Explaining Program Execution in Deductive Systems

Tarun Arora Raghu Ramakrishnan
William G. Roth Praveen Seshadri Divesh Srivastava

Computer Sciences Department, University of Wisconsin–Madison, 1210 West Dayton Street, Madison, WI 53706, USA.

Abstract. Programs in deductive database and programming systems have a natural meaning that is based upon their mathematical reading as logical rules. High-level 'explanations' of a program evaluation/execution can be constructed to provide added functionality: (1) To debug a program by following a chain of deductions leading to an unexpected (and possibly incorrect) conclusion; (2) To follow the derivation of certain *correct* conclusions to determine why and how they are reached; (3) To identify consequences of a (typically, incorrect or unexpected) fact. This functionality can be utilized either to perform post-mortem analysis of a session, or to interactively develop programs by running queries and viewing their deductions simultaneously.

'Explanations' of programs are especially important in the context of deductive databases for three reasons: (1) These programs could involve recursion, and hence, the chain of inferences is often not evident. (2) When the input data set is large, it is very difficult for a user to inspect the data and determine which facts lead to which answers, and exactly how. (3) Such programs do not guarantee a fixed evaluation strategy, and this makes it difficult for a user to comprehend unexpected behavior of a program.

With this motivation, we have designed and implemented an explanation facility for the CORAL deductive database system. The design is based on the representation of a program evaluation as a set of derivation trees, and the facility provides a high-level explanation of the inferences carried out during program execution. A notable feature of the implementation is the boot-strapped use of CORAL in the implementation of the explanation tool. We believe that an explanation system can provide a novel approach to interactively *querying* data, and is useful even for standard relational databases.

1 Introduction

Rule-based systems offer a declarative programming paradigm and encourage a relatively high-level formulation of programs. If we consider systems based upon

The work of the authors was supported by a David and Lucile Packard Foundation Fellowship in Science and Engineering, a Presidential Young Investigator Award with matching grants from DEC, Tandem and Xerox, and NSF grant IRI-9011563.
The authors e-mail addresses are {arora,raghu,roth,praveen,divesh}@cs.wisc.edu.

extensions of Horn-clause rules, a *program* can be understood as a collection of statements of the form: 'If certain facts are true, additional facts can be inferred'. Such a reading of the program is natural and non-operational.

Based on this semantics, *explanations*, that visualize the results of program execution at a suitably high level, can be constructed. Such a capability is useful in a variety of situations. For example, it can be used to debug a program, by following a chain of deductions leading to an unexpected (and possibly incorrect) conclusion. Once an incorrect conclusion is pinpointed, the system can be used to identify other conclusions that arise from it. It can also be used to understand how certain *correct* conclusions are reached — for example, just as the parse tree for a sentence can be viewed as an explanation of why a certain string is accepted as a sentential form, so too, the facts deduced by a rule-based expert system can be 'explained' at the level of the rules and facts used in their derivations.

In this paper, we describe $\mathcal{E}xplain$, a menu-driven graphical tool for visualizing fact derivations in a logic programming/deductive database language. It is designed to operate in conjunction with the CORAL deductive database system [RSSS93b], and deals specifically with (extended) Horn-clause rules evaluated using bottom-up techniques. It differs significantly from debugging tools available for Prolog-style languages, which are designed for a top-down, backtracking evaluation strategy. A major difference between bottom-up and top-down strategies (used in Prolog-like systems) is that no guarantees are offered with respect to execution order in bottom-up evaluation, and thus, some non-operational abstraction of the computation must be used in explanations. The abstraction used in the $\mathcal{E}xplain$ system is the 'derivation tree'.

The contribution of our work lies in the design and implementation of a clean and powerful explanation facility. The features for navigating derivation trees surpass what is available in other proposed systems, and we provide performance figures that show the system to be capable of handling large data sets with interactive response times. The $\mathcal{E}xplain$ system is now available along with the CORAL system.[1]

2 Motivation

There are many features of deductive databases that do not lend themselves to using traditional debuggers:

- Programs are more sophisticated than non-recursive programs in languages such as SQL, and thus, the chain of inferences is often not obvious.
- The user of a program may be familiar with the rules of the program, but typically views a large set of facts as the *input*. For example, a program analyzing stocks accepts a set of market quotations as input. The derivation of an unexpected conclusion (e.g. 'buy IBM') might depend critically upon some of the input facts. Since the input set can be large, identifying precisely which facts led to the conclusion, and how they did so, is a non-trivial task.

[1] Source code in C++ is available via anonymous ftp from ftp.cs.wisc.edu.

Visual inspection of the input data is impossible because of its size, and it is difficult to query a database system to return such information.

- Bottom-up evaluation of logic programs does not guarantee a fixed evaluation strategy, and this makes it difficult for a user to understand the behavior of a program in operational terms.

We believe that an explanation system can provide a novel approach to interactively *querying* data, and is useful even for standard relational databases.

Example 1 : Stock Market Analysis A stock is considered to be *volatile*, if its price falls by more than half in a one-week period:

$$\begin{aligned} volatile(Stock) :- \ & close(Day1, Stock, P1), close(Day2, Stock, P2), \\ & Day2 > Day1, Day2 - Day1 <= 7, P1/P2 > 2. \end{aligned}$$

A common trading heuristic is to use the moving average of the stock price to estimate whether a stock is moving upward or downward. The following rule states that a stock should be sold if its moving average falls by 10% in one week:

$$\begin{aligned} should\_sell(Stock) :- \ & movavg(Day1, Stock, A1), movavg(Day2, Stock, A2), \\ & Day2 > Day1, Day2 - Day1 <= 7, A1/A2 > 1.1. \end{aligned}$$

The database relation *close* contains the daily closing prices for each stock, and can be quite large. The *movavg* predicate is also computed using the *close* relation; we omit the definition for brevity. If a user types in the query $?volatile(S)$, and receives an unexpected response $S =$'$Walmart$' indicating that Walmart is a volatile stock, it is likely that (s)he will want to follow up by asking why the stock was classified as volatile.

