Generating Deductive Database
Explanations

Sarah Mallet\textsuperscript{1}, Mireille Ducasse\textsuperscript{1}
IRISA/INSA

Abstract

Existing explanation systems for deductive databases show forests of proof trees. Although proof trees are often useful, they are only one possible interesting representation. We argue that an explanation system for deductive databases must be able to generate explanations at several levels of abstraction. One possible and well known technique to achieve this flexibility is to instrument meta-interpreters. It is, however, not often used because of its inefficiency. On the other hand, deductive databases often generate intermediate information stored in the physical database. This information can be considered as a low-level trace giving a faithful picture of what has happened at the relational level. The deductive reasoning is lost but can be very easily recovered by a meta-interpreter. In this article we describe a technique to generate explanations by integrating a relational trace and an instrumented meta-interpreter. The expensive aspects of meta-interpretation are reduced by the use of the trace which avoids many costly calculations. The flexibility of meta-interpretation is preserved, as illustrated by the generation of three different kinds of explanations: a box-oriented trace, a multi-SLD-AL tree and abstract AND trees. This technique enables powerful explanation systems to be implemented with very few modifications of the deductive database mechanism itself.

1 Introduction

Explaining the behaviour of a program consists of showing an abstraction of its execution. Existing explanation systems for deductive databases [20, 16, 1] show forests of proof trees. Although proof trees are often useful, they are only one possible interesting abstraction.

Deductive databases (DDB) have various kinds of users: implementors who develop the deductive part, knowledge engineers who maintain the data, and end-users who query the database. An explanation system for DDB must be useful for all of them. It must therefore be able to generate explanations at several levels of abstraction. A key feature of an explanation system for DDB is therefore its flexibility.

\textsuperscript{1}Correspondance address: IRISA/INSA Campus Universitaire de Beaulieu, CS 14315, F - 35042 Rennes Cedex, France; email: \{Mireille.Ducasse\} \{Sarah.Mallet\}@irisa.fr
A well known and very flexible technique to produce explanations consists of instrumenting meta-interpreters (see for example [21]). This instrumentation can be easily adapted to users' needs, but it is in general inefficient. On the other hand, DDB often generate intermediate information stored in the physical database. This information can be considered as a low-level trace giving a faithful picture of what has happened at the relational level. In the DDB system that we studied, this relational trace was intended mainly for developers private usage. Indeed, the deductive reasoning was lost and only knowledgeable people could interpret the trace. We show in this article that the deduction can be easily recovered by a meta-interpreter hence making the low-level trace useful to more people.

Thus, we propose a technique for generating explanations which integrates a low-level trace with an instrumented meta-interpreter. The trace efficiently gives precise and low-level information about the extraction of data from the relational database. The meta-interpreter gives explanations about the deduction. The expensive aspects of meta-interpretation are reduced by the use of the trace which avoids many costly calculations. In particular, the accesses to the relational database are not repeated. When necessary, the meta-interpreter uses the intermediate information generated by the data extraction system, accessible via the relational trace. This feature is especially suited here as a DDB program handles a large quantity of data. Avoiding recalculations of these data saves a significant amount of time. In addition, the flexibility of meta-interpretation enables different traces to be easily produced, as illustrated at the end of the article.

We have implemented our technique for the Validity system based on EKS [19], in a prototype called Myrtle. Two specifics of DDB prevent usual Prolog meta-interpreters to be straightforwardly reused: set-oriented management of data and termination. These specifics are taken into account in our extension of Multi-SLD [15]: Multi-SLD-AL.

The main contribution of this work is the integration of a relational trace and a meta-interpreter. The practical impact of such a technique is important. As already mentioned, users of DDB have many different profiles. The flexibility at reasonable cost offered by our technique enables the explanations to be adapted, in particular, to end users. They will better accept the results if they can understand how they were produced. Furthermore, this technique requires very few modifications of the DDB mechanism itself.

