OPIUM: An Extendable Trace Analyser for Prolog

Mireille Ducassé

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

OPIUM: An Extendable Trace Analyser for Prolog

Mireille Ducassé, IRISA/INSA

N ° 3257

Septembre 1997

THÈME 2

ISSN 0249-6399

OPIUM: An Extendable Trace Analyser for Prolog

Mireille Ducassé, IRISA/INSA *

Thème 2 — Génie logiciel
et calcul symbolique
Projet LANDE

Rapport de recherche n3257 — Septembre 1997 — 55 pages

Abstract: Traces of program executions are a helpful source of information for automated debugging. They, however, usually give a too low level picture of the executed program.

Opium, our extendable trace analyser for Prolog, is connected to a "standard" tracer. Opium is programmable and extendable. It provides a trace query language and abstract views of executions which solve the problems of low-level traces.

Opium has shown its capabilities to build abstract tracers and automated debugging facilities. This article describes the trace query mechanism, from the model to its implementation. Characteristic examples are detailed. Extensions written so far on top of the trace query mechanism are listed. Two recent extensions are presented: the abstract tracers for the LO (Linear Objects) and the CHR (Constraint Handling Rules) languages. These two extensions were specified and implemented within a few days. They show how to use Opium for real applications.

Key-words: Software Engineering, Automated Debugging, Trace Query Language, Program Execution Analysis, Abstract Views of Program Executions, Prolog.

(Résumé : tsvp)

To appear in the Journal of Logic Programming

* 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

OPIUM : Un analyseur de trace extensible pour Prolog

Résumé : Les traces d'exécutions de programmes sont une source d'information très utile pour le débogage automatisé. Elles donnent cependant une image de trop bas niveau du programme exécuté.

Opium, notre analyseur de trace extensible pour Prolog, est connecté à un traceur "standard". Opium est programmable et extensible. Il fournit un langage de requête sur les traces ainsi que des vues abstraites d'exécutions qui résolvent le problème des traces de trop bas niveau.

Opium a montré sa capacité à construire des traceurs abstraits ainsi que des facilités de débogage automatisé. Cet article décrit le mécanisme de requête, du modèle jusqu'à l'implémentation. Des exemples caractéristiques d'utilisation sont détaillés. Les extensions écrites au dessus du mécanisme de requête sont listées. Deux extensions récentes sont présentées : les traceurs abstraits pour les langages LO (Linear Objects) et CHR (Constraint Handling Rules). Ces deux extensions ont été spécifiées et implémentées en quelques jours. Elles montrent comment utiliser Opium pour des applications réelles.

Mots-clé : Génie Logiciel, Débogage automatisé, Langage de requête sur des traces, Analyse d'exécutions de programmes, Vues abstraites d'exécutions de programmes, Prolog.

Contents

1 INTRODUCTION

Debugging is a costly process, and automating it would significantly reduce the costs of software production and maintenance. However, it is unrealistic to aim at fully automating the task. In particular, programmers have to *understand* rapidly changing situations, examining *large amounts of data*. In the current state of the art, taking the place of programmer's understanding is beyond the capabilities of computers. Nevertheless, computers can significantly help programmers to select the data to be analysed [14].

The data used by program analysis are often restricted to the source code of the analysed programs. However, there is a complementary source of information, namely *traces of program executions*. Indeed, while debugging, programmers develop hypotheses about the errors they are seeking. These hypotheses are often related to execution details and require a tracer to be verified.

Tracers provide programmers with detailed information about steps of program execution. Users of tracers, although admitting that the tools provide useful information, often complain that they do not fulfil their needs. A step by step trace with all its details is too much information at too low a level of abstraction. Users cannot permanently concentrate their attention and are then likely to overlook relevant pieces of information. Furthermore, as the relevant pieces of information are not contiguous, programmers cannot use their "visual register of sensorial information" [35, p 69] which would allow them to perform pattern matching at low cost. A consequence of the low-level information provided by tracers is that their commands are not expressive enough; programmers cannot formulate what they want to verify in a precise enough way. Their short term memory is quickly overwhelmed by the sub-strategies which allow them to reach the objectives of the main strategies. Hence tracers do no fulfil Shneiderman's recommendation: *Since terminal users strive for "closure" in their work, interactions should be defined in sections so completion can be attained and information released [34, p225].*

Tracers achieve an important job, namely they extract information about executions. The problem is that this information should, in general, not be shown in its "raw" form to users. We propose to connect a trace analyser to a tracer. This can solve the abovementioned user interaction problems.

Opium, our extendable trace analyser for Prolog, provides a trace query model which is a solution to the ever growing command sets of usual tracers. With four primitives plus Prolog, users can already specify more precise trace queries than with the hard coded Commands of usual tracers. Programmers can therefore specify precisely the hypotheses they want to verify. Since the commands are selective, related pieces of information are contiguous. Abstraction mechanisms allow users to examine the executions at the levels of abstraction which suit them.

An efficient synchronous trace querying scheme makes the model practical for trace analysis "on the fly". With this scheme, *millions* of execution events can be analysed with satisfactory response times.

When trace analysis requires to also trace backwards, Opium handles a simple database which can be analysed in a transparent way from the user point of view. When storing and

analysing at the same time, *thousands* of execution events can be analysed with satisfactory response times. This is sufficient to fine tune reasonable chunks of programs.

The actual format of trace information depends on the traced language (in this case Prolog), and so does the fine-grained design of the efficient synchronous scheme. However, the proposed model of trace query depends only on the sequentiality of the execution, and the principles behind the design of the synchronous scheme do not depend on Prolog.

Opium is implemented in Eclipse, ECRC's Prolog system [27]. The proposed primitives have been used to develop high level debugging strategies.

In the following, we describe the most important mechanism of Opium: the trace query mechanism which allows debugging programs to be written. We define trace events and their attributes. The efficient scheme for forward trace querying and its extension for backward querying are presented. The implementation is described in some detail. Appendix A lists all the primitives implementing all the interfaces between the different modules. We, then, give two brief examples of how to use Opium to trace a particular program at abstract level. We list the current automated debugging extensions written in Opium and we detail two of them, two abstract tracers for two logic programming languages, LO and CHR. These two extensions were written within a few days, and they show how to use Opium for real applications.

2 A TRACE QUERY MODEL

In this section, we present the conceptual model underlying the trace query mechanism of Opium.

Our trace query model assumes that the execution under examination is modeled into a trace which is a history of execution events. It also assumes that the history is fully available at any time. Trace queries search this history, both forward and backward, in chronological order. Note that we do not discuss here how to produce trace histories. Note also that the actual implementation does *not* require a full history to be physically present, as discussed in sections 4 to 6.

An event is a structure which contains information about execution points (usually called breakpoints). The actual breakpoints of interest and the information contained in the structure both depend on the language being traced and on the potential applications of the trace analyser.

However, the model described in the following does not depend on the actual breakpoints or of their representation. It is only assumed that the information related to these breakpoints is structured and can be read by a program. How events are actually modeled in Opium is discussed in section 3.

Only four primitives are necessary to retrieve trace information from the trace history. They handle a pointer to the history, the **current event pointer**.

current(Event) returns the trace information related to the current execution event. When the execution starts, the **current event pointer** is initialized to the first execution event.

next moves the **current event pointer** one step forward in the history. If the pointer was already on the last event, the primitive fails and the pointer remains there.

previous moves the **current event pointer** one step backward in the history. If the pointer was already on the first event, the primitive fails and the pointer remains there.

repeat(Mark) is a variant of the usual `repeat` Prolog primitive. It will repeat only until the **current event pointer** reaches the mark. The marks are the end of the trace history (called **eot**) and the begining of trace (called **bot**). The specification of the primitive is as follows.

```
repeat(Mark) :- ( current_event_point_at(Mark)
                -> !
                ; true).
repeat(Mark) :- repeat(Mark).
```

These primitives together with Prolog make a powerful trace query language. In the following, we first show how the usual hard-coded tracing commands can be generalized by simple trace queries. We then show more sophisticated queries which, in general, are impossible to specify with usual tracers. The debugging session in section 7 shows how the model is actually applied inside Opium.

To describe the examples, we use the standard Prolog syntax. In particular, terms starting with an upper case letter are variables.

The given queries are top-level goals, they can be typed in by users. However, any complex query can be abstracted by defining a new command:

```
new_command :- <complex query>.
```

2.1 Tracing The Very Next Event

The information related to the next event is retrieved and displayed, as follows:

```
?- next, current(Event), display(Event).
```

Users can implement various `display` procedures dedicated to their application. For example, several procedures can be implemented to display a data structure at different levels of abstraction. Users can then decide "on the fly" which one to use at a given moment.

2.2 Exhaustive Trace

Providing an exhaustive trace is straightforward with the backtracking possibilities of Prolog. The following goal can be read as "*repeatedly get and display the next event until the end of the trace history is reached.*" The request will, of course, ultimately fail, but all the execution events will have been traced.

```
?- repeat(eot), next, current(Event), display(Event), fail.
```

As the sequence `repeat(eot), next, current(Event)` is often used, it is turned into a new predicate `next(Event)`:

```
next(Event) :- repeat(eot), next, current(Event).
```

2.3 Conditional Breakpoints

Breakpoints are the basis of traditional tracers. More sophisticated tools provide their users with a set of possible conditions and actions attached to breakpoints. In our model, as we use predicates and we can access the trace data, any condition on the retrieved event can be inserted between the retrieval and the display. The following goal can be read as "*repeatedly get the next events until one satisfies the* `condition`*, and display this* `Event`".

```
?- next(Event), condition(Event), display(Event).
```

A condition can, of course, check any aspect of the events. For example, a user interested in failures could type in:

```
?- next(Event), isa_failure(Event), display(Event).
```

Displaying is just one example of possible actions. A more general query is:

```
?- next(Event), condition(Event), action(Event).
```

where `action` can be any predicate dealing with the information contained in an event. For example, a user may want to count the number of failures of an execution. He could type in:

```
?- next(Event), isa_failure(Event), count(Event), fail.
```

Much can be achieved with the previous scheme; however, programmers specify query according to what they currently see, in particular according to the current event. Therefore conditions often refer to both the initial event (starting point of the search) and the retrieved event. In the following query the initial event is retrieved and used to check the further retrieved events.

```
?- current(Event0), next(Event), cond(Event0,Event), display(Event).
```

For example, a common tracing command, often called `skip`, jumps to the end of the execution of the current goal. A high-level specification of it is:

```
skip :-
   current(Event0),
   next(Event),
   related_to_same_goal(Event0, Event),
   isa_success_or_failure(Event),
   display(Event).
```

Whenever `skip` is called, the current event is retrieved, then the next events are repeatedly retrieved until one is related to the same goal as the initial event and it is related to either a success or a failure of the goal. The successful event is then displayed.

Note that once the user has typed in the whole query and has seen the first event which satisfies the condition, the next related event can be accessed by simply relying on Prolog backtracking (typing ";" at top-level).

2.4 Verifying Hypotheses

As emphasized earlier, users develop hypotheses about possible errors. They want to verify these hypotheses. For example, one may want to know whether a particular behaviour occurs. In such a case it is not useful to display the related event(s), a simple `yes/no` answer is sufficient. The following goal will check whether there is an event which satisfies a given condition:

```
?- current(Event0), next(Event), condition(Event0, Event).
```

If so the tracer will be positioned on the first event after the current event which satisfies the condition, otherwise the goal will fail and the "current event" pointer will be positioned on the last event.

2.5 Sets of Events

Prolog can be used in a natural way to stop tracing where the user desires. For example, the following goal can be read as "*repeatedly print the forthcoming events which fulfil the condition until one of them fulfils the stop condition (or the trace is exhausted)*".

```
?- next(Event), condition(Event), display(Event), stop_at(Event).
```

For example, if we want to trace all successes until we reach a success of the current goal we could specify a query like:

```
?- current(E0),next(E),isa_success(E),display(E),same_goal(E0, E).
```

If the execution is finished before the stop condition is satisfied the query simply fails.

One could take further advantage of Prolog facilities to collect, for example, a set of events satisfying a condition and analyse them afterwards:

```
?- bagof(E, (next(E),condition(E)), Events), analyse(Events).
```

2.6 Querying Backward

The `previous` primitive can be used in trace queries in the same way as it has been illustrated so far for the `next` primitive. For example, the following goal can be read as "*repeatedly get the previous events until one of them fulfils the condition (or the beginning of the trace is reached), if such an event is found display it.*".

```
?- repeat(bot), previous, current(E), condition(E), display(E).
```

As for `next/1`, the sequence `repeat(bot), previous, current(Event)` is turned into a new predicate `previous(Event)`:

```
previous(Event) :- repeat(bot), previous, current(Event).
```

A specification of a command which displays the event related to the call of the current goal is:

```
call_event :-
     current(E0),
     (  isa_entry(E0)
     -> display(E0)
     ;  previous(E), isa_entry(E),same_goal(E0,E),display(E)
     ).
```

If the current event is related to the call of the current goal it is displayed. Else the trace history is repeatedly searched backwards for the first event which is related to a call and whose goal is the same as the goal of the initial event.