The price graph for WalMart may involve a sudden sharp spike on a single day, which the user may decide is an aberration. The next step is to determine if there are any other reasons why Walmart should be considered *volatile*. Assuming that the above rule is the only rule defining *volatile*, this reduces to checking if there is another instantiation of this rule such that Walmart is in *volatile*. Let us suppose that there is no other such instantiation. Then, the user may very well decide that the data for that day should be ignored. However, this poses a new problem: what other inferences have been made using the closing price of Walmart on that day? For example, the moving average of Walmart may well have dropped by over 10% over the week due to the (aberrant) sharp drop on that day, leading to a decision that Walmart stock should be sold!

$\mathcal{E}xplain$ gives a user a menu-driven graphical tool to explore the derivation steps associated with the evaluation of the original query, and to thereby investigate each of the related follow-up questions mentioned above. In contrast, we invite the reader to consider how such questions could be posed in a standard SQL system. Note that the issue is not recursion, which is the feature traditionally recognized as missing from SQL. What is missing is a way of asking, given an SQL query and a fact, 'What answers to the query depend upon this fact?' While this example illustrated the difficulty of following query execution when the volume of data is large, a similar argument holds for programs like the bill-of-materials computation, where the difficulty is due to the complexity of the computation, not the size of the data.

Example 2 : Bill of Materials Our next example illustrates the importance of having explanations for programs that are more sophisticated than typical SQL queries. It shows that even when the data set is modest in size, explanations can be useful in tracing complex derivation steps. The following program computes the bill-of-materials for a complex mechanical part by recursively computing the bill-of-materials for each of its component parts:

$$
\begin{array}{ll}
b\_o\_m(Part, sum(< C >)) & :- \ subpart\_cost(Part, SubPart, C). \\
subpart\_cost(Part, Part, Cost) & :- \ basic\_part(Part, Cost). \\
subpart\_cost(Part, Subpart, Cost) & :- \ assembly(Part, Subpart, Quantity), \\
 & \quad b\_o\_m(Subpart, TotalSubcost), \\
 & \quad Cost = Quantity * TotalSubcost.
\end{array}
$$

This kind of query is common in practice. It involves a combination of recursion and aggregation, and depends for correctness upon the acyclicity of the *assembly* relationship. It is certainly not trivial to determine how a b_o_m fact is computed from the input relations (*basic_part* and *assembly*). For example, suppose that the cost of a toilet is computed to be $800 (as has been known to be the case!). It would be most interesting to find out precisely which components or subcomponents contributed significantly to this high cost. Again, $\mathcal{E}xplain$ allows us to examine the derivation of the b_o_m tuple of interest, and to answer this question.

3 An Overview of $\mathcal{E}xplain$

3.1 The Requirements

We wanted to develop a graphical tool that is able to visualize the evaluation of a CORAL [RSSS93a] query. Such a tool could be used either for post-mortem analysis of query execution, or for co-ordinated program development and debugging. In either mode, it should support two operations:

1. Trace how a fact is *generated*, in order to pinpoint 'interesting' premises, and
2. Trace how a fact — derived or base — is *used* to generate other facts (e.g. to identify consequences of incorrect premises).

Further, it should be possible to easily interleave the two operations. The ability to present information at this level of facts and rules is important for any reasonable understanding of how answers are computed. The tool should be able to handle the full generality allowed by CORAL, including features like non-ground facts and aggregation over multisets.

Another desirable operation is to trace why a fact is *not* generated. Typically, if a certain fact is missing and a negated goal is therefore satisfied, a user may want to know why the missing fact could not be generated. This can be done in $\mathcal{E}xplain$, but is not easy to do. (See Section 6.2 for details.)

To better understand the issues involved in the design and implementation of $\mathcal{E}xplain$, a brief summary of the CORAL database system is in order. CORAL is a deductive database system that has been developed at the University of Wisconsin at Madison. It encourages the construction of queries as *modular* programs expressed in the style of (possibly recursive) logic rules [RSS92]. The evaluation of such rules is performed primarily in a bottom-up fashion, though other evaluation strategies are also supported. The system provides support for advanced features like negation and aggregation in rule bodies, and the presence of variables inside database facts. CORAL provides the user with a command-prompt interface for interactive use, and an interface with C++ to support application development. Persistence is supported for Datalog-style data via an interface to the Exodus storage manager.