In the following we first present the existing explanation systems for DDB and introduce our approach. We then informally describe the multi-SLD-AL resolution and the set-oriented meta-interpreter. The Validity 'relational' trace and its integration with the meta-interpreter are explained in a following part. Lastly, we show three different abstractions of executions constructed by instrumenting the meta-interpreter: a box-oriented trace, a multi-SLD-AL tree and abstract AND trees.
2 Related Work

Generally, three stages can be distinguished in a debugger [6]. First, the trace is extracted from a source program or its execution, then it is filtered to be abstracted and finally the results are presented, often with a visualization tool. In the following we present several explanation tools for deductive systems and give their characteristics with respect to these three steps.

The first explanation system for DDB we are aware of was developed for Dedex [11] by Wieland [20]. The extracted trace is dedicated to the construction of proof trees, which are the only abstraction proposed for the execution. Wieland redefines an independent inference system which generates the trace. An interface allows proof trees to be visualized. This method involves a slow explanation system, disconnected from the initial system and a fixed type of abstraction.

The Explain system [1] was developed by Arora and al. for CORAL [13]. Like the previous system, the abstraction structure is the proof tree. The implementation of the Explain trace generation consists of storing derivation information during the evaluation of the query. An efficient visualization tool allows users to navigate among the proof trees. This technique of trace generation is similar to the one used in Validity as described in section 5.1 except that the information stored for Explain is completely dedicated to the construction of proof trees. The explanation system seems reasonably efficient but it has a fixed type of abstraction. Moreover, for optimization purposes, the user program is transformed by the Magic Set transformation [14], and unfortunately the traced program is the transformed one, not the user one.

The system designed by Specht [16] for LOLA [17] also uses proof trees as explanations. Its principle is to transform the user program to insert trace generation. In contrast to the other systems, it does not modify the deductive engine. The transformed program is queried as usual. This method, very simple to implement, makes the performance of evaluation decrease when the size of the proof trees and their number grow. With this method, to extract operational information is tricky.

Some theorem provers like Satcho [10] are implemented in Prolog and have to manage an internal database. Their debugging problems are similar to the DDB explanation ones. SNARKS [7] is a graphical tool for debugging and explaining Satcho’s programs. The selected abstraction for explaining executions is a tree reflecting the principles of Satcho’s resolution. The trace is generated by instrumenting the deductive engine. The trace is completely dedicated to the visualization tool and it is not possible to construct other abstractions.

In active database systems, rules provide automatic mechanisms to react to events. In order to understand interactions between events, rules and databases, debuggers are needed. The context is approximatively the same as in DDB: rules interact with databases; the difference is that the database can
be updated during rule evaluation. Chakravarthy et al. [4] developed a tool for visualizing and explaining executions in active databases. They proposed two different causal graphs as abstractions of the interactions between the different events and rules occurring during execution. The generated trace is a general log file but it is only used for graph visualization. There are no facilities proposed to construct other abstractions. This tool is efficient, and has a general initial trace but there are no flexibility possibilities. The abstractions are subordinated to the visualization tool.

3 Explanations and flexibility

Figure 1: Explanation tools and abstractions

Figure 1.a reflects the approach of the systems discussed in the previous section. It consists of constructing an abstraction, namely proof trees, from a low-level trace. Users cannot get different points of view of the execution. In particular, if they want a global picture with the set by set database accesses, proof trees are totally inadapted. A query on a million tuple database can generate several million answers, each of them possibly generating several proof trees. Hence, the number of proof trees is unacceptable.

Figure 1.b illustrates our approach: users can chose among a scale of possible abstractions starting from the closest to implementation, ending with inputs/outputs of the program execution. Existing intermediate abstraction-s are a box-oriented trace, the Multi-SLD-AL tree which is an extension of the multi-SLD tree of Smith [15], and Abstract AND trees [2]. Alternate views could also include the abstract trees presented by Naish for declarative debugging [12] or by Comini et al. [5] for abstract debugging.

