Efficient Automated Trace Analysis: Examples with Morphine

Mireille Ducassé [1]

IRISA/INSA
Campus Universitaire de Beaulieu
F - 35042 Rennes Cedex, France

Erwan Jahier [2]

IRISA/IFSIC
Campus Universitaire de Beaulieu
F - 35042 Rennes Cedex, France

Abstract

Opium, Morphine and Coca are three automated trace analyzers based on the same principles for three different programming languages. An automated trace analyzer is connected to an event-oriented tracer. The traced program is run in coprocessing with the trace analysis session in which the user enters high-level queries about the traced execution. The trace is then automatically processed according to the query. The key of efficiency is that most of the work is done (1) on the fly and (2) in the traced process. In this article, we first present these mechanisms and then illustrate them through a debugging session and monitoring examples with Morphine.

1 Introduction

Opium [5], Morphine [12,11] and Coca [4] are three automated trace analyzers, for three different programming languages: respectively Prolog [18], Mercury [16] and C. They are based on the same principles. An automated trace analyzer is connected to an event-oriented tracer. The traced program is run in coprocessing with a trace analysis session in which the user enters high-level queries about the traced execution. The trace is then automatically processed according to the query. Our debuggers offer flexible and powerful query mechanisms. The evaluation of these queries has a very small overhead

[1] Email: ducasse@irisa.fr
[2] Email: jahier@irisa.fr

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

when compared to hard-coded counterparts. The key of efficiency is that most of the work is done on the fly in the traced process.

In following, we illustrate the fact that the principles are not dependent of the traced language by presenting them on one language and giving the examples on another one. section 2 briefly introduces the basic mechanisms on the C language. The debugging session of Section 3 illustrates the presented mechanisms with Morphine on a buggy Mercury program which plays the well known mastermind game. In particular, for a simple query, more than one million trace events can be filtered within a couple of seconds. Section 4 illustrates the use of Morphine to implement monitors.

These examples should be understood by people that have no knowledge of Mercury or Prolog. We paraphrase all the programs and therefore believe that no previous knowledge of logic programming is required to follow the debugging session and the monitoring examples.

2 Principles of Opium, Morphine, and Coca

In this section we present on the C language the basic mechanisms of the three systems.

Events vs Line breakpoints

Debuggers such as GDB [17] or UPS [2] offer two basic mechanisms: line-oriented breakpoints and inspection of variables.

Line-oriented breakpoints are easy to implement but they do not correspond to much semantics. For example, if the debugger stops at a line such as

```
{ for(i=0, i++, i==20) { g(i) } }
```

where is the execution ? After entering the new block ? Inside the `for` loop ? At the exit of the loop ? At the exit of the evaluation of the `g` function ? Inspecting the value of the variables at this breakpoint is thus meaningless because it is hard to know where exactly the execution is positioned. This is, to our point of view, the major problem of the DUEL system [7]. DUEL enables to explore states of C executions in a powerful way, by providing a language to specify expressions to evaluate. However, as DUEL relies on line breakpoints, the accuracy of the responses is questionable.

An alternative to line breakpoints is event-oriented breakpoints, attached to syntactical constructs of the traced language. For example, the previous line can be decomposed into several events: `<enter block>`, `<enter for>`, `(<incr for>, <test for>`, `<enter function g>`, `<exit function g>)*`, `<exit block>`. Inspecting the variable values at precise event is meaningful because one knows exactly where the execution is positioned.

Events traditionally have attributes associated to the breakpoints [1,3,6,8]. These attributes can be seen as a tuple of information relative to the execution.

For example, in Coca, at event <enter for> some meaningful attributes are:

- encompassing function: the function in which the `for` loop is used.
- encompassing block: the block in which the `for` loop is defined.
- source file: in which the `for` loop is defined.
- line numbers: the numbers in the source file corresponding to the definition of the `for` loop.

Variable information

Inspection of variables is very important. In our debuggers, the information about variables is also modeled into a structure, dynamically valued. For example, in Coca some variable attributes are:

- name: variable identifier
- value: value of the variable at the current event
- encompassing function: function in which the variable is currently used.
- source file: in which the variable is declared.
- line number: the number in the source file corresponding to the declaration of the variable.

Connection to a tracer

It is not the aim of this article to describe in detail how to implement an event-oriented tracer. In the following paragraphs we rather concentrate on trace analysis which is the common part of the three systems.

