Building efficient tools to query execution traces

Mireille Ducassé, Benjamin Sigonneau

HAL Id: inria-00070720

https://hal.inria.fr/inria-00070720

Submitted on 19 May 2006

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Building efficient tools to query execution traces

Mireille Ducassé , Benjamin Sigonneau

N˚5280

Juillet 2004

Systèmes symboliques

*R*apport
de recherche

Building efficient tools to query execution traces

Mireille Ducassé* , Benjamin Sigonneau†

Systèmes symboliques
Projet Lande

Abstract: Understanding how a program execution proceeds often helps debug the program. An execution can be seen as a succession of computation steps. Tracers give information about these steps and what occurs at each of them. The traceable steps are traditionally called *breakpoints*. An execution frequently produces several millions of breakpoint occurrences. Programmers can therefore not analyze by hand all the computation steps. Existing tracers often have *conditional breakpoints* to inspect only steps which satisfy certain conditions. Unfortunately, the conditions that can be specified do not meet all the needs. To palliate this problem, an execution trace can be seen as a relational database. Each breakpoint information is represented by a tuple and queries simply use the language of the database management system. The problem is then that the time to create the database is much too long. We have, therefore, proposed an improved framework for trace querying. The interrogation is processed in two distinct steps: firstly, the trace is filtered *on the fly* with respect to the basic conditions of the queries; secondly, the remaining part of the query is then processed. Our approach has two main advantages. Firstly, filtering is very efficient. Done on the fly, it creates very few new data structures, the trace does not even have to be stored at all. In this report, we describe how to implement trace query tools based on the above framework. We propose an architecture where a tracer driver, containing a very efficient trace filtering algorithm, is integrated in the tracer process. The implementation guidelines are based on our experience building four prototypes for different sorts of programming languages.

Key-words: Debugging, various programming languages, execution trace analysis, efficient trace filtering, trace querying, implementation guide

(Résumé : tsvp)

* IRISA/INSA ; email : Mireille.Ducasse@irisa.fr
† IRISA/IFSIC

Construction d'outils efficaces pour interroger les traces d'exécution

Résumé :   Comprendre comment une exécution de programme procède aide souvent à mettre au point le programme. Une exécution peut être vue comme succession d'étapes de calcul. Les traceurs fournissent des informations sur ces étapes et ce qui se produit à chacune d'elles. Les étapes traçables s'appellent traditionnellement *points d'arrêt*. Une exécution produit fréquemment plusieurs millions d'occurrences de points d'arrêt. Les programmeurs ne peuvent donc pas analyser à la main toutes les étapes de calcul. Les traceurs existants ont souvent *des points d'arrêt conditionnels* qui permettent de n'inspecter que les étapes qui satisfont certaines conditions. Malheureusement, les conditions qui peuvent être posées ne satisfont pas tous les besoins. Pour pallier ce problème, une trace d'exécution peut être vue comme une base de données relationnelle. Chaque point d'arrêt est un tuple et les requête utilisent simplement le langage du système de gestion de la base de données. Le problème est alors que le temps nécessaire à la création de la base de données est beaucoup trop long. Nous avons donc proposé un cadre amélioré pour la requête de trace. L'interrogation est traitée en deux étapes distinctes: premièrement, la trace est filtrée *à la volée* relativement aux conditions de base des requêtes; deuxièmement, la partie restante de la requête est alors traitée. Notre approche a deux avantages principaux. Premièrement, le filtrage est très efficace. Effectué à la volée, il crée très peu de nouvelles structures de données, la trace n'a même pas besoin être stockée. Dans ce rapport, nous décrivons comment mettre en œuvre des outils de requête de trace basés sur le cadre ci-dessus. Nous proposons une architecture où un pilote de traceur, contenant un algorithme de filtrage de trace très efficace, est intégré dans le processus de traceur. Les consignes d'implémentation sont basées sur notre expérience acquise lors de la construction de quatre prototypes pour différents types de langages de programmation.

Mots-clé :   Débogage, langages de programmation variés, analyse de traces d'exécution, filtrage de traces efficace, requête sur les traces, guide d'implémentation

Contents

1   Introduction

The result of a program execution is often surprising. This result may differ much from what was expected. Possible causes are either that this result is incorrect ; or that the programmer had badly understood the functional specification of the programme ; or that certain subtleties of the language escape him. It may also be that the program is completely or partially foreign, for example people must maintain a program which they did not write.

Understanding how the execution proceeds often helps solve the underlying problems. In general, a program produces some output but those "natural" manifestations of the program execution are, however, seldom sufficient to have a precise idea of what went wrong and why. The most extreme situation is an execution which fails or crashes without giving any result. Dedicated tools are therefore required to observe program behaviors.

Existing debugging tools   A program execution can be seen as a succession of steps of computation. Tools, called tracers or debuggers, make it possible to know what these steps are and what occurs at each one of them. The tracable steps are traditionally called *breakpoints*. The nature of possible breakpoints varies from one language to another. The number of their occurrences varies from an execution to another. This number frequently reaches several millions for an execution of less than a second. It is, thus, out of question that a programmer interested in execution details analyzes by hand all the computation steps.

Existing operational tracers often have a mechanism called *conditional breakpoints* which makes it possible to inspect only steps which satisfy certain conditions (see for example GDB [20] or UPS [2]). It should be noted that, unfortunately, the conditions that one can specify are far from meeting all the needs. It is even often difficult to know which language is to be used in order to specify the conditions and on what data the condition can be set. It seems that the design of these tracers has been made "bottom-up", guided by implementation constraints. The result is that they are efficient but incomplete.

A relational database approach   A way to improve the situation is to specify which computation steps correspond to which breakpoints, and to associate attributes to these breakpoints. A breakpoint with attributes is called an *event*. An execution trace can then be regarded as a sequence of events.

As an event is, in fact, a tuple, research prototypes consider the trace as an actual relational database and provide the mechanisms of the data base management system to query the trace (see for example Hy$^+$ [5]). This solution has two significant advantages: the trace is well modelled and the interrogation language is very powerful. These prototypes are, however, also very inefficient. Indeed, the database must be built before the programmer can enter the least request. The management systems of relational databases are conceived for persistent databases. An entered data has a long lifespan and it will be retrieved many times. The database management systems optimize therefore data recovery at the price of an expensive creation. That is perfectly justified in the case of a persistent database. It does,

however, not suit to debugging where the data, i.e. the trace events, have a short lifespan. After each modification of the program the previous traces become obsolete. Moreover, an execution of less than a second can generate several millions of tuples. Simply creating these tuples in a database (even in main memory) requires several minutes. That introduces too much delay into the "edition-test-debugging" cycle of the program development. It is only for a very serious problem, and when all the other possibilities have been exhausted, that a programmer will be ready to wait for the effective creation of an actual relational trace database.

Our approach: a "good" compromise between efficiency and expressive power
We propose a framework for trace querying which is also based on events and trace interrogation. This interrogation, however, is processed in two distinct steps: firstly, the trace is filtered *on the fly* with respect to the basic conditions of the queries; secondly, the remaining part of the query in then processed. For this purpose, a tracer driver containing a filtering mechanism is integrated into the tracer. The sophisticated analysis takes place in a distinct process. It contains an interpreter of a high level programming language. The tracer driver receives basic filtering queries from the analysis process and sends back filtered information to it. The trace does not need to be stored to be filtered. Our approach has two main advantages:

- filtering is very efficient. It is done on the fly, creating very few new data structures. Moreover, it is carried out in the tracer process, without any execution context switches.

- sophisticated requests can be programmed in the programming language of the analysis module.

Therefore, simple requests are processed in a very efficient way. Certain sophisticated analyses can take some time, but one can conjecture, firstly, that the programmer would spend even more time to find the relevant information by hand and, secondly, that he will be ready to pay the price for these requests as long as the simple ones do not cost him much.

Existing prototypes We have built four independent prototypes based on these principles: Opium [9, 8] for Eclipse Prolog [1], Morphine [10] for Mercury, a functional logic programming language [19], Coca[7] for C and Mescaline [12, 13] for constraint logic programming with finite domains. These four prototypes showed that the filtering mechanism implemented in the tracer by procedure call is efficient enough [14] and that powerful analyses could be programmed in the dedicated process [6, 11]. The programming paradigms are very diverse, the approach does not depend on the execution model of the traced language. In order to implement it, it is enough to have a tracer able to give tuples of information at each beakpont. In this tracer one can insert calls to the tracer driver.

In the four cases, the code of the tracer driver and the code of the core of the trace analysis module together are a couple of thousand lines only. The integration of the tracer

driver module in the tracer is at the same time a repetitive and a different process for each prototype. On the one hand, for example, the communication between the tracer and analysis processes, which is carried out by "sockets", can be shared. In the same way, most of the core of the analysis module does not depend on the traced language and thus does not need to be re-implemented. On the other hand, even if the filtering algorithm is basically always the same, it must be specialized each time for the specific trace model of the tracer. It must especially be implemented in the implementation language of the tracer. Indeed, filtering is called at each event and it can have to consider millions of events before a matching one is found. Experience shows that the cost of the filtering mechanism cannot be higher than the cost of a call to a simple procedure [14]. It is thus excluded to have costly interfaces between different languages. Moreover, each tracer has its characteristics, some have significant consequences. For example, if the tracer can identify the end of the trace at the very moment of the last event (and not after the execution is finished), the analysis in progress can be finished in a much cleaner way.

Towards LSD, a multi-language debugging environment   In order to facilitate the connection of trace analyzers to new tracers, we now provide a core of analyzer with inter-process communication procedures. Thus, instead of being implemented from scratch, the new trace analyzers are generated starting from a common program. We call LSD_t, an instance of trace analyzer for tracer t.

We are currently re-implementating three of the existing prototypes in this new framwork : LSD_{coca}, LSD_{opium} and $LSD_{mescaline}$. The fourth prototype, Morphine comprises a functionality richer than what is described above. Indeed, the module of filtering integrated into the tracer is also able to collect information. This last mechanism is currently very dependent on the traced language. It will be integrated in LSD in a second time.

The objective of the current work on LSD is to manufacture new instances of trace analyzers for other programming languages and other tracers[1]. These instances will cohabit together in the same analysis process. This report is an implementation guide for new LSD_t instances. A longer-term objective will be to automate as much as possible the specialization of the filtering algorithm and of the tracer driver. At present, this specialization must be made mostly by hand.

Organization of the report   In the following, Section 2 shows an example of debugging session for a C program with LSD_{coca}. Section 3 details the principles of filtering. Section 4 describes the architecture of a LSD_t instance, by giving some details on the existing instances. Section 5 describes the models of trace of the three tracers used in the three instances. Section 6 sketches the preliminary work done towards a multilanguage LSD environment. Appendix A gives the complete grammar of the events of the Coca tracer.

---

[1]Preliminary work on Java show that it should be feasible for object-oriented languages [18]

```
int legal(int board[4], int lineNb, int colNb)
{
  int i,ok;

  for (i=0,ok=1; (i<colNb) && ok; i++) {
    if ((board[i]==lineNb) ||
        (abs(board[i]-lineNb)==abs(i-colNb))) {
      ok=0;
    }
  }
  return ok;
}
```

Figure 1: Source code of a C function, called "legal". It tests a newly added queen on a board and contains bugs

2   An example of debugging session for C

We illustrate in this section the functionalities of LSD_{coca}, one of our prototypes, through an example of debugging session. We give intuitive explanations. Thorough descriptions of primitives and events can be found respectively in section 3 and section 5.

2.1   The debugged program

The debugged C program tries to solve the N-queens problem. 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. For N=4 a solution is, for example:

Let us assume that we have tested the program for N=4. We have noticed that a number of incorrect solutions have been produced and that the correct ones have not been produced. We suspect strongly the function `legal` (see source code in Figure 1) which tests that a newly added queen does not attack queens already on the board. Let us assume that we are, for example, debugging somebody else's program and that we are not familiar with the program. We can use LSD_{coca} to gain some understanding of the behavior of `legal`.

2.2   Conventions

In the following, [`lsd_coca`] is the prompt of the trace analyser. Everything between each occurrence of [`lsd_coca`] and the following "." is a debugging query, typed in by a user. A query is a sequence of predicates, interpreted by a specialized Prolog interpreter. Following the Prolog convention, identifiers beginning with an upper-case letter are logical variables unless they are surrounded by quotes. Identifiers beginning with a lower-case letter or surrounded by simple quotes are atoms. Identifiers surrounded by double quotes are strings. For example, in

$$\text{fget(func="legal" and chrono=Chr),}$$

`fget`, `func`, `and`, `chrono` are atom, `"legal"` is a string and `Chr` is a (free) variable. In Prolog, different predicates can have the same name when they have a different number of arguments (parameters). A predicate identifier is therefore a name followed by a number which gives the number of its arguments (called "arity"). For example, `display_board/2` denotes the `display_board` predicate with two arguments.

Whenever users answer `yes` to a `More?` prompt, further solutions are searched for. For example, if a query asks for an event with some characteristics and LSD$_{\text{Coca}}$ finds one such event, asking for more will prompt LSD$_{\text{Coca}}$ to search for the next such event. If a query asks for variable information, and LSD$_{\text{Coca}}$ finds an accurate visible variable, asking for more will prompt LSD$_{\text{Coca}}$ to search for the next accurate visible variable. When the search space is exhausted LSD$_{\text{Coca}}$ answers `no (more) solution`.

2.3   Commented session

Let us assume that we want to skip the execution until entering function `legal`. This can be expressed by the following query.

```
[lsd_coca] fget(func="legal" and type= function).

   (18) enter function legal
```

If we want to see which variables are visible at this point we can use the `current_data` primitive asking for names, values and types, using the logical variables `X`, `V` and `T` as follows.

```
[lsd_coca] current_data(name=X and val=V and type=T).


X = i
V = 0
T = int      More? yes

X = ok
V = 0
T = int      More? yes
```

```
X = colNb
V = 0
T = int      More? yes

X = lineNb
V = 0
T = int      More? yes

X = board
V = array(0, -1, -1, -1)
T = array(int,4)   More? yes

no (more) solution.
```

The C variables `i`, `ok`, `colNb` and `lineNb` are integer which can be easily interpreted. On the other hand, `board` is somehow cryptic. We can easily write a small Prolog predicate, `display_board/2`, to display a board graphically. This predicate can be connected to the trace **on the fly** with the `current_data` primitive. In the following query, we retrieve the current values of `board` and `lineNb`, and call `display_board/2` with the proper parameters.

```
[lsd_coca]  current_data(name=board and val=B),
            current_data(name=lineNb and val=L),
            display_board(B,L).
```



```
B = array(0, -1, -1, -1)
L = 0
```

We can foresee that the previous query will be very useful and make a new command out of it, called `display_board/0`. This is straightforward by using the `[user]` command of the Prolog interpreter which allows new predicates to be compiled online incrementally.

```
[lsd_coca] [user].
  display_board :-
       current_data(name=board and val=B),
       current_data(name=lineNb and val=L),
       display_board(B,L).
```

As we suspect `legal` to return wrong results we ask to see the next `exit` of the function where a board disposition was accepted. The following query reads as follows: "*I would*

like to see the next event of type `function`, *whose function is* `legal`, *whose port* [2] *is* `exit`, *and where variable* `ok` *has value* `1` *(ie the disposition is accepted). I would also like to get the chronological number of the event* [3] *and see the accepted board displayed."* There are several such events, typing `yes` asks for the following one. Subsequently, the execution goes on along with the trace.

```
[lsd_coca] fget(type=function and func="legal" and
                port=exit and chrono=Chr),
          current_data(name=ok and val=1),
          display_board.
```



```
Chr = 32     More? yes
```



```
Chr = 69     More? yes

[...] /* two solutions edited out by hand */

Chr = 143     More? yes
```



```
Chr = 648     More? yes
```

The first disposition has only one queen. It is acceptable. The second one has two queens in the same column, it should not be accepted. We have asked to see a number of solutions to the query until we are sure that there is also a problem with diagonals: queens in the same diagonal can also be accepted.