4 A Multi-SLD-AL meta-interpreter

Two specifics of DDB prevent usual Prolog meta-interpreters to be straightforwardly reused: set-oriented management of data and termination. Tuples
of the relational database are, in general, retrieved several at the same time, and not one at a time; a possibly large number of tuples can be extracted from the base during a single access. Hence, in Myrtle substitutions are managed in a set-oriented way as described in multi-SLD [15]. Lastly, the restriction to Datalog and dedicated search strategies ensure that a request on a deductive database always terminates [14]. Myrtle implements such a strategy: the SLD-AL one [19].

4.1 Principles of the Multi-SLD-AL Resolution

A DDB program defines two databases: the extensional database composed of data in the database and the intensional database defined by the deductive program. Tuples of the database are defined by predicates called database predicates. Predicates defined by rules of the deductive program are called derived predicates. The previous notions are illustrated on Fig. 2. This example is the classical ancestor example where \( anc/2 \) is a derived predicate, defined with rules and \( p/2 \) a database predicate defined by tuples stored in the database.

\[
\begin{align*}
(c1) \quad & anc(X,Y) :- p(X,Y). \\
(c2) \quad & anc(X,Y) :- anc(X,Z), p(Z,Y). \\
\end{align*}
\]

\[
\begin{align*}
\text{Database tuples} & \quad p(a,b) \quad p(e,f) \quad p(f,g) \\
& \quad p(b,c) \quad p(d,b)
\end{align*}
\]

Figure 2: Definition of the \( anc \) program

There exist different techniques to solve the deductive part of the different DDB systems, see for example the survey of Ramakrishan and Ullman[14]. In particular, the system we are working on, Validity, is based on SLD-AL resolution (SLD with test of Admissibility and resolution on Lemmas) described by Vieille in [19]. This form of resolution is an optimization of SLD resolution which cuts infinite branches from the search tree.

The aim of this resolution is to do the calculations only once. Therefore two kinds of information have to be stored: goals, solved or in the process of resolution, and solutions, produced by the solved goals. During the resolution new goals are compared with the set of stored goals. When a goal is a variant of one of the stored goals, it is called non-admissible and it is solved using both the solutions already produced and those that will be further produced. The solutions are called lemmas. Only derived predicates are concerned by the notion of non-admissibility because they are the only ones that can induce non termination. Goals using database predicates are not stored.

The SLD-AL resolution manipulates tuples one by one. On the example \( anc/2 \) of Fig 2, to solve \( p(X,Y) \), one branch of resolution is created by tuple in the database unifying with \( p(X,Y) \). However, in DDB connected to a relational database, database accesses are achieved set by set and in this case only one branch is actually created to solve \( p(X,Y) \). To express this set manipulation, we introduce substitution sets in the SLD-AL resolution in the
same way as in multi-SLD resolution presented by Smith [15].

The new resolution is called multi-SLD-AL and the resulting search tree of this resolution is called a multi-SLD-AL tree. A node of the tree is labeled with the resolvent and a set of substitutions. An edge is labeled with the type of the transition: the clause, if the solved goal is a derived predicate; database, if it is a database predicate; lemmas, if it is a non-admissible goal solved using lemmas and builtins, if it is a built-in predicate.

Figure 3 represents a multi-SLD-AL tree for the query \( \text{anc}(X,Y) \) on the program of Fig. 2. The first branch of the tree corresponds to the use of the clause \( c1 \). At the end of this branch, some solutions (lemmas) are produced for the atom \( \text{anc}(X,Y) \). Lemmas are global to the whole search space of the multi-SLD-AL resolution. The second branch, created using clause \( c2 \), uses the produced lemmas (1) to solve \( \text{anc}(X,Z) \) which is a variant of \( \text{anc}(X,Y) \). The substitution set is enriched with these lemmas. After the evaluation of \( p(Z,Y) \), the result of the database access \( \{\{Z/a,Y/b\}, \{Z/b,Y/c\}, \{Z/e,Y/f\}, \{Z/d,Y/b\}, \{Z/b,Y/g\}\} \) is joined with the previous substitution set \( \{\{X/a,Z/b\}, \{X/b,Z/c\}, \{X/e,Z/f\}, \{X/d,Z/b\}, \{X/f,Z/g\}\} \). Some of the substitutions cannot be joined with the tuples selected during the database access, for example \( \{X/b,Z/c\} \) cannot be joined as there is no solution \( \{Z/c,Y/\ldots\} \). They are suppressed from the substitution set. As soon as new lemmas are produced, new transitions are possible. The production of lemmas at the end of the second branch creates a new possible transition from the node \( \text{anc}(X,Z), p(Z,Y) \) with an empty substitution set using the new lemmas (2). This third branch ends with a failure because the join between the substitution set and the database results gives an empty set.

