Published in *Theory and Practice of Logic Programming* **2** (4&5): 613–645, September 2002.

Generic Program Monitoring by Trace Analysis∗

ERWAN JAHIER and MIREILLE DUCASSÉ †
IRISA/INSA, Campus Universitaire de Beaulieu, F-35042 Rennes cedex, France

Abstract

Program execution monitoring consists of checking whole executions for given properties, and collecting global run-time information. Monitoring gives valuable insights and help programmers maintain their programs. However, application developers face the following dilemma: either they use existing monitoring tools which never exactly fit their needs, or they invest a lot of effort to implement relevant monitoring code. In this article we argue that, when an event-oriented tracer exists, the compiler developers can enable the application developers to easily code their own monitors. We propose a high-level primitive, called `foldt`, which operates on execution traces. One of the key advantages of our approach is that it allows a clean separation of concerns; the definition of monitors is totally distinct from both the user source code and the language compiler. We give a number of applications of the use of `foldt` to define monitors for Mercury program executions: execution profiles, graphical abstract views, and two test coverage measurements. Each example is implemented by a few simple lines of Mercury.

Keywords: monitoring, automated debugging, trace analysis, test coverage, Mercury

1 Introduction

Program maintenance and trace analysis. Several experimental studies (e.g., (Hatton, 1997)) show that maintenance is the most expensive phase of software development: the initial development represents only 20 % of the cost, whereas error fixing and addition of new features after the first release represent, each, 40 % of the cost. Thus, 80 % of the cost is due to the maintenance phase.

A key issue of maintenance is program understanding. In order to fix logical errors, programmers have to analyze their program symptoms and understand how these symptoms have been produced. In order to fix performance errors, programmers have to understand where the time is spent in the programs. In order to add new functionality, programmers have to understand how the new parts will interact with the existing ones.

Program analysis tools help programmers understand programs. For example, type checkers (Pfenning, 1992) help understand data inconsistencies. Slicing tools (Gallagher & Lyle, 1991; Tip, 1995) help understand dependencies among parts of

∗ This article extends (Jahier & Ducassé, 1999) and (Jahier, 2000b).
† This work has been partially sponsored by the Esprit project ARGO, Industrial RTD Project no 25503

a program. Tracers give insights into program executions (Eisenstadt & Brayshaw, 1988).

Some program analysis tools automatically analyze program execution traces. They can give very precise insights of program (mis)behavior. We have shown how such trace analyzers can help users debug their programs. In our automated debuggers, a trace query mechanism helps users check properties of parts of traced executions in order to understand misbehavior (Ducassé, 1999a; Ducassé, 1999b; Jahier & Ducassé, 1999a).

In this article, we show that trace analysis can be pushed toward monitoring to further help understand program behavior. Whereas debuggers are tools that retrieve run-time information at specific program points, monitors collect information relative to the whole program executions. For example, some monitors gather statistics which help detect heavily used parts that need to be optimized; other monitors build graphs (e.g., control flow graphs, dynamic call graphs, proof trees) that give a global understanding of the execution.

Execution monitoring. Monitors are trace analyzers which differ from debuggers. Monitoring is mostly a "batch" activity whereas debugging is mostly an interactive activity. In monitoring, a *set* of properties is specified beforehand; the whole execution is checked; and the global collected information is displayed. In debugging, the end-user is central to the process; he specifies on the fly the very next property to be checked; each query is induced by the user's current understanding of the situation at the very moment it is typed in. Monitoring is therefore less versatile than debugging. The properties specified for monitoring have a much longer lifetime, they are meant to be used over several executions.

It is, nevertheless, impossible to foresee all the properties that programmers may want to check on executions. One intrinsic reason is that these properties are often driven by the application domain. Therefore monitoring systems must provide some genericity.

Existing approaches to implement monitors. Unfortunately, monitors are generally implemented by ad hoc instrumentation. This instrumentation requires a significant programming effort. When done at a low level, for example by modifying the compiler and the runtime system, it requires deep knowledge that mostly only the language compiler implementors have. However, the monitored information is often application-dependent, and application programmers or end-users know better what has to be monitored. But instrumenting compilers is almost impossible for them.

An alternative to low-level instrumentation is source-level instrumentation. For example, run-time behavior information can be extracted by source-to-source transformation, as done, .e.g., for ML (Tolmach & Appel, 1995; Kishon *et al.*, 1991) and Prolog (Ducassé & Noyé, 2000)). Such instrumentation, although simpler than low-level compiler instrumentation, can still be too complex for most programmers. Furthermore, for certain new declarative programming languages like Mercury (Somogyi *et al.*, 1996), instrumentation may even be impossible. Indeed, in Mercury, the declarative semantics is simple and well defined, but the operational semantics is complex. For example, the compiler reorders goals according to its needs. Furthermore, input and output can be made only in deterministic predicates. This complicates code instrumentation.

Thus, ad hoc instrumentation is tedious at a low level and it may be impossible at a high level. On the other hand, the difficult task of instrumenting the code to extract run-time information has, in general, already been achieved to provide a debugger. Debuggers, which help users locate faults in programs are based on tracers. These tracers generate execution traces which provide a precise and faithful image of the operational semantics of the traced language. These traces often contain sufficient information to base monitors upon them.

Our Proposal. In this article, we propose a high-level primitive built on top of an event oriented execution tracer. The proposed monitoring primitive, called `foldt`, is a `fold` which operates on a list of events.

An event oriented *trace* is a sequence of events. An *event* is a tuple of event attributes. An *event attribute* is an elementary piece of information that can be extracted from the current state of the program execution. Thus, a trace can be seen as a sequence of tuples of a database ordered by time. Many tracers are event-oriented: for example, Prolog tracers based on Byrd box model (Byrd, 1980), tracers for C such as Dalek (Olsson *et al.*, 1990) and Coca (Ducassé, 1999a), the Egadt tracer for Pascal (Fritzson *et al.*, 1994), the Esa tracer for Ada (Howden & Shi, 1996), and the Ebba tracer for distributed systems (Bates, 1995).

One of the key advantages of our approach is that it allows a clean separation of concerns; the definition of the monitors is totally distinct from both the user source code and the language compiler.

We have implemented `foldt` on top of the Mercury trace. We give a number of applications of the `foldt` operator to compute various monitors: execution profiles, graphical abstract views, and test coverage measurements. Each example is implemented by a few lines of Mercury which can be written by any Mercury programmer. These applications show that the Mercury trace, indeed, contains enough information to build a wide variety of interesting monitors. Detailed measurements show that, under some requirements, `foldt` can have acceptable performance for executions of several millions of execution events. Therefore our operator lays the foundation for a generic and powerful monitoring environment. The proposed scheme has been integrated into the Mercury environment. It is fully operational and part of the Mercury distribution.

Note that we have implemented the `foldt` operator on top of Mercury mostly for historical reasons. We acknowledge that some of the monitors were particularly easy to write thanks to the neatness of Mercury libraries, in particular the set library (e.g., Figure 10). Nevertheless, `foldt` could be implemented for any system with an event-oriented tracer.

Plan. In Section 2, we introduce the `foldt` operator and describe its current implementation on top of the Mercury tracer. In Section 3, we illustrate the genericity of `foldt` with various kinds of monitors. All the examples are presented at a level of detail that does not presuppose any knowledge of Mercury. Section 4 discusses performance issues of `foldt`. Section 5 compares our contribution with related work. A thorough description of the Mercury trace can be found in Appendix A. Appendix B lists a Mercury program solving the n queens problem, which is used at various places in the article as an input for our monitors.

2 A high-level trace processing operator: `foldt`

In this section, we first define the `foldt` operator over a general trace in a language-independent manner. We describe an implementation of this operator for Mercury program executions, and then present its current user interface.

2.1 Language independent `foldt` *definition*

A trace is a list of events; analyzing a trace therefore requires to process such a list. The standard functional programming operator `fold` encapsulates a simple pattern of recursion for processing lists. It takes as input arguments a function, a list, and an initial value of an accumulator; it outputs the final value of the accumulator; this final value is obtained by successively applying the function to the current value of the accumulator and each element of the list. As demonstrated by Hutton (Hutton, 1999), `fold` has a great expressive power for processing lists. Therefore, we propose a `fold`-like operator to process execution traces; we call this operator `foldt`.

Before defining `foldt`, we define the notions of event and trace for sequential executions.

Definition 1
(Execution event, Event attributes, Execution trace)
An execution event is an element of the Cartesian product $\mathbb{E} = A_1 \times ... \times A_n$, where A_i for $i \in \{1, ..., n\}$ are arbitrary sets called *event attributes. An execution trace* is a (finite or infinite) sequence of execution events; the set of all execution traces is denoted by $\mathbb{T}$. We note $|t|$ the size (its number of events) of a finite trace $t \in \mathbb{T}$ and $|t| = \infty$ the size of infinite traces.

The following definition of `foldt` is a predicative definition of a `fold` operating on a finite number of events of a (possibly infinite) trace. The set of predicates over $\tau_1 \times ... \times \tau_n$ is denoted by $pred(\tau_1, ..., \tau_n)$.