3.2 Design Issues

We began with a complete and working version of CORAL, and we needed to add an explanation facility to it. The goal was to provide explanation of programs evaluated bottom-up.[2]

The following decisions were reached with regard to the design:

- The most meaningful way to describe the deduction of answers from facts is by maintaining and displaying derivation trees, that describe the instantiations of rules that lead to the generation of facts. (Derivation trees are described in greater detail in Section 4.) The interface must be graphical and menu-driven as far as possible, and reflect the abstraction of derivation trees. There must be the ability to view multiple derivations side-by-side and switch smoothly between the *derivation* of facts and the *use* of facts.
- The explanation tool should be an independent process that is not compiled into the rest of the CORAL system. In fact, it should be as independent of CORAL as possible, since this approach to visualization could potentially be extended to other rule-based systems. Hence, all interaction should use a clearly defined interface that is as general as possible.
- Since CORAL provides a clean abstraction of program modules, it is natural to look for explanations at the granularity of modules. This would have to be explicitly turned on or off on a per-module basis from within CORAL. The $\mathcal{E}xplain$ tool should be able to display derivation information about any module execution that the CORAL user has specified. If the explanation facility is not in use, there should be no deterioration in the performance of CORAL commands.
- In general, the implementation of the $\mathcal{E}xplain$ tool should support as many CORAL features (like aggregation, non-ground facts, etc.) as possible. Since

[2] Our tool does not currently support explanation of CORAL programs evaluated top-down. In principle, this could be done using a Prolog-style debugger.

the explanation information is going to be queried by the user, what better way to answer queries than to use CORAL to process them? $\mathcal{E}xplain$ should therefore be implemented as an independent CORAL application using the CORAL/C++ features extensively. Its basic input data interface should be files of database facts, which can be handled by the CORAL query processing engine with minimal impedance mismatch. This is also aesthetically pleasing, in that CORAL is being used to explain its own execution.

We shall describe some of these decisions in greater detail at the end of Section 7 after the implementation of the system has been presented. At that stage, we will also discuss some alternative design decisions that we could have taken, and the motivation for the choices that we made. The broad design decisions have been described here so that subsequent section on the usage and the implementation of $\mathcal{E}xplain$ might be better understood.

4 Derivation Trees

The representation of *derivations* is central to the design of the system. We use the *derivation tree* as the conceptual structure representing the generation of a fact due to the evaluation of logic rules.

Consider a derivation step in which a rule r (with n body literals) is used to infer a fact f; let $f1 \ldots fn$ be the facts used to instantiate the body of the rule. This derivation step is represented as a *tree fragment*, with root f and children $f1$ through fn, in order. (A base fact in a database relation is the root of a tree fragment with no children, since base facts can be viewed as trivial rules with empty bodies.)

A *derivation tree* for a fact is defined recursively as follows. For all base facts in database relations, the associated tree fragments are derivation trees. For all other facts f, a derivation tree consists of a tree fragment with root f, in which each child fi of f is the root of a derivation tree for fi.

From the above definitions, it should be clear that a derivation tree reflects precisely how a fact is inferred, starting from facts in database relations. *We can thus think of a derivation tree as an explanation of why or how a given fact is derived*; if there are several trees for a given fact, it can be derived in many ways, and each tree represents an explanation of one of these derivations. If a given program execution involves the generation of a particular set of facts, the set of all derivation trees of all these facts provides an explanation of the entire program. Thus, program execution can be represented by a collection of derivation trees. Obviously, such a representation of a program retains no information about the order of execution of derivations. The effects of the computation (in terms of the facts generated) are represented, but not the order in which they occurred. This reflects the model-based semantics provided by CORAL, where the meaning of a program is a model of the world that satisfies the rules of the program.

Since many derivation trees may share common sub-trees, it is highly inefficient to store the entire derivation tree for every fact generated in the course of a program execution. Tree fragments (essentially, encodings of a single rule instantiation) are recorded during program execution, and all actual derivation trees can be constructed by composing stored tree fragments in the appropriate manner. This is much more practical than storing entire derivation trees. The user of $\mathcal{E}xplain$ can essentially view any portion of the entire collection of derivation trees that corresponds to the program execution. What the tool displays at any point of time is a selective window on this collection, and the displayed portions are built incrementally from tree fragments as directed by the user.

5 How $\mathcal{E}xplain$ Is Used

5.1 Enabling Explanation during CORAL Execution

Queries to the CORAL database system are usually submitted by typing them in at a command-prompt (just as in most Prolog systems). Various commands are also permitted at the prompt, one of which is a command to turn on the generation of information that will be used by the explanation tool. This command can be specified selectively for some program modules, or generally across all modules. Complementary commands exist to turn off the generation of such information. The actual information generation involves the writing of facts to a file that will be accessed by $\mathcal{E}xplain$. There is one such 'dump' file for every CORAL program module for which explanation generation is turned on.

5.2 Layout of the User Interface

$\mathcal{E}xplain$ works in a windowing environment with a point and click user interface that consists of two basic components: a Definition Window and a Uses Window.

module anc.
export anc(bf).

$anc(X,Y) :- edge(X,Y).$
$anc(X,Y) :- edge(X,Z), anc(Z,Y).$

end_module.

Fig. 1. The *anc* module

Figure 1 shows a CORAL program module that defines the recursive ancestor relation. Figure 2 shows how the *Definition window* displays the dump of an execution of the `anc` module, showing how facts are 'defined' using program rules. There are two scrollable browsers in the Definition window. The top browser labeled `Predicates` shows a list of all the unique predicate names

