Automated analysis of clp(fd) program execution traces

Mireille Ducassé* Ludovic Langevine†

Deliverable D1.2.2.1 of the RNTL project OADYMPPAC.
February 2002

1 Introduction

Clp(fd) programs can solve complex problems but they are quite difficult to develop and maintain. In particular, their operational behavior is subtle and not easy to understand.

Execution tracers can give some insight of executions, but they are mapped onto the operational semantics of the language. This, in general, is a too low-level picture of the execution. In particular, application developers and end-users do not need to know all the details of the execution steps. They need abstract views of the behavior of the programs, for example they want to know how accurate the constraint propagation is.

Nevertheless, tracers and low-level traces are very useful in that they give a sound view of the executions. We propose an approach where high-level views are built on top of low-level tracers. We provide an analysis module which can filter, tailor and in general process the execution traces produced by a tracer.

We propose a trace query language based on Prolog augmented by a handful of primitives. With this language, end-users can directly ask questions about executions; application programmers can provide explanation programs dedicated to the application; solver developers can provide explanation programs dependent of their solvers.

The advantages of our approach are as follows. Firstly, at present, interesting tools exist, see for example [6], however their implementation is always ad hoc. They can hardly be ported from one system to another. With our scheme, operational trace analyses can be easily ported from one system to another provided that both systems offer tracers based on the same trace model. Secondly, retrieving information about executions requires to know a lot of the details of the implementation of the compiler. Not many programmers can program tools if they have to insert code in the compiler. With our scheme, a tracer is built once for all and this can be done by the compiler developer once the trace model is known. Thirdly, the application programmers or the end-users, who know best which abstract views could be useful, can then easily program trace analysis tools. They only need to know the

*IRISA/INSA de Rennes ; email : Mireille.Ducasse@irisa.fr
†INRIA Rocquencourt ; email : Ludovic.Langevine@inria.fr

trace model and the few primitives that we will present in the following. They do not need to open the code of the compiler.

Furthermore, we propose mechanisms to analyze the trace *on the fly.* These mechanisms allow most of the trace filtering to be done in the tracing process, hence without storing or even generating any extra information. Therefore, the cost of the analysis does not depend of the potential information that the tracer can generate. It depends solely of the amount of information actually required by the analysis and the complexity of the analysis itself. The advantage of such an optimization is significant: *the actual trace generation is directed by the analysis.* If a view or analysis does not need much information, the tracer will generate very little and only if the analysis requires much information the tracer will generate a lot. In the later case, we can conjecture that the problem to solve is hard and that the person requesting the analysis is keen to pay the higher price. A direct, and very important, consequence is that the trace model can be as rich as people want. Because the price of the richness will be paid only if the information is used. In traditional debuggers, for example, designers always have to solve a dilemma between efficiency and power, because the tracer will generate (and even sometime store) the whole information at each event. As efficiency is often the key, their trace information never contains "heavy" information. It is therefore poorer than it has to be.

The bases of the analysis scheme were initially designed for the analysis of Prolog programs [9]. They had been extended in order to cope with the global variables of C [8]. In this document, we extend them further to address the constraint store of clp(fd). We also apply them to clp(fd) dedicated analyses. This shows that the scheme is generic enough to accommodate a wide range of languages and tracers.

In the following, Section 2 recalls a powerful trace model that we have designed for clp(fd). Section 3 describes our trace analysis scheme. Section 4 explains how we have used these mechanisms to implement two abstract graphical views. Section 5 briefly discusses related work.

2 A clp(fd) trace model

A program execution can be represented by a stream of elementary events, called an execution trace. A trace model specifies, firstly, which events are of interest and, secondly, what information is attached to them. We give some details of the two aspects in the following. A full presentation of the model and its semantics can be found in [13].

Type of events Specifying the events of interest consists of defining which points of the execution should be traced. They are traditionally called breakpoints in the debugging literature. In the logic programming debugging community, the type of events is traditionally called ports. The ports of an imperative language are basically "enter" and "exit". For example when entering or exiting a function or a block. In logic programming the ports are more sophisticated because they have to take backtracking into account. Typically Prolog tracers provide the following ports: "call", "exit, "fail", "redo" [2]. They respectively mean that a goal is called, that it exited, that it fails and that its choice points are retried.

