An Automated Debugger for Mercury - Opium-M 0.1 User and Reference Manuals

Mireille Ducassé, Erwan Jahier

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

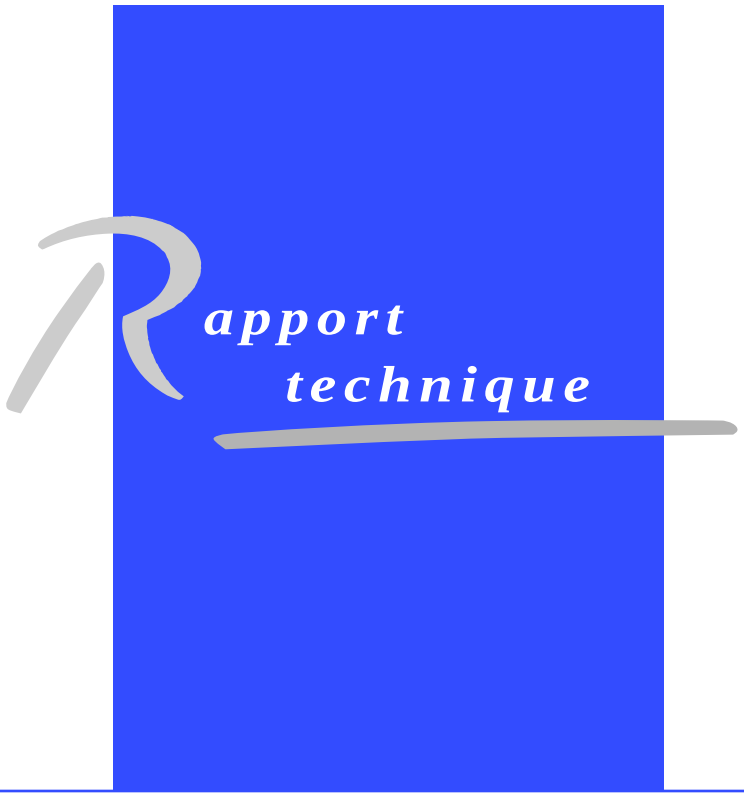
An automated debugger for Mercury - Opium-M 0.1 User and Reference Manuals.

Mireille Ducassé and Erwan Jahier, IRISA/INSA

No 0231

mai 1999

THÈME 2



ISSN 0249-0803

An automated debugger for Mercury - Opium-M 0.1 User and Reference Manuals.

Mireille Ducassé and Erwan Jahier, IRISA/INSA *

Thème 2 — Génie logiciel
et calcul symbolique
Projet Lande

Rapport technique n ° 0231 — mai 1999 — 104 pages

Abstract: This document gathers the user manual and the reference manual of Opium-M, an analyser of execution traces of Mercury Programs. Opium-M is an adaptation to Mercury of Opium a trace analyser for Prolog.

Mercury is a new logic programming language. Its type, mode and determinism declarations enable codes to be generated that is at the same time more efficient and more reliable than with current logic programming languages. The deterministic parts of Mercury programs are as efficient as their C counterparts. Moreover, numerous mistakes are detected at compilation time. However, our industrial partner experience shows that the fewer remaining mistakes, the harder they are to be diagnosed. A high-level debugging tool was thus necessary.

Program execution traces given by traditional debuggers provide programmers with useful pieces of information. However, using them requires to analyse by hand huge amounts of information.

Opium-M is connected to the traditional tracer of Mercury, it allows execution trace analyses to be automated. It provides a relational trace query language based on Prolog which enables users to specify precisely what they want to see in the trace. Opium-M, then, automatically filters out information irrelevant for the users.

Key-words: Logic programming, Mercury, Trace analyser, Trace query language, Automated debugging, User manual, Reference manual.

(Résumé : tsvp)

Sponsored by ARGo, Esprit Industrial RTD Project No 25503

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

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

Un débogueur automatisé pour Mercury - Manuels utilisateur et de référence d'Opium-M 0.1.

Résumé : Ce document rassemble le manuel utilisateur et le manuel de référence d'Opium-M, un analyseur de traces d'exécutions de programmes Mercury. Opium-M est l'adaptation à Mercury d'Opium un analyseur de traces pour Prolog.

Mercury est un nouveau langage de programmation logique. Ses déclarations de type, de mode et de déterminisme permettent de générer un code à la fois plus efficace et plus sûr qu'avec les langages de programmation logique actuels. Les parties déterministes des programmes Mercury sont aussi rapides que leurs équivalents en C. De plus, beaucoup de fautes sont détectées dès la compilation. L'expérience de nos partenaires industriels montre cependant que les fautes résiduelles sont d'autant plus difficiles à localiser et à comprendre qu'elles sont peu nombreuses. Un outil de débogage de haut niveau était donc nécessaire.

Les traces d'exécutions de programmes données par des débogueurs traditionnels donnent des informations utiles aux programmeurs, mais leur utilisation demande d'analyser à la main de très grandes quantités d'informations.

Opium-M est connecté au traceur traditionnel de Mercury et permet d'automatiser l'analyse des traces d'exécutions. Opium-M fournit un langage de requête relationnel basé sur Prolog qui permet aux programmeurs de spécifier de manière précise ce qu'ils veulent voir dans la trace. Opium-M élimine alors automatiquement les informations non pertinentes pour les programmeurs.

Mots-clé : Programmation logique, Mercury, Analyseur de traces d'exécutions, Langage de requête sur des traces, Débogage automatisé, Manuel utilisateur, Manuel de référence.

Contents

Part I
Opium-M – User Manual

1 Introduction

Opium-M is an extensible debugging environment for Mercury [7, 8] which offers support for high-level debugging strategies. A broad scope of Mercury programmers ranging from beginners to experienced Mercury users should find accurate debugging support in Opium-M. This introduction lists potential users of Opium-M and briefly describe its basic features.

Opium-M is an adaptation to Mercury of the Opium debugger which was designed for Prolog. A number of articles describe in detail the principles of Opium, see in particular [2, 4] [1]. An adaptation of Opium to C called coca is described in [3].

1.1 Why should you use Opium-M?

If you think that simply stepping through execution traces is all you need, you should still try Opium-M because with a handful of commands plus Prolog you can specify easily the *very* trace line(s) you want to see. Start with Section 3 which explains the minimum you need to know in order to run Opium-M.

If your program has huge data structures and you would like to have them abstracted with several levels of abstractions, Opium-M has powerful general mechanisms to do this. Furthermore, it will help you program dedicated display procedures and integrate them in the environment.

If you have always been complaining that tracers were not powerful enough and longing for a debugger that will enable you to write powerful macros, then Opium-M should have all that you could wish for. It is not only macros but real debugging programs that you can easily write. Section 5 explains how to tailor the debugging environment to your taste and needs.

If you are writing an application which in turn requires some debugging facilities, then Opium-M should be a great help. You can provide your users with tracing and debugging facilities at the proper level of abstraction without touching your source code! Section 6 tells you how to extend Opium-M.

1.2 Basic features of Opium-M

In this section, each of the following features is detailed in turn. Opium-M is fully programmable and the programmed commands are grouped into "scenarios". The environment is extensible and customizable. Opium-M is running in coprocessing with the debugged session.

[1] Preprint versions of the articles are accessible from http://www.irisa.fr/lande/ducasse/

1.2.1 Programmability

Tracers usually print the information they retrieve. By contrast, Opium-M handles the debugging information as *programming data*. The debugging information (trace and source) can then be analysed by programs before it is printed.

Some tracers provide "macros", Opium-M offers a full *programming language* to implement debugging commands and programs. Opium-M's language is Prolog extended by a set of debugging primitives. Backtracking and unification enable the user to write concise and powerful commands on the fly.

1.2.2 Scenario paradigm

Tracers usually provide a flat set of commands. In Opium-M, the set of debugging commands is structured into *scenarios*. Each scenario corresponds to a certain *debugging strategy*, or to a certain aspect of debugging. For example, there is a scenario for examining the traced execution step-by-step, and there is a scenario supporting the user on examining the source code. Hence learning Opium-M and navigating through it is relatively easy although the number of (interesting) functionalities is high. The scenario structure is only a "virtual" structure to help people use Opium-M. Any command visible from the current module can be executed at any time. The scenario paradigm is also the basis of the customization and extensibility mechanisms (see sections 5 and 6).

1.2.3 Customization and extensibility

Thanks to Prolog, new predicates can be easily added and directly used as commands. Furthermore, existing commands can be *customized* without knowing the details of their implementation. They can also be reset to their default values. New scenarios can be added. With a minimum of declarations, Opium-M ensures that the new scenario is *integrated* into the system. Interface, help, manual, and some consistency checking will be added automatically (see sections 5 and 6).

2 Understanding the Mercury trace

2.1 Introduction

Mercury programs are heavily transformed before any code is generated. Procedures are put into *superhomogeneous form*, high order predicates are transformed into first order predicates, some procedures can be inlined; under the commutative semantics, conjunctions and disjunctions can be evaluated in any order. Hence, it is sometimes difficult to relate Mercury source code with its corresponding trace, even if the program is compiled using the strict sequential semantics with all the optimizations switched off. In order to relate the trace with the initial program, it might sometimes be helpful to have a look at the HLDS code (see the Mercury User and Language Reference Manuals [6, 5]).

Opium-M is connected to the traditional tracer of Mercury. We describe in sub-sections 2.2 to 2.4 the information contained in the trace and we specify in what order each type of events occurs. These descriptions give another point of view of the "Tracing Mercury Programs" section of the Mercury user manual[2].

2.2 Trace event representation

We call *execution events* particular points of the program execution where information concerning the current state of the execution can be retrieved. Mercury events contain several kinds of information: information describing the *control flow* of the execution, information about the variable instantiations (*data* information), and information that refers to source code (*source connection information*).

A trace is therefore a sequence of events, where each event can be seen as a tuple of a relational database. This trace model is the basis of the trace query mechanism.

The different *event attributes* are as follows. The identifiers in teletype font are the names used in Opium-M to denote event attributes.

Control flow information

- *Chronological event number* (`chrono`) attached to an event.

  Each event has a unique event number according to its rank in the trace. It is a counter of events.

- *Goal invocation number* or *call number* (`call`) attached to a goal.

  All events related to a given goal have a unique goal number given at invocation time.

- *Execution depth* (`depth`) attached to a goal.

  It is the depth of the goal in the proof tree, namely the number of its ancestor goals + 1.

- *Event type or port* (`port`) attached to an event.

  The different Mercury event types are as follows. We distinguish between external events which are the traditional ports introduced by Byrd [1], and the internal events which refers to what is occurring inside a procedure.

  External events

  - `call` a new procedure is invoked
  - `exit` the current procedure succeeds
  - `fail` the current procedure fails
  - `redo` another solution for the current procedure is asked for on backtracking, and a subgoal of the procedure has a choice point.