Definition 2
(`foldt`)
A `foldt` *monitor of type* $\tau \times \tau'$ is a 3-tuple : $(\texttt{initialize}, \texttt{collect}, \texttt{post\_process}) \in pred(\tau) \times pred(\mathbb{E}, \tau, \tau) \times pred(\tau, \tau')$ such that: $\forall t = (e_i)_{i>0} \in \mathbb{T}$, either

$$(1)\ |t| < \infty\ \wedge (\exists!(V_0, ..., V_n) \in \tau^{n+1}.$$
$$(\texttt{initialize}(V_0)\ \bigwedge_{i=1}^{n} \texttt{collect}(e_i, V_{i-1}, V_i) \wedge \texttt{post\_process}(V_n, Res)))$$

(2) $\exists! n < |t|, \exists!(V_0, ..., V_n) \in \tau^{n+1}, \forall x \in \tau.$
$(\texttt{initialize}(V_0) \wedge \bigwedge_{i=1}^{n} \texttt{collect}(e_i, V_{i-1}, V_i) \wedge \texttt{post\_process}(V_n, Res)$
$\wedge\ \neg\texttt{collect}(e_{n+1}, V_n, x))$

Res is called *the result of the monitor* (initialize, collect, post_process) *on trace t*. We use the notation $\exists! n$ to mean that there exists a unique n, and $(e_i)_{i>0}$ for the sequence (in the mathematical sense) $e_1, e_2, e_3, ...$

Operationally, an accumulator of type τ is used to gather the collected information. It is first initialized (V_0). The predicate `collect` is then applied to each event of the trace in turn, updating the accumulator along the way (V_i). There are two ways to stop this process: (1) the folding process stops when the end of the execution is reached if the trace is finite ($|t| < \infty$); (2) if `collect` fails before the end of the execution is reached ($\forall x \in \tau.\ (\neg\texttt{collect}(e_{n+1}, V_n, x))$). In both cases, the last value of the accumulator (V_n) is processed by `post_process`, which returns a value (Res) of type τ' ($\texttt{post\_process}(V_n, Res)$).

Note that this definition holds for finite and infinite traces (thanks to the second case of Definition 2). This is convenient to analyze programs that run permanently. The ability to end the `foldt` process before the end of the execution is also convenient to analyze executions part by part as explained in Section 2.3.3. A further interesting property, which is useful to execute several monitors in a single program execution, is the possibility to simultaneously apply several `fold` on the same list using a tuple of `fold` (Bird, 1987); in other words:

$\texttt{foldt}(i_1, c_1, p_1) \times ... \times \texttt{foldt}(i_n, c_n, p_n) = \texttt{foldt}(i_1 \times ... \times i_n, c_1 \times ... \times c_n, p_1 \times ... \times p_n)$
where:
$\forall a_1, ..., a_n \in \tau_1 \times ... \times \tau_n,$
$\quad i_1 \times ... \times i_n(a_1, ..., a_n) \Leftrightarrow i_1(a_1)\ \wedge ...\ \wedge i_n(a_n),$
$\forall e \in \mathbb{E}, \forall a_1, ..., a_n \in \tau_1 \times ... \times \tau_n, \forall a'_1, ..., a'_n \in \tau'_1 \times ... \times \tau'_n,$
$\quad c_1 \times ... \times c_n(e, a_1, ..., a_n, a'_1, ..., a'_n) \Leftrightarrow c_1(e, a_1, a'_1)\ \wedge ...\ \wedge c_n(e, a_n, a'_n),$
$\forall a_1, ..., a_n \in \tau_1 \times ... \times \tau_n,$
$\quad p_1 \times ... \times p_n(a_1, ..., a_n, a'_1, ..., a'_n) \Leftrightarrow p_1(a_1, a'_1)\ \wedge ...\ \wedge p_n(a_n, a'_n).$

2.2 An implementation of `foldt` *for Mercury*

We prototyped an implementation of `foldt` for the Mercury programming language. After a brief presentation of Mercury and its trace system, we describe our `foldt` implementation.

2.2.1 Mercury and its trace

Mercury (Somogyi *et al.*, 1996) is a logic and functional programming language. The principal differences with Prolog are as follows. Mercury supports functions and higher-order terms. Mercury programs are free from side-effects; even input and output are managed in a declarative way. Mercury strong type, mode and determinism system allows a lot of errors to be caught at compile time, and a lot of optimizations to be done.

The trace generated by the Mercury tracer (Somogyi & Henderson, 1999) is adapted from Byrd box model (Byrd, 1980). Its attributes are the event number, the call number, the execution depth, the event type (or port), the determinism, the procedure (defined by a module name, a name, an arity and a mode number), the live arguments, the live non-argument variables, and the goal path. A detailed description of these attributes together with an example of event is given in appendix A.

2.2.2 The `foldt` *implementation*

An obvious and simple way to implement `foldt` would be to store the whole trace into a single list, and then to apply a `fold` to it. This naive implementation is highly inefficient, both in time and in space. It requires creating and processing a list of possibly millions of events. Most of the time, creating such a list is simply not feasible because of memory limitations. With the current Mercury trace system, several millions of events are generated each second, each event requiring several bytes. To implement realistic monitors, run-time information needs to be collected and analyzed simultaneously (on the fly), **without explicitly creating the trace.**

In order to achieve analysis on the fly, we have implemented `foldt` by modifying the Mercury trace system, which works as follows: when a program is compiled with tracing switched on, the generated C code[1] is instrumented with calls to the tracer (via the C function `trace`). Before the first event (resp. after the last one), a call to an initialization C function `trace_init` (resp. to a finalization C function `trace_final`) is inserted.

When the trace system is entered through either one of the functions `trace`, `trace_init`, or `trace_final`, the very first thing it does is to look at an environment variable that tells whether the Mercury program has been invoked from a shell, from the standard Mercury debugger (mdb), or from another debugger (e.g., Morphine (Jahier & Ducassé, 1999a)). We have added a new possible value for that environment variable which indicates whether the program has been invoked by `foldt`. In that case, the `trace_init` function dynamically links the Mercury program under execution with the object file that contains the object code of `collect`, `initialize`, and `post_process`. Dynamically linking the program to its monitor is very convenient because neither the program nor the monitor need to be recompiled.

Once the monitor object file has been linked with the program, the C function `trace_init` can call the procedure `initialize` to set the value of a global variable `accumulator_variable` (of type τ). At each event, the C function `trace` calls the procedure `collect` which updates `accumulator_variable`. If `collect` fails or if the last event is reached, the C function `trace_final` calls the procedure `post_process` with `accumulator_variable` and returns the new value of this accumulator (now of type τ').

[1] Currently, the only Mercury back-end that has a tracer is one that relies on a C compiler to produce its executable code.

```
1       % 1 - Define the type of the accumulator:
2       :- type accumulator_type == < A Mercury type >.
3
4       % 2 - Initialize the accumulator:
5       initialize(Accumulator) :-
6               < Mercury goals which initialize the accumulator >.
7
8       % 3 - Update the accumulator:
9       collect(Event, AccumulatorIn, AccumulatorOut) :-
10              < Mercury goals which update the accumulator >.
11
12      % 4 - Optionally, post-process the last value of the accumulator:
13      :- type collected_type == < A Mercury type >.
14
15      post_process(Accumulator, FoldtResult) :-
16              < Mercury goals which post-process the accumulator >.
```

Figure 1. What the user needs to define to use `foldt`

2.3 The current user interface of `foldt` *for Mercury*

In this Section, we first describe what the user needs to do in order to define a monitor with `foldt`. Then, we show how this monitor can be invoked.

2.3.1 Defining monitors

We chose Mercury to be the language in which users define the `foldt` monitors to monitor Mercury programs. As a matter of fact, it could have been any other language that has an interface with C, since the trace system of Mercury is written in C. The choice of Mercury, however, is quite natural; people who want to monitor Mercury programs are likely to be Mercury programmers.

The items users need to implement in order to define a `foldt` monitor are given in Figure 1. Lines preceded by `'%'` are comments. First of all, since Mercury is a typed language, one first needs to define the type of the accumulator variable `accumulator_type` (line 2). Then, one needs to define `initialize` which gives the initial value of the accumulator, and `collect` which updates the accumulator at each event(line 9). Optionally, one can also define the `post_process` predicate which processes the last value of the accumulator. `post_process` takes as input a variable of type `accumulator_type` (τ) and outputs a variable of type `collected_type` (τ'). If `collected_type` is not the same type as `accumulator_type`, then one needs to provide its definition too (line 13). Types and modes of predicates `initialize`, `collect` and `post_process` should be consistent with the following Mercury declarations:

```
:- pred initialize(accumulator_type::out) is det.
:- pred collect(event::in,accumulator_type::in,accumulator_type::out)
        is semidet.
:- pred post_process(accumulator_type::in,collected_type::out) is det.
```

```
1   :- import_module int.
2   :- type accumulator_type == int.
3   initialize(0).
4   collect(Event, C0, C) :-
5     if port(Event) = call then C = C0+1 else C = C0.
```

Figure 2. count_call, a monitor that counts the number of calls using foldt

These declarations state that initialize is a deterministic predicate (is det), namely it succeeds exactly once, and it outputs a variable of type accumulator_type; collect is a semi-deterministic predicate, namely it succeeds at most once, and it takes as input an event and an accumulator. If collect fails, the monitoring process stops at the current event. This can be very useful, for example to stop the monitoring process before the end of the execution if the collecting data is too large, or to collect data part by part (e.g., collecting the information by slices of 10000 events). This also allows foldt to operate over non-terminating executions.

The type event is a structure made of all the event attributes. To access these attributes, one can use specific functions whose declarations are of the form: ":- func <attribute_name>(event::in) = <attribute_type>::out.", which means that each such function takes an event and outputs the event attribute corresponding to its name. For example, the function call depth(Event) returns the depth of Event. The complete list of attribute names is given in Appendix A.

Figure 2 shows an example of monitor that counts the number of predicate invocations (calls) that occur during a program execution. We first import library module int (line 1) to be able to manipulate integers. Predicate initialize initializes the accumulator to '0' (line 3). Then, for every execution event, collect increments the counter if the event port is call, and leaves it unchanged otherwise (line 5). Since collect can never fail here, the calls to collect proceed until the last event of the execution is reached.

Note that those five lines of code constitute *all the necessary lines* for this monitor to be run. For the sake of conciseness, in the following figures containing monitors, we sometimes omit the module importation directives as well as the type of the accumulator when the context makes them clear.

2.3.2 Invoking foldt

Currently, foldt can be invoked from a Prolog query loop interpreter. We could not use Mercury for that purpose because there is no Mercury interpreter yet.

