Abstract Views of Prolog Executions in Opium

Mireille Ducassé

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

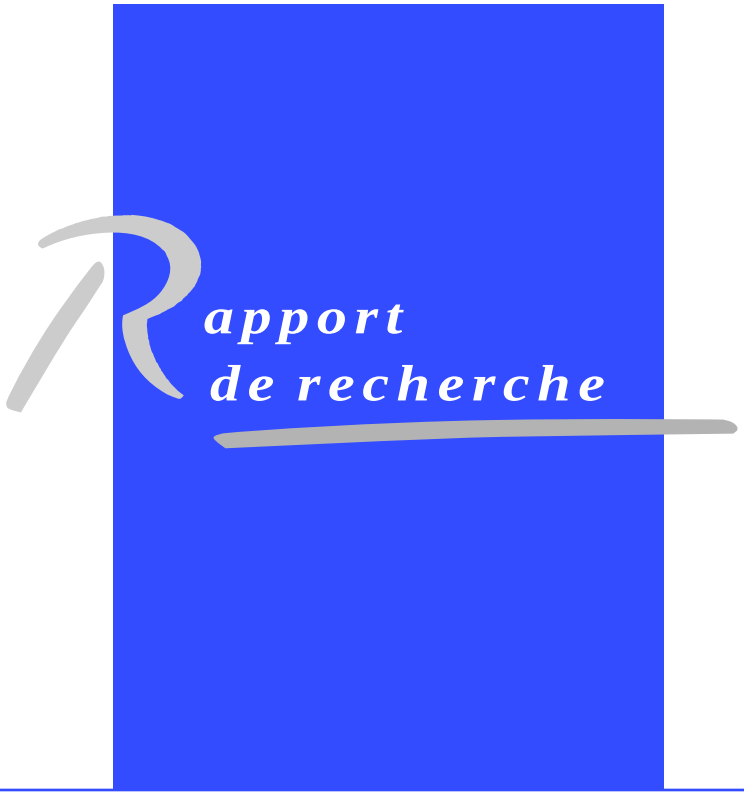
Abstract Views of Prolog Executions in Opium

Mireille Ducassé, IRISA/INSA

N ° 3531

Octobre 1998

THÈME 2



ISSN 0249-6399

Abstract Views of Prolog Executions in Opium

Mireille Ducassé, IRISA/INSA *

Thème 2 — Génie logiciel
et calcul symbolique
Projet LANDE

Rapport de recherche n3531 — Octobre 1998 — 23 pages

Abstract:

Opium is a system for analysing and debugging Prolog programs. Its kernel comprises an execution tracer and a programming language with a set of primitives for trace and source analysis.

In this report we show the power of Opium for supporting abstract views of Prolog executions. Abstract views give high-level points of view about executions. They *filter out* irrelevant details; they *restructure* the remaining information; and they *compact* it so that the information given at each step has a reasonable size. The examples of abstract views given in the following are a goal execution profile, some data abstractions, an instantiation profile, a failure analysis, a loop analysis, and a kind of explanation for an expert system written in Prolog.

Key-words: Software engineering, Programming environment, Automated debugging, Trace query mechanism, Debugging language, Program behavior understanding, Debugging tool, Prolog.

(Résumé : tsvp)

To appear in P. Brna, B. du Boulay and H. Pain, editors, *Learning to Build and Comprehend Complex Information Structures: Prolog as a Case Study*, Ablex, Cognitive Science and Technology, 1999. This article revises and extends the version published in, V. Saraswat and K. Ueda editors, Proceedings of the International Logic Programming Symposium, San Diego, October 1991, MIT Press.
The reported work was mostly achieved while the author was at the European Computer-Industry Research Centre, Arabellastr. 17, D-81925 Munich, Germany

* Correspondance address: INSA- Dept Informatique, 20 av des Buttes de Coësmes, F-35043 Rennes Cedex ; email : Mireille.Ducasse@irisa.fr, w3: http://www.irisa.fr/lande/ducasse/

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

Vues abstraites d'exécutions Prolog avec Opium

Résumé :

Opium est un système qui permet d'analyser et de déboguer des programmes Prolog. Son noyau comprend un traceur d'exécution et un langage de programmation muni d'un ensemble de primitives pour l'analyse des traces et du code source.

Dans ce rapport nous montrons la puissance d'Opium pour mettre en oeuvre des vues abstraites d'exécutions Prolog. Les vues abstraites donnent un point de vue de haut-niveau sur ces exécutions. Elles éliminent les détails non pertinents ; elles restructurent l'information restante et elles la compactent afin que l'information présentée à chaque étape ait une taille raisonnable. Les exemples de vues abstraites présentés dans la suite sont : un profil d'exécution de but Prolog, une abstraction de données, un profil d'instantiation, une analyse d'échec, une analyse de boucle et des explications pour un système expert écrit en Prolog.

Mots-clé : Génie logiciel, environnement de programmation, débogage automatisé, profil d'exécution, language de débogage, compréhension du comportement des programmes, outil de débogage, Prolog.

1 Introduction

Program analysis, such as type checking and abstract interpretation, has many uses in program development and maintenance. The data used by program analysis is often only the source code of the analysed programs. However, there is a complementary source of information, namely *traces of program executions*. An execution trace contains less general information than the program source, but it tells exactly how the program behaves in particular cases. Furthermore, there is intrinsically dynamic information in a program which is best analysed at execution time, for example uses of assert/retract, and read/write.

With "traditional" tracers users have to filter and sort a huge amount of execution trace with rudimentary support. Such tracers usually present histories of execution events where each event represents a low-level step in the execution process. An execution history contains, indeed, the information necessary to understand how a program is executed. The functionalities of current tracers, however, mostly implement variants of single step tracing. Much more can be derived from an execution trace than simple single stepping.

Single stepping in a trace execution is like watching a tennis match and following only the ball. This is the casual spectator's point of view. A player may be interested in other aspects. He can fix his attention on one player; he can count how often the first service is successful; he can analyse the backhand; etc.