In order to get the trace information, Opium is connected to the tracer of the Eclipse Prolog system [9]. Morphine is connected to the tracer of Mercury [15]. Both tracers are event-oriented. For Coca, a possible choice could have been the Alamo system [13,19]. This system, however, is not yet in a stable enough state. We have therefore implemented an event oriented tracer by source to source transformation.

The tracers implement mechanisms which enable executions to stop at each event and to retrieve the attribute values. Then at each event they call the automated trace analyzer. When the automated tracer requires more trace information the tracer is resumed.

Trace analysis: three essential primitives

A trace is thus a sequence of events, each event is a tuple of attributes representing execution information. As a consequence, a trace is a data structure which can be automatically analyzed. In particular we will show how it can be efficiently filtered.

Our three systems provide 3 basic primitives which enable powerful trace analysis:

current: to get the values of the current event attributes.

current_var: to get the current value of the attributes of the currently visible variables.

fget: to search the sequence of events for an event whose attributes match a given filter.

With these primitives, one can specify very precise queries about the traced execution. For example, one can ask to stop the execution when the execution reaches entry points of function `g` and when value of variable `i` is equal to `18`:

```
fget(port = enter and type= function  and function = g)
current_var(name = i and value = 18).
```

In our three systems, the queries are specified using the Prolog syntax and semantics. Processing the queries is modeled by the search mechanism of Prolog. For example, for the previous query, the search engine will search forward in the sequence of events until the first event which corresponds to entering function `g`; then the value of variable `i` will be retrieved; if it is different from `18`, the engine backtracks and searches for another event which corresponds to entering function `g`; the value of `i` is checked; and so on until either the end of the execution is reached, or an event is found such that the value of variable `i` is `18`.

Coroutining between the traced program and the debugging session

In most debuggers, the debugging commands are executed as subroutines. At each breakpoint a debugging command is entered and executed; when it is terminated the traced program execution is resumed.

The queries of our systems require that the traced execution and the debugging session run in coroutine. This is illustrated by Figure 1. The traced execution proceeds until it reaches the first event; then it gives the control back to the debugging session in which the user can enter a query. This query is typically a sequence of commands. In the figure we use the same query as before. The debugging session then sends the `fget` part of the query to the traced program. At each event, a filtering procedure is called which checks whether the current attributes match the filter (namely, here, whether the event corresponds to entering function `g`). The traced execution proceeds until the first matching event. Then the hand is given back to the debugging session which retrieves the current value of variable `i` and checks whether this value is equal to `18`. When this is not the case the execution backtracks and gives the hand back to the traced execution. When an event matches and the value of `i` is equal to `18`, the query succeeds and the user can enter another query.

Note that the query cannot be called as a subroutine. Firstly, `fget` can find several matching events on backtracking. Secondly, there could be several `fget` commands in a query, and the commands called in between them could