We have implemented a Prolog predicate named run_mercury, which takes a Mercury program call as argument, and which forks a process in which this Mercury program runs in coroutining with the Prolog process. The two processes communicate via sockets. When the first event of the Mercury program is reached, the hand is given to the Prolog process which waits for a foldt query.

The command foldt has two arguments; the first one should contain the name of the file defining the monitor to be run; the second one is a variable that will be

```
[morphine]: run_mercury(queens), foldt(count_call, Result).
                                        A 5 queens solution is  [1, 3, 5, 2, 4]
     Last event of queens is reached
     Result = 146     More? (;)
[morphine]:
```

Figure 3. Invoking `foldt` monitor of Figure 2 from an interpreter

unified with the result of the monitor. When `foldt` is invoked, (1) the file containing the monitor is used to automatically produce a Mercury module named `foldt.m` (by adding the declarations of `initialize`, `collect`, and `post_process`, as well as the definitions of the `event` type and the attribute accessing functions); (2) `foldt.m` is compiled, producing the object file `foldt.o`; (3) `foldt.o` is dynamically linked with the Mercury program under coroutining. Of course, steps (1) and (2) are only performed if the file containing the monitor is newer than the object file `foldt.o`.

A monitor stops either because the end of the execution is reached, or because the `collect` predicate failed; in the latter case, the current event (i.e., the event the next query will start at) is the one occurring immediately after the event where `collect` failed.

A possible session for invoking the monitor of Figure 2 is given in Figure 3. At the right-hand side of the '`[morphine]:`' prompt, there are the characters typed in by a user. The line in italic is output by the Mercury program; all the other lines are output by the Prolog process. We can therefore see that the program `queens` (which solves the 5 queens problem, cf Appendix B) produces 146 procedure calls.

2.3.3 Illustration of the advantage of calling `foldt` *from a Prolog query loop*

Being able to call `foldt` from a Prolog interpreter loop enables users to write scripts that control several `foldt` invocations. Figures 4 and 5 illustrate this. The monitor of Figure 4 computes the maximal depth for the next 500 events. In the session of Figure 5, a user (via the `[user].` directive) defines the predicate `print_max_depth` that calls the monitor of Figure 4 and prints its result in loop until the end of the execution is reached. This is useful for example for a program that runs out of stack space to check whether this is due to a very deep execution and to know at which events this occurs.

Note that the fact that the monitor is dynamically linked with the monitored program has an interesting side-effect here: one can change the monitor during the `foldt` query resolution (by modifying the file where this monitor is defined). Indeed, in our example, one could change the interval within which the maximal depth is searched from 500 to 100. The monitor would be (automatically) recompiled, but the `foldt` query would not need to be killed and rerun. This can be very helpful to monitor a program that runs permanently; the monitored program is simply suspended while the monitor is recompiled.

As a matter of fact (as the prompt suggests), the Prolog query loop that we use is Morphine (Jahier & Ducassé, 1999a), an extensible debugger for Mercury "à la

```
initialize(acc(0, 0)).
collect(Event, acc(N0, Depth0), acc(N0+1, max(Depth0, depth(Event)))) :-
        N0 < 500.  % stops after 500 events
```

Figure 4. Monitor that computes the maximal execution depth by interval of 500 events

```
[morphine]: [user].
  print_max_depth :-
        foldt(max_depth, acc(_, MaxDepth)),
        print("The maximal depth is "), print(MaxDepth), nl,
        print_max_depth.
^D
[morphine]: run_mercury(qsort), print_max_depth.

   The maximal depth is 54
   The maximal depth is 28
   The maximal depth is 50
                                          [0, 2, 4, 6, 7, 8, ..., 94, 95, 99, 99]
   Last event of qsort is reached
   The maximal depth is 53
[morphine]:
```

Figure 5. A possible session using the monitor of Figure 4

Opium" (Ducassé, 1999b). The basic idea of Morphine is to build on top of a Prolog query loop a few coroutining primitives connected to the trace system (like `foldt`). Those primitives let one implement all classical debugger commands as efficiently as their hand-written counter-parts; the advantage is, of course, that they let users implement more commands than the usual hard-coded ones, fitting their own needs.

Invoking `foldt` from a debugger has a further advantage; it makes it very easy to call a monitor during a debugging session, and vice versa. Indeed, some monitors are very useful for understanding program runtime behavior, and therefore can be seen as debugging tools.

3 Applications

In this section, we describe various execution monitors that can be implemented with `foldt`. We first give monitors which compute three different execution profiles: number of events at each port, number of goal invocations at each depth, and sets of solutions. Then, we describe monitors that produce two types of execution graphs: dynamic control flow graph and dynamic call graph. Finally, we introduce two test coverage criteria for logic programs, and we give the monitors that measure them.

```
1   :- import_module int, array.
2   :- type accumulator_type == array(int).
3
4   :- mode acc_in  :: array_di.
5   :- mode acc_out :: array_uo.
6
7   initialize(Array) :-
8       init(5, 0, Array).
9
10  collect(Event, Array0, Array) :-
11      Port = port(Event),
12      port_to_int(Port, IntPort),
13      lookup(Array0, IntPort, N),
14      set(Array0, IntPort, N+1, Array).
15
16  :- pred port_to_int(port::in, int::out) is det.
17  port_to_int(Port, Number) :-
18      ( if    Port = call then Number = 0
19      else if Port = exit then Number = 1
20      else if Port = redo then Number = 2
21      else if Port = fail then Number = 3
22      else  Number = 4 ).
```

Figure 6. A monitor that counts the number of events at each port

3.1 Execution profiles

3.1.1 Counting the number of events at each port

In Figure 2, we have given a monitor that counts the number of goal invocations. Figure 6 shows how to extend this monitor to count the number of events at each port. We need 5 counters that we store in an array. In the current implementation of `foldt`, the default mode of the second and third argument of `collect`, respectively equal to `in` and `out`, can be overridden; here, we override them with `array_di` and `array_uo` (lines 4 and 5). Modes `array_di` and `array_uo` are special modes that allow arrays to be destructively updated. Predicate `initialize` creates an array `Array` of size 5 with each element initialized to 0 (line 8). Predicate `collect` extracts the port from the current event (line 11) and converts it to an integer (line 12)[2]. This integer is used as an index to get (`lookup/3`) and set (`set/4`) array values. The goal `lookup(Array0, IntPort, N)` returns in `N` the `IntPort`th element of `Array0`. The goal `set(Array0, IntPort, N+1, Array)` sets the value `N+1` in the `IntPort`th element of `Array0` and returns the resulting array in `Array`.

```
1  % Module importation and accumulator type (array of int) omitted
2
3  initialize(Acc) :-
4          init(32, 0, Acc).
5
6  collect(Event, Acc0, Acc) :-
7      ( if port(Event) = call then
8          Depth = depth(Event),
9          ( if semidet_lookup(Acc0, Depth, N) then
10             set(Acc0, Depth, N+1, Acc)
11         else
12             size(Acc0, Size),
13         resize(Acc0, Size*2, 0, Acc1),
14         set(Acc1, Depth, 1, Acc)
15         )
16     else
17         Acc = Acc0
18     ).
```

Figure 7. A monitor that counts the number of calls at each depth

3.1.2 Counting the number of calls at each depth

Figure 7 implements a monitor that counts the number of calls at each depth. Predicate `initialize` creates an array of size 32 with each element initialized to 0 (line 4). At call events (line 7), predicate `collect` extracts the depth from the current event (line 8) and increments the corresponding counter (lines 10 and 14). Whenever the upper bound of the array is reached, i.e., whenever `semidet_lookup/4` fails (line 9), the size of the array is doubled (lines 13).

3.1.3 Collecting solutions

The monitor of Figure 8 collects the solutions produced during the execution. We define the type `solution` as a pair containing a procedure and a list of arguments (line 1). The collected variable is a list of `solutions` (line 2), which is initialized to the empty list (line 4). If the current port is `exit` (line 8) and if the current solution has not been already produced (lines 9,10), then the current solution is added to the list of already collected solutions (line 12).

Note that for large programs, it would be better to use a table from predicate names to set of solutions instead of lists.

3.2 Graphical abstract views

Other execution abstract views that are widely used and very useful in terms of program understanding are given in terms of graphs. In the following, we show how

[2] As a matter of fact, there are more ports than the ones handled by **port_to_int/2** in Figure 6 (cf Appendix A); we ignore them here for the sake of conciseness.

```
:- type solution ---> proc_name/arguments.
:- type accumulator_type == list(solution).

initialize([]).

collect(Event, AccIn, AccOut) :-
    ( if
        port(Event) = exit,
        Solution = proc_name(Event)/arguments(Event),
        not(member(Solution, AccIn))
      then
        AccOut = [Solution | AccIn]
      else
        AccOut = AccIn
    ).
```

Figure 8. A monitor that collects all the solutions

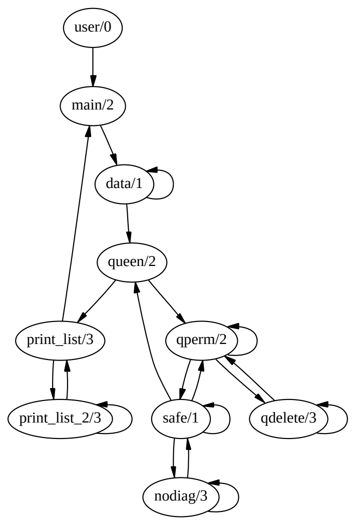

Figure 9. The dynamic control flow graph of 5 queens

to implement monitors that generate graphical abstractions of program executions such as control flow graphs and dynamic call graphs. We illustrate the use of these monitors by applying them to the 5 queens program given in Appendix B. This 100 line program generates 698 events for a board of 5×5. In this article, we use the graph drawing tool `dot` (Koutsofios & North, 1991). More elaborated visualization tools such as in (Stasko *et al.*, 1998) would be desirable, especially for large executions. This is, however, beyond the scope of this article.

```
1   :- type predicate ---> proc_name/arity.
2   :- type arc ---> arc(predicate, predicate).
3   :- type graph == set(arc).
4   :- type accumulator_type ---> collected_type(predicate, graph).
5
6   initialize(collected_type("user"/0, set__init)).
7
8   collect(Event, Acc0, Acc) :-
9       Port = port(Event),
10      ( if (Port = call ; Port = exit ; Port = fail ; Port = redo) then
11           Acc0 = collected_type(PreviousPred, Graph0),
12           CurrentPred = proc_name(Event) / proc_arity(Event),
13           Arc = arc(PreviousPred, CurrentPred),
14           set__insert(Graph0, Arc, Graph),
15           Acc = collected_type(CurrentPred, Graph)
16        else
17           % other events
18           Acc = Acc0
19      ).
```

Figure 10. A monitor that calculates dynamic control flow graphs

3.2.1 Dynamic control flow graphs

We define the *dynamic control flow graph* of a logic program execution as the directed graph where nodes are predicates of the program, and arcs indicate that the program control flow went from the origin to the destination node. The dynamic control flow graph of the 5 queens program is given in Figure 9. We can see, for example, that, during the program execution, the control moves from predicate `main/2` to predicate `data/1`, from predicate `data/1` to predicate `data/1` and predicate `queen/2`. Note that such a graph (or variants of it) is primarily useful for tools and only secondarily for humans.