In clp(fd), in addition to the Prolog four basic ports which could still be useful, the constraint propagation has to be modeled. We introduce the following types of events: tell, reduce, wake-up, suspend, true, reject, select and told. They refer to events related to the constraint store.

- 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)
- 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

Event attributes In our model, most attributes are common to all events. For some ports, specific attributes are added. For example, reduce events have three additional attributes: the updated variable, the withdrawn values for the updated variable, and the types of updates made, such as a modification of the domain upper bound. The attributes are as follows.

Attributes for all events

- `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: one of reduce, wake-up, suspend, true, reject, select, tell and told;
- `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`);
  - the invocation context, namely the Prolog goal from which the tell is performed.
- `domains`: the set of the values of the variable domains before the event occurs;
- `store`: the content of the constraint store

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`;

Specific attribute for wake-up events

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

Figure 1 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, constraint 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, depth and port, the constraint concerned by the event and its variable domains. At reduce events, the updated variable as well as the type of the update and 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 the domain of y (#6) and '3' from the domain of z (#8). The first reduction modifies the lower

```
sorted([X, Y, Z]):-
   [X, Y, Z] ::  1..3,        % At the beginning, all domains are equal to [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] Tell    X##Y  X:[1,2,3] Y:[1,2,3]
 2 [1] Suspend X##Y  X:[1,2,3] Y:[1,2,3]
 3 [2] Tell    X#>=Y X:[1,2,3] Y:[1,2,3]
 4 [2] Suspend X#>=Y X:[1,2,3] Y:[1,2,3]
 5 [3] Tell    Y#>Z  Y:[1,2,3] Z:[1,2,3]
 6 [3] Reduce  Y#>Z  Y:[1,2,3] Z:[1,2,3] Y[1]
 7 [3] Wake-up X#>=Y X:[1,2,3] Y:[2,3]
 8 [3] Reduce  Y#>Z  Y:[2,3]   Z:[1,2,3] Z[3]
 9 [3] Suspend Y#>Z  Y:[2,3]   Z:[1,2]
10 [3] Select  X#>=Y X:[1,2,3] Y:[2,3]
11 [3] Reduce  X#>=Y X:[1,2,3] Y:[2,3]   X[1]
12 [3] Suspend X#>=Y X:[2,3]   Y:[2,3]
13 [4] Tell    X#=2  X:[2,3]
14 [4] Reduce  X#=2  X:[2,3]             X[3]
15 [4] Wake-up X#>=Y X:[2]     Y:[2,3]
16 [4] Wake-up X##Y  X:[2]     Y:[2,3]
17 [4] True    X#=2  X:[2]
18 [4] Select  X#>=Y X:[2]     Y:[2,3]
19 [4] Reduce  X#>=Y X:[2]     Y:[2,3]   Y[3]
20 [4] Wake-up Y#>Z  Y:[2]     Z:[1,2]
21 [4] True    X#>=Y X:[2]     Y:[2]
22 [4] Select  X##Y  X:[2]     Y:[2]
23 [4] Reduce  X##Y  X:[2]     Y:[2]     X[2]
24 [4] Reject  X##Y  X:[]      Y:[2]
25 [4] Told    X#=2  X:[]
26 [4] Tell    X#=3  X:[2,3]
27 [4] Reduce  X#=3  X:[2,3]             X[2]
28 [4] Wake-up X##Y  X:[3]     Y:[2,3]
29 [4] True    X#=3  X:[3]
30 [4] Select  X##Y  X:[3]     Y:[2,3]
31 [4] Reduce  X##Y  X:[3]     Y:[2,3]   Y[3]
32 [4] Wake-up Y#>Z  Y:[2]     Z:[1,2]
33 [4] True    X##Y  X:[3]     Y:[2]
34 [4] Select  Y#>Z  Y:[2]     Z:[1,2]
35 [4] Reduce  Y#>Z  Y:[2]     Z:[1,2]   Z[2]
36 [4] True    Y#>Z  Y:[2]     Z:[1]
37 [4] Told    X#=3  X:[3]
38 [3] Told    Y#>Z  Y:[2,3]   Z:[1,2]
39 [2] Told    X#>=Y X:[1,2,3] Y:[1,2,3]
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]]
updated_var= (X, 1)
withdrawn  = [3]
update     = [any, ground, max]
store      = [(4, X#=2), (3, Y#>Z), (2, X#>=Y), (1, X##Y)]

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      = [(4, X#=2), (3, Y#>Z), (2, X#>=Y), (1, X##Y)]
```

Figure 1: A trace of the execution of program `sorted([X, Y, Z])`, all events are present with attributes: event number, [depth], constraint C, domain of variables of C. Events #14 and #16 are displayed with all their attributes.

bound of the domain of 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 domain of x is {2, 3}, the domain of y is {2, 3} and the domain of z is {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 the domain of y is {3}, the domain of z is {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`}.

