Myrtle : A Set-Oriented Meta-Interpreter Driven by a "Relational" Trace for Deductive Databases Debugging

Sarah Mallet, Mireille Ducassé

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Myrtle : A set-oriented meta-interpreter driven by a "relational" trace for deductive databases debugging

Sarah Mallet and Mireille Ducassé

IRISA/INSA

N ° 3598

Janvier 1999

THÈME 2

ISSN 0249-6399

Myrtle : A set-oriented meta-interpreter driven by a "relational" trace for deductive databases debugging

Sarah Mallet* and Mireille Ducassé*
IRISA/INSA

Thème 2 — Génie logiciel
et calcul symbolique
Projet Lande

Rapport de recherche n3598 — Janvier 1999 — 22 pages

Abstract: Deductive databases manage large quantities of data and, in general, in a set-oriented way. The existing systems of explanation for deductive databases do not take these constraints into account. We propose a tracing technique which consists of integrating a "relational" trace and an instrumented meta-interpreter using substitution sets. The relational trace efficiently gives precise information about data extraction from the relational database. The meta-interpreter manages substitution sets and gives explanation on the deduction. The expensive aspects of meta-interpretation are reduced by the use of the trace which avoids many calculations. The flexibility of meta-interpretation is preserved. It allows different profiles of trace to be easily produced.

Key-words: deductive databases, debugging, trace, multi-SLD-AL, meta-interpreter, substitution set, instrumentation

(Résumé : tsvp)

* Correspondance address: INSA- Dept Informatique, 20 av des Buttes de Coësmes, F-35043 Rennes Cedex ; email : {Mireille.Ducasse}{Sarah.Mallet}@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Myrtle : Un méta-interprète ensembliste piloté par une trace "relationnelle" pour le débogage de bases de données déductives

Résumé : Les bases de données gèrent des quantités importantes de données et, en général, de manière ensembliste. Les systèmes d'explication existants pour les bases de données déductives ne prennent pas en compte ces contraintes. Nous proposons une technique de traçage qui consiste à intégrer une trace "relationnelle" avec un méta-interprète instrumenté utilisant des ensembles de substitutions. La trace relationnelle donne, de manière efficace, de l'information précise sur l'extraction de données de la base relationnelle. Le méta-interprète ensembliste gère des ensembles de substitutions et donne des explications sur la déduction. Les aspects coûteux de la méta-interprétation sont réduits par l'utilisation de la trace qui évite beaucoup de calculs. La flexibilité de la méta-interprétation est conservée. Elle permet de produire facilement des traces de profils différents.

Mots-clé : bases de données déductives, débogage, trace, multi-SLD-AL, méta-interprète, ensembles de substitutions, instrumentation

1 Introduction

The existing explanation systems for deductive databases [22, 17, 1] give information in the shape of forests of proof trees. Although proof trees are often useful, this representation is not sufficient.

Debugging tools for deductive databases (DDB) must be useful for their various kinds of users: implementors who develop the deductive part, knowledge engineers who maintain the data, and end-users who query the database. Implementors need to get rather low-level pictures of executions of the DDB in order to understand and fix errors in the kernel. Knowledge engineers do not necessarily know the details of the implementation of the kernel, but they need to understand how the data, they add or modify, fit with the existing ones ; they therefore need to get pictures of executions abstract enough for them not to be overwhelmed by implementation details. End-users, in general, do not have an in-depth knowledge of computers, but they nevertheless have to rely on the results of the system. They, therefore, need to get very high-level views of executions to understand how the answers to their queries was produced in order to accept these answers. A key feature of a debugging tool for deductive databases is therefore its **flexibility**. It must be able to provide information and explanations at various levels of abstraction.

A well known very flexible technique to produce explanations consists in instrumenting meta-interpreters (see for example [23]). This instrumentation can be easily adapted to users' needs. However, this technique is in general inefficient. On the other hand, kernel developers usually instrument their compilers to produce some sort of low-level traces of executions. These traces are mostly intended at "private" usage and do not necessarily contain the information necessary to the explanations, but they are generated efficiently.

We propose a tracing technique which takes the best of the previous two techniques. In Myrtle, we integrate 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 and allows more abstract views to be built. The expensive aspects of meta-interpretation are reduced by the use of the trace which avoids many calculations. In particular, the accesses to the relational database are not repeated. When necessary, the meta-interpreter uses the transient information generated by the DB 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 to recalculate these data saves a significant amount of time. In addition, flexibility of meta-interpretation enables different traces to be easily produced, as illustrated at the end of the article.

Two specifities of DDB prevent usual Prolog meta-interpreters to be straightforwardly reused: *set-oriented management of data* and *termination*. Tuples of the relational DB 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. 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 strategy [20].

The main contribution of this work is the integration of a relational trace and a meta-interpreter, which is, to our knowledge, original. The practical impact of such a technique is important. As already mentioned, users of deductive databases have many different profiles. The flexibility at reasonable cost offered by our technique enables the debugger to be adapted, in particular, to end users. They will better accept the results if they can understand how they were produced. One can conjecture that such a tool can help widen the acceptance of logic programming technologies.