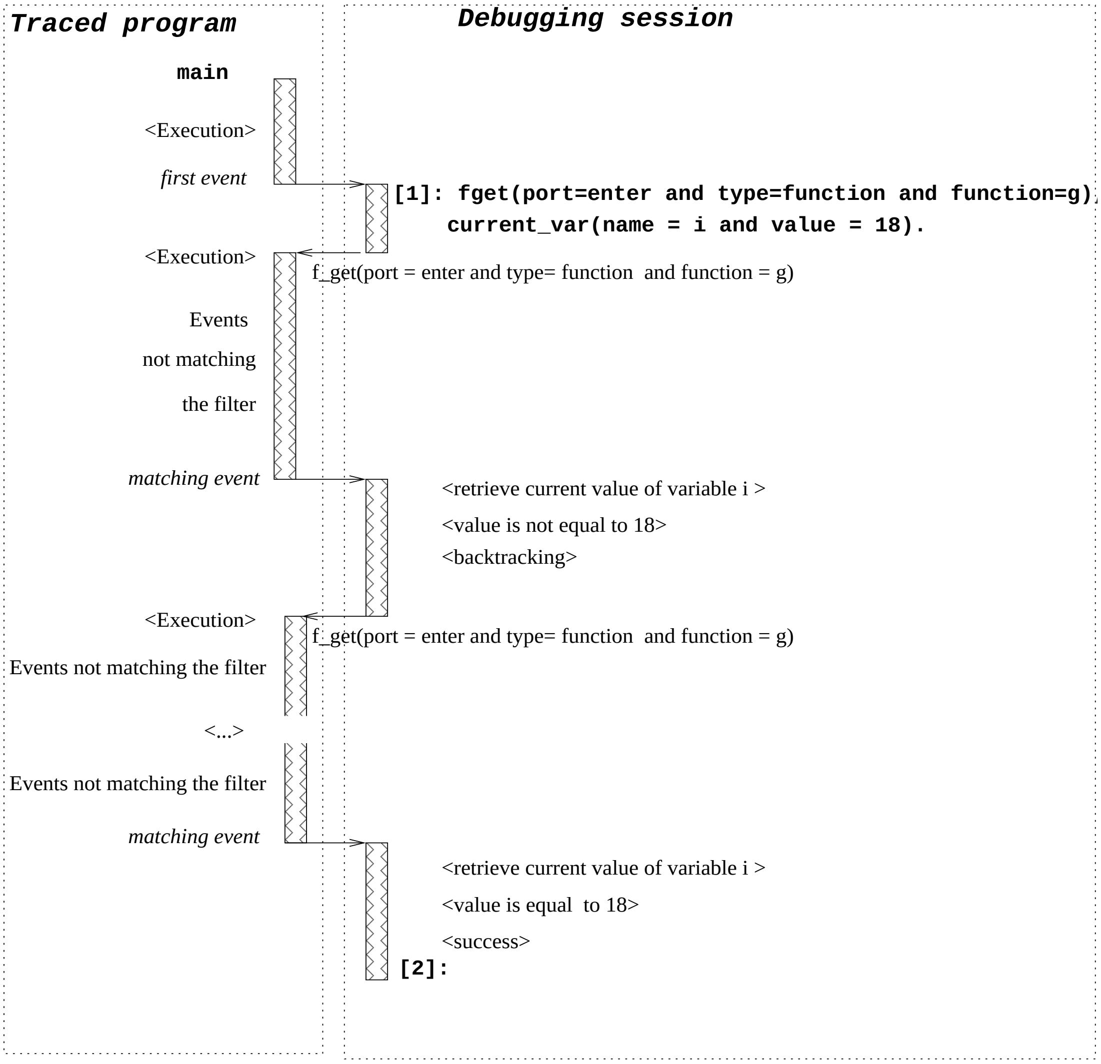

Figure 1. Coroutining between the traced program and the debugging session

be arbitrarily complex Prolog programs. It is therefore important to keep the context of execution of the debugging query when the traced execution is resumed.

A two process architecture

In order to implement coroutining between the traced program and the debugging session, we have chosen to use two different processes communicating with sockets. The advantages are manifold. Firstly, the traced and debugging programs are clearly separated. The debugging programs cannot have unintended side-effects on the traced execution. Secondly, the same debugging module can be used to debug programs written in different languages. This is currently almost the case, Opium, Coca and Morphine basically run the same Prolog interpreter. We are currently revising their architecture in order to use exactly the same one.

Swapping from one process to another, and writing information over the sockets do take a significant amount of time. The `fget` primitive has been designed in order to avoid them. Indeed, the matching of events according to the filter is done **inside** the traced process. This has two main advantages. Firstly, it avoids a huge number of context switches. Secondly, it enables filtering **on the fly**, namely **no trace needs be stored**. The information is available among the data structures of the traced execution and only the required attribute values are explicitly retrieved. This mechanism is the key of efficient trace filtering.

More sophisticated queries are still possible. With primitives `current` and `current_var`, the debugging programs can retrieve any information about the execution and process it via sophisticated Prolog programs if needed.

Monitoring

The previous mechanism is appropriate for interactive debugging. The user usually has a limited number of hypotheses that he wants to check at a given time (usually even a single hypothesis). When the debugger has found a matching event, it shows its results, and the user think for another hypothesis. The context switches are therefore not so numerous and there is not much information which travels from one process to another. The time taken by information retrieval is comparable to the time taken by user interaction, in particular console inputs and outputs. Response times are therefore satisfactory.

Monitoring, namely when a program is entirely supervised by another program with rare user interactions, is more demanding. Examples of monitoring actions are test coverage measurements, counting the number of times a given procedure is called, verifying that it is never the case that a given variable is above a given value. For most monitors, some treatments have to be performed for each event. Whereas this can be done with the previous framework, context switches and information traveling become much to high and the performances drop dramatically [10].