3 A trace analysis scheme

The trace previously defined is a sequence of events. In this section we describe a scheme to analyze this sequence. At a given moment of the analysis, there is always a particular event, called the *current event*, which is the one under scrutiny by the analysis. The events are considered one at a time. The analysis works on the current event and can then move the focus to another event. We focus our presentation on forward moves in the sequence of events. See section 3.6 for a brief discussion about backward moves.

As already mentioned in the introduction, we propose a trace query language which is Prolog augmented by some primitives. We introduce the primitives in the following. The examples illustrate both the specific functionality of the primitive and the benefit of using Prolog.

In the following, Section 3.1 presents the primitive which moves forwards in the trace and filters the events. Section 3.2 introduces two primitives to retrieve the attributes of the current event. Section 3.3 explains how to combine the previous three primitives to specify sophisticated search. Section 3.4 explains how the trace queries are processed. Section 3.5 argues that the presented scheme is a good compromise between efficiency and expression power. Section 3.6 discusses some pending issues and sketches future work.

3.1 Event filtering

To move forward in the sequence of events, we provide a primitive, called *fget*. This primitive also filters the events.

`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.

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 can either be `=` (unification) or `in` (an item is a member of a set.)

For example, the following query stops at the next event whose depth is 2.

```
:- fget(depth = 2).
```

The following query stops at the next reduce event where variable 'X' has just been emptied.

```
:- fget(port=reduce and updated_var=('X', _) and update=[emptied]).
```

The following query stops at the next event whose port is either select or reduce.

```
:- fget(port in [select, reduce]).
```

Thanks to the backtracking of Prolog, the following query displays an exhaustive trace. `fget(-)` retrieves the very next event; `print_event` displays all the attributes of the current event; `fail` forces the backtracking to `fget(-)` until the end of the trace is reached.

```
:- fget(-), print_event, fail.
```

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. 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. Namely, from the set of attributes described in Section 2, the `set of domains` and the `store` are not included in the `EventPattern`. 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. The attributes of `EventPattern` can be retrieved with the `current` primitive. The other attributes, the `set of domains` and the `store`, represent sets of items (they are called *set-attributes* in the following.) Retrieving the whole set in one go would be too costly. We therefore propose a second primitive, `current_data`, which retrieves set-attributes item by item.

`current(EventPattern)`: verifies that the attributes of the current event matches 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 set-attributes of the current event which matches the `DataPattern`.

If one solution is found, on backtracking `current_data(DataPattern)` will retrieve the next item of the set-attributes of the current event which matches the `DataPattern`.

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

The `Sub_attribute_names` are actually sub-attributes of the set-attributes. There are therefore different for each type of set attributes, except the `type` which specifies which set-attribute is concerned. For the `set of domains` the type of the items is `var`(iable) and the sub-attributes are `ident`(ifier), `name`, `domain`. For the `store` the type of the items is `const`(raint) and the sub-attributes are `ident`(ifier), `source`, `repr`(esentation), `context`.

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

For example the following 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`.

```
:- current(chrono=Ch and depth=4 and port=Po and constraint=Co).
```

The following query retrieves the name and the domain state of the 19^{th} variable:

```
:- current_data(type=var and ident=19 and domain=D and name=N).
```

The following queries verifies that there is a `diffN` constraint in the store and retrieves one of its identifiers.

```
:- current_data(type = const and ident = Id and repr = diffN(_,_,_)).
```

3.3 Sophisticated search

We have described earlier that the set-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 the following examples.