The equivalent can be done for executions. One can fix one's attention on a goal; one can follow a particular variable; one can analyse backtracking; etc. This is the aim of *abstract views*: to give high-level points of view about the execution. They *filter out* irrelevant details; they may *restructure* the remaining information; they may *compact* it so that the amount of information has a reasonable size; and they *eventually display* the resulting information.

The need for different levels of abstraction in the trace is especially obvious in Prolog where the logic and the control of the programs are clearly distinguished. It is sometimes necessary to investigate some procedural details about the execution but this should not condemn Prolog users to always facing all the procedural details. On the other hand, only investigating executions at the declarative level may hamper the debugging of procedural parts of programs, such as failure driven loops.

Opium, our debugging environment for Prolog, provides general mechanisms to enable Prolog programs to be analysed by other Prolog programs. Tracers usually provide "volatile" information. Once a part of the program has been executed its trace is no longer available. To provide abstractions, one has to be able to display the trace information in a different order than it is produced. Opium offers *persistent* trace information, enabling it to be restructured in a way which fits the user's needs [1]. Tracers usually print the information they retrieve. By contrast, Opium handles the debugging information as *programming data*. The debugging information (trace and source) can then be analysed by programs before it is displayed. Some tracers provide "macros", Opium provides a general *programming language*

[1] Note, however, that if an "on the fly" analysis is sufficient, the mechanisms of Opium are as efficient as traditional tracers.

to implement debugging commands and programs. Opium's language is Prolog extended by a set of debugging primitives.

High-level program analyses have been written in Opium. Besides declarative and procedural tracing, there are many more levels of abstraction at which an execution can be examined. In the following we present a number of examples programmed in Opium. A goal profile summarizes the control flow information related to the execution of a given goal. Some data abstraction queries are shown which enable users to deal with complex and high volume data. An instantiation profile abstracts, with respect to some variables, the data flow information related to the proof of a goal. A failure analysis and a loop analysis illustrate how trace information can be selected to explain how a particular bug symptom has been produced. Last, an abstract tracer for a small expert system shows how to trace Prolog applications at an appropriate level of abstraction.

It is worth mentioning that these abstract views include all the abstract views detected by the cognitive study of [Bergantz & Hassell, 1991] although the initial version of the current chapter was written before the cognitive study was published.

Graphical debuggers and visualization tools provide global pictures of executions. These pictures are at a higher level than the single steps of the simple tracers. For example the notion of goal is better supported. However, they often contain much more information than needed, and more than the users can grasp at a glance. Hence, graphical tools also require abstraction mechanisms.

In the following we use a "trace line" oriented formalism because it facilitates *computerized* filtering of the debugging information. Our work concentrates on which debugging information to select and how to select it *automatically*. How to ultimately display the selected information is beyond the scope of this chapter. Interested readers can find some references on the latter in [Ducassé & Noyé, 1993].

The remaining of the chapter is organized as follows. The programmability mechanisms of Opium are briefly described in section 2. Some abstraction criteria are then illustrated in turn sections 3 to 7. Each section first describes which abstraction can be useful, gives an example, and give some details of the implementation in Opium. Although only small portions of programs are given, all the cited programs exist and the displayed abstracts are, indeed, actual results of Opium programs. A comparison with related work can be found in section 8. Buggy programs used through the examples can be found in the appendices.

2 Opium's programmability

This section briefly describes the basic mechanisms of Opium which enable trace abstraction programs to be written. An exhaustive description of Opium can be found in [Ducassé, 1999].

Trace format It is based on the box model of [Byrd, 1980]. A trace history, composed of a sequence of trace events, is accessible. A trace event has six slots: Chrono (chronological event number, something like a time stamp), Call (goal invocation number), Depth (depth of the goal in the execution tree), Port (control flow indication, can be "call", "unify", "exit",

"fail", "redo"), Pred (predicate corresponding to the traced goal), Arg (value of the arguments of the traced goal at the moment of the event).

Trace query language Opium's language is Prolog extended by a set of debugging primitives. A pointer in the trace history corresponds to the "current event". This pointer can be moved forwards and backwards by some dedicated primitives (`f_get`, `b_get`). The contents of the current event can be checked or retrieved (`curr_line`, `curr_arg`). A connection between trace and source information is provided (`curr_clause`). These primitives are used throughout the examples in the rest of the chapter. A more detailed description of them follows.

`f_get(Chrono, Call, Depth, Port, Pred)` moves the trace pointer forwards through the execution, or the trace database, until either the first event which matches the specified attribute values or the end of the trace history is encountered. In the first case it succeeds, while in the second case it fails.

If the argument corresponding to an attribute is:

- a variable, this attribute is not taken into account in the pattern matching.
- a value, the attribute value of the current event is checked against the required value. Users can specify an exact value, a list of possible values, a negated value , or, for integer attributes, an interval.

`b_get/5` does the same as `f_get/5` but backwards.

`f_get` and `b_get/5` have a family of related primitives. Among them `f_get_np/5` does not print the event it moves to; `det_f_get/5` can succeed only once.

`curr_line(Chrono, Call, Depth, Port, Pred)` retrieves the value of all the control flow attributes of the current event. Whether these slots have been displayed or not does not make any difference. If the argument corresponding to an attribute is:

- `-`, nothing is done for this attribute.
- a variable, the current value of the attribute is retrieved and unified with the variable.
- a value, the current value of the attribute is retrieved and checked against the value. If unification fails, `curr_line` fails. Note that this has *no* side-effect on the traced goals.

Each slot can be retrieved individually with a `curr_<slot-name>` primitive.

`curr_arg(ArgList)` retrieves the value of the arguments of the current goal in the current event.

`curr_clause(Clause)` retrieves the clause which has spawned the current event's goal.

3 Control flow abstraction

Abstracting the control flow information is very desirable. All Prolog tracers provide users with the facility to hide information related to "procedural" details of the execution and keep only its "declarative" aspects. Some tracers, including Opium, provide a breadth-first trace (called top-down zooming). Users may also want to collect information to illustrate a particular aspect of the control flow, for example a goal's behaviour or recursion behaviour.