[2] available at "http://www.cs.mu.oz.au/research/mercury/"

Internal events:

- `disj` the execution is entering a branch of a disjunction
- `switch` the execution is entering a branch of a switch
- `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
- `exception` the execution raises an exception

- *Determinism* (`det`) attached to a goal.

  The different Mercury determinism marks are as follows. They characterize the number of potential solutions for a given goal.

  - `det` exactly one solution
  - `semidet` 0 or 1 solution
  - `nondet` 0 or any number of solutions
  - `cc_nondet` 0 or one solution
  - `multi` at least one solution
  - `cc_multi` exactly one solution
  - `failure` no solution
  - `erroneous` leads to a runtime error

- *Procedure* (`proc`) attached to a goal. A procedure 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`)
  - *a name* (`name`)
  - *an arity* (`arity`) and a
  - *a mode number* (`mode_number`).

    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 vue. They may be different if the procedure has been inlined.

    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.

Data information

- *List of live Arguments* (`args`) attached to a goal and an event.

  It gives the current instantiation of the live procedure arguments. We say that 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. When it is known that a variable will not be used again, its content is not usually kept by the Mercury compiler. However, when tracing is on, the content of input arguments are kept live until the procedure exits, destructive input (`di` mode) arguments excepted.

- *List of live Argument types* (`arg_types`) attached to a goal and an event

  It gives the types of the live procedure arguments.

- *List of local live variables* (`local_vars`)

  It gives the instantiation, the types and the names of live local variables that are not arguments of the current procedure.

Source connection information

- *Goal path* (`GoalPath`) attached to a goal and an internal event.

  The goal path indicates in which part of the code the current event occurs. *then* and *else* branches of an *if-then-else* are denoted by `e` and `t`; *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.

2.3 An example of Mercury event

The event structure is illustrated by Figure 1. The displayed structure is related to an event of the execution of the `qsort` program which sorts the list of integers `[3, 1, 2]` using a *quick sort* algorithm. The full code of qsort.m is given in Appendix A. The information contained in that structure indicates that procedure `qsort:partition/4-0` is currently invoked, it is the 10^{th} trace events being generated, the 6^{th} goal being invoked, and it has 4^{th} ancestors (depth is 5). At this point, only the first two arguments of `partition/4` are instantiated: the first one is bounded to the list of integer `[1, 2]` and the second one to the integer `3` ; third and fourth variable are not live which is indicated by the '-'. The two live local variables are `H`, bound to the integer `1`, and `T` bound to the list of integer `[2]`. This event occurred in the `then` branch of the second conjunction of the first `switch` of the code of `partition/4`.

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__list(int), int, -, -]
local_vars	[[live_var("H", 1, int), live_var("T", [2], list(int))
goal_path	[s1, c2, t]

The default display of this event is:

```
10:     6 [5] then partition([1, 2], 3, -, -) [s1, c2, t]
```

Figure 1: An example of Mercury event

2.4 The sequence of events

We call *trace* of a program execution a sequence of events of the same format; in a sense, a trace is a discretisation of the execution. The Mercury trace can be seen as an extended version of the Byrd box model [1] that binds execution events to goals. In this sub-section, we give a specification of the Mercury trace from the point of view of a given goal in extended BNF notation. Depending on the determinism of the procedure, different sequences of events will be generated.

- det procedures:
  `<det> --> call {disj | switch | then | else}* exit`
- semidet procedures:
  `<semidet> --> call {disj | switch | then | else}* {exit | fail}`
- multi procedures:
  ```
  <multi> --> call <multi_end>
  <multi_end> --> {disj | switch | then | else}*
       {exit redo <multi_end> | fail}
  ```

- cc_multi procedures:
  `<cc_multi> --> call {disj | switch | then | else}* exit`
- nondet procedures:
  ```
  <nondet> --> call <nondet_end>
  <nondet_end> --> {disj | switch | then | else}*
       {exit redo <nondet_end> | exit | fail}
  ```
- cc_nondet procedures:
  ```
  <cc_nondet> --> call {disj | switch | then | else}*
       {exit | fail}
  ```
- failure procedures:
  `failure  --> call fail`
- erroneous procedures:
  They will not generate any event.
- C code fragments:
  `<c_code> --> first {later}*`

2.5 An example of Mercury trace

Figure 2 shows an exhaustive trace of the execution of the `qsort` program (see Appendix A) on the list `[3, 1,2]`. In this trace, we only print the following event attributes: `Chrono :Call [Depth] Port Proc/Arity ArgList GoalPath`. We also filtered out all the procedures that were defined in the module `printlist.m` which are not of interest here.

Note that input argument of main procedure is not available at exit ports (event `65`) since it is destructively updated. Note also the hole between events `48` and `65` that corresponds to events occurring in `printlist.m` module.

3 Getting started with Opium-M

3.1 Setting up the Unix environment

The current version of Opium-M runs with a Mercury release of the day later than April 99[3]. Currently, it has been tested on sparc-sun/Solaris 2.{5,6,7} and on pc-i686/Linux-2.0.34, but it should work on any other unix systems. It requires the ECLiPSe[4] Prolog system; it has been tested with ECLiPSe 3.4.3, 3.5.2, 4.O and 4.1.

Note: if you are using ECLiPSe 4.0 or a previous version, you need to add the following line in your `.opium-m-rc` file:

[3] you can download a release of the day at "http://www.cs.mu.oz.au/research/mercury/download/rotd.html"

[4] Free licence for academic sites and free trial 6 month licence are available from http://www.ecrc.de/eclipse/eclipse.html

```
[Opium-M]: fget([module = not(printlist)]), fail.
  1: 1 [1] call main(state('<<c_pointer>>'), -)
  2:  2 [2] call main3(-, state('<<c_pointer>>'), -)
  3:   3 [3] call main1(-)
  4:    4 [4] call data(-)
  5:    4 [4] exit data([3, 1, 2])
  6:    5 [4] call qsort([3, 1, 2], -, [])
  7:    5 [4] switch qsort([3, 1, 2], -, []) [s1]
  8:     6 [5] call partition([1, 2], 3, -, -)
  9:     6 [5] switch partition([1, 2], 3, -, -) [s1]
 10:     6 [5] then partition([1, 2], 3, -, -) [s1,c2,t]
 11:      7 [6] call partition([2], 3, -, -)
 12:      7 [6] switch partition([2], 3, -, -) [s1]
 13:      7 [6] then partition([2], 3, -, -) [s1,c2,t]
 14:       8 [7] call partition([], 3, -, -)
 15:       8 [7] switch partition(-, -, -, -) [s2]
 16:       8 [7] exit partition([], 3, [], [])
 17:      7 [6] exit partition([2], 3, [2], [])
 18:     6 [5] exit partition([1, 2], 3, [1, 2], [])
 19:     9 [5] call qsort([], -, [])
 20:     9 [5] switch qsort(-, -, []) [s2]
 21:     9 [5] exit qsort([], [], [])
 22:     10 [5] call qsort([1, 2], -, [3])
 23:     10 [5] switch qsort([1, 2], -, [3]) [s1]
 24:      11 [6] call partition([2], 1, -, -)
 25:      11 [6] switch partition([2], 1, -, -) [s1]
 26:      11 [6] else partition([2], 1, -, -) [s1,c2,e]
 27:       12 [7] call partition([], 1, -, -)
 28:       12 [7] switch partition(-, -, -, -) [s2]
 29:       12 [7] exit partition([], 1, [], [])
 30:      11 [6] exit partition([2], 1, [], [2])
 31:      13 [6] call qsort([2], -, [3])
 32:      13 [6] switch qsort([2], -, [3]) [s1]
 33:       14 [7] call partition([], 2, -, -)
 34:       14 [7] switch partition(-, -, -, -) [s2]
 35:       14 [7] exit partition([], 2, [], [])
 36:       15 [7] call qsort([], -, [3])
 37:       15 [7] switch qsort(-, -, [3]) [s2]
 38:       15 [7] exit qsort([], [3], [3])
 39:       16 [7] call qsort([], -, [2, 3])
 40:       16 [7] switch qsort(-, -, [2, 3]) [s2]
 41:       16 [7] exit qsort([], [2, 3], [2, 3])
 42:      13 [6] exit qsort([2], [2, 3], [3])
 43:      17 [6] call qsort([], -, [1, 2, 3])
 44:      17 [6] switch qsort(-, -, [1, 2, 3]) [s2]
 45:      17 [6] exit qsort([], [1, 2, 3], [1, 2, 3])
 46:     10 [5] exit qsort([1, 2], [1, 2, 3], [3])
 47:    5 [4] exit qsort([3, 1, 2], [1, 2, 3], [])
 48:   3 [3] exit main1([1, 2, 3])
 64:  2 [2] exit main3([1, 2, 3], -, state('<<c_pointer>>'))
 65: 1 [1] exit main(-, state('<<c_pointer>>'))
```


Figure 2: Execution trace of qsort([2, 1, 3])

```
:- set_parameter(socket_domain, [inet]).
```

It will make Opium-M use the internet sockets even if the Mercury and the Opium-M process run on the same machine. Since internet sockets make the communication between the two processes a little bit slower, you should remove better remove those lines once you have upgraded to ECLiPSe 4.1.

3.2 Building your Mercury compiler

To be able to run Mercury programs under the control of Opium-M, you need to compile your Mercury program as for mdb, the internal Mercury debugger. There are basically two methods:

1. `mmc --trace <trace_option> <program_name>`

   where `<trace_option>` can be `none`, `shallow`, `deep` or `default`.

   Example:

   ```
   mmc --trace deep hello.m
   ```

2. `mmc --grade <debug_grade> <program_name>` or
   `mmc -s <debug_grade> <program_name>`

   where `<debug_grade>` is a grade that includes the word `debug`.

   Example: if your Mercury compiler has been built in the asm_fast.gc.tr.debug grade:

   ```
   mmc -s asm_fast.gc.tr.debug hello.m
   ```

   or

   ```
   mmake hello.depend
   mmake -s asm_fast.gc.tr.debug hello.m
   ```

For more details about preparing a program for debugging, please refer to: http://www.cs.mu.oz.au/research/mercury/information/doc/user_guide_7.html, section "Preparing a program for debugging".

3.3 Starting an Opium-M session

In your working window type "Opium-M"; you enter Eclipse extended by the Opium-M libraries. Note that you can use it as a simple Prolog session.

```
...
Loading /home/swann/d01/lande/jahier/.opium-m-rc...
ECLiPSe Constraint Logic Programming System [kernel]
Version 4.1.0, Copyright IC-Parc and ICL, Sun Feb 21 18:37 1999
[Opium-M]:
```

To run the Mercury program hello (that you can find in the "sample" directory of your Mercury distribution), use the command run/1.

```
[Opium-M]: run(hello)
 1: 1 [1] CALL main(state('<<c_pointer>>'), -)
```

A complete example session can be found in Section 4.

Note that to quit an Eclipse session, you need to type `^d`.

3.4 A first try with Opium-M

You can start to enter debugging commands in the Opium-M session.