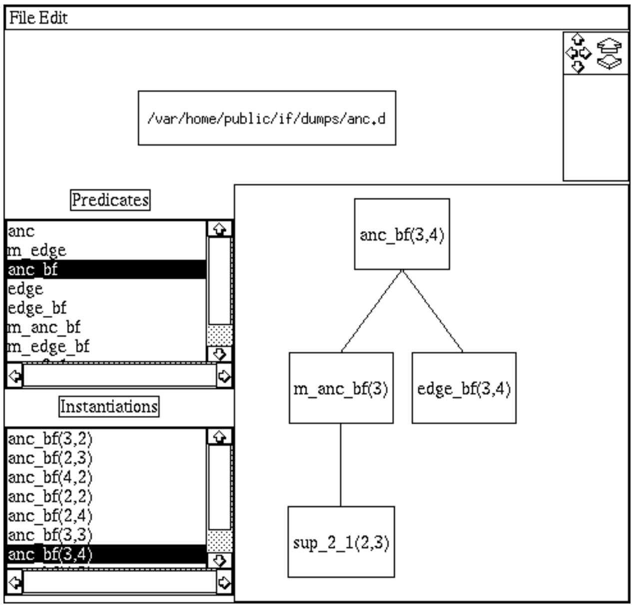

Fig. 2. The Definition Window

found in the dump file. When the user selects one of the predicates, all of the facts corresponding to that predicate are shown in the lower scrollable browser labeled `Instantiations`. When a particular fact is selected, its derivation tree is shown in the graphics window. The Definition window maintains a one level 'undo' operation for all derivation trees that are removed from the screen.

The *Uses window* shows the rule instances where a particular fact occurs in the body of the rule. An example of the Uses window is shown in Figure 3. The window has the same layout as the Definition window, though the browsers are different. The top browser shows the fact whose uses are being displayed. The lower browser shows the instantiated rules where the fact is used. To display a derivation (i.e. an instantiated rule), the user selects a rule in the lower browser, and the corresponding derivation tree is displayed. Once such a derivation tree is displayed, the Uses window provides functionality similar to that provided by the Definition window, in terms of selecting nodes and further exploring the derivation tree. For instance, by selecting $anc\_bf(4,2)$ in the example of Figure 3, rule instances in which this fact is used are shown; thus, the tree can 'grow upward'.

5.3 Typical Use of $\mathcal{E}xplain$

There are different kinds of insights that a user may wish to gain into the evaluation of a module.

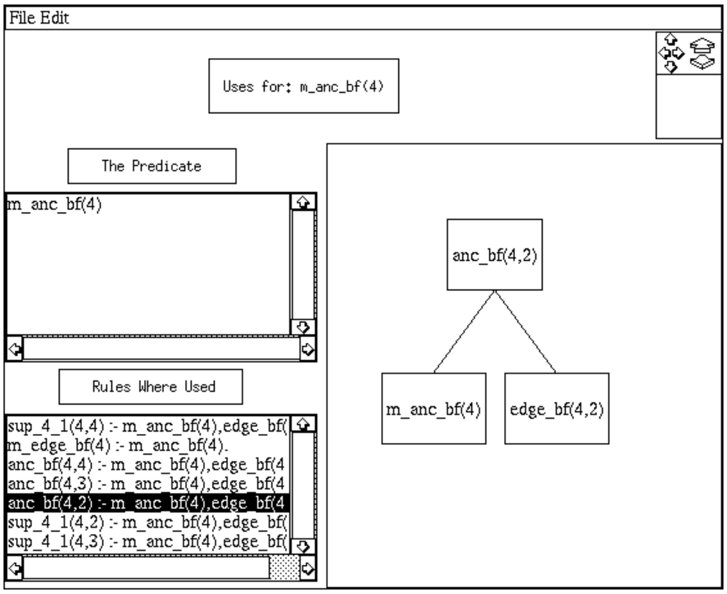

Fig. 3. The Uses Window

- The user might want to know: 'Which facts corresponding to a particular predicate have been generated?' In order to do this, the user selects a predicate name in the `Predicates` browser in the Definition window. As a result, all the facts corresponding to that predicate will appear in the `Instantiations` browser. These facts were derived as the result of successful instantiations of some rules in the module. To see a derivation of any particular fact, the user selects that fact from the `Instantiations` browser. This selection causes an appropriate instantiated rule to be displayed in the graphics window in the form of a derivation tree fragment. The root of this tree fragment is the fact derived using the rule, and the leaves are the facts that instantiate the body of the rule.
  - The leaves of the tree fragment can be expanded to show their derivations.
  - There are occasions when a fact can be derived in more than one way, and these can be scanned.
  - The derivation of a fact can also be 'un-expanded' by selecting the root fact of a derivation and pruning the derivation tree.
- The user might want to say: 'Show me all the rules where this fact is used?'. The desired fact is selected and the Uses Window is invoked to display all rules where the fact is used. Each particular use can be viewed as a portion of a derivation tree.

  (Note that the fact considered need not be a base fact — an incorrectly specified rule might lead to an unexpected derived fact even if all base facts are ‘correct’.)

6 Advanced Features

So far, we discussed simple Datalog rules that are evaluated without optimizing transformations. However, CORAL supports complex terms and functor symbols, and it also allows the use of variables inside facts. CORAL also uses various optimization strategies that include the rewriting of the original rules using transformations like magic rewriting [BR87, Ram88]. We discuss some of these issues in this section.

6.1 Program Transformations

We note that since $\mathcal{E}xplain$ records and displays actual derivations, these derivations are in terms of rules of the rewritten program, if the original program is optimized by applying some program transformation. Some program transformations are based upon special properties of the original program, and in general, it is not easy to map a derivation of the transformed program to a derivation of the original program, even when such a mapping exists. Clearly, a derivation tree in terms of the user’s original program is likely to be more understandable, and it is important to try to present explanations in the context of the original program.

An important transformation which is usually applied by default is the Magic Sets transformation [BR87, Ram88]. This transformation is purely syntactic, and can be used on an arbitrary program. The objective is to constrain a forward-chaining fixpoint computation to generate only those facts that are relevant to answering a given user query. While a general discussion of this technique is beyond the scope of this paper, we make the following observations.