We developed this technique for the Validity system based on EKS [21], which uses the SLD-AL technique to ensure termination, and the rule language DEL. In this paper we consider a restriction of this language to Datalog.

In the following we first present the existing debugging systems and compare our approach with different debugging techniques. 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. Finally we show three different abstractions of executions constructed in instrumenting the meta-interpreter.

2 Related Work

Generally, three stages can be distinguished in a debugger [7]. 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 debugging tools for deductive systems and give their characteristics with respect to these three steps. We, then, discuss different approaches of debugging.

2.1 Debugging Systems For Deductive Databases

The first explanation system for deductive databases was developed for Dedex [11] by Wieland [22]. The extracted trace is dedicated to the construction of proof trees, which are the only abstraction proposed for the execution. The initial system is not exploited since Wieland redefines an independent inference system which generates the trace. Thus, the trace does not reflect the system execution. Finally, 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 selected abstraction structure is the proof tree. The implementation of the *Explain* trace generation consists in storing information on derivations during the evaluation of the query. A 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 it is described in section 4.1 except that the information stored for Explain is completely dedicated to the construction of proof trees. It induces an efficient systems but with a fixed type of abstraction. Moreover, for optimization purposes, the user program is transformed by the Magic Set transformation [2], and unfortunately the traced program is the transformed

one not the user one. In this context, final users may have difficulties to understand the corresponding proof trees.

The third system, designed by Specht [16] for LOLA [18], uses also 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 fall with the increase of the size of the proof trees and the number of these trees (a possibly large number). With this method, to extract operational information is tricky.

Some theorem provers like Satchmo [10] are implemented in Prolog and have to manage an internal database. Their debugging problems are similar to the deductive databases ones. SNARKS [8] is a graphical tool for debugging and explaining Satchmo's programs. The selected abstraction for explaining executions is a tree reflecting the principles of Tableaux 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 interaction between events, rules and databases, debuggers are needed. The context is approximatively the same as in deductive databases: rules interacts with databases ; the difference is that there can be updates of the database during rule evaluation. Chakravarthy et al. developed a tool for visualizing and explaining execution in active databases [5]. 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 interesting because efficient, and with a general initial trace but there no flexibility possibilities. The abstractions are subordinated to the visualization tool.

2.2 Different approaches of debugging

In all these systems the extracted trace is already of high level dedicated to one or two particular abstractions. A lot of information has been lost, in particular on operational semantics. *The trace is too specialized.*

The filtering part is absent. The only filtering is the transition between the trace and the visualized structure. The proposed abstraction is fixed. The filtering step is merged with the visualization step.

Users can not get another point of view of the execution. In particular in deductive databases, if they want a global image with set by set database accesses, proof trees are totally inadapted. Considering that there is one proof tree (or several) per answer, the number of proof trees that can be obtained with a query on a million tuples database can be unacceptable.

Different ways of conceiving trace and abstraction levels are representing on Fig. 1. This representation is split up in two sub-figures corresponding to different approaches of debugging.

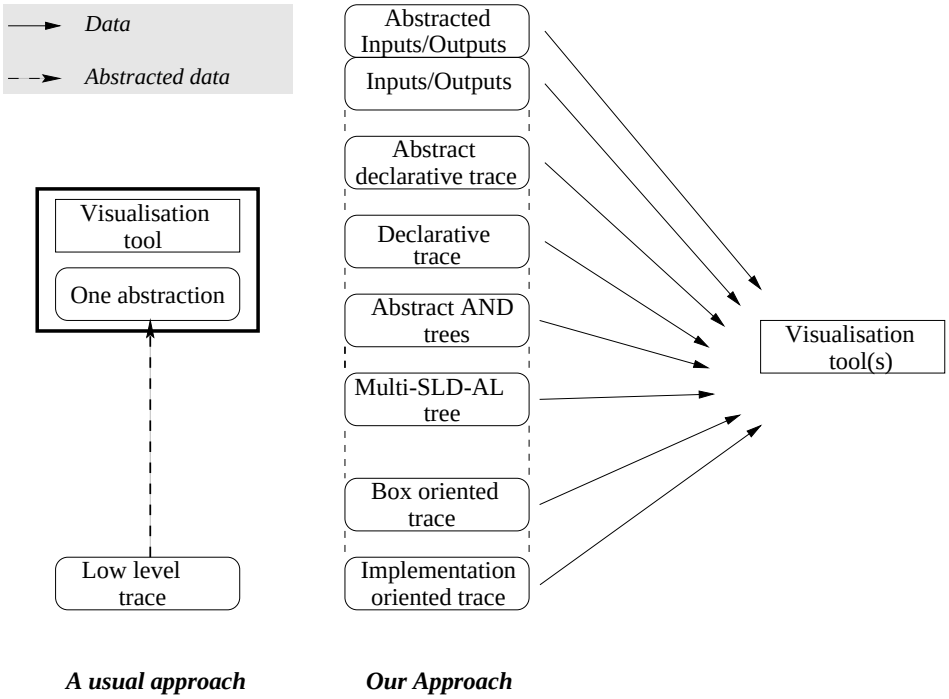

Figure 1: Debugging tools and abstractions

The first approach reflects the systems discussed in the previous section. It consists in constructing an abstraction from a low-level trace. The level of abstraction is set once for all. As we have already mentioned, filtering is mixed with visualization. The contents of the trace and of the abstraction are in terms of the visualization. There is no possibility to construct higher abstractions.