We recommend to start with the `step_by_step_M` scenario, and in particular with the following commands to trace the program execution (the abbreviations of the commands are given in curly braces):

`print_event {p}` print the current trace event.

`next {n}` print the following trace event.

Next, you can try a simple mechanism which provides you with the facilities to trace a part of the execution, and to stop on a certain trace event. This mechanism is called *trace-till-condition*. It is based on the fact that those commands which can be used to move forwards in the trace (like `next`) are *backtrackable*. They can be used in the following way:

```
[Opium-M]: next, <condition>.
```

This debugging goal will cause Opium-M to print the following trace events, until the condition is fulfilled. The condition can be any Prolog goal. In order to specify a condition depending on the values of the trace events (see section 2.2), you can use the `current` command which is described in the Reference Manual (Part II). Try for example:

```
[Opium-M]: next, current(port = exit).
```

Note that unification can help to specify precisely when the condition shall succeed. For example, if you want to trace until predicate `permutation/2` is called and where the first argument is a list of length 2, you can use the following Opium-M goal:

```
[Opium-M]: next, current(proc = permutation/2 and arg = [[A1,A2], _]).
```

If you want to skip a part of the trace, and to specify more precisely which trace events you actually want to see, this can be achieved by using the *skip-till-condition* mechanism. It is based on a set of primitives which enable moving forwards in the trace like the commands

listed above, but *without actually printing any trace event* (indicated by the ending `_np` = "no print"):

`next_np {n_np}` move forwards to the next trace event.

In order to skip a part of the trace till a certain condition is fulfilled, you can enter for example

```
[Opium-M]: next_np,
   current(proc = sort/2 and port = exit and arg = [-, List]),
   length(List, Length),
   Length > 7,
   print_event.
```

This will repeat to move forwards to the next trace event without printing it, until the condition is true, and then print the current trace event. Here, the condition is that the second argument of procedure `sort/2` should be a list of length greater than 7 at exit ports. Note the use of the "-" in the arguments list that avoids us to retrieve the first argument as we do not need it; retrieve arguments one by one may be useful if some unused arguments are very big.

The sophisticated command of Opium-M is fget/1; it filters efficiently through execution traces. Section 4 illustrates in depth how to use it and the Reference manual (Part II page 47) describes it.

3.5 Getting some help

You can use the on-line help in scenario `help` to get a survey of the other debugging strategies provided by Opium-M. Start by calling command `help/0` which will tell you how to use the on-line help and will list the different commands of the help scenario. You can get some help on the Opium-M commands, primitives and procedures thanks to man/1 command and get some help on the ECLiPSe specific commands with help/1. Note that you can use the commands and primitives of *all* the scenarios visible in the current module. The scenario paradigm is only a virtual structure which does not restrict you to the use of one scenario at a time.

4 A debugging session with Opium-M

In this section we show a session illustrating Opium-M facilities. The commands used in the following are described in the reference manual (see Part II page 47). You can also get an on-line description of those commands by invoking the man/1 command within Opium-M. Note that commands that moves to an event and print a trace line (like `fget/1`) have a corresponding command which moves to this event without printing any trace line (`fget_np/1`).

In the following example, the full names of the commands are used to ease the understanding. However, most of them have abbreviations which can be used to save typing. Whenever queries repeat part of previous ones, the corresponding abbreviations are used.

The trace queries typed in by the user are the goals following the Opium-M prompt (`[Opium-M]:`). All the rest is written by the system.

The analyzed program is the buggy mastermind program given in Appendix B. Here is what happens when we run the buggy mastermind program:

```
%mastermind
Enter a mastermind problem : mastermind(red, red, red, red, white).

The solution was found with 5 guesses.
The successive guesses were :
     1: blue blue blue blue blue (0 Bulls and 3 Cows).
     2: white white white white black (0 Bulls and 3 Cows).
     3: yellow yellow yellow yellow white (1 Bulls and 3 Cows).
     4: yellow gray gray gray gray (0 Bulls and 3 Cows).
     5: red red red red white (5 Bulls and 0 Cows).
```

The Tcl/TK output of this program is given in Figure 3. The numbers of cows is wrong; we call Opium-M to understand why.

```
%Opium-M

[...]

Loading /home/swann/d01/lande/jahier/.opium-m-rc...
ECLiPSe Constraint Logic Programming System [kernel]
Version 4.1.0, Copyright IC-Parc and ICL, Sun Feb 21 18:37 1999
[Opium-M]:
```

To start the execution of the mastermind within Opium-M, we invoke the run/1 command.

```
[Opium-M]: run(mastermind).

Start debugging mastermind program.
```

Before starting to display trace lines, we way wonder what attributes are displayed (`slot_dislay` parameter). We can get that information with the `print_displayed_attributes/0` command.

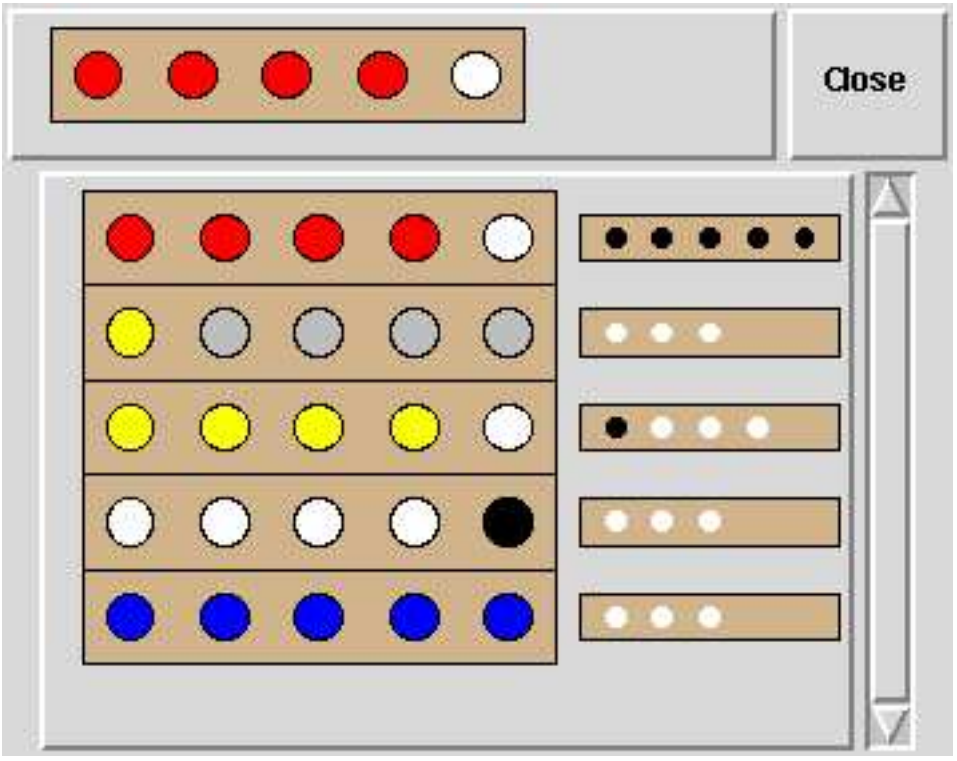


Figure 3: Board output of the buggy mastermind program

```
[Opium-M]: print_displayed_attributes.

 port name(arg) goal_path
```

As we would like to display the chrono number too, we toggle that slot.

```
[Opium-M]: toggle(chrono), print_displayed_attributes.

chrono:  port name(arg) goal_path
```

Now on, each trace line will display the chrono, port, name, args, and goal_path attributes. All the existing attributes are described in section 2.2 and can be retrieved on demand using the current/1 command.

The top level predicate that counts bulls and cows from a code and a submission is `check_mastermind/4`. Let us go to the event where this predicate first exits to see if it exits with sensible argument values.

We invoke the `fget/1` command to move to the events where that predicate exits.

```
[Opium-M]: fget(name = check_mastermind and port = exit).

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

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

Note that fget has filtered aver one million trace events here. In order to determine more easily whether `check_mastermind` outputs correct values, we can use a little program which displays data in a user-friendly manner (useful with big or cryptic data). In this case, we can reuse the `display_mastermind/4` predicate that is used in mastermind.m to display the mastermind boards. To gather the current values of the arguments, we simply use `current/1` with the attributes `args`.

```
[Opium-M]: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
```

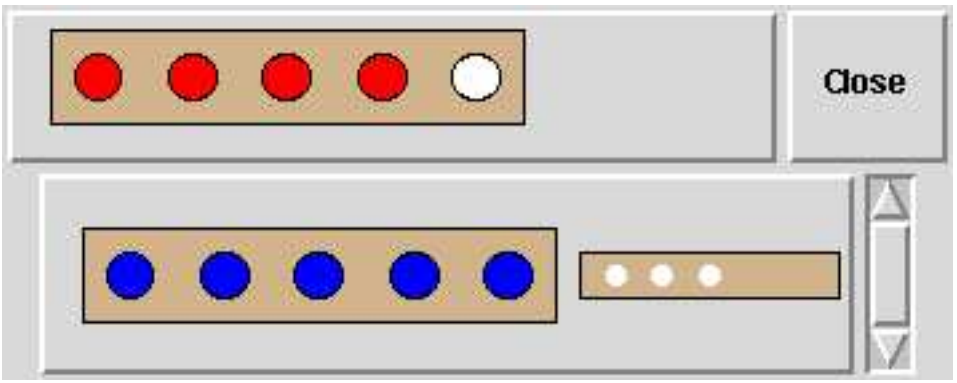


Figure 4: Board output of display_mastermind/4

The output of the call to `display_mastermind` is given in Figure 4.

Indeed the number of cows is wrong. We can investigate source files thanks to the command `listing/2` of the source scenario. Even simpler, when the current predicate is the one we want to source, we can use `listing_current_procedure/0` (`lcp/0`) command. That is what we do here:

```
[Opium-M]: listing_current_procedure.

:- pred check_mastermind(mastermind, mastermind, int, int).
:- mode check_mastermind(in, in, out, out) is det.
check_mastermind(Guess, Code, Bulls, Cows) :-
         count_bulls(Guess, Code, Bulls, Guess2, Code2),
         % Code2 and  Guess2 is the same as Code and Guess except that
         % we have removed the well placed whatsits.
         count_cows(Guess2, Code2, Cows).
```

Since the bug symptom is a bad number of cows, the bug is probably inside the `count_cows/2` predicate. Let us have a look at it.

```
[Opium-M]: listing(mastermind_checker, count_cows).