1. The Magic Sets transformation modifies the original program in a way that makes it easy to convert derivations over the rewritten program into derivations over the original program. The rewriting essentially adds ‘magic’ or ‘filter’ literals to each of the rules in the original program, and introduces new rules defining the filter predicates. To convert a derivation, we can simply discard any derivation (sub-)tree with a ‘magic’ fact as the root. Although the implemented version of $\mathcal{E}xplain$ does not perform such a conversion, it is straightforward to implement, and results in the explanation of the execution at the level of the original user program.
2. In dealing with programs that contain negation or set-grouping, the rewriting introduces additional ‘done’ literals (which signal when the computation of a subgoal that must be fully evaluated is complete). Facts corresponding to these additional literals can also be discarded from derivation trees, if the explanation is desired in terms of the original program.

6.2 Handling Advanced CORAL Features

CORAL provides support for complex terms like functors (e.g. 'f(2,3)'), lists (e.g. [1,2,3]) and arbitrarily nested terms (e.g. 'f(2, g(4), [1,2,f(3)])'). Facts could also contain variables (e.g. $descend(adam, X)$ means that everybody is a descendant of *adam*); such facts are called *non-ground*. CORAL allows multisets to be created using rules, and allows aggregate functions to be expressed over these sets. A restricted form of negation is also supported in CORAL rules. These advanced features need to be handled effectively by the explanation system, so that the meaning of programs using such features can be naturally represented.

Sets and Aggregation Consider the rule:

$total(X, sum(<Y>)) :- value(X, Y).$

This rule uses the aggregate function *sum* over the set of Y values that have a common X value. Set generation via grouping, and the use of aggregate operations in the head, need to be handled in a special manner since facts are produced incrementally. For example, if there are two facts $value(1, 2)$ and $value(1, 3)$ in the database, a query $?total(1, Answer)$ is evaluated by CORAL as follows: the first *value* fact retrieved might be $value(1, 3)$, and this creates an intermediate fact $total(1, 3)$. The next fact retrieved is $value(1, 2)$, and this results in the replacement of the intermediate fact by the new fact $total(1, 5)$.

The approach we take is to record the derivation of each of these intermediate *total* facts. In each derivation that uses the intermediate fact, the intermediate fact is actually included in the body of the recorded derivation (even though the fact really corresponds to the 'current value' of the head fact). Such intermediate facts are clearly marked in recorded derivations, and only appear in derivations leading upto the 'correct' fact; in the above example, the 'correct' fact is $total(1, 5)$. While this example dealt with grouping followed by the aggregate operation *sum*, the same mechanism is used when a set-value is generated. For instance, if the previous example is modified by omitting *sum* in the head literal, the intermediate fact would be $total(1, \{3\})$, and the 'correct' fact would be $total(1, \{2, 3\})$.

Negation Consider the rule:

$p(X) :- value(X, Y), not\ q(Y).$

At run-time, evaluation of the q goal checks to see that the (instantiated q-) fact is not derivable. If such a fact is derivable, this rule instantiation is unsuccessful. If such a q-fact is not derivable, this rule instantiation is successful, and is recorded. For example, if $q(5)$ cannot be derived, and there is a fact $value(1, 5)$, then the following derivation is recorded:

$p(1) :- value(1, 5), not\ q(5).$

Explain treats '*not* $q(5)$' as a base fact, in that a tree containing this fact cannot be further expanded by expanding this fact. Thus, while it is clear that $q(5)$ could not be derived, the reasons for this must be pursued independently, by examining the execution of the goal $?q(5)$. The relevant set of derivations can be identified by looking for uses of the fact $magic\_q(5)$ if the computation uses one of the Magic Sets related rewriting algorithms. Essentially, the generation of the goal $?q(5)$ corresponds to the generation of the fact $magic\_q(5)$; this fact triggers all subsequent derivations in the solution of this goal. (For details, we refer the reader to [BR87, Ram88].)

In general, a weakness of *Explain* is that it does not provide direct support for understanding why some facts are 'missing', i.e., why some facts that a user expects to be generated are not. As sketched in the previous paragraph, the 'relevant' portion of the computation can be examined if the user has some knowledge of the Magic Sets transformations, but this is not entirely satisfactory. Since the information dumped by *Explain* completely describes the computation of negative goals, it should be possible to provide a more friendly interface to this information that enables a user to explore "why not" queries without knowledge of Magic Sets. This is an important direction for future work. Shmueli and Tsur [ST90] consider "why not" queries in a broader sense — "Why is a fact missing from the least model?", instead of "Why wasn't a fact generated (in the computation of a given query)?" — and suggest that the user's expectations must be captured in some 'intended model', and an explanation tool should provide ways for the user to test where the actual computation diverges from this model.

7 Overview of the Implementation

The *Explain* system contains two components. The first component is a set of modifications to the CORAL run-time system that support dumping of instantiated rules into a file during program execution. The second component is a tool that allows the user to visualize the information in the dump file. In this section, we describe the modifications made to CORAL to facilitate dumping, as well as the implementation of the visualization tool.

7.1 Dumping Fact Derivations

Dumping of fact derivations is performed on a per-module basis and is done only when the user specifies the module for which the fact derivations need to be viewed.