We have therefore pushed the mechanism further and designed another primitive,`foldt`, which enables monitors to be entirely executed within the traced process. The flexibility of the above approach is kept; the monitor can be started by a debugging program and its results can be processed further. The use of this primitive is illustrated in section 4.

foldt: apply a filter to an accumulator variable and to all events of a trace. This filter collects requested information from this trace. More precisely, the user provides an initial value for an accumulator variable and a definition of the filter; foldt applies the filter in turn at each event to the event attributes and updates the accumulator; the final value of the accumulator is the result of the monitor.

Summary

Our approach is a good compromise between flexibility and efficiency. On the one hand, we provide users with a rich trace model and a powerful trace query language. Users can therefore see exactly what they want to see from a given execution. On the other hand, we provide optimized filtering and monitoring mechanisms which enable "standard" debugging commands to be as efficient as their hard-coded counterparts. Therefore, users have the same efficiency as with standard debuggers on usual debugging and monitoring commands. They have more power because our basic commands are already more generic than standard commands, and they have a good efficiency. If users are ready to wait for longer time (which occurs easily if they have a difficult debugging program to solve), they can ask for more sophisticated queries.

3 A debugging session

In this section, we illustrate the `current` and `fget` primitives by a debugging session with Morphine.

The analyzed program is a Mercury buggy mastermind program too long to be given here. Traditionally there are two players for a mastermind game: the first player, A, sets a code in the form of a hidden list of colored pins; the second player, B, tries to guess this code. Player B proposes a list of colored pins and A checks it. A tells how many pins are well placed (called "bulls") and how many of the remaining ones are in the code but at another position (called "cows"). Player B propose new guesses until either the code is found or the maximum number of guesses is reached. In the later, player B looses the game.

Our mastermind program somehow simulates the two players. A user enters a code in the form of 5 colored pins at 5 locations. Then, the *guessing module* tries to find the code given by the user, and the *checking module* checks the guesses against the code to break. Then, and until the guess is correct, the guessing module makes one guess and the checking module tells how good this guess is. We wrote a small Tcl/TK script for the mastermind program where bulls are represented by black pins and cows by white ones. In our example, a user has entered the code `'red, red, red, red, white'`. When the buggy mastermind program is run for this code the inputs and (incorrect) outputs are as follows.

```
Enter a mastermind code : mastermind(red, red, red, red, white).
The solution was found with 5 guesses:
     1: blue blue blue blue blue (0 Bull and 3 Cows).
     2: white white white white black (0 Bull and 3 Cows).
     3: yellow yellow yellow yellow white (1 Bull and 3 Cows).
     4: yellow gray gray gray gray (0 Bull and 3 Cows).
     5: red red red red white (5 Bulls and 0 Cow).
```

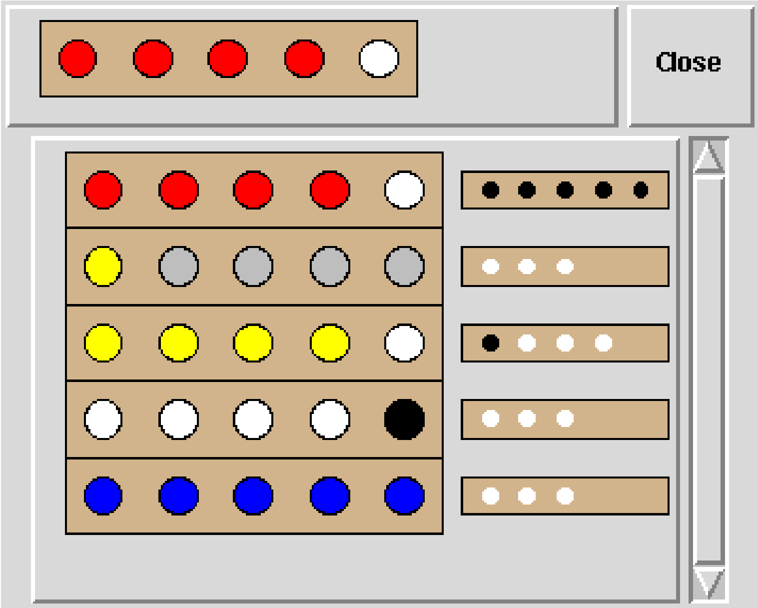

Figure 2. Board output of the buggy mastermind program

The Tcl/TK output of this program is given in Figure 2. The top combination is the code to be found. The successive combinations starting from the bottom one are the successive guesses of the guessing module. On the right hand side are the answers of the checking module. One can tell from the very first answer that the number of cows is wrong. Indeed, there is no blue pin at all in the code and yet the checking module tells that there are three not well placed pins.