:- pred count_cows(mastermind, mastermind, int).
:- mode count_cows(in, in, out) is det.
count_cows(Guess, Code, Cows) :-
        % Outputs the number of cows in the Guess.
        Guess = mastermind(G1, G2, G3, G4, G5),
        Code = mastermind(Color1, Color2, Color3, Color4, Color5),
```

```
        List1 = [G1, G2, G3, G4, G5],
        remove_cows_from_list(Color1, List1, List2),
        remove_cows_from_list(Color2, List2, List3),
        remove_cows_from_list(Color3, List3, List4),
        remove_cows_from_list(Color4, List4, List5),
        remove_cows_from_list(Color5, List5, List6),
        list__length(List6, M),
        Cows is 5 - M.
```

The only non-library predicate that is called within `count_cows/2` is the predicate `remove_cows_from_list/3`. So we would like to check if this predicate outputs correct data. We want to go to the exit of the first call to `remove_cows_from_list/3` that occurs inside `count_cows/2`. This can be translated in Opium-M into the following request:

```
[Opium-M]: fget([name = count_cows, port = call]),
        fget([name = remove_cows_from_list, port = exit]),
        current(depth = D).

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? (;) ;

1432633:  exit remove_cows_from_list(hole, [blue], [hole])
D = 13     More? (;)
```

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/3` which outputs the list `[hole]` should have output the list verb+[blue]+.

As *hole* is a special color, we could want to make sure that `remove_cows_from_list/3` is also wrong with normal colors. We can filter out events where the first argument of `remove_cows_from_list/3` is *hole*.

```
[Opium-M]: fget_np([name = remove_cows_from_list, port = exit]),
        current(args = [X, -, -]),
        X \= hole,
        print_event.

1432724:  exit remove_cows_from_list(white, [], [])
```

```
X = white     More? (;) ;
1432725:  exit remove_cows_from_list(white, [hole], [white])

X = white     More? (;)
1432726:  exit remove_cows_from_list(white, [hole, hole], [white, white])
```

That is not exactly what we were looking for. We would like to filter out events where the hole appears in the list of the second argument of `remove_cows_from_list/3` too.

```
[Opium-M]: fget_np([name = remove_cows_from_list, port = exit]),
        current(args = [X, [Y], -]),
        X \= hole,
        Y \= hole,
        print_event.

1434263:  exit remove_cows_from_list(black, [white], [black])

X = black
Y = [white]
Z = white     More? (;)
```

Now we are definitely convinced that `remove_cows_from_list/3` is buggy: here the answer should have been `remove_cows_from_list(black, [white], [white])`.

```
[Opium-M]: 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) :-
        (
                C = X,
                C \= hole
         ->
                ListOut = Xs
        ;
                remove_cows_from_list(C, Xs, List),
                ListOut = [C | List]
        ).
```

Assuming that we still do not see where the bug is, we can do a breath-first trace of this predicate. to do that, we can define ourselves a `zoom/1` command+ that will perform a breath-first trace.

```
[Opium-M]: [user].
zoom(D) :-
         current(depth = D1),
         D2 is D1 + 1,
         fget_np([depth = [D1, D2]]),
         current(depth = D),
         ( D == D2 ->
                 current(port = P),
                 external(port = P)
         ;
                 true
         ),
         print_event.

external_port(call).
external_port(exit).
external_port(fail).
external_port(redo).
^d
 user       compiled traceable 788 bytes in 0.00 seconds
```

Before starting the breath-first trace, we go to an 'interesting' event namely: a call to `remove_cows_from_list` with first argument a color that is not a *hole* and second one a list containing one color that is not a *hole*.

```
[Opium-M]: retry.
1364204:  det remove_cows_from_list(yellow, [white], -) []

[Opium-M]: zoom(D).
1364215:  switch remove_cows_from_list(yellow, [white], -) [s1]

D = 23     More? (;)
1364216:  else remove_cows_from_list(yellow, [white], -) [s1,c2e]

D = 23     More? (;)
1364217:  call remove_cows_from_list(yellow, [], -)

D = 24     More? (;)
```

```
1364219:  exit remove_cows_from_list(yellow, [], [])

D = 24     More? (;)
1364220:  exit remove_cows_from_list(yellow, [white], [yellow])
```

Indeed, `remove_cows_from_list` should output `[X | List]` and not `[C | List]`.

5 Setting up your debugging environment

There are many facilities in Opium-M to let you tailor your environment to your taste and needs. If an existing functionality does not behave exactly as you would like it to do, first check the parameters of the related scenario. You may be able to get the behavior you want by simply setting a parameter appropriately. It is the easiest way to adapt your environment. If there is no parameter which will achieve what you need you may have to add simple commands or to customize existing ones. This Chapter should help you to find your way through the modification and customization facilities.

More details about scenario handling can be found in Chapter 6.

5.1 Profiles

All the modifications and the customizations described in the following can be done on the fly but can also be collected in a file which is then compiled in the Opium-M session. Hence you can have profile files containing Opium-M predicates and directives to set up your environment.

If there is a file ".opium-m-rc" in your home directory it will be compiled every time you enter Opium-M. It should contain the permanent modifications.

In addition, we recommend that you have Opium-M files attached to your applications containing settings dedicated to the applications. These files have to be compiled in Opium-M explicitly when needed.

5.2 Setting parameters

The behavior of many Opium-M commands depends on the values of Opium-M parameters. Modifying the value of parameters is the easiest way to adapt your environment. There are two types of parameters, "single" and "multiple". A parameter of type "single" is a global variable, it has exactly one value. A parameter of type "multiple" is more like a predicate, it may have several values, or it may not be set at all. Most parameters are of type "single".

`show_all(parameters, ScenarioName)` or `show_all(parameters)` will list the existing parameters. Before you try to set a parameter to a new value (`set_parameter/2`) we recommend that you first get the current value (`get_parameter/2`). Some parameters have many arguments or convey long lists of values, of which you may want to tune some only (see the example in the following). You can set a parameter back to its default value using

`set_default/1`. If a parameter is of type "multiple" you have to use `unset_parameter/2` to remove a particular value.

On a tty interface you can either set the parameter interactively with `set_parameter/1` or explicitly using `set_parameter/2`. The latter is useful to set parameters from within programs or in your ".opium-m-rc".

Example for a "single" parameter. Parameter `attribute_display` is a "single" parameter. It sets the attributes of a trace event which are displayed by command `print_event`. It can be set and reset as follows. Initially, the default setting is used to print the current trace event.

```
   [Opium-M]: set_default(attribute_display), print_event.
18:     6 [5] exit partition([1, 2], 3, [1, 2], []) []
```

Then the current value of the `attribute_display` parameter is retrieved.

```
   [Opium-M]: get_parameter(attribute_display, L).
     L = [on, on, on, off, on, off, off, off, on, off, off, on, off, off, on]
```

Assume we want to see what the full trace event looks like. We use `set_parameter/2` to set all the attributes of a trace event to be displayed.

```
   [Opium-M]: set_parameter(attribute_display,
       [on, on, on, on, on, on, on, on, on, on, on, on, on, on, on]),
   print_event.
18:     6 [5] exit  det (predicate)  {qsort} qsort: partition(
    [1, 2] {list__list(int)}, 3 {int}, [1, 2] {list__list(int)},
    [] {list__list(int)})/4-0 []
```

Now, there is too much information in a full trace event to have it displayed at each step. To hide parts of it we use the interactive setting command (`set_parameter/1`) because we do not know which argument of the parameter corresponds to which attribute of the event.

```
   [Opium-M]: set_parameter(attribute_display).
Chrono:        [on, off] or abort ? on.
Call:          [on, off] or abort ? on.
Depth:         [on, off] or abort ? on.
Deter:         [on, off] or abort ? off.
Port:          [on, off] or abort ? on.
PredOrFunc:    [on, off] or abort ? off.
DeclModule:    [on, off] or abort ? off.
DefModule:     [on, off] or abort ? off.
Name:          [on, off] or abort ? on.
Arity:         [on, off] or abort ? off.
```

```
ModeNumber:    [on, off] or abort ? off.
ListArg:       [on, off] or abort ? on.
ListNonArgVar: [on, off] or abort ? off.
Type:          [on, off] or abort ? off.
GoalPath:      [on, off] or abort ? off.

[Opium-M]: print_event.
 18:     6 [5] exit partition([1, 2], 3, [1, 2], [])
```

For this particular attribute_display parameter, you can also switch on and off each event attribute display with `toggle/1`. `toggle/1` was straightforwardly implemented with `get_parameter/2` and `+set_parameter/2`.

```
[Opium-M]: toggle(args), print_event.
 18:     6 [5] exit partition()
```

Note that any hidden attribute can be retrieved by using `current(<attribute-name> = ...)`. In the following `current/1` is used to retrieve the current chronological event number.

```
[Opium-M]: current(chrono = X).
  X= 18
```

Example for a "multiple" parameter Parameter `arg_undisplay/2` is a "multiple" parameter. It specifies which arguments of which predicates should not be displayed by command `print_event`. Initially the default setting is used to print the current trace event, all arguments of all predicates are displayed.

```
[Opium-M]: set_default(arg_undisplay), print_event.
 18:     6 [5] exit partition([1, 2], 3, [1, 2], []) []
```

Assume we want to avoid seeing the first argument of predicate `partition/4`. We can use `set_parameter/2` as for "single" parameters. Then, when printing a trace event related to `partition/4` an "ersatz" is displayed instead of the first argument. By default the ersatz is "..." (see procedure `write_ersatz/0`)

```
[Opium-M]: set_parameter(arg_undisplay, [partition / 4, 1]), print_event.
 18:     6 [5] exit partition(..., 3, [1, 2], []) []

yes.
```

Note that you can recover the value of the undisplayed arguments using `current(args = ...)`.

```
[Opium-M]: current(args = [X, _]).
  X = [3]
```

If you want to set back displayable *one* of the arguments which are currently set undisplayable, use `unset_parameter/2`.

```
[Opium-M]: unset_parameter(arg_undisplay, [partition/4, 1]), print_event.
 18:     6 [5] exit partition([1, 2], 3, [1, 2], []) []
```

If you want to set back displayable *all* the arguments which are currently set undisplayable, use `set_default/1`.

