, \{u1, u2, u3\} >$$
$$Intermediates =< \{q1, q2, q3\}, \{v1, v2, v3\} >$$
$$Output =< \{r1, r2, r3\}, \{w1, w2, w3\} >$$
$$S =< Inputs, Intermediates, Outputs >$$
$$D = \{< v1, \{1, 2, 3, 4\} >, < w2, \{10, 20\} >\}$$
$$\begin{aligned} R = {} & (p1 \wedge \neg p2 \wedge (u1 < u2 \vee u2 < u3) \rightarrow q1) \wedge \\ & (p2 \wedge u2 > 0 \rightarrow v1 = u2) \wedge \\ & ((p3 \wedge q1) \vee v1 = 1 \rightarrow \neg r1) \wedge \\ & (v1 = 1 \wedge \neg p3 \rightarrow w1 = u1 + u2 + u3) \wedge \\ & (v1 = 1 \wedge p1 \rightarrow r2) \end{aligned}$$
$$C = (u1 > 0) \wedge (u2 = 0 \vee u3 = 0)$$

The input state described by $(p1 \wedge p2 \wedge \neg p3 \wedge u1 = 5 \wedge u2 = 1 \wedge u3 = 0)$ is a valid input state since it satisfies C. However, the input state described by $(p1 \wedge p2 \wedge \neg p3 \wedge u1 = -5 \wedge u2 = 1 \wedge u3 = 0)$ is not a valid input state since it does not satisfy $u1 > 0$. Some of the conclusions for this KB are: $\{r1, \neg r1, r2, \neg r2, w1 = 0, w1 = 1\}$. As an example of inference in KBR knowledge bases, under the valid input state described by $(p1 \wedge p2 \wedge \neg p3 \wedge u1 = 5 \wedge u2 = 1 \wedge u3 = 0)$ the following conclusions can be inferred: $\{\neg r1, r2, w1 = 6\}$.

Anomalies in Knowledge Bases

We can define three types of anomalies for KBR knowledge bases. It should be noted that evidence of these anomalies in a real knowledge base is often indicative of a more fundamental problem. KB-Reducer3 points out anomalies, but it is the responsibility of a knowledge base developer or maintainer to decide not only what caused the anomaly but how to fix the underlying problem. For a more complete discussion of this issue, see (Dahl & Williamson 1993).

Consistency

KB is said to be **inconsistent** iff there is a p in V, a k1 in K, and a k2 in K, such that (1) $(<KB, p> \vdash k1)$, (2) $(<KB, p> \vdash k2)$, and (3) $(k1 \wedge k2)$ is unsatisfiable. KB is **consistent** iff it is not inconsistent. Consider the following examples. The rules $(p1 \wedge p2 \rightarrow r1) \wedge (p2 \rightarrow \neg r1)$ lead to an inconsistency since any input state where $(p1 \wedge p2)$ is true leads to the inference of $(r1 \wedge \neg r1)$ which is unsatisfiable. The rules $(p1 \wedge u1 > 50 \rightarrow u2 = 1) \wedge (p2 \wedge u1 > 100 \rightarrow u2 = 2)$ also lead to an inconsistency since any input states satisfying $(p1 \wedge p2 \wedge u1 > 100)$ leads to $(u2 = 1 \wedge u2 = 2)$. An inconsistency could result from a more complex chain of reasoning. For example, given the rules $(p1 \wedge u1 > 50 \rightarrow q1) \wedge (q1 \wedge p2 \rightarrow r1) \wedge (p1 \wedge u1 > 100 \rightarrow q2) \wedge (q2 \wedge p3 \rightarrow \neg r1)$, any input state satisfying $(p1 \wedge p2 \wedge p3 \wedge u1 > 100)$ leads to $(r1 \wedge \neg r1)$.