Our approach is to have a scale of possible abstractions starting from the lowest one, the closest to implementation. It allows users to choose their level of abstraction. There can be several intermediate levels of abstraction till for example the levels presented in declarative debugging by Naish [12] and in abstract debugging by Comini et al.[6]. There exist some higher levels of trace abstraction in terms for example of inputs and outputs of the program. All these abstractions can then be visualized with tools dedicated to the abstractions and it is not the abstraction that is dedicated to the visualization tool.

The way we propose to implement this approach is to combine the implementation oriented trace, in our case the relational trace, with an instrumented meta-interpreter. The result is a very flexible tool. As an illustration of this flexibility, the generation of three abstracted traces is described in section 5: the box-oriented trace, the multi-SLD-AL tree and Abstract AND trees.

3 A Multi-SLD-AL Meta-Interpreter

3.1 Principles of Multi-SLD-AL Resolution

A DDB program defines two databases : the intentional database composed of data in the database and the extensional database defined by the deductive program. Predicates defining the intentional database are intentional predicates or, more simply, database predicates. Predicates defining the extensional database are extensional predicates or 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.

```
Rules
(c1)  anc(X,Y) :- p(X,Y).
(c2)  anc(X,Y) :- anc(X,Z), p(Z,Y).

Database
p(a,b)   p(e,f)    p(f,g)
p(b,c)   p(d,b)
```

Figure 2: Definition of the *anc* program

There exist different techniques to solve the rule part of the different DDB systems, see for example the survey of Ramakrishnan and Ullman[14]. In particular, the system we are working on, the Validity system, is based on SLD-AL resolution (*SLD with test of*

Admissibility and resolution on Lemmas) described by Vieille in [20]. 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 information have to be stored: solved goals, goals in the process of resolution and produced solutions 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 this set, it is called *non-admissible goal* and it is solved using 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 induce non termination. Goals using database predicates are not stored.

The SLD-AL resolution manipulates tuples one by one. On the example 2, to solve *p(X,Y)*, one branch of resolution is created by tuple in the database unifying with *p(X,Y)*. However most of the time, database accesses are set by set accesses 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.

These substitution sets are introduced 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 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 *builtin* if it is a built-in predicate.

Figure 3 represents a multi-SLD-AL tree for the query *anc(X,Y)* on 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 *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 *anc(X,Z)* which is a variant of *anc(X,Y)*. The substitution set is enriched with these lemmas. After the evaluation of *p(X,Z)*, the result of the database access 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 can not be joined with the tuples selected during the database access. They are suppressed from the substitutions set like for example $\{X/d, Z/b\}$.

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 $anc(X,Z), p(Z,Y)$ 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 gives an empty set.

3.2 A 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.

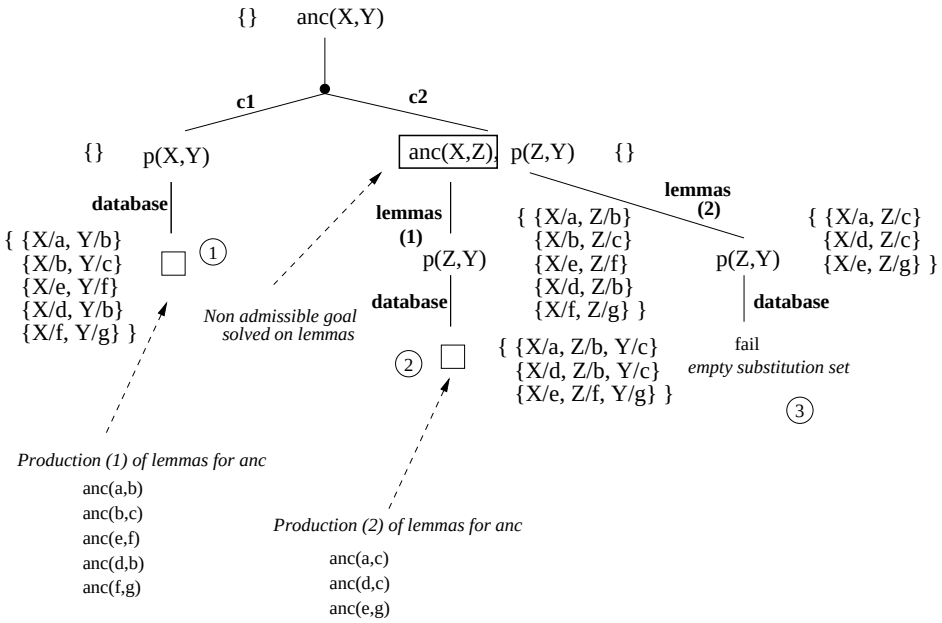

Figure 3: A multi SLD-AL tree

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, taking into account the non admissible goals but without trying to solve them. In the second pass *solve_na/2* solves the non admissible goals using lemmas.