Now we want to check when `legal` rejects a disposition. We specify almost the same query as previously, except that this time we ask to see the `exit` of `legal` when the `val` of `ok` is `0` (ie the board is rejected).

---

[2] `port` indicates whether the execution is entering or exiting a construct.
[3] Note that logical variables (eg `Chr`) are used to get event attribute values while atoms (eg `exit`) set event attribute values.

```
[lsd_coca] fget(type=function and func="legal" and
               port=exit and chrono=Chr),
          current_data(name=ok and val=0),
          display_board.
```

```
Chr = 1674      More? yes
```

The last added queen is in the same diagonal as the first queen, hence this rejection is valid. This means that the function is not always wrong.

We want to check whether correct dispositions are accepted. We therefore ask to see the result of `legal` (ie the `val` of `ok` at the `exit`) on a disposition that we know is valid. *Note the use of "-" to tell that any value can occur.*

```
[lsd_coca] fget(type=function and func="legal" and
               port=exit and chrono=Chr),
          current_data(name=lineNb and val=1),
          current_data(name=board and val=array(0,2,-,-)),
          display_board,
          current_data(name=ok and val=OK).
```

```
Chr  = 3519
OK = 0      More? yes
```

The disposition is rejected. The test inside the `for` loop of `legal` seems lunatic. However, let us assume that, when analyzing its source code, we cannot understand where the problem is.

We can ask to see the value of the variables when the `if` test is performed. To get a precise idea, we ask to see them either when a new `for` loop is entered or when the `if` is exited. We ask to see the value of the `board` when entering a `for` and the values of the variables at each step[4]. In order to get a clear idea we first re-run and re-trace the execution from the beginning.

---

[4]The syntax of `if Cond then Inst1 else Inst2` in Prolog is `(Cond -> Inst1 ; Inst2)`

```
[lsd_coca] retrace.
[lsd_coca] fget(func="legal" and type in [for,if]),
        ( current(type=for and port=enter)
        -> display_board,
            current_data(name=lineNb and val=Lin),
            current_data(name=colNb and val=Col)
        ;   ( current(type=if  and port=exit)
            -> current_data(name=i and val=I),
                current_data(name=board(I) and val=BdI),
                current_data(name=ok and val=Ok)
            )).
```



```
Lin = 3
Col = 0     More? yes
```



```
Lin = 3
Col = 1     More? yes

I   = 0
BdI = 0
Ok  = 1     More? yes
```

The result of this query is indeed surprising. After the first board has been displayed there should have been at least one `if` test. The last positioned queen attacks the queen on the third line. Instead, a new board is displayed indicating that **no test was performed**. Could it be that the loop is iterating on `colNb` instead of `lineNb`? Returning to the source code we can then notice that this is indeed the case. Actually `colNb` and `lineNb` had been swapped in the entire function, they were not consistent with the way the board was coded.

This ends the debugging session.

3   Trace filtering

In the previous section we have introduced, through a debugging session, our scheme to filter and analyze execution traces. In this section we introduce systematically the primitives and query language which sustain this scheme. A thorough description of the LSD instances, including the actual architecture, can be found in section 4.

As mentioned in the introduction, a trace is a sequence of events ; an event is a tuple of couples "attribute_name, value". The precise events with their actual attributes are not needed to understand the principles of the scheme. Their description for the three current instances of LSD can be found in the next section. All the actual examples of this section are from $LSD_{mescaline}$. The queries are systematically paraphrased and should be understandable even without a deep knowledge of constraint logic programming.

We have already stated in the introduction that the analysis is done in two stages, first simple filtering then the rest of the analysis. During the filtering, the events are considered one at a time. We call *current event*, the particular event under scrutiny by the filtering algorithm at a given moment. If the current event matches the requested filter it is forwarded to the analysis module. If the current event does not match the filter the focus is moved to the next event in the trace, which then becomes the next current event. The analysis module is basically an interpreter of a high level language. We propose to use Prolog whose well known prototyping capabilities fits many of our needs.

In the following we describe the most important primitives of our scheme, first the filtering ones, then the ones to retrieve the attributes of the current event. We then show that both kinds of primitives can be used for generalized filtering. We describe how queries are processed. Finally, we argue that our approach is a good compromise between efficient and expressive power.

3.1   Event filtering

The main primitive is *fget* which moves forward in the sequence of events, it also filters the events according to a pattern.

fget(EventPattern) moves forward in the trace sequence until either the end of the trace is reached or an event matches the given pattern (EventPattern). In the first case fget fails, in the second case it succeeds and the current event is the matching event.

The primitive is resatisfiable. If one event is found, on backtracking fget(EventPattern) retrieves the next event which matches the EventPattern.

EventPattern is a conjunction of conditions. Each condition is of the form
Attribute_name Operator Attribute_value, where the operator is one of the following: = (unification) <> (different) > (greater than) >= (greater or equal than) < (less than) <= (less or equal than) in (an attribute value is a member of a set) not_in (an attribute value is not a member of a set) contains (an attribute value contains a subset.)

```
:- fget(depth = 2).
:- fget(port=reduce and updated_var='X'
                    and update=[emptied]).
:- fget(port in [select, reduce]).
```

Figure 2: Examples of queries with primitive `fget`

Some of the operators apply, of course, only when the type of the attribute allows it.

Figure 2 shows four examples of queries. The first query stops at the next event whose depth is 2. The second query stops at the next `reduce` event where variable 'X' has just been emptied. The third query stops at the next event whose port is either `select` or `reduce`.

Attributes in the EventPattern   Ideally all the event attributes should be usable in the `EventPattern`. However, some of the attributes are of unknown length or of high complexity, for example the store of constraints in CLP, the set of global variables in C, the list of ancestors in Prolog. It is therefore difficult to fully specify them in an `EventPattern` of a `fget` query. Furthermore, their matching would complicate significantly the filtering algorithm whose aim is to be as fast as possible (see discussion of Section 3.5).

We therefore propose a compromise: some attributes cannot be used in the `EventPattern`, but they can still be retrieved. We show in section 3.3 that this compromise does not reduce the expressive power of the queries.

3.2   Current event

All the attributes of the current event can be retrieved. Namely, the attributes of `EventPattern` can be retrieved with the `current` primitive. The other attributes with a second primitive, `current_data`. The attributes which are not allowed in `EventPattern` are called *compound-attributes*, they basically represent sets of items, each item is a tuple of fields. One such compound attribute is the set of global variables (actually present in the three LSD instances, even in Eclipse Prolog). Each variable has several fields, for example, its name, its value, and possibly its file declaration and its type. For compound attributes, retrieving the whole set in one go would be much too costly. We therefore propose `current_data`, which retrieves compound-attributes item by item.

`current(EventPattern)`: verifies that the attributes of the current event match the `EventPattern`. The `EventPattern` is of the same form as for the `fget` primitive.

`current_data(DataPattern)`: verifies that there exists at least one item of the compound-attributes of the current event which matches the `DataPattern`. Like `fget`, `current_data` is resatisfiable.

```
:- current(chrono=Ch and depth=4 and port=Po and contrainst=Co).
:- current_data(type=var and ident=19 and domain=D and name=N).
:- current_data(type=const and ident=Id and repr=diffN(_,_,_)).
```

Figure 3: Examples of queries with primitives `current` and `current_data`

```
:- fget(port = tell and constraint = (_,_,diffN(X,Y,N),_)),
     current_data(type = var  and domain = [1,2]).


:- fget(port in [reject, solution]),
   ( current_data(type = var and ident = N and domain = D),
     printf("%w :: %w \n", [N,D]),  fail
   ; printf("no more variable \n", []) ).
```

Figure 4: Examples of queries with primitives `fget` and `current_data`

`DataPattern` is a conjunction of conditions. Each condition is of the form
`field_name = field_value`, where the operator = represents the unification.

For both primitives `current` et `current_data`, the attribute values which are specified as Prolog variables in the pattern are unified with their actual current value.

Figure 3 shows three examples of queries with primitives `current` and `current_data`. The first query checks that the current depth is indeed 4 and, if it is the case, respectively unifies variables `Ch`, `Po` and `Co` with the value of attributes `chrono`, `port` and `constraint`. The second query retrieves the name and the domain state of the 19th variable. The third query retrieves one of the identifiers of a `diffN` constraint if there exists one in the store.

3.3   Generalized filtering

We have described earlier that the compound-attributes of an event cannot be used in the filtering primitive. It does not mean that the search cannot use them. Indeed, thanks to the backtracking mechanism of Prolog, the `current_data` primitive can be used to refine the search of the filtering primitive. This is illustrated in Figure 4. The first query stops at the next event corresponding to a `tell` of a `diffN` constraint where there exists a variable with a domain equal to `[1,2]`. When `fget` finds an event corresponding to a `tell` of a `diffN` constraint, `current_data` is called. For every variable, it checks in turn whether its value has a domain equal to `[1,2]`, until the first variable which satisfies the request or there is no more variable to check. If no variable satisfies the request, the query backtracks to `fget` which tries to find a further event matching the filter. If it succeeds `current_data` is called again. And so on until either the end of the execution is reached or an event matching both the filters of `fget` and `current_data` is found.
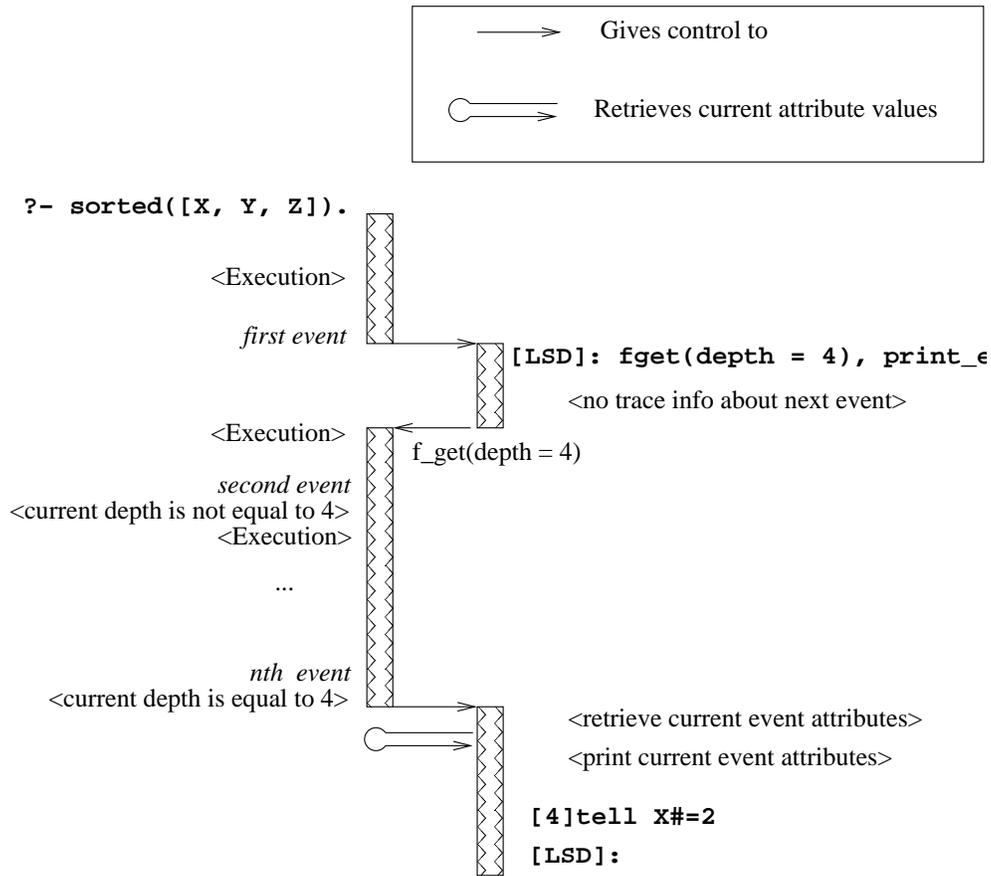
Figure 5: Illustration of the processing of a filtering query

Any Prolog construction and mechanism can be used in the trace queries. For example, the second query of Figure 4 stops at the next `solution` or `reject` event. Then a "repeat-fail" construction is used to retrieve in turn each variable and its domain in order to display them. When no more variable is accessible the query says so. Note that the information about variables and domains could be displayed in a more sophisticated way.

3.4 Processing of the event filtering

Events are filtered as the traced program is executed. There are two execution processes, one for the traced execution, and one for the LSD trace analyser. When a traced execution is started, LSD is notified when the first event takes place, the traced execution then waits.

LSD executes trace queries locally until a `fget/1` primitive is encountered. Then the traced execution is resumed and LSD waits. When the requested event is reached, the traced execution stops and LSD takes back the hand until a new `fget/1` primitive is encountered, and so on.

Figure 5 illustrates how the filtering primitive works. Let us assume that the programmer wants to query the execution trace of the CLP program given in Figure 22, page 44. When the execution reaches the first event, it notifies LSD which prompts the programmer for a trace query. The programmer enters a goal in order to search forward until an event at depth 4 is found (`fget(depth=4)`). This event should then be displayed (`print_ev`). At that moment, LSD 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 locally checks whether the current depth is equal to 4. Note that there is no need to generate the current depth as it is already present in the context of the traced process. As the current depth is not the requested one, the traced execution is resumed until the next event is reached. The depth is again locally checked. Forward moves and checking are done in turn until the first event whose depth is 4 is reached. LSD is notified and proceeds. The current event attributes are 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.