In the following, we show how to use our trace query mechanism to find the bug. The trace queries typed in by the user are the goals following the Morphine prompt (`[Morphine]:`) and terminated by a dot. All the rest is displayed by the system.

Let us assume that we have re-started the execution of the buggy mastermind program under Morphine and that we are positioned at the very first event of the execution. The predicate that counts the bulls and cows of a guess with respect to a code is `check_mastermind`. Let us go to the event where this predicate succeeds to see if it exits with sensible argument values.

In order to do so, we invoke the `fget` command and ask to move to the next event where `check_mastermind` exits.

```
[Morphine]: fget(name = check_mastermind and port = exit).

Enter a mastermind code : mastermind(red, red, red, red, white).

1432557:  exit check_mastermind(
                mastermind(blue, blue, blue, blue, blue),
                mastermind(red, red, red, red, white), 0, 3)
```

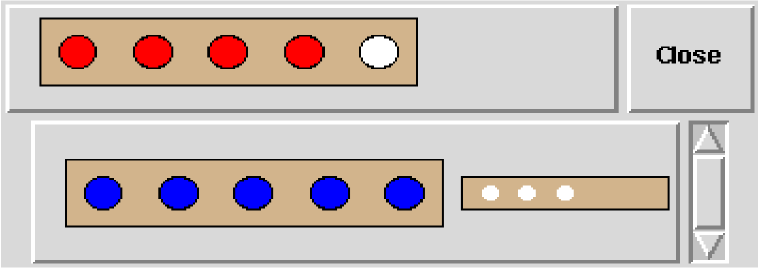

Figure 3. Board output of display_mastermind

The event is represented by its chronological number (`1432557`), its port [3] (`exit`), the name of the predicate (`check_mastermind`), and the list of arguments with their current instantiation (`mastermind(blue, blue, blue, blue, blue), mastermind(red, red, red, red, white), 0, 3`).

We can already see that `fget` has filtered over one million trace events here.

We can now reuse the `display_mastermind` predicate that is used in the mastermind program to display the mastermind boards. This will make the interpretation of the answers easier. The only thing that we have to do is to get the current value of the arguments of `check_mastermind`. We use `current` with the attribute `args` for that purpose [4]. The corresponding output of the call to `display_mastermind` is given in Figure 3.

```
[Morphine]: current(args = [Guess, Code, Bulls, Cows]),
            display_mastermind(Code, Guess, Bulls, Cows).

Code = mastermind(red, red, red, red, white)
Guess = mastermind(blue, blue, blue, blue, blue)
Bulls = 0
Cows = 3
```

As we mentioned earlier, the number of cows is indeed wrong. It should be 0 and not 3. After investigating the source code (not shown here), we could see that the only non-library predicate that is called within the subgoal which counts the cows is the predicate `remove_cows_from_list`. We, therefore, would like to check if this predicate outputs correct data. The following query asks to go to the next call of `count_cows`, then to the next exit of `remove_cows_from_list` and retrieve the current execution depth.

```
[Morphine]: fget(name = count_cows and port = call),
            fget(name = remove_cows_from_list and port = exit),
            current(depth = D).
```

[3] Other possible ports are for example `call`, `fail`, `redo`, `exception`

[4] In pure logic programming there is no global variables. Hence in Morphine and Opium `current_var` is not strictly necessary

```
1432614: call count_cows(mastermind(hole,hole,hole,hole,blue),
                         mastermind(hole,hole,hole,hole,white),-)
1432632: exit remove_cows_from_list(hole, [], [])
D = 14     More? [yes/no] yes

1432633:  exit remove_cows_from_list(hole, [blue], [hole])
D = 13     More? [yes/no] no
```

Note that we have taken benefit from the backtracking mechanism of Prolog in order to get several events which match the filter specified in the query. Here for example, we can tell that `remove_cows_from_list` is a recursive predicate. The first `exit` found refers to the base case stopping the recursion, this case is not very interesting for our problem. The following matching event is more interesting.

The goal `remove_cows_from_list(Elt, ListIn, ListOut)` is supposed to output in `ListOut` the list `ListIn` where all the occurrences of `Elt` have been removed. Here, the predicate `remove_cows_from_list` which outputs the list `[hole]` should have output the list `[blue]`.

