Coca: A Debugger for C Based on Fine Grained Control Flow and Data Events

Mireille Ducassé

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Coca: A Debugger for C Based on Fine Grained Control Flow and Data Events

Mireille Ducassé, IRISA/INSA

N ° 3489

Septembre 1998

THÈME 2

ISSN 0249-6399

Coca: A Debugger for C Based on Fine Grained Control Flow and Data Events

Mireille Ducassé, IRISA/INSA *

Thème 2 — Génie logiciel
et calcul symbolique
Projet LANDE

Rapport de recherche n3489 — Septembre 1998 — 20 pages

Abstract: We present Coca, an automated debugger for C, where the breakpoint mechanism is based on events related to language constructs. Events have semantics whereas source lines used by most debuggers do not have any. A trace is a sequence of events. It can be seen as an ordered relation in a database. Users can specify precisely which events they want to see by specifying values for event attributes. At each event, visible variables can be queried. The trace query language is Prolog with a handful of primitives. The trace query mechanism searches through the execution traces using both control flow and data whereas debuggers usually search according to either control flow or data. As opposed to fully "relational" debuggers which use plain database querying mechanisms, Coca trace querying mechanism does not require any storage. The analysis is done on the fly, synchronously with the traced execution. Coca is therefore more powerful than "source line" debuggers and more efficient than relational debuggers.

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

(Résumé : tsvp)

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

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

Coca : Un debogueur pour C basé sur des événements de flot de controle et de données à grain fin

Résumé : Nous présentons Coca, un débogueur automatisé pour C, où le mécanisme des points d'arrêt est basé sur des événements relatifs à des constructions du langage. Ces événements ont une sémantique, alors que les lignes du code source utilisées par la plupart des débogueurs n'en ont pas.

Une trace est une séquence d'événements. Elle peut être vue comme une relation ordonnée de base de données. Les utilisateurs peuvent spécifier exactement quels événements ils veulent voir en précisant des valeurs pour les attributs des événements. À chaque événement, les variables visibles peuvent être investiguées. Le langage d'interrogation de trace est Prolog augmenté par une poignée de primitives.

Le mécanisme d'interrogation de trace cherche dans la trace d'exécution en utilisant des informations à la fois sur le flot de contrôle et sur les données alors que les débogueurs effectuent habituellement leur recherche, soit en fonction du contrôle, soit en fonction des données.

Contrairement aux débogueurs totalement relationnels qui utilisent vraiment une base de données, le mécanisme d'interrogation de trace de Coca ne nécessite aucun stockage. L'analyse est faite à la volée, synchrone avec l'exécution.

Coca est donc plus puissant que les débogueurs dont les points d'arrêt sont des lignes du code source et plus efficace que les débogueurs relationnels.

Mots-clé : Génie logiciel, environnement de programmation, débogage automatisé, mécanisme d'interrogcation de trace, language de débogage, compréhension du comportement des programmes, outil de débogage.

1 INTRODUCTION

State-of-the-practice debuggers (e.g. GDB [14], UPS [2]) provide users with a set of commands which enable them to query the execution of programs in order to understand their behaviors.

Although useful and in daily use these tools could be dramatically improved. Indeed they are based on a breakpoint mechanism witch relates to source code *lines*. But source code lines have no semantics.

We propose a breakpoint mechanism which is based on events related to language constructs.

Concurrent and parallel language debuggers also use the notion of events but these events are mostly related to the synchronization of the different sequential processes (see for example [1] or [4]).

We propose a fine-grained set of events which can model the sequential processing of C programs.

Once executions are modeled into sequences of events, a second aspect of the state-of-the-practice debuggers can be improved, namely the querying mechanism at users disposal.

Indeed, the debugging process is quite complicated and many systems tend to provide a huge set of ad hoc commands. GDB, for example, has more than 200. Even if these sets are powerful, they can be very hard to cope with. In order to avoid this problem, a number of state-of-the-art debuggers provide a set of basic primitives with a generic language.