The resolvent structure given to *solve_goal/4* reflects the local selection rule. Solving A if it unifies with the head of the rule $H \leftarrow B_1, \ldots, B_m$ adds
$subgoal([B_1, \ldots, B_m], H)$ in the resolvent. The last atoms introduced in a resolvent are the first to be selected. With this method, it is easy to produce lemmas because the subgoal for which answers have been produced is known immediately.

Fig. 5 defines the resolution of a goal. It is composed of two stages: initially, the atom to be evaluated is chosen (*select/3*) then *solve_atom/6* 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

```
solve(Query, SubstSet) :-
    save_subquery(Query, emptyset, AdmissibleSet),
    init,
    solve_goal([subgoals([Query], true)], AdmissibleSet,
               SubstSet, Query),
    save_answer( Query, SubstSet).

solve(Query, SubstSet) :-
    solve_na(Query, SubstSet),
    save_answer(Query, SubstSet).


solve_na(Query, SubstSet) :-
    not new, !, fail.

solve_na(Query, SubstSet) :-
    retract_all(new),
    na(Atom, SubstSet0, Continuation, Query),
    answer_set(Atom, SubstSet0, SubstSet1),
    solve_goal(Continuation, SubstSet1, SubstSet, Query).
```

Figure 4: Multi-SLD-AL meta-interpreter: search tree traversal

```
solve_goal([], SubstSet, SubstSet, _).

solve_goal([subgoals([], SubGoal)| Rest], SubstSet0, SubstSet,
           Query) :-
    save_answer(SubGoal, SubstSet0),
    solve_goal(Rest, SubstSet0, SubstSet, Query).

solve_goal( Goal, SubstSet0, SubstSet, Query) :-
    select(Goal, Atom, NewGoal),
    solve_atom(Type, Atom, NewGoal, SubstSet0, SubstSet,
Query) .


solve_atom(base, Atom, Resolvent, SubstSet0, SubstSet, Query) :-
    is_basis(Atom), !,
    access(Atom, SubstSet0, SubstSet1),
    solve_goal( Resolvent, SubstSet1, SubstSet, Query).

solve_atom(builtin, Atom, Resolvent, SubstSet0, SubstSet, Query) :-
    is_builtin(Atom), !,
    answer_builtin(Atom, SubstSet0, SubstSet1),
    solve_goal( Resolvent, SubstSet1, SubstSet, Query).

solve_atom(na, Atom, Resolvent, SubstSet0, SubstSet, Query) :-
    variants_subquery(Atom, SubstSet0, NonAdmSet),
    ( all_non_admissible(NonAdmSet) -> !
     ;
      true),
    save_na(Atom,  NonAdmSet, Resolvent, Query),
    fail.

solve_atom(rule, Atom, Resolvent, SubstSet0, SubstSet, Query) :-
    save_subquery(Atom, SubstSet0, AdmSet),
    choose_rule(Head, Body),
    unify(Atom, Head, AdmSet, SubstSet1),
    composed(Body, Resolvent, NewResolvent),
    solve_goal(NewResolvent, SubstSet1, SubstSet, Query).
```

Figure 5: Multi-SLD-AL meta-interpreter: resolution

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 to store 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.

4 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 propose 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.

4.1 The Validity "Relational" Trace : Generation

At compilation time, flows of data inside each rule are defined via relations and via operations on these relations (projection, join). These flows are then assembled in a static graph composed of relational operations.

The execution instantiates this graph with the tuples actually present in the database. To handle non admissibility of goals, the execution manages and traverses an internal dynamic graph.

During this execution, that is to say when the user queries the database using defined predicates, a "relational" trace is generated. A new trace is generated each time the user asks for a new query. There is no trace generated for modification of the database. Updates are taking into account only in the subsequent queries.

This trace has been added ad hoc by implementors in order to have some low level information for debugging. It describes a succession of events which reflect the execution steps. These events are of two types: management of the dynamic graph and operations on the data.

The management of the graph gives information on the non admissible goals. The data events give pointers to tables stored in the same database as the extentional one. 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 debugging 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.

4.2 Information in the Trace

Reducing the Non Determinism The trace is the 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 in retrieving this information. In the same way the type of this atom is present in the table of symbols. The clause of *solve_atom/6* which is used can now be fixed in advance using indexing. *Type* in *solve_atom(Type, Atom, NewGoal, SubstSet0, SubstSet, Query)* is instantiated before invocation. Furthermore, the rule selected for the resolution of a derived predicate is also present in the trace. In this case, the choice point, created when there exist different clauses which can unify with the selected atom, has to be maintained to preserve backtracking in the meta-interpreter. The predicate *choose_rule/2* is defined on Fig. 6. The principle is to repeat the choice of rules until no more rule unifies with the current goal.

```
choose_rule(Head, Body) :-
      <Extract selected rule from the trace> .

choose_rule(Head, Body) :-
       /* Create the choice point */
  ( choose_rule(Head, Body)
  ;
   !, fail).
```

Figure 6: Simulating a choice point

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.