An implementation of a monitor that produces such a graph is given in Figure 10. Graphs are encoded by a set of arcs, and arcs are terms composed of two predicates (lines 1 to 3). The collecting variable is composed of a predicate and a graph (line 4); the predicate is used to remember the previous node. The collecting variable is initialized with the fake predicate `user/0`, and the empty graph (line 6). At `call`, `exit`, `redo`, and `fail` events (line 10), we insert in the graph an arc from the previous predicate to the current one (lines 11 to 14).

Note that in our definition of dynamic control flow graph, the number of times each arc is traversed is not given. Even if the control goes between two nodes several times, only one arc is represented. One can imagine a variant where, for example, arcs are labeled by a counter; one just needs to use multi-sets instead of sets. The result of such a variant applied to the 5 queens program is displayed Figure 11. Note that here, the queens program was linked with a version of the library that has been compiled without trace information. This is the reason why one should not be surprised not to see any call to, e.g, `io__write_string/3` in this figure.

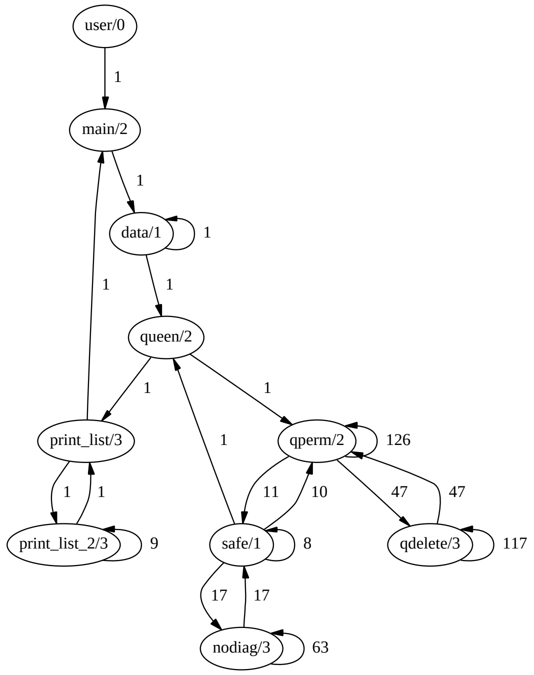

Figure 11. The dynamic **control flow graph** of 5 queens annotated with counters

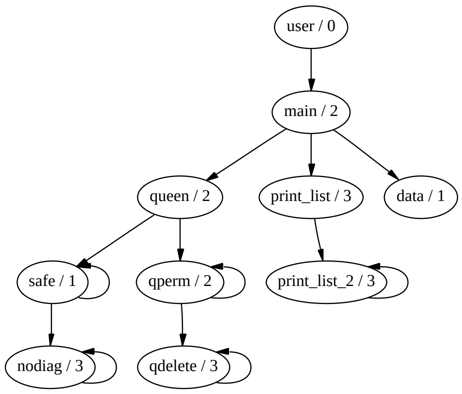

Figure 12. The dynamic **call graph** of 5 queens

3.2.2 Dynamic call graphs

A *static call graph* of a program is a graph where the nodes are labeled by the predicates of the program, and where arcs between nodes indicate potential predicate calls. We define the *dynamic call graph* of a logic program execution as the sub-graph of the (static) call graph composed of the arcs and nodes that have actually been traversed during the execution. For example, in Figure 12, we can see

```
1   % Definition of pred, arc, and graph types omitted (cf previous monitor)
2   :- type accumulator_type ---> ct(stack(predicate), graph).
3
4   initialize(ct(Stack, set__init)) :-
5         stack__push(stack__init, "user"/0, Stack).
6
7   collect(Event, ct(Stack0, Graph0), Acc) :-
8         Port = port(Event),
9         CurrentPred = proc_name(Event) / proc_arity(Event),
10        update_call_stack(Port, CurrentPred, Stack0, Stack),
11        ( if Port = call then
12              PreviousPred = stack__top_det(Stack0),
13              set__insert(Graph0, arc(PreviousPred, CurrentPred), Graph),
14              Acc = ct(Stack, Graph)
15          else
16              Acc = ct(Stack, Graph0) ).
17
18  :- pred update_call_stack(trace_port_type::in, predicate::in,
19        stack(predicate)::in, stack(predicate)::out) is det.
20  update_call_stack(Port, CurrentPred, Stack0, Stack) :-
21        ( if ( Port = call ; Port = redo ) then
22              stack__push(Stack0, CurrentPred, Stack)
23          else if ( Port = fail ; Port = exit ; Port = exception ) then
24              stack__pop_det(Stack0, _, Stack)
25          else % other events
26              Stack = Stack0 ).
```

Figure 13. A monitor that computes dynamic call graphs

that predicate `main/2` calls predicates `data/1`, `queen/2`, and `print_list/2`. In this particular example, the static and dynamic call graphs are identical.

An implementation of a monitor that builds the dynamic call graphs is given in Figure 13. In order to define this monitor, we use the same data structures as for the previous one, except that we replace the last traversed predicate by the whole call stack in the collected variable type (line 2). This stack is necessary in order to be able to get the direct ancestor of the current predicate. The set of arcs is initialized to the empty set (lines 4) and the stack is initialized to a stack that contains a fake node `user/0` (line 5). In order to construct the set of arcs, we insert at call events an arc from the previous predicate to the current one (line 12). The call stack is maintained on the fly by the `update_call_stack/4` predicate; the current predicate is pushed onto the stack at `call` and `redo` events (line 22), and popped at `exit`, `fail`, and `exception` events (line 24). The result of the execution of this monitor applied to the `5 queens` program is displayed in Figure 12.

Note that the call stack is actually available in the Mercury trace. We have intentionally not use it here for didactical purpose in order to demonstrate how this information can easily (but not cheaply!) be reconstructed on the fly.

3.3 Test coverage

In this section, we define two notions of test coverage for logic programs, and we show how to measure the corresponding coverage rate of Mercury program executions using the `foldt` primitive. The aim here is not to provide the ultimate definition of test coverage for logic programs, but rather to propose two possible definitions, and to show how the corresponding coverage rate measurements can be quickly prototyped. As a consequence, the proposed monitors cannot pretend to be optimal either in functionality, or in implementation.

3.3.1 Test coverage and logic programs

The aim of test coverage is to assess the quality of a test suite. In particular, it helps to decide whether it is necessary to generate more test cases or not. For a given coverage criterion, one can decide to stop testing when a certain percentage of coverage is reached.

The usual criterion used for imperative languages are *instruction* and *branch* criteria (Beizer, 1990). The *instruction coverage rate* achieved by a test suite is the percentage of instructions that have been executed. The *branch coverage rate* achieved by a test suite is the percentage of branches that have been traversed during its execution.

One of the weaknesses of instruction and branch coverages is due to Boolean expressions. The problem occurs when a Boolean expression is composed by more than one atomic instruction: it may be that a test suite covers each value of the whole condition without covering all values of each atomic part of the condition. For example, consider the condition 'A or B' and a test suite where the two cases '$A = true$, $B = false$' and '$A = false$, $B = false$' are covered. In that case, every branch and every instruction is exercised, and nevertheless, B never succeeded. If B is erroneous, even 100% instruction and branch coverage will miss it. Whereas in imperative programs, you get conditional branches only in the conditions of if-then-else and loops, in logic programs you get them at every unification and call (whose determinism allows failure); therefore this issue is crucial for logic programs.

3.3.2 Predicate coverage

In order to address the above problem, we need a coverage criterion that checks that each single predicate defined in the tested program succeeds and fails a given number of times. But we do not want to expect every predicate to fail because some, like printing predicates, are intrinsically deterministic. Therefore, we want a criterion that allows the test designer to specify how many times a predicate should succeed and fail. Therefore we define a *predicate criterion* as a pair composed of a predicate and a list of `exit` and `fail`. In the case of Mercury, we can take advantage of the determinism declaration to automatically determine if a predicate should succeed and fail. Here is an example of predicate criterion that can be automatically defined according to the determinism declaration of each predicate:

'det' predicates:	1 success
'semidet' predicates:	1 success, 1 failure
'multi' predicates:	2 successes
'nondet' predicates:	2 successes, 1 failure

Then, we define the *predicate coverage rate of a logic program test suite* as the percentage of program predicate criteria that are covered during the execution of the suite. To compute that rate, one just needs to look at `exit` and `fail` events to see which criterion is covered.