Some systems provide users with sophisticated mechanisms to investigate data structures and execution states (for example DUEL [8], Acid [16] or OOTQ [10]). These mechanisms are very useful for users. However, one of the most important problem in debugging is to search through a huge amount of events before being in the situation to investigate the relevant data. Very often users know some partial information about the data they want to investigate and would expect the debugger to take advantage of this information. In the previously mentioned systems, information about the data can hardly guide the search for relevant information.

We propose a trace query mechanism where control flow and data can be both used in synergy to search through the execution traces.

Some other systems uses relational querying mechanisms inspired from database languages. This is very powerful and the queries have a nice declarative flavor. The other advantage is the flexibility. Indeed, it is very easy to define new types of events just by introducing a new relation. To our point of view this technique is, however, currently too costly. Initial works in this direction were using real databases (for example Omega [12], Yoda [9]). The cost of creating a database is prohibitive. Indeed, in the "compile, test, debug, edit" cycle programs change very frequently. Hence the debugging data has to be generated often. Furthermore, an execution can be easily composed of millions of events. Users, unless desperate, cannot wait for a database to be created.

More recent works handle dedicated mechanisms without going to a real database, they either use files (eg Traceview [11], HY$^+$ [4]) or main memory storage (eg FLEA [3])). Our experience shows that any kind of storage can become a problem with "real size" programs.

We propose a trace querying mechanism which does not require any storage. The analysis is synchronous with the traced execution. The trace language has a relational flavor. It is not as declarative and flexible as a full relational one but it is a compromise which provides reasonable efficiency.

The trace query language is Prolog with a handful of primitives. Casual users, however, do not need to know much about Prolog in order to benefit from the advantages listed above. They basically only need to understand what is a predicate, and that the searching primitives can search further if subsequent predicates fail. Expert users, can of course, make full usage of Prolog, in particular to write extensions. The approach is quite similar the one of Emacs [13] where casual users only know the set of commands whereas expert users know how to program new commands in Lisp.

The proposed framework for C is an adaptation of Opium designed for Prolog [6]. The power of Prolog as a trace-query language has been illustrated in [5].

The adaptation to C was not straightforward for two main reasons. First, Prolog has no global variables and no problems with variable scoping in general. The data investigation was not such a problem as in C. Second, Prolog predicates have a much more uniform structure than C functions. Extracting events was therefore much easier for Prolog than it is for C. We address the two problems in the following.

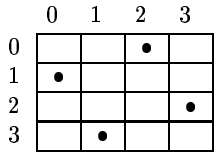
In the following, we first detail an example of debugging session to illustrate the power of the tool. Next we explain what are events for C executions. We describe the primitives and then the architecture of the system.

2 A DEBUGGING SESSION

We illustrate in this section the functionalities of Coca through an example of debugging session. We give intuitive explanations. A thorough description of the primitives can be found 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` which tests that a newly added queen does not attack queens already on the board.

```
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;
}
```

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 coca to gain some understanding of the behavior of `legal`.

2.2 Conventions

In the following, `[coca]` is the prompt of the debugger. Everything between each occurrence of `[coca]` and the following "." is a debugging query, typed in by a user. A query is a sequence of predicates. 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.

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 Coca finds one such event, asking for more will prompt Coca to search for the next such event. If a query asks for variable information, and Coca finds an accurate visible variable, asking for more will prompt Coca to search for the next accurate visible variable. When the search space is exhausted Coca answers `no (more) solution`.

2.3 Commented session

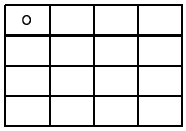
Let us assume that we have traced the execution until event number 18 which corresponds to entering function `legal`. If we want to see which variables are visible at this point we can use the `current_var` primitive asking for names, values and types, using the logical variables `X`, `V` and `T` as follows.

```
  (18) enter function legal

[coca] current_var(X, 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_var` primitive. In the following query, we retrieve the current values of `board` and `lineNb`, and call `display_board/2` with the proper parameters.