Avoiding database accesses One of the principal reasons to connect the trace and the meta-interpreter is to avoid re-accessing the database. The relevant information has been stored in temporary relations accessible from within the trace.

The sets of substitutions are not transmitted along the meta-interpreter because it would cost a lot of time and memory. At each point of construction of a set (namely in *answer_set/3*, *access/3*, *answer_builtin/3*, *unify/4*), the substitutions are directly extracted from the temporary relations. The arguments *SubstSet* and *SubstSet0* are no more used in *solve_atom/4* 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. The correspondance between relations and query is contained in the trace file.

For certain literals, it is not possible to obtain the new substitution set from the trace only. Indeed, in the case, for example, of several database predicates to be evaluated one after the other, the various operations to be carried out on the database are gathered in a single one and the obtained substitution set refers to the end of this operation. However, the meta-interpreter is able to distinguish the processing of each literal. In the current implementation, in a first stage, the result of the gathered operations is given. If the user asks for the detailed evaluations, it will be possible in the second stage to give this information by querying directly of the database.

4.3 The difficulty: Synchronizing the meta-interpreter with the trace events

The meta-interpreter and the trace must be traversed in parallel. The progression is synchronized. The problem is that an event does not necessarily correspond to one literal only, but the literals to which it corresponds are well-known. The current event has to be stored as long as information it contains are useful for the meta-interpreter. If the meta-interpreter backtracks, the event must stay the same. This is not the case if the event is an argument of the resolution predicates. For this reason, it has been necessary to use a global variable for this information. The current event is changed as soon as all the related literals are treated.

To control the reading of new events, three new predicates have been defined in the meta-interpreter: *manage_trace_call*, *manage_trace_exit* and *manage_trace_na*. The first one is inserted in the third clause of *solve_goal/2* before a call to *select/3* as shown on fig. 7. It checks that the event is correct before selecting the next atom. *manage_trace_exit* is used in the first and second clauses of *solve_goal/2* to consume events that correspond to answer for lemmas production. *manage_trace_na* prepares the current event in *solve_na* before the call to *na/3*. The current literal is incremented at the end of each clause of *solve_atom/4*.

Figure 7 sums up the connection on a small part of the meta-interpreter. In *solve_goal/2*, *manage_trace_call* reads a new event if necessarily, then *select/3* extracts relevant information from the trace. In *solve_atom/4*, *choose_rule/2* extract the selected rule and a choice point is created as explained before. *unify/4* gives the set of substitions corresponding to the resolution of the atom and then increments the current literal to progress in the trace.

5 Instrumentation of the Meta-Interpreter: Abstract Traces

Once trace information has been associated with multi-SLD-AL resolution, the second step is now to filter in order to produce abstract views of executions which we call more simply explanations. As already mentioned in the introduction, it is necessary to have different abstractions to adapt to different users and different debugging or understanding problems. In this article, the filtering step will not be described in details. The aim of this part is to show that abstractions can be easily integrated in the meta-interpreter by instrumentation.

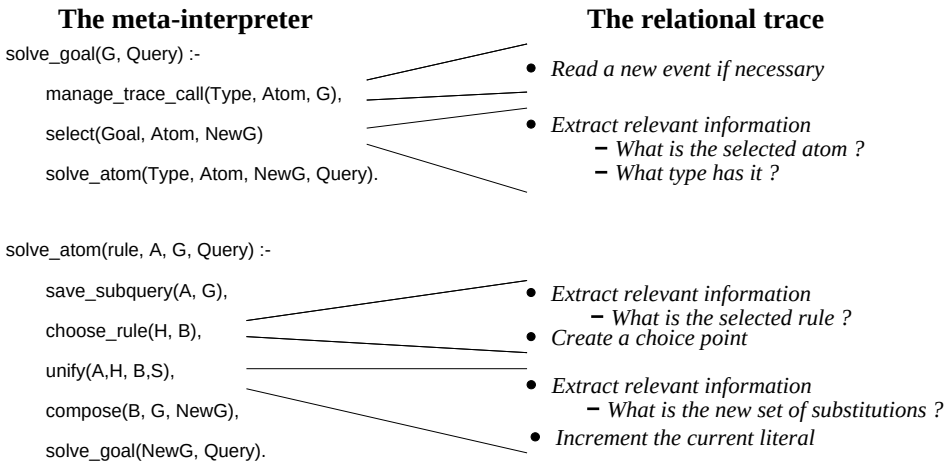

Figure 7: Using relational trace information

Three possible abstractions of the execution are presented. The order of presentation corresponds to a growing level of abstraction starting from the level close to execution. The first one, a relatively low level, is a box-oriented representation of execution, which is interesting for developers. 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.

5.1 Box-oriented trace

A box-oriented trace gives a sequence of events inspired by those proposed by Byrd in [4]. The meta-interpreter is instrumented following the tracing methods of Prolog programs described in [19, 23]. 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_na, fail_na, exit_na, redo_na*. The *_na* suffix refers to non-admissible goals.