3.5   A good compromise between efficiency and expressive power

The scheme previously described is a good compromise between expressive power and efficiency. Indeed, the filtering is done in the traced process. This has two main advantages. Firstly, the attributes do not need to be "created", they are present in the tracer and they only need to be tested. A creation or copy is several orders of magnitude more expensive than a test. Secondly, only two context switches are needed for a `fget`, one to switch from the analysis session to the traced session, and one to switch back to the analysis session once a matching event is found. Context switches can be very costly, avoiding them is a good policy. As a consequence a query which uses only a `fget` will be very efficient.

However, `fget` alone does not provide the whole needed power. As illustrated above, generalized filtering requires to use `current_data`. As a consequence, more context switches will occur and possibly large chunks of information will have to be copied. Note, nevertheless, that `current_data` is designed in such a way that the information can be retrieved item by item. Hence only the useful parts of the compound-attributes will be retrieved. Future work should study the possibility to integrate data filtering in the tracer process.

Thus, filtering is very efficient and the cost of sophisticated search depends of the amount of compound-attributes that has to be checked and of the number of events where the checking has to be done. If most of the search can be done by `fget`, sophisticated queries can still be very efficient. It should be noted that however long a sophisticated search takes, it will always be much faster than the equivalent search done by hand on standard tracers. The scheme is therefore already usable.
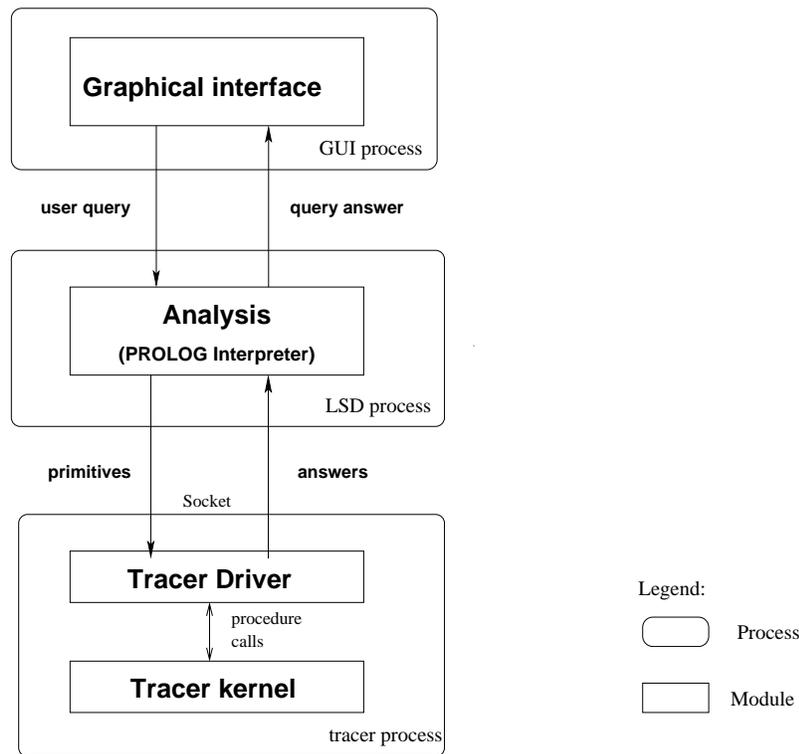
Figure 6: Architecture of an LSD instance

4 Implementation of LSD instances

Section 3 describes the principles of LSD. In this section we present implementation guide lines, illustrated by the current LSD instances.

All instances of LSD have the same architecture. As illustrated by Figure 6 there is a graphical interface to help users enter trace queries. Users queries and trace analysis programs are executed by a Prolog interpreter. Whenever a primitive (e.g. `fget`, `current` or `current_data`) is encountered, it is sent to the tracer driver which communicates with the tracer to get trace information. When the filtering algorithm of the tracer driver has found the relevant information, it sends the answer back to the Prolog interpreter. When the Prolog interpreter has completed the user query processing it sends the answer back to the user interface.

The overal software runs in three processes. One for the graphical interface, called the *GUI process*[5], one for the Prolog interpreter, called the *analysis* or the *LSD* process, and

---

[5]Not detailed in this report.

one for the tracer called the *tracer process*. The filtering module has been inserted in the tracer process. The interprocess communications run through sockets[6]. Inside processes, communications are made via simple procedure calls.

In an `fget` query execution, there is a critical path which should be optimized as much as possible. Indeed, several millions of events may have to be filtered before one fullfils the event pattern specified in the query. Therefore, any unnecessary overhead in the instructions executed each time a new event is encountered by the tracer can result in unbearable slowdown.

The filtering mechanism of the tracer driver is integrated inside the tracer process in order to avoid major overheads due to context switches and data copies. In order to optimize the critical path, further design guidelines must be followed when implementing the filtering mechanism. For example, the simple procedure calls between the filtering procedure and the tracer are essential to the good performances of the overal mechanism.

Outside the critical path, the implementation of the basic mechanisms does not have to be much optimized. For example, the performance of the communication between the GUI process and the LSD process is not critical. A query is entered from time to time, the communication time is neglictable compared to the time taken by the users to enter the queries. Similarly, the communication between the LSD process and the tracer process is not too critical because the integration of the filtering procedure inside the tracer process drastically reduces the amount of communications between the two processes. Programmers of analyses should, however, pay attention to the complexity of each analysis, but this is outside the focus of the current report.

In the following we focus on the filtering mechanism because it is essential in the critical path. A description of the other aspects of the tracer driver can be found in [17]. We first describe in some details the execution of the three filtering primitives `current`, `fget` and `current_data`. In particular, for each primitive, we specify the interactions between the LSD process and the filtering module, as well as between the filtering module and the tracer. We present the filtering algorithm and how to specialize it to a given tracer in order to optimize the critical path.

The examples of this section use a fake trace model with 5 attributes for the events and 6 fields for the data. This model could be a simplified possibility for an imperative language like C. The event attributes are chrono (the rank of the event in the trace), port (the type of event), func (the function name), file (the definition file of the function), and depth (the depth in the call stack). The data fields are kind (kind of data, for example variable), name, type, line, file, value.

4.1   The filtering primitives

In this section, we first briefly describe the consistency checking which must be done in the LSD process for all the primitives. We then refine the description of the processing of the three primitives `current`, `fget`, and `current_data`.

---

[6]The current instances are all implemented on Solaris/Unix.

User query :	`current(port=enter and chrono=C)`
Internal form :	`current_l2f([wanted,wanted,nop,nop,nop])`

Figure 7: Example of a `current` query and its internal form

Each primitive has a *user form* and an *internal form*. The user form is actually the Prolog predicate as it is typed by the users. This predicate is verified by the LSD Prolog interpreter. When the verification succeeds an internal form is generated. This internal form is sent to the filtering module. The convention is to add `_l2f` (which means LSD to filtering) to the user predicate name, and the arguments are put into a format more directly useable by the filtering.

4.1.1 Consistency checking of the patterns

Before being processed every user query is first verified. In Section 3 we have already presented that each of the three primitives work on a pattern, either an event pattern or a data pattern. This pattern must be well formed. First the type is checked. For example, `current(chrono=a)` is rejected in $LSD_{mescaline}$ because the `chrono` attribute is an integer (see Section 5.2). Then the consistency of the query is checked. For example, in $LSD_{eclipse}$ `current(port=call and det_exit=D)` is rejected because `det_exit` is specific to the `exit` port (see Section 5.1).

As much as possible of this verification must be achieved in the analysis module. Indeed, there is no need to filter millions of events, when it can be known immediatly that the request is inconsistent. Furthermore, these types of verification are very easy to program in Prolog. Last but not least, these verifications can be generic because the trace format has to be formally described. Therefore these verifications can be programmed once for all and used for all the instances.

4.1.2 The `current` primitive

The `current` primitive has the simplest execution schema because it does not involve any search. Let us recall that the `current` primitive retrieves the current value of the event attributes and checks that these values match the values that are specified in the pattern.

It is important to note that only the retrieval is done by the internal primitive `current_l2f`. The consistency checking of the query and the matching of the current values are done in the LSD process.

Internal form If the query is valid, it is first put into a internal form. Namely, the event pattern P is transformed into a list PL. If an event has n attributes, PL has n elements. For each attribute, if it is present in P, the element at its rank in PL is `wanted` otherwise it is `nop`. The primitive name is replaced by its internal form, namely `current_l2f`.

Figure 7 shows an example of `current` query, and its internal form. The query specifies that the port attribute must be `enter`. This is reflected by the second `wanted` of the list of

Filtering   LSD

current_l2f([wanted,wanted,
nop,nop,nop])

:− current(port=exit
and chrono=C).

[153,enter,nop,nop,nop]

test : exit = enter ?
no

No.

current_l2f([wanted,wanted,
nop,nop,nop])

:− current(port=enter
and chrono=C).

[153,enter,nop,nop,nop]

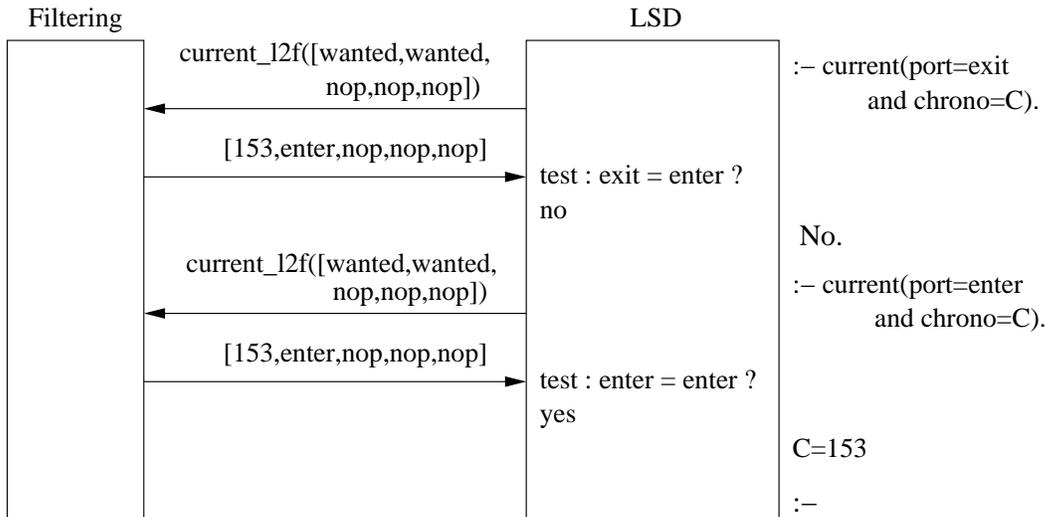test : enter = enter ?
yes

C=153

:−

Figure 8: Example of communication between LSD and the filtering for a `current` query

the internal form. The query also tells that the chronological number has to be retrieved. This is reflected by the first `wanted` of the list of the internal form. The three other attributes are not requested.

Interaction between LSD and the filtering module The query in its internal form is sent to the filtering module, as shown in Figure 8. The filtering module returns a Prolog list, `[153,enter,nop,nop,nop]`, which gives the value of `chrono` (`153`) and `port` (`enter`) which were wanted and `nop` for the other attributes. LSD then tests whether those values are consistent with the values that may have been given in the original query. In our example, the first query specifies that the port should be `exit`. This is different from `enter`, the query fails. The second query specifies that the port should be `enter`. The internal form of this query is the same as the internal form of the first query. The answer is also the same. LSD then tests whether the retrieved value of `port` (`enter`) is the same as the specified one (`enter`). It succeeds. The free variables of the pattern P are bound to the values of the corresponding attributes of the current event. Namely in the example, C is bound to 153.

Interaction between the filtering module and the tracer The communication between the filtering and the tracer simply consists of procedure calls. There is a procedure, for each attribute, that returns the value of this attribute, namely `event_attr_value_xxx()`, where `xxx` stands for the name of the concerned attribute. The filtering calls the procedures corresponding to the `wanted` attributes, and the returned values are used to construct the Prolog list which is then sent to LSD. This is shown in figure 9. The user query wanted the
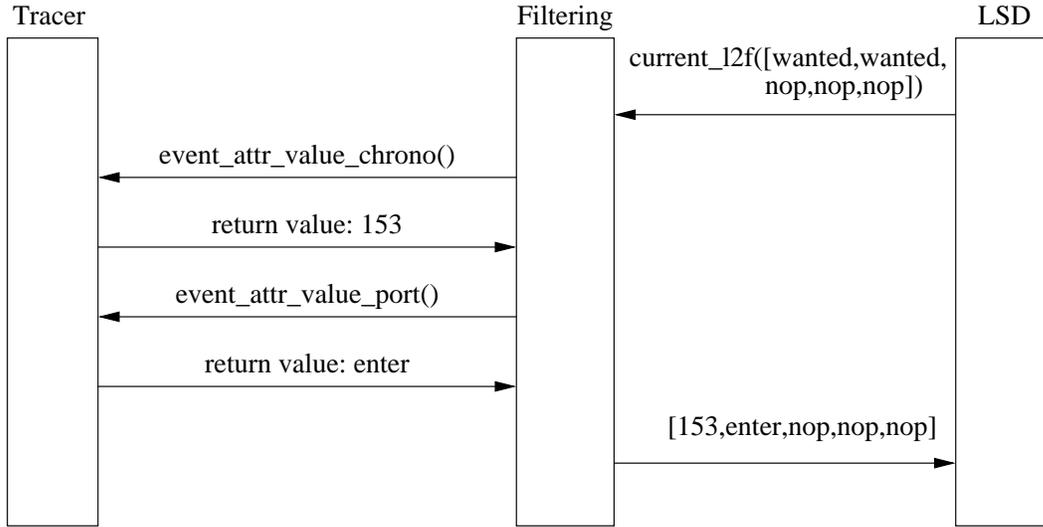
Figure 9: Example of communication between the filtering and the tracer for the `current` query of Figure 8

chrono and `port` attributes. The filtering module retrieves them from the tracer by calling the two dedicated procedures `event_attr_value_chrono` and `event_attr_value_port`. It then uses the returned value to build the list `[153,enter,nop,nop,nop]` which is sent back to the LSD module.

4.1.3 The `fget` primitive

The `fget` primitive searches forward in the sequence of execution events until one of them matches a specified event pattern. When one event is found the attributes which were `wanted` are retrieved.