```
[coca]  current_var(board, val=B),
        current_var(lineNb, val=L),
        display_board(B,L).
```




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

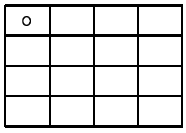
We can foresee that the previous query will be very useful and make a new command out of it, called `display_board/0`.

```
[coca] new_command.
```

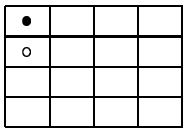
```
display_board :-
      current_var(board, val=B),
      current_var(lineNb, val=L),
      display_board(B,L).
```

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: "retrieve the next event of type `function`, whose function is `legal`, whose port is `exit`, and where variable `ok` has value `1` (ie the disposition is accepted), then give the value of `chrono` and display the accepted board. There are several such events, typing `yes` asks for the following one. Subsequently, the execution goes on along with the trace.

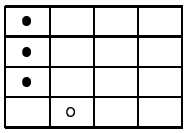
```
[coca] fget(type=function and func=legal and
            port=exit and chrono=Chr),
       current_var(ok, val=1),
       display_board.
```



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



```
Chr = 69     More? yes

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

Chr = 143     More? yes
```



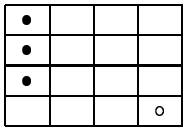
```
Chr = 648     More? yes
```

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. *We have edited out 2 displayed boards where the queens where in the same column.*

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

```
[coca] fget(type=function and func=legal and
            port=exit and chrono=Chr),
       current_var(ok, val=0),
       display_board.
```

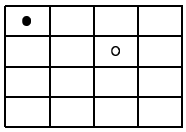


```
Chr = 1674     More? yes
```

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.*

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

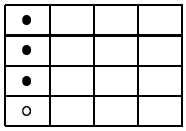


```
Chr  = 3519
OK = 0     More? yes
```

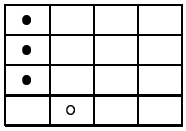
The disposition is rejected. The test inside the `for` loop of `legal` seems lunatic. However, when analyzing its source code, we cannot understand where the problem is.

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. In order to get a clear idea we first re-run and re-trace the execution from the beginning.

```
[coca] retrace.
[coca] fget(func=legal and
            ((type=for and port=enter) or
             (type=if  and port=exit))),
   ( if   current(type=for)
     then display_board,
          current_var(lineNb, val=Lin),
          current_var(colNb, val=Col)
     else current_var(i, val=I),
          current_var(board(I), val=BdI),
          current_var(ok, 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.

2.4 Remarks

Events are easier to handle than source lines. For example, requesting to see the `exit` of function `legal` is straightforward in Coca. Using breakpoints on lines, one would have to figure out where to put a breakpoint in order to see what is happening just before (or just after) the function exits. In so doing one is distracted from his main stream of thought.

Syntax	Event name	::=	Event components
for ($EXPR_1$; $EXPR_2$; $EXPR_3$) BLOCK	FOR_EV	::=	(enter, for) $INIT\_EV_1$ $TEST\_EV_2$ {BLOCK_EV $INCR\_EV_3$ $TEST\_EV_2$}* (exit, for)
if ($EXPR_1$) $BLOCK_2$ else $BLOCK_3$	IF_EV	::=	(enter, if) $TEST\_EV_1$ THEN_ELSE_EV (exit, if)
	THEN_ELSE_EV	::=	(enter, then) $BLOCK\_EV_2$ (exit, then) | (enter, else) $BLOCK\_EV_3$ (exit, else)
EXPR	EXPR_EV	::=	(enter, expr) (exit, expr)

Figure 1: Extract of the grammar of events

As shown by the conjunction of `fget` and `current_var`, searching in the trace is done according to control flow **and** data criteria.