The following query stops at the next event corresponding to a tell of a `diffN` constraint where the 12th variable has a domain equal to `[1,2]`. When `fget` finds an event corresponding to a tell of a `diffN` constraint, `current_data` is called. It checks whether the 12th variable has a domain equal to `[1,2]`. If the verification fails, 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.

```
 :- fget(port = tell and constraint = (_,_,diffN(X,Y,N),_)),
    current_data(type = var and ident = 12 and domain = [1,2])
```

Any Prolog construction and mechanism can be used in the trace queries. For example, the following query stops at the next tell event. Then a "repeat-fail" construction is used to retrieve in turn each visible variable and its domain in order to display them. When no more variable are visible the query says so. Note that the information about variables and domains could be displayed in a more sophisticated way. This point is illustrated in the examples of section 4.

```
:- fget(port = tell),
   ( current_data(type = var and ident = N and domain = D),
     printf("%w :: %w \n", [N,D]),
     fail
```

```
:- fget(port = select and constraint = C),
    no_impact(C).

no_impact((CId, _, RepInt, _)) :-
    constraint_vars(CId, VarS),
    setof( (V,D),
           (member((_, V), VarS),
            current_data(type = var and ident = V and domain = D)),
           DS),
    no_impact(RepInt, DS).
```

Figure 2: A query which stops at useless select events

```
;
  printf("no more variable \n", [])
)
```

The query in Figure 2 stops at the next select event whose constraint has no impact on the current domains, namely where no domain will be reduced. Some solvers are optimized enough so that the case should never occur. For the other solvers, it may be important for the programmer to check that its labelling strategies do not induce too many such events.

This query uses a very powerful predicate, `no_impact/1` which can be provided with the traced system. We do not give all the details of its implementation; `constraint_vars(CId, VarS)` gives all the variables of a constraint and `no_impact/2` does the actual checking. We simply illustrate how the repeat-fail mechanism can be used to retrieve all the domains, here with the help of the `setof` primitive.

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 trace analysis (called LSD in the following). 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 3 illustrates how the filtering primitive works. Let us assume that the programmer wants to query the execution trace of the program given in Figure 1. 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`). 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 checks locally 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

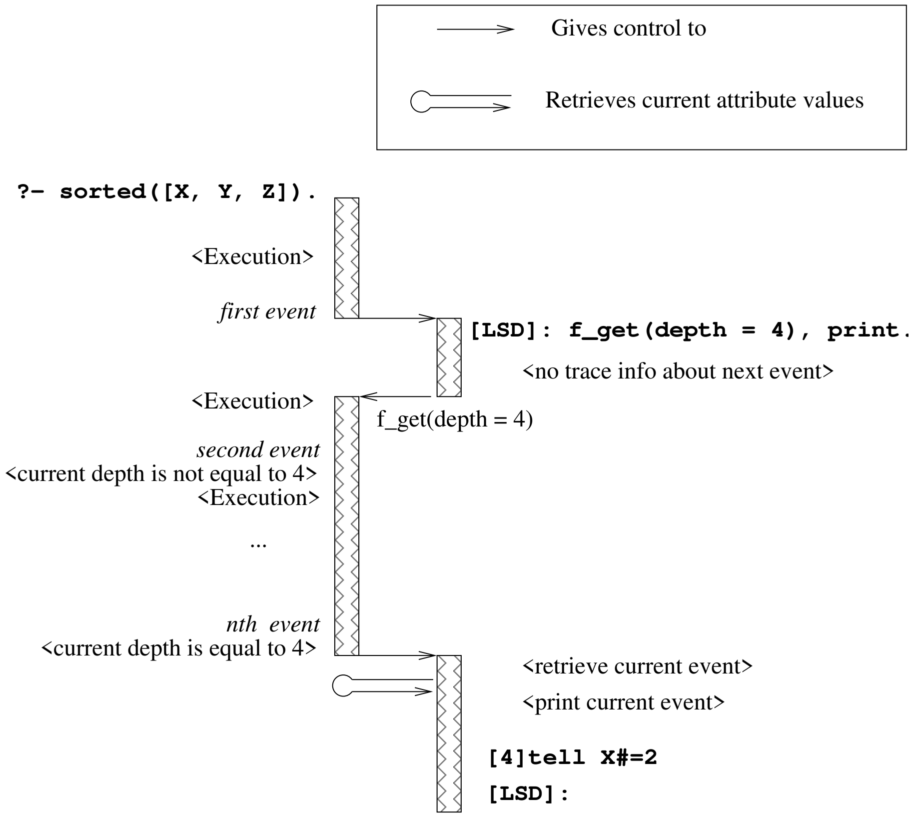

Figure 3: Illustration of the processing of a filtering query

requested one, the traced execution is resumed until the next event is reached. The depth is checked again locally. Forward moves and checking are done in turn until the first event whose depth is 4. 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

1. The attributes do not need to be "created" in any way, they are present in the traced execution context and they only need to be tested. A creation or copy is several orders of magnitude more expensive than a test.
2. 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, powerful search 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 set-attributes will be retrieved.