Internal form An `fget` query with an event pattern P is put into a internal form in a similar way as a `current` query. This time PL consists of two Prolog lists L_1, to specify the matching pattern, and L_2, to specify the `wanted` attributes. If an event has n attributes attached to it, then both L_1 and L_2 have n elements. L_1 is computed in the following way: if an attribute is instantiated in P, then the corresponding element of L_1 is computed using the rules shown in the table of Figure 10, where *term* refers to the value of the instanciated attribute, otherwise, that element is `nop`. L_2 is computed in the same way as the pattern of `current`: if an attribute is present but not instantiated in P, then the corresponding element of L_2 is `wanted`, otherwise, that element is `nop`.

operator in P	operator in PL
=	exact(term)
<	less(term)
<=	less_eq(term)
>	gt(term)
>=	gt_eq(term)
<>	diff(term)
in	in(term)
not_in	not_in(term)

Figure 10: Rules to build the first list (the matching pattern) of `fget` internal form

```
User query :     fget(chrono > 101 and port in [enter,exit] and
                     func = F and depth = D and port = P)
Internal form :  fget_l2f([gt(1),in([enter,exit]),nop,nop,nop],
                     [nop,wanted,wanted,nop,wanted])
```

Figure 11: Example of an `fget` query and its internal form

Figure 11 shows an example of `fget` query, and its internal form. The query specifies that the chronological number of the event must be greater than 101 and that the `port` attribute must be either `enter` or `exit`. This is reflected in the first list of the internal form. The query also asks that the actual values of the function, the `depth` and the `port` have to be retrieved. This is reflected in the second list of the internal form. Note that, at matching events, the `port` can be either `enter` or `exit`. In order to actually know which one it is, the user has to ask for the `port` value.

Interaction between LSD and the filtering module   Figure 12 shows the communication between LSD and the filtering module during the execution of the previous query. The internal form is sent to the filtering module. The filtering module examins every trace event until an event satisfying the matching pattern is reached or the end of the execution is reached. If the end of the execution is reached the filtering module sends `end_of_trace` to LSD. If an event is found, the filtering module then sends to LSD a Prolog list whose elements are the values of the attributes that were marked as `wanted` in L_2, or `nop` otherwise. In the example, the port, the function and the execution depth were requested, the returned list is `[nop, enter,"legal",nop,8]`. In LSD, the free variables corresponding to the `wanted` attributes are bound to the returned values (`F=legal, D=8, P=enter`).

The user can decide to search the sequence of events for another event satisfying the same matching pattern. In that case, as illustrated in Figure 12 he simply answers yes (`y`) when LSD asks whether it should search for more solutions (`more ?`). LSD sends `continue_search_l2f` to the filtering which answers as described earlier. If the user an-
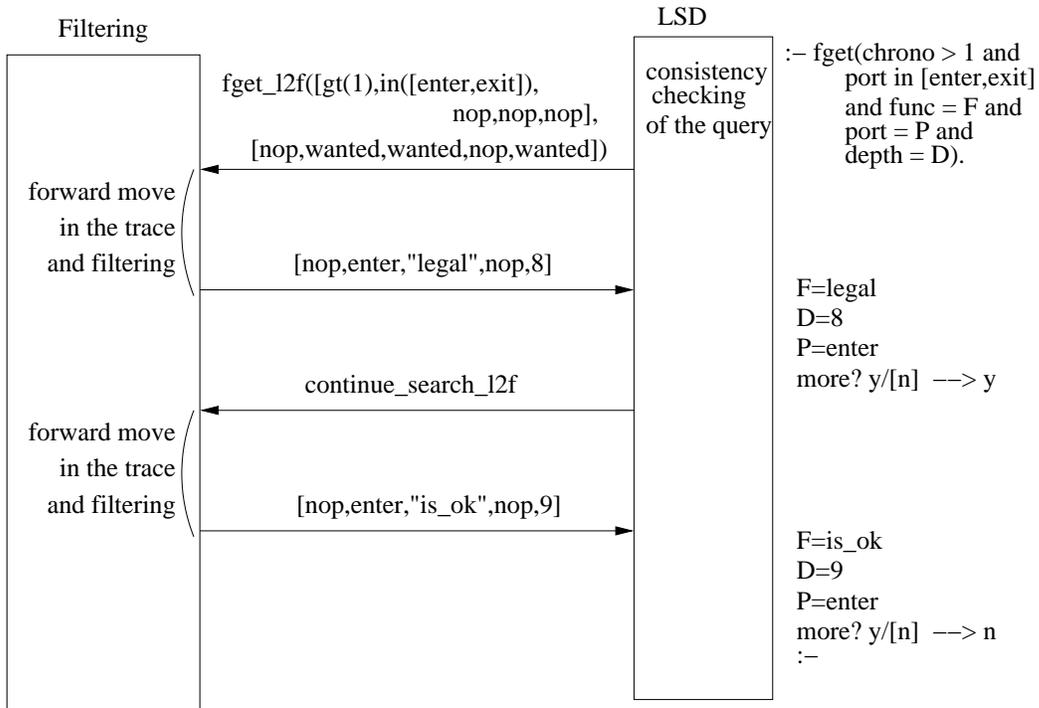
Filtering   LSD

fget_l2f([gt(1),in([enter,exit]),   consistency   :– fget(chrono > 1 and
                          nop,nop,nop],   checking   port in [enter,exit]
             [nop,wanted,wanted,nop,wanted])   of the query   and func = F and
                                                                               port = P and
                                                                               depth = D).
forward move
   in the trace
and filtering   [nop,enter,"legal",nop,8]   F=legal
                                                                         D=8
                                                                         P=enter
                                                                         more? y/[n] ––> y
                     continue_search_l2f
forward move
   in the trace
and filtering   [nop,enter,"is_ok",nop,9]   F=is_ok
                                                                         D=9
                                                                         P=enter
                                                                         more? y/[n] ––> n
                                                                         :–

Figure 12: Example of communication between LSD and the filtering for an `fget` query

legend ◄····· return from the filtering call ·····► call of the filtering from the tracer

◄——— P call of procedure P ———► V returned value V

fget_l2f([i instanciated
attributes (+nops)],
[j "wanted" attributes
(+ nops)])

Tracer Filtering LSD

next event
reached · · · · · · filtering · · · · · · ►

repeat (i times
at most)

◄——— event_attr_value_xxx

———► attribute value

Is value consistent
with pattern ?
yes   no

test upon the
instanciated attributes

repeat exactly j times

◄——— event_attr_value_xxx

———► attribute value

retrieval of the
"wanted" attributes

list of the values of
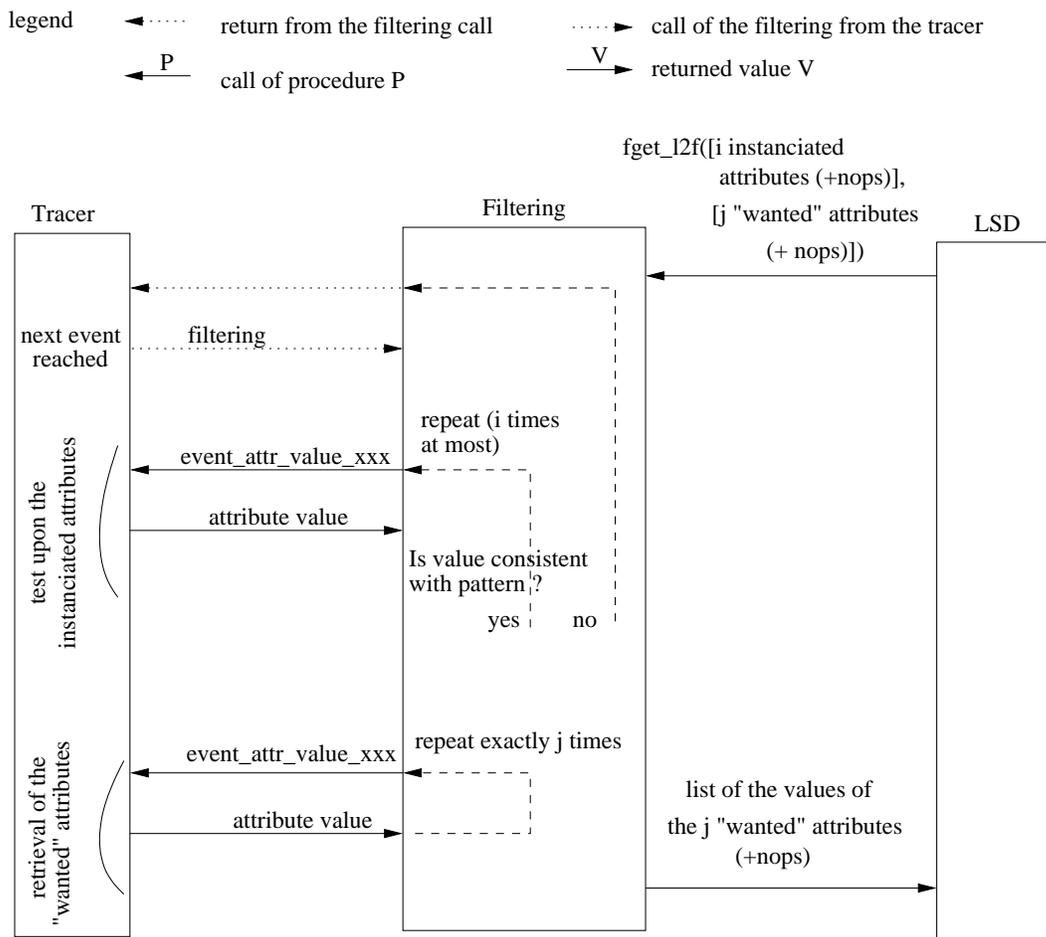the j "wanted" attributes
(+nops)

Figure 13: General scheme of communication between the filtering and the tracer for an `fget` query

swers no (`n`), the execution of the query is finished and LSD prints the prompt (`[lsd]`). In our example, the user asks for a second event and gets a new sets of values for the requested attributes (`F=is_ok, D=9, P=enter`). The user then decides that he does not want to see more events satisfying the pattern, and the execution of the query is finished.

Interaction between the filtering module and the tracer   Figure 13 shows the principle of the communication between the filtering and the tracer. Note that this is basically the critical path discussed in section 4.2.2. LSD sends an `fget_l2f` internal form with a list

L_1 containing i constrained attributes and a list L_2 containing j wanted attributes. When the filtering receives the query, it processes as follows.

1. The filtering procedure, which had been called by the tracer when the current event was reached, returns.

2. The tracer resumes the traced execution and stops it again at the very next breakpoint. There it calls the filtering procedure.

3. The filtering enters the "test loop", *ie* it retrieves one by one the values of the i attributes that are constrained in L_1 and confronts them to the matching pattern. The test loop can exit in three ways:

   - when the end of the execution is reached, the "test loop" fails and the filtering module returns `end_of_trace` to LSD.
   - when a retrieved attribute value does not match its associated constraint in L_1, the event does not match the pattern. The "test loop" fails and the filtering algorithm goes back to point 1,
   - when all of the i attributes are processed, the event does match the pattern. The "test loop" succeeds and the filtering algorithm goes to point 4;

   Note that, for a given event, the "test loop" makes at most i calls to the procedures which retrieve the attribute values;

4. When a "test loop" succeeds, the filtering enters the "retrieving loop". The values of the j attributes marked as wanted in L_2 are retrieved by calling the relevant j `event_attr_value_xxx()` functions. The "retrieving loop" always succeeds.

At this stage, the whole information needed to build the answer for LSD is known by the filtering procedure. This answer is built and then sent to LSD.

Using the same example as in Figure 12, Figure 14 shows the communication between the filtering and the tracer for an actual query.

4.1.4   The `current_data` primitive

The `current_data` primitive is at the same time similar and different from both the previous two primitives. Like `current` it retrieves current values, but this time it retrieves the values of data fields. Let us recall that the data addressed by `current_data` are compound information, for example a set of variables or a set of constraints or a set of ancestors. Therefore, and like `fget`, `current_data` searches for a data matching the given pattern. However, and unlike `fget`, `current_data` does not move forward in the traced execution. The search is done only on state of the data corresponding to the current event.

There is another significant difference which has a major impact on the design and implementation of the `current_data` primitive, namely it has to handle the values of the
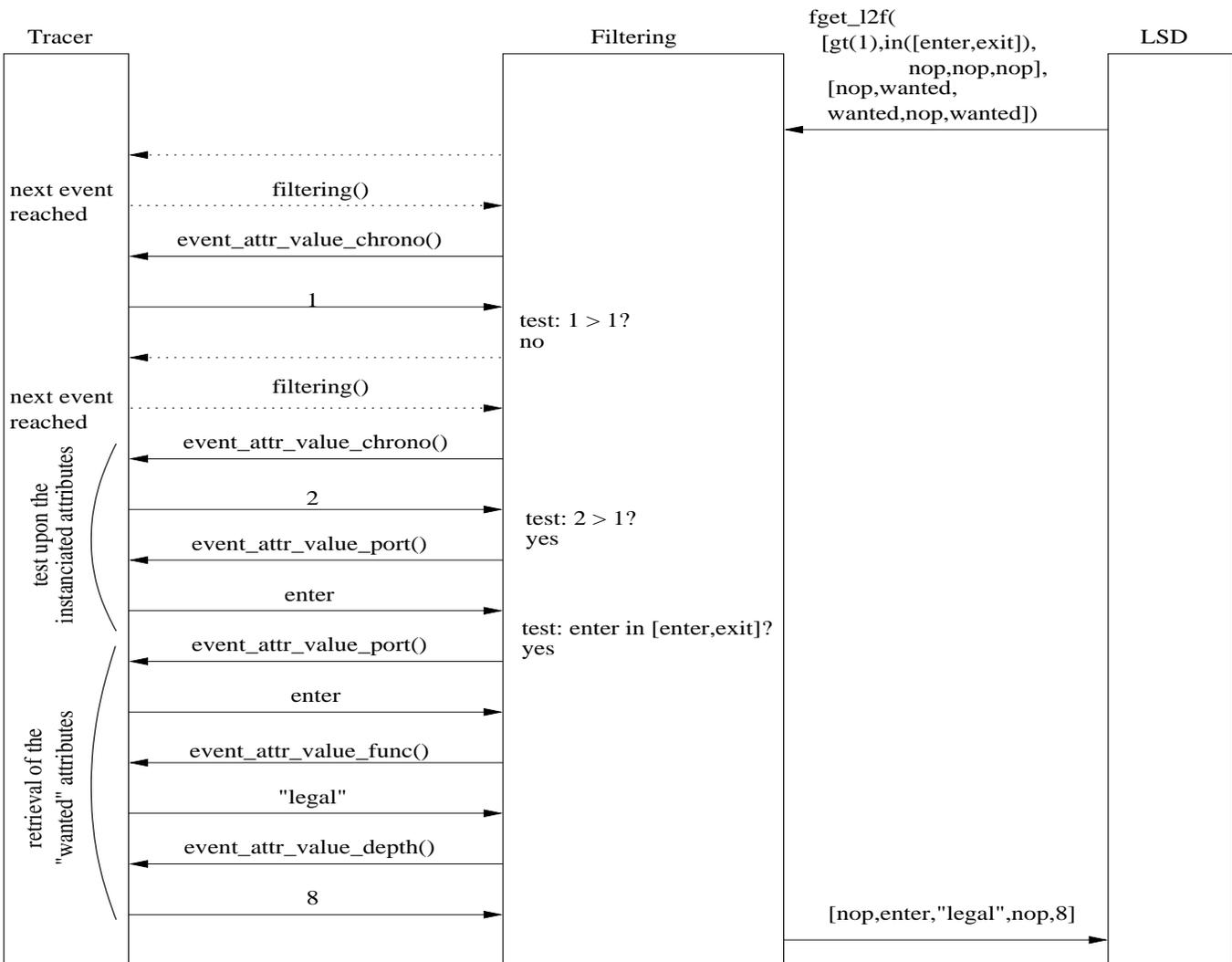
Figure 14: Communication between the filtering and the tracer for the fget query used in figure 12