Displaying all the trace events related to a goal execution in one step gives a more global idea of the goal than just showing the events when they appear in the execution history. Indeed, they are not necessarily contiguous in the trace. The number of trace events related to a given goal can be very large (for example, for a generate and test program, the goal which generates the solution can succeed many times).

We can first compute a *profile* of the goal execution which gives the number of trace events related to the given goal. The user will then ask to see what he estimates to be reasonable.

For example, let us assume that the user wants to investigate the execution of goal `permutation([1, 2, 3, 4], X)` from the N-queen program given in appendix A. The profile of its execution

```
call permutation([1,2,3,4], X)
     succeeds 24 times
     ultimately fails
```

indicates that it succeeds 24 times (there are 24 "exit" events) and that it ultimately fails (its solutions have been exhausted). This profile is sufficient to get a first idea of the global behaviour of `permutation([1,2,3,4], X)`. The information that 24 proofs were performed tells that the correct number of permutations has been performed before the execution fails. The program was supposed to succeed. The user can then decide either to check the solutions of `permutation([1,2,3,4], X)` (i.e. the 24 "exit" events) or to first investigate the failing goals of the program.

The portion of the implementation of `goal_profile` which uses Opium primitives follows. The rest of the program is standard Prolog code.

```
goal_profile(GoalNo) :-
   goto_call_line(GoalNo, CallTime),
   find_other_events(GoalNo, 0, Exits, 0, Redos, Fail)),
   write_profile(CallTime, Exits, Redos, Fail).

find_other_events(GoalNo,Exits0,Exits,Redos0,Redos,Fail) :-
   det_f_get_np(_,GoalNo,_,[exit,redo,fail],_),
   curr_port(Port),
   count_port(Port,GoalNo,Exits0,Exits,Redos0,Redos,Fail).
find_other_events(_GoalNo,Exits0,Exits0,Redos0,Redos0,0).
```

Predicate `goto_call_line` moves the trace pointer to the invocation event of the reference goal. `find_other_events` moves to the next trace event related to the goal (in a deterministic way), and retrieves the value of the current port. According to this value `count_port` increments the proper counter (Exit, Redo or Fail) and either stops if Port is "fail" or recursively calls `find_other_events`. Predicate `write_profile` eventually moves to the CallTime event, prints the invocation of the goal, and prints the rest of the profile as previously displayed.

Note that all the user-defined predicates of this example are shown. Hence those which are not explicitly named in the explanations are either Opium primitives or Prolog built-ins.

4 Data abstraction

If the data structures of the traced program are complicated, if predicates have many arguments or if arguments contain very long lists, it is likely that users will not want to examine all the arguments and the whole of them at each event of the trace. It is also possible that a graphical display of the data would be more informative than a Prolog term. As many debuggers do, Opium allows users to customize the standard display of arguments so that the data are abstracted. What is new about Opium is that whenever the users need more details about the data or a different representation they can retrieve and print any argument value, with any format, *on the fly*.

For example, let us assume that lists are currently displayed in truncated form. Assume also that the user does not want to change this setting because in the rest of the tracing session lists are indeed too large to be fully displayed. Then, while tracing the same N-queens program as previously, he can retrieve the value of the argument of safe/1 with Opium primitive `curr_arg`.

```
[opium]: print_line.
       428[2] fail safe([3, 1, ...]) % list currently truncated
[opium]: curr_arg([[_,_,X |_]]).     % recovering some hidden
    X= 4                             %                 values
[opium]: curr_arg([X]).              % recovering all values
    X= [3, 1, 4, 2]
```

Note that unification and partially instantiated structures enable users to retrieve exactly what they need. No extra mechanisms are needed at user level.

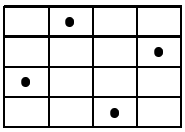
Let us assume that the user has written a Prolog program (`show_queens/1`) to graphically display a list of positions for a 4x4 board. He can then program the following command to integrate `show_queens` into Opium. Predicate `permutation/2` is also user-defined.

```
display_queens :-
   curr_pred(safe/1),        % only valid for safe/1
   curr_arg([X]),            % value of safe/1's argument
   permutation([1,2,3,4],X), % check it is a 4x4 board
```

```
   show_queens(X).           % display graphically
display_queens.
```

He can then use the `display_queens` command on the fly.

```
[opium]: print_line, display_queens.
       428[2] fail safe([3, 1, ...])
```



5 Data flow abstraction

The previous example abstracted terms within a given event. Abstracting the data flow of a program can be as interesting as abstracting its control flow. For example, to track a wrong solution, it could be useful to know where computed values are coming from.

An instantiation profile, similar to the goal profile of section 3, can tell whether a value appearing at an "exit" event was already instantiated at invocation time (at least partially); whether it comes from unification (at least partially); or whether it comes from subgoal proofs (at least partially). For example, taking the "exit" event of `select(3, [2, 3], [2])`, a goal of the previously mentioned N-Queens program, the instantiation profile of the third argument looks as follows.

```
 exit select(3, [2, 3], [2])
[opium]:  instantiation_profile(3).

      Value [2] comes from:
              unification time ([2|Zs])
              subgoal proofs

      Unified clause:
         select(X, [Y|Ys], [Y|Zs]) :-
              select(X, Ys, Zs).
```

The clause used at unify time is also displayed. This profile could be the starting point of a wrong solution analysis in the same vein as the failure analysis discussed in section 6.1. For example, if Value was already there at invocation time, the analysis should investigate the father call; if Value comes from the subgoals proofs, the analysis should determine which subgoals uses the corresponding variable(s); etc.