Queries can of course mix forward and backward tracing in any way.

3 MODELING PROLOG EXECUTIONS

In this section we describe a possible specification of Prolog execution events. We do not discuss in this article how to produce them. This requires a full article of its own, some hints can be found in [16].

As already mentioned the actual representation of events is independent of the trace query models presented in the previous section. The principles of the optimizations presented in the forthcoming sections are also independent from the event representation, it is, however, easier to describe them using the actual representation.

3.1 A slightly extended box model

We use an execution model close to the box model of Byrd [7] in which execution events are traditionally bound to goals. The types of events are called `ports`. A *call* event tells that g is invoked and gives the instantiation of its arguments at the moment of the invocation. A *fail* event tells that g fails. An *exit* event tells that g succeeds and gives the resulting

$$
\begin{array}{lll}
Goal\_Events & ::= & call \quad Contd \\
 & & \\
Contd & ::= & Direct\_Failure \\
 & | & Direct\_Proof \\
 & & Proof\_On\_Backtracking^{*} \\
 & & [Fail\_On\_Backtracking] \\
 & & \\
Direct\_Failure & ::= & unify^{*} \quad fail \\
 & & \\
Direct\_Proof & ::= & unify^{+} \quad exit \\
 & & \\
Proof\_On\_Backtracking & & \\
 & ::= & redo \quad unify^{*} \quad exit \\
 & & \\
Fail\_On\_Backtracking & & \\
 & ::= & redo \quad unify^{*} \quad fail
\end{array}
$$

Figure 1: Specification of the event sequence of a goal execution in extended BNF notation. Terminals are in lowercase and represent the type of the event, namely the "port". Non-terminals start with an uppercase. The meaning of the symbols is as follows. | : or, [] : contents must be present at most once, + : at least once, $*$: any number of times, possibly zero.

instantiation of the arguments. A *redo* event tells that the execution is backtracking either to g or to one of its subgoals. We have added the $unify$ event which tells when the execution finds a clause that unifies with g and gives the resulting instantiation of the arguments, it also gives the unified clause. [1].

Figure 1 specifies the sequence of the execution events of a given goal g . Ports are in lower case letters. The execution of a goal g starts by a goal invocation (*call*), then either g directly fails or it directly succeeds. After a proof, g can succeed again if the execution later backtracks to g or one of its subgoals. Ultimately g can fail.

A failure, either direct or on backtracking, may or may not be preceded by $unify$ events. It may be that

- there is no clause which can be unified (there is no $unify$ event);
- for each unified clause there exists a subgoal that fails (there are possibly several $unify$ events).

In order to get a direct success, there must be at least one $unify$ event. The last one gives the clause used to "produce" the success.

For a success on backtracking, the unified clause (which contains the choice point) is not necessarily one related to g. It can be related to one of its subgoals which previously succeeded. In such a case, there is no $unify$ event. On the other hand, if g clauses are tried, it may be that several of them need to be tried before the execution succeeds (there are possibly several $unify$ events).

In an exhaustive trace the events related to a given goal are not necessarily consecutive. Events related to subgoals and siblings are intertwined with their own. This is illustrated, among other points, in Figure 3 section 3.3.

3.2 Event Representation

Prolog executions can easily be represented as histories of events of the same format. In Opium an execution event is represented by a structure which contains three types of information: *control flow*, *data*, and *source connection*.

Control flow information

time stamp chronological number attached to an event. Each event has a unique time stamp according to its rank in the trace [2].

goal invocation number attached to a goal. All events related to a given goal have a unique goal number given at invocation time.

[1] The tracer of Eclipse provides Opium with more ports. For example, it defines a *cut* port which gives detailed and useful information about the behavior of cuts. Two ports, *delay* and *resume*, give trace information about coroutining. The design and implementation of these ports require some fine tuning which is out of the scope of this article. The number of ports has no influence on the design of Opium.

[2] Note that the time stamp is only a counter of event and not a real time stamp as in [8].

Time stamp (`Chrono`)	`12`
Invocation number (`Call`)	`5`
Execution depth (`Depth`)	`3`
Event type (`Port`)	`unify`
Executed predicate (`Pred`)	`nqueens: range/3`
Argument values (`ArgList`)	`(2, 4, Ns)`
Unified clause (`Clause`)	`nqueens:range/3.1`

The default display of this event is: `12 5[3] unify range(2, 4, Ns)`

Figure 2: An example of event structure

execution depth attached to a goal. It is the depth of the goal in the proof tree, namely it is the number of its ancestor goals.

event type (port) attached to an event. It is one of the previously defined ports (call, exit, fail, redo, unify).

executed predicate attached to a goal. It is the name of the goal functor.

Data information

argument instantiation attached to a goal and an event. It gives the instantiation of the goal arguments when the event just occurred. In particular, for *unify* events it is the value of the argument after unification. For *redo* event it is the same instanciation as for the previous *exit* event.

Source connection information (to be able to link trace analysis with static analysis)

clause attached to a goal and an event. For *unify* events it gives the clause just unified. For other events it is irrelevant.

The event structure is illustrated by Figure 2. The identifiers in parentheses are the names used in Opium. The displayed structure is related to an event of the execution of a program solving the well-known Nqueens problem. A buggy version is given in Appendix B. It shows that the goal `range(2,4,Ns)` has been called in module `nqueens`. It was the 5th goal to be invoked, and it has only 2 ancestors (the execution depth is 3). Its execution failed at time `2741`. The clause from which this goal was called is the first clause of `range/3` in module `nqueens`.

```
p(X) :- q(X),r(X).     q(X) :- s(X).          s(a).
                       q(X) :- t(X).          s(b).

r(X) :- fail.          t(X) :- fail.

?- p(X).

1  1[1] Call  p(X)            18 2[2]   Exit  q(b)
2  1[1] Unify p(X)            19 6[2]   Call  r(b)
3  2[2]   Call  q(X)          20 6[2]   Unify r(b)
4  2[2]   Unify q(X)          21 7[3]     Call fail
5  3[3]     Call  s(X)        22 7[3]     Fail fail
6  3[3]     Unify s(a)        23 6[2]   Fail  r(b)
7  3[3]     Exit  s(a)        24 2[2]   Redo  q(b)
8  2[2]   Exit  q(a)          25 3[3]     Redo   s(b)
9  4[2]   Call  r(a)          26 3[3]     Fail   s(X)
10 4[2]   Unify r(a)          27 2[2]   Unify q(X)
11 5[3]     Call fail         28 8[3]     Call  t(X)
12 5[3]     Fail fail         29 8[3]     Unify t(X)
13 4[2]   Fail  r(a)          30 9[4]       Call fail
14 2[2]   Redo  q(a)          31 9[4]       Fail fail
15 3[3]     Redo   s(a)       32 8[3]     Fail  t(X)
16 3[3]     Unify  s(b)       33 2[2]   Fail  q(X)
17 3[3]     Exit   s(b)       34 1[1] Fail p(X)
```

Figure 3: An example of exhaustive trace

3.3 Example

Figure 3 shows an exhaustive trace for a toy program. A trace "line" displays the control flow and data slots of the corresponding event structure.

The invocations of `r/1` lines 9 and 19 do not relate to the same goals. They have different invocation numbers, respectively 4 and 6, which help to distinguish them.

The goal events of `q` are:

```
3  2[2]   Call  q(X)
4  2[2]   Unify q(X)
8  2[2]   Exit  q(a)
14 2[2]   Redo  q(a)
18 2[2]   Exit  q(b)
24 2[2]   Redo  q(b)
27 2[2]   Unify q(X)
```

```
33 2[2]   Fail  q(X)
```

The `q` goal directly succeeds and produces the solution `X = a` then, thanks to one of its subgoal, it succeeds on backtracking and produces the solution `X = b` , it eventually fails on backtracking after exhaustion of all of its choice points. Note that the time stamps are not consecutive.

On line 14, the *redo* event tells that the execution backtracks inside the box of `q`. It actually backtracks to `s`, as told by the *unify* related to `s` on line 16. At some point the execution does backtrack exactly to `q`, which is shown line 27.

4 AN EFFICIENT SCHEME FOR FORWARD TRACE QUERIES

Our trace query model assumes that a trace history is available. A trace history can represent a huge amount of data but in this section we show that it does not need to be explicitly created when searching forward.

With our trace query model much can be achieved with forward queries alone (ie with the `current` and `next` primitives). For example, the abstract tracers described in sections 8 and 10 do not need to search the trace history backwards.

Forward trace querying can be done synchronously with the execution and event structures do not need to be stored. Furthermore, a pre-filtering mechanism further optimizes the model. This section describes these two aspects which make our model practical when querying large executions forwards. Issues related to handling, when needed, an actual database of events are discussed in section 5.

4.1 Synchronous Trace Querying

Synchronous trace querying means that there are two execution contexts, one for the traced execution, and one for the tracing execution (called Opium in the following). When a traced execution is started, Opium is notified when the first event takes place, the traced execution then waits. Opium executes trace queries locally until a `next/0` primitive is encountered. Then the traced execution is resumed and Opium waits. When the next event is reached, the traced execution stops and Opium takes back the hand until a `next/0` primitive is encountered, and so on.

Figure 4 illustrates a synchronous trace query. The programmer wants to query the execution trace of the Nqueens program for a 4x4 board given in Appendix B. When the execution reaches the first event, it notifies it to Opium which prompts the programmer for a trace query. The programmer enters a goal in order to search forward until an event at depth `2` is found. This event should then be displayed. At that moment Opium can only get information about the current event. It therefore returns control to the traced execution. When the traced execution reaches the next event it notifies it to Opium, which retrieves the current depth from the traced execution context and checks whether it is equal to `2`.

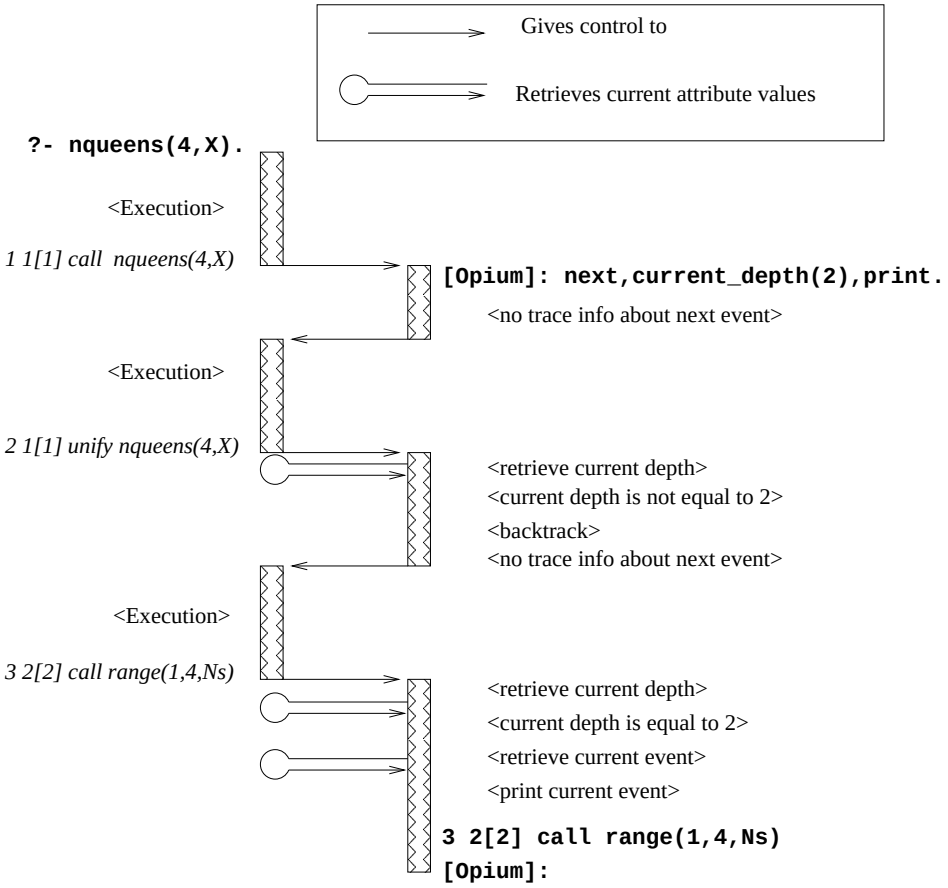

Figure 4: Illustration of a synchronous trace query

This is not the case and Opium execution backtracks. As, again, information about the next event is required, the traced execution is resumed. When the next event is reached, Opium verifies that the current depth is equal to the requested value. The current event is retrieved by the `print` command which displays the related information. The execution of the trace query is completed. The programmer is then prompted for another one.