Redundancy

Let KB1 = <S,D,R1,C> and KB2 = <S,D,R2,C> be KBR knowledge bases. KB1 and KB2 are said to be **equivalent** iff for all p in V, for all k in K, $(<KB1, p> \vdash k)$ iff $(<KB2, p> \vdash k)$; that is, they have the same input/output behaviour (on valid input-states). For any kbr-rule r in R, let (R-r) denote the conjunction formed by removing r from R. KB is **redundant** iff there is some kbr-rule r in R such that KB is equivalent to <S,D,(R-r),C>. KB is **irredundant** iff it is not redundant. For example, a knowledge base containing the rules $(p1 \rightarrow r1) \wedge (p1 \wedge p2 \rightarrow r1)$ is redundant, since the second rule can be removed while maintaining the input/output behaviour of the knowledge base. As another example, consider the knowledge base whose rules are: $(p1 \rightarrow r1) \wedge (p1 \rightarrow q1) \wedge (p1 \rightarrow q2) \wedge (q1 \wedge q2 \rightarrow r1)$. Here there is redundancy since either the first or last rule can be removed while maintaining the input/output behaviour.

Completeness

Let <v,{c1,c2,...,cn}> be a member of D. KB is said to be **complete with respect to** variable v iff for all c in {c1,c2,...,cn}, there exists a p in V such that $(<KB, p> \vdash v = c)$. KB is said to be **complete** iff for all variables v that have a domain in D, KB is complete with respect to variable v. As an example, consider the domain $< w1, \{1, 2, 3, 4, 5\} >$ and the rules: $(p1 \rightarrow w1 = 1) \wedge (p2 \rightarrow w1 = 2) \wedge (p3 \rightarrow w1 = 3)$. These rules are not complete with respect to variable w1 since there is no input state that would lead to the inference of either $(w1 = 4)$ or $(w1 = 5)$. On the other hand, the knowledge base consisting of the rule: $(p1 \rightarrow w1 = u1)$ is complete with respect to w1 since, in the absence of any constraints, u1 can take on the values $\{1, 2, 3, 4, 5\}$.

KB-Reducer3 Procedure

Now that we have defined knowledge bases and their anomalies, we will define a procedure that analyzes a knowledge base. The KB-Reducer3 procedure first computes the labels of a knowledge base, and then analyzes those labels looking for the various types of anomalies. Before describing this procedure, we need some more definitions.

Generalized Conclusions and Labels

A **generalized conclusion** for KB is a kbr-wff of the form $(Var = Expr)$ or (Lit), where Var is in V3, Expr is a kbr-expression over Inputs, and Lit is a literal over P3. Let GK be the set of generalized conclusions, and g be some element of GK. For any input-state i, let **g(i)** denote the conclusion resulting from the arithmetic evaluation of the expression obtained after substituting the values given in i for the variables in g. A **literal over** Inputs is a literal formed from Inputs and the standard arithmetic relations and functions (eg., $=, >, <, +, -, *, /$, etc.). A **product term over** Inputs is a conjunction of literals over Inputs. A kbr-wff p is an **implicant** for g given KB iff (1) p is a product term over Inputs, (2) $(C \wedge p)$ is satisfiable, and (3) for all v in V, if v satisfies p then $(<KB, v> \vdash g(v))$. Let **imp(g,KB)** denote the set of implicants for g given KB. A kbr-wff p is a **prime implicant** for g given KB iff p is in imp(g,KB) and for all q in imp(g,KB), if $(p \models q)$ then $(q \models p)$. A prime-implicant for g is also called an **environment** for g. Syntactically, a prime-implicant is a minimal implicant; semantically, a prime-implicant contains a maximal set of models. The **label** for g is the set of environments for g. A label can be thought of as being in minimal disjunctive normal form (dnf). The label for g is a succinct representation of the set of input states that leads to the derivation of the conclusions described by g. As an example, consider the rules:

$$\begin{aligned}&(r \wedge p \rightarrow s)\wedge\\&(r \rightarrow s)\wedge\\&(x > 0 \rightarrow y = x)\wedge\\&(s \rightarrow y = 1)\wedge\\&(y > z + 10 \rightarrow t = y)\end{aligned}$$

The following labels describe this rule base:

$$\begin{aligned}&\text{label}(s) = r\\&\text{label}(y = x) = x > 0\\&\text{label}(y = 1) = s\\&\text{label}(t = x) = (x > z + 10) \wedge (x > 0)\\&\text{label}(t = 1) = (1 > z + 10) \wedge s\end{aligned}$$