The derivations for each fact corresponding to a predicate are stored in two relations that are unique to that particular predicate. The two relations are called *def*_`predicate-name` and *use*_`predicate-name` and have two fields each. The def relation has in its first field the result fact. The second field contains the derivation tree fragment, stored as a list. The def relation is used to retrieve the derivations of a given result fact. The use relation contains in its first field any fact that was part of the derivation of the result fact, i.e. any fact in the body of a rule used to derive the result fact. The second field contains the derivation list, of which the value in the first field is a part. The *use* relation is used to retrieve the derivations where a given fact is used. For example, consider a derivation of the form:

$anc(2,3) :- anc(2,1), edge(1,3).$

One tuple of the def_anc relation and two tuples of the use_anc relation are dumped:

$$def\_anc(anc(2,3), [anc(2,3), anc(2,1), edge(1,3)]).$$

$$use\_anc(anc(2,1), [anc(2,3), anc(2,1), edge(1,3)]).$$
$$use\_anc(edge(1,3), [anc(2,3), anc(2,1), edge(1,3)]).$$

Given these three tuples, we can easily determine how $anc(2,3)$ is derived or in which derivations $anc(2,1)$ and $edge(1,3)$ are used, by querying the two relations. This is exactly what $\mathcal{E}xplain$ does. Querying the data to retrieve derivation tree fragments involves relatively simple CORAL queries. The dump file is written at a point where a handle on an entire instantiated rule is available. This corresponds to the point at which the new fact is inserted into the relation at the head of the rule.

7.2 The Visualization Component

We now give a brief description of the implementation details of the visualization component. $\mathcal{E}xplain$ is implemented as an application of CORAL/C++. The data in the dump files is consulted and stored by $\mathcal{E}xplain$ in CORAL relations, and the CORAL/C++ interface routines are used to query, update and manipulate the data. The CORAL database system is used in three distinct ways in the implementation of the $\mathcal{E}xplain$ visualization component: embedded CORAL, consulted CORAL code, and imperative CORAL. These three components are all part of the CORAL language [RSS92]. *Embedded CORAL* is, as the name suggests, CORAL code that is embedded in C++ source. Any collection of CORAL commands can be embedded at any point within C++ code [RSSS93b]. CORAL rule programs stored in separate files are also ‘consulted’ from within the C++ code, and these rule programs can be executed from within the C++ code with the help of a library of interface routines designed to provide an ‘imperative’ interface to CORAL. The $\mathcal{E}xplain$ system is therefore an example of a significant application that is built *using* the features of CORAL.

Performance Considerations $\mathcal{E}xplain$ typically offers good interactive performance, in part because of the efficient index implementation of the underlying CORAL database system. We now provide some performance figures that show the system to be capable of handling substantial programs and data sets. There

Measure	Time
Execution time w/o recording derivations	11.57 sec.
Execution time with recording of derivations	52.43 sec.
Time to load dump file into graphical tool	235 sec.
Response time for graphical operations	$<$ 1.0 sec.

Table 1. *$\mathcal{E}$xplain* Timings

are three timings of interest in considering a system like *$\mathcal{E}$xplain*: the slowdown in program execution due to the 'dumping' of derivations, the time taken to load the dumped information when the graphical interface tool is invoked, and the response time for operations at the graphical interface (such as expanding or pruning trees). We have run *$\mathcal{E}$xplain* on a number of programs, and typically, we have noticed that execution is slowed down by a factor of about four due to the dumping of derivations, and that the response time for operations at the graphical interface is never more than a fraction of a second. The time to load data is not unduly large, but can be irksome when tens of thousands of derivations are dumped. More concretely, Table 1 gives the numbers for the a bill-of-materials program, running on a parts hierarchy of about 15,000 tuples and 100 basic parts. The computation generated about 30,000 derivation tree 'fragments'. The query and data set are from the Nail-Glue benchmark [DMP93].

Our conclusion is that a practical limit to the use of *$\mathcal{E}$xplain* is likely to stem from the user's ability to comprehend large sets of derivations, rather than the system's capacity. *$\mathcal{E}$xplain* can be made to record information on a per-module basis, in order to avoid unnecessary explanations about computations in well-understood (perhaps previously debugged) modules. It is also reasonable to expect that users will try to graphically understand the execution of a program by focusing on a relatively small computation (or a restricted portion of a large computation), and that they will extrapolate the insight that they derive from it to comprehend larger computations.

8 Discussion of Implementation Issues

8.1 Derivation of Complex Tuples

The use of CORAL in the *$\mathcal{E}$xplain* tool as the underlying database system is crucial in the context of handling complex terms. Since CORAL has the capability to handle complex terms, no attempt to flatten them is required while dumping explanation information. Similarly, non-ground facts present no problem, since a variable can be dumped out as such, and will not cause any problems to the *$\mathcal{E}$xplain* tool. We note that arithmetic and other built-in predicates are handled correctly in dumping rule instantiations. The treatment of negation and set generation has already been discussed in detail 6.2.

8.2 Choice of Schema

Our main design choice in implementing the dump facility within CORAL was the choice of the schema for the dumped data. We decided on the current approach of def and use relations, since we intended using CORAL to process the dumped data. CORAL is well equipped to handle function symbols and complex terms like lists. Thus, we could store an entire derivation fragment (i.e. an instantiated rule) as a single fixed-arity tuple in the def relation. If CORAL did not have the ability to manipulate lists, we would have been forced to flatten our tuples, and break them up into smaller pieces.

When a derivation tree fragment is displayed using $\mathcal{E}xplain$, we expect that a very common action of the user will be to grow the tree an additional level by expanding one of the leaf facts into its derivation fragment. The representation of the instantiated rule body as a list of functors makes it easy to perform this operation, since one of the list elements can be directly used as the selection attribute for a CORAL lookup to retrieve the tuple for the new fragment. CORAL implements efficient index retrieval even when the indexed attributes are fields of complex terms.