Figure 3: A multi SLD-AL tree
4.2 A set oriented meta-interpreter

The Prolog meta-interpreter, which we propose for implementing the multi-SLD-AL resolution, is an extension of SLD-AL meta-interpreters introduced in [9] which did not take substitution sets into account.

The meta-interpreter can be divided in two parts: the search tree traversal and the resolution which computes answers as sets of substitutions. We first describe the traversal and then the resolution.

Figure 4 defines the traversal of the search tree of the multi-SLD-AL resolution. This traversal starts with the predicate solve/2. In a first pass, solve_goal/4 evaluates the query like a multi-SLD-AL query, leaving the non admissible goals on the side. In the second pass solve_na/2 solves the non admissible goals using lemmas.

The first argument of solve_goal/4 is the resolvent. Its structure may seem unusual. Indeed, the head of the unified clause is kept in the resolvent (hence the true ← Query). This allows lemmas to be generated easily, the information about the solved subgoal is still present in the resolvent.

The resolution of a subgoal is composed of two stages. Initially, the atom to be evaluated is chosen (select/3) then solve_atom/7, defined on Fig. 5, solves it according to its type following the four transitions previously described on the multi-SLD-AL tree. The evaluation of non admissible atoms is delayed. In this case, the goal and its environment are stored and the procedure fails in order to continue the traversal of the search tree.

Some functions calculate the new substitution sets of the states when accessing the database (access/3), evaluating built-in predicates (answer_builtin/3), unifying the head of a clause with the selected atom (unify/4) and solving the non admissible goals using lemmas (answer_set/3).

The multi-SLD-AL evaluation requires storage of some information. save_answer/2 and answer_set/3 are respectively used to store and recover the produced lemmas. save_subquery/3 and variants_subquery/3 respectively store and compare subgoals to detect non admissible subgoals. Finally save_na/4 and na/4 respectively store and recover the non admissible states.

5 Driving the meta-interpreter with the trace

The meta-interpreter is not efficient. In particular, the construction of the answers and the storage of information take a lot of time. We show here how to drive it by a relational (low-level) trace to lighten some of the problems. Information stored in the trace do not need to be rebuilt in the meta-interpreter. The non determinism in the meta-interpreter can be reduced and substitution sets are already present in the trace.

In the following we first describe the generation of the relational trace. We, then, explain how to use information stored in the trace. Finally, we discuss the synchronization of the meta-interpreter with the trace events.
/* Query resolution: multi-SLD and AL part */

    /* multi-SLD part */
solve(Query, SubstSet) :-
    <initialisations>,
    solve_goal([true <- Query], emptySet, SubstSet, Query).

    /* AL part */
solve(Query, SubstSet) :-
    solve_na(Query, SubstSet).

/* Resolution of non-admissible states */

solve_na(Query, SubstSet) :-
    IF <no new answer produced>
    THEN fail
    ELSE /* Take a non-admissible state */
    na(Atom, SubstSet0, Cont, Query),
    /* Solve it on existing lemmas */
    answer_set(Atom, SubstSet0, SubstSet1),
    /* Continue the resolution with Cont */
    solve_goal(Cont, SubstSet1, SubstSet, Query).

/* The query has been solved, a set of answers SubstSet has been produced */

solve_goal([true <- []], SubstSet, SubstSet, _).