The first two rules illustrate minimality of environments. Both (r) and $(r \wedge p)$ are implicants for s, but only (r) is a prime implicant. The last rule illustrates that a single rule may result in the need to find labels for multiple generalized conclusions. As another example, let us reconsider the knowledge base given at the beginning of this paper. The labels for some generalized conclusions are:

$$\begin{aligned}&\text{label}(q1) = (p1 \wedge \neg p2 \wedge (u1 < u2))\vee\\&\qquad (p1 \wedge \neg p2 \wedge (u2 < u3))\\&\text{label}(v1 = u2) = p2 \wedge (u2 > 0)\\&\text{label}(\neg r1) = (p3 \wedge p1 \wedge \neg p2 \wedge (u1 < u2))\vee\\&\qquad (p3 \wedge p1 \wedge \neg p2 \wedge (u2 < u3))\vee\\&\qquad (p2 \wedge (u2 = 1))\\&\text{label}(w1 = u1 + u2 + u3) = p2 \wedge (u2 = 1) \wedge \neg p3\\&\text{label}(r2) = p2 \wedge (u2 = 1) \wedge p1\end{aligned}$$

The first two are straightforward. The third one illustrates some aspects of the definitions, and gives an idea as to how the general algorithm for computing labels works. Consider each disjunct in the antecedent of the rule. For $(p3 \wedge q1)$, we plug in the label for q1 and convert to dnf, yielding $((p3 \wedge p1 \wedge \neg p2 \wedge (u1 < u2)) \vee (p3 \wedge p1 \wedge \neg p2 \wedge (u2 < u3)))$. For v1=1, we substitute u2 for v1 and then conjoin the label for v1=u2, resulting in $p2 \wedge (u2 > 0) \wedge (u2 = 1)$, which is equivalent to $p2 \wedge (u2 = 1)$.

Rule Dependencies

Given two kbr-rules

$$\begin{aligned}&r1 : \mathit{condition}1 \rightarrow \mathit{action}1,\\&r2 : \mathit{condition}2 \rightarrow \mathit{action}2,\end{aligned}$$

rule r1 **depends-on** rule r2 iff either (1) action2 is a literal involving propositional symbol p2, and p2 is referenced in condition1, or (2) action2 is of the form v2 = expr2, action1 is of the form v1 = expr1, and v2 is referenced in either condition1 or expr1. This rule dependency relation induces a partial order on rules. A KBR knowledge base is **acyclic** iff there is no rule r such that <r,r> is in the transitive closure of the rule dependency relation. The KB-Reducer3 algorithm is not defined for cyclic knowledge bases. From here on, we will assume that our knowledge bases are acyclic. Given a kbr-rule r, we recursively define:

level(r) = 0 iff r depends on no other rules

level(r) = 1 + the maximum level of any rule that r depends on

Reducing the Rule Base

The algorithm for computing the labels of a KBR knowledge bases is given by the following Refine^{TM} program fragment (Refine is a trademark of Reasoning Systems Inc. of Palo Alto, CA). In this code, note that the term conclusion actually refers to generalized conclusions.

```
function compute-labels (k:kb) =
enumerate i from 0 to max-rule-level do
  enumerate r over { r2 | r2 in rule-defs(k) and
                     rule-level(r2)=i } do
    rule-cond ← make-dnf(subst-labels(rule-condition(r)));
    substitutions ← subst-for-rule(rule-action(r),rule-cond);
    (enumerate s over substitutions do
      conclusion ← apply-substitution(s,rule-action(r));
      conclusions-for-rule(r) ←
           conclusions-for-rule(r) union {conclusion};
      condition ← compute-rule-label(rule-cond,s);
      rule-label(<r,conclusion>) ← minimize-dnf(
        disjoin(rule-label(<r,conclusion>),condition)));
    (enumerate c over conclusions-for-rule(r) do
      label-for-conclusion(c) ←
        minimize-dnf(disjoin(label-for-conclusion(c),
                     rule-label(<r,c>))))

function compute-rule-label (d:dnf, s:substitution): dnf =
(enumerate c over s do
  pts0 ← { p | p in pterms-of-dnf(d) and
              subst-variable(c) in variables(p) };
  pts1 ← setdiff(pterms-of-dnf(d),pts0);
  pts2 ← { apply-substitution({c},p) | p in pts0 };
  pts3 ← { conjoin(pt2,pt3) | pt2 in pts2 and
              pt3 in label-for-conclusion(c) };
  d ← disjoin(pts3,pts1));
result ← 'false';
(enumerate pt over minimize-dnf(d) do
  if valid(pt) then result ← disjoin(result,pt));
result

function subst-labels (w:wff): wff = ...
  substitutes labels for propositional literals in wff w

function subst-for-rule (action:conclusion, condition:wff):
              set(substitution) = ...
  returns all combinations of substitutions for
              non-input variables in a rule

function minimize-dnf (d:dnf): dnf = ...
  minimize a dnf by removing non-prime implicants

function valid (p:product-term): boolean = ...
  is p satisfiable and consistent with the constraints?
```