A portion of the implementation of `instantiation_profile` follows.

```
instantiation_profile(NthArg) :-
   curr_port(exit),
```

```
nth_arg(NthArg, ExitVal),
not var(ExitVal),
!,
curr_clause(UnifClau),
curr_chrono(ExitTime),
curr_call(GN),
det_b_get_np(_, GN, _, unify, _),
nth_arg(NthArg, UnifVal),
det_b_get_np(_, GN, _, call, _),
nth_arg(NthArg, CallVal),
unif_abstract(CallVal, UnifVal, ExitVal, Abstr),
write_unif_abstract(Abstr,ExitVal,UnifClau,CallVal,UnifVal),
goto_np(ExitTime).
```

It first checks that the current event is an "exit" event. Predicate `nth_arg/2` retrieves the nth argument of the goal in the current event. Then the value whose instantiation is tracked is checked for not being a variable. The clause which has been used to "prove" the goal is retrieved. The chronological number of the current event is retrieved so that the analysis can eventually return to it. The invocation number is retrieved so that the corresponding "unify" and "call" events can be retrieved. At each event the value of the tracked argument is retrieved. Then `unif_abstract` determines which items are responsible for the `ExitVal` and `write_unif_abstract` writes the abstract as displayed above.

6 Symptom-driven abstraction

By "bug symptom" we denote a clue that the program is misbehaving. Bug diagnosis starts when a bug symptom has been detected. The diagnosis task, from our point of view, consists of *understanding* how the symptom has been produced, locating the erroneous place(s), and fixing them. A symptom-driven abstract should explain how the bug symptom has been produced. In Opium there are currently two symptom analyses, a tracking of failures and an analysis of apparently non-terminating executions.

6.1 A failure analysis

If an execution fails the first things to examine are failing goals (called "failures" in the following). However, there can be very many failures. They cannot be simply displayed one after the other. The set of failures has to be filtered and structured.

The failures that we are especially interested in are *leaf failures*. A leaf failure is a goal which fails without any attempt to solve subgoals. It is either because unification fails or because it is a system predicate whose execution fails. A leaf failure is usually a good starting point of a backtracking process. Indeed, leaf failures are the basis of the negative explanations for expert systems of [Saurel, 1987]. However, there might be many leaf failures, and investigating them still requires some structuring and filtering.

A natural way to structure and filter is to display the failures in a top-down zooming way as in the "retrospective zooming" system of [Eisenstadt, 1985]. Considering the whole AND/OR execution tree, and starting from the top level goal(s) the failing subgoals of the next level are displayed. This implicitly hides away some failing goals: those which have an ancestor goal that eventually succeeds. These failures are said to be irrelevant in the "rational debugging" system of [Pereira, 1986]. They might actually be relevant to find the bug, but in a first approximation they have only local repercussions on the overall execution. At each stage of the debugging session, we will distinguish between *visible* failures, and *invisible* failures (hidden inside subgoal proof trees). In a first stage, we are interested only in visible failures, i.e. failing subgoals of failing goals.

Showing only the failures which are visible from a given goal already reduces the number of failures that the user has to examine at each stage. However, not all the visible failures have the same interest. There are goals failing straight away and goals failing on backtracking only. The first ones initiate the backtracking process while the latter merely sustain it. Hence, one more level of abstraction can be added. We distinguish between *direct* and *indirect* failures. A goal directly fails if it fails without being ever proved. A goal indirectly fails if it fails and it has been proved at least once.

We have designed and implemented in Opium a command called "leaf failure tracking (*lft*)". It works on a directly failing goal and retrieves all the visible directly failing subgoals of the reference goal. If there is one only (this case is frequent) *lft* is recursively applied to this subgoal. If there are several failing subgoals *lft* displays (a profile of) them, and the user decides how to go on. If there are none the tracking process is finished and the leaf failure is displayed.

The result of the leaf failure tracking on a simple bugged program which tries to simplify summations follows. This program is listed in appendix B.

```
  1 1[1] call simplify(1 + a + b + a, S)
[abstracts]: leaf_failure_tracking(1).
   22 10[5] call treat(a, 1, [], I1, V1)
      24 11[6] call integer(a)
      25 11[6] fail integer(a)
   26 10[5] redo treat(a, 1, [], I1, V1)
      28 12[6] call incr(a, [], V1)
      29 12[6] fail incr(a, [], V1)
   30 10[5] fail treat(a, 1, [], I1, V1)

[abstracts]: listing(incr).
        incr(A,[A/N|L],[A/N1|L]) :-
                !,
                N1 is N+1.
        incr(A,[S|L],[S|L1]) :-
                incr(A,L,L1).
```

From level 1 to level 4 there is only one visible directly failing subgoal at each level. At level 5, there are 2 such subgoals. The user must decide which failing branch to investigate further.

It seems reasonable that `integer(a)` fails. Regarding the failure of `incr(a,[],V1)`, the listing of `incr/2` is then enough to understand that either a base clause is missing to unify `incr(_,[],_)` or that `[]` should have never appeared as second argument of `incr`. Not all the cases are as straightforward as this one. Investigating the leaf failures does not necessarily point to the bug but it is usually a significant step towards understanding why the execution is failing.

The current implementation of the leaf failure tracking command is 160 lines long. A more detailled description can be found in [Ducassé, 1992].

6.2 A loop analysis

We give here some details of the analysis of apparently non-terminating computations.

Only parts of the trace and source which are important to understand the -possibly endless- loop will be presented to the user. Our non-termination analysis locates one looping pattern in the computation to tell the user *where* the looping process occurs. It generates an abstracted version of the trace and source which helps the user to understand *how* the looping process starts and continues. Last, it tries to find a bug which explains *why* the program does not terminate. A detailed description of the algorithm used to detect the looping pattern, of the different types of loops and of the bug heuristics can be found in [Emde & Ducassé, 1990].