A *hole* is a special color, it refers to pins that have already been counted. We could want to make sure that `remove_cows_from_list` is also wrong with normal colors. Note that commands that move to an event and print a trace line (like `fget`) have a corresponding command which moves to this event without printing any trace line (`fget_np`). Actually, `fget` is defined as follow:

```
fget(Filter) :- fget_np(Filter), print_event.
```

We can therefore constraint further the filtering by inserting predicates between moving forward to an event and printing it. If the constraints are not satisfied, the execution backtracks to `fget`, and this until either the constraints are satisfied or the execution is terminated.

The following query refines the previous one. It specifies that the first argument of `remove_cows_from_list` should not be *hole*.

```
[Morphine]: fget_np(name = remove_cows_from_list  and  port = exit),
            current(args = [X, -, -]),
            not(X = hole),
            print_event.

1432724:  exit remove_cows_from_list(white, [], [])
X = white     More? [yes/no] yes

1432725:  exit remove_cows_from_list(white, [hole], [white])
X = white     More? [yes/no] no
```

The previous query was not refined enough. We would also like to filter out events where the hole appears in the list of the second argument of `remove_cows_from_list`. We therefore constraint further the arguments.

```
[Morphine]: fget_np(name = remove_cows_from_list  and  port = exit),
            current(args = [X, [Y], -]),
            not(X = hole),
            not(Y = hole),
            print_event.

1434263:  exit remove_cows_from_list(black, [white], [black])
X = black
Y = white
```

Now, we are definitely convinced that `remove_cows_from_list` is buggy: here the answer should have been `remove_cows_from_list(black, [white], [white])`. We can display the source code of `remove_cows_from_list`.

```
[Morphine]: listing_current_procedure.

:- pred remove_cows_from_list(color, list(color), list(color)).
:- mode remove_cows_from_list(in, in, out) is det.
% remove_cows_from_list(Elt, ListIn, ListOut) outputs in ListOut
% the list ListIn where all the occurrences of Elt has been removed.
remove_cows_from_list(_, [], []).
remove_cows_from_list(C, [X | Xs], ListOut) :-
        ( if
                C = X,
                C \= hole
          then
                ListOut = Xs
          else
                remove_cows_from_list(C, Xs, List),
                ListOut = [C | List]
        ).
```

We can see that in the case where `C` is different from `X`, the returned list should be unchanged. Thus `Listout` should be `[X | List]` and not `[C | List]`. The last line contains the bug.

Discussion

We have shown a debugging session where sophisticated queries have been illustrated. Some of them filter out several millions of events in acceptable response time.

One can start a debugging session with very simple query (something like a traditional breakpoint), and then refine the query until the trace events found illustrate the very problem that was looked for. For example, our debugging session starts to search for events related to a given predicate, then refines the port, then constraints the values of the arguments until we know where exactly we have to look in the source in order to fix the bug.

```
:- import_module int.
:- type collected_type == int.

initialize(0).

collect(Event, AccIn, AccOut, continue) :-
        ( if port(Event) = call
          then    AccOut = AccIn + 1
          else    AccOut = AccIn ).
```

Figure 4. A simple monitor that counts procedure calls

Simple traditional commands are available, users satisfied with them are not forced to enter long queries. As already stated, the time overhead compared to the underlying hard-coded tracer, in such cases, is negligible. If the user has more precise hypotheses to check, then the tool can support him in so doing.

Complex queries can require a significant amount of time to be executed. We conjecture that proof-checking by hand this kind of hypotheses would take much more time.

4 Monitoring examples

As introduced in Section 2, the `foldt` monitoring primitive collects information from program executions. It is intended to enable users to easily implement their own monitors with acceptable performances. To use it within Morphine, one needs to define in a file 3 items only, using the Mercury syntax:

(i) a `collected_type` which is the type of the collecting variable that will contain the result of the monitoring activity.

(ii) The predicate `initialize` which initializes this collecting variable.

(iii) The predicate `collect` which updates the collecting variable at each execution event. Predicate `collect` also outputs a variable that indicates whether to stop collecting. If this variable is set to `stop`, the `foldt` process stops; if it is set to `continue`, it continues. If this variable is always set to `continue`, collecting will process until the last event is reached.

Then, the defined file is used to generate a Mercury module which is compiled and dynamically linked with the current execution. When a `foldt` query is made from Morphine, a variable of type `collected_type` is first initialized (with `initialize`) and then updated (with `collect`) for each event of the remaining execution. When the `stop` variable of `collect` is set, or when the end of the execution is reached, the last value of the collecting variable is sent to Morphine.