For experienced users, an alternate and more compact syntax can be offered for the frequent case where the attributes are all connected with `and`. For example `fget(type=function and func=legal and port=exit and chrono=Chr)` could be specified `fget(function, exit, legal, -,Chr,-,-,-,-)`.

The exhibited session could be done in two passes. This illustrates the fact that much can be done without storing the trace.

3 EVENTS

A trace is a sequence of events, which correspond to C constructs. They therefore have a better semantics than source lines.

3.1 A Grammar of Events

A grammar of events specifies which events are associated to C constructs. Figure 1 shows examples of events associated to two constructs: `for` and `if`. A `for` has four components : an initialization, a test, an iteration and a block of action. 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 sequences related to the execution of the block, the iteration and the test, and 5) exiting the `for`. An example sequence of basic events generated for an occurrence of `for` loop is detailed in Section 6. The full grammar is about one page long.

Expressions are not detailed. However complicated they are, only two events will be generated: one before its evaluation, one after. If users are puzzled by the evaluation of a particular expression, there exists an expression evaluation simulator which can be decomposed at will. This simulation is actually straightforward to implement with Prolog and `current_var`.

3.2 The Event Structure

Each event is a structure containing 8 slots referred to by the `fget` and `current` primitives. Note that as nothing is stored this structure does not appear as such in the code of Coca. It is a *conceptual* structure.

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`, `create`, `expr`, `del`, `test`, `incr`, `init`.

The `create`, `expr`, `del`, `test`, `incr`, `init` correspond to data handling.

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`, `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 for recursive calls.

line: in the source file where the event is defined.

file: in which the encompassing function is defined.

For example, the complete representation of the event corresponding to the last board displayed in the previous debugging session is:

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

The execution is entering a `for` loop in function `legal`.

We do not claim that this structure is complete, it may be that we have to add new slot in order to cover new analyses. In particular some coordinate to be able to distinguish between different constructs of the same type inside a function could be quite useful.

4 DATA

Each variable is represented by a structure containing 7 slots. Here again, it is a *conceptual* structure only.

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

linedecl: line where the variable is declared

filedecl: file where the variable is declared.

For example the values of all the attributes of the first visible variable of event 626 are

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

5 PRIMITIVES

With the execution modeled as a sequence of events three primitives and Prolog are sufficient to query the trace on the fly. A pointer in the sequence of events corresponds to the "current event". This pointer can be moved forwards by a dedicated primitive, `fget`. The contents of the current event can be checked or retrieved by `current`. Descriptions of variables can be checked or retrieved by `current_var`. In the remaining of this section we describe in some depth the functionalities of those three primitives, the next section gives some details of their implementation.

fget(EventPattern) moves the trace pointer forwards through the sequence of events, until either the first event which matches the specified pattern or the end of the trace history is encountered. In the first case it succeeds, while in the second case it fails.

The pattern can be either - or any combination of couples of the form <attribute_name> <op> <attribute_value> connected with `and`, `or` or `not`. The possible attribute names are those of the event structure described in the previous section, namely `type`, `port`, `func`, `chrono`, `cdepth`, `line`, and `file`. The possible operators are = and <> for all types of attributes and <, <=, >, >= for integer attributes.

If an attribute value is:

- instantiated: the attribute value of the current event is checked against the required value(s).

- a logical variable: the attribute is retrieved when the checking of the instantiated attributes is successful.

The very next event can be reached by `fget(-)`.

current(EventPattern) checks or retrieves the values of the current event attributes according to the specified pattern. The pattern description is the same as for `fget`. If the attribute verification fails `current` fails, otherwise it succeeds and instantiates the requested logical variables.

current_var(Name, VarPattern) checks or retrieves, according to the specified pattern, the values of the attributes of the variable(s) visible at the current event.