This example calls for a major remark. When the traced execution reaches an event we have shown the event contents in italic in order to ease the understanding of readers. However, when the traced execution reaches an event there is no need to actually copy its value inside a physical structure. Providing a function per attribute of the structure is sufficient. These functions will be called when attributes values are needed. The event structure is, therefore, only a *virtual* structure in the case of synchronous trace querying.

This is important in terms of performance. Systematically copying all the current values of all the attributes in the current structure would take a lot of time, almost as much time as storing the trace history in a database. The trace queries usually need a limited number of attributes. Here, for example, only the `depth` is of interest. The profiling extension briefly described in section 9 uses only the `port` attribute. The attribute retrieval functions mean that programmers "pay" only for what they really verify.

4.2 Pre-filtered Retrieval

Synchronous trace querying is already reasonably efficient for many types of trace queries. However, there are still possibly many changes of context. At each of them, some information is copied from the traced execution context to the Opium context. For applications which check many attributes at each event and select few events, the response time can become prohibitive.

In order to alleviate the problem, we introduce a pre-filtering primitive which enables users to a priori constrain the search for events. The pre-filtering tests are processed in the same context as the traced execution. Whereas a non-optimized request requires to copy information from the traced execution context to the Opium context at every new execution event, the optimized request requires this copy only when an event satisfies the constraints, and values of the current event attributes are needed for further processing.

The pre-filtering primitive is `f_get(Chrono,Call,Depth,Port,Pred,ArgList, Clause)`, which applies to the event attributes described earlier. A user can specify values for some or all attributes. In the traced execution, when the next event is reached, the attributes are matched against the instantiated arguments of `f_get/7`. If the matching is successful the hand is given back to Opium, otherwise the traced execution is resumed until the next event. Note that the matching *does neither* modify the traced execution, nor the arguments of `f_get/7`.

Figure 5 illustrates a pre-filtered synchronous trace query. The traced program and the beginning of the session are the same as in Figure 4. This time, the programmer enters a goal which specifies an equivalent query but in an optimized form, `f_get(_,_,2,_,_,_,_)` replaces `next, current_depth(2)`. Opium returns control to the traced execution. When the traced execution reaches the next event, it checks locally whether the current depth is

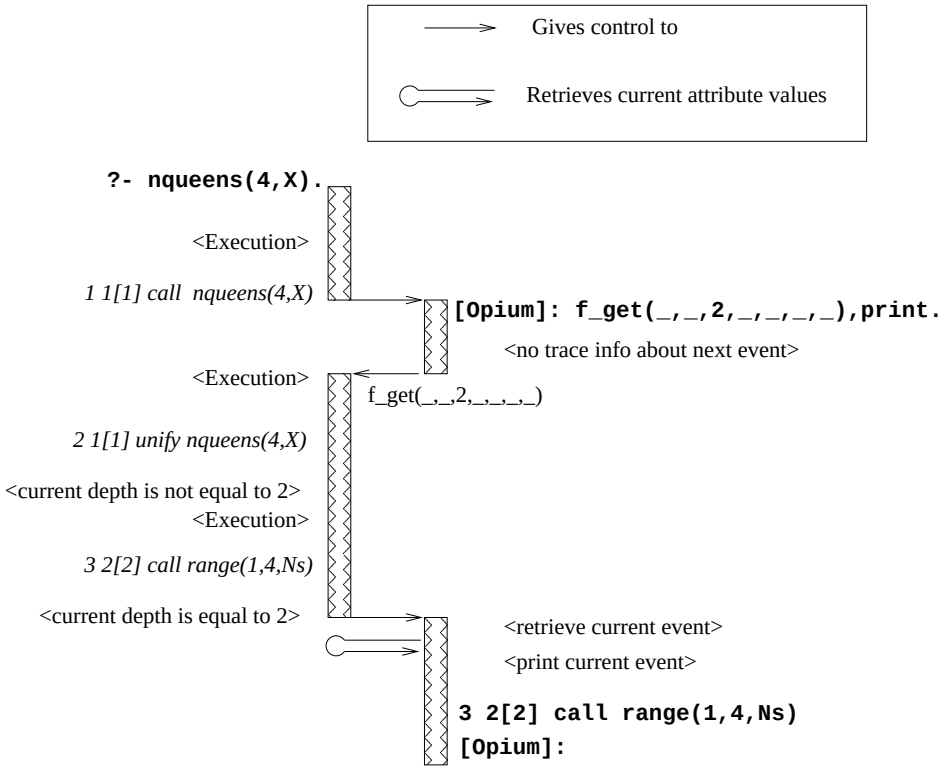

Figure 5: Illustration of a pre-filtered synchronous trace query

equal to 2. Here, there is no need to "retrieve" the current depth as it is already present in the context. As the current depth is not the requested value, the traced execution is resumed until the next event is reached. The traced execution verifies that, this time, the current depth is equal to the requested value. Opium is notified and proceeds. The current event is retrieved by the `print` command which displays the related information. The execution of the trace query is completed. The programmer is then prompted for another one.

The following query is more sophisticated. It checks whether `foo/2` is called at depth `3` with its second argument equal to an empty list:

```
?- f_get(_, _, 3, call, foo/2, [_, []], _).
```

Note that the `next/0` primitive is defined as:

```
next :- f_get(_,_,_,_,_,_,_,_).
```

For historical reasons, the current implementation of the `f_get` primitive only deals with control flow attributes but it is designed to allow complex matching which goes beyond simple unification. Besides exact values, users can also specify a list of possible values, an interval, or a negated value against which the matching is performed.

For example, the following request asks Opium to display the next event whose time stamp is less than `350`, whose execution depth is `3`, whose port is different from `redo`, whose predicate is either `foo` arity 2 or `bar` arity 2, and whose second argument is an empty list:

```
?- f_get(< 350, _, 3, ~redo, [foo/2, bar/2]),
   curr_arg([_, []]),
   print_line.
```

It optimizes the following query:

```
?- next, curr_line(Chrono, _, 3, Port, Pred),
   (Chrono >= 350 -> !, fail ; true),
   Port \== redo,
   (Pred == foo/2 ; Pred == bar/2),
   curr_arg([_,[]]),
   print_line.
```

In the following the examples use `f_get/5` as they reflect actual implementations.

5 EXTENDING THE SCHEME FOR FORWARD AND BACKWARD TRACE QUERIES

The previous scheme is very powerful for top-level queries. More sophisticated functions require to write Prolog programs, which we call *extensions* in the following.

In particular, we added backward facilities in order to be able to build different abstract views of the same portion of the execution as illustrated in [11]. Such extensions typically require to roll back and forth the execution several times.

In the previous scheme, at any given moment, Opium can only access information related to the last executed event. When searching the trace history backwards this is insufficient.

We have chosen to implement a simple trace database, easy to implement and use. We do not claim that this is the most efficient solution. The response times are however reasonable for middle sized executions (several thousands of events).

The trace query mechanism described in the previous section is based on the fact that the information related to the current event is modeled into a structure. It is then straightforward to have a simple database by storing all the event structures in their chronological order. This has the advantage of keeping the history of events consistent whether the queries are processed synchronously or from the database. The pre-filtering retrieval mechanism described in the previous section is therefore transparent as will be illustrated below. Furthermore, some primitives allow users to control what is stored in the trace database.

5.1 Pre-filtered retrieval

Adding a database does not change the trace query model, and the scheme described in the previous section is mostly untouched.

The backward tracing pre-filtering primitive is `b_get(Chrono,Call,Depth,Port, Pred, ArgList,Clause)` which is similar to the `f_get/7` primitive, except that it moves backwards and retrieves only stored events.

The search pilots the traced execution as before. Once the event structures are stored in the trace database, retrieving events "a posteriori", backwards or forwards, is similar to retrieving them on the fly. Instead of giving the control back to the traced execution, the pre-filtering primitives search in the trace database.

Searching forward synchronously or in the database is transparent for the user. A query will first search the database, and if not enough has been executed yet it will resume the traced execution.

This is illustrated in Figure 6. The Opium query is more sophisticated than in Figure 5. It reads as follows:

- stop at the next event related to the invocation of a goal at depth 2.
- retrieve the predicate name
- then go back to its parent goal invocation
- retrieve the time stamp
- retrieve the goal invocation number
- go to the success of this goal (if it exists),

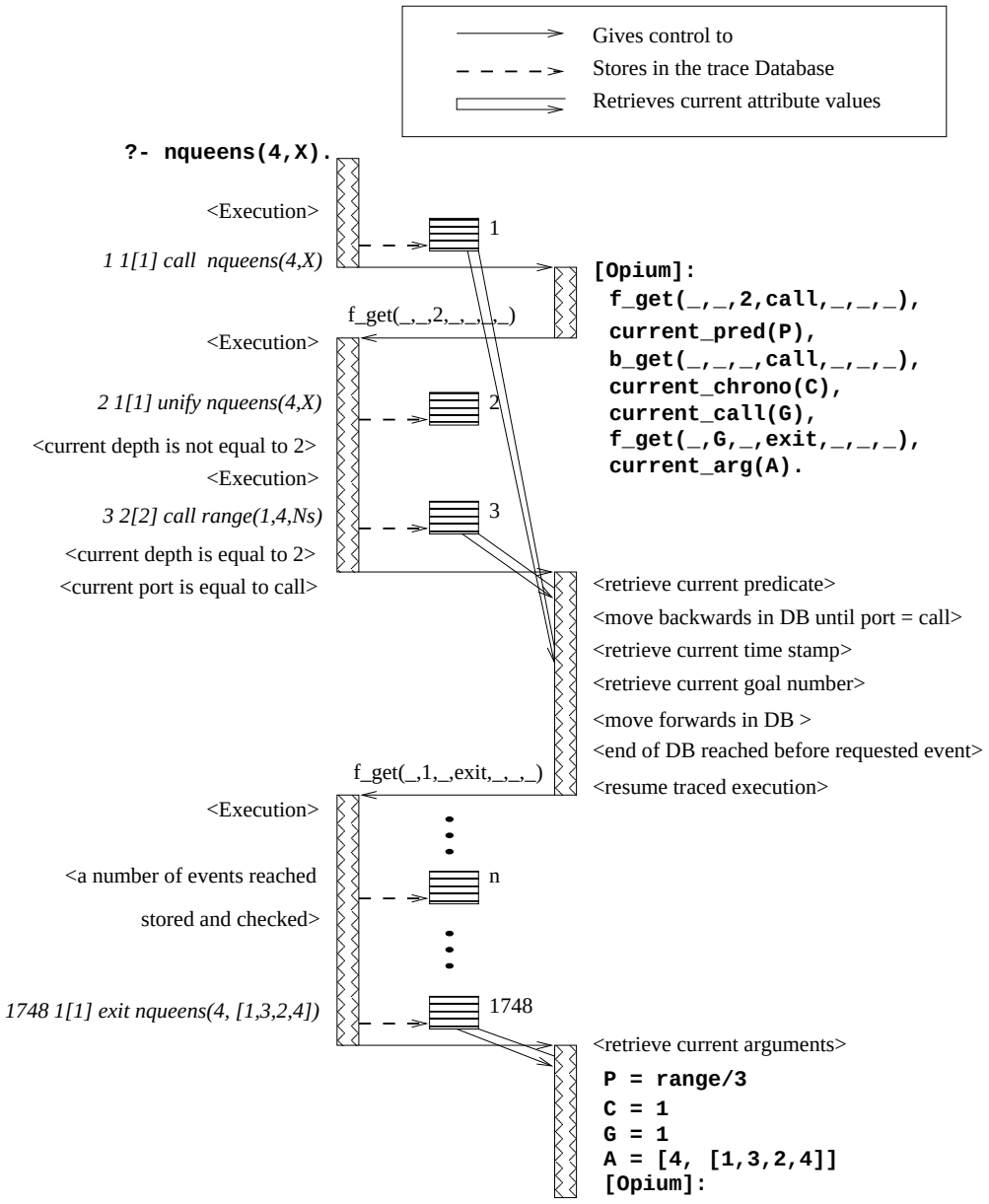

Figure 6: A trace query using the trace database

- if the goal did succeed, retrieve the resulting value of its arguments.