Reducing an Example Knowledge Base

The algorithm will be described by considering the following example:

$$\begin{aligned} &r1 : (q1 \vee q2) \rightarrow q \\ &r2 : p1 \rightarrow a = 1 \\ &r3 : p2 \rightarrow a = 2 \\ &r4 : p3 \rightarrow b = 2 \\ &r5 : q \wedge a = b \rightarrow c = a \end{aligned}$$

First, rules r1 through r4 are level zero rules. The labels resulting from the processing of these rules are straightforward and given by: label(q) = $(q1 \vee q2)$, label($a = 1$) = $p1$, label($a = 2$) = $p2$, label($b = 2$) = $p3$. Now, let us consider the processing of rule r5. The function call subst-labels($(q \wedge a = b)$) returns $((q1 \vee q2) \wedge a = b)$. This gets turned into the dnf formula $((q1 \wedge a = b) \vee (q2 \wedge a = b))$. The function call subst-for-rule($(c = a)$,$((q1 \vee q2) \wedge a = b)$) returns $\{\{1/a, 2/b\}, \{2/a, 2/b\}\}$. So the innermost loop in compute-label is executed twice (once for each of the two substitutions).

Consider the first iteration, $s = \{1/a, 2/b\}$. The variable conclusion gets $(c = 1)$. Compute-rule-label works as follows. The first enumeration loop executes twice. The first time through this loop, the variable d gets assigned $(q1 \wedge (1 = b) \wedge p1) \vee (q2 \wedge (1 = b) \wedge p1)$. The second time through this loop, the variable d gets assigned $(q1 \wedge (1 = 2) \wedge p1 \wedge p3) \vee (q2 \wedge (1 = 2) \wedge p1 \wedge p3)$. Since neither of these product terms are satisfiable, the result is $false$.

Consider the next iteration, $s = \{2/a, 2/b\}$. The variable conclusion gets $(c = 2)$. Compute-rule-label works as follows. The first enumeration loop executes twice. The first time through this loop, the variable d gets assigned: $(q1 \wedge (2 = b) \wedge p2) \vee (q2 \wedge (2 = b) \wedge p2)$. The second time through this loop, the variable d gets assigned $(q1 \wedge (2 = 2) \wedge p2 \wedge p3) \vee (q2 \wedge (2 = 2) \wedge p2 \wedge p3)$. So the rule-label for $(c = 2)$ is: $((q1 \wedge p2 \wedge p3) \vee (q2 \wedge p2 \wedge p3))$.

Now, had their been other rules that could conclude $(c = 2)$, they would have been processed as above and their contribution to the label for $(c = 2)$ would have been disjoined into that result (and then minimized). For example, if there had been a rule $(q2 \wedge p3 \rightarrow c = 2)$ then the final label for $(c = 2)$ would have been $((q1 \wedge p2 \wedge p3) \vee (q2 \wedge p3))$.

Basic Anomaly Checks

The tests for redundancy and consistency are similar to KB-Reducer2 (Ginsberg & Williamson 1989). As there, a rule is redundant iff all of its rule instances are redundant. A rule instance (a substitution instance of a rule) is redundant iff it does not uniquely contribute to some environment in the label for the generalized conclusion of that rule instance. This test is (one of the reasons) why we save rule-labels for each rule during the computation of labels.

For inconsistencies, we look at the labels for contradictory conclusions; ie., either $(p \wedge \neg p)$ or $(v = expr1 \wedge v = expr2)$ where expr1 and expr2 can not possibly be equal (as an example, we check for expr1 and expr2 being two different constants). Then we do the subset and union tests (these names are historical (Ginsberg 1988 and Ginsberg & Williamson 1989)) on the labels for those conclusions. As an example, suppose we have label($v = 1$) = $(e1 \vee e2 \vee ... \vee en)$, and label($v = 2$) = $(f1 \vee f2 \vee ... \vee fm)$, where the ei and fj are environments. The subset test checks to see if any of the environments in one of the labels entails an environment in the other label. The union test checks to see if there is some combined environment $(ei \wedge fj)$, for some i and j, that is satisfiable. While the union tests clearly subsume the subset tests, they are kept as separate checks, since we have found that subset tests tend to indicate the inconsistencies that are the most problematic.