Figure 4 gives an example of a simple monitor that counts procedures calls. Assuming that these definitions are in a file called `count_call`, the command

```
:- import_module set, stack.

:- type arc ---> arc(proc_name/arity, int, proc_name/arity).
:- type collected_type ---> ct(stack(predicate), set(arc)).

initialize(ct(stack__push(stack__init, "user"/0), set__init)).

collect(Event, Acc0, Acc, continue) :-
    Port = port(Event),
    Acc0 = ct(Stack0, Graph0),
    ( if    Port = call
    then    PreviousPred = stack__top_det(Stack0),
            CurrentPred = proc_name(Event) / proc_arity(Event),
            Graph = promise_only_solution(
                update_graph(PreviousPred, CurrentPred, Graph0)),
            Acc = ct(stack__push(Stack0, CurrentPred), Graph)
    else if Port = redo
    then    CurrentPred = proc_name(Event) / proc_arity(Event),
            Acc = ct(stack__push(Stack0, CurrentPred), Graph0)
    else if ( Port = fail ; Port = exit ; Port = exception )
    then    stack__pop_det(Stack0, _, Stack),
            Acc = ct(Stack, Graph0)
    else    % other events
            Acc = Acc0 ).

update_graph(Pred0, Pred, Graph0, Graph) :-
    ( if    member(arc(Pred0, N, Pred), Graph0)
    then    delete(Graph0, arc(Pred0, N, Pred), Graph1),
            insert(Graph1, arc(Pred0, N+1, Pred), Graph)
    else    insert(Graph0, arc(Pred0, 1, Pred), Graph) ).
```

Figure 5. Monitor that computes dynamic control flow graphs

`run(program), foldt(count_call, Result)` at the Morphine prompt unifies `Result` with the number of calls that occur during the current `program` execution.

A more sophisticated monitor is presented in Figure 5. It computes dynamic control flow graphs of Mercury programs. Firstly, library modules *set* and *stack* are imported. In order to define this monitor, a call stack and a list of arcs representing procedure calls are maintained. Arcs are labeled by a counter representing the number of time a predicate has been called. The stack and the set of arcs are initialized to the empty stack and to the empty set respectively. At call events, an arc is inserted from the previous predicate to the current one and the current predicate is pushed on the stack (function `stack__push/2`). At redo events, only the call stack is updated by pushing on