The pattern can be any combination of couples of the form <attribute_name> <op> <attribute_value> connected with `and`, `or` or `not`. The possible attribute names are those of the data structure described in the previous section, namely `type`, `val`, `addr`, `size`, `linedecl`, `filedecl`. The possible operators are the same as for `fget`. `Name` as well as the attribute values can be either instantiated or logical variables. Values of structured data can be partially instantiated (as illustrated by `current_var(board, val=array(0,2,-,-))`). Pointers can be dereferenced, indexes and structures fields can be retrieved.

`fget` and `current_var` are re-satisfiable, they search for the next event or variable which matches the specified pattern on backtracking. This feature inspired by the "repeat-fail" mechanism of Prolog is very powerful. Users specify one event or variable pattern; when they have seen what they wanted they can either stop this particular search (type "return") or go on (type `yes`). They do not need to worry about how many they want to see at the time the query is specified.

Five more primitives are necessary to initialize and stop the trace.

start(Hostname) opens a GDB process on a possibly remote host.

stop kills the GDB process.

trace(Executable) runs a given executable file under GDB. Stops at the very first event. `start/1` and `trace/1` can be easily combined to provide `trace/2`:

```
trace(Host, Exec) :- start(Host), trace(Exec).
```

retrace reruns the currently traced executable. Stops at the very first event.

notrace finishes the execution without any further tracing.

Thereafter all the commands that can be proposed to users are simply a combination of these **height** primitives. Our experience with Opium shows that the possibilities of new commands are countless [5]. It is, however, very comforting for users to know that all they really need to master are these 8 primitives. Especially when one compares to existing debuggers where one does not know where to start with the commands.

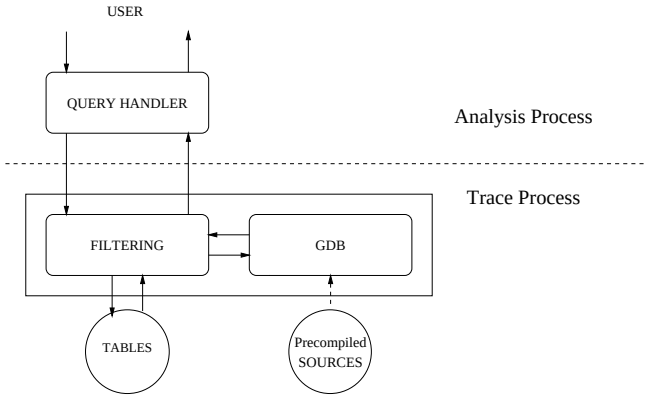


Figure 2: Architecture of the run-time system

6 SYSTEM DESCRIPTION

We view a debugger as conceptually composed of three distinct modules [7]: 1) **extraction** which builds a (possibly low-level) trace, 2) **analysis** which provides user-oriented trace query mechanisms and higher-level views of executions, and 3) **visualization**.

Coca concentrates on the second module, namely on building powerful trace analysis facilities. It is built on top of GDB [14], reusing its low-level trace extraction facilities.

GDB has no notion of events, we therefore start by transforming the initial source such that source lines better match events. We can then use the line-oriented breakpoint mechanism of GDB. We call this step *precompilation*.

Figure 2 shows the architecture of the run-time system. GDB and the query handler (ie a Prolog interpreter) are in two different processes which enables them to run on two different machines. It is out of question that inter-process communication takes place for each event. We have therefore designed a filtering procedure tightly coupled with GDB. The filtering procedure uses some static information not available via GDB and generated by the precompilation. GDB executes the precompiled code.

In the following we describe further the design of the precompilation and of the run-time system. We then give some implementation details of the current prototype.

6.1 Precompilation

As mentioned above, the precompilation step has two objectives: 1) prepare the source code such that the line-oriented breakpoint mechanism of GDB can be used; 2) generate tables containing static information not available via GDB.