The execution of the query starts as before, Opium sends a pre-filtered request. As before, the traced execution checks the `depth` and `port` values of each event until they are respectively `2` and `call`. The novelty is that it also stores a new structure with contains the event value each time an event is encountered (i.e. filtering does not influence the trace database storage). When the event is as wanted, 3 events have been stored in the database. The traced execution waits. Opium proceeds and retrieves the current predicate from the database. Then it moves the current event pointer backwards in the trace database until the `port` is `call`. The time stamp and the goal number are retrieved from the database. Then Opium starts to search forward in the database for the next event whose goal number is `1` and port is `exit`. When it reaches the 3rd event the database is exhausted, the required events has not been found and the traced execution is not finished. The traced execution is therefore resumed. A number of events are reached, stored in the database and checked against the required values. When the required event is found, Opium takes the control, and retrieves the current arguments. The top-level of Prolog prints the values of the four variables `P`, `C`, `G` and `A`. The execution of the trace query is completed. The programmer is then prompted for another one.

5.2 Pre-filtered storage

As storing the events in the database takes time and space, it is necessary that users can decide not to use the trace database. The `set_recording` primitive enables them to switch the storage on or off. When recording is off only synchronous trace querying is allowed.

If an execution is made up of millions of events, it is not reasonable to store the exhaustive trace. However optimized the storage is, it still generates at least some bytes of information per event. If millions of events are stored, this results in a database of several megabytes. Considering that this database is not permanent, the costs in time and space can become prohibitive. It is, nevertheless, no problem to store several thousands of events and thus to analyse significant portions of executions a posteriori.

We have thus introduced 4 more primitives so that users and extensions can control the amount of trace which is stored: `record_line` can store isolated lines ; `reset_recording` empties the trace database ; `set_recorded_attributes/7` tells which attributes should be stored by default ; `record_attribute` can add the current value of an attribute to the current event structure in the database. If this structure was not yet recorded it has to be added to the database. The latter two primitives are not present in the current implementation of Opium.

6 A TWO-PROCESS IMPLEMENTATION

We have implemented the coroutining between Opium and the traced contexts with two processes communicating via messages.

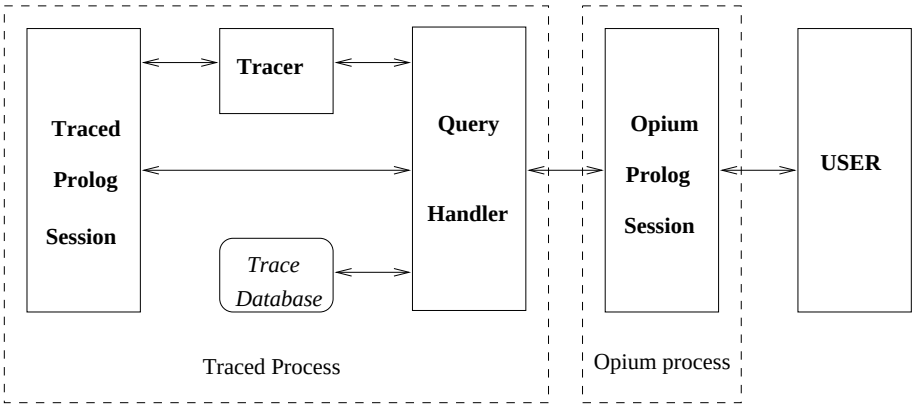

Figure 7: Architecture of Opium

The main advantage of a two process implementation is that it allows *meta-tracing*. Opium is mostly Prolog plus a handful of dedicated primitives, the debugging facilities developed for Prolog can therefore be applied to Opium programs. The first Opium session becomes the traced session of the second Opium session. Stable extensions of Opium have been used to debug debugging programs under development. The extensions can be large (there are currently over 5000 lines of debugging programs) it is therefore necessary to be able to debug them. As a matter of fact, a first implementation of Opium used coroutining inside the same process and we badly missed the debugging facilities.

In the following we first describe the architecture of the system. We then describe in some detail the synchronization done by the central part of the implementation. The precise interfaces between the query handler and the other components are given by lists of signals and primitives in Appendix A.

6.1 Architecture

Figure 7 shows the organization of the modules of the system.

The tracer stops the traced execution whenever an event is reached. Furthermore, the tracer knows were to fetch the information in the traced context to recollect the values of the event attributes. Hence the `current_<attribute>` primitives are implemented in the tracer.

The query handler synchronizes all the modules and implements the pre-filtering retrieval primitives, `f_get` and `b_get`. It stores event structures in the internal database when needed. It decides when to resume the traced execution or when to search in the

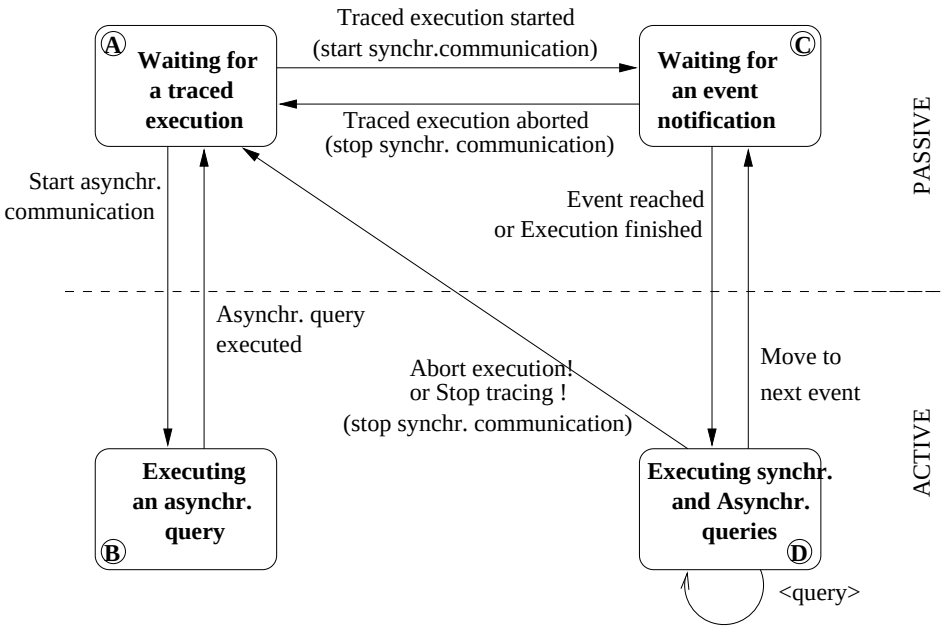

Figure 8: State diagram of the query handler

database. It executes a Prolog goal in the context of the traced execution, using the traced program. It pushes a new execution environment on top of the currently traced execution. In the current implementation we use the `break/0` primitive of Eclipse.

A remote execution can affect the traced program if new clauses are asserted. This is the only case where a query can change something in the traced execution context.

The Opium session executes the Prolog trace queries, typed in by the user, until a primitive is encountered which needs to call the query handler. It resumes its execution when the answer comes back.

We have grouped in the same process: the traced execution, the tracer, the query handler and the trace database in order to avoid information copying and context switches. With the current implementation, when checking attribute values the query handler only needs to call functions of the tracer which has a direct access to the traced execution data structures. The trace database also benefits from this access, some information is stored in terms of pointers to the traced program data structures.

6.2 The States of the Query Handler

The query handler is the central part of the implementation. It is a subroutine of the tracer. Its states are shown in Figure 8, and can be described as follows.

Initially the query handler is waiting for a traced execution to take place (A).

From state (A), if it receives a signal from the tracing session that an asynchronous query is to be sent it moves to state (B), waiting for the asynchronous query. When the query is executed and the result sent back to the tracing session the query handler moves back to state (A), waiting for a traced execution to start or another asynchronous query.

From state (A), if an execution is started the query handler starts a synchronous communication and moves to state (C) where it waits for the notification that an event is reached.

From state (C), if the query handler receives a signal from the extraction module telling that the execution has been aborted the synchronous communication is stopped and the query handler moves back to state (A).

From state (C), if the extraction module tells that an event is reached or the traced execution is finished the query handler moves to state (D).

At state (D), an interpreter of queries is called, which can execute both synchronous and asynchronous queries. The interpreter runs as long as the information that it can access is sufficient to answer the queries. Information can be in the trace database or in the current state of the traced execution context. When it lacks information corresponding to an execution step which has not yet been processed, the query handler sends a signal to the extraction module and moves back to state (C); control is given back to the main execution. If a debugging query requires to abort the traced execution or to stop tracing, a signal is sent to the extraction module, the query handler moves back to state (A) and the synchronous communication is stopped.

Note that when the query handler is in states (A) or (C), it is "passive", i.e. the main execution process is running in the traced session and some global variables tell in which state the query handler is. On the opposite, for states (B) and (D), procedures are called either on interrupt for (B) or regularly by the extraction process for (D). When both states are left, their corresponding procedures end and the main execution process is resumed.

When the traced execution is aborted either from the tracing session or from the traced session, we have chosen to stop the synchronous communication (i.e. to prevent users from sending any further queries about the execution). We estimated that if an execution is aborted it means that there is no more interest in it. On the other hand, when the execution is finished, the user may still be interested in it. For example, it might be that an execution has to run till it is finished because a debugging query is asking for an event that does not actually occur. The user has to be notified of such a case, and he can still examine the last event of the execution or the trace database if it exists.

6.3 Implementation and Performance Details

Opium is implemented inside Eclipse, ECRC's Prolog system [27], reusing the tracer of Eclipse. The two processes are connected by pipes, communicating with `write` and `read` Prolog primitives. Signals are sent using the interrupt-handler of Eclipse. Opium runs on Sun workstations SunOS under Unix.

When no trace database is stored, the pre-filtering primitive `f_get/5` which optimizes queries on control flow information can parse several million execution events with reasonable

response times. It takes only 1.5 of the time required by the standard debugger of Eclipse to check for a normal breakpoint. As the standard debugger of Eclipse is fast [28], pre-filtering of execution events according to control flow attributes offers a response time equivalent to the response time of equivalent functionalities hard coded in other Prolog debuggers. Therefore, the flexibility and precision of Opium do not slow down the tracing process if the pre-filtering primitive is used where possible. The time taken by other queries depends on their complexity.

When a trace database is stored the response time is still reasonable for storing and filtering at the same time thousands of events. This is sufficient to fine tune parts of programs. Note that once the trace database is recorded, pre-filtered retrieval applies and optimizes queries, the time overhead is thus paid only once.

Considering the gain in functionality, the implementation of Opium's kernel is relatively simple. The described primitives are implemented in about 1000 lines of well documented C, and 1000 lines of well documented Prolog. Over 5000 lines of debugging programs have already been written on top of the kernel.

7 A DEBUGGING SESSION WITH OPIUM

In this section we show a session illustrating Opium facilities. More commented examples can be found in Opium user's manual [15].

As a debugging session mainly depends on the understanding of the programmer, there is no deterministic way to debug. We only show possible paths. In the following examples, the full names of the commands are used to ease the understanding. However most of them have abbreviations which can be used to save typing. Whenever queries repeat part of previous ones the corresponding abbreviations are used.

The analysed program is a buggy Nqueens program given in appendix. The toplevel goal of the traced execution is `nqueens(4, Qs)` which should give the solutions for a 4x4 board. We can use the `continue` command of the top down zooming extension to see whether the goal succeeds or fails.

```
    1 1[1] call nqueens(4, Qs)
[opium]: continue.
    2746 1[1] fail nqueens(4, Qs)
```

The computation fails without producing any solution, we can use the leaf failure tracking described in [12].

```
[opium]: leaf_failure_tracking(1).
    1 1[1] call nqueens(4, Qs)
       56 23[2] call safe([1, 2, 3, 4])
       106 23[2] fail safe([1, 2, 3, 4])
                    subgoal failures abstracted (24 altogether).
       2643 697[2] call safe([4, 3, 2, 1])
```

```
      2689 697[2] fail safe([4, 3, 2, 1])
   2746 1[1] fail nqueens(4, Qs)
```

We can see that all the 24 invocations of `safe/1` fail whereas two should succeed. Note that it was important to abstract away these failures. Otherwise 50 lines would have been traced in one go. It would have been too much information for a user to grasp in one glance. In order to examine the failures of `safe/1` we can use the trace query language. We can set a breakpoint on `safe/1` using the `spy` command.

```
[opium]: spy safe/1.
```

As recording was on, we can go back to the first line of the traced execution using `goto`, otherwise we could have re-executed because the program is side-effect free. Then the `leap` command retrieves and prints lines related to spied points.

```
[opium]: goto(1).
    1 1[1] call nqueens(4, Qs)
[opium]: leap.
    56 23[2] call safe([1, 2, 3, 4])
[opium]: leap.
    57 23[2] unify safe([1, 2, 3, 4])
```

At this point, we can realize that a simple breakpoint is not precise enough. What we really wanted to see were the *failures* of `safe/1` and not all the lines related to it. We can refine the query by adding a condition after the leap. We then use `leap_np` which leaps to the next breakpoint but does not print the line. A line is only printed when it is related to a failure.

```
[opium]: leap_np, curr_port(fail), print_line.
    104 25[4] fail safe([3, 4])
```

Note that `curr_port(fail)` is equivalent to `curr_port(P), P == fail`.