It should be noted that this schema is space-inefficient in that instantiated rule bodies are stored multiple times (in both def and use relations). Storing multiple tuples for every fact produced is certainly an inefficient use of storage, and we are looking at new schemes to make this more efficient.

8.3 Separate Process Structure

Our decision to implement $\mathcal{E}xplain$ as a separate process from the CORAL system allowed the system development to proceed at a much faster rate than if we had attempted to build $\mathcal{E}xplain$ inside CORAL. It also involved minimal changes to the CORAL system itself.

9 Conclusion

$\mathcal{E}xplain$ is very useful in identifying the source of error when a program produces an unexpected result. The erroneous result could be caused by a poorly written set of rules, or bad data in the database. $\mathcal{E}xplain$ allows the user to examine all the rule instantiations, organized into derivation trees, that contributed to the creation of the result fact. To our knowledge, no other system with all of $\mathcal{E}xplain$'s capabilities has been implemented. In this section, a discussion of related research is presented, along with possible future extensions to this work.

9.1 Related Work

In [ST90], a debugging system for the LDL deductive database is presented. One of the operations supported by the LDL debugger is to take a generated fact which the user believes to be incorrect, and return an instantiated rule with this fact as the head. The user can then repeat this process, identifying one of the facts in the body of the instantiated rule as an incorrect fact. This operation corresponds in $\mathcal{E}xplain$ parlance to selecting a fact in the Definition Window to expand the current derivation tree. However, the LDL debugger is more limited in the following respects: whereas $\mathcal{E}xplain$ allows expansion and pruning of trees by choosing any 'fringe' node of a displayed portion of a tree, the LDL debugger only allows the expansion of a single path in the tree; the LDL debugger has a textual, non-graphical interface; and more importantly, the LDL debugger does not provide any support for the operations supported by the Uses Window abstraction in $\mathcal{E}xplain$. (Thus, it is not possible to identify the consequences of an incorrect, or unexpected, fact.)

Further, the outline of the LDL debugger implementation[3] indicates that the set of facts generated during a computation is dumped rather than the set of actual derivations. This means that the rules have to be used as indexes into the set of saved facts while constructing 'explanations' (using our terminology); thus, the explanation component is more closely tied to the underlying system than is the case with $\mathcal{E}xplain$. This design decision has some important implications — notably, the fact that the actual derivations are not recorded decreases the utility of the debugger in identifying errors in the run-time system. For example, if there is a program rule and a set of facts that can be used to instantiate the rule body, this represents a derivation that ought to be carried out (and is so identified by the explanation component of LDL). However, if this rule instantiation is in fact not carried out due to a system error, this error cannot be detected unless actual derivations are recorded! The separation of the dumping and the explanation components in $\mathcal{E}xplain$ also has significant advantages with respect to extending the system to offer more functionality or to deal with other kinds of rule-based systems.

An interesting aspect of [ST90] is that the issue of 'missing facts', namely facts that the user expects to see generated, but that are not generated, is addressed. It is pointed out that given a missing fact, a partly instantiated rule can be returned to the user, and if the user instantiates this rule in accordance with his/her expectations of how the computation should have proceeded, the debugger can assist the user in searching 'down' a path in a derivation tree to identify a missing database fact. We have not addressed the issue of missing facts, since our focus has been on recording and presenting the derivations actually carried out during execution. However, we note that if the missing fact is associated with a negative goal, then the set of derivations relevant to (an attempt to generate) the missing fact can indeed be identified, as discussed in Section 6.2.

There has also been some interesting work with respect to the understanding of null answers to a query, and the identification of erroneous pre-suppositions contained in the query [Mot86, Mot90]. This is closely related to the issue of 'missing facts', and our research does not address this aspect of query explanation.

[3] This is described only briefly in [ST90], and the debugger is not available with the LDL system we have.

There is also a connection between $\mathcal{E}xplain$ and Prolog debuggers (see e.g., [Sha83, Sta, Duc92]), since both are used to debug logic programs, but there are many important differences. Deductive databases provide a model-based semantics, where the meaning of a program is a model of the world that satisfies the rules of the program. This semantics is independent of the particular execution strategy employed, and in fact, no assumptions can be made about the order in which facts are generated. Given this semantics, derivation trees are the best way to represent program computations, and in $\mathcal{E}xplain$, the focus is on recording the *set* of derivations, rather than the order in which these derivations occur. A Prolog-like system, on the other hand, provides explicit guarantees about the execution strategy, and this allows programmers to think and program in an imperative fashion. In such an environment, the order in which derivations occur is crucial to the understanding of the execution. [4]

Thus, the set of derivation trees (as defined here) is not an appropriate abstraction of a Prolog execution. In fact, given the Prolog strategy of backward chaining, it is difficult to efficiently generate explicit derivation trees. First, Prolog's depth-first search does not guarantee termination even if all derivation trees are finite (for example, programs without function symbols or arithmetic). (Iterative evaluation, on the other hand, ensures that all derivation trees are enumerated, and that evaluation terminates if all the trees are finite.) Second, the strategy of storing fragments and composing trees "on demand", which is essential for efficiently representing the set of derivation trees, is not easy to implement. While it is possible to write a simple Prolog meta-interpreter that constructs a derivation tree corresponding to a given execution path, this does not allow the kind of "browsing" of derivation trees that $\mathcal{E}xplain$ supports.

9.2 Future Work

As discussed in Section 6.2, it is desirable to provide a friendly interface for exploring why certain facts are not generated. At a broader level, it is our belief that derivation trees closely mirror the manner in which users think of rule-based programs. Extending our ideas to other rule-based systems offers an interesting challenge. Some other extensions are discussed below.