The result of a loop analysis follows. The analysed program computes ancestors, it is listed in appendix C. First, a computation abstract is given:

```
1 1[1] call prince(david)
  3 2[2] call ancestor(Y, david)
  7 2[2] exit ancestor(john, david)
  8 4[2] call king(john)
  9 4[2] fail king(john)
  12 2[2] next ancestor(Y, david)
          ...
              235 57[16] call ancestor(Y, david)
              239 57[16] exit ancestor(john, david)
            242 53[15] exit ancestor(peter, david)
            243 60[15] call father(Y, peter)
            244 60[15] fail father(Y, peter)
              250 57[16] next ancestor(Y, david)
              ...
                 252 61[17] call ancestor(Y, david)
                 256 61[17] exit ancestor(john, david)
              259 57[16] exit ancestor(peter, david)
```

```
260 64[16] call father(Y, peter)
261 64[16] fail father(Y, peter)
  267 61[17] next ancestor(Y, david)
```

This computation abstract tells that the only ancestors of `david` which are computed are `john` and `peter`. The computation first fails to prove that `john` is a king. The `ancestor` predicate can then backtrack and compute in a recursive call that `john` is an ancestor of `david` which proves that `peter` is an ancestor of `david`. However the computation failed to find a father for `peter`. The `ancestor` predicate backtracks again and again find the same solution which leads to the same failure, and again the same backtracking behaviour. The related clauses are then listed in the source abstract which is generated simultaneously by the analysis, as follows.

```
      LOOP STARTED BY:
prince(X) :-
        ancestor(Y, X),
        king(Y).

      LOOPING PREDICATE(S):
ancestor(X, Y) :-
        father(X, Y).
ancestor(X, Z) :-
        ancestor(Y, Z),
        father(X, Y).

      FAILING PREDICATE(S):
king(george).

father(john, david).
father(peter, john).
father(george, pater).
```

There is an obvious bug in the last clause of `father`: `pater` should be `peter`. What the computation abstract shows, however is that every time `father` will fail the program will endlessly loop. The recursive clause of `ancestor` must be revised, maybe it should not be left recursive. Tracking the bug by simply stepping through the execution would certainly lead to discover either of the two problems but it would be hard for a programmer to keep both in mind. A typical case is to fix the error in `father` and to forget to analyse the recursive "looping" clause. Then the current test case terminates, but the next one will most certainly endlessly loop again; the programmer thought he had fixed the problem whereas he had only fixed part of it.

The actual implementation in Opium of the non-termination analysis takes more than 3000 lines.

7 Application-driven abstraction

If an interpreter or expert system has been implemented in Prolog, its users will certainly request some tracing facilities, but at the level of abstraction of their programs. They do not want to see details of Prolog executions. Application programmers can provide their users with Opium programs, tracing at the proper level of abstraction without modifying their application source code. Several examples of abstract tracing with Opium can be found in [Ducassé, 1999].

We give here an example of what can be done on a simple expert system taken from an article of [Brayshaw & Eisenstadt, 1988]. The source code follows.

```
:- op(900, xfx, ':').    :- op(870, fx, if).
:- op(880, xfx, then).   :- op(550, xfy, or).
:- op(540, xfy, and).    :- op(100, xfx, [gives,eats,has,isa]).

solve(Goal):-   fact:Goal.
solve(Goal):-   Rule:if Cond then Goal,solve(Cond).
solve(Goal1 and Goal2):- solve(Goal1), solve(Goal2).
solve(Goal1 or Goal2):- (solve(Goal1); solve(Goal2)).

fact: sheba gives milk.
fact: sheba eats meat.

m_rule:  if (A has hair or A gives milk)
         then A isa mammal.
c_rule: if (A isa mammal and A eats meat)
        then A isa carnivore.
```

Tracing the execution of goal `solve(X isa carnivore)` with the Prolog execution model gives the following lines (there are actually 66 lines altogether).

```
1[1] call sepia: solve(X isa carnivore)
 2[2] call fact : X isa carnivore
 2[2] fail fact : X isa carnivore
1[1] redo solve(X isa carnivore)
 3[2] call Rule : if(Cond) then X isa carnivore
 3[2] redo Rule : if(Cond) then X isa carnivore
 3[2] exit c_rule : if(X isa mammal and X eats meat) then X
isa carnivore
 4[2] call solve(X isa mammal and X eats meat)
  5[3] call fact : X isa mammal and X eats meat
  5[3] fail fact : X isa mammal and X eats meat
 4[2] redo solve(X isa mammal and X eats meat)
```

As stated by Brayshaw and Eisenstadt this is not the level at which a user of the expert system wants to see a trace of the execution. There is too much redundant information and the details of backtracking encountered while choosing a rule are not interesting. Tracing the whole execution at the proper level of abstraction gives the following trace.

```
call solve(X isa carnivore)
  TRYING c_rule
  call solve(X isa mammal and X eats meat)
    call solve(X isa mammal)
      TRYING m_rule
      call solve(X has hair or X gives milk)
        call solve(X has hair)
        fail solve(X has hair)
        call solve(X gives milk)
          FACT: sheba gives milk
        exit solve(sheba gives milk)
      exit solve(sheba has hair or sheba gives milk)
    exit solve(sheba isa mammal)
    call solve(sheba eats meat)
      FACT: sheba eats meat
    exit solve(sheba eats meat)
  exit solve(sheba isa mammal and sheba eats meat)
exit solve(sheba isa carnivore)
```

To obtain the previous abstracted trace, the programmer of the expert system can provide a dedicated `expert_next` command which traces step by step at the proper level. A portion of its implementation follows.

```
expert_next :-
   f_get_np(_,_,_, [call, exit, fail], [solve/1,(:)/2]),
   curr_pred(Pred),
   my_print_line(Pred).

my_print_line(solve/1) :-
   print_line.
my_print_line((:)/2) :-
   curr_port(exit),
   curr_arg(ArgList),
   expert_arg(ArgList).

expert_arg(ArgList) :-
   write_standard_indent,
   expert_arg_do(ArgList).
```

```
expert_arg_do([fact, Fact]) :-
   !, printf('FACT: %w\n', [Fact]).
expert_arg_do([RuleName|_]) :-
   printf('TRYING %w\n', [RuleName]).
```

Display parameters are set so that by default only the "port", "pred" and "arguments" slots of a event are displayed. Indentation is set "on". Predicate `expert_next` retrieves only the `solve/1` and `:/2` predicates, and for these predicates only the "call", "exit" and "fail" events. Thus it hides away the backtracking information. Then it retrieves the predicate slot of the current event and predicate `my_print_line` prints the event according to this predicate value. Events related to `solve` are displayed normally. Events related to `:/2` are displayed only if they are "exit" events (ie once a rule or a fact has been chosen). If `my_print_line` fails `f_get_np` will backtrack and retrieve a new event. Predicate `expert_arg` prints the indentation which would have been printed by the Opium primitive `print_line`. Predicate `expert_arg_do` writes whether a fact has been found or a rule is being tried and prints the actual value of the fact or rule.