```
[Opium-M]: set_default(arg_undisplay).
```

5.3 Adding simple commands

You can program new functionalities in Prolog and simply compile them in the proper module of the Opium-M session. For example, let us assume that we want to be able to graphically display the arguments of the `check_mastermind/4` predicate of the mastermind program given in Appendix B. Suppose that we have the Prolog predicate `display_mastermind/4` that calls a Tcl/TK script that graphically displays the Mastermind board of Figure 5 (the code of this predicate is an adaptation of the Mercury version of `display_mastermind/4` which is in appendix B). We can define ourselves a command `display_mastermind` that will check that we are on the rigth procedure at the rigth port before calling `display_mastermind/4`:

```
dm :- display_mastermind.  % abbreviation for easy use

display_mastermind :-
    % make sure this is the proper procedure
    current(proc = mastermind_checker:check_mastermind/4-0),
    % make sure we are on a exit port
    current(port = exit),
    % get current argument of check_mastermind/4
    current(args = [Guess, Code, Bulls, Cows]),
    % display it graphically
    display_mastermind(Code, Guess, Bulls, Cows).
```

If these definitions are compiled in the Opium-M session you can then use `display_mastermind` or `dm` on the fly in complement with the standard display of arguments.

The new functionalities which are simply compiled in the Opium-M session do not have entries in the interface. They are not listed by the `show_all` commands, they appear neither in the manual, nor in the window-based user interface menus. This has no importance if the new commands have a limited lifetime. However, if you think you will need them again or if somebody else is supposed to use them you should include them in a scenario. The scenario handler will then provide the interface automatically (see section 6).

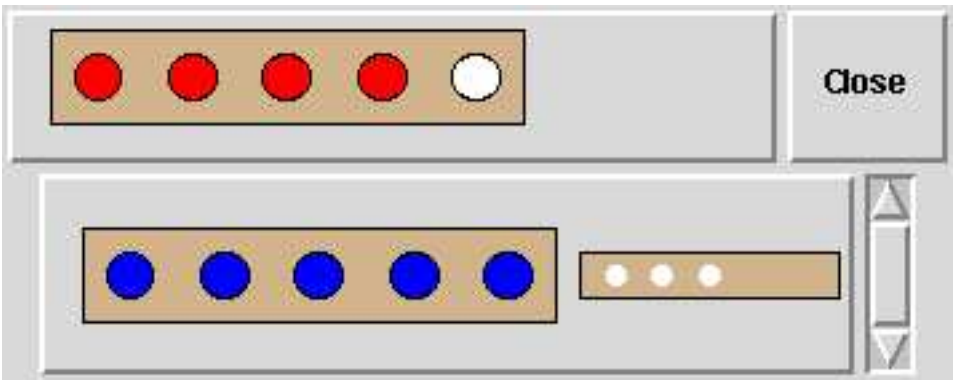


Figure 5: Board output of display_mastermind/4

5.4 Customizing existing objects

This section will explain only the minimum about the scenario handler to help you customize the existing objects. See section 6 for a precise description of what is generated automatically from the Opium-M declarations. In the following we first give some advices. The general mechanism which enables customizations to be done is described. Then we explain how to customize commands and other objects.

5.4.1 Advice

Before undertaking any major customization check whether modifying an existing parameter would solve your problem (`show_all(parameters)`, `show_all(parameters, ScenarioName)`).

If you would like to change a particular sub-behavior of a command or a primitive check whether by chance there exists a procedure which performs this sub-behavior. Customizing this procedure would be easier than customizing the whole command or primitive.

Test your changes on the fly before you change static settings. For example, if you want to change the display of arguments, first program a predicate to print them your way; test it on the fly using `current(args = ...)`; only then replace the default argument display by your predicate.

Try to reuse as much as possible the default implementations. Some of the commands take into account many aspects, which may take you some time to "rediscover". If you build on top of them you will benefit from the existing know-how. You will also benefit from updates of the default objects. We will show in the following how you can reuse default implementations.

If you want to add many features and make many changes it might be time to `make` a proper scenario (see Chapter 6).

If you want to have the default scenario and the modified one both available make the scenario you want to modify local in another module (if possible) and test your customizations there.

Some objects are used everywhere, so consider the side-effects. Unfortunately, there is currently no dedicated Opium-M tool which can easily tell you which objects use a given object.

5.4.2 General mechanism

A basic mechanism enables the default implementation of objects to be reused while modifying them. The name of objects and their implementations are two different items linked together automatically by Opium-M. In general an Opium-M object has (roughly) the following code.

```
object :-
        object_impl.
```

Hence customizing `object` can be done by providing a new implementation reusing (if possible) its default implementation, such as the following:

```
[Opium-M]: [user].
 my_object :-
        do_something,
        object_impl,                    % default implementation
        do_something_else.
```

Then ask Opium-M to replace the default implementation by the customized one using `rebuild_object/5`:

```
:- rebuild_object(ObjectType, object/0, my_object, Scenario, Module).
```

Command `rebuild_object/5` will roughly generate the following.

```
object :-
        my_object_impl.
```

In order to know what is the name of the default implementation of the object you want to customize use `implementation_link/4`. If you prefer the default implementation of an object you can set it back using `set_default/4`. Examples are given in the following. Note that at the moment there is no easy way to change the declaration of an object.

5.4.3 Customizing commands

Before customizing a command you need to know its type, the scenario it belongs to and the module in which this scenario is loaded. This information is available by typing `man(CommandName)` on line. There are some subtleties to customize commands of type "trace" and "tool". Some examples are given in the following.

Customizing "opium" commands Commands of type "Opium-M" are the most common commands of Opium-M and the simplest. The mechanism described above can be applied straightforwardly. For example, assume that we want to customize the `print_event/0` command. This is a good example where it is recommended to try to reuse the default implementation as much as possible. Indeed, it is a tricky predicate which takes into account many aspects such as Opium-M parameters, operators... We first get some information about `print_event/0`, in particular its type and its related scenario and module.

```
[Opium-M]: man(print_event).

print_event    {p}
Command which prints the current trace event according to the value of the
display parameters.
type of command : opium
scenario : display (global in Opium-M)
```

Now we know `print_event/0` is an "opium" command and it belongs to scenario `display` in module opium_kernel_M. We then retrieve the name of its default implementation.

```
[Opium-M]: implementation_link(command, print_event/0, Impl, 'Opium-M').
  Impl = print_event_Op
```

Assume we want to print the arguments of the traced goals only for "call" and "exit" events. We can program a `new_print_event` predicate which checks the current port. If is is not "call" or "exit" the "argument" attribute is switched off before printing and set back to its previous value after printing. For printing the default implementation is used.

```
[Opium-M]: [user].
new_print_event :-
    current(port = Port),
        (
             (Port == call ; Port == exit)
        ->
             print_event_Op
        ;
             toggle(args),
             print_event_Op,
             toggle(args)
        ).
```

Then we link the new implementation to the command name.

```
[opium]: rebuild_object(command, print_event/0, new_print_event, display,
         'Opium-M').
```

Now, when `print_event` is used what is actually executed is `new_print_event`. This is true whether `print_event` is run by a command at toplevel or used by a debugging program, for example `next` will print the event it retrieves using the customized `print_event`.

If after all we prefer the default implementation we can link the command back to its default implementation.

```
[Opium-M]: set_default(command, print_event/0, display, 'Opium-M').
```

Customizing "trace" commands For "trace" commands, which retrieve and print exactly one trace event, Opium basically generates two clauses

```
command :-
        command_np,
        print_event.

command_np :-
        command_impl.
```

where `command_np` is a primitive which does *not print* the event it has moved to. If you implement a variant for `command_impl`, command `rebuild_object/5` will actually link `command_np` to your implementation:

```
command_np :- my_command_impl.
```

Thus if you want to customize a "trace" command using `rebuild_object/5` remember that your implementation should *not* print the event it has moved to. You also have to pay attention that your implementation does not call `command_np` otherwise it will loop.

For example, assume you want the `next` command to print a newline before call events. You can customize the existing `next/0` by linking it to `new_next/0`. In the following we basically follow the same script as for the "opium" command.

```
[Opium-M]: man(next).

next    {n}
Command which moves forward to the next trace event according
to the "traced_ports" parameter.
type of command : trace
scenario : step_by_step-M (global in Opium-M)

next(N)    {n}
Command which prints the N next trace events according to the
"traced_ports" parameter.
N       : integer
type of command : opium
scenario : step_by_step-M (global in Opium-M)
```

```
[Opium-M]: implementation_link(command, next/0, Impl, 'Opium-M').
 Impl = next_Op

[Opium-M]: [user].
new_next_np :-
    next_Op,                    % move to the same place as default next
    ( current(port = call) ->  % if it is a call event
        opium_nl(trace)         % print a newline
    ;
        true
    ).

[Opium-M]: rebuild_object(command, next/0, new_next_np, 'step_by_step-M',
             'Opium-M').
```

Customizing "tool" commands Commands of type "tool" are like the tools of Eclipse, that is the implementation (tool body) has one more argument than the command to take the actual caller module into account. Hence, your implementation should have the arity of the command + 1.

For example, assume that you would like `show_parameters/1` to remind you that it shows only the instantiated parameters of the scenario. You can customize it in the following way.

```
[Opium-M]: man(show_parameters).

show_parameters(Scenario)
Command which shows the values of all the parameters related to a
scenario visible in the current module.
Scenario        : is_opium_scenario
type of command : tool
scenario : scenario_handler (global in Opium-M)

show_parameters
Command which shows the values of all the parameters of all scenarios.
type of command : opium
scenario : scenario_handler (global in Opium-M)

[Opium-M]: implementation_link(command, show_parameters/2, Impl, 'Opium-M').
  Impl = show_parameters_Op

[Opium-M]: [user].
my_show_parameters_body(S,M) :-
```

```
        printf("Instantiated parameters of scenario %w:", [S]),
        show_parameters_Op(S,M).

[Opium-M]: rebuild_object(command, show_parameters/1,
        my_show_parameters_body, 'opium_kernel-M', 'Opium-M').
```

5.5 Customizing primitives, procedures and types

Primitives, procedures and types can be customized in exactly the same way as "opium" commands.

For example, assume we want to change the way the depth attribute is displayed by `print_event/0`. We want to replace the square brackets by '*'. We can customize the `write_attribute/2` procedure as follows.

```
[Opium-M]: man(write_attribute).

write_attribute(AttributeName, AttributeValue)
Procedure which displays a attribute of the trace event. Where
AttributeName is one of {chrono, call, depth, port, module, arity}.
To customize the way arguments are displayed you should rather
modify write_arg_attribute/3.
scenario : display (global in Opium-M)

[Opium-M]: implementation_link(procedure, write_attribute/2, Impl,
               'Opium-M').
  Impl = write_attribute_Op

[Opium-M]: [user].
new_write_attribute(depth, D) :-        % if depth attribute
        !,
        write_trace(*),                 % print a '*'
        write_trace(D),                 % print the depth
        write_trace(*).                 % print a '*'
new_write_attribute(N, V) :-            % display other attributes
        write_attribute_Op(N, V).       % in the standard way