The previous query still is not precise enough. We are only interested in the toplevel invocations of `safe/1` (i.e. at depth 2). We can add this new condition to the query. For the repeated commands we use their abbreviations. Note that an experienced user would have specified this query in one step but we wanted to show how progressive the use of the query language can be.

```
[opium]: l_np, c_port(fail), curr_depth(2), p.
    106 23[2] fail safe([1, 2, 3, 4])
```

In order to determine more easily whether the failure of `safe/1` is correct or not, we can use a program which displays the 4x4 board. We first retrieve the value of the arguments using the `curr_arg` primitive and display them in an adapted way using the `show_queens` predicate. This predicate is, of course, not an Opium predicate but it can be easily written by the programmer. It is an example of tracing facilities dedicated to a particular application.

```
[opium]: curr_arg([X]), show_queens(X).

             --- --- --- ---
            |   |   |   | x |
             --- --- --- ---
            |   |   | x |   |
             --- --- --- ---
            |   | x |   |   |
             --- --- --- ---
            | x |   |   |   |
             --- --- --- ---
X = [1, 2, 3, 4]
```

All the four positions are aligned, this failure is valid. We can now retrieve the next failure of `safe/1` at toplevel, displaying the board as previously, by simply concatenating the previous two queries.

```
[opium]: l_np,c_port(fail),c_depth(2),p, c_arg([X]), show_queens(X).
    328 67[2] fail nqueens: safe([1, 3, 2, 4])

             --- --- --- ---
            |   |   |   | x |
             --- --- --- ---
            |   | x |   |   |
             --- --- --- ---
            |   |   | x |   |
             --- --- --- ---
            | x |   |   |   |
             --- --- --- ---
X = [1, 3, 2, 4]     More? (;) ;
```

This failure is also valid. Using the usual toplevel backtracking facility of Prolog we can ask Opium to find the next appropriate line by simply typing ";".

```
    450 95[2] fail nqueens: safe([1, 3, 4, 2])
             --- --- --- ---
            |   |   | x |   |
             --- --- --- ---
            |   | x |   |   |
             --- --- --- ---
            |   |   |   | x |
             --- --- --- ---
            | x |   |   |   |
             --- --- --- ---
```

```
X = [1, 3, 4, 2]     More? (;) ;

     553 127[2] fail nqueens: safe([1, 4, 2, 3])
             --- --- --- ---
            |   | x |   |   |
             --- --- --- ---
            |   |   |   | x |
             --- --- --- ---
            |   |   | x |   |
             --- --- --- ---
            | x |   |   |   |
             --- --- --- ---
X = [1, 4, 2, 3]     More? (;)
```

The previous failures are also valid. It can become tedious to examine the 24 failures. If we are testing the program, we know that `safe/1` should succeed for `[2,4,1,3]` and `[3,1,4,2]`. We can directly ask Opium to check whether `safe/1` would not fail for these values. In such a case printing the full line is not even needed, showing the board is enough.

```
[opium]: l_np, c_port(fail), c_depth(2), c_arg([X]),
         (X == [2, 4, 1, 3] ; X == [3, 1, 4,2]),
         show_queens(X).

             --- --- --- ---
            |   | x |   |   |
             --- --- --- ---
            |   |   |   | x |
             --- --- --- ---
            | x |   |   |   |
             --- --- --- ---
            |   |   | x |   |
             --- --- --- ---
X == [2, 4, 1, 3]     More? (;)
```

The `safe/1` predicate indeed fails for this, actually safe, position. We can use the leaf failure tracking to understand why. We need the invocation number of this goal which can be retrieved using `curr_call`.

```
[opium]: curr_call(C), lft(C).

    1282 320[3] call not attack(2, [4, 1, 3])
    1335 320[3] fail not attack(2, [4, 1, 3])

 This is the end of the failing path.
```

The execution fails, because `not attack(2, [4, 1, 3])` fails, i.e. because `attack(2, [4, 1, 3])` succeeds. The leaf failure tracking stops because the symptom changes. Indeed, the problem is no longer that a predicate fails whereas it should succeed, but the opposite. The `attack(2, [4, 1, 3])` goal should not succeed as can be seen from the previously displayed board. Here, we can either list the code of `attack`:

```
[opium]: listing(attack).
    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,
            attack(X, N1, Ys).
```

and see that the recursive call of `attack/3` is incorrectly constructed, it should be `N1 is N + 1` and not `N1 is N - 1`. Or if we have not understood enough yet and want to trace further the execution of `attack/2`. We can go to the call of `attack/2` (therefore skipping the details of execution of `not`), and there `zoom` into its execution.

```
[opium]: f_get_np(_,_,_, call, attack/2), c_arg([2, [4,1,3]]), zoom.
    1286 322[5] call attack(2, [4, 1, 3])
        1288 323[6] call attack(2, 1, [4, 1, 3])
        1318 323[6] exit attack(2, 1, [4, 1, 3])
```

The latter goal should not succeed. As we know that the predicate has several clauses we can ask which one has been used to produce the solution:

```
[opium]: unif_clause.
    attack(X, N, [_Y|Ys]) :-
            N1 is N - 1,
            attack(X, N1, Ys).
```

Here again the error could be detected, but it is more obvious if we zoom into the execution of `attack(2, 1, [4, 1, 3])`.

```
[opium]: zoom.
    1296 323[6] next attack(2, 1, [4, 1, 3])
        1298 326[7] call N1 is 1 - 1
        1299 326[7] exit 0 is 1 - 1
```

It seems valid that one minus one is equal to zero. We ask to trace further at this level using the `continue` command.

```
[opium]: continue.
        1300 327[7] call attack(2, 0, [1, 3])
        1317 327[7] exit attack(2, 0, [1, 3]
```

Here one can notice that the second argument should never be `0` and that the recursive clause of `attack/3` is thus badly constructed.

8 EXAMPLES OF ABSTRACT TRACING

This section shows two small examples of more sophisticated abstract tracing. They go beyond trace queries, towards debugging programs.

It is noteworthy that the following demo of the puzzle program behaviour or the sketch of explanation for the expert system could be designed without touching the code of the programs. The tracing program is a separate program running in the Opium session.

8.1 A Puzzle

We give an example of abstract tracing of a Prolog puzzle, taken from Sterling and Shapiro's book [36, p259-260].

> *Three friends came first, second and third in a programming competition. Each of the three has a different first name, likes a different sport and has a different nationality. Michael likes baseball and did better than the American. Simon, the Israeli, did better than the tennis player. The cricket player came first. Who is the Australian? What sport does Richard play?*

The Prolog program is given in appendix. It is very elegant but people not familiar with Prolog may have two problems to understand it. Firstly, the data structures are a little abstruse. Secondly, the program does a systematic "generate and test" search. It can be hard to believe that "it works". Examining the behaviour of the program can help them with the latter problem. However, the bare Prolog trace is even worse than the code with respect to the first problem. For example, a typical line of trace looks like:

```
     16 8[4] call call((did_better(M1C1, M2C1, [friend(N1, C1, S1),
friend(N2, C2, S2), friend(N3, C3, S3)]), (name_is(M1C1, michael),
(sport(M1C1, basketball), nationality(M2C1, american)))))
```

When analysing the program one can notice that, in order to give a first approximation of the program behaviour, tracing the predicate `solve/1` can be sufficient. We can produce an abstract trace such as what follows.

```
[23] Trying to solve the following clue:
        Man1Clue1 did better than Man2Clue2 and
        name_is(Man1Clue1, simon)
```

```
        nationality(Man1Clue1, israeli)
        sport(Man2Clue2, tennis)
with:
        michael is C1 and plays basketball
        N2 is american and plays S2
        N3 is C3 and plays S3
```

A sketch of the Opium command, which produces this abstract trace follows. Note that it is used in the same way as usual trace commands.

```
clue :-
    f_get(_,_,_, [call, exit, fail], solve/1),
    curr_arg([L]),  L \== [],          % end of recursion, no interest
    curr_chrono(Chrono),               % number of the Prolog line
    printf("\n[%w] ", [Chrono]),
    curr_port(Port),                   % the Port is either call,
    write_header(Port),                %   exit or fail
    curr_arg([[(Pred, Goals) | _]]),   % the argument of solve is a
    write_clue( Pred, Goals, Friends), %   list of pairs
    printf("with:\n", []),
    write_friends(Friends).            % current state of solution
```

The first goal, `f_get(_,_,_, [call, exit, fail], solve/1)`, will get the first trace line which is related to entering or exiting the `solve` predicate. The rest of the clause retrieves different parts of the information contained in the Prolog trace line and prints it according to what the person designing the demo for the puzzle program thought it was appropriate. The `curr_...` predicates are Opium primitives. The `write_...` are user defined, indeed they depend on the particular program beeing traced.

This is not to say that this tracing command is "the" tracing command which will be meaningful to everybody. We just want to show that the Opium framework is flexible and powerful enough to adapt to the needs. In particular, one can refine the trace and provide commands which will give more details.

8.2 A Simple Expert System

We give an example of building a sketch of explanation for a simple expert system. The expert system and the specifications for abstract tracing were written by Brayshaw and Eisenstadt [4].

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 sort of lines (there are actually 66 lines altogether).

```
   1[1] call solve(X isa carnivore)
    2[2] call fact : X isa carnivore
    2[2] fail fact : X isa carnivore
   1[1] next solve(X isa carnivore)
    3[2] call Rule : if(Cond) then X isa carnivore
    3[2] next 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] next 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 a better 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 a better 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(M:solve/1) :-
     print_line.
my_print_line(M:(:)/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 line 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` displays the event according to this predicate value. Lines related to `solve` are displayed normally. Lines 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 Opium primitive `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.
```

9 TRACE ANALYSIS EXTENSIONS

Existing extensions are listed in the following. The last two, which are used to trace two logic programming languages, are detailed in the next section.

A loop analysis. The objective of this analysis is to support users on debugging apparently non-terminating Prolog executions by automating as much as possible the search for relevant debugging information. Only those parts of trace and source which are *essential* to understand the endless loop are 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 source and trace 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 description of this analysis can be found in [17].

A failure analysis. The objective of this analysis is to support users on debugging Prolog executions ending with *unexpected failures*, that is executions which result are simply "`no`" when something else is expected.

If an execution fails the first things to examine are failing goals (called "failures"). However, there can be very many failures. They cannot be simply displayed one after the other. Our failure analysis filters and structures the set of failures. Automated traversal of the failing search trees is provided. A complete description of this analysis can be found in [13, Ch. 8].

Abstract views of executions. The objective of abstract views of Prolog executions is to give high-level points of view about some aspects of the execution. Abstract views filter out irrelevant details; they restructure the remaining information; they compact it so that the amount of information given at each step has a reasonable size; and they eventually print the resulting information.

The abstract views provided so far include:

- abstract control flow information to illustrate a goal behaviour;
- data abstracts to display data values at different levels of abstraction;
- data flow abstracts to show the instantiation of variables.

In general, all the abstract views detected by the cognitive study of [2] are present (or can be very easily programmed) in Opium. More detailed descriptions of abstract views of Prolog executions can be found in [11].

A dynamic slicing tool for Prolog. This extension is currently under development. The objective is to adapt the slicing algorithm developed by Weiser [37] to Prolog. It analyses data and control dependencies to eliminate from a given program the parts which cannot be responsible for incorrect variable values [33]. Using execution traces gives a finer answer than simply using the source code.

Profiling the search component of a Prolog system. Opium has been used for automated profiling. A first extension was used to compare the backtracking efficiency of the Sepia [27] and KCM [30] Prolog compilers. A second extension was used to assess the potential gains of re-computation vs copying when implementing OR-parallelism. For this purpose a profiled search tree of a sequential execution containing both space and time information is built and this tree is analysed to provide overall cost information. This extension reached the limits of Opium, while space information could be retrieved and analysed satisfactorily, the time information was not accurate enough. Both extensions required thorough analysis of profiling information which could only be offered by an extendable profiler/debugger.