/* A sub-goal has been solved */

solve_goal([SubGoal <- [] \ Rest], SubstSet0, SubstSet, Query) :-
    /* Production of lemmas for SubGoal */
    save_answer(SubGoal, SubstSet0),
    /* The resolution goes on */
    solve_goal(Rest, SubstSet0, SubstSet, Query).

/* Resolution of a SubGoal */

solve_goal([SubGoal <- ToSolve|Rest], SubstSet0, SubstSet, Query) :-
    /* Selection of Atom to solve local selection rule */
    select(ToSolve, Atom, RestToSolve),
    /* Resolution of the selected Atom */
    solve_atom(Type, Atom, [SubGoal <- NewToSolve|Rest], NewResolvent,
                 SubstSet0, SubstSet, Query),
    /* Continue the resolution with NewResolvent */
    solve_goal( NewResolvent, SubstSet1, SubstSet, Query).

Figure 4: Multi-SLD-AL meta-interpreter: resolution of a goal
/* Resolution of an atom of base type */

solve_atom(base, Atom, ToSolve, ToSolve, SubstSet0, SubstSet, Query) :-
  /* Test of the atom type */
  is_basis(Atom), !,
  /* Database access*/
  access(Atom, SubstSet0, SubstSet).

/* Resolution of an atom of builtin type */

solve_atom(builtin, Atom, ToSolve, ToSolve, SubstSet0, SubstSet, Query) :-
  /* Test of the atom type */
  is_builtin(Atom), !,
  /* Resolution of the builtin */
  answer_builtin(Atom, SubstSet0, SubstSet).

/* Resolution of an atom of non-admissible type */

solve_atom(na, Atom, Resolvent, _, SubstSet0, _, Query) :-
  /* Test of the non-admissibility of the state */
  variants_subquery(Atom, SubstSet0, NonAdmSet),
  /* If the state is non-admissible on all the substitution set, */
  /* other branches of resolution are cut */
  IF all_non_admissible(NonAdmSet)
  THEN <cut other branches>,
  /* The non-admissible state is saved */
  save_na(Atom, NonAdmSet, Resolvent, Query),
  /* And fail (it will backtrack to the next choice point) */
  fail.

/* Resolution of an atom of rule type */

solve_atom(rule, Atom, Resolvent, [Atom :- Body|Resolvent], SubstSet0, SubstSet, Query) :-
  /* Save the atom */
  save_subquery(Atom, SubstSet0, AdmSet),
  /* Selection of a rule to solve Atom */
  choose_rule(Head, Body),
  /* Unification of Atom and the Head of the clause */
  unify(Atom, Head, AdmSet, SubstSet).

Figure 5: Multi-SLD-AL meta-interpreter: resolution of an atom
5.1 The Validity "Relational" Trace: Generation

An ad hoc trace has been added by implementors in order to have some low level information for debugging. It describes a succession of events which reflect the interaction with the relational database. These events are of two types: management of the control flow and operations on data.

The management of the control flow gives information about the non-admissible goals. The data events give pointers to tables stored in the database. The tables contain descriptions of relations manipulated during the operations. *These tables are generated by the execution whether the trace is requested or not.* They remain accessible after the execution. Thus, at explanation time the whole information related to database accesses is available without re-executing these accesses, and this at no extra cost in terms of space.