The meta-interpreter builds a trace of the resolution of the various atoms appearing during the evaluation. Only the predicates *solve_goal/4*, *solve_atom/6* and *solve_na/2* are instrumented in order to trace each resolution of an atom. The instrumentation of *solve_na/2* is presented on Fig. 8. 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 example 2 is presented on Fig. 9. Compared to a Prolog trace, substitutions are not integrated in the atoms but given by sets on the side. At unify time the type of transititon is mentioned (rule, base, non-admissible, builtin). 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.

```
solve_na(Query, SubstSet) :-
    retract_all(new),

    (  trace(call_na, Atom)
     ; trace(fail_na, Atom),
       fail),

    na(Atom, SubstSet0, Continuation, Query),
    answer_set(Atom, SubstSet0, SubstSet1),

    (  trace(exit_na, Atom, SubstSet)
     ; trace(redo_na, Atom),
       fail),

    solve_goal(Continuation, SubstSet1, SubstSet, Query).
```

Figure 8: Instrumentation: generation of a box oriented trace

```
call      anc(X,Y)
unify     anc(X,Y)   rule
call      p(X,Y)
unify     p(X,Y)     base
exit      p(X,Y)     (X,Y) in {(a,b),(e,f),(b,c),(d,b),(f,g)}
exit      anc(X,Y)   (X,Y) in {(a,b),(e,f),(b,c),(d,b),(f,g)}
redo      anc(X,Y)
unify     anc(X,Y)   rule
call      anc(X,Z)
unify     anc(X,Z)   non-admissible
fail      anc(X,Z)
fail      anc(X,Y)
call_na   anc(X,Z)
exit_na   anc(X,Z)   (X,Z) in {(a,b),(e,f),(b,c),(d,b),(f,g)}
call      p(Z,Y)
unify     p(Z,Y)     base
exit      p(Z,Y)     (X,Z,Y) in {(a,b,c),(e,f,g),(d,b,c)}
exit      anc(X,Y)   (X,Y) in {(a,c),(e,g),(d,c)}
```

Figure 9: A part of the trace of example 2

5.2 The Multi-SLD-AL Tree

This abstraction gives an operational view of execution in terms of a multi-SLD-AL tree (see Section 3.1).

5.2.1 The tree generation

The management of the nodes occurs at the time of the resolution of the selected atoms. The predicates *solve_atom/6* and *solve_goal/6* are instrumented to build the tree. Fig. 10 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*. In order to have single node identifiers, a global variable, *current_node* is used.

```
solve_atom(base, Atom, Resolvent, SubstSet0, SubstSet, Query,  Father ):-
    is_basis(Atom), !,

    value(NewCurrentNode),
    create_node(NewCurrentNode, Atom, SubstSet0),
    create_edge(Father, NewCurrentNode),
    incr_current_node,

    access(Atom, SubstSet0, SubstSet1),
    solve_goal( Resolvent, SubstSet1, SubstSet, Query,  NewCurrentNode ).
```

Figure 10: 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, like in the abstraction we choose, only the type of the transition.

It is necessary in the case of the non admissible goals to enrich *save_na* with the identifier of the associated node. Indeed, when the evaluation of the goal is taken again, it should be known where to hang the sub-tree.

5.2.2 Tree visualization

The meta-interpreter creates a description file of the tree. We use the graph description language GML (Graph Modeling Language). The procedures creating nodes and edges generate GML data.