An abstract tracer for the LO language. LO (Linear Object) is an object oriented programming language based on linear logic [1], for which an interpreter written in Prolog exists. The objective was to provide LO users with a tracer which would give information following the operational semantics of LO (and not of Prolog). A few days were sufficient to build an operational tracer which could be customized further by the user to give demos of his own application.

An abstract tracer for the CHR language. CHR (Constraint Handling Rules) is a constraint logic programming language [19]. The objective was again to provide CHR users with a tracer at the proper level of abstraction. As the tracer was developed at the same time as the compiler, it also helped to develop the system itself.

10 SOME DETAILS OF TWO ABSTRACT TRACERS

The two abstract tracers mentioned above have been developed for real applications. Some details are given in the following.

Note that the brief description of the two languages relate to their state at the time where the Opium extensions were written. Readers interested in the two languages should search for more recent litterature.

10.1 The LO Abstract Tracer

LO (Linear Object) is an object oriented programming language based on linear logic [1]. At the time we wrote the Opium extension, there was merely one user of the LO interpreter. The operational semantics of LO was designed but the implementors were still working on the system. On the one hand, the LO designers and developers had little time to invest in writing a tracer. On the other hand, the LO user was badly in need of some support tool, especially as the interpreter was still changing.

The LO interpreter was written in Prolog, it was therefore possible to use Opium to provide some tracing facilities for the LO user at low costs. The LO Opium extension filters the execution of the LO interpreter and recognizes the events related to the LO user program.

10.1.1 LO Execution Model.

A computation in LO can be viewed as the activity of concurrent communicating agents. The rules describing the evolution of an agent are called methods. There are two kinds of methods, reactive methods which depend on receiving a message to be triggered and internal methods.

As for Prolog, the execution can be modeled with the help of "ports". Each port, except the last one, characterizes the status of one agent. The argument *State*, when displayed, refers to the state of the agent.

try (State) The agent has performed a transition and is ready to try the internal methods (first) to proceed.

become_ i (State) The agent has committed to an internal method and is ready to perform the corresponding transition.

bcast (Msgs) The agent has committed to a method (internal or reactive) which contains broadcast instructions, and is ready to broadcast the requested messages.

become_ e (Msg, State) The agent has committed to a reactive method triggered by the message *Msg*, and is ready to perform the corresponding transition.

sleep (State) The agent has tried - unsuccessfully - all the methods (internal or reactive) and must wait for further messages.

wake (Msgs, State) The agent has been reactivated by the incoming messages *Msgs*.

lookmsg (Msgs, State) The agent has finished to apply all the possible internal methods and is ready to explore the reactive methods with its message queue *Msgs*.

trymsg (Msg, State) The agent is ready to try the reactive methods capable of processing the message *Msg* which has been picked from its message queue.

deadlock All the living agents are asleep and only an external input may wake them.

One can notice that we are far from the traditional 4 ports of Prolog.

A LO event consists of a chronological number, a goal invocation number, a LO port (among the list given above), a list of messages (if any) and the state of the agent.

10.1.2 Implementation in Opium.

The implementation of the LO Opium extension could be done without any modification of the code of the LO interpreter. The two basic commands for the LO extension retrieve the next LO event and print it.

The command which retrieves the next LO event is basically as follows.

```
lo_get(Chrono, Call, LOPort, Messages, State) :-
        implementation_predicates(PredList),
        f_get(_,_,_, [call, cut, delay, resume], PredList),
        build_lo_event(Chrono, Call, LOPort, Messages, State).
```

First the list of the main Prolog predicates which implement the LO interpreter is retrieved. Then the Opium primitive `f_get/5` is used to retrieve only the Prolog events related to these predicates. Note that some filtering is also done on the Prolog ports, only `call`, `cut`, `delay`, and `resume` ports are of interest. The `build_lo_event` then finishes to filter the Prolog events and reconstitutes the LO events:

```
build_lo_event(Chrono, Call, LOPort, Messages, State) :-
        curr_line(Chrono, Call, _, PrologPort, ImplemPred),
        select_event(PrologPort, ImplemPred, LOPort, Messages, State).

select_event(call, prove/2, try,       _, State) :-  % Agent activation
        curr_arg([State,_]).
select_event(cut,  prove/2, become_i,  _, State) :-  % Internal transition
        curr_arg([State,_]).
select_event(call, prove/4, trymsg, Mesg, State) :-  % Start treating Message
        curr_arg([Message, State |_]).
% ...
```

First the attribute values of the Prolog event which are of interest, either to send back or to compute further information, are retrieved with the `curr_line` Opium primitive. Then according to the implementation predicate and the Prolog port, the LO port is deduced and the Messages and State of the agent are retrieved from the arguments of the Prolog implementation predicates. Note that `prove/2` and `prove/4` are Prolog predicates implementing the LO interpreter.

Dedicated primitives display the messages and state of an agent. The state of an agent is a multiset of resources. The following one shows how, from the internal list representation of a given state, the external representation can be easily reconstructed. The external representation, the LO programmers are used to, is of the form `Literal1@Literal2@...@Literaln`.

```
write_lo_state([Lit]) :-
        !,
        write(Lit).
write_lo_state([Lit | LitRest]) :-
        write(trace, Lit),
        write(trace, ' @ '),
        write_lo_state(LitRest).
```

For a given application, agents are usually more specific and the programmer can customize the `write_lo_state/1` primitive to take the characteristics into account. This is actually what happened in our case, the LO programmer decided to display the "important" literals first, then the less important ones, and did not display the literals which were not important. He could also use several display primitives according to whom the trace was shown. This feature helped him to use the LO Opium extension to give demonstrations of his application.

10.1.3 Discussion.

The resulting extension is only a few hundreds lines long. Even if it is not very sophisticated, it covers the basic needs for tracing LO. It took a couple of days to be designed and implemented (including the model described earlier). The clear operational semantics of LO was a major help.

One can notice that the Opium implementation depends on the actual names of the predicates used to implement the LO interpreter. This led to some problems when the interpreter was changed without updating the Opium extension. Nevertheless, the LO user could easily get some tracing (and demonstration) support which he would have hardly got without Opium.

10.2 The CHR Abstract Tracer

Constraint handling rules (CHR) are a tool for rapid prototyping of constraint handlers [19]. They are embedded in a host language, in this case Prolog, and enable user-defined constraints to be written. They are essentially a committed-choice language consisting of guarded rules with multiple heads.

Constraint rules are basically of two kinds, simplification and propagation. Simplification replaces constraints by simpler constraints while preserving logical equivalence (e.g. `X>Y,Y>X <=> fail`). Propagation adds new constraints which are logically redundant but may cause further simplification (e.g. `X>Y,Y>Z ==> X>Z`).

A CHR program contains both constraint rules and plain Prolog code. The CHR compiler transforms the constraint rules into Prolog code. A compiled CHR program is, thus, a Prolog program which can be traced with Opium.

At the time we started to work on the Opium extension, the operational semantics was still under progress. The Opium extension was written at the same time as the compiler and helped to debug the compiler. Once a tracing prototype had been designed with Opium, a hard-coded tracer could be easily implemented.

10.2.1 CHR Execution Model.

As already mentioned a CHR program contains both constraint rules and plain Prolog code. The model joins the Prolog model and new ports to cover the handling of constraint rules. The following description is adapted from [5].

add A new constraint is added to the constraint store.

already_in A constraint to be added was already present.

try_rule A rule is tried.

delay_rule The last tried rule cannot fire because the guard did not succeed.

fired_rule The last tried rule fires.

try_label A label declaration is checked.

delay_label The last label declaration delays because the guard did not succeed.

fired_label The last tried label declaration succeeds, therefore the clauses of the associated constraint will be used for built-in labeling.

Each constraint is identified with a unique integer identifier. Each rule is identified either with its name as given in the source, or, if a rule has no name, with a unique integer identifier.

A CHR event has 6 attributes: the chronological event number, the CHR port (as listed above), the current constraint(s) to be rewritten, the constraint number(s), the associated rule and the rule name.

10.2.2 Implementation in Opium.

The implementation of the CHR Opium extension is slightly more complicated than the previous one. Indeed, a compiled CHR program contains two kinds of Prolog code: firstly, the user Prolog code which has not been modified by the compiler and which must be traced straightforwardly; and secondly, the constraint handling code which must be traced in an abstract way.

Tracing constraint handling could not be easily deduced from spy-points as in the LO extension. The solution taken for the CHR was to modify the compiled code to introduce a "fake" predicate gathering the interesting trace information in its argument. Furthermore, the predicates handling the constraints are marked untraceable.

For example, the following rule, called `min_eq`,

```
min_eq @ minimum(X, X, Y) <=> X = Y.
```

tells that the minimum of two equal numbers is that number. It is basically compiled into

```
'CHRminimum'(minimum(X1, X1, Y1), CstrNb) ?-
        coca(try_rule(CstrNb,
                      minimum(X1, X1, Y1),
                      min_eq,
                      minimum(X, X, Y),
                      replacement,
                      true,
                      =(X, Y)  )),
        !,
        coca(fired_rule(min_eq)),
        =(X1, Y1).

?- untraceable('CHRminimum_3').
```

The only aim of the `coca/1` predicate is to gather trace information. The execution of `coca/1` always succeeds. Note that the `=(X1, Y1)` goal is a normal Prolog goal which will be straightforwardly traced.

The basic two primitives of the CHR extension first move to the next event (either plain Prolog or constraint handling) and second retrieve the value of the event attributes. The command which moves to the next event is basically as follows.

```
chr_get :-
        f_get(_,_,_,_,_),
        allowed_event.

allowed_event :-
        curr_pred(_Module:Pred),
        ( Pred = coca/1
         ->
          curr_port(call)
        ; true
        ).
```

It moves to the very next event (remember that many predicates are not even traced), then checks whether the event is allowed. An event is allowed if either it corresponds to the invocation of `coca/1`, or it traces another predicate (in which case it is necessarily a Prolog predicate of the user's source code).

Prolog events are then retrieved in a standard way whereas CHR events are retrieved using the following primitive:

```
chr_curr_event(Chrono, ChrPort, Cst, CstNb, Rule, RuleName) :-
        curr_pred(coca/1),
        curr_chrono(Chrono),
        curr_arg([Event]),
```

```
        build_chr_event(Event, ChrPort, Cst, CstNb, Rule, RuleName).

build_chr_event(prolog_call(CstNb, Cst),   chr_call, Cst, CstNb, true, "").
build_chr_event(add_one_constraint(CstNb, Cst), add, Cst, CstNb, true, "").
build_chr_event(try_clause(CstNb, Cst, Head, Guard),
               try_call, Cst, CstNb, (callable Head if Guard), "").
build_chr_event(try_rule(CstNb, Cst, RuleName, Head, Kind, Guard, Body),
               try_rule, Cst, CstNb, Rule, RuleName) :-
        ( Kind = augmentation
        ->
          Rule = (Head ==> Kind | Guard)
        ;  Kind = replacement
          ->
           Rule = (Head <=> Kind | Guard)
        )
%...
```

The `chr_curr_event` predicate first checks that the traced predicate is indeed `coca/1` then it retrieves the chronological number and the argument of `coca/1`, which allows the contents of the CHR event to be reconstructed.

10.2.3 Discussion.

The CHR Opium extension is several hundreds lines long. It has been developed at the same time as the compiler, and helped to refine the operational semantics of the language.

The internal CHR predicates can be easily set traceable, the compiler implementor could trace at the same time both the low-level details and the abstract CHR level. This helped to develop the compiler.

The specification of the `coca/1` predicate was a first step towards a hard-coded tracer. Because the Opium machinery was already present the compiler designer could easily play with the trace, hence designing smoothly the CHR tracer.

11 DISCUSSSION AND RELATED WORK

Information modeling

Early programmable debuggers tried to model the debugging process. Debugging is, however, a sophisticated creative process which is far from being understood. As a result the number of primitives of these systems was very large. For example, the descriptions of the Dispel language designed by Johnson [23] or the language of Lazzerini and Lopriore [24] take approximatively 10 pages each. Moreover, this approach cannot lead to a satisfactory debugger as there is no criteria to determine if the set of primitives covers all the needs of a user.

Opium essentially *models the execution data.* Extensions, of course, provide some debugging processing but the `current` and `get` primitives plus a full programming language make sure that users can query on whatever aspect of the execution they need to.

Some recent programmable debuggers also model execution data. Duel, by Golan and Hanson [20] and Acid by Winterbottom [38], emphasize program states. Opium, as Forman by Fritzson et al. [18] and Dalek by Olsson et al. [31], is based on events which also model the control and dataflow of executions.