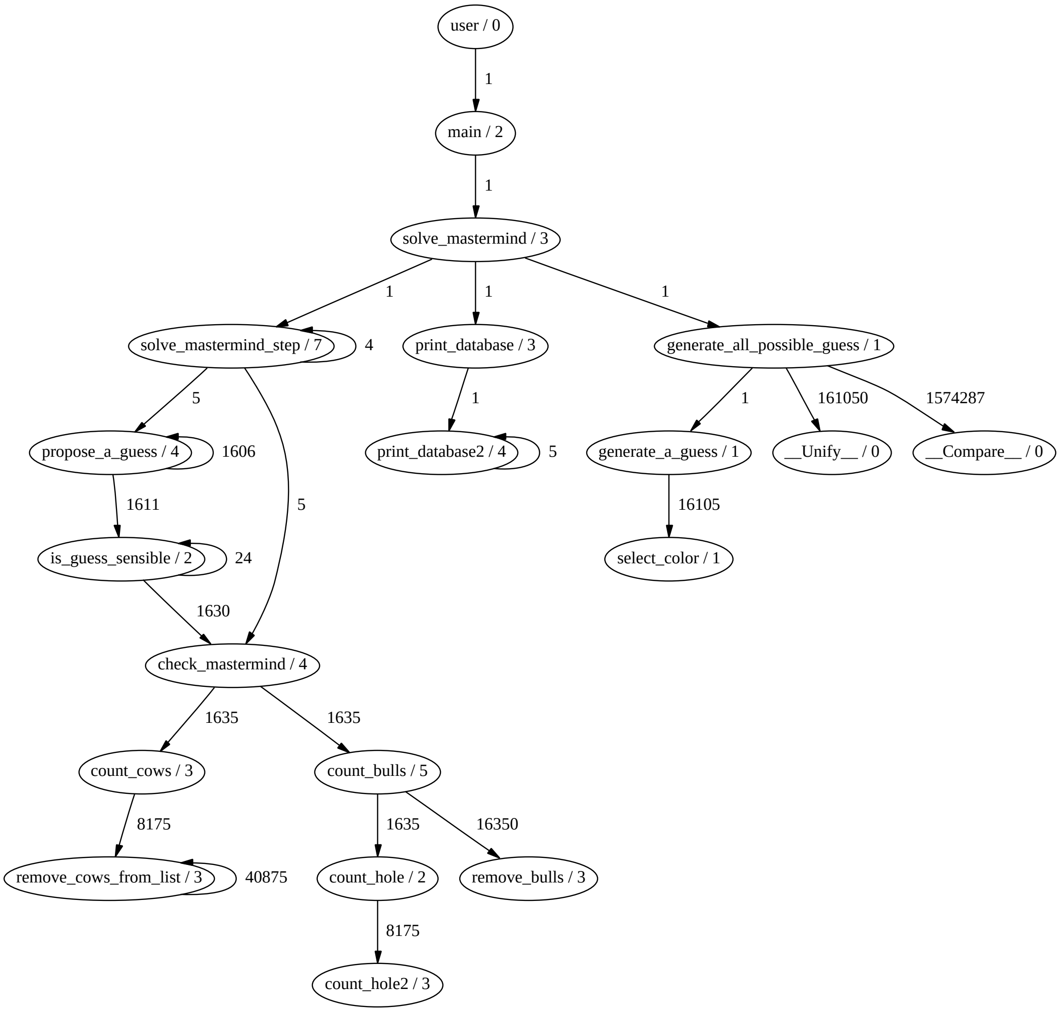

Figure 6. The dynamic control flow graph of the mastermind program

it the current predicate. At fail, exit, and exception events, the top element of the stack is removed (predicate `stack__pop_det/3`).

Note that the call stack that is computed on the fly is available as a trace event attribute in the Mercury tracer. We reconstructed it for didactical purpose and because many tracers do not provide this information straightforwardly.

At the Morphine prompt, the invocation of goal

```
run(mastermind), foldt(dcg, Result).
```

unifies `Result` with a set of arcs that defines the dynamic call graph of the `mastermind` program. We wrote a very small Prolog program that post-processes this result and generate a graph in the format of the graph drawing tool dot [14]. The corresponding postscript output of dot is given in Figure 6.

More examples of monitors, in particular measurements of the coverage of a test suite, can be found in [12,11].

5 Conclusion

In this article we have presented the principles of three automated tracer analyzers for three different languages. They provide a good compromise between flexibility and efficiency. We have illustrated this with a debugging session and two monitors. Sophisticated queries can be specified on the fly. Some of them filter out several millions of events in acceptable response time. Monitors can easily be implemented and executed with performances equivalent to their hard-coded counterparts.

The Morphine prototype is available at http://www.irisa.fr/lande/jahier/.

References

[1] Peter C. Bates. Debugging heterogeneous distributed systems using event-based models of behavior. *ACM Transactions on Computer Systems*, 13(1):1–31, February 1995.

[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] 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.

[4] 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.

[5] 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).

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

[7] 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.

[8] W. E. Howden and G. M. Shi. Linear and structural event sequence analysis. In Steven J. Zeil, editor, *Proceedings of the 1996 International Symposium on Software Testing and Analysis*, pages 98–106, New York, January8–10 1996. ACM Press.

[9] IC.Parc, London. *The ECLiPSe Constraint Logic Programming System, ECLiPSe 4.1 - User Manual*, February 1999. http://www-icparc.doc.ic.ac.uk/eclipse/.

[10] E. Jahier. *Analyse dynamique de programmes : mise en oeuvre automatisée d'analyseurs performants et spécification de modèles d'exécution.* PhD thesis, INSA de Rennes, Décembre 2000. Partly in English.

[11] E. Jahier. Collecting graphical abstract views of Mercury program executions. In *Proceedings of the International Workshop on Automated Debugging (AADEBUG2000)*, Munich, August 2000. Refereed proceedings, the COmputer Research Repository (CORR) cs.SE/0010038.

[12] E. Jahier and M. Ducassé. A generic approach to monitor program executions. In D. De Schreye, editor, *Proceedings of the International Conference on Logic Programming*, pages 139–153. MIT Press, November 1999.

[13] C. Jeffery, W. Zou, K. Templer, and M. Brazell. A lightweight architecture for program execution monitoring. In *Workshop on Program Analysis for Software Tools and Engineering.* ACM SIGPLAN, June 1998.

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

[15] 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.

[16] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of logic programming*, 29:17–64, October-December 1996.

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

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

[19] K.S. Templer and C.L. Jeffery. A configurable automatic instrumentation tool for ANSI C. In *Proceedings of the Automated Software Engineering Conference.* IEEE Computer Society, 1998.