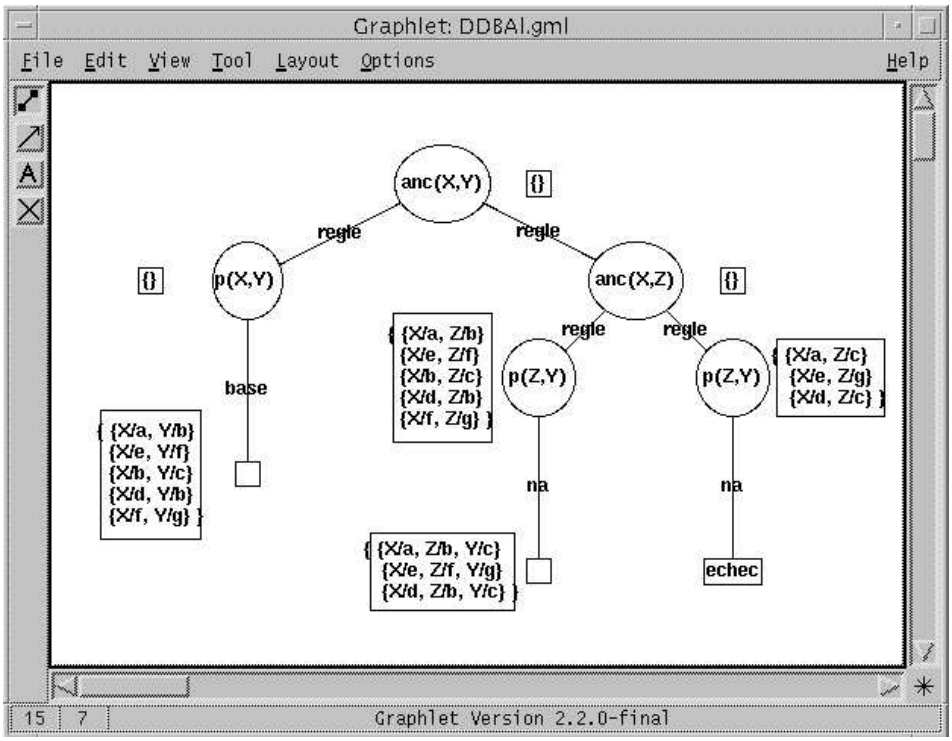

Figure 11: Visualisation of the multi-SLD-AL tree of example 2 with Graphlet

The result can then be displayed using a graph editor, we use the Graphlet[1] editor developed at the university of Passau. The obtained result for example 2 is presented on Fig. 11. The resolvent is not represented on this abstraction. Users can ask for it.

5.3 Forest of Abstract AND Trees

This abstraction consists in 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 [3]. 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.

These trees are constructed by instrumentation of the meta-interpreter. Instrumentations of the first and third clause of *solve_goal/6* and of the fourth clause of *solve_atom/8* are presented on Fig. 12. Two new arguments are added to *solve_goal/4*, *solve_na/2* and *solve_atom/6*: 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/6*. In this clause, *traceInstantiateTree/2* adds the substitutions set to the skeleton of the tree and constructs the corresponding tree. In the third clause of *solve_goal/6*, 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/8*, 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.

As for the multi-SLD-AL tree, the generated trees are in GML format in order to be visualised with Graphlet. The result of this abstraction on example 2 is presented on Fig. 13. The 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 abstract AND trees correspond to eight proof trees. Users can be interested by the detail 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 user to consult this information. In this case a tool as proposed in Explain is useful [1].

6 Conclusion

We have presented a tracing technique which consists in integrating a relational trace with an instrumented set-oriented meta-interpreter. The relational trace reduces the non determinism of meta-interpretation and avoids many calculations at debugging stage that have been already performed at execution. The meta-interpreter allows different abstract views of the execution to be constructed, for example by instrumentation.

This technique has been illustrated on the Validity system. The meta-interpreter is grounded on the multi-SLD-AL operational semantics, which is an extension of the multi-

[1] http://www.fmi.uni-passau.de/Graphlet/

```
solve_goal([], SubstSet, SubstSet, _, AbstractAndTree, _) :-

    traceInstantiateTree(AbstractAndTree, SubstSet) .

solve_goal( Goal, SubstSet0, SubstSet, Query,
                AbstractAndTree, CurrentNode) :-
    select(Goal, Atom, NewGoal),

    modifyCurrentNode(Atom, AbstractAndTree, NewCurrentNode),

    solve_atom(Type, Atom, NewGoal, SubstSet0, SubstSet,
                Query, AbstractAndTree, NewCurrentNode) .


solve_atom(rule, Atom, Resolvent, SubstSet0, SubstSet, Query,
               AbstractAndTree, CurrentNode) :-
    save_subquery(Atom, SubstSet0, AdmSet),
    choose_rule(Head, Body),

    createSons(AbstractAndTree, CurrentNode, Body, NewAbstractAndTree),

    unify(Atom, Head, AdmSet, SubstSet1),
    composed(Body, Resolvent, NewResolvent),
    solve_goal(NewResolvent, SubstSet1, SubstSet, Query,
                NewAbstractAndTree, CurrentNode).
```

Figure 12: Instrumentation: generation of Abstract AND trees

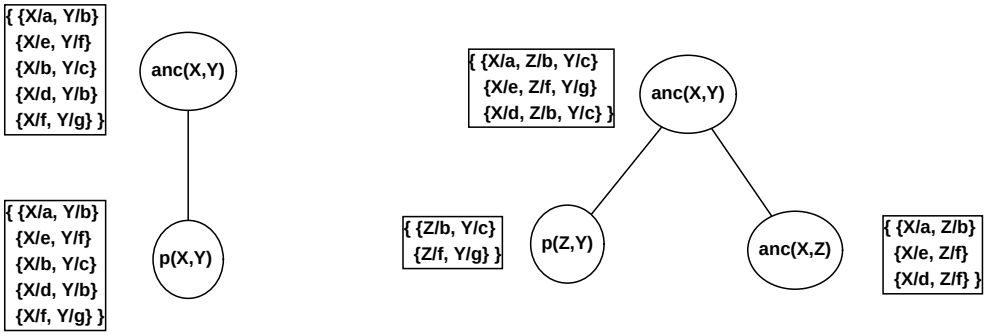

Figure 13: Visualisation of the two Abstract AND trees of example 2 with Graphlet

SLD operational 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.

The flexibility at reasonable cost offered by our technique enables the debugger to be adapted, in particular, to end users.

As further work, we will extend our meta-interpreter to negation and agregates present in the DEL language.