All the debuggers which model the execution data share a simplicity of design and implementation.

Current state analysis

We have emphasized the importance of the `current` primitive which enables to retrieve parts of the current state. This is conforted by Duel [20] whose main functionality is to enable users to investigate states of executions. They have designed and implemented an ad hoc interpreter, inspired by the Icon language, [21], with "generators" for expressions generating 0 or more solutions. In Opium we use Prolog which is satisfactory as such. We did not need to customize it.

In Deet, by Hanson and Korn [22], data structures can be displayed graphically using Tcl/Tk. Opium uses the full Prolog environment of Eclipse which provides, among other features, an interface to Tcl/Tk. Users can therefore choose to graphically display their data.

Trace querying

Most debuggers rely on "static" breakpoints. The user decides to spy a particular place in the source. A breakpoint is planted in the symbol table and whenever the execution relates to this place it stops and gives the hand to the programmer. When the user wants to see another place he first has to remove the static breakpoint and to plant another one. These breakpoints can have conditions attached to them.

In Opium we also have static breakpoints attached to predicate definitions. However, the main mechanism is fully dynamic. The prefiltering primitives, `f_get` and `b_get` can analyse *all* events. This gives a much finer grained analysis and we have shown that it is efficient enough.

Furthemore, a simple "if condition then action" scheme is not powerful enough. We have given examples of queries in section 2 which combine condition and action predicates in very sophisticated way. A langage based on predicate logic is mandatory in order to state precise and concise trace queries on the fly.

Synchronous tracing and Storage

Opium does not need to wait for an execution to finish in order to analyse it, as opposed to the systems which need an actual trace database: the Omega environment [32], the Yoda

debugger [25], the IL system [9] and the Traceview system of Malony et al. [26]. Traceview can handle traces of several languages. This cannot be easily provided by Opium because of optimizations which require the implementation of part of the analysis module in the traced session. We believe, however, that to be able to efficiently trace synchronously is an important feature of a debugger. In the four systems previously cited *all* events are first stored in some sort of database whereas in Opium users can control the tracing process while the execution is processed. Secondly, the *whole* information related to an event has to be computed, whereas in Opium the costly parts are computed only if needed.

On the other hand, the synchronous programmable systems do not enable to store anything. Thus sophisticated analyses which require to go back and forth in an execution cannot be implemented. In Opium, parts of the trace can be stored, enhancing the debugging capacity while keeping performances reasonable.

Opium is therefore the only debugging system where efficient synchronous trace querying coexists with a trace database management.

Debugging language

Some programmable debuggers are based on a full programming language. For example, UPS designed by Bovey et al. [3], Cola designed by Dencker [10], Dalek [31] and Acid [38] offer C-like interpreters. However, a procedural language like C does not seem appropriate here. Only a few primitives need to be efficient. What is important is the prototyping power of the debugging language so that users can easily enter trace queries on the fly to adjust their diagnosis process. Prolog, with its expressive power, is better suited to this than procedural languages. Failure driven loops and unification help to express queries which otherwise would require very complicated control structures. The IL system of Cohen and Carpenter [9] provides regular expressions to specify debugging commands, which is more satisfactory than procedural languages but does not offer the power of a full programming language.

Meta-debugging

Opium is the only programmable debugger which offers meta-debugging. Stable extensions of Opium have been used to debug debugging programs under development. Dalek [31] seems a powerful system, for example it offers interesting extraction control, but it only offers limited meta-tracing hooks. We cannot agree with its authors who argue that debugging the debugging programs does not require sophisticated tools. Our experience writing over 5000 lines of debugging programs shows exactly the opposite.

Debugging vs Monitoring

Opium is very well suited for debugging where programmers need to understand what is happening to their program. They, in general, specify one hypothesis about the execution at a time, and they follow it until they know enough about it. The repeat-fail mechanism of

Prolog is perfectly adapted to this search. The `f_get` primitive backtracks until either the hypothesis under consideration is confirmed or the execution is finished.

Monitoring is a slightly different topic. Programmers specify a priori a set of properties that the program must always verify, these properties are often called assertions. The verification must be done concurrently as interwined events can be involved in different assertions. The search must therefore be more sophisticated than a repeat-fail execution.

Two systems enable to state assertions on executions, the path rules of Kraut designed by Bruegge and Hibbard [6] and FORMAN [18]. Kraut can build an automaton to test a large set of conditions in a reasonably efficient way. This has not been done in Opium.

On the other hand, the previous two systems do not have the flexibility of Opium. They are more heavy than Opium for debugging, while Opium is more tricky to use for monitoring.

Declarative debugging

Opium is often opposed to declarative debugging because the full trace gives an operational view of an execution. Naish has defined a framework where, according to the bug symptom, a tree structure and erroneous nodes are defined, the declarative debugger is thus a dedicated tree traversal [29]. Such a scheme emphasizes that the proof tree is not always the appropriate data structure for the debugger to reason upon. Furthermore, the scheme can be implemented on top of Opium. Indeed the tree structures can be recollected from the low-level trace and dedicated tree traversal primitives can be implemented. Therefore Opium can be an implementation and prototyping suppport for declarative debugging.

12 CONCLUSION

In the first part of this article we have presented a trace query model in which three primitives provide execution traces as programming data; the latter can then be processed by Prolog programs. With some optimizations, which we described, this model can work on very large executions. The unique functionalities offered by our model are

- a concise and efficient trace query language,
- synchronous trace querying and trace database coexistence, and
- meta-debugging capabilities.

The trace query language shows that Prolog is a powerful basis for "on the fly" trace querying.

In the second part of this article, we have shown Opium capabilities to build abstract tracers and automated debugging facilities.

Acknowledgements

Opium has been designed and implemented while the author was at ECRC (European Computer-Industry Research Centre, Munich). Anna-Maria Emde engineered Opium into its pre-release stage and contributed many ideas. Joachim Schimpf helped to implement and maintain the system.

Jacques Noyé designed and implemented the profiling extension. Jean-Marc Andreoli helped to design the LO abstract tracer, which Norbert Eisinger customized further. Pascal Brisset designed and implemented the CHR abstract tracer.

Stéphane Schoenig, Jacques Noyé, Sarah Mallet, Yves Deville and Pascal Brisset gave fruitful comments on this and previous versions of (parts of) this article.

References

[1] J-M. Andreoli and R. Pareschi. Communication as fair distribution of knowledge. In *Proc. of OOPSLA'91*, Phoenix, U.S.A., 1991.

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

[3] J.D. Bovey, M.T. Russel, and O. Folkestadt. Direct manipulation tools for Unix workstations. In *Proceedings of the EUUG Autumn'88*, pages 311–319, October 1988.

[4] M. Brayshaw and M. Eisenstadt. Adding data and procedure abstraction to the Transparent Prolog Machine TPM. In R.A. Kowalski and K.A. Bowen, editors, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 532–547, Seattle, USA, August 1988. MIT Press. JICSLP'88.

[5] P. Brisset, T. Früwirth, P. Lim, M. Meier, T. Le Provost, J. Schimpf, and M. Wallace. Eclipse 3.4 - extensions user manual. Technical report, ECRC, January 1994.

[6] B. Bruegge and P. Hibbard. Generalized path expressions: A high-level debugging mechanism. *The Journal of Systems and Software*, 3:265–276, 1983. Elsevier.

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

[8] M. Carro, L. Gómez, and M. Hermenegildo. 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, 1993. MIT Press.

[9] J. Cohen and N. Carpenter. A language for inquiring about the run-time behaviour of programs. *Software-Practice and Experience*, 7:445–460, 1977.

[10] P. Dencker. *Debugger chapter of the Alsys Ada System User's Manual.* Alsys Gmbh, May 1991.

[11] M. Ducassé. Abstract views of Prolog executions in Opium. In V. Saraswat and K. Ueda, editors, *Proceedings of the International Logic Programming Symposium*, pages 18–32, San Diego, USA, October 1991. MIT Press. ILPS'91.

[12] M. Ducassé. Analysis of failing Prolog executions. In *Actes des Journées Francophones sur la Programmation Logique*, Mai 1992.

[13] M. Ducassé. *An extendable trace analyser to support automated debugging.* PhD thesis, University of Rennes I, France, June 1992. European Doctorate.

[14] M. Ducassé. A pragmatic survey of automated debugging. In P. Fritzson, editor, *Proceedings of the First Workshop on Automated and Algorithmic Debugging*, volume 749 of *Lecture Notes in Computer Sciences*, Linkoeping, Sweden, May 1993. Springer-Verlag. AADEBUG'93.

[15] M. Ducassé and A.-M. Emde. A high-level debugging environment for Prolog. Opium user's manual. Technical Report TR-LP-60, ECRC, May 1991.

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

[17] A.-M. Emde and M. Ducassé. 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, May 1990.

[18] P. Fritzson, M. Auguston, and N. Shahmehri. Using assertions in declarative and operational models for automated debugging. *Journal of Systems Software*, 25:223–239, 1994.

[19] T. Früwirth. Constraint simplification rules. Technical Report ECRC-92-18, ECRC, 1992.

[20] M. Golan and D.R. Hanson. DUEL- A very high-level debugging language. In *Proceedings of the Winter USENIX Technical Conference*, San Diego, January 1993. Also Research Report CS-TR-399-92, Princeton University.

[21] R.E. Griswold and M.T. Griswold. *The Icon programming language.* Prentice Hall, 1990. second edition.

[22] D.R. Hanson and J.L. Korn. A simple and extensible graphical debugger. In *Proceedings of the Winter USENIX Technical Conference*, pages 173–184, 1997.

[23] M.S. Johnson. Dispel: A run-time debugging language. *Computer languages*, 6:79–94, 1981.

[24] B. Lazzerini and L. Lopriore. Abstraction mechanisms for event control in program debugging. *IEEE Transactions on Software Engineering*, 15(7):890–901, July 1989.

[25] C.H. LeDoux. *A knowledge-based system for debugging concurrent software.* PhD thesis, University of California, Los Angeles, 1985.

[26] A. Malony, D. Hammerslag, and D. Jablonowski. Traceview: A trace visualization tool. *IEEE Software*, pages 19–28, September 1991.

[27] M. Meier, A. Aggoun, D. Chan, P. Dufresne, R. Enders, D. Henry de Villeneuve, A. Herold, P. Kay, B. Perez, E. van Rossum, and J. Schimpf. SEPIA - an extendible Prolog system. In *Proceedings of the IFIP '89*, 1989.

[28] M. Meier and J. Schimpf. Sepia system evaluation. Internal Report IR-LP-13-35, ECRC, August 1991.

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

[30] J. Noyé. An overview of the Knowledge Crunching Machine. In M. Abdelguerfi and S. Lavington, editors, *Emerging Trends in Database and Knowledge-base Machines.* IEEE Computer Society Press, 1994.

[31] R.A. Olsson, R.H. Crawford, and W.W. Ho. A dataflow approach to event-based debugging. *Software-Practice and Experience*, 21(2):209–229, February 1991. J. Wiley and Sons.

[32] M.L. Powell and M.A. Linton. A database model of debugging. In M.S. Johnson, editor, *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on high-level debugging*, pages 67–70. ACM, March 1983.

[33] S. Schoenig and M. Ducassé. A backward slicing algorithm for prolog. In R. Cousot and D.A. Schmidt, editors, *Static Analysis Symposium*, pages 317–331, Aachen, September 1996. Springer-Verlag, LNCS 1145.

[34] B. Shneiderman. *Software Psychology, Human Factors in Computer and Information Systems.* Computer Systems Series. Little, Brown and Company, Boston, Toronto, 1980.

[35] J.C. Sperandio. *L'Ergonomie du Travail Mental.* Collection de Psychologie Appliquée. Masson, Paris, 1984.

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

[37] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[38] P. Winterbottom. Acid: a debugger built from a language. In *Proceedings of the Winter USENIX Technical Conference*, pages 211–222, 1994.

A THE OPIUM PRIMITIVES

This appendix specifies in some detail the primitives of Opium and the interfaces between the different modules of the system which have been introduced in section 6. Let us recall them briefly.

The *Opium session* executes the Prolog trace queries, typed in by the user, until a primitive is encountered which needs to call the query handler. It resumes its execution when the answer comes back.

The *query handler* is the central module of Opium, it synchronizes all the modules. Furthermore, it implements the pre-filtering retrieval primitives, `f_get` and `b_get`. It stores event structures in the internal database when needed. It decides when to resume the traced execution or when to search in the database. It executes a Prolog goal in the context of the traced execution, using the traced program.