Figure 14 shows a `foldt` monitor that measures the predicate coverage rate of the `queens` program. The accumulator variable is a table (`map`) from procedure name to predicate criterion (line 2). A predicate criterion is represented by a list of `exit` and `fail` events; the type `pred_crit` also contains a list of call numbers (line 1), initially empty (lines 6 to 14). They are used to remember encountered exits (lines 37 to 43). Indeed, if an execution produces two `exit` events for a predicate, it does not mean that a given call of this predicate has succeeded twice; it can be due to another call, for example recursive. Hence, we can remove an `exit` atom from the list of ports to be covered only if, either it is the first time the predicate exits (line 24), or if the current call number has been encountered before (line 28). Symmetrically, since all multi and nondet predicates that are called end up with a `fail` event, a failure can be considered as covered only if no `exit` events occurred for the current call number before (lines 24 and 30).

The current distribution of Morphine[3] have support to automatically generate such monitors, run them, and compute the coverage percentage. Monitors are generated by parsing the source files in order to get the procedure determinisms that are necessary to be able to produce the right number of `exit` and `fail` atoms.

3.3.3 Call site coverage

The previous coverage criterion only checks that at least one exit for each predicate is covered. The problem is that 100% predicate coverage does not imply 100% instruction nor branch coverage. To ensure 100% instruction and branch coverage, we need a criterion that ensures that every *predicate invocation* in the program succeeds and fails. Hence we need a definition attached to call sites (or goals) and not only to predicates. To achieve that, we just need, for example, to take advantage of line numbers and having a table from procedure names and line numbers to list of ports as accumulator. The monitor that measures call site coverage rate of Mercury program executions is therefore roughly the same as the one given in Figure 14, where the accumulator type becomes:

```
:- type proc ---> p(declared_module_name, proc_name, line_number).
:- type call_site_crit ---> csc(list(call_number), list(port)).
```

[3] cf the file `extras/morphine/source/generate_pred_cov.m` and the `pred_cov` Morphine command (both in `extras.tgz` available on the Mercury ftp and web sites).

```
:- type pred_crit ---> pc(list(call_number), list(port)).
:- type accumulator_type == map(proc_name, pred_crit).

initialize(Map) :-
  map__init(Map0),
  map__det_insert(Map0, "main",        pc([], [exit]), Map1),
  map__det_insert(Map1, "data",        pc([], [exit]), Map2),
  map__det_insert(Map2, "print_list",  pc([], [exit]), Map3),
  map__det_insert(Map3, "print_list_2",pc([], [exit]), Map4),
  map__det_insert(Map4, "safe",   pc([], [exit,fail]), Map5),
  map__det_insert(Map5, "nodiag", pc([], [exit,fail]), Map6),
  map__det_insert(Map6, "qperm",  pc([], [exit,exit,fail]), Map7),
  map__det_insert(Map7, "qdelete",pc([], [exit,exit,fail]), Map8),
  map__det_insert(Map8, "queen",  pc([], [exit,exit,fail]), Map).

collect(Event, Map0, Map) :-
  Port = port(Event), Proc = proc_name(Event), CallN = call(Event),
  ( if
      (Port = exit ; Port = fail), pc(CNL0, PL0) = map__search(Map0, Proc)
    then
    ( if
          CNL0 = []
        then
          remove_port(Port, PL0, PL)
        else if
          member(CallN, CNL0)
        then
          (if Port = exit then remove_port(exit, PL0, PL) else PL = PL0)
        else % not member(CallN, CNL0) and not (CNL0 = [])
          (if Port = exit then PL = PL0 else remove_port(fail, PL0, PL))
      ),
      ( if
          PL = []
        then
          map__delete(Map0, Proc, Map)
        else
          ( if
              Port = exit, not member(CallN, CNL0)
            then
              CNL = [CallN | CNL0]
            else
              CNL = CNL0
          ),
          map__update(Map0, Proc, pc(CNL, PL), Map))
    else
      Map = Map0 ).

:- pred remove_port(port::in, list(port)::in, list(port)::out) is det.
remove_port(Port, L0, L) :-
  if list__delete_first(L0, Port, L1) then L = L1 else L = L0.

:- type collected_type == assoc_list(proc_name, pred_crit).
post_process(Map, AssocList) :- map__to_assoc_list(Map, AssocList).
```

Figure 14. A monitor that measures the predicate coverage rate of the n queens program

```
:- type accumulator_type == map(proc, call_site_crit).
```

Here again, such monitors are generated automatically parsing the source files[4].

4 Experimental Evaluation

In the previous section we have shown the flexibility and power of the `foldt` primitive. The aim of this section is to assess the performance of the current `foldt` implementation. When executing a monitor, some time is spent in the normal program execution (T_{prog}); and some extra time is spent in the trace system of Mercury (Δ_{trace}), in the interface between the tracer and the `foldt` mechanism [5] (Δ_{inte}), in the basic `foldt` mechanism (Δ_{foldt}), and also in the monitor itself ($\Delta_{<monitor>}$). Hence, if we call T the execution time of a monitored program, we have:

$$T = T_{prog} + \Delta_{trace} + \Delta_{inte} + \Delta_{foldt} + \Delta_{<monitor>}$$

In the following, we measure: T_{prog} , $T_{trace} = T_{prog} + \Delta_{trace}$, $T_{inte} = T_{prog} + \Delta_{trace} + \Delta_{inte}$, and $T_{foldt} = T_{prog} + \Delta_{trace} + \Delta_{inte} + \Delta_{foldt}$.

We compare T_{trace}, T_{inte}, and T_{foldt} against T_{prog}. We will therefore compute the following ratios:

$$R_t = T_{trace}/T_{prog}, \quad R_i = T_{inte}/T_{prog} \quad \text{and} \quad R_f = T_{foldt}/T_{prog}$$

4.1 Methodology

Hardware and software. The measurements given in the following show the results of experiments run on a DELL inspiron 7500, with a 433 MHz Celeron, 192 Mb of RAM, running under the Linux 2.2.14 operating system. The machine was very lightly loaded; no X server, no network, simply the basic operating system and a Prolog process in a console to run the measurement scripts. The Prolog system is Eclipse 4.1 (Eclipse, 1999). The Mercury compiler is a stable snapshot of 14 June 2001[6]. The results are consistent with experiments run on a SUN Sparc Enterprise 250 (2 x UltraSPARC-II, 296MHz, 512 Mb of RAM) running Solaris 2.7 (figures not given here).

Time measuring command. In order to measure the program execution times, we use the `benchmark_det` predicate of the `benchmarking.m` Mercury standard library. This predicate repeats the body of a program any given number of times. This is very important for small programs, as the startup cost very often dominates the execution cost. In the following experiments, each program is re-executed until it runs at least for 20 seconds. Each experiment has been done five times, and the deviation was smaller than 1 %.

[4] cf the file `extras/morphine/source/generate_call_site_cov.m` and the `call_site_cov` Morphine command.
[5] The Mercury predicate `collect` is called from the Mercury tracer which is written in C.
[6] The last official release is numbered 0.10.1

Monitored programs. The monitored programs are the Mercury benchmark suite[7], composed of programs adapted from the Prolog benchmark suite of (Van Roy & Despain, 1992). In order to have a wider range of execution sizes, we also measure `n queens` for n=10,11, as well as `mastermind`, a 1100 lines Mercury program which solves a mastermind game[8].

Mercury compilation grades. In the following, the compilation grade $\mathtt{g_{nt}}$ refers to Mercury modules compiled with the command `mmc --grade asm_fast.gc.picreg`, which means that no trace event is generated. It is the grade used to measure the plain execution time of programs (T_{prog}).

The compilation grade $\mathtt{g_t}$ refers to Mercury modules compiled with the command `mmc --grade asm_fast.gc.picreg --trace deep --trace-optimized`, which means that all events related to all the predicates of the module, except library predicates, are generated. This grade is used in the following to measure the time spent in the basic trace system (T_{trace}), the time spent in the interface between the basic tracer and the `foldt` mechanism (T_{inte}), the time spent in the basic `foldt` mechanism (T_{foldt}) and the time spent in the monitors ($T_{<monitor>}$).

Measuring T_{prog}. If the programs are compiled in grade $\mathtt{g_{nt}}$, then their execution does not produce any trace and their measured execution duration ($T_{measured}$) is exactly T_{prog}.

$$T_{prog} = T_{measured} \quad \text{if compilation grade is } \mathtt{g_{nt}}$$

Measuring T_{trace}. If the programs are compiled in grade $\mathtt{g_t}$, then their execution calls the Mercury tracer. In order to measure the cost of the basic tracer, we have to ensure that the tracer is systematically called at each event, but that it does not do anything else than entering and exiting the top-level switch of the trace system.

$$\begin{gathered} T_{trace} = T_{measured} \quad \text{if compilation grade is } \mathtt{g_t} \text{ and} \\ \Delta_{inte} = 0, \ \ \Delta_{foldt} = 0, \ \ \mathit{and} \ \ \Delta_{<monitor>} = 0 \end{gathered}$$

In order to ensure that $\Delta_{inte} = 0, \ \ \Delta_{foldt} = 0 \ \ \mathit{and} \ \ \Delta_{<monitor>} = 0$, we use the `continue` command of the Mercury tracer without specifying any break-point. Indeed, with that command, at each event, the trace system is entered; if the event does not correspond to one of the specified breakpoints, the normal execution is resumed. In our measurements, as no break-point is specified, the whole execution is traversed, and nothing is executed but the basic tracing mechanism.

Measuring T_{inte}. In order to measure the cost of the interface between the basic tracer and the `foldt` mechanism, we have to ensure that the tracer is systematically called at each event, that it enters and exits the top-level switch of the trace system, that it prepares the context to call the `collect` predicate defined for the monitor,

[7] The source code of this benchmark suite can be found on the Mercury ftp site `ftp://www.mercury.cs.mu.oz.au/pub/mercury/mercury-tests-*.tar.gz`

[8] The full source code of the Mercury mastermind program can be found at first author web site

but that it does not compute anything else, in particular it should not retrieve any event attribute.

$$T_{inte} = T_{measured} \quad \text{if compilation grade is } \mathtt{g_t} \text{ and}$$
$$\Delta_{foldt} = 0, \quad and \ \ \Delta_{<monitor>} = 0$$

In order to ensure that $\Delta_{foldt} = 0$, we have implemented a degenerate `foldt` such that no event attribute is computed (we have replaced these computations by void values). In order to ensure that $\Delta_{<monitor>} = 0$, we use a monitor that does not compute anything (`collect(_E, A, A).`).