The preparation for GDB faces two problems. First, lines and events do not map. Second GDB does not stop at execution points where we consider that an event occurs, for example

Initial source code	line	Precompiled	Simplified table of events
for(a= 1;a< 3;a++){	1	a = 1 ;	(enter,for) (enter,init)
printf("%d",a);	2	STOP;	(exit,init)
}	3	for (; a < 3 ;){	(enter,test)
	4	printf ("%d", a) ;	(exit,test) (enter,block) (enter,expr)
	5	a ++ ;	(exit,expr) (exit,block) (enter,incr)
	6	}	(exit,incr)
	7	STOP ;	(exit, test) (exit,for)

Figure 3: Example of precompiled code and associated events

at the end of an `if` construct. We therefore re-arrange source lines such that a particular events fits on a single line, with possibly several events per lines. Further we introduce a void construct, called `STOP` in the following, to force GDB to stop at places it would not naturally do so. This construct does not change the operational semantics of the traced program. It is currently implemented by `if(0)`.

Figure 3 shows one examples of such transformations for a `for` loop. The initialization of the iteration variable has been moved before the body of the loop while its incrementation has been moved to the end of the body. A `STOP` has been added line 2 to mark the end of the initialization which could not be at the beginning of the for, line 3, as the execution will come again on that line when iterating. When the test succeeds its exit is marked at line 3, when it fails at line 6. STOP, at line 6, forces GDB to stop after the exit of the loop.

The static information is generated in:

the event table: it contains the list of events associated with each line of the precompiled source code. For each event the values of the attributes which can be determined statically are stored, namely `type`, `port`, and `func`; `chrono` is generated by the filtering mechanism, `cdepth` will be given by GDB; `line`, `file` are also given by GDB and are the indexes of the event table. A simplified event table is shown in Figure 3.

the symbol table: it contains all the static information related to variables. For each variable the values of the attributes which can be determined statically are stored, namely `name`, `type`, `linedecl`, `filedecl`. `val`, `addr` and `size` will be given by GDB.

the variable visibility table: for each line of the precompiled code, it enables the list of visible variables to be obtained. This information is essential for `current_var`. The variable scoping problem mentioned in the introduction is therefore addressed by the precompilation.

the line mapping table: it enables to map a line of the precompiled source to a line interval of the initial source. This is essential in order to go back to the original code. The interval allows to capture whole constructs.

6.2 Run-time Analysis

When a user issues a query, this query is parsed by the query-handler and primitive queries are sent over to the filtering procedure. The latter then interacts with GDB and the tables containing the static information.

The filtering procedure has three main components: 1) breakpoint preparation, 2) pattern matching, 3) backtracking. Breakpoint preparation asks GDB to set a number of breakpoints on lines which could possibly correspond to events requested by a `fget`. Pattern matching checks the attributes of events or variables. Backtracking moves to the next breakpoint (`fget`) or the next visible variable (`current_var`). In the following we give more details of each of those components.

Breakpoint preparation When a `fget` query is sent over to the filtering procedure, it, first, computes a superset of lines which may correspond to the requested events, and sets GDB breakpoints to them. Indeed, using the `step` command of GDB to move systematically to the very next event is too costly [1]. We, therefore, use the breakpoint mechanism of GDB, and try to limit the number of places where pattern matching is done.

A query can be seen as a logical tree where the nodes are logical operators and the leaves correspond to a single attribute checking. For each leave we compute a superset of corresponding lines. Some attributes are very constraining (eg names of function) while others are no so informative, especially when set alone (eg ports). Furthermore, two attributes are totally dynamic, `cdepth` and `chrono`, and the search cannot be apriori constrained upon their values. Hence the set of breakpoints can only be a superset. The used algorithm guarantees that no left out lines could correspond to the requested events.

Then for and-nodes we perform an intersection, for or-nodes an union. For not-nodes we cannot perform a simple complementation as we would not get supersets. In a first stage, we have chosen in that case to resort to a full tracing.

Therefore, when the query is very constrained we set breakpoints which enable to restrict the search; when the query is not so constrained we use the basic step-by-step. This compromise is a reasonable one. Indeed, the algorithm is simple and cheap to execute. Furthermore, in the case where the less efficient mechanism is used, there are many events which correspond to the query and therefore the user does not have to wait long before an accurate event is found.