5.2 Information in the Trace

**Avoiding database accesses** The sets of substitutions are no longer transmitted along the meta-interpreter to construct solutions. At each point of construction or enrichment of a set (namely in answer_set/3, access/3, answer_builtin/3, unify/4), the substitutions are replaced by an indication of how to obtain them from the intermediate relations if necessary. These sets can be accessed one by one on user request during the visualisation step or in the chosen abstraction or ignored. The arguments SubsitSet and SubsitSet0 are no longer used in solve_atom/5 and solve_goal/2. In the temporary relations, tuples are tuples of values. To reconstruct substitutions, these values have to be associated with the variables present in the corresponding query. That correspondence is present in the trace file.

```
The meta-interpreter

solve_goal([SG:-ToSolveR], Query) :-
manage_trace_call([Type, Atom, [SG:-ToSolveRI]],
select(ToSolve, Atom, NewTS),
solve_atom(Type, Atom, [SG:-NewTSIR], Query),
solve_goal(NewR, Query).

solve_atom(rule, A, R, [A:-BIR], Query) :-
save_subquery(A, R),
choose_rule(H, B),
unify(A, H, S).

The relational trace

- Read a new event if necessary
- Extract relevant information
  - What is the selected atom?
  - What type has it?
- Extract relevant information
- Create a choice point
- Extract relevant information
  - How to obtain the associated set of substitutions?
- Increment the current literal
```

Figure 6: Using relational trace information
Reducing the non determinism  The trace is an image of an execution, where choices were already made. It is not necessary to remake them in the meta-interpreter. The atom selected at a given resolution step is known from the trace. The selection function select/3 consists then of retrieving this information. In the same way, the type of this atom is present in the table of symbols. The clause of solve_atom/7 which is used can now be set in advance using indexing. Type in solve_atom(Type, Atom, Res, NewRes, SubstSet0, SubstSet, Query) is instantiated before invocation. Furthermore, the rule selected for the resolution of a derived predicate is also present in the trace. The choice points normally created by Prolog are no longer created. It is nevertheless necessary for the meta-interpreter to backtrack in order to follow correctly the execution. These choice points have then to be simulated: the predicate choose_rule/2 is redefined. The principle is to repeat the choice of rules until no longer rule unifies with the current goal in the trace. In the same way, the lemmas used to solve non admissible goals can also be found in the trace. Therefore the meta-interpreter does not need to manage lemmas, now save_answer/2 does nothing. For the non admissible states, the resolvent is not available in the trace, save_na(Atom, Resolvent, Query) continues to save it but does not save the associated set of substitutions anymore because they are stored in the trace.

5.3 Synchronizing the meta-interpreter with the trace events

Figure 6 sums up the connection on a small part of the meta-interpreter. In solve_goal/2, manage_trace_call/3 reads a new event if necessary, then select/3 extracts relevant information from the trace. In solve_atom/5, choose_rule/2 extracts the selected rule and a choice point is created as explained before. unify/3 gives the set of substitutions corresponding to the resolution of the atom and then increments the current literal to progress in the trace.

6 Constructing abstractions by instrumentation

Once trace information has been associated with the multi-SLD-AL resolution, the next step is to produce abstract views of executions. As already mentioned in the introduction, it is necessary to have different abstractions to adapt to different users and different debugging or understanding problems. Three possible abstractions of executions are presented. The order of presentation corresponds to a growing level of abstraction. The first one, relatively low level, is a box-oriented representation of execution. The second one reflects the operational semantics level, it is a representation of the multi-SLD-AL tree. The last one is closer to declarative semantics. The execution is abstracted by a forest of proof trees combined with substitution sets.
A box-oriented trace A box-oriented trace gives a sequence of events inspired by those proposed by Byrd in [3]. The meta-interpreter is instrumented following the tracing methods of Prolog programs described in [18, 21]. The trace format that we choose contains eight ports, including four ports for the non-admissible goals. These ports are call, fail, exit, redo, call naïa, fail naïa, exit naïa, redo naïa. The naïa suffix refers to non-admissible goals.