Measuring T_{foldt}. In order to measure the cost of the basic `foldt` mechanism, we have to ensure that `collect` is called at each event for a monitor that computes nothing.

$$T_{foldt} = T_{measured} \quad \text{if compilation grade is } \mathtt{g_t} \text{ and } \Delta_{<monitor>} = 0$$

In order to make sure that $\Delta_{<monitor>} = 0$, we call `foldt` with a trivial monitor, that does not compute anything (`collect(_E, A, A).`).

Two of the current attributes are very costly to retrieve: the live arguments and the line number. The live arguments can be very large data structures. The line number corresponds to the line where the goal is called and not where the predicate is defined. It is dynamically retrieved. Many interesting monitors can be run without these attributes. Indeed, for the monitors we propose in this article, only one monitor uses the live arguments and one monitor uses the line number. Monitors that do not use these costly attributes can disable them. As a consequence, the measurements of T_{foldt} are made with these two attributes disabled.

4.2 Resulting table

Table 1 illustrates the cost of the basic tracer, the cost of the interface between the tracer and `foldt`, as well as the cost of `foldt` on the benchmark programs described in the previous sections. The first column contains the names of the monitored programs. The second column contains the numbers of execution events generated by the program executions compiled in grade $\mathtt{g_t}$ (all events are generated, except events relative to library predicates). The programs are sorted wrt this number of events, from `queens-5`, 698 events, to `queens-11`, more than 32 millions of events. The third column contains the execution times of the programs compiled without any trace information (T_{prog}). The fourth column contains the execution times of the programs compiled in grade $\mathtt{g_t}$ and run under the control of the tracer without tracing anything ($T_{trace} = T_{prog} + \Delta_{trace}$). The fifth column contains the overhead factor of the basic trace mechanism ($R_t = T_{trace}/T_{prog}$). The sixth column contains the execution times of the programs compiled in grade $\mathtt{g_t}$ and run under the control of the tracer, and where the degenerate `foldt` is called with an empty monitor ($T_{inte} = T_{prog} + \Delta_{trace} + \Delta_{inte}$). The seventh column contains the overhead factor of the trace and the interface between the tracer and `foldt` ($R_i = T_{inte}/T_{prog}$). The eighth column contains the execution time of the programs compiled in grade $\mathtt{g_t}$ and

Table 1. *Cost of the* `foldt` *mechanisms on benchmarks.* $T_{trace} = T_{prog} + \Delta_{trace}$, $T_{inte} = T_{prog} + \Delta_{trace} + \Delta_{inte}$, $T_{foldt} = T_{prog} + \Delta_{trace} + \Delta_{inte} + \Delta_{foldt}$, $R_t = T_{trace}/T_{prog}$, $R_i = T_{inte}/T_{prog}$, $R_f = T_{foldt}/T_{prog}$, $R_f^* = (T_{foldt} - \Delta_{inte})/T_{prog}$

program	events	T_{prog} in ms	T_{trace} in ms	R_t	T_{inte} in ms	R_i	T_{foldt} in ms	R_f	R_f^*
queens-5	698	0.03	0.24	7.5	2	61.5	2.07	63	9.5
query	935	0.09	0.28	3	1.53	16.5	1.58	17	3.5
deriv	1,540	0.05	0.11	2.5	0.64	14	0.66	14.5	3
qsort	1,564	0.1	0.48	5	4.03	42	4.16	43.5	6.5
nrev	1,619	0.14	0.54	4	4.44	32.5	4.58	33.5	5
primes	2,192	0.21	0.8	4	6.23	30	6.51	31	5.5
cqueens	3,789	0.14	1.26	9.5	11.11	81	11.39	82	11.5
crypt	4,602	0.72	1.8	3	11.16	16	11.54	16.5	3.5
poly	79,070	6.44	29	4.5	226.2	35.5	233.4	36.5	6
tak	190,831	3.88	57.1	15	553.7	142	572.4	148	20
queens-10	4,289,986	257	1530	6	12760	50.5	13200	51.5	8
mastermind	9,106,510	3630	6500	2	30490	8.5	31520	9	2.5
queens-11	32,384,320	2103	12030	6	97010	46.5	100190	48	7.5

run under the control of `foldt` with an empty monitor ($T_{foldt} = T_{prog} + \Delta_{trace} + \Delta_{inte} + \Delta_{foldt}$). The ninth column contains the overhead factor of the trace, the interface and the basic `foldt` mechanism ($R_f = T_{foldt}/T_{prog}$). The tenth column contains the overhead factor of the trace and the basic `foldt` mechanism, with the interface cost divided ($R^*_{foldt} = (T_{foldt} - \Delta_{inte})/T_{prog}$). The time measurements have been rounded off two digits after the dot. The ratios have been rounded up to the nearest half.

4.3 Discussion

In this section, we discuss the resulting ratios of Table 1.

Two extremes: `tak` and `mastermind`. The `tak` program has ratios much higher than the other programs. The tracer overhead is already 15, the interface overhead is 142 and the `foldt` overheads are 148 and 20. This program is actually a single predicate four times recursive. It already broke the stacks of the reference tracer used in (Ducassé & Noyé, 2000). This code is an extreme case to test compiler optimization capabilities. As many optimizations, such as last call optimization, cannot be applied to traced code, the better the compiler is, the worse debugger and monitor ratios look. Program `Tak` is very untypical of real life programs. The ratios related to `tak` are not taken into account in the averages given below.

On the other hand `mastermind` has very low ratios. The tracer overhead is 2, the interface overhead is 8.5 and the `foldt` overheads are 9 and 2.5. The `mastermind` program uses a lot of library predicates which are not traced[9] and monitored in detail. This is typical of real life programs. It is encouraging that the more realistic

[9] their calls and exits are traced, but not what happens inside these calls

program has the best ratios. However, the other benchmarks do not use untraced libraries, in order to be fair, the ratios related to `mastermind` are not taken into account in the averages given below.

In the following, averages are, thus, computed without the figures related to `tak` and `mastermind`.

The ratios are not correlated with the number of events. Program `queens-5`, which has only 698 events, has very bad ratios, whereas `crypt`, almost five thousand events, and `poly`, almost eighty thousand events, have better ratios than the average. The same program, `n queens`, run for n= 5,10 and 11, always has comparable ratios. This seems to indicate that the overheads depend mostly of the monitored program and is somewhat constant for a given program.

Overhead of the tracer. The average of the tracer overhead is 5. It is a very good ratio. Prolog tracers can easily have an overhead over 20 (Ducassé & Noyé, 2000). A low ratio for the tracer is, of course, a good starting point to build efficient generic monitors.

Overhead of the interface between the tracer and `foldt`. The average of the interface overhead is 39. This is very high and it is the main source of inefficiency of our current implementation. It illustrates how crucial the implementation of this interface is for efficient generic monitoring.

The problems comes from the fact that the monitor programs have to be integrated in the tracer. In our particular case, the Mercury predicate `collect` is called from the Mercury tracer which is written in C. In order to call Mercury code from C with the current (low-level back-end of the) Mercury compiler, machine registers need to be saved and restored. Since the `collect` predicate is called several million times, this has a noticeable influence on the performance. A way to remove this overhead could be to use the new MLDS back-end of the Mercury compiler, which generates high-level C code that does not use machine registers; unfortunately, the trace system is not supported for the MLDS back-end at the time of this writing. Once the MLDS Mercury back-end is available, calling the `collect` predicate will actually be compiled as a simple C procedure call from within C code. The overhead of the interface between the tracer and `foldt` should thus become smaller.

One important lesson learned from these measurements is as follows. Whether the monitors and the tracer are implemented in the same programming language or not, the integration of the compiled monitors should not cost more than a procedure call. The monitors must therefore be compiled into the same target language as the tracer. Furthermore, no run-time verifications should be made. The monitors should therefore have no side-effect on the traced execution. It should thus be statically checked that monitors only update their own (fresh) variables.

Overhead of `foldt`. The average of the `foldt` overhead is 40. Most of it is due to the overhead of the interface discussed above. Assuming that the above fix could be done and that the interface overhead indeed becomes negligible, we have computed an ideal ratio: $R_f^* = (T_{foldt} - \Delta_{inte})/T_{prog}$. The average of the overhead of `foldt` without the interface cost is 6.5. As the average of the tracer overhead is 5, we can say that the `foldt` overhead is acceptable.

In the context of Mercury, this is especially true because the developers of Mercury claim that Mercury programs executed in trace mode are faster than the equivalent Prolog programs executed in optimized mode in the faster Prolog systems (Somogyi & Henderson, 1999).

Unused event attributes. As already mentioned, some attributes can be very costly to compute. When they are not needed it should be possible to disable them. In the current implementation of `foldt`, this is already the case for the list of live arguments and the line numbers. The foldt overhead has been measured without the cost of the live arguments and the line numbers. Some measurements, not reported here, showed that this has an impact on the performance.

Granularity of the instrumentation. In order to measure the worst case, we have made the Mercury tracer systematically generate all the possible events. Not all monitors need such a fine-grained instrumentation. For example, for the monitor that counts the number of events at each depth, only `call` events are necessary. When one (hard-)codes a specific monitor, one only instruments the code where it is necessary. As a matter of fact, the Mercury tracer enables users to specify what type of events, if any, should be generated for a given module (the only restriction is that, if some events are generated for a predicate, `call` events must be present). Hence, programmers can already take advantage of this possibility to optimize their monitor. As further work, we plan to automate this optimization, namely, to automatically generate the appropriate compiling option for a given monitor.