Note that `expert_next` can backtrack, hence the following goal will generate an exhaustive trace.

```
[opium]: expert_next, fail.
```

8 Related work

The following concentrates on related works which provide some sorts of abstract views of sequential Prolog executions. Abtract views of parallel logic programming languages are mostly dedicated to parallelism features, see for example [Carro *et al.*, 1993, Kusalik & Prestwich, 1993, Vaupel *et al.*, 1997, Schulte, 1997]. Readers interested in a comparison with systems for other paradigms can consult the related work section of [Ducassé, 1999].

Many tracers also provide breadth-first tracing (often called top-down zooming) [Shapiro, 1983, Eisenstadt, 1985, Moroshita & Numao, 1986, Lloyd, 1987, Yalcinalp, 1991]. To achieve this most of them run a dedicated Prolog interpreter, thus they can hardly provide users with different points of view. Goal views are provided by graphical tracers (such as the Transparent Prolog Machine of [Eisenstadt & Brayshaw, 1988]) and visualization tools (such as Pictorial Janus of [Kahn & Saraswat, 1990]). [Neufeld *et al.*, 1997] proposes a colouring of the search tree which illuminates the unification process. Graphical tools usually display the whole execution which can easily be overwhelming. Furthermore, [Mulholland, 1997]shows that a graphical tool can be disturbing for novice programmers who spend more time enquirying about the tool than about the execution they are analysing.

Among the related systems two graphical tracers offer many abstraction mechanisms, the Transparent Prolog Machine (TPM) and the Dewlap system [Dewar & Cleary, 1986]. In TPM there are two levels of granularity for the trace of the execution tree, and there is a "hard-coded" way of specifying application-driven abstraction. In Dewlap the size of

the nodes is proportional to the level of execution. Both have static data and data-flow abstraction mechanisms. All these mechanisms are powerful and useful; they are, however, all ad hoc. There is no generic mechanism which allows the users to *design* new abstraction mechanisms and implement them easily.

Usually tracers provide some data abstraction, the user can specify how the standard display of trace events should look like. However, they usually suffer from two drawbacks. First, what is hidden cannot be retrieved on the fly. Users first have to reset the default display then reprint the whole event. This can be very tedious, especially if the abstract display has to be reset afterwards. Second, there is no easy way to use different display procedures on the fly, hence to have several levels of abstraction for the same data require ad hoc implementations [Lichtenstein & Shapiro, 1988, Cochard, 1990]. In Opium any hidden information (about terms but also about slots of the event) can be retrieved on the fly without modifying the default display parameters. Furthermore, any Prolog predicate can be used on the fly, hence allowing several levels of abstraction or different points of view, without any extra mechanism. The display programs still have to be written but it is straightforward to integrate them inside Opium.

Many systems provide ad hoc symptom abstractions for Prolog. The algorithmic debugging family, following [Shapiro, 1983], mainly concentrates on an algorithmic traversal of trees related to executions, the algorithm depending on the symptom. Most systems use mainly the proof tree, but Naish in a recent article came up with a framework which can model executions with other types of trees depending on the symptom [Naish, 1997]. The Rational Debugging system [Pereira, 1986, Pereira & Calejo, 1988] uses two heuristics to select events which are very close to our instantiation profile and our leaf failure tracking. The expert systems explanation tools of [Saurel, 1987] and [Yalcinalp, 1991] are also very close to our leaf failure tracking. The main problem of these tools is that they are inflexible. If the user cannot answer a query the debugging session is aborted; if he realizes that he has answered wrongly, he cannot backtrack; if he has hypotheses about the bug, he cannot use them. Furthermore, none of the above systems provide abstraction mechanisms when the execution does not terminate.

Our loop analysis gives some insight in the behaviour of what Brna et al. call "endless building" in their specification of a framework to analyse non-terminating executions [Brna *et al.*, 1993]. They consider other types of non-terminations but as far as we know they have no implementation. The system of [Pelhat, 1987] consists of a static phase where the source code is checked for possible endless loops which then are especially watched at run-time. If it is suspected that a cycle has been entered, the control is given to the user who can get further information by a symbolic evaluation of the cycle. Our approach combines static and dynamic analyses the other way round, we statically analyse only those predicates involved in the looping pattern. This enables us to run static analyses with a much finer granularity.

The system of [Casson, 1990] enables application-driven abstraction. However, the tracer is a subroutine of the meta-interpreter. The abstracted trace must therefore be displayed in exactly the same order as it is produced. The trace information cannot be restructured.

9 Conclusion

In this chapter we have shown how the Opium system can be used to provide abstract views of Prolog program executions. The power and flexibility of Opium has been illustrated using six widely differing examples.

The list, surely, is not exhaustive. It can be noted that there are many possible abstraction criteria. Therefore, building ad hoc tools for each abstraction criterion would be much too costly. Moreover, some of the abstraction criteria depend on the application or on the user's taste, hence they *can not* be provided by a debugger implementor. This and the examples shown previously should convince the reader that a user-friendly debugger must be *programmable* and *extensible* so that users have at their disposal the accurate abstraction criteria at the proper moment.