Extending the Implementation The current implementation of $\mathcal{E}xplain$ has acceptable performance, but can be improved upon. The storage of derivations should become more space efficient by retaining a single copy of each instantiated rule in a separate table. This will mean a noticeable saving of space since currently we are storing multiple copies of rule instantiations. This will however be at the cost of some increase in the response time.

A problem arises when a large number of derivations are dumped. A large number of facts may have to be displayed in the browser, and scrolling through them to find a desired fact may be difficult. The display of facts, which was

[4] The fixed execution strategy also itself to stack-based debuggers and tracing tools [Duc92, DE88, Sta].

intended to support cooperative querying, could in this case produce a most uncooperative system! In the future, we expect to add a feature that will allow the user to filter out all 'uninteresting' facts and only view the ones of interest. Since all dumped data is stored as CORAL relations, such a filter can be implemented using CORAL's query processing facilities.

Modes of Operation The explanation facility will be be expected to satisfy different requirements in different contexts. It should be usable either for post-mortem analysis of program execution, or for coordinated program development and debugging. The first mode is useful when the user wishes to run programs, and when (s)he wants to see why the particular answer was generated. The dump files are created and can be analyzed at some later time. Such a post-mortem mode of operation is assumed for the example that we described in Section 5.

There are situations (e.g., when programs are being developed or when the CORAL system is being debugged) when it is useful to operate interactively, switching between CORAL and $\mathcal{E}xplain$. The user runs a query, sees how it is evaluated, performs some appropriate modification, and runs it again. This kind of alternation between query evaluation and answer explanation is easily possible in a windowing environment.

There is a third mode of operation which might be desirable, but is not currently implemented. Consider a CORAL program that for some reason results in *infinite looping* behavior. It would be very useful to be able to view the state of the computation at any given time, in terms of the derivation trees that have been generated upto that point.

Implementing this functionality would involve establishing a communication channel between CORAL and the $\mathcal{E}xplain$ tool (currently, the two communicate through the file system, which is ideal for a post-mortem mode of operation). It is quite easy to establish two-way communication between the two processes instead. Computation would proceed in lock-step, with every successful derivation accompanied by the instantiation information being sent to the tool, and an acknowledgement being returned. The acknowledgement could be delayed while the user examines and comprehends the current information displayed by the tool. Here again, there is more than one possible design choice. Using two-way communication, the $\mathcal{E}xplain$ process could step through the execution of the program one derivation at a time. The other approach is to create fixed 'break-points' in CORAL at which the execution of a particular query will stop and wait for the explanation tool to send it a response.

This mode could be used by someone with reasonable knowledge of CORAL's execution algorithms to debug either a CORAL program or the CORAL system itself. It is essential for debugging programs that do not terminate. Adding this mode of operation should require little effort, since most of the functionality is already present in both systems.

Acknowledgements: We thank Mike Carey for discussions and suggestions about an early version of $\mathcal{E}xplain$. Mireille Ducassé of ECRC in Munich and François Staes of the University of Antwerp, Belgium responded generously to our literature request on logic and prolog debugging on the `comp.lang.prolog` USENET newsgroup. Eben Haber helped us understand the intricacies of InterViews.

References

[BR87] Catriel Beeri and Raghu Ramakrishnan. On the power of Magic. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 269–283, San Diego, California, March 1987.

[DE88] M. Ducassé and A. M. Emde. A review of automated debugging systems. In *Proceedings of the 10th International Conference on Software Engineering*, pages 162–171, April 1988.

[DMP93] Marcia A. Derr, Shinichi Morishita, and Geoffrey Phipps. Design and implementation of the glue-nail database system. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1993.

[Duc92] M. Ducassé. A general trace query mechanism based on prolog. In *Proceedings of the Fourth International Symposium on Programming Language Implementation and Logic Programming*, August 1992.

[Mot86] Amihai Motro. Seave: A mechanism for verifying user presuppositions in query systems. *ACM Transactions on Office Information Systems*, 4(4):312–330, October 1986.

[Mot90] Amihai Motro. Flex: A tolerant and cooperative user interface to databases. *IEEE Transactions on Knowledge and Data Engineering*, 2(2):231–246, June 1990.

[Ram88] Raghu Ramakrishnan. Magic templates: A spellbinding approach to logic programs. In *Proceedings of the International Conference on Logic Programming*, pages 140–159, Seattle, Washington, August 1988.

[RSS92] Raghu Ramakrishnan, Divesh Srivastava, and S. Sudarshan. Coral - control, relations and logic. In *Proceedings of the International Conference on Very Large Databases(VLDB)*, pages 238–249, 1992.

[RSSS93a] Raghu Ramakrishnan, Praveen Seshadri, Divesh Srivastava, and S. Sudarshan. *The CORAL User Manual*. University of Wisconsin,Madison, 1.0 edition, January 1993.

[RSSS93b] Raghu Ramakrishnan, Divesh Srivastava, S. Sudarshan, and Praveen Seshadri. Implementation of the CORAL deductive database system. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1993.

[Sha83] Ehud Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, Cambridge, Massachusetts, 1983.

[ST90] O. Shmueli and S. Tsur. Logical diagnosis of LDL programs. In *Proceedings of the Seventh International Conference on Logic Programming*, 1990.

[Sta] F. Staes. *The Interactive LOCO Debugger : User Manual*. University of Antwerp, UIA.

This article was processed using the LaTeX macro package with LLNCS style