Conclusion on performance. The cost of monitors is generally superseded by the cost of the `foldt` mechanism, except for time demanding monitors such as the one that computes dynamic call graphs and coverage rates for which we have yet another slowdown of a factor ranging from 1.5 to 5. Hence, our conclusion is that with a fast tracer, an interface between the tracer and `foldt` reduced to procedure calls, and the possibility to disable the computation of heavy non-necessary attributes, generic monitoring can be efficient enough.

5 Related work

Programmable debuggers. We designed 3 programmable debuggers, Opium for Prolog (Ducassé, 1999b), Coca for C (Ducassé, 1999a) and Morphine for Mercury (Jahier & Ducassé, 1999a). They are based on a Prolog query loop plus a handful of coroutining primitives connected to the trace system. Those primitives allow the Prolog interpreter to communicate with the debugged program. Opium, Coca and Morphine are full debugging programming languages in which all classical debuggers commands can be implemented straightforwardly and efficiently. However, it appears that their set of primitives are not well suited for monitoring. All the monitors implemented with `foldt` can easily be implemented with the debugger set of primitives (Jahier & Ducassé, 1999b), but the resulting monitors require too many context switches and too much socket traffic between the program and the monitor. With programs of several million execution events, such monitors are four orders of magnitude slower than their counterparts that use `foldt` (Jahier, 2000a).

Automated development of monitors. Jeffery et al. designed Alamo (Templer & Jeffery, 1998; Jeffery *et al.*, 1998; Jeffery, 1999), an architecture that aims at easing the development of monitors for C programs. As in our approach, their monitoring architecture is based on event filtering, and monitors can be programmed. Their system performs trace extraction whereas we rely on an already available tracer; this saves us a tedious task which has already been done and optimized. On the other hand, we do not have the full control of the information available in the trace. Note however that, so far, we have be able to reconstruct missing information, for example the call stack of Figure 13. Moreover, to avoid code explosion, Jeffery et al. perform part of the event filtering at compilation time. This means that they need to recompile the program each time they want to execute another monitor, whereas we only need to dynamically link the monitor to the monitored program. Alamo and the monitored program are running in coroutining, but within the same address space.

Eustace and Srivastava developed Atom (Eustace & Srivastava, 1995), a system that also aims at easing the implementation of monitors. The difference with Alamo is that monitors are implemented with procedure calls and global variables which is much more efficient than coroutining. However, the language provided by Atom is less expressive than the Alamo's. Alamo and Atom have influenced the design of `foldt` and we tried to take the best of both: a full and high-level programming language implemented by procedure calls. The advantages of our architecture over (Eustace & Srivastava, 1995) and (Jeffery *et al.*, 1998) are the following:

- A higher level interface makes the code of our monitors compact, easy to write and read, and therefore to maintain. Of course, this point is difficult to assess. We hope that the numerous and various examples given in Section 3 convince the reader that it is indeed the case. Nonetheless, we see three conjectures explaining why the code is more compact and more elegant. Firstly, users do not have to deal with code instrumentation directives; the instrumentation has already been done. Secondly, we take advantage of the expressive power of `fold`; using high order predicates such as `fold` for processing lists has proven to be concise and far less prone to error than processing lists manually. A third reason is that our monitors are written in Mercury, which is a considerably higher level language than C or C-like languages which are used in (Eustace & Srivastava, 1995; Jeffery *et al.*, 1998).
- Provided that an event-oriented tracer exists, the `foldt` operator is easy to implement. To implement it, the work done inside the Mercury runtime system, which corresponds to the really technical part, amounts to only 61 modified lines and 292 new lines of (C) code.

Kishon and al. (Kishon & Hudak, 1995) use a denotational and operational continuation semantics to formally define monitors for a simple functional programming language. The kind of monitors they define are profilers, debuggers, and statistic collectors. From the operational semantics, a formal description of the monitor, and a program, they derive an instrumented executable file that performs the specified monitoring activity. The semantics of the original programs is preserved. They use partial evaluation to make their monitors reasonably efficient. The main disadvantage with this approach is that they are rebuilding a whole execution system from scratch, without taking advantage of existing compilers. We strongly believe that it is important to have the same execution system for debugging, monitoring and producing the final executable program. As noted by (Brooks *et al.*, 1992), some errors only occur in presence of optimizations, and vice versa; some programs can only be executed in their optimized form because of time and memory constraints; when searching for "hot spots", it is better to do it as much as possible with the optimized program as many things can be optimized away; and finally, sometimes, the error comes from the optimizations themselves. In our setting we can easily mix traced and non-traced code.

Efficient monitoring. Patil and Fisher (Patil & Fischer, 1997) address the problem of performance monitoring by delegating the monitoring activities to a second processor that they call a shadow processor. Their approach is very efficient; the monitored program is practically not slowed down, but the set of monitoring commands they propose cannot be extended. We mentioned in the previous section that we could reduce the number of events generated by the tracer. For example, in (Ball & Larus, 1992), given a static control flow graph, algorithms can place tracing instructions in optimal ways for computing statistics on imperative program executions.

6 Conclusion

In this article we have proposed a generic monitoring framework based on `foldt`[10], a high-level primitive that allows users to easily specify what they want to monitor. We illustrated it on various examples that demonstrate its genericity and its simplicity of use. We defined two preliminary notions of test coverage for logic programs and showed how to prototype coverage rates measurements with our primitive. Testing and monitoring tools are missing from many declarative systems: `foldt` allows some of these tools to be easily defined and implemented. Measurements showed that the performance of the primitive on the above examples can be acceptable for executions of several million trace events.

To sum up the advantages of our framework, we can say that it is:

- Easy to implement: because it is based on an existing tracer (292 new, and 61 modified lines of codes in our current implementation).

[10] Available in Morphine, which can be downloaded from the Mercury ftp and web sites.

- Efficient: because the trace is not stored.
- Flexible and easy to use: as illustrated by the given applications about execution profiles, graphical abstract displays and test coverage.

A Mercury execution events

The Mercury trace is an adaptation of Byrd's box model (Byrd, 1980). In this section, we describe the Mercury execution events that constitute the Mercury execution trace. More information about the Mercury tracer can be found in (Somogyi & Henderson, 1999). The different *attributes* provided by the Mercury tracer are:

1. *Chronological event number* (`chrono`[11]). Each event has a unique event number according to its rank in the trace. It is a counter of events.
2. *Goal invocation number* or *call number* (`call`). Unlike chronological event number, several events have the same goal invocation number. All events related to a given goal have a unique goal number given at invocation time.
3. *Execution depth* (`depth`). It is the depth of the goal in the proof tree, namely the number of its ancestor goals + 1.
4. *Event type or port* (`port`). We distinguish between *external events* that occur at procedure entries and exits, which are the traditional ports introduced by Byrd (Byrd, 1980), and the *internal events* which refers to what is occurring inside a procedure. External events are:
   - `call` a new goal is invoked
   - `exit` the current goal succeeds
   - `fail` the current goal fails
   - `redo` another solution for the current goal is asked for on backtracking.
   - `exception` the execution raises an exception

   Internal events are:
   - `disj` the execution is entering a branch of a disjunction
   - `switch` the execution is entering a branch of a switch (a *switch* is a disjunction in which each branch unifies a ground variable with a different function symbol. In that case, at most one disjunction provides a solution).
   - `if` the execution is entering the condition branch of an if-then-else
   - `then` the execution is entering the "then" branch of an if-then-else
   - `else` the execution is entering the "else" branch of an if-then-else
   - `first` the execution enters a C code fragment for the first time
   - `later` the execution re-enters a C code fragment
5. *Determinism* (`det`). It characterizes the number of potential solutions for a given goal. The determinism markers of Mercury are: `det` for procedures which have exactly 1 solution, `semidet` for those which have 0 or 1 solution, `nondet` for those which have any number of solutions, `multi` for those which have at least 1 solution, `failure` for those that have no solution, and `erroneous` for those which lead to a runtime error.

[11] The names of the attribute accessing functions are in bold in between parentheses.

6. *Procedure* (`proc`). It is defined by:
   - *a flag* telling if the procedure is a function or a predicate (`proc_type`)
   - *a definition module* (`def_module`)
   - *a declaration module* (`decl_module`) The declaration module is the module where the user has declared the procedure. The defining module is the module where the procedure is effectively defined from the compiler point of view. They may be different if the procedure has been inlined.
   - *a name* (`name`)
   - *an arity* (`arity`)
   - *a mode number* (`mode_number`).
     The mode number is an integer coding the mode of the procedure. When a predicate has only one mode, the mode number of its corresponding procedure is 0. Otherwise, the mode number is the rank in the order of appearance of the mode declaration.
7. *List of live arguments* (`args`). A variable is *live* at a given point of the execution if it has been instantiated and if the result of that instantiation is still available in the runtime system. Destructive input (`di` mode), for example, are not kept until the procedure exits.
8. *List of live Argument types* (`arg_types`).
9. *List of local live variables* (`local_vars`). Some live variables are not arguments of the current procedure.
10. *Goal path* (`goal_path`). The goal path indicates in which part of the code the current internal event occurs. `if`, `then` and `else` branches of an *if-then-else* are denoted by `?`, `e` and `t` respectively; *conjuncts*, *disjuncts* and *switches* are denoted by `ci`, `di` and `si`, where `i` is the conjunct (resp. disjunct, switch) number. For example, if an event with goal path `[c3, e, d1]` is generated, it means that the event occurred in the first branch of a disjunction, which is in the else branch of an if-then-else, which is in the third conjunction of the current goal. External events have an empty goal path.
    The event structure is illustrated by Figure A 1. The displayed structure is related to an event of the execution of a `qsort` program which sorts the list of integers `[3, 1, 2]` using a *quick sort* algorithm. The information contained in that structure indicates that procedure `qsort:partition/4-0`[12] is currently invoked, it is the tenth trace event being generated, the sixth goal being invoked, and it has four ancestors (depth is 5). At this point, only the first two arguments of `partition/4` are instantiated: the first one is bound to the list of integers `[1, 2]` and the second one to the integer `3` ; the third and fourth arguments are not live, which is indicated by the atom '-'. There are two live local variables: `H`, which is bound to the integer `1`, and `T`, which is bound to the list of integers `[2]`. The goal path tells that this event occurred

[12] `'-0'` denotes the mode number; here, `'0'` means that `qsort` was declared with only one mode (namely, `:- mode qsort(in, in, out, out) is det`). If more than one mode is declared, `'0'` denotes the first mode, `'1'` the second one, etc.

```
chrono         10
call           6
depth          5
port           then
det            det
  proc_type    predicate
  def_module   qsort
  decl_module  qsort
  name         partition
  arity        4
  mode_number  0
arg            [ [1, 2], 3, -, - ]
arg_types      [list(int), int, -, - ]
local_vars     [ live_var("H", 1, int), live_var("T", [2], list(int)) ]
goal_path      [s1, c2, t]
```

Figure A 1. A Mercury trace event

in the `then` branch (`t`) of the second conjunction (`c2`) of the first `switch` (`s1`) of `partition/4`.