The current mechanisms of Opium are one way of providing programmability in tracers. We are aware that a set of more declarative debugging primitives would be desirable. However, to our knowledge Opium is the only environment to provide such a general framework. The abstract views existing in other systems can be easily integrated and indeed, many already are. Innovative abstract views are also provided. Furthermore casual users can extend the environment at the cost of writing Prolog programs, which do not need to be long to be powerful. Last but not least, the choice of Prolog as command and programming language gives a high flexibility. These features make Opium a unique tool for supporting program (execution) analysis.

Acknowledgements Anna-Maria Emde engineered Opium into a pre-release state and contributed many ideas. She, Thierry Le Provost, Pascal Van Hentenryck and Mark Wallace gave fruitful comments on early drafts of this article. Steven Prestwich helped with the English.

References

[Bergantz & Hassell, 1991] D. Bergantz and J. Hassell. (1991). Information relationships in Prolog programs: how do programmers comprehend functionality? *International Journal of Man-Machine Studies*, 35:313–328.

[Bratko, 1986] I. Bratko. (1986). *Prolog Programming for Artificial Intelligence*. International Computer Science. Addison-Wesley.

[Brayshaw & Eisenstadt, 1988] M. Brayshaw and M. Eisenstadt. (1988). Adding data and procedure abstraction to the Transparent Prolog Machine TPM. In Kowalski and Bowen [Kowalski & Bowen, 1988], pages 532–547. JICSLP'88.

[Brna *et al.*, 1993] P. Brna, A. Bundy, and H. Pain. (1993). A framework for the principled debugging of Prolog programs: How to debug non-terminating programs. In D.R. Brough, editor, *Logic Programming - New Frontiers*. Oxford: Intellect Books, 1992.

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

[Carro *et al.*, 1993] M. Carro, L. Gómez, and M. Hermenegildo. (1993). Some paradigms for visualizing parallel execution of logic programs. In D.S. Warren, editor, *Proceedings of the International Conference on Logic Programming*, pages 184–201, Budapest, Hungary. MIT Press.

[Casson, 1990] A. Casson. (1990). Event abstraction debuggers for layered systems in Prolog. In *Proceedings of the UK Logic Programming Conference*. Association for Logic Programming- UK Branch.

[Cochard, 1990] J.-L. Cochard. (1990). A graphical representation environment for complex Prolog structures. In M. Ducassé et al. (eds), *Proceedings of ICLP'90 Workshop on Logic Programming Environments*, Eilat, June 1990. Technical Report, ECRC IR-LP-31-25, European Computer-Industry Research Centre. LPE'90.

[Dewar & Cleary, 1986] A.D. Dewar and J.G. Cleary. (1986). Graphical display of complex information within a Prolog debugger. *International Journal of Man-Machine Studies*, 25(5):503–521.

[Ducassé, 1992] M. Ducassé. (1992). Analysis of failing Prolog executions. In *Actes des Journées Francophones sur la Programmation Logique*, Lille. http://www.irisa.fr/lande/ducasse/

[Ducassé, 1999] M. Ducassé. (1999). Opium: An extendable trace analyser for Prolog. *The Journal of Logic programming*. Special issue on Synthesis, Transformation and Analysis of Logic Programs, A. Bossi and Y. Deville (eds).

[Ducassé & Noyé, 1993] M. Ducassé and J. Noyé. (1994). Logic programming environments: Dynamic program analysis and debugging. *The Journal of Logic Programming*, 19/20:351–384.

[Eisenstadt, 1985] M. Eisenstadt. (1985). Retrospective zooming, a knowledge based tracing and debugging methodology for logic programming. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence*. Morgan Kaufmann.

[Eisenstadt & Brayshaw, 1988] M. Eisenstadt and M. Brayshaw. (1988). The Transparent Prolog MachineTPM: an execution model and graphical debugger for Logic Programming. *Journal of Logic Programming*, 5(4):277–342.

[Emde & Ducassé, 1990] A.-M. Emde and M. Ducassé. (1990). Automated debugging of non-terminating Prolog programs. In S. Bourgault and M. Dincbas, editors, *Actes du Séminaire de programmation en Logique*, pages 89–103. CNET, Lannion.

[Kahn & Saraswat, 1990] K.M. Kahn and V.A. Saraswat. (1990). Complete visualizations of concurrent programs and their executions. In *Proceedings of the Workshop on Visual Languages*. IEEE, October 1990. Also published as Xerox PARC Technical Report SSL-90-38.

[Kowalski & Bowen, 1988] R.A. Kowalski and K.A. Bowen, editors. (1988). *Proceedings of the Joint International Conference and Symposium on Logic Programming*, Seattle, USA, August 1988. MIT Press. JICSLP'88.

[Kusalik & Prestwich, 1993] A.J. Kusalik and S.D. Prestwich. (1993). Programmer-oriented visualisation of parallel logic program execution. In M. Ducassé et al. (eds), *Proceedings of ILPS'93 Workshop on Logic Programming Environments*, Vancouver, October 1993. Publication IRISA, Campus de Beaulieu, 35042 Rennes, France. LPE'93.

[Lichtenstein & Shapiro, 1988] Y. Lichtenstein and E. Shapiro. (1988). Abstract algorithmic debugging. In Kowalski and Bowen [Kowalski & Bowen, 1988]. JICSLP'88.

[Lloyd, 1987] J.W. Lloyd. (1987). *Foundations of Logic Programming*. Springer-Verlag, Heidelberg. Second edition.

[Moroshita & Numao, 1986] S. Moroshita and M. Numao. (1986). Prolog computation model BPM and its debugger PROEDIT2. In *Proceedings of the 5th Logic Programming Conference*, pages 147–158, Tokyo. Springer-Verlag.

[Mulholland, 1997] P. Mulholland. (1997). Incorporating software visualization into Prolog teaching: a challenge, a restriction, and an opportunity. In A. Kusalik et al. (eds), *Proceedings of ICLP'97 Postconference Workshop on Logic Programming Environment*, pages 33–42. http://www.cs.usak.ca/projects/envlop/8WLPE.

[Naish, 1997] L. Naish. (1997). A declarative debugging scheme. *The Journal of Functional and Logic Programming*, 1997(3). http://www.cs.tu-berlin.de/journal/jflp/articles/1997/A97-03/A97-03.htm.