Tracer

Filtering

LSD

fget_l2f(
[gt(1),in([enter,exit]),
nop,nop,nop],
[nop,wanted,
wanted,nop,wanted])

next event reached

filtering()

event_attr_value_chrono()

1

test: 1 > 1?
no

next event reached

filtering()

event_attr_value_chrono()

test upon the instanciated attributes

2

test: 2 > 1?
yes

event_attr_value_port()

enter

test: enter in [enter,exit]?
yes

event_attr_value_port()

enter

retrieval of the "wanted" attributes

event_attr_value_func()

"legal"

event_attr_value_depth()
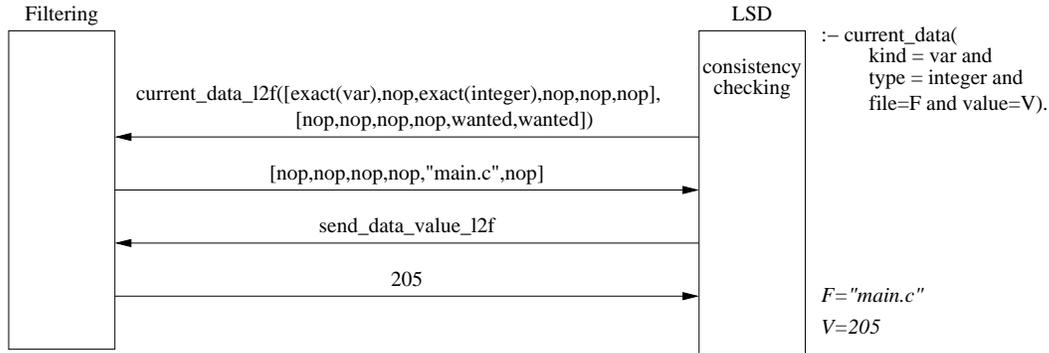
8

[nop,enter,"legal",nop,8]

Figure 15: Example of communication between LSD and the filtering for a `current_data` query (the value of the data is requested by the user query)

data. As opposed to the other fields and the attributes. All the possible types of the values are unknown at the time an LSD instance is generated. Indeed, the user may introduce new types. We will discuss the consequences of this problem in the following. Note that at present, the value is the only for which this problem has been identified. When designing a new LSD instance, it is important to check whether the problem could occur for other attributes and fields.

Internal form   A `current_data` query with an event pattern P is put into a internal form PL. PL consists of two Prolog lists L_1, to specify the matching pattern, and L_2, to specify the `wanted` fields. They are constructed using the same algorithm as for an `fget` query (see the description in the previous section).

Interaction between LSD and the filtering module   The internal form is sent to the filtering module. The filtering module then looks for a data that matches the instantiated part of the query (*ie* L_1). If such a data cannot be found, the filtering answers `no_more_data` to LSD. LSD tells that there is "No (more) solution" and gives the hand back to the user. If the data is found, the result is constructed following L_2.

The result is sent to LSD possibly in two messages. The first message contains the values of all the wanted fields except the "data value" field. If the data value field was requested by the query, a second message contains its value. Indeed, as already mentioned, the types of the data values change for each traced program whereas the type of all the other fields are set at the construction of the LSD instance. The procedure sending all the fields except the data value can be specialized at the construction of the LSD instance, whereas the procedures sending and receiving the data value field must "parse" the value, and code the structure at one end, and decode the structure at the other end.
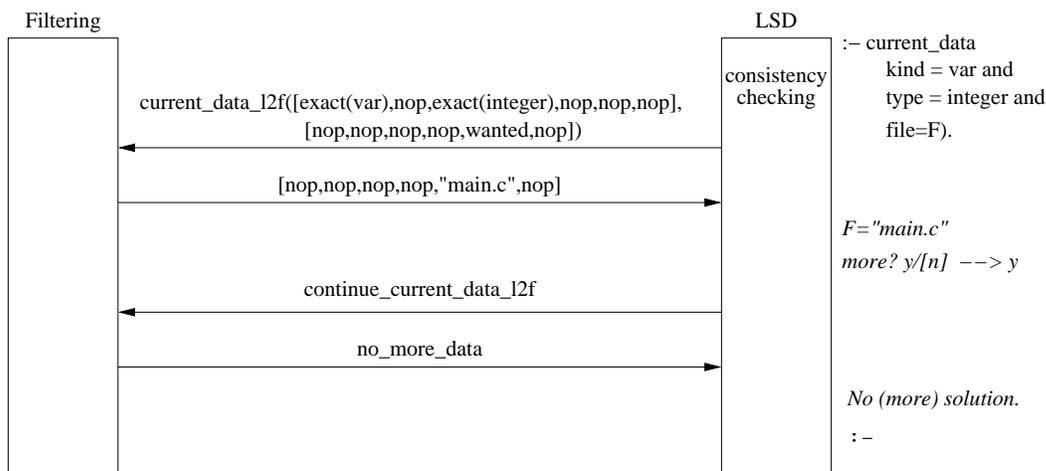
Filtering   LSD

```
                                                          consistency   :– current_data
                                                          checking        kind = var and
  current_data_l2f([exact(var),nop,exact(integer),nop,nop,nop],           type = integer and
                   [nop,nop,nop,nop,wanted,nop])                          file=F).

       [nop,nop,nop,nop,"main.c",nop]

                                                                       F="main.c"
                                                                       more? y/[n]  --> y
          continue_current_data_l2f

               no_more_data

                                                                       No (more) solution.
                                                                       :–
```

Figure 16: Example of communication between LSD and the filtering for a `current_data` query (the value of the data is not requested by the user query)

The format of the first message is similar to the one used for the `fget` primitive. When LSD receives the first part of the answer, if the data value was requested it sends a `send_data_value_l2f` request to the filtering module which sends the actual value in a canonical format. Note that this canonical format is not defined, yet. In the current instances, the procedures to code and decode the values are currently totally ad hoc and probably bugged.

The two messages are illustrated by Figure 15. The query requests the value and the definition file of a variable of type integer, if any such variable exists. One variable is found and the file information is sent in the first message. The value of the variable is sent in the second message.

Figure 16 illustrates the processing of a query when no data value is requested. Only the list of "regular" fields is sent back to LSD.

If the user decides to search for another data satisfying the same query, LSD sends `continue_current_data_l2f` to the filtering. Otherwise the user is prompted for another LSD query. This is also visible in Figure 16, in that case there is no other variable of type integer. The filtering module sends `no_more_data` and LSD answers `No (more) solutions`.

Interaction between the filtering module and the tracer   The general scheme of the detailed processing of the `current_data` primitive is shown on Figure 17. The data filtering algorithm is as follows:

1. The filtering sends `next_data_f2t` to the tracer;
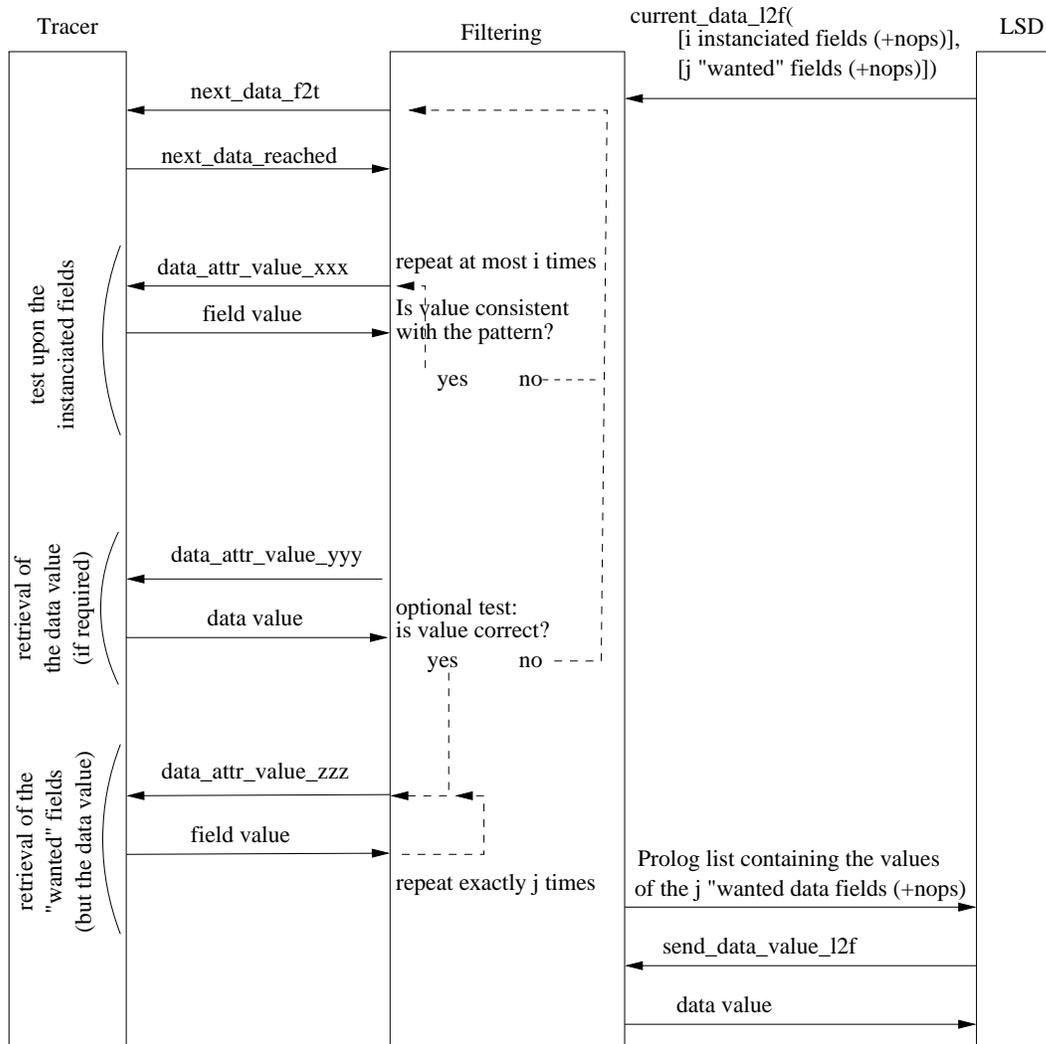
2. The tracer answers `next_data_reached`;

Figure 17: General scheme of communication between the filtering and the tracer for a `current_data` query

3. The filtering enters the "test loop", *ie* it retrieves the values of the i fields that are constrained in L_1 one after another and confronts them to the matching pattern.

   This is done by calling the functions `data_attr_value_xxx()`, where `xxx` stands for the name of the concerned field. The designer of the tracer is in charge of implementing those functions in the tracer;

   The test loop can be exited in three ways:

   - when there are no more data to be considered, the "test loop" fails, and `no_more_data` is sent to LSD.

   - when the retrieved value of the field does not match its associated constraint in L_1, the "test loop" fail and the filtering algorithm goes back to point 1,

   - when all of the i fields were processed, the "test loop" succeeds, the filtering algorithm goes to point 4.

4. When the test loop succeeds, the filtering module enters the "retrieving loop". The values of the j fields marked as wanted in L_2 are retrieved one after another by calling te relevant j `data_attr_value_xxx()` functions. The "retrieving loop" always succeeds.

Two examples of communication between the filtering and the tracer are shown in figures 18 and 19. Those two examples differ only in that the latter involves a constraint upon the value field of the data. When such a constraint exists, the test between the actual value of the data and the value mentioned in the query is done by the filtering. Each operator on such a value requires a special implementation. At present, the only allowed operator is the unification operator `=`.

4.2 The event filtering algorithm and its specialization

The tracer driver is the interface between the LSD process and the tracer. We have already emphasized that one of the keys of the efficiency of event filtering comes from the fact that the filtering procedure, part of the tracer driver, is inserted in the tracer process. In this section we first describe the overal activity of the tracer pilot which handles all the primitives coming from LSD. We then describe the *critical path* which corresponds to the event filtering and which *must* be optimized as much as possible. We then give some excerpts of the specialization of the event filtering algorithm for Eclipse and Mescaline, as well as for Coca.

4.2.1 The tracer driver

The tracer driver handles the query coming from LSD, it pilots the tracer, it retrieves event attributes and data fields, it filters the retrieved information and constructs the answers that it sends back to LSD.

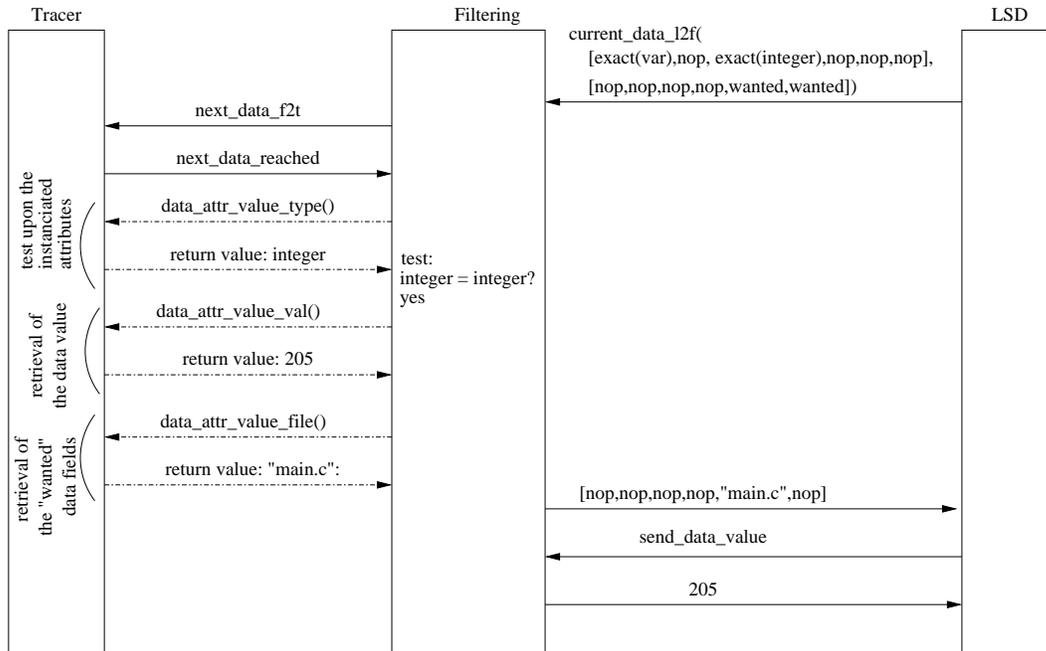Piloting the tracer is achieved via a simple finite state automaton which is updated at each event of the execution.