Pattern matching The pattern matching algorithm performed either at each new event found (`fget`), or at the current event (`current`), or on the currently visible variable (`current_var`) is what one can expect.

The only point worth mentioning is that the filtering procedure retrieves only the attribute values necessary to process the requested pattern matching. This retrieval is simply done by call to dedicated procedures of GDB or by investigating the static tables. Note that *no extra dynamic information needs to be stored.* This feature is one of the keys of the reasonable performances of Coca.

[1] Note, however, that using the `step` command of the tracer used in Opium gives reasonable performances.

Backtracking If the pattern matching succeeds, the filtering procedure sends the results back to the query handler. If it fails the filtering procedure searches again until either the pattern matching succeeds or the end of the search is encountered. For `fget` the latter occurs when the end of the execution is reached; for `current` it occurs as soon as the pattern matching fails; for `current_var` it occurs when no more variables are visible.

Note that inter-process communication only takes place when a primitive query is satisfied. For simple trace queries, this means that there is only a very limited number of such communications. This is another key of the reasonable performances of Coca.

Very sophisticated complementary filtering can be programmed within the query-handler module [2]. In such a case there are, of course, more inter-process communications and the response time can become an issue. We are, indeed, working on optimizing further the implementation of user queries. Note, however, that the time taken by Coca to find events is always much more straightforward and faster than doing it by hand with a low level debugger.

6.3 Modifications of GDB

The main problem is that GDB has not be designed to pass the extracted trace to other programs but to display parts of the trace as soon as they are extracted. The first step was to make GDB quiet, as all the `printf` scattered in the procedures would take a significant amount of time if they were executed which would make the performance collapse. One essential principle of Coca is to avoid as much as possible run-time printing (whether to external devices or in internal variables or databases). The next step was to change the functions which extract information such that they would return the requested information.

6.4 Implementation Details

A prototype is operational under SunOS 5.6 Solaris, with GCC and GDB, as well as the Eclipse Prolog system 3.5.2. The precompilation is achieved by modifying the Bison grammar of the GCC compiler. The interprocess communication is implemented using RPC functionalities.

The source code of Coca is composed of approximatively 15000 lines of code:

- precompilation: 2500 lines of C, 2100 lines of bison grammar.
- run-time: 5400 lines of C, 2300 lines of ML, 1000 lines of Prolog
- GDB: 400 lines added and less than 50 lines modified among the 221800 (!) existing ones.
- Expression evaluator: 1100 lines of Prolog

There is a fair amount of comments included in those numbers.

[2] For example, in Opium we have programmed an endless loop analysis which accounts for 3000 lines of Prolog

7 DISCUSSION

In this section we compare Coca with two systems which we believe are the closest to Coca, DUEL and CCI. We then discuss the use of GDB for the implementation of Coca.

DUEL [8] is a system also built on top of GDB which enables to explore states of C execution in a powerful way. It provides a language to specify expressions to be evaluated. These expressions are a superset of C language. Duel implements, in particular, "generators" which are very close to our backtracking facilities. As we have already mentioned Coca also implements an expression evaluator using the power of `current_var` and of Prolog. Our design enables two features which are mentioned as "further work" in the above cited DUEL article: 1) the expression evaluation is tightly connected with a breakpoint facility; 2) partially instantiated attributes unable a wider search.

CCI [15] is a configurable automatic instrumentation tool for C. It gives a generic framework to build monitoring systems. One can specify a set of events that are of interest and CCI generates a new C file which contains "enodes" which can be seen as breakpoints. Their set of possible events is larger than what is offered by Coca. They offer "masks" to enable events to be generated only if their are needed. This filtering is **static** the requested events are known before the execution. CCI does not offer the dynamic filtering procedure of Coca.