Instrumentation

<table>
<thead>
<tr>
<th>solve naïa(Query) :-</th>
<th>call</th>
<th>anc(X,Y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF &lt;no new answer produced&gt;</td>
<td>unify</td>
<td>anc(X,Y)</td>
</tr>
<tr>
<td>THEN fail</td>
<td>call</td>
<td>p(X,Y)</td>
</tr>
<tr>
<td>ELSE</td>
<td>unify</td>
<td>p(X,Y)</td>
</tr>
<tr>
<td>( trace(call naïa, Atom)</td>
<td>exit</td>
<td>p(X,Y)</td>
</tr>
<tr>
<td>; trace(fail naïa, Atom),</td>
<td>exit</td>
<td>anc(X,Y)</td>
</tr>
<tr>
<td>fail),</td>
<td>redo</td>
<td>anc(X,Y)</td>
</tr>
<tr>
<td>naïa(Atom, Cont, Query),</td>
<td>unify</td>
<td>anc(X,Y)</td>
</tr>
<tr>
<td>answer_set(Atom, S),</td>
<td>call</td>
<td>anc(X,Z)</td>
</tr>
<tr>
<td>( trace(exit naïa, Atom)</td>
<td>unify</td>
<td>anc(X,Z)</td>
</tr>
<tr>
<td>; trace(redo naïa, Atom),</td>
<td>fail</td>
<td>anc(X,Z)</td>
</tr>
<tr>
<td>fail),</td>
<td>fail</td>
<td>anc(X,Y)</td>
</tr>
<tr>
<td>solve_goal(Cont, Query).</td>
<td>call naïa</td>
<td>anc(X,Z)</td>
</tr>
<tr>
<td></td>
<td>exit naïa</td>
<td>anc(X,Z)</td>
</tr>
<tr>
<td></td>
<td>call</td>
<td>p(Z,Y)</td>
</tr>
<tr>
<td></td>
<td>unify</td>
<td>p(Z,Y)</td>
</tr>
<tr>
<td></td>
<td>exit</td>
<td>p(Z,Y)</td>
</tr>
<tr>
<td></td>
<td>exit</td>
<td>anc(X,Y)</td>
</tr>
</tbody>
</table>

Figure 7: Instrumentation: generation of a box oriented trace

The meta-interpreter builds a trace of the resolution of the various atoms appearing during the evaluation. The predicates solve_goal/2, solve_atom/5 and solve naïa/1 are instrumented in order to trace each resolution of an atom. The instrumentation of solve naïa/1 is presented on Fig. 7. A predicate trace/2 is used to trace information concerning the current port and the current atom. It should be noted that the proposed instrumentation is not the only possible one. It remains adaptable. The trace obtained for the example of Fig. 2 is presented on Fig. 7. At unify time, the type of transition is mentioned (rule, base, non-admissible, built-in). It actually corresponds to the first two branches of the multi-SLD-AL tree. This trace could be enriched with additional information such as the call depth or an action number for example.

The Multi-SLD-AL Tree This abstraction gives an operational view of execution in terms of a multi-SLD-AL tree (see Section 4.1). The management of the nodes occurs at the time of the resolution of the selected atom. The predicates solve_atom/5 and solve_goal/7 are instrumented to build the tree. Fig. 8 gives the instrumentation of one clause of solve_atom/7. The predicates create_node/3 and create_edge/2 are adding new nodes and new edges. An inherited argument, which is the identifier of the father node,
has been added to `solve_goal` and `solve_atom`.

\[
\text{solve_atom} \text{base, } A, B, R, Q, \text{ Father, NewCurrentNode}:=
\]

\[
! , \text{access}(A, S), \text{value}(\text{NewCurrentNode}),
\]

\[
\text{create_node}(\text{NewCurrentNode}, A, S),
\]

\[
\text{create_edge}(\text{Father, NewCurrentNode}).
\]

Figure 8: Instrumentation: generation of a multi-SLD-AL tree

Information is associated with the nodes during their creation. It can depend on the expected use of the tree. The identifier of a node is a mandatory information, then the atom, the remainder of the goal and the associated set of substitutions can be kept. Information associated with the edges concerns transitions and can be, as in the abstraction we choose, only the type of the transition. It is necessary for non admissible goals to enrich `save_na` with the identifier of the associated node. Indeed, when the evaluation of the goal is resumed again, it should be known where to hang the subtree.