Figure 18: Example of communication between the filtering and the tracer for the current_data query from figure 15 (the data value is not constrained)
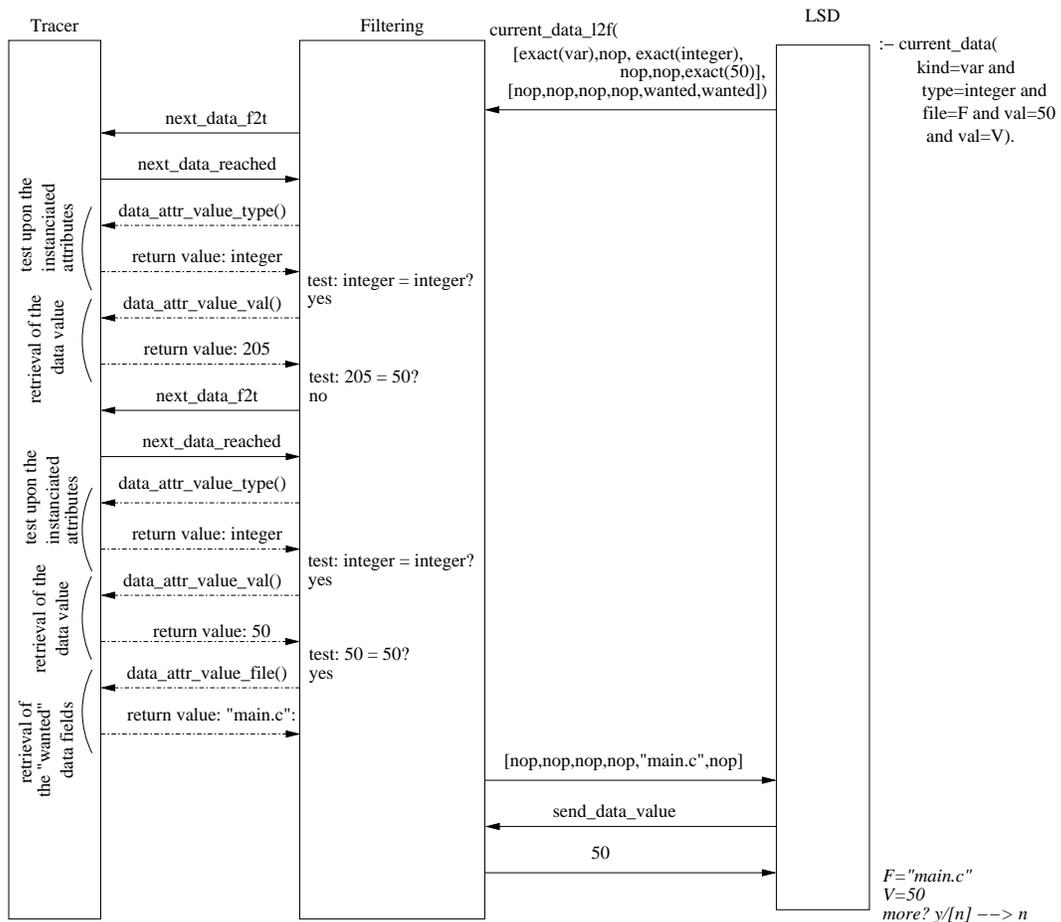
Figure 19: Example of communication between the filtering and the tracer for the `current_data` query from figure 15 (the data value is constrained)
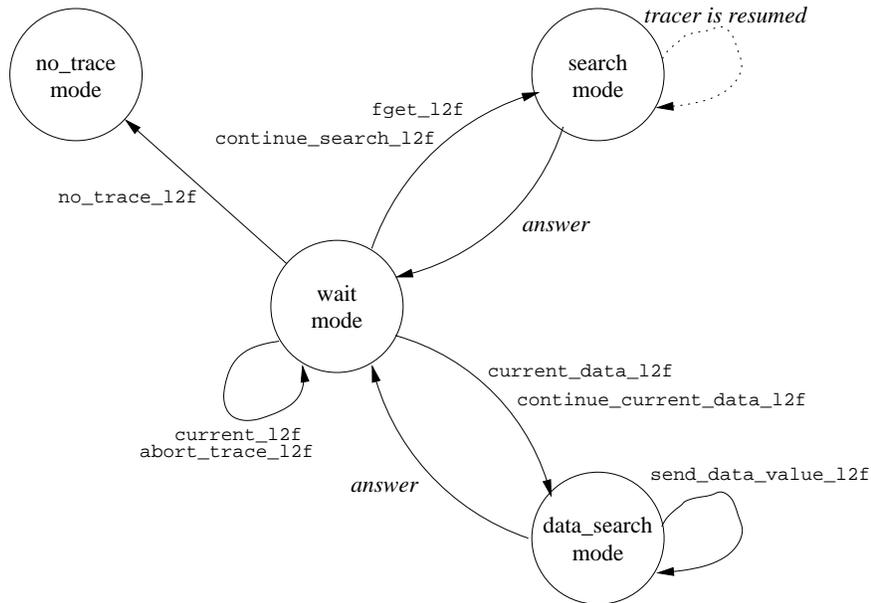
Figure 20: Finite state automaton of the tracer driver

This automaton, presented on Figure 20, has four states: wait mode, search mode, data_search mode and no_trace mode. The automaton behaves as follows.

- At the beginning of each traced execution, the automaton is set in wait mode.

- While in wait mode, the filtering module reads and parses a query from LSD. If the query corresponds to

  - `fget_l2f` or `continue_search_l2f`: the automaton switches to search mode;
  - `current_l2f` or `abort_trace_l2f`: the automaton stays in wait mode. For `current_l2f` it retrieves the requested attribute values and sends the answer to LSD. For `abort_trace_l2f` it aborts the traced execution and thus stops the trace.
  - `current_data_l2f` or `continue_current_data_l2f`: the automaton switches to data_search mode;
  - `no_trace_l2f`: the automaton switches to no_trace mode.

- While in search mode, the filtering module checks whether the current event satisfies the query. If so, the answer is sent to LSD and the automaton goes back to wait mode so as to read the next query. Otherwise the tracer is told to jump to the very

next event and the automaton remains in search mode. If the end of the execution is reached before a matching event is found the automaton returns to wait mode.

- While in data_search mode, the filtering module checks whether there exists some data which satisfy the query. If so, the requested fields are sent to LSD otherwise the answer is "no". In any case the automaton goes back to wait mode.

- The no_trace mode means that the execution of the program under scrutiny can continue without being traced and filters. The tracer therefore resumes the execution and will not called the filtering module again for the current execution.

Note that when LSD runs, the state of the automaton is always wait_mode. However, when the tracer runs the hand the state of the automaton is wait mode before the first event; then it is always search mode until either no_trace has been encountered or the end of the execution is reached.

4.2.2   The critical path

Only a part of this automaton requires special attention so that the filtering can be efficient. This part is called the *critical path*, it consists of most of the sequence of instructions executed in search mode. It corresponds to what the tracer and the filtering module compute at each event in order to process an `fget` query. The algorithm checks whether the values of the attributes of the event pattern match their current values. If not, the tracer is told to jump to the next event.

The event filtering algorithm seems simple enough. If we remember, however, that the filtering procedure may have to test several millions of events before finding a matching one, we can understand that every single optimization, even the smallest one, has an enormous impact on the response time. A number of considerations should be taken into account.

- The first consideration is that everything which is not in the critical path does not need to be optimized. In particular, in wait mode' when receiving a `fget` query and before switching to search mode, everything that can be done to prepare the job of the criticical path must be done. In the excerpts of the instances we show that this can be pushed rather far.

- An essential point is that the event filtering **must** be implemented in the same programming language as the tracer. The interface between the tracer and the filtering cannot cost more than a procedure call. This is visible from the measures that we made for the Morphine prototype. Some problems were hidden in the interface between Mercury code and C code. This resulted in a slowdown of more than an order of magnitude [14].

- The number of attribute tests should be as limited as possible, therefore the code should be as specialized as possible. For example, it is forbidden to test attributes which are not constrainted in the event pattern.

- The order of the tests is significant. The cheapest tests should be performed the earliest. Indeed, testing the equality of two integers is much cheaper than testing that a string is a member of a list. To define the order of the tests, both the type of the attributes and the complexity of the operators must be taken into account.

- Any programming trick is, for once, allowed in order to reduce the cost of the critical path. This means that for each instance, the critical path must be identified, and optimizations depending of the language in which the tracer is programmed should be done, possibly by hand.

The overal optimization of the filtering procedure is actually a specialization of the generic (and very simple) algorithm to dedicated tracers, trace format, programming languages. Some research prototypes of specializer exist (for example Tempo for C [4]). At present we have no idea whether they could be used in the context of LSD and this is out of the scope of this report.

4.2.3   Specialization of the event filtering for Eclipse and Mescaline

$LSD_{eclipse}$ and $LSD_{mescaline}$ have roughly the same filtering alogorithm.

The tracer pilot is implemented by four predicates, each of them corresponding to a state of the finite state automaton of Figure 20. They are namely `p_wait/2`, `p_search/2`, `p_data_search/2` and `p_no_trace/2`. The transitions in the automaton are implemented by the `traite_requete/2` predicate which is called by the tracer at each event.

The programming trick used here is that this predicate is dynamically re-compiled on each transition between distinct states for efficiency purposes. This is made possible by Prolog incremental compilation capability. For example, the automaton is initialized in the *wait* mode by the following Prolog goal:

```
:- compile_term(traite_requete(A1, A2) :- p_wait(A1, A2)).
```

The *search* mode of the automaton is implemented as follows:

```
p_search(_, Traceline) :-
      ( fget_answer(Traceline, Result) ->
            send_message_to_lsd(Result),
            compile_term(traite_requete(A1, A2) :- p_wait(A1, A2)),
            traite_requete(_, Traceline)
      ;
            true  % fget pattern not satisfied => go to next event
      ).
```

That is to say, if we can build an answer to the query, then we send the result and switch the automaton in the *wait* mode. Otherwise, we stay in *search* mode so as to find a satisfying event later. Most of the critical path is hidden in the `fget_answer` predicate, which retrieves and tests the bounded attributes of the pattern by calling an `fget_test/1` predicate. This predicate is defined as:

```
fget_test(_Traceline, [], []).
fget_test(Traceline, [AttrName |Ref], [Test |TestFilter]) :-
        retrieve_event_attr(Traceline, AttrName, Value),
        test_ok(Test, Value),
        fget_test(Traceline, Ref, TestFilter).
```

Each relevant attribute of the pattern is retrieved and compared to the filter. If the comparison is a success, the investigation goes on, otherwise it fails.

It should be noted that for efficiency purposes, only the relevant attributes of the query are used in the recursion. This is done by dynamically recompiling the `fget_test` predicate each time a new fget query is entered. The filtering module is then in wait mode; as it is outside the critical path, taking time to prepare for the critical path is therefore a good policy.

4.2.4 Specialization of the event filtering for Coca

The LSD_{coca} instance that works with the Coca tracer implements the filtering algorithm in the C language. This implementation is not stable.

The finite state automaton of Figure 20 is implemented in the `traiteRequete()` function which is called by Coca at each execution event. This function consists of a `switch` statement whose cases are the states of the automaton.

The critical path is implemented by the `filtre` function as follows:

```
int filtre(requete* req) {
  if (req->typ == FGET)
    return (
        ((req->ctr[0] == NULL) || (verif_contrainte(0,ENUM,req->ctr[0])))
        &&
        ((req->ctr[1] == NULL) || (verif_contrainte(1,ENUM,req->ctr[1])))
        &&
        ((req->ctr[2] == NULL) || (verif_contrainte(2,INTEGER,req->ctr[2])))
        &&
        ((req->ctr[3] == NULL) || (verif_contrainte(3,INTEGER,req->ctr[3])))
        &&
        ((req->ctr[4] == NULL) || (verif_contrainte(4,INTEGER,req->ctr[4])))
        &&
        ((req->ctr[5] == NULL) || (verif_contrainte(5,STRING,req->ctr[5])))
        &&
        ((req->ctr[6] == NULL) || (verif_contrainte(6,INTEGER,req->ctr[6])))
        &&
        ((req->ctr[7] == NULL) || (verif_contrainte(7,STRING,req->ctr[7])))
        &&
        ((req->ctr[8] == NULL) || (verif_contrainte(8,STRING,req->ctr[8])))
    );
  /* snipped */
}
```

In the above code, `req` represents the matching pattern and `req->ctr[`i`]` represent the attributes of the events. `req->ctr[`i`]` is NULL if the ith attribute is not present in the query, and `verif_contrainte` returns true if the actual attribute value satisfies the query.

5 Trace models

This section describes the trace models of the three tracers currently connected to LSD, namely the Eclipse Prolog tracer, the constraint logic programming tracer, Mescaline and the C tracer, Coca.

As already mentioned, the filtering scheme previously described can only work when connected to an *event-oriented tracer*. Such a tracer is able, firstly, to stop at execution points of interest (called *breakpoints*) and, secondly, to retrieve information related to these points. It can thus generate a sequence of tuples (called *events*) which can be automatically filtered by the `fget` primitive, and retrieved by the `current` primitive.

Furthermore, if the traced language has large data structures of unknown length, or possibly many global variables, the tracer should be able to retrieve the values of these data item by item upon request from the `current_data` primitive, as specified in the previous section.

When designing a trace model it is very important to determine properly the separation between event attributes and data fields. Indeed, some data have fixed length and are not too costly to retrieve and filter, they should be put in the event attributes in order to participate to the efficient filtering. This is, in particular, the case for the arguments of Prolog predicates. Even if some arguments can be very large, their number is known with the arity of the predicate. When implementing the filtering primitive, only the required part of the required arguments should be retrieved, then the filtering can be more efficient than with an explicit `current_data`.

A last issue which has not been addressed in the previous section is that some events or data kinds require specific attributes, for example in CLP(FD) when a domain is reduced it is very useful to see how it has been reduced, this information has no meaning at the other events. Specific attributes should be carefully taken into account in the design of the primitives. It is therefore essential to detect all of them before starting to implement the primitives.