For completeness checks, we look at each of the kbr-domains $< v, \{c1, ..., cn\} >$. For all c in $\{c1, ..., cn\}$, v=c can be concluded by the knowledge base iff there is some generalized conclusion $v = expr$ in conclusions-for-variable(v) such that there is some environment p in label($v = expr$) such that $(p \wedge c = expr)$ is satisfiable.

Other Reported Anomalies

In addition, KB-Reducer3 reports on other anomalies found in a knowledge base. For example, it reports on undefined variables, assignments to input variables, invalid conclusions (that violate some kbr-domain), unfirable rules, rules that are subsumed by another rule, unused variables, uncomputed variables, dead-end rules, and overlapping rules. This latter case would find that the rules $(p \vee q \rightarrow w) \wedge (q \vee r \rightarrow w)$ are overlapping since they both contribute q to the label for w. While this is not necessarily a problem in a knowledge base, it is sometimes worth pointing out to the maintainer of a knowledge base that such overlaps exist.

Use of Limited Theorem Proving

An important detail that has been glossed over involves the use of a theorem prover to determine (1) if an environment is consistent, and (2) if one environment entails another environment. Given that such issues are undecideable for first order logic with equality and arithmetic, we decided to use some fast heuristic rules to determine satisfiability and entailment properties. In general, our strategies may lead us to miss some anomalies, but only rarely do we incorrectly report something as being anomalous. Currently, for satisfiability, the problem we may run into is saying a product term is satisfiable when it is not, while for entailment, we may say that one product term does not entail another product term when in fact it does. Let us examine both cases to determine the implications of an incorrect answer to these questions.

Satisfiability is used in three different places in KB-Reducer3. Primarily it used to prune invalid environments out of labels, however it is also used in the union inconsistency tests, and in completeness checks (to see if some expression can result in some constant value). In the latter case, an incorrect answer to the satisifiability question may result in overlooking some incompletenesses. In the second case, we may point out some inconsistencies (from union tests) that can not actually occur. It is the first case that leads to the most problems. Basically, this allows a label to contain an invalid environment. This may result in (1) reporting inconsistencies that can not actually happen, (2) reporting of redundancies that are not, and (3) overlooking some cases of incompleteness. In practise, this has not occurred much, and when it does, we have been able to modify our heuristic rules to accommodate the cases that we encounter. However, we must recognize that KB-Reducer3 occasionally reports errors that are not really errors, and it may overlook some errors.

Entailment is used in three different places in KB-Reducer3. Primarily, it is used to minimize labels, but it is also used in the inconsistency checks (in the subset tests), and in the redundancy checks (to see if a rule contributes to a label). An incorrect answer to entailment in the first case does not lead to the situation where we report a problem that is not real. In the second case, we may miss an inconsistency. In the latter case, we may both miss a redundancy and report one that is not a real case of redundancy. Again, we must recognize that our procedures occasionally report errors that are not real errors.

This issue is one of the most problematic issues for KB-Reducer3. There are fundamental computational hurdles involved with reasoning over first order logic with arithmetic. In an effort to be pragmatic, we have used some heuristic techniques. Occasionally these lead to incorrect results. This may make a user of KB-Reducer3 question the value in using the tool. We hope that this does not happen much, and when it does, we hope the benefit of using the tool far out-weighs the inaccuracies and costs associated with using the tool. So far, this has been our experience.

Practical Use of KB-Reducer3

KB-Reducer3 is written in about 2,700 lines of Refine^{TM} code. It runs on either the IBM RS 6000 or the Sun Sparc Station 2. It took about 6 person months to write the initial version of KB-Reducer3, and an additional 3 person months to fine tune the algorithms and get it ready for production use. An additional person month was required to write a language translator that takes the actual source code of production knowledge bases and translates them into the language that is accepted by KB-Reducer3. While KB-Reducer3 is not yet in full production use, planning is underway to determine how best to integrate it into our knowledge base maintenance process.