B The Mercury queens program

```
:- module queens.

:- interface.

:- import_module io.

:- pred main(io__state, io__state).
:- mode main(di, uo) is cc_multi.

:- implementation.

:- import_module list, int.

main -->
    ( { data(Data), queen(Data, Out) } ->
        io__write_string("A 5 queens solution is "), print_list(Out)
    ;
        io__write_string("No solution\n")
    ).

:- pred data(list(int)).
:- mode data(out) is det.

:- pred queen(list(int), list(int)).
:- mode queen(in, out) is nondet.

:- pred qperm(list(T), list(T)).
:- mode qperm(in, out) is nondet.

:- pred qdelete(T, list(T), list(T)).
:- mode qdelete(out, in, out) is nondet.
```

```
:- pred safe(list(int)).
:- mode safe(in) is semidet.

:- pred nodiag(int, int, list(int)).
:- mode nodiag(in, in, in) is semidet.

data([1,2,3,4,5]).

queen(Data, Out) :-
    qperm(Data, Out),
    safe(Out).

qperm([], []).
qperm([X|Y], K) :-
    qdelete(U, [X|Y], Z),
    K = [U|V],
    qperm(Z, V).

qdelete(A, [A|L], L).
qdelete(X, [A|Z], [A|R]) :-
    qdelete(X, Z, R).

safe([]).
safe([N|L]) :-
    nodiag(N, 1, L),
    safe(L).

nodiag(_, _, []).
nodiag(B, D, [N|L]) :-
    NmB is N - B,
    BmN is B - N,
    ( D = NmB ->
        fail
    ; D = BmN ->
        fail
    ;
        true
    ),
    D1 is D + 1,
    nodiag(B, D1, L).

:- pred print_list(list(int), io__state, io__state).
:- mode print_list(in, di, uo) is det.

print_list(Xs) -->
    ( { Xs = [] } ->
        io__write_string("[]\n")
    ;
        io__write_string("["),
        print_list_2(Xs),
        io__write_string("]\n")
    ).

:- pred print_list_2(list(int), io__state, io__state).
:- mode print_list_2(in, di, uo) is det.

print_list_2([]) --> [].
print_list_2([X|Xs]) -->
    io__write_int(X),
    ( { Xs = [] } ->
        []
    ;
        io__write_string(", "),
        print_list_2(Xs)
```

```
).
```

Acknowledgments

We would like to thank Fergus Henderson for his technical support and his many contributing ideas; Pierre Deransart, Baudouin Le Charlier and Olivier Ridoux for fruitful discussions; Jean-Philippe Pouzol for his comments on earlier versions of this article. Finally, we are grateful to the thorough anonymous reviews which have helped us a lot improve the article.

References

Ball, T., & Larus, J.R. (1992). Optimally profiling and tracing programs. *Pages 59–70 of: 19th ACM Symposium on Principles of Programming Languages.*

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

Beizer, B. (1990). *Software testing techniques.* Vol. 2nd ed. Int. Thomson Computer Press.

Bird, R. S. (1987). An introduction to the theory of lists. *Pages 3–42 of:* Broy, M. (ed), *Logic of Programming and Calculi of Discrete Design.* Springer-Verlag.

Brooks, G., Hansen, G.J., & Simmons, S. (1992). A new approach to debugging optimized code. *Pages 1–11 of: SIGPLAN'92 Conference on Programming Language Design and Implementation.*

Byrd, L. (1980). Understanding the control flow of Prolog programs. *Pages 127–138 of:* Tränlund, S.-A. (ed), *Logic Programming Workshop.*

Ducassé, M. (1999a). Coca: An automated debugger for C. *Pages 504–513 of: Proceedings of the 21st Int. Conference on Software Engineering.* ACM Press.

Ducassé, M. (1999b). Opium: An extendable trace analyser for Prolog. *Journal of Logic Programming*, **39**, 177–223. Special issue on Synthesis, Transformation and Analysis of Logic Programs, A. Bossi and Y. Deville (eds).

Ducassé, M., & Noyé, J. (2000). Tracing Prolog programs by source instrumentation is efficient enough. *Journal of Logic Programming*, **43**(2), 157–172.

Eclipse. (1999). *The ECLiPSe Constraint Logic Programming System, ECLiPSe 4.1 - User Manual.* IC.Parc. http://www-icparc.doc.ic.ac.uk/eclipse/.

Eisenstadt, M., & Brayshaw, M. (1988). The Transparent Prolog Machine TPM: an execution model and graphical debugger for Logic Programming. *Journal of Logic Programming*, **5**(4), 277–342.

Eustace, A., & Srivastava, A. (1995). ATOM: A flexible interface for building high performance program analysis tools. *Winter 1995 USENIX Conference.*

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

Gallagher, K.B., & Lyle, J.R. (1991). Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, **17**(8), 751–761.

Hatton, L. (1997). Does OO sync with the way we think? *IEEE Software*, **15**(3), 46–54.

Howden, W.E., & Shi, G.M. (1996). Linear and structural event sequence analysis. *Pages 98–106 of:* Zeil, Steven J. (ed), *Proceedings of the 1996 Int. Symposium on Software Testing and Analysis.* ACM Press.

Hutton, G. (1999). A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, 355–372.

Jahier, E. (2000a). *Analyse dynamique de programmes : mise en oeuvre automatisée d'analyseurs performants et spécification de modèles d'exécution.* Ph.D. thesis, INSA de Rennes. Partly in English.

Jahier, E. (2000b). Collecting graphical abstract views of Mercury program executions. *Pages 100–115 of: Proceedings of the 4th Int. Workshop on Automated Debugging.*

Jahier, E., & Ducassé, M. (1999). A generic approach to monitor program executions. *Pages 139–153 of:* Schreye, D. De (ed), *Int. Conference on logic programming.* MIT Press.

Jahier, E., & Ducassé, M. (1999a). *Morphine 0.2 User and Reference Manuals.*

Jahier, E., & Ducassé, M. (1999b). Un traceur d'exécutions de programmes ne sert pas qu'au débogage. *Pages 297–311 of:* Fages, F. (ed), *Actes des Journées Francophones de Programmation Logique et par Contraintes.* Hermès.

Jeffery, C. (1999). *Program monitoring and visualization.* Springer. ISBN 0-387-98644-8.

Jeffery, C., Zhou, W., Templer, K., & Brazell, M. (1998). A lightweight architecture for program execution monitoring. *ACM SIGPLAN Notices*, **33**(7), 67–74.

Kishon, A., & Hudak, P. (1995). Semantics directed program execution monitoring. *Journal of Functional Programming*, **5**(4), 501–547.

Kishon, A., Hudak, P., & Consel, C. (1991). Monitoring semantics: a formal framework for specifying, implementing and reasoning about execution monitors. *ACM SIGPLAN Notices*, **26**(6), 338–352.

Koutsofios, E., & North, S. (1991). *Drawing graphs with* dot. TR 910904-59113-08TM. AT&T Bell Laboratories.

Olsson, R.A., Crawford, R.H., & Ho, W.W. (1990). Dalek: A GNU, improved programmable debugger. *Pages 221–232 of:* USENIX Association (ed), *Summer 1990 USENIX Conference.* USENIX Association.

Patil, H., & Fischer, C. (1997). Low-cost, concurrent checking of pointer and array accesses in C programs. *Software Practice and Experience*, **27**(1), 87–110.

Pfenning, F. (ed). (1992). *Types in Logic Programming.* MIT Press. ISBN 0-262-16131-1.

Somogyi, Z., & Henderson, F. (1999). The implementation technology of the Mercury debugger. *Pages 35–49 of: Proceedings of the 10th Workshop on Logic Programming Environments*, vol. 30(4). Elsevier, Electronic Notes in Theoretical Computer Science.

Somogyi, Z., Henderson, F., & Conway, T. (1996). The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, **29**, 17–64.

Stasko, J., Domingue, J., Brown, M. H., & Price, B. A. (eds). (1998). *Software visualization: Programming as a multimedia experience.* MIT Press.

Templer, K.S., & Jeffery, C.L. (1998). A configurable automatic instrumentation tool for ANSI C. *Pages 249–259 of: Proceedings of the Automated Software Engineering Conference.* IEEE Computer Society.

Tip, F. (1995). Generic techniques for source-level debugging and dynamic program slicing. *Pages 516–530 of:* Mosses, Peter D., Nielsen, Mogens, & Schwartzbach, Michael I. (eds), *TAPSOFT '95: Theory and Practice of Software Development.* LNCS, vol. 915. Springer-Verlag.

Tolmach, A., & Appel, A.W. (1995). A debugger for Standard ML. *Journal of Functional Programming*, **5**(2), 155–200.

Van Roy, P., & Despain, A. M. (1992). High-performance logic programming with the Aquarius Prolog compiler. *Computer*, **25**(1), 54–68.