Acknowledgments Alexandre Lefebvre, Laurent Vieille and Bernard Wappler from Next Century Media[2] sacrificed part of their time to explain the operation of Validity. Moreover, Bernard Wappler has modified the relational trace of Validity in order to facilitate integration with the meta-interpreter. Olivier Ridoux gave fruitful comments to an earlier version of this article

References

[1] T. Arora, R. Ramakrishnan, W.G. Roth, P. Seshadri, and D. Srivastava. Explaining program execution in deductive systems. In S. Ceri, K. Tanaka, and S. Tsur, editors, *Proceedings of the Deductive and Object-Oriented Databases Conference*, volume 760 of *LNCS*. Springer-Verlag, December 1993.

[2] C. Beeri and R. Ramakrishnan. On the power of magic. *Journal of Logic Programming*, 3(10):255–299, 1991.

[3] Maurice Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10:91–124, 1991.

[4] L. Byrd. Understanding the control flow of Prolog programs. In S.-A. Tärnlund, editor, *Logic Programming Workshop*, Debrecen, Hungary, 1980.

[5] S. Chakravarthy, Z. Tamizuddin, and J. Zhou. A Visualization and Explanation tool for debugging ECA rules in active databases. In Timos K. Sellis, editor, *Rules in Database Systems, Second International Workshop*, volume 985 of *LNCS*, pages 197–209, Athens, September 1995. Springer.

[6] M. Comini, G. Levi, and G. Vitiello. Abstract debugging of logic programs. In L. Fribourg and F. Turini, editors, *Proc. Logic Program Synthesis and Transformation and Metaprogramming in Logic*, volume 883 of *LNCS*, pages 440–450, Berlin, 1994. Springer-Verlag.

[7] M. Ducassé. Automated debugging with trace analysis: the case of Opium. In C. Beckstein, editor, *Proceedings of the KI'94 Workshop on Development, Test and Maintenance of Declarative AI-Programs*, Saarbruecken, Germany, September 1994. GMD-studies.

[8] M. Kettner and N. Einsinger. The tableau browser SNARKS. In William McCune, editor, *14th International Conference on Automated Deduction CADE'97*, volume 1249 of *LNCS*, pages 408–411, Australia, July 1997. http://www.pms.informatik.uni-muenchen.de/~snarks/index.html.

[9] A. Lefebvre. *Evaluation de requêtes dans les bases de données déductives : aspects théoriques et pratiques.* PhD thesis, Université René Descartes of Paris V, France, juin 1991.

[2] http://www.nextcenturymedia.com

[10] R. Manthey and F. Bry. SATCHMO : a theorem prover implemented in prolog. In *9th Conference on Automated Deduction, CADE 88*. Springer Verlag, 1988.

[11] R. Marti, C. Wieland, and B. Wüthrich. Adding inferencing to a relational database management system. In T. Härder, editor, *Datenbanksysteme in Büro, Technik und Wissenschaft*, pages 266–270. Springer Verlag, 1989.

[12] Lee Naish. A declarative debugging scheme. *Journal of Functional and Logic Programming*, 3, 1997.

[13] R. Ramakrishnan, D. Srivastava, and P. Sheshadri. Coral : Control, relations and logic. In Li-Yan Yuan, editor, *Proceedings of the 18th International Conference on Very Large Databases*, 1992.

[14] R. Ramakrishnan and J. D. Ullman. A survey of deductive database systems. *Journal of Logic Programming*, 23:125–149, 1995.

[15] Donald A. Smith. MultiLog: Data Or-Parallel Logic Programming. *Journal of Logic Programming*, 29:195–244, 1996.

[16] G. Specht. *Source-to-source Transformationen zur Erklärung des Programmverhaltens bei deduktiven Datenbanken.* PhD thesis, Technische Universität München, Juni 1992. In German.

[17] G. Specht. Generating explanation trees even for negations in deductive database systems. In M. Ducassé, B. Le Charlier, Y.-J. Lin, and U. Yalcinalp, editors, *Proceedings of ILPS'93 Workshop on Logic Programming Environments*, Vancouver, October 1993. LPE'93.

[18] G. Specht and B. Freitag. Amos : a natural language parser implemented as a deductive database in LOLA. In R. Ramakrishnan, editor, *Applications of logic databases*, pages 197–214. Kluwer Academic Publishers, 1995.

[19] L. Sterling and E. Shapiro. *The Art of Prolog, second edition.* MIT Press, Cambridge, Massachusetts, 1994. ISBN 0-262-19338-8.

[20] L. Vieille. *Bases de données déductives : évaluation et optimisation de programmes logiques récursifs.* PhD thesis, Université Paris 6, septembre 1988.

[21] L. Vieille, P. Bayer, V. Küchenhoff, and A. Lefebvre. EKS-V1, a short overview. In *Workshop on Knowledge Base Management System*, Boston, USA, July 1990. AAAI-90.

[22] C. Wieland. Two explanation facilities for the deductive database management system DeDex. In H. Kangassalo, editor, *Proceedings of the 9th Conference on Entity-Relationship Approach*, pages 189–203, 1990. ETH Zurich.

[23] L.Ü. Yalcinalp. *Meta-programming for knowledge based systems in Prolog.* PhD thesis, Case Western Reserve University, Cleveland, Ohio 44106, August 1991. Technical Report TR 91-141.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399