As a side remark, the two processes of Coca enables meta-debugging to take place. When expert users write debugging programs in Coca (to debug C programs) they basically write Prolog programs which in turn can be debugged using Opium.

CCI authors claim that a two-process implementation cannot be efficient enough. This claim is not invalidated by Coca, in spite of its two processes. Indeed, CCI is mostly concerned by what we called the extraction phase in section 6 which corresponds to our connection with GDB which is indeed in the same process as the filtering procedure.

Using CCI could be an alternate solution for Coca implementation replacing both the pre-compilation and the connection to GDB. It would provide with more flexibility. Checking that the performances are sufficient is further research.

Indeed, using GDB has advantages and drawbacks. The main advantages are that GDB is a solid software, freely available and widely used. It also supports many languages and architectures. The main drawbacks is that it has not be designed to be connected to a trace analysis tool. The current modifications in Coca make a functioning system but benchmarks suggest that the basic functionalities we are using may not be efficient enough for our purposes, especially when compared with the behavior of the tracer used in Opium. It would be interesting to redesign the internals of GDB such that the required basic functionalities would be used optimally but the more than 220000 lines of code are a threat.

8 CONCLUSION

We have presented Coca, a trace analyzer for C programs. Coca is more powerful than "source line" state-of-the-practice debuggers because it is uses semantics-based events instead

of lines. It is also more powerful than state-of-the-art debuggers because it uses both control flow and data information in synergy. It is more efficient than relational debuggers because it does not request any information to be stored at run-time. It is more comfortable for users because they only need to master height primitives (five of which are obvious) and some logics.

Coca is connected with GDB a state-of-the-practice tracer for C freely available and running on many architectures. The performances of Coca are reasonable because 1) breakpoints are used to restrict the search for events, 2) only necessary information is retrieved, and 3) inter process communication is restricted by a filtering procedure running within the GDB process.

ACKNOWLEDGMENTS

I would like to thank Serge Le Huitouze for enlightening discussions and Remi Douence for helpful comments on this article. A number of undergraduate students at the Institut National des Sciences Appliquées de Rennes have participated in this project. I am especially grateful to Arnaud Hallais who worked out the implementation details of the fine integration with GDB, and to Sébastien Ferré who prepared the demos, designed and implemented the breakpoint preparation algorithm, and fixed hundreds of problems.

References

[1] P. Bates. Debugging heterogeneous distributed systems using event-based models of behavior. In *Workshop on Parallel and Distributed Debugging*, pages 11–22, Sigplan Notices, Vol 24, Number 1, January 1989.

[2] J. Bovey, M. Russel, and O. Folkestadt. Direct manipulation tools for Unix workstations. In *Proceedings of the EUUG Autumn'88*, pages 311–319, October 1988.

[3] D. Cohen, M. S. Feather, K. Narayanaswamy, and S. Fickas. Automatic monitoring of software requirements. In *Proceedings of the 19th International Conference on Software Engineering*, pages 602–603. IEEE Press, 1997.

[4] 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.

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

[6] M. Ducassé. Opium: An extendable trace analyser for Prolog. *The Journal of Logic programming*, 1999.

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

[8] M. Golan and D. Hanson. DUEL- A very high-level debugging language. In *Proceedings of the Winter USENIX Technical Conference*, 1993.

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

[10] R. Lencevicius, U. Hölzle, and A. Singh. Query-based debugging of object-oriented programs. In *Proceedings of the OOPSLA Symposium.* ACM, October 1997.

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

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

[13] R. Stallman. Emacs: The extensible, customizable, self-documenting display editor. *Sigplan Notices*, 16(6), June 1981.

[14] R. Stallman and R. Pesch. *Debugging with GDB, the GNU Source-level debuger.* The Free Software Fondation, Inc, 4.09 edition, April 1993.

[15] K. Templer and C. Jeffery. A configurable automatic instrumentation tool for ANSI C. In *Proceedings of the Automated Software Engineering Conference*, 1998.

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

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

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