In the remaining of this section the trace models describe thus

1. *The event types* which specify the interesting breakpoints.

2. *The EventPattern attributes* which specify the attributes of the events which can be handled by the `fget` and `current` primitives.

3. *The DataPattern fields* which specify the fields of the data which can be handled by the `current_data` primitive.

4. *The specific attributes and fields*

The descriptions are systematic but remain informal. A formal specification of a trace model for pure Prolog can be found in [15]; and the formal trace model of Mescaline can be found in [16].

5.1 The Eclipse Prolog trace model

The trace model of the Eclipse Prolog tracer is a superset of the box model of Byrd [3].

The event types The types of events are called `ports`. All events are related to a goal, called *the current goal* in the following. The list of ports is as follows:

`call`: the current goal is invoked. The instantiation of its arguments at the moment of the invocation is given;

`exit`: the current goal succeeds. The resulting instantiation of its arguments is given;

`redo`: the execution is backtracking either to the current goal or to one of its subgoals;

`fail`: the current goal fails;

`leave`: the execution of the current goal is "aborted" as a result of an `exit_block` exception;

`delay`: the current goal is delayed;

`resume`: the current goal, which had previously being delayed, is resumed;

`next`: an alternative clause is tried to solve the current goal;

`else`: an alternative in inline disjunction (;) is tried to solve the *parent* goal;

The EventPattern attributes The event attributes of the Eclipse tracer are:

`port`: the type of the event, as listed above;

`depth`: in the proof tree, namely the number of ancestors of the current goal plus one;

`invoke`: invocation number of the current goal;

`context_module`: name of the module in which the current goal is invoked;

`definition_module`: name of the module where the predicate of the current goal is defined;

`name`: name of the predicate of the current goal;

`arity`: arity of the predicate of the current goal;

arguments: List of the instanciated arguments[7] of the current goal, if any. This attribute is unavailable at *fail* and *leave* events.

det_exit : either `yes` (the exit is deterministic, no choice point is left) or `no` (the exit is not deterministic, there are choice points left). It is a specific attribute of `exit` ports;

parent: the parent goal, which is a tuple containing all the attributes of a goal except the parent goal. The attributes in the list are ordered following the order of the declaration of the attributes in the trace format specification file.

The DataPattern fields   The state information gathers three different kinds of data: global variables (very similar to C global variables), constraint store, ancestor goal stack.

Note that ancestors and constraints are goals, as a consequence their fields are all the attributes of an event except the `port` which does not make sense it this context. The `depth` is also absent from the the constraint fields. Indeed, constraints are not handled like ordinary goals and there is therefore no notion of ancestors for a constraint.

The data fields are as follows:

- **kind** = global (global variable)

  name: name of the variable;

  type: type of the variable;

  value: value of the variable;

  module: definition module of the variable.

- **kind** = constraint

  invoke : invocation number of the current constraint

  context_module: context of the invocation of the current constraint

  definition_module: module of definition of the current constraint

  name: of the predicate of the current constraint

  arity: of the predicate of the current constraint

  arguments of the current constraint

- **kind** = ancestor

  depth: in the proof tree;

  invoke: invocation number of the current ancestor;

  context_module: context of the invocation of the current ancestor;

---

[7]Arguments should not to be confused with variables, an argument can be built with variables but does not have to

```
                                   1 [1]  call    ancestor(maryvonne, Y)
                                   2 [2]  call    parent(maryvonne, Y)
                                   2 [2]  exit    parent(maryvonne, ben)
                                   1 [1] *exit    ancestor(maryvonne, ben)
parent(jean, ben).
parent(maryvonne, ben).         Y = ben (More ?) y
parent(simone, jean).
parent(roger, jean).
                                   1 [1]  redo    ancestor(maryvonne, Y)
                                   3 [2]  call    parent(maryvonne, Z)
ancestor(X, Y) :-                  3 [2]  exit    parent(maryvonne, ben)
    parent(X, Y).                  4 [2]  call    ancestor(ben, Y)
ancestor(X, Y) :-                  5 [3]  call    parent(ben, Y)
    parent(X, Z),                  5 [3]  fail    parent
    ancestor(Z, Y).                4 [2]  next    ancestor(ben, Y)
                                   6 [3]  call    parent(ben, Z)
                                   6 [3]  fail    parent
                                   4 [2]  fail    ancestor
                                   1 [1]  fail    ancestor
                                No more solution
```

Figure 21: Trace of the execution of goal `ancestor(maryvonne, Y)` by the Eclipse tracer

> `definition_module`: module of definition of the predicate of the current ancestor;
>
> `name`: of the predicate of the current ancestor;
>
> `arity`: of the predicate of the current ancestor;
>
> `arguments` of the current ancestor.

Specific attributes   As said above, some attributes of the Eclipse trace model are specific to given ports.

`arguments`: not defined on *fail* and *leave* ports;

`det_exit`: defined on *exit* events only.

`name` : the else port gives a predicate name which is actually the name of the parent goal. The meaning of the predicate name attribute at "else" events is therefore different than at other ports. Even if this has no consequence on the implementation of the filtering mechanisms, the attribute should be interpreted with care by analyses.

Example of trace   Figure 21 shows the source code of two predicates, `ancestor` and `parent`. It also shows the execution trace of `ancestor(maryvonne, Y)`. The trace events are shown with some event attributes, namely the invocation number, the depth within square brackets, the port, the name of the traced predicate and its arguments whenever they are available. At `exit` events, a star in front of the port name tells wether there are

still some choice points (line #4). The ancestor `ben` is found by using the first clause of the `ancestor` predicate and calling the `parent` predicate. This first solution is displayed and the user asks for more solution. The execution than backtracks to the resolution of the first goal and tries the second clause of the `ancestor` predicate. It calls `parent` again, and again finds `ben`. This time `ancestor` is recursively called to find the descendants of `ben`. As `ben` is the parent of nobody, trying both the first clause and the second clause lead to failures. There is therefore no more solution to `ancestor(maryvonne, Y)`.

Remark   The Eclipse tracer also has some ports dedicated to spying data. This is not properly integrated in LSD$_{eclipse}$, yet. These events are not goal oriented.

The extra ports are:

`spyterm`: a new data spy point is set;

`modify`: spied data were modified.

The attributes are:

`term_spied`: the name of the variable that is being spied, or that was spied and modified;

`value_affected`: the value being given to the modified spied variable. This attribute is specific to the `modify` port.

5.2   The Mescaline CLP(FD) trace model

The following description is taken from [16] where a this model is explained in detail.

The event types   Whereas Prolog tracers provide ports which model the control flow of the execution, in Mescaline, ports model the constraint propagation:

`tell`: a new constraint is added to the store;

`reduce`: a domain is reduced by a constraint;

`wake_up`: a constraint is put in the propagation queue;

`suspend`: a constraint is suspended;

`true`: a constraint is fully solved (it cannot lead to further reductions);

`solution`: the state of the domains is a solution of the problem constraints;

`reject`: a constraint is rejected because it has emptied some domains;

`select`: a constraint of the propagation queue has become active;

`told`: a constraint is withdrawn from the store.

The EventPattern attributes   In this model, most attributes are common to all events. The common attributes are as follows, the specific attributes are given below.

`chrono`: the event number (starting with 1);

`depth`: the depth of the execution, starting with 0, it is incremented at each `tell` and decremented at each `told`;

`port`: the event type as presented above;

`constraint`: the concerned constraint, represented by a quadruple:

- a unique identifier generated at its `tell`,
- an abstract representation, identical to the source formulation of the program,
- a concrete representation (e.g. `diffN(X, Y, N)` for `X ## Y + N` or `X - N ## Y`) and
- the invocation context, namely the Prolog goal from which the `tell` is performed;

The DataPattern fields

`domains`: the set of the values of the variable domains before the event occurs;

`store`: the content of the constraint store, represented by 5 components. Each set of constraints is represented by a list of pairs (constraint identifier, external representation):

`store_A` the set of *active* constraints;

`store_S` the set of *suspended* constraints;

`store_Q` the *propagation queue*;

`store_T` the set of *solved* constraints;

`store_R` the set of *rejected* constraints.

Specific attributes
For `reduce` events:

`updated variable`: The name and unique identifier of the variable which is reduced;

`withdrawn`: The withdrawn domain;

`update`: The list of update types. An update type can be one of `ground, any, min, max, min-max, emptied`.

For `wake_up` events:

`cause`: The verified part of the awakening condition;

```
sorted([X, Y, Z]):-
    [X, Y, Z] :: 1..3,          % At the beginning, Dx = Dy = Dz = [1..3]
    X ## Y, X # >= Y, Y # > Z,   % 3 constraints : x ≠ y, x ≥ y and y > z
    labelling([X, Y, Z]).        % labelling phase, with a "first fail" strategy
```

```
 1 [1] post    X##Y   X:[1,2,3] Y:[1,2,3]           21 [4] true    X#>=Y  X:[2]     Y:[2]
 2 [1] suspend X##Y   X:[1,2,3] Y:[1,2,3]           22 [4] select  X##Y   X:[2]     Y:[2]
 3 [2] post    X#>=Y  X:[1,2,3] Y:[1,2,3]           23 [4] reduce  X##Y   X:[2]     Y:[2]    X[2]
 4 [2] suspend X#>=Y  X:[1,2,3] Y:[1,2,3]           24 [4] reject  X##Y   X:[]     Y:[2]
 5 [3] post    Y#>Z   Y:[1,2,3] Z:[1,2,3]           25 [4] told    X#=2   X:[]
 6 [3] reduce  Y#>Z   Y:[1,2,3] Z:[1,2,3] Y[1]      26 [4] post    X#=3   X:[2,3]
 7 [3] wake_up X#>=Y  X:[1,2,3] Y:[2,3]             27 [4] reduce  X#=3   X:[2,3]          X[2]
 8 [3] reduce  Y#>Z   Y:[2,3]   Z:[1,2,3] Z[3]      28 [4] wake_up X##Y   X:[3]     Y:[2,3]
 9 [3] suspend Y#>Z   Y:[2,3]   Z:[1,2]             29 [4] true    X#=3   X:[3]
10 [3] select  X#>=Y  X:[1,2,3] Y:[2,3]             30 [4] select  X##Y   X:[3]     Y:[2,3]
11 [3] reduce  X#>=Y  X:[1,2,3] Y:[2,3]    X[1]     31 [4] reduce  X##Y   X:[3]     Y:[2,3] Y[3]
12 [3] suspend X#>=Y  X:[2,3]   Y:[2,3]             32 [4] wake_up Y#>Z   Y:[2]     Z:[1,2]
13 [4] post    X#=2   X:[2,3]                       33 [4] true    X##Y   X:[3]     Y:[2]
14 [4] reduce  X#=2   X:[2,3]              X[3]      34 [4] select  Y#>Z   Y:[2]     Z:[1,2]
15 [4] wake_up X#>=Y  X:[2]     Y:[2,3]             35 [4] reduce  Y#>Z   Y:[2]     Z:[1,2] Z[2]
16 [4] wake_up X##Y   X:[2]     Y:[2,3]             36 [4] true    Y#>Z   Y:[2]     Z:[1]
17 [4] true    X#=2   X:[2]                         37 [4] told    X#=3   X:[3]
18 [4] select  X#>=Y  X:[2]     Y:[2,3]             38 [3] told    Y#>Z   Y:[2,3]   Z:[1,2]
19 [4] reduce  X#>=Y  X:[2]     Y:[2,3]    Y[3]     39 [2] told    X#>=Y  X:[1,2,3] Y:[1,2,3]
20 [4] wake_up Y#>Z   Y:[2]     Z:[1,2]             40 [1] told    X##Y   X:[1,2,3] Y:[1,2,3]
```

```
chrono     = 14
depth      =  4
port       = reduce
constraint = (4, X#=2, assign(var(1, X), 2), labelling([X, Y, Z]))
domains    = [X::[2..3], Y::[2..3], Z::[1..2]]
withdrawn  = X::[3]
update     = [X->any, X->ground, X->max]
store_A    = [(4, X#=2)]                    store_S = [(2, X#>=Y), (3, Y#>Z), (1, X##Y)]
store_Q    = []                store_T = []    store_R = []

chrono     = 16
depth      =  4
port       = wake_up
constraint = (1, X##Y, diff(var(1, X), var(2, Y)), sorted([X, Y, Z]))
domains    = [X::[2], Y::[2..3], Z::[1..2]]
cause      = [X->ground]
store_A    = [(4, X#=2)]                    store_S   = [(3, Y#>Z), (1, X##Y)]
store_Q    = [(2, X#>=Y)]    store_T = []    store_R = []
```

Figure 22: A trace of the execution of program sorted([X, Y, Z]), all events are present with the following attributes and data fields: chrono, [depth], port, constraint, domains of related variables, and updated_variable and withdrawn at reduce events. Events #14 and #16 are displayed with all their attributes, and the domains of all the related variables.

Syntax	Event name	::=	Event components
for (EXPR; EXPR; EXPR) BLOCK	FOR_EV	::=	(enter, for) EXPR_EV TEST_EV {BLOCK_EV EXPR_EV TEST_EV}* (exit, for)
if (EXPR) BLOCK else BLOCK	IF_EV	::=	(enter, if) TEST_EV THEN_ELSE_EV (exit, if)
	THEN_ELSE_EV	::=	(enter, then) BLOCK_EV (exit, then)
		|	(enter, else) BLOCK_EV (exit, else)
EXPR	EXPR_EV	::=	(enter, expr) (exit, expr)

Figure 23: Extract of the Coca C grammar of events

Example of trace Figure 22 shows the source code of program `sorted(L)`. The program sorts three numbers between 1 and 3 in a very naive way. Following the convention of many systems, constraints operators are prefixed by a "`#`". The figure also shows a trace of the execution. All events are listed but only with a few event attributes: the event number and port, the constraint concerned by the event and its variable domains. At `reduce` events, the variable whose domain is being reduced as well as the withdrawn values are added.

The first two constraints are entered (`tell`) and suspended without any reduction (events #1 to #4). The `tell` of the third one, `Y #> Z` gives two value withdrawals, '1' from D_y (#6) and '3' from D_z (#8). The first reduction modifies the lower bound of D_y and so wakes the suspended constraint `X #>= Y` (#7). After those two reductions the constraint is suspended and the waiting one is selected (#10). At event #12, the domains are $D_x = \{2,3\}, D_y = \{2,3\}$ and $D_z = \{1,2\}$. Then the labelling phase begins. With our simple "first fail" strategy, the first added constraint is `X #= 2`. `X` is ground and equal to 2 and this constraint is solved (#17). Two other constraints are solved during the propagation, but it leads to $D_y = \{3\}, D_z = \{1,2\}$ and an empty domain for `X` (#25). Another labelling constraint is tried (#26), `X #= 3` and leads to the unique solution `{X:3, Y:2, Z:1}`.