[Opium-M]: rebuild_object(procedure, write_attribute/2,new_write_attribute,
                        display, 'Opium-M').
```

6 Extending the debugging environment

In the previous section, we explained how the Opium-M debugging environment can be customized "on the fly" by setting parameters, and by modifying or adding debugging commands

and primitives. Such customizations can be reused by collecting them in a file and compiling them into the Opium-M session. Thus, you can easily set up your personal debugging environment. However, if you only compile a collection of predicates you, and especially other users who might want to benefit from your work, are not supported on using your modifications. For example, your new functionalities cannot be seen from the manual.

This chapter will guide you through the process of extending the environment so that other users can straightforwardly benefit from your extensions. Note that all the existing scenarios are build within the frame described here. You have to declare the objects that you want people to use. These objects will then have a nice interface.

6.1 When to make a scenario

Assume you have developed a new debugging strategy, and you have implemented and tested the commands needed to apply it. Now you would like to add them to the Opium-M environment, in order to get the same support when using your debugging strategy as you get for the strategies already implemented in Opium-M, and also to make it available for other users as well. This is the time when you have to *make* a new debugging scenario.

When you want to make a new scenario, you have to provide the scenario's source code which consists of the Prolog code implementing the debugging strategy, as well as *declarations* of those functionalities of your scenario that you want others to use, that is the commands, primitives, procedures, parameters, and types. The implementation and the declarations may be distributed among several source files.

The scenario is made by the *scenario handler*. This is the part of Opium-M which is responsible for the integration of a scenario into the debugging environment. Its task is twofold:

- First, the scenario handler *translates* the scenario's source files. It reads the declarations of Opium-M objects contained in the source files, and uses them to generate some code which is collected in the *object files* of the scenario. The idea is that as much as possible of the code related to the objects is generated automatically. This helps to avoid inconsistencies in the code, to reduce redundant information, and to complete the source code by checkings in order to avoid run-time errors. The declarations themselves are also collected in the object files.
- Secondly, the scenario handler *loads* the scenario. The source files as well as the objects files related to the scenario are compiled in the debugging session, in order to run the scenario. While the scenario is loaded, some predicate flags for the predicates defined in the scenario are set according to the *load options*. The parameters related to the scenario are initialized automatically. Furthermore, the declarations of the Opium-M objects are added to the database. They are used to give help on the objects, thus the help information is always up-to-date and syntactically consistent with the code.

In the following sections, we present everything you need to know when you want to make an Opium-M scenario. First, we give a detailed description of the declarations of Opium-M objects, and explain what the scenario handler does for each of them. Then we give some

advices which might help you when designing your first scenario. Finally, we explain how a scenario is made, and what you have to care about, and *not* to care about when making a scenario. As an example, the source code of scenario `step_by_step_M` is given in appendix C.

6.2 The declaration of Opium-M objects

The Opium-M objects which shall be treated by the scenario handler have to be announced to Opium-M by declarations. The declarations are added to the Prolog code implementing the scenario. This gives a better chance that the declarations are always up-to-date with respect to the current state of the implementation.

In the following, we present an example declaration for each kind of Opium-M object. We explain what kind of information the scenario handler gathers from the declarations, and what it is going to do with this information when the scenario is made. This will also tell you what the user has to provide, and what will be automatically done by Opium-M.

6.2.1 Common parts of the declarations

The declaration of an Opium-M object looks like

```
opium_<object>(
        slot1 : value1,
              :
        slotn : valuen
              ).
```

where each slot contains certain information about the object.

There are some slots which appear in all the Opium-M declarations, or at least in most of them. These slots are explained in this section. The slots which are special in a particular declaration are explained in the respective section.

name the name of the Opium-M object. It gives access to the object, and it is used to find help information about the object in the on-line manual, and in the reference manual;

arg_list the list of arguments of the Opium-M object. The names of the arguments are used in help messages and error messages, therefore they should be meaningful. The arguments have to be variables, to ensure that the types can be checked automatically;

arg_type_list the list of types of the object's arguments. There are four admissible types:

- `T`, where `T` is an Opium-M type,
- `B`, where `B` is a built-in predicate of Eclipse,
- `is_member(L)`, where `L` is a list of values,

- `is_subset(L)`, where `L` is a list of values.

implementation the predicate which is implementing the object. The idea of separating name and implementation is to support the user on customizing the object (cf. section 5.4). The scenario handler connects the object's name to its implementation, possibly adding some checkings;

message the help message giving a short description of the object. This message is displayed by the on-line help, and it is also listed in the reference manual.

6.2.2 The Opium-M scenario

The declaration of an Opium-M scenario has to be contained in the scenario's *basefile*, that is in a file called "`<scenario>.op`". The declaration of an Opium-M scenario looks like:

```
opium_scenario(
        name         : opium_kernel_M,
        files        : [ opium_kernel_M,
                         forward_move_M,
                         current_arg_M,
                         current_slots_M,
                         event_attributes_M,
                         exec_control_M,
                         coprocess_M],
        scenarios    : [],
        message      :
"Scenario opium_kernel_M contains all the basic mechanisms of Opium-M \
which are needed to debug Mercury programs. \n\
"
        ).
```

The slots in the declaration of a scenario have the following meaning:

name the name which can be used to get help information about the scenario, and which is also used as the title of the respective section in the reference manual;

files the (relative) names of the Opium-M files related to the scenario. All the files related to a scenario have to be in the same directory, called *source directory*. They have to have an extension "`.op`" indicating that they contain Opium-M code;

scenarios the names of the scenarios which are required in order to run the actual scenario;

message the help message giving a short description of the scenario, and of its intended use.

This declaration tells the scenario handler which source files are part of the scenario, that is which files have to be treated when the scenario is made. Furthermore, it says which scenarios have to be made together with the declared scenario if they are not yet present in the module where the scenario is loaded.

When the scenario is loaded, its name is added to the menu of scenarios in the window-based user interface.

6.2.3 The Opium-M command

The commands are the objects for which the scenario handler provides the most support. The declaration of a command looks like:

```
opium_command(
        name            : next,
        arg_list        : [N],
        arg_type_list   : [integer],
        abbrev          : n,
        interface       : menu,
        command_type    : opium,
        implementation  : next_Op,
        parameters      : [traced_ports],
        message         :
'Command which prints the N next trace lines according to the
 "traced_ports" parameter.'
        ).
```

The slots in the declaration of a command have the following meaning (for the slots not mentioned here, see section 6.2.1):

abbrev the command's abbreviation. If there is no abbreviation for the command, this slot has to contain a variable;

interface says how this command shall appear in the window-based user interface. A command may be placed in a `button` or in a `menu`, or it may be `hidden` (see below);

command_type there are three possible types of a command: `opium`, `trace`, and `tool` (see below);

parameters the list of parameters which can be set in order to modify the behavior of this command. The meaning of this slot is only to give help information.

For all the commands, the scenario handler connects the name of the command to its implementation. If the command has arguments, a call to predicate `check_arg_type/4` is

inserted before the implementation is actually called. This predicate checks the types of the arguments, and allows to correct them on the fly if they are wrong . The arguments of a command have to be variables to ensure that unification always succeeds, so that the call to `check_arg_type/4` is always reached in order to give proper error messages. For command `next/1` whose declaration is given above, the following predicate is generated by the scenario handler:

```
next(N) :-
    check_arg_type([N], ['N'], [integer], NewList),
    Cmd =.. [next_Op | NewList],
    Cmd.
```

where `NewList` is the list of checked (and possibly corrected) arguments, and `next_Op/1` is the name of the command's implementation.

If an abbreviation for the command is given, it is connected to the name of the command. Thus, the command's abbreviation will be updated automatically if the number of arguments is changed. For example, for command `next/1` the following connection is generated:

```
n(N) :-
    next(N).
```

Some actions of the scenario handler depend on the type of the command, where the type may be `trace`, `tool`, or `opium`.

- A command of type `trace` is intended to determine and show exactly one trace line. *The printing of the trace line is not part of the commands implementation.* Instead, a call to predicate `print_line/0` is added to this command when the scenario is made. The reason is that together with the command a primitive is automatically declared which does exactly the same as the command, except printing the trace line. This primitive is called `<command>_np` (no print), and its abbreviation is called `<abbrev>_np`. This primitive can be used by a debugging program to retrieve the same trace line as it is done by the command, without printing it (see the example for skip-till-condition, section 3.4). In case of command `next/0` whose implementation is called `next_Op/0`, the scenario handler generates the following code:

  ```
  next :-          % command
      next_np,
      print_event.

  next_np :-       % primitive
      next_Op.

  n_np :-          % abbreviation
      next_np.
  ```

  If the command has arguments, the checking of argument types is done in the command, not in the primitive.

- A command of type `tool` shall be declared as a tool of Eclipse, that is the name of the module where it is called is automatically passed to its implementation. Therefore, the number of arguments in the implementation has to be one more than the number of arguments given in the argument slot, namely the module has to be added as a last argument. The declaration of the command as a tool is done by the scenario handler. For example, command `set_default/1` has to set the value of the parameter known in the current module, therefore it is declared as a command of type `tool`. Its implementation is called `set_default_Op/2`. The scenario handler generates the following code:

```
:- tool(set_default/1, set_default_body/2).

set_default_body(Parameter, Module) :-
    check_arg_type([Parameter, Module], ['Parameter', 'Module'],
                   [is_opium_parameter, is_opium_module], NewList),
    BodyCmd =.. [set_default_Op | NewList],
    BodyCmd.
```

- If a command is of type `opium`, only its interface is generated by the scenario handler. The behavior of the command is completely controlled by the implementation, including the command's output. The listing of `next/1` gives an example for the code generated by the scenario handler for a command of this type (see above).

The integration of the command into a window-based user interface can be done according to the entry in the interface slot:

- A command is put on a `button` if its name or abbreviation is short enough. The maximum number of buttons in a scenario is six. If you have declared more than six commands of type `button`, the remaining ones will be put in the command menu.
- A command with interface `menu` is added to the command menu.
- A command with interface `hidden` does not appear in the window-based user interface. Thus, this type has to be used for commands which make sense in the tty interface only.

Currently, there is no Window User Interface for Opium-M.

6.2.4 The Opium-M primitive

A primitive is similar to a command, but it does not provide any user interface facilities, because it is supposed to be used in debugging programs. The declaration of a primitive looks like:

```
opium_primitive(
    name            : current_arg_types,
    arg_list        : [ListArgTypes],
    arg_type_list   : [is_list_or_var],
    abbrev          : curr_at,
    implementation  : current_arg_types_Op,
```

```
        message         :
"Gets or checks the list of the arguments types of the current procedure. \
Unify non-live arguments with the atom '-'"
        ).
```

The slots in the declaration of a primitive have the following meaning (for the slots not mentioned here, see section 6.2.1):

abbrev the primitive's abbreviation. If there is no abbreviation for the primitive, this slot has to contain a variable.

The scenario handler connects the name of the primitive to its implementation. If an abbreviation for the primitive is given, it is connected to the name of the primitive. For example, the following code is generated for primitive `current_arg_types/1`:

```
current_arg_types(Goal) :-          % primitive
    current_arg_types_Op(Goal).

curr_at(Goal) :-                    % abbreviation
    current_arg_types(Goal).
```

When the scenario is loaded, the primitive is added to the menu of primitives in the window-based user interface.

6.2.5 The Opium-M procedure

A procedure is a basic predicate which is used to implement commands and primitives. Its declaration looks like:

```
opium_procedure(
        name            : write_arg,
        arg_list        : [Arg],
        implementation  : write_arg_Op,
        parameters      : [term_display, list_display],
        message         :
'Procedure which prints an argument in the trace line.'
    ).
```

The slots in the declaration of a procedure have the following meaning (for the slots not mentioned here, see section 6.2.1):

parameters the list of parameters which can be set in order to modify the behavior of the procedure. The meaning of this slot is only to give help information.

The scenario handler connects the name of the procedure to its implementation, thus the following predicate is generated for procedure `write_arg/1`:

```
write_arg(Arg) :-
    write_arg_Op(Arg).
```

When the scenario is loaded, the procedure is added to the menu of procedures in the window-based user interface.

6.2.6 The Opium-M parameter

An Opium-M parameter can be used to modify the behavior of a command, a primitive, or a procedure. The declaration of a parameter looks like:

```
opium_parameter(
        name             : arg_undisplay,
        arg_list         : [Pred, ArgNo],
        arg_type_list    : [is_pred, integer],
        parameter_type   : multiple,
        default          : nodefault,
        commands         : [write_arg],
        message          :
'Parameter which tells which arguments of which predicates have
to be NOT displayed. There must be one "arg_undisplay" clause for
each argument which shall not be displayed.'
        ).
```

The slots in the declaration of a parameter have the following meaning (for the slots not mentioned here, see section 6.2.1):

parameter_type there are three possible types of a parameter, with the following meaning:

`single`	the parameter always has exactly one value;
`multiple`	the parameter may have several values, or it may not be set at all;
`c`	the parameter is a system parameter which is actually implemented in C.

default the list of default values for the parameter's arguments. If the parameter's type is `multiple`, the entry may also be `nodefault`, saying that this parameter does not have any value when the scenario is loaded;

commands the list of commands whose execution is influenced by the value of this parameter. The meaning of this slot is only to give help information.

The scenario handler declares the parameter as dynamic predicate, as the values of the parameter will be stored in clauses asserted in Opium-M. For example, parameter `arg_undisplay/2` is declared in the following way:

```
:- dynamic arg_undisplay/2.
```

If a default value is given, a first clause of the dynamic predicate with this default value will be generated and asserted when the scenario is loaded, in order to initialize the parameter.

When the scenario is loaded, the parameter is added to the menu of parameters in the window-based user interface.

6.2.7 The Opium-M type

A type is a predicate which is used to check the arguments of commands and parameters, and to give help information on the arguments. Its declaration looks like:

```
opium_type(
        name            : is_proc,
        implementation  : is_proc_Op,
        message         :
"Type which succeeds for terms of the form
[ProcType+][Module:]ProcName[/Arity][-ModeNum] where terms betwenn square
bracquets are optional, ProcType has type is_proc_type_attribute/1,
Module and ProcName have type is_atom_attribute/1, Arity and ModeNum have
type is_integer_attribute/1.
").
```

For the meaning of the slots, see section 6.2.1.

The scenario handler connects the name of the type to its implementation. Thus, the following code is generated for type `is_pred/1`:

```
is_pred(X) :-
    is_pred_Op(X).
```

When the scenario is loaded, the type is added to the menu of types in the window-based user interface.

6.2.8 Declaring Opium objects with Emacs

The Opium environment contains a file "`opium-mode.el`" with extensions for the Emacs editor. These extensions are interactive functions which support the user on declaring Opium objects. The names of the functions are

- `scenario`
- `command`

- `primitive`
- `procedure`
- `parameter`
- `type`
- `demo`

according to the names of the Opium objects. If a function is called, it will prompt for the entries in the declaration, giving some help on the possible values. When the declaration is complete, it is added at the current cursor position.

6.3 Some advices on designing a new scenario

Assume you have implemented all the predicates needed to run your debugging strategy. Now you want to make the scenario, in order to get support on using it. First, you have to add the declaration of the scenario to the scenario's basefile. This declaration contains the names of all the files related to the scenario, as well as the names of other scenarios which are needed to run the current one.

Then you have to decide upon the commands, primitives, and procedures:

- A predicate which shall be used at toplevel only is a *command*.
- A predicate which shall be used at toplevel, but can also be used in a debugging program (possibly in another scenario), is a *primitive*.
- A predicate used by commands and primitives which you or other users might want to customize, or to use in another scenario, is a *procedure*.

When you enter the argument types of commands and primitives, you may notice that the set of Opium types is not sufficient. If this is the case, you have to declare (and to define!) a new *type* which is suitable for the arguments used by your commands and primitives.

If parts of your debugging program are variable with respect to certain items, these items should be declared as *parameters*. This provides flexibility when the scenario is used.

If you want to encourage other users to try your scenario, you can also add automated *demos* which illustrate the features of your debugging strategy , and give some ideas of how the strategy is supposed to be used.

When deciding upon the objects, remember that the objects are those predicates related to the scenario which can be easily used by another scenario, thus they implement an interface between the scenarios. This is also reflected by the fact that the module declarations for Opium objects are automatically added when the scenario is loaded (cf. section 6.4). Therefore, you should not add any explicit module declaration for a predicate defined in your scenario. If you think this is needed, it means that this predicate should be declared as an Opium object.

For all the objects, add a help message which describes the functionalities of the object, and explains how it is related to other objects. If you need some further help on how to fill the slots of the Opium declarations, you can also have a look at the implementations of the current Opium scenarios.

6.4 How to make a scenario

The scenario handler which is used to make a new scenario is started by command `make`. This command is defined in scenario `scenario_handler`. Actually, there are four commands called `make`, with a different number of arguments. We explain the use of the most general one, command `make/5`. The semantics of the other commands can be seen from the manual. All of them are calling command `make/5`. In order to test a scenario, use `make/1` in a test module.

In a goal `make(Scenario, Module, OptionList, SrcDir, ObjDir)`, argument `Scenario` is the name of the scenario, `Module` is the name of the module where the scenario shall be loaded, `OptionList` is the list of options telling how the scenario shall be loaded (see below), `SrcDir` is the full path name of the directory containing all the source files related to the scenario, and `ObjDir` is the full path name of the directory where the *object files*, that is the files resulting from the translation of the scenario shall be stored. Note: it is important that the full path name is given in exactly the same way as this is done by Eclipse's built-in `getcwd/1`, as this is used to determine the path name of the current directory which might be compared to the path names given in the `make` command. The object directory is created automatically if it does not exist yet.

When command `make/5` is called, it reads the scenario's base file "`<scenario>.op`" in order to determine the names of the files related to the scenario from the scenario declaration. Then for each source file it checks whether it has been modified since it has been translated for the last time. If this is the case, and only then, the source file is translated again.

When the scenario handler translates a scenario file, it checks the file for Opium declarations, and collects some information from the declarations (see section 6.2). This information is stored in the object files which are compiled when the scenario is loaded. For each source file, there are actually two object files: a *load file* which is compiled when the scenario is loaded active (extension "`.load`"), and an *autoload file* which is compiled when the scenario is loaded inactive (extension "`.autoload`").

When the object files for all the source files related to the scenario are up-to-date, the scenario is loaded, according to the load options given in argument `OptionList`. There are three pairs of options: `active/inactive`, `traceable/untraceable`, `local/global`. The options have to be given in this order. They have the following meaning:

- if a scenario is made `active`, its source files and load files are compiled, and it can be immediately used; if a scenario is made `inactive`, only the autoload files containing the autoload information are compiled, that is the implementation will actually be loaded only when a command related to this scenario is called;
- if a scenario is made `traceable` its implementation can be traced by another Opium session; if a scenario is made `untraceable`, the details of its implementation will be skipped, thus it takes less space, and the execution is faster;
- if a scenario is made `local` it is only available in the module where it is defined; if a scenario is made `global` its objects can be used in every module, except in the modules where a local scenario with the same name exists.

The latter two kinds of load options are needed even when the scenario is loaded inactive, because they are reused when the implementation is autoloaded.

6.5 Remaking an existing scenario

The scenario handler can also be used to remake an *existing* scenario. For example, you might want to change the load options of a scenario in order to make a local scenario global, or to make a scenario untraceable when it has been tested. Furthermore, you can make a local version of an existing scenario in a new module before modifying it, thus the modifications do not affect the original scenario, and the other scenarios which are using it. The following basic scenarios have to be global as they are used by all the extensions, and they cannot be remade in a module other than `opium_kernel`:

- `scenario_handler`
- `opium_kernel_M`
- `interface`
- `source_M`
- `display_M`

You also might want to remake a scenario because you want some extensions to be integrated into this scenario, or you want to modify the scenario in a deeper way than the modifications presented in section 5. This case is currently not really supported by Opium. There is the following problem. If you want to modify or extend part of the original source code of a scenario, there are two possibilities. Either you make a private copy, but then you do not benefit from updates of Opium, or you modify the installed copy, but then all Opium users have to use your modifications. Therefore, we advise you to create an additional scenario instead, and to load it in the same module as the scenario you want to extend. The only difference is that in the on-line help, and in the window-based user interface you have to check both scenarios instead of one only.

6.6 Error messages given by the scenario handler

The scenario handler gives an error message in the following cases:

- if a scenario shall be made global, but it is already defined global in another module;
- if the scenario is not declared in the basefile;
- if a scenario file given in the scenario's declaration does not exist;
- if the declaration of an Opium object is not correct or not complete;
- if the default value of an Opium parameter given in the declaration is not correct.

The error message which is given if an Opium declaration is not correct or not complete currently does not give any hint on the kind of error. If the scenario handler complains about a declaration, the following might be wrong:

- a slot is missing,
- a slot entry does not have the proper type (list, list of atoms, etc.),
- an slot entry does not have an admissible value (`command_type`, `interface, default`),
- there is an argument of a command which is not a variable,

- the number of arguments and argument types is inconsistent.

Furthermore, you might get error messages from Eclipse when you put explicit module directives into the source code of your scenario, as these may interfere with the work of the scenario handler.

6.7 Initialization of scenarios

If there are parameters or any other dynamic predicates in a scenario whose values are changed when the scenario is used, you might want to reinitialize these predicates when the execution of a new goal is started in the traced session. For example, parameter `zoom_depth/1` in the Opium-M `zooming`[5] scenario has to be reset to 1 in order to start the examination of the new goal from depth 1.

This can be achieved by adding a call to the command `initialization/1` to the source code of the scenario. Its argument is a goal which will be executed every time a new execution is started in the traced session. For example, the `zooming` scenario contains a goal