The *tracer* stops the traced execution whenever an event is reached. Furthermore, the tracer knows were to fetch the information in the traced context to recollect the values of the event attributes. Hence the `current_<attribute>` primitives are implemented in the tracer.

In the following we will list the Prolog primitives at user disposal. All the Opium extensions have been written on top of them. We then detail the interfaces between the query handler and all the other modules. In particular, we described the functionalities required from a tracer to implement Opium on top it.

A.1 The Prolog Primitives at User Disposal

This section is the basis of the user manual regarding the kernel of Opium. Users can load any Prolog program and enter any Prolog goal at the top-level of the tracing prolog session. The library of Prolog primitive predicates which is the basis of all Opium extensions is as follows.

All primitives have been introduced previously, except those related to spy points. Traditionally, a spy point is a "static" breakpoint set on a predicate. Several predicates can be spied at the same

time. The spy points are active until they are explicitly removed. A primitive, `leap`, enables users to stop the execution at events related to spied predicates. The `leap` primitive can be used with the same trace query model as the `get` primitives, as illustrated in the example session in section 7. Spy points and `leap` are present in usual prolog tracers.

The following primitives do not display anything, as explained in section 2 any display can be written with `current/7` and Prolog. Therefore the primitives are suffixed by `_np` as introduced in section 7. There exists a default primitive to display events called `print/0`. Any command which displays the retrieved event is implemented automatically by Opium as follows (see the Opium user's manual for details [15]):

```
<command>_np :- <command> , print.
```

A.1.1 To move in the trace history (whether stored or not)

f_get_np(Chrono, Call, Depth, Port, Pred, Arg, Clause) moves forwards through the execution, or the database, until the first event which matches the specified attribute values or the end of the trace history is encountered.

b_get_np(Chrono, Call, Depth, Port, Pred, Arg, Clause) moves backwards through the database until the first event which matches the specified attribute values or the beginning of the trace history is encountered.

For `f_get_np/7` and `b_get_np/7` if the argument corresponding to an attribute is:

- a variable, named or anonymous, pre-filtering does not take this attribute into account.
- a value, pre-filtering checks the attribute value of the current event against the required value. For control flow attributes and the clause attribute users can specify an exact value, a list of possible values, an interval, or a negated value. For the argument attribute, users must specify a value, possibly partially instantiated.

f_leap_np moves forwards through the execution, or the database, until the first event related to a spy point or the end of the trace history is encountered. The usual `leap` command is defined as `leap :- f_leap.`

b_leap_np moves backwards through the database until the first event related to a spy point or the beginning of the trace history is encountered.

The `f_get_np/7`, `b_get_np/7`, `f_leap_np/0` and `b_leap_np/0` primitives can backtrack and they fail when they reach the bounderies of the trace history (beginning or end of trace). Thus, the `repeat(Mark)` primitive, introduced in section 2, is hard-coded inside them.

spy(Predicate) sets a spy point to a predicate. Predicate must be a defined predicate.

nospy(Predicate) removes a spy point from a predicate. Predicate must be a defined predicate.

is_spied(Predicate) tells whether the predicate is a spy point. Predicate can either a defined predicate or a variable.

A.1.2 To retrieve the attribute values of the current event

curr_line(Chrono, Call, Depth, Port, Pred, Arg, Clause) retrieves and checks the value of the attributes of the current event. If the argument corresponding to an attribute is:

- the anonymous variable, nothing is done for this attribute.
- a named 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, `current_line` fails.

  In the actual Opium environment, besides `curr_line` there also exists seven primitives, one for each attribute.

A.1.3 To control the traced execution

abort_trace aborts the traced execution and, as a consequence, aborts the tracing session, too.

no_trace stops tracing. The traced execution is resumed, without extracting any further trace information.

A.1.4 To control the amount of data stored in the trace database

set_recording(Flag) sets the `recording` parameter `on` or `off`.
Recording can be set on in the middle of an execution.

set_recorded_attributes(Chrono, Call, Depth, Port, Pred, Arg, Clause) sets the `recorded_attributes/7` parameter. If an argument is `on`, the corresponding attribute is to be stored by default.

record_line stores the current event attributes according to the `recorded_attributes/7` parameter.

record_attribute(AttributeName) records the current value of an attribut if it is not already in the stored structure. If the structure is not present creates it first.

reset_recording empties the trace database.

A.1.5 To execute goals in the context of the traced execution

remote_call_once(Goal, Solution) (resp. **remote_call_all(Goal, SolutionList)**) returns the first (resp. all) solution(s) of Goal, which is executed using the traced program context.

A.2 Interface Between the Opium Process and the Query Handler

The query handler and the Opium session are in two different processes. They have to be synchronized before exchanging messages. Indeed, both processes contain a Prolog session which can be used standalone if nothing is traced.

Some primitives require that an execution is currently being traced, they have to be *synchronous* with an execution. Some of the primitives, however, can be executed even if there is no traced execution running. These primitives are called *asynchronous*.

In order to implement the user primitives, the Opium process simply checks that they are used consistently and calls the query handler. There is therefore almost a one to one matching beween the user primitives and the primitives sent to the query handler. In order to differentiate the latter from the former we call them *kernel* primitives and we prefix them by `k_`.

In the following we first detail the coprocessing mechanism. We then list the synchronous primitives and the asyncrhonous ones. In the following we will only detail the kernel primitives when their arguments are different from those of the corresponding user primitives.

A.2.1 The coprocessing primitives

signal_to_opium If a traced execution is started the query handler sends a signal to Opium to start a synchronous session.

signal_to_prolog If Opium needs to execute an asynchronous query and no synchronous session is active, it sends a signal to the query handler to start an asynchronous query execution.

write_to_opium(Atom) sends a Prolog atom from the traced Prolog session to the Opium session.

write_to_prolog(Atom) sends a Prolog atom from the Opium session to the traced Prolog session.

read_from_opium(Atom) receives a Prolog atom from the Opium session

read_from_prolog(Atom) receives a Prolog atom from the traced Prolog session.

When an asynchronous query has been executed by the query handler it returns the result and the asynchronous session is closed.

When a synchronous session has been started, it is "on" until Opium sends an `abort_trace` or `no_trace` query. Thus even if a traced execution is finished, users can still query the last event and the trace database if it is present.

Note that because the atoms are written and read by standard Prolog built-ins the two Prolog sessions must use the same Prolog syntax.

A.2.2 The synchronous kernel primitives

The synchronous kernel primitives that are similar to the corresponding user primitives are: **k_f_leap**, **k_b_leap**, **k_abort_trace**, **k_record_line**, **k_record_attribute(AttributeName)**, **k_reset_recording**.

The following ones either have different parameters or implement only partially the user primitive.

k_f_get(ChrStat, ChrVal, CallStat, CallVal, DepthStat, DepthVal, PortStat, PortVal, PredStat, PredVal, ArgStat, ArgVal, ClauseStat, ClauseVal) moves forwards through the execution, or the database, until the first event which matches the specified attribute values.

k_b_get(ChrStat, ChrVal, CallStat, CallVal, DepthStat, DepthVal, PortStat, PortVal, PredStat, PredVal, ArgStat, ArgVal, ClauseStat, ClauseVal) moves backwards through the database until the first event which matches the specified attribute values.

For each attribute of `f_get` and `b_get` there is a status and a value. The status is one of

- `nop`: do not take this attribute into account for pre-filtering.
- `exact`: the value is an exact atom
- `neg`: the value is the negation of an atom

- `list`: the value is a list of possible atoms
- `interval`: the value is an interval of integers

Sorting the different kinds of values is done at Prolog level, it eases the implementation of the C functions in the query handler.

For `k_f_get`, `k_b_get`, `k_f_leap`, and `k_b_leap`, if one event is found returns `SUCCESS` otherwise `FAILURE`.

k_curr_line(ChrStat, ChrVal, CallStat, CallVal, DepthStat, DepthVal, PortStat, PortVal, PredStat, PredVal, ArgStat, ArgVal, ClauseStat, ClauseVal) retrieves the value of the attributes of the current event according to its status. If the status corresponding to an attribute is:

- `nop`: nothing is done for this attribute.
- `do`: the current value of the attribute is retrieved and unified with the variable.

The verification of values with unification is, at present, done in the Opium session.

k_abort_trace aborts the traced execution. The end of the current Opium execution is done by the Opium process itself.

k_no_trace stops tracing. The traced execution is resumed, without extracting any further trace information. The end of the current Opium execution is done by the Opium process itself.

A.2.3 The asynchronous kernel primitives

The asynchronous primitives are **k_spy(Predicate)**, **k_nospy(Predicate)**, **k_is_spied(Predicate)**, **k_remote_call_once(Goal)**, **k_remote_call_all(Goal)**, **k_set_recording(Flag)**, **k_set_recorded_attributes(Chrono, Call, Depth, Port, Pred, Arg, Clause)**.

A.3 Interface Between the Query Handler and the Tracer

The query handler is a subroutine of the tracer. The tracer *calls* the query handler when an execution is started, and whenever an execution event is reached. A parameter is used to inform the query handler of the status of the execution:

new a traced execution is started.

normal the traced execution is processing normally.

finished the traced execution is finished.

aborted the traced execution has been aborted since the previous call to the query handler. Note that the traced execution cannot abort without notifying the query handler so that the latter can close the Opium session properly.

When the query handler wants to resume the traced execution, it simply returns to the tracer with a status parameter telling what the tracer should do next:

next stop at next reached event.

leap stop at next spy point.

abort abort traced execution and do not call the query handler until next toplevel traced execution.

notrace stop tracing but finish the (no longer) traced execution. Do not call the query handler until next toplevel traced execution.

The query handler calls a number of primitives, implemented in the tracer, which return the current values of the attributes: **t_curr_call**, **t_curr_port**, **t_curr_depth**, **t_curr_pred**, **t_curr_arg**, **t_curr_clause**. At present, the value of `chrono` is managed by the query handler module.

The spy point primitives are also implemented in the tracer and called by the query handler: **t_spy(Predicate)**, **t_nospy(Predicate)**, **t_is_spied(Predicate)**.

A.4 Interface Between the Query Handler and the Trace Database

In the current implementation, the structures are stored as a double linked list in main memory.

The events are stored in two parts: the control part and the rest. The control part is small and of fixed size. The data part and the source connection part can be large and are of unknown size. The event structure contains the actual values of the control attributes and pointers to the arguments and clause values which are stored in a private heap.

The query handler therefore implements a **store_event** primitive, which will take into account the `recorded` attributes. It also implements primitives to retrieve the attribute values from the database: **d_curr_chrono**, **d_curr_call**, **d_curr_port**, **d_curr_depth**, **d_curr_pred**, **d_curr_arg**, **d_curr_clause**.

B THE BUGGY NQUEENS PROGRAM

This is the simple "generate-and-test" version of the Nqueens taken from [36, 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:    queens(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).
```

C THE PUZZLE PROGRAM

```
/*  From Sterling and Shapiro, MIT Press, 2nd edition p259-260 */
/*
 *  General puzzle solver
 */
solve_puzzle(puzzle(Clues,Queries,Solution), Solution) :-
        solve(Clues),
```

```
        solve(Queries).

solve([Clue|Clues]) :-
        Clue,
        solve(Clues).
solve([]).

/*
 *  Description of puzzle data
 */
test_puzzle(Name, Solution) :-
        structure(Name, Structure),
        clues(Name, Structure, Clues),
        queries(Name, Structure, Queries, Solution),
        solve_puzzle(puzzle(Clues, Queries, Solution), Solution).

structure(test, [friend(N1,C1,S1), friend(N2,C2,S2), friend(N3,C3,S3)]).

clues(test, Friends, [(did_better(M1C1, M2C1, Friends),      % Clue 1
                       first_name(M1C1, michael),
                       sport(M1C1, basketball),
                       nationality(M2C1, american)),
                      (did_better(M1C2, M2C2, Friends),      % Clue 2
                       first_name(M1C2, simon),
                       nationality(M1C2, israeli),
                       sport(M2C2, tennis)),
                      (first(Friends, MC3),                  % Clue 3
                       sport(MC3, cricket))]).

queries(test, Friends, [member(Q1, Friends),
                        first_name(Q1, Name),
                        nationality(Q1, australian),         % Query 1
                        member(Q2, Friends),
                        first_name(Q2, richard),
                        sport(Q2, Sport)],                   % Query 2
        [['The Australian is ', Name], ['Richard plays ', Sport]]).

did_better(A, B, [A,B,C]). first_name(friend(A,B,C), A).
did_better(A, C, [A,B,C]). nationality(friend(A,B,C), B).
did_better(B, C, [A,B,C]). sport(friend(A,B,C), C).

first([X|Xs], X).
```

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