5.3 The trace model of the C tracer Coca

The following trace model is used in the Coca instance of LSD. The Coca tracer had initially been hacked into GDB [7] but GDB was not at all adapted to the handling of trace information and the current prototype of tracer is implemented by program transformation. More information can be found in the implementation documentation of LSD.

The event types Events are associated to C constructs. This is specified by a grammar; Figure 23 shows the events associated to three constructs: the `for` loop, the `if-then-else` conditional and expression evaluation. A `for` has four components: an initialization, a test, an iteration and a block of actions. The corresponding events are 1) entering the `for`, 2) the sequence of events related to the evaluation of the initialization, 3) the sequence of events

related to the test, 4) the (possibly empty) sequence of instructions related to the execution of the block, the iteration and the test, and 5) exiting the `for`. The full grammar is given in appendix A.

One can notice that there are no events to give details about the evaluation of expressions. However complicated the expression, only two events will be generated: one before its evaluation, one after. The reason for this design choice is that generating events dedicated to expression would be very costly, while there is no evidence that this information is much needed. We nevertheless offer a compromise. If users are puzzled by the results of the evaluation of a particular expression, there exists an expression evaluation simulator which can decompose evaluation of expressions at will. This simulation is quite straightforward to implement with Prolog and `current_data`.

The EventPattern attributes   The event attributes are:

   type: name of the construct the event is associated with. It can be one of `if`, `then`, `else`, `for`, `while`, `do_while`, `switch`, `case`, `default`, `block`, `function`, `return`, `break`, `continue`, `goto`, `label`, `create`, `expr`, `del`, `test`, `incr`, `init`.

   port: indicates whether the execution is entering or exiting the construct. It can be one of `enter`, `exit`, `nil`.

   The `nil` port is reserved for constructs for which `enter` and `exit` do not make sense, namely `break`, `continue`, `label`, `goto`, `create` and `del`.

   func: name of the function in which the construct is defined.

   chrono: the chronological number of the event (a time stamp).

   cdepth: call depth of the encompassing function. This tells how many functions are in the stack of calls. Especially useful to distinguish recursive calls.

   bdepth: block depth of the construct inside the encompassing function.

   file: in which the encompassing function is defined.

   line: in the source file where the construct related to the event is defined.

The DataPattern fields

   name: variable descriptor (C "lvalue")

   type: C type of the variable

   val: variable value

   addr: variable address in memory

   size: memory size of the variable

`file`: file where the variable is declared

`line`: line where the variable is declared

`bdepth`: block depth of the variable declaration inside the encompassing function.

`status`: tells whether the variable is global or local.

Specific attributes   Currently, the Coca C trace model does not make use of any specific attribute. However, the `nil` port can be considered like a "patch". In a further implementation, `port` should be a specific attribute of `type`.

Examples of trace   In the execution of the program given in section 2, the event corresponding to entering a `for` loop in function `legal` is:

```
[lsd_c] print_event.
  type:   for         port:   enter
  func:   legal       cdepth: 6
  chrono: 626         line:   76
  file: coca/demo/reines.c
```

The values of all the fields of the first visible variable of the previous event is

```
[lsd_c] print_data.
name : ok         type : int
val :  1          addr : -268438984
size : 4          linedecl : 75
filedecl : coca/demo/reines.c
```

6   Towards a multilanguage LSD environment

In the previous sections, LSD is considered as a set of trace analyser *instances*, and a *generator*. This is not the whole truth as LSD provides a multi-module mechanism which allows several instances to work from a same LSD. The instances can be for different or identical traced languages.

Therefore, LSD tends to be a multi-language debugging environment (see Figure 24). The analyser of an LSD session is made of global code (the main LSD module) and of one *instance specific module* or more. Each instance specific module inside an LSD session is independant from the other instance specific modules if any. Each instance specific is connected to a unique tracer through a filtering procedure. The multi-process communication is done via a socket.

This architecture has several advantages. Firstly, the maintenance of the analysis module is easier because there is only place to look at. Secondly, the extensions developed in the
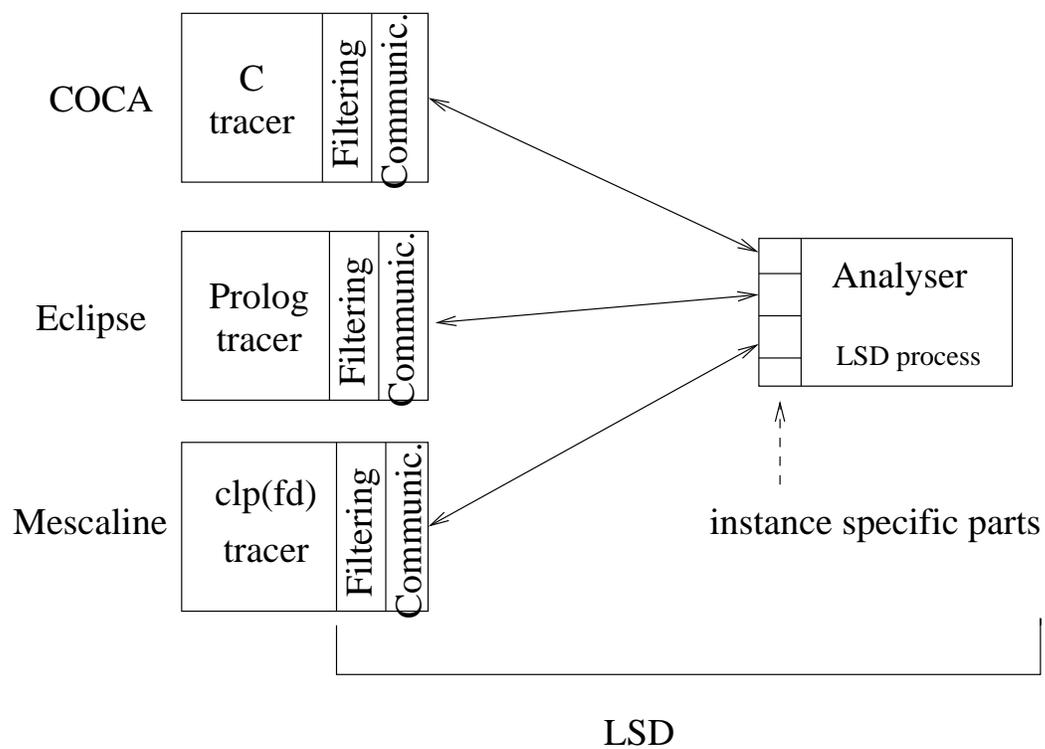
Figure 24: LSD general architecture

analyser for one instance can be made more easily available for the other ones. Indeed, even if many analyses depend on the trace model, they do not depend on the actual implementation of the tracer. Hence if we had two instances for two Prolog tracers which provide the same trace model, analyses developed for one instance can be easily adapted to the other instance. Furthermore, some analyses are not so dependent on the traced languages. The step by step tracer is the most obvious example. Lastly, running two executions in parallel, one can compare traces of two versions of the same program and therefore use the result of one version to interpret the result of another. This can be especially useful in the case of non-regression testing. In that case the previous version can act as the oracle for the current version under test.

7   Conclusion

In this report, we describe how to implement trace query tools which enable sophisticated queries to be asked. We propose an architecture where a tracer driver, containing a very efficient trace filtering algorithm, is integrated in the tracer process. In that way most queries are very efficient. The implementation guidelines are based on our experience accumulated while building four prototypes for different sorts of programming languages. Furthermore we are currently integrating these prototypes inside a single environment.

References

[1] The ECLiPSe constraint logic programming system.   http://www-icparc.doc.ic.ac.uk/eclipse/.

[2] 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.

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

[4] C. Consel, L. Hornof, J. Lawall, R. Marlet, G. Muller, J. Noyé, S. Thibault, and E.N. Volanschi. Tempo: Specializing systems applications and beyond. *ACM Computing Surveys*, 30(3), September 1998.

[5] M. Consens, M. Hasan, and A. Mendelzon. Visualizing and querying distributed event traces with Hy+. In W. Litwin and T. Risch, editors, *Applications of Databases, First International Conference*, pages 123–141. Springer, Lecture Notes in Computer Science, Vol. 819, 1994.

[6] M. Ducassé. Abstract views of Prolog executions with Opium. In P. Brna, B. du Boulay, and H. Pain, editors, *Learning to Build and Comprehend Complex Information Structures: Prolog as a Case Study*, Cognitive Science and Technology, chapter 10, pages 223–243. Ablex, 1999.

[7] M. Ducassé. Coca: An automated debugger for C. In *Proceedings of the 21st International Conference on Software Engineering*, pages 504–513. ACM Press, May 1999.

[8] M. Ducassé. Opium: An extendable trace analyser for Prolog. *The Journal of Logic programming*, 39:177–223, 1999. Special issue on Synthesis, Transformation and Analysis of Logic Programs, A. Bossi and Y. Deville (eds), Also Rapport Technique INRIA 3257.

[9] M. Ducassé and A.-M. Emde. Opium: a debugging environment for Prolog development and debugging research. *ACM Software Engineering Notes*, 16(1):54–59, January 1991. Demonstration presented at the Fourth Symposium on Software Development Environments.

[10] M. Ducassé and E. Jahier. *An automated debugger for Mercury - Opium-M 0.1 User and reference manuals*, May 1999. RT-231 INRIA. Also PI 1234 IRISA.

[11] M. Ducassé and E. Jahier. Efficient automated trace analysis: Examples with morphine. *Electronic Notes in Theoretical Computer Science*, 55(2), 2001. K. Havelund and G. Rosu (Eds), proceedings of the first Workshop on Runtime Verification.

[12] M. Ducassé and L. Langevine. Analyse automatisée de traces d'exécution de programmes CLP(FD). In M. Rueher, editor, *Actes des Journées Francophones de Programmation en Logique avec Contraintes*, pages 119–134. HERMES science publications, Mai 2002.

[13] M. Ducassé and L. Langevine. Automated analysis of CLP(FD) program execution traces. In P. Stuckey, editor, *Proceedings of the International Conference on Logic Programming*, pages 470–471. Lecture Notes in Computer Science 2401, Springer-Verlag, July 2002. Poster. Extended version available at http://www.irisa.fr/lande/ducasse/.

[14] E. Jahier and M. Ducassé. Generic program monitoring by trace analysis. *Theory and Practice of Logic Programming*, 2(4-5):611–643, July-September 2002. Also Rapport de Recherche INRIA RR-4323 and Publication Interne IRISA PI1425.

[15] E. Jahier, M. Ducassé, and O. Ridoux. Specifying Prolog trace models with a continuation semantics. In K.-K. Lau, editor, *Logic Based Program Synthesis and Transformation*. Springer-Verlag, Lecture Notes in Computer Science 2042, 2001.

[16] L. Langevine, P. Deransart, M. Ducassé, and E. Jahier. Tracing executions of **clp(fd)** programs: a trace model and an experimental validation environment. Rapport de Recherche RR 4342, INRIA, Novembre 2001.

[17] L. Langevine and M. Ducassé. Un pilote de traceur pour la plc. déboguer, auditer et visualiser une exécution avec un même traceur. In F. Mesnard, editor, *Actes des Journées Francophones de Programmation en Logique avec Contraintes*. HERMES Science Publications, June 2004.

[18] B. Lewis and M. Ducassé. Using events to debug Java programs backwards in time (demonstration). In R. Crocker and G. Steele Jr., editors, *Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 96–97. ACM, October 2003.

[19] Z. Somogyi and F. Henderson. The implementation technology of the Mercury debugger. In *Proceedings of the Tenth Workshop on Logic Programming Environments*, volume 30(4). Elevier, Electronic Notes in Theoretical Computer Science, 1999. http://www.elsevier.nl/cas/tree/store/tcs/free/entcs/store/tcs30/cover.sub.sht.

[20] R.M. Stallman and R.H. Pesch. *Debugging with GDB, the GNU Source-level debuger*. The Free Software Fondation, Inc, April 1998.

A   The full event grammar for the C trace model

PROGRAM	PROG_EV	::=	$\{CREATE\_EV\}^{0/1}$ FONC_EV
BLOC	BLOC_EV	::=	(enter, bloc) $\{CREATE\_EV\}^{0/1}$ SEVENT $\{DEL\_EV\}^{0/1}$ (exit, bloc)
	SEVENT	::=	EVENT SEVENT
		|	ε
	EVENT	::=	WHILE_EV
		|	DOWHILE_EV
		|	FOR_EV
		|	IF_EV
		|	SWITCH_EV
		|	BREAK_EV
		|	CONTINUE_EV
		|	GOTO_EV
		|	EXPR_EV
		|	RETURN_EV
		|	FONC_EV
while (EXPR) BLOC	WHILE_EV	::=	(enter, while) TEST_EV {BLOC_EV TEST_EV}* (exit, while)
do BLOC while (EXPR)	DOWHILE_EV	::=	(enter, dowhile) {BLOC_EV EV_TEST}+ (exit, dowhile)
for (EXPR; EXPR; EXPR) BLOC	FOR_EV	::=	(enter, for) EXPR_EV TEST_EV {BLOC_EV EXPR_EV TEST_EV}* (exit, for)
if (EXPR) BLOC $\{else\ BLOC\}^{0/1}$	IF_EV	::=	(enter, if) TEST_EV THEN_ELSE_EV (exit, if)
	THEN_ELSE_EV	::=	(enter, then) BLOC_EV (exit, then)
		|	(enter, else) BLOC_EV (exit, else)

switch (EXPR) {
 	
case Cst : BLOC
case Cst : BLOC
 	
$\{default\ BLOC\}^{0/1}$
}
	SWITCH_EV
	
	TESTSWITCH_EV
	EXEC_EV
		::=
		
		::=
		::=
			(enter, switch) TESTSWITCH_EV EXEC_EV (exit, switch)
			$\{(enter, case)\}+ \{(enter, default)\}^{0/1}$
			{BLOC_EV (exit, case)}* $\{BLOC\_EV (exit, default)\}^{0/1}$

return EXPR	RETURN_EV	::=	(enter, return) EXPR_EV (exit, return)
break	BREAK_EV	::=	(-, break)
continue	CONTINUE_EV	::=	(-, continue)
goto Etq	GOTO_EV	::=	(-, goto)
fct(EXPR,...) BLOC	FONC_EV	::=	(enter, fonction) BLOC_EV (exit, fonction)
... TEST ...	TEST_EV	::=	(enter, test) (exit, test)
variable definition	CREATE_EV	::=	(-, create)
EXPR	EXPR_EV	::=	(enter, expr) (exit, expr)
variables destruction	DEL_EV	::=	(-, del)