[Neufeld *et al.*, 1997] E. Neufeld, A. Kusalik, and M. Dobrohoczki. (1987). Visual metaphors for understanding logic program execution. In W. Davis et al. (eds), *Graphics Interface*, pages 114–120.

[Pelhat, 1987] S. Pelhat. (1987). Les boucles dans Prolog : Structures et origines. In *6e Seminaire de Programmation en Logique*, pages 153–172, Tregastel, France. CNET Lannion. In French.

[Pereira, 1986] L.M. Pereira. (1986). Rational debugging in logic programming. In E. Shapiro, editor, *Proceedings of the International Logic Programming Conference*, volume 225 of *Lecture Notes in Computer Science*, pages 203–210, London, UK, July 1986. Springer-Verlag.

[Pereira & Calejo, 1988] L.M. Pereira and M.C. Calejo. (1988). A framework for Prolog debugging. In Kowalski and Bowen [Kowalski & Bowen, 1988]. JICSLP'88.

[Saurel, 1987] C. Saurel. (1987). *Contribution aux Systèmes Experts : Développement d'un Cas Concret et Etude du Problème de la Génération d'Explications Négatives.* PhD thesis, ENSAE, Toulouse, Decembre 1987. In French.

[Schulte, 1997] C. Schulte. (1997). Ox explorer: a visual constraint programming tool. In L. Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 286–300. MIT Press.

[Shapiro, 1983] E.Y. Shapiro. (1983). *Algorithmic Program Debugging.* MIT Press, Cambridge, MA. ISBN 0-262-19218-7.

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

[Vaupel *et al.*, 1997] R. Vaupel, E. Pontelli, and G. Gupta. (1997). Visualization of and/or-parallel executions of logic programs. In L. Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 271–285. MIT Press.

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

A The buggy N-queens program

This is the simple "generate-and-test" version of the N-queens taken from [Sterling & Shapiro, 1994, p253, program 14.2], in which a bug has been added to the `attack/3` predicate. "The N queens problem requires the placement of N pieces on a N-by-N-rectangular board so that no two pieces are on the same line: horizontal, vertical or diagonal".

```
GOAL:    nqueens(4,Qs).
CORRECT: [3,1,4,2] ; [2,4,1,3] ; no (more) solutions
BUGGY:   no (more) solutions
BUG:     N-1 should be N+1 in the last clause of  attack/3.

nqueens(N, Qs) :-
        range(1, N, Ns),
        permutation(Ns, Qs),
        safe(Qs).
/*
 *  range(M, N, Ns)
```

```
 *  Ns is the list of integers between M and N inclusive
 */
range(M, N, [M|Ns]) :-
        M < N,
        M1 is M + 1,
        range(M1, N, Ns).
range(N,N,[N]).

permutation(Xs, [Z|Zs]) :-
        select(Z, Xs, Ys),
        permutation(Ys, Zs).
permutation([], []).

/*
 *  select(X, HasXs, OnelessXs)
 *  The list OneLessXs is the result of removing
 *  one occurrence of X from the list HasXs.
 */
select(X, [X|Xs], Xs).
select(X, [Y|Ys], [Y|Zs]) :-
        select(X, Ys, Zs).
/*
 *  safe/1
 */
safe([Q|Qs]) :-
        safe(Qs),
        not attack(Q, Qs).
safe([]).

attack(X, Xs) :-
        attack(X, 1, Xs).

attack(X, N, [Y|_Ys]) :-
        X is Y + N.
attack(X, N, [Y|_Ys]) :-
        X is Y - N.
attack(X, N, [_Y|Ys]) :-
        N1 is N - 1,            % fix: N1 is N + 1
        attack(X, N1, Ys).
```

B The buggy simplify

The problem, taken from [Bratko, 1986, p162, ex 7.1], is to write a procedure `simplify` to symbolically simplify summation expressions with numbers and symbols. Let the procedure rearrange the expressions so that all the symbols precede numbers. The program has been written by a programmer which had already programmed in other languages and was learning Prolog.

```
GOAL:    simplify(1 + a + b + a, S).
CORRECT: S = 2 * a + b + 1
BUGGY:   no (more) solutions.
BUG:     A clause missing for the ``incr'' predicate to match when
         the 2nd parameter = [].

simplify(Init,Sexpr) :-
        simplify2(Init,Int,Vars),
        construct(Int,Vars,Sexpr).

simplify2(L+A,I,Var) :-
        !,
        simplify2(L,I1,V1),
        treat(A,I1,V1,I,Var).
simplify2(A,I,V) :- treat(A,0,[],I,V).

treat(A,I,V,I1,V) :-
        integer(A),!,
        I1 is I+A.
treat(A,I,V,I,V1) :-
        incr(A,V,V1).
                                        % fix: add incr(A,[],[A/1]) :-!.
incr(A,[A/N|L],[A/N1|L]) :-
        !,N1 is N+1.
incr(A,[S|L],[S|L1]) :-
        incr(A,L,L1).

construct(I,[],I)  :- !.
construct(0,L,P)   :- !, construct1(L,P).
construct(I,L,P+I) :- !, construct1(L,P).
construct1([S],P)  :- !, pri(S,P).
construct1([S|L],L1+P) :- pri(S,P), construct1(L,L1).

pri(A/1,A) :-!.
pri(A/N,N*A).
```

C The buggy ancestor

This program is a toy program using a left recursive version of the well known `ancestor` predicate to check whether somebody is a king.

```
GOAL:    prince(david).
CORRECT: yes.
BUGGY:   endless loop.
BUG:     There is a typo in the last fact of father.

prince(X) :-
        ancestor(Y, X),
        king(Y).

ancestor(X,Y) :-
        father(X,Y).
ancestor(X,Z) :-
        ancestor(Y,Z),   % left recursion
        father(X,Y).

father(john, david).
father(peter, john).
father(george, pater).   % typing error pater should be peter

king(george).
```

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