Thus, filtering is very efficient and the cost of sophisticated search depends of the amount of set-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 primitive tracers. The scheme is therefore already usable.

3.6 Further work

The presented scheme could be enhanced at least in three directions. In the following we hint that efficient filtering should also be done on set-attributes. We then briefly present the problem of backward search. Lastly, we sketch current work on the automated generation of trace analyzers.

Filtering on set-attributes In the current design of the scheme, if most of the search cannot be achieved by `fget` and if the data retrieved by `current_data` are large, then the analysis can become very slow.

It is therefore planned to study the extension of the filtering mechanisms such that they can take the set-attributes into account. The objectives are that the filtering of set data can also be done in the traced process in order to avoid useless context switches and information creation. This should be done without reducing the efficiency of the filtering algorithm on simple attributes.

Forward vs backward search As is illustrated in the following and in previous analyses for Prolog [7], much can be achieved by forward, "on the fly", analysis. However, sometimes, it is useful to move backwards in the trace.

We have shown that once a trace is recorded the previous scheme also works well backwards for the simple attributes [9]. The primitives `fget` and `current` simply retrieves their attributes from the stored trace instead of from the execution. However, several problems are still pending

1. Storing the set-attributes at each event is not possible. The amount of data is much too large.
2. Storing the simple attributes, even if more manageable, still takes a lot of time.
3. Moving backwards does not necessarily require to store the trace. The program could also be re-executed,

It is therefore planned to study the compatibility of the scheme with re-execution. If it is satisfactory we will then investigate partial re-execution.

Automated generation of trace analyzers One can notice that our trace analysis scheme is very generic. A Prolog process is connected to a tracer and exchanges messages of known types with it. The implementations of this connection to different tracers are very similar.

The `fget` primitive must, however, be tightly integrated in the traced process. Furthermore, the three primitives depends of the number of types of the event attributes.

We are currently working on a system which, from the specification of the trace attributes, will automatically generate the trace analysis process and its connection with the tracer.

4 Examples of clp(fd) trace analyses

In this section we present two examples of trace analyses written in Prolog extended by the primitives presented above: the reconstruction of a search tree and a graphical display of variable reduction. The computation of the first analysis is entirely achieved "on the fly". The second analysis proceeds in two stages: the first one is "on line", synchronized with the actual execution, it gives a first view which is stored in an intermediate file. The second one is "off line", it processes the data collected in the file. At each stage, the level of abstraction is increased.

4.1 Search-Tree Reconstruction

The search-tree is a classical view of a clp(fd) execution (see for example [18]). It abstracts the behavior of the search-procedure by representing the various choice points (the internal nodes of the tree), the failures and the solutions (the leafs of the tree). An arc represents a

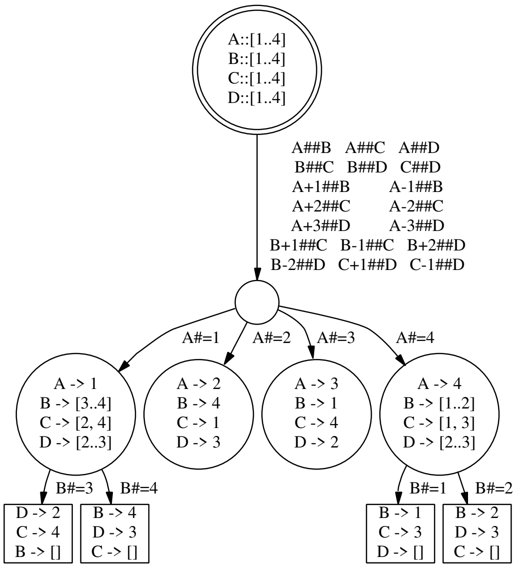

Figure 4: The Search-Tree of the 4-queens problem

choice leading to a new choice point or to a leaf. A choice is a constraint added by the search procedure in order to split the search space; for example, a given variable can be bound to a given value. A node abstracts all the propagations induced by a choice; a summary of these propagations is attached to the node.