**Forest of Abstract AND Trees** This abstraction consists of giving one proof tree per set of solutions to a query, that is to say one proof tree per success branch of the multi-SLD-AL tree. This structure is called abstract AND tree by Bruynooghe [2]. This view preserves the notion of set manipulation of data without operational information. It is interesting for users querying the database who are not interested in operational information but have notions of database relations. Instrumentations of the first and third clause of `solve_goal/4` and of the fourth clause of `solve_atom/7` are presented on Fig. 9. Two new arguments are added to `solve_goal/2, solve_na/1` and `solve_atom/5`: the skeleton of the abstract AND tree and the current node of this tree. The skeleton of the abstract AND tree is composed of only the solved atoms, the substitution sets are not interesting during the resolution. They are only interesting when some answers are found in the first clause of `solve_goal/4`. In this clause, `traceTree/2` instantiates the skeleton of the tree with values and constructs the corresponding tree. In the third clause of `solve_goal/4`, after selecting a new atom, `modifyCurrentNode/3` puts as current node the node which corresponds to the selected atom in the skeleton. Finally, in each clauses of `solve_atom/7`, new nodes are created. In particular, in the fourth clause, one son per atom in the body of the clause is added to the current node by `createSons/4`. On the example of Fig. 2, the obtained forest contains two trees, one per success branch of the multi-SLD-AL tree. Each abstract AND tree represents several proof trees. On the example, the two trees together correspond to eight proof trees. Users can be interested in the details of proof trees. As the number of proof trees can be very important and some of them can be really big, a tool to manipulate these proof trees can be used to help the user to consult this information. In this case a tool as proposed in *Explain* is useful [1].
Instrumentation

```prolog
solve_goal( [SG :- TS[R], Q, AbstractTree, Node] :-
  manage_trace_call(T, A, [SG :- TS[R]],
  select(TS, A, NTS),
  modifyCurrentNode(A, AbstractTree, NewNode),
  solve_atom(T, A, [SG :- NTS[R], NRes, Q, AbstractTree, NewNode],
  solve_goal(NRes, Q, AbstractTree, NewNode).

solve_goal([[true :- []], _, AbstractTree, _] :-
  manage_trace_exit(S),
  traceTree(AbstractTree, S).

solve_atom(rule, A, B, [A :- B[R], Q, AbstractTree, Node] :-
  save_subquery(A),
  choose_rule(H, B),
  createSons(AbstractTree, Node, B, NewAbstractTree),
  unify(A, H, S).
```

**Corresponding abstraction**

```
anc(X, Y)  \{\{Xa,Yb\},
          \{Xe,Yf\},
          \{Xb,Yc\},
          \{Xd,Yb\},
          \{Xf,Yg\}\}

p(X, Y) \{\{Xa,Yb\},
          \{Xe,Yf\},
          \{Xb,Yc\},
          \{Xd,Yb\},
          \{Xf,Yg\}\}

anc(X, Y) \{\{Xa,Yc\},
          \{Xe,Yg\},
          \{Xd,Yc\}\}

anc(X, Z) \{\{Xa,Zb\},
          \{Xe,Zf\},
          \{Zb,Yc\},
          \{Zf,Yg\}\}
```

> Figure 9: Instrumentation: generation of Abstract AND trees

### 7 Conclusion

We have presented a technique of explanation generation which consists of integrating a relational trace with an instrumented set-oriented meta-interpreter. The relational trace reduces the non determinism of meta-interpretation and avoids many costly calculations at the debugging stage that have been already performed at execution. The meta-interpreter allows different abstract views of the execution to be constructed by instrumentation. This technique has been illustrated on the Validity system. The meta-interpreter is grounded on the multi-SLD-AL semantics, which is an extension of the multi-SLD semantics to the AL technique. In order to connect the meta-interpreter to the trace produced by Validity, some calculation functions and some choice functions of the meta-interpreter have been modified.

**Acknowledgments** Alexandre Lefebvre, Laurent Vieille and Bernard Wappler from Next Century Media² sacrificed part of their time to explain the operation of Validity. Olivier Rideaux gave fruitful comments to an earlier version of this article.

²http://www.nextcenturymedia.com
References