However, KB-Reducer3 has been used to analyze six knowledge bases. One knowledge base with 123 rules was analyzed in just over 5 minutes. Another knowledge base with 156 rules was analyzed in roughly 120 hours (elapsed wall clock time). In general, we have found that simply using the number of rules in a knowledge base as a gage of the complexity of KB-Reducer3 is inadequate. This metric does not gage the interconnectivity of those rules, etc. In general, most of the knowledge bases that we have tried to analyze have bogged down in the label computation part of the processing, and have not yet made it through to the error analysis.

This has led us to come up with two approaches to the partial analysis of a knowledge base. In the first approach, the analyst interrupts the label computation, sets a global boolean variable, and then resumes the computation. Special code in the reduction process periodically checks this global variable, and when it is found to be set, the code cleans up some global data structures, skips the remaining rules, and proceeds to the error analysis part of KB-Reducer3. The error analysis will be valid for the subset of rules that have been processed; ie., any reported errors in this subset will be errors in the original knowledge base (however, there may be errors in the full knowledge base that will be missed). In practice, this approach has allowed us to analyze roughly two-thirds of the rules in the knowledge bases that we had been unable to completely analyze.

A second approach to the partial analysis of knowledge bases is to create subsets of large knowledge bases, and then analyze those subsets completely. We created two tools for creating subsets of a knowledge base. The first one is given a set of rules, and returns all those rules that those rules depend on (recursively). The second one is given a set of variables, and returns all those rules necessary to completely determine the values for all those variables. While the approach described in the previous paragraph gives a horizontal slice through a rule base, the approach described here gives a vertical slice.

So what have we found in these analyses? Virtually all error types were found in these knowledge bases; inconsistencies, redundancies, incompletenesses, etc. KB-Reducer3 has pointed out multiple areas for improvement in the logic of our knowledge bases. In using this tool, we were reminded that this sort of analysis simply points out symptoms of errors, and not their causes. Errors such as redundancies may be caused by many situations (for example, an analyst left off a condition on a rule). Thus, the results of an analysis by KB-Reducer3 requires careful interpretation by the knowledge engineers who are developing or maintaining the knowledge bases. See (Dahl & Williamson 1993) for a more complete discussion of the use of KB-Reducer3.

Conclusion

This paper describes KB-Reducer3, a tool that analyzes rule bases containing equations for various types of anomalies. We have formally defined the intended behaviour of this program, and then informally described the algorithm that it uses to search for anomalies. We briefly described our experiences in using this tool on real knowledge bases.

The bottom line on KB-Reducer3 is that it produces useful results. However, it is a very computationally intensive application, often requiring many hours of compute time. Complexity is a fundamental problem. Current research is underway on developing an incremental version of KB-Reducer3 (this work is similar to the work reported in (Meseguer 1992)). In such an approach, once the reduction of a knowledge base is computed, it is saved in a persistent object base. Then, when the knowledge base is updated (eg., to add or delete a rule), only that portion of the reduction that is affected by this change needs to be recomputed. In such a view, knowledge base reduction becomes an integral part of the development and maintenance process, with the reduction and error analysis continually proceeding as the knowledge engineers evolve their knowledge bases.

References

1. Dahl, M., and Williamson, K. 1992. A Verification Strategy for Long Term Maintenance of Rule-Based Systems. In the AAAI Workshop on Knowledge Base Verification and Validation, San Jose, CA.

2. Dahl, M., and Williamson, K. 1993. Experiences of using Verification Tools for Maintenance of Rule-Based Systems. In the AAAI Workshop on Knowledge Base Verification and Validation, Washington D.C.

3. Genesereth, M., and Nilsson, N. 1987. *Logical Foundations of Artificial Intelligence*, Los Altos, CA: Morgan Kaufmann Publishers.

4. Ginsberg, A. 1988. Knowledge Base Reduction: A New Approach to Checking Knowledge Bases for Inconsistency and Redundancy. In Proceedings of the Seventh National Conference on AI.

5. Ginsberg, A., and Williamson, K. 1989. Checking Quasi-First-Order-Logic Rule-Based Systems for Inconsistency and Redundancy. AT&T Bell Laboratories Technical Memorandum 11354-891229-02TM.

6. Meseguer, P. 1992. Incremental Verification of Rule-Based Expert Systems. In Proceedings of the European Conference on AI.