Figure 4 shows the search-tree our analysis generates for an execution of a program solving the well known 4-queens problem. The program has a first-fail and minimum-first strategy. The root contains the initial state of the domains, its single son represents the state of the domains after the reductions due to the initial constraints (in this case, there is no reduction at all). Then, each arc is labeled by the corresponding choice and the resulting node contains the state of the updated domains after the propagation of the choice. The shape of the leafs distinguishes the solutions (circles) and the failures (squares).

Given that the search procedure applies a depth-first strategy, the execution trace describes a preorder traversal of the search-tree. This traversal is analyzed to perform the abstract search-tree. As is illustrated by Figure 5, during the search, the moves in the tree are reflected by the following ports: tell goes from the current node to one of its sons, told goes back to its father, reject and solution respectively mean that the current node is a failure or a success leaf.

The source code of the analysis is given by Figure 6. Firstly, the goal `search_tree(T)` retrieves all the variables and the initial state of their domains (`D1`, line #2), `D1` is the label of the root node. Then the analysis moves forward to the first tell of a constraint whose context is "`search`" (#3-4). This context results from an annotation of the search procedure code. The analyzer retrieves all the constraints in the store (#5) to label the `Arc` between the root and the choice point. The state of the domains just before the search (`D2`) is also

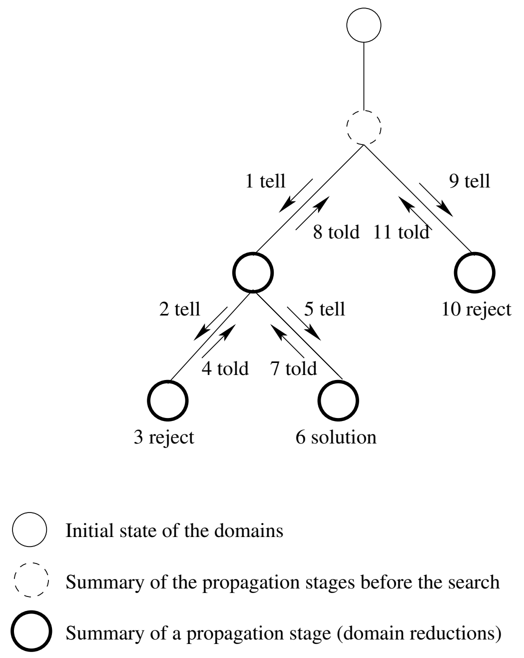

Figure 5: Trace events seen as the description of a tree traversal

retrieved (#6). The comparison between `D1` and `D2` (#7) gives the list of the domains that have been reduced since the beginning (`FirstRed`). This is the label of the first choice point. The remaining events describe the sons of this choice point. The sons are built by a `setof/3` calling the recursive predicate `build_node/1` (#8).

The predicate `build_node/1` is called at the tell of a choice C and exits at the corresponding told. It stores the state of the domains `D1` on this tell (#28) and goes to the event that ends the propagation stage, *i.e.* another tell, a reject or a solution (#29). It retrieves the domains again and deduces which ones were updated (`Updates`, #30-31). The type of the current event gives the type of the current node (a choice point, a failure or a success, #32-44). This type (`Kind`) and the updated domains make the label of the node (`NodeLabel`, #45). In case of a leaf, the recursion ends with an empty list of sons. In case of a choice point, the predicate is recursively called through a `setof` which builds the list of sons (#35).

Thus, the analysis generates a Prolog term containing the whole tree. This term is translated into a postscript file by the *dot* tool [12].

4.2 Generation of a 3D Variable-Update View

A second experiment is the generation of a complementary view a la TRIFID [3]. As illustrated by Figure 7 this view shows in three dimensions the evolution of the domains

```
search_tree(D1 - (FirstRed, Arc, SonList)):-
    get_all_domains(D1),
    fget(    port = tell
         and constraint = (_,_,_,search)),
    setof(C, current_data(type=const, repr=C), Arc),
   get_all_domains(D2),
   difference(D2, D1, FirstRed),
   setof(OneSon, buildNode(OneSon), SonList).

get_domains(ListOfVars, Domains):-
    setof( (V,D),
           (member(V, ListOfVars),
            current_data(    type = var
                         and ident = V
                         and domain = D)),
           Domains )
    -> true
    ;  Domains = [].

get_all_domains(AllDomains):-
    setof(V, current_data(    type = var
                          and ident = V),
          AllVariables),
    get_domains(AllVariables, AllDomains).
```

```
buildNode((ArcLabel, NodeLabel, SonList)):-
    current(    port=tell
            and constraint = (_,ArcLabel,_,_)),
    get_all_domains(D1),
    fget(port in [tell, reject, solution]),
    get_all_domains(D2),
    difference(D2, D1, Updates),
    current(port = Port),
    (Port = tell
     -> Kind = choice_point,
        setof(Son, buildNode(Son), SonList)
     ;
     Port = reject
     -> Kind = failure,
        SonList = []
     ;
     Port = solution
     -> Kind = solution,
        SonList = []
    ),
    NodeLabel = (Kind, Updates),
    fget(port = told).
```

Figure 6: Source Code of the Search-Tree Reconstruction

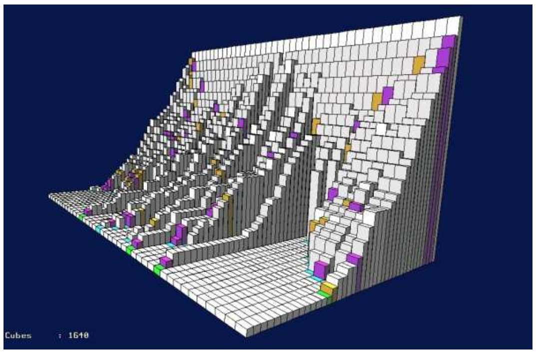

Figure 7: 3D Variable-Update View for the 40-Queens Problem

during the computation [1]. The vertical axis represents the domain sizes, initially each queen can take one among 40 positions. The horizontal axis going to the back represents the variables, 40 queens have to be positioned on the board. The horizontal axis going to the front represents the time. We can tell that the labelling strategy used in this execution is rather good because the domains decrease very rapidly and uniformly.

The analysis uses an intermediate text file: each row represents a given moment in the execution (the initial state, the state before each search-constraint addition and the state on each solution or failure). Each row consists of two columns per variable: the size of its domain at this point of the execution and an integer which says how the domain was modified in the last computation stage (for instance: no update, modification of its lower bound or domain emptied). Then this file is processed by a program written in C that we could reuse: each row in the first file is converted into a line of VRML boxes, each pair of integers gives the size and the color of one box. The final VRML file can be displayed by any VRML viewer.

The key idea of the trace analysis is to manage a flag for each variable. This flag says how its domain has been reduced by the last choice. Those flags are updated on each reduction (`reduce` event) and reinitialized when a row is dumped in the file. A flag only records the "most relevant" update found. For example, when the upper bound of a domain is minimized in such a way that the domain is reduced to a singleton, the most relevant update is *ground* (the variable is assigned). The possible updates, sorted by relevance, are "emptied" (the domain has become empty), "labeled" (the domain has been made ground by the search procedure), "ground" (the domain has been made ground by the propagation stage), "minmax" (both upper and lower bounds have been modified), "min" and "max" (only one bound has been updated), "any" (at least one internal value has been withdrawn) and "nothing" (no domain reduction). Those flags are managed through global variables. This system is accessed through three predicates: `reset_flags/0`, `update_flag(V,U)` which takes the update `U` into account for the variable `V` and `get_flag(V,F)` which gives the current value `F` of the flag of `V`.

The goal that performs the whole analysis is `upView(FileName)`. The source code is given in Figure 8. First of all, it opens a new file to output the result and initializes the flag system (lines #2 and #3). Then the execution is driven to the first search constraint addition (#5). A list of all variables and their domain is retrieved (#6 and #7). This initial state is dumped into the file by the `dump_domains/1` predicate (#8). A failure driven loop (#14-28) is used to process the remaining trace. On each relevant event (reduction, constraint addition, failure or solution) the execution is stopped (#15). If the current event is a domain reduction, the kind of update is sent to the flag system to be recorded. On a solution, a failure or a constraint addition the domains are dumped with the corresponding flags into the file by `dump_domains/0` (#23). The flags are then reinitialized. After the last execution event, `fget/1` fails and the intermediate file is closed (#30).

[1] Note that even if it is not visible here, colors have been introduced to display specific events as in the variable update view of [19].

```
upView(File):-
  open(File, write, updateView),
  reset_flags,

  fget(    port = tell
       and constraint = (_,_,_,search)),
  record_labeling,
  setof(V,
        current_data(type=var and identifier=V),
        AllVariables),
  get_domains(AllVariables, InitDomains),
  dump_domains(InitDomains),

  (fget(port in [reduce,tell,solution,reject]),
   current(port = Port),
    (Port = reduce
     -> current(    updated_var = Var
                and update = Update),
        update_flag(Var, Update)
     ;
        % Port in [tell, solution, reject]
        get_domains(AllVariables, DomainState),
        dump_domains(DomainState),
        reset_flags,
        ( Port = tell
          -> record_labeling
          ;  true ),
        fail)
 ;
  close(updateView)).

dump_domains(DomainState):-
  member((V,Dv), DomainState)
  dom_size(Dv, Size),
  get_flag(V, Flag),
  printf(updateView,"%d %d ",[Size,Flag]),
  fail
 ;
  printf(updateView, "\n", []).

record_labeling:-
  current(constraint = (Id,_,_,_)),
  const_vars(Id, [V]),
  update_flag(V, labeled).
```

Figure 8: Source Code of the 3D Variable Update View Analysis

5 Related work

A very detailed discussion of work related to our analysis scheme can be found in [9]. In the following we only recall the most important arguments.

Current state analysis We have emphasized the importance of the two primitives `current` and `current_data` which enable to retrieve parts of the current state. This is reinforced by Duel [10] whose main functionality is to enable users to investigate states of executions. They have designed and implemented an ad hoc interpreter, inspired by the Icon language, [11], with "generators" for expressions generating 0 or more solutions. We use Prolog which is satisfactory as such. We did not need to customize it.

Synchronous tracing and Storage Our filtering primitive has a flavor of database retrieval. However, it does not need to wait for an execution to finish in order to start to analyze it, as opposed to the systems which need an actual trace database: the Omega environment [17], the Yoda debugger [14], the IL system [4] and the Traceview system of Malony et al. [15]. In the four systems previously cited *all* events are first stored in some sort of database whereas in our scheme the analysis is done "on the fly", without any storage. Secondly, the *whole* information related to an event has to be computed, whereas in our scheme the costly parts are computed only if needed.

Debugging language Some programmable debuggers are based on a full programming language. For example, UPS designed by Bovey et al. [1], Cola designed by Dencker [5], Dalek [16] and Acid [20] offer C-like interpreters. However, a procedural language like C does not seem appropriate here. Only a few primitives need to be efficient. What is important is the prototyping power of the debugging language so that users can easily enter trace queries on the fly to adjust their diagnosis process. Prolog, with its expressive power, is better suited to this than procedural languages. Failure driven loops and unification help to express queries which otherwise would require very complicated control structures. The IL system of Cohen and Carpenter [4] provides regular expressions to specify debugging commands, which is more satisfactory than procedural languages but does not offer the power of a full programming language.

Adaptation of the scheme to clp(fd) The original scheme has been developed for Prolog [9], where no global data is available. It has been extended for C in order to take into account the global variables [8]. In this article we show that the `current_data` primitive is generic enough to accommodate both variables and constraints of clp(fd). Prolog with its sophisticated control flow, C with its sophisticated data and clp(fd) with its sophisticated data flow represent a wide range of paradigms. The success of the adaption of our trace analysis scheme to the three languages shows its genericity.

6 Conclusion

In this article we have presented a trace analysis scheme, successfully adapted to clp(fd). We have described two examples of non-trivial graphical abstract views of executions which could be implemented in a few lines of code with our scheme. This shows that our trace analysis scheme is generic. Once the trace analyzer generator will be operational, it will become very easy to share and port trace analyses from one platform to another.

Acknowledgment We would like to thank E. Zoumman who implemented the C procedures which generate the VRML code for the second analysis. Pierre Deransart made fruitful comments on earlier drafts of this article.

References

[1] 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.

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

[3] M. Carro and M. Hermenegildo. The VIFID/TRIFID tool. [6], chapter 10.

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

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

[6] P. Deransart, M. Hermenegildo, and J. Maluszynski. *Analysis and visualization tools for constraint programming*, volume 1870 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[7] 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.

[8] 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.

[9] 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.

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

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

[12] E. Koutsofios and S. North. Drawing graphs with *dot*. TR 910904-59113-08TM, AT&T Bell Laboratories, 1991.

[13] 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.

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

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

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

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

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

[19] H. Simonis and A. Aggoun. Search-tree visualisation. [6], chapter 7.

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

Contents