```
:- initialization(set_default(zoom_depth)).
```

which will reset the parameter used by the zooming commands.

[5] Not yet available

7 Bibliographical References

References

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

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

[3] M. Ducassé. Coca: An automated debugger for C. In IEEE Press, editor, *Proceedings of the 21st Int. Conf. on Software Engineering*, May 1999.

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

[5] F. Henderson, T. Conway, Z. Somogyi, D. Jeffery, and P. Ross. *The Mercury Language Reference Manual.* University of Melbourne, January 1999. Available at "http://www.cs.mu.oz.au/research/mercury/".

[6] F. Henderson, T. Conway, Z. Somogyi, D. Jeffery, P. Schachte, S. Taylor, and C. Speirs. *The Mercury User's Guide.* University of Melbourne, January 1999. Available at "http://www.cs.mu.oz.au/research/mercury/".

[7] F. Henderson, Z. Somogyi, and T. Conway. Determinisism analysis in the Mercury compiler. In *Proceedings of the Australian Computeur Science Conference*, pages 337–346, Department of Computeur Science, University of Melbourne Parkville, 3052 Victoria, Australia, January 1996.

[8] 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.

Part II
Opium-M – Reference Manual

The following sections make the reference manual. It has been generated (mostly) automatically by Opium-M. The advantages are manifold. It is a *complete* picture of the state of Opium-M at the time it has been generated. The syntax is *accurate* as arguments and types are deduced from the actual implementation of objects. Last but not least, *you* can generate an up-to-date version at any time using command `manual/1`. Yet, it, too often, lacks examples, and currently the items are not listed in alphabetical order. The index at the end of the manual should help palliate the latter problem.

In the reference manual messages are listed in the same way as in the on-line manual. For example, the message for the `manual` command is:

> **manual**(File)
> Command which shows all the scenarios, their commands and the corresponding explanations in the file "File" (in LaTeX format). It also does some fixes in the LaTeX file. The LaTeX file will then be called "File.tex" afterwards. In order to get a printable "File.dvi", use command latex_manual/1.
>
> *File : atom*
> *type of command : opium*
> *scenario : help (global in Opium-M)*

Beside the actual help message this says that command `manual` is called with a single parameter, `File`, which has to be of type `atom`. The type of the command is "opium". Commands can also be of type "trace" which means that they print a trace line.

8 Scenario "opium_kernel_M"

Scenario opium_kernel_M contains all the basic mechanisms of Opium-M which are needed to debug Mercury programs.

Commands

fget(AttributeConstraints) *{fg}*
Moves forwards through the execution until the first event that satisfy the list of constraints specified in AttributeConstraints (*). AttributeConstraints can be either a conjunction of attribute constraints, separated by "and" (fget(AC1 and AC2 and ...)) or a list of constraints (fget([AC1, AC2, ...])). The different attributes for fget are :

1. chrono: chronological event number of the event,
2. call: call event number of the event,
3. depth: depth in the proof three (number of ancestors - 1) of the event,
4. port: type of the event,
5. proc_type: tells if the current procedure is a predicate or a function,
6. decl_module: module name where the procedure is declared,
7. def_module: module name where the procedure is defined,
8. name: procedure name,
9. arity: procedure arity,
10. mode_number: procedure mode number,
11. proc: procedure ([proc_type+][decl_module:](name[/arity][-mode_number]) where only the attribute name is mandatory],
12. det: procedure determinism,
13. goal_path: goal path of the call of the procedure.

(*) An attribute constraint is a term of the form "AttributeAlias = Term" where AttributeAlias is an alias of a Mercury event attribute and Term can be:

- an exact value (attribute = ground_term),
- a negated value (attribute = not(ground_term)),
- a list of values (attribute = [ground_term1, ground_term2, ...]),

and for integer attributes (chrono, call, depth, arity),

- an interval (attribute = bottom..up).

Each attribute has a list of possible aliases that you can list with the command list_alias_attributes/0.

Example: the Opium-M goal fget(chrono = [20, 789] and depth = 3..6 and proc = foo/2) will make the execution move forwards until the first event which chronological event number is 20 or 789, depth is 3, 4, 5 or 6, procedure name is foo and arity is not 2. You can also use a list as an argument of fget: fget([chrono=[20, 789], depth = 3..6, proc = foo/2]) will have the same effect as the previous goal.
AttributeConstraints : is_list_or_conj_of_attribute_constraints_fget
type of command : trace

det_fget(List)
It is the deterministic version of fget/1.
List : is_list_or_conj_of_attribute_constraints_fget
type of command : trace

retry
Restarts execution at the call port of the current goal. The command will fail unless the values of all the input arguments are available at the current port. (The compiler will keep the values of the input arguments of traced procedures as long as possible, but it cannot keep them beyond the point where they are destructively updated.)

The debugger can perform a retry only from an exit or fail port; only at these ports does the debugger have enough information to figure out how to reset the stacks. If the debugger is not at such a port when a retry command is given, the debugger will continue forward execution until it reaches an exit or fail port of the call to be retried before it performs the retry. This may require a noticeable amount of time.
type of command : trace

current(AttributesConjunctOrList) *{curr}*
Gets or checks the values of the event attributes specified in AttributesConjunctOrList. AttributesConjunctOrList is a conjunction or a list of terms of the form attribute = Value. If Value is a free variable, it is unified with the current value of the attribute. If Value is a ground term, the current value of the attribute is retrieved and checked against Value. The different attributes for current/1 are :

1. chrono: chronological event number of the event,
2. call: call event number of the event,
3. depth: depth in the proof three (number of ancestors - 1) of the event,

4. port: type of the event,
5. proc_type: tells if the current procedure is a predicate or a function,
6. decl_module: module name where the procedure is declared,
7. def_module: module name where the procedure is defined,
8. name: procedure name,
9. arity: procedure arity,
10. mode_number: procedure mode number,
11. proc: procedure ([proc_type+][decl_module:](name[/arity][-mode_number]) where only the attribute name is mandatory),
12. det: procedure determinism,
13. goal_path: goal path of the call of the procedure,
14. args: list of procedure arguments (*),
15. arg_names: list of procedure argument names,
16. arg_types: list of procedure argument types, v
17. ars: list of the currently live variables,
18. var_names_and_types: list of the currently live variable names and types,
19. local_vars: list of the currently non-argument local live variables.

For example, current(chrono = Chrono and name = Name) (or current([chrono = Chrono, name = Name])) will unify Chrono with the chronological event number and Name with the procedure name of the current event. current(depth = 3) will succeed iff the depth of the current event is 3. current(args = [Arg1, -, -]) will unify Arg1 with the first argument of the current procedure if is live.

(*) non lived arguments are unified with '-' and if you do not want to retrieve all the arguments (because one of them is very big for example), you can use the atom '-': for example, current(arg = [X, -, -]) will only retrieve the first argument. Note that current(arg = [X, _, _] will have the same behaviour, but arguments will be retrieved through the socket.
AttributesConjunctOrList : is_list_or_conj_of_attributes_current
type of command : opium

current_live_var(VarId, VarValue, VarType) *{clv}*
Gets or checks the name, the value and the type of the currently live variables. VarId can be a string representing the variable name or, if it is an argument of the current procedure, an integer representing the rank the argument. Example: current_live_var("HeadVar__3", VarValue, _Type) (or equivalently current_live_var(3, VarValue, _Type)) binds VarValue with the current value of the third argument of the current predicate if it exists and if it is live, fails otherwise. You can get all the live variables by querying current_live_var(VarId, VarValue, VarType) and typing ";" at the prompt to search for other solutions. You can also get the list of all the currently live variables of type int with the Opium-M query setof((Name, Value), current_live_var(Name, Value, int), List).
VarId : is_string_or_integer_or_var
VarValue : is_term
VarType : is_atom_or_var
type of command : opium

stack
Displays the ancestors stack.
type of command : opium

list_attribute_aliases *{laa}*
List the available aliases for the different Mercury event attributes (fget/1 and current/1).
type of command : opium

re_init_opium
Re-initializes the Opium-M debugging session. This command migth be useful if Opium-M is broken.
type of command : opium

run(ProgramCall)
Executes a Mercury program from Opium-M.

Example: run("./cat(filename)") will run under the control of Opium-M the mercury program "cat" that takes "filename" as argument. run(hello) will run the Mercury program hello.
ProgramCall : is_atom_or_string
type of command : opium

abort_trace *{a}*
Aborts the current execution in the traced session.
type of command : opium

no_trace *{o}*
Continues execution until it reaches the end of the current execution without printing any

further trace information.
type of command : opium

rerun *{r}*
Runs again the last executed program.
type of command : opium

goto(Chrono)
Moves forwards the trace pointer to the event with chronological event number Chrono. If the current event number is larger than Chrono, it fails.
Chrono : integer
type of command : trace

Primitives

fget_np(AttributeConstraints) *{fg_np}*
Primitive which does the same as command fget except printing a trace line.
AttributeConstraints : is_list_or_conj_of_attribute_constraints_fget

det_fget_np(List)
Primitive which does the same as command det_fget except printing a trace line.
List : is_list_or_conj_of_attribute_constraints_fget

retry_np
Primitive which does the same as command retry except printing a trace line.

current_arg(ArgumentList)
Gets or checks the values of the currently live arguments of the current event. It will unify non-live arguments with the atom '-'. Example: if the first argument of the current procedure is 2, the second is [4, 6] and the third is not live, current_arg(Arg) will unify Arg with the list [2, [4, 6], -].

If you do not want to retrieve an argument (because it is very big for example), you can use the atom '-': for example, current_arg([X, -, -]) will only retrieve the first argument.
ArgumentList : is_list_or_var

current_arg_names(ListArgNames)
Gets or checks the list of the names of the current procedure arguments. Unify non-live arguments with the atom '-'.
ListArgNames : is_list_or_var

current_arg_types(ListArgTypes)
Gets or checks the list of the arguments types of the current procedure. Unify non-live arguments with the atom '-'
ListArgTypes : is_list_or_var

current_vars(LiveArgList, OtherLiveVarList)
Gets or checks the values of the currently live (*) variables of the current event. These variables are separated in two lists: one containing the live arguments of the current predicate, one containing other currently live variables.

(*) We say that 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 (which is not the case for for destructively updated variables).
LiveArgList : is_list_or_var
OtherLiveVarList : is_list_or_var

current_live_var_names_and_types(ListVarNames)
Gets or checks the list of names and types of the currently live variables. Each live variable is represented by the term live_var_names_and_types(VariableName, TypeOfTheVariable).
ListVarNames : is_list_or_var

current_live_var_names_and_types
current_live_var_names_and_types/0 gets and displays the live variable names and types. You can change this display by customizing the procedure display_list_var_names.

nondet_stack
Prints the contents of the fixed attributes of the frames on the nondet stack. This command is intended to be of use only to developers of the Mercury implementation.

stack_regs
Prints the contents of the virtual machine registers that point to the det and nondet stacks. This command is intended to be of use only to developers of the Mercury implementation.

init_opium_session
Initializes Opium-M.

goto_np(Chrono)
Primitive which does the same as command goto except printing a trace line.
Chrono : integer

end_connection *{ec}*
Ends the connection with the traced program.

Procedures

opium_write_debug(X)
This procedure is used to print information to debug Opium.

opium_write_debug(Stream, X)
This procedure is used to print information to debug Opium.

opium_printf_debug(Format, X)
This procedure is used to print information to debug Opium.

opium_printf_debug(Stream, Format, X)
This procedure is used to print information to debug Opium.

Parameters

socket_domain(Domain)
Parameter which tells which domain is used by the socket communication betwenn the two processes.
Domain : is_member([unix, inet])
default value : socket_domain(unix)
type of parameter : single

debug_opium(OnOff)
Prints additional information in the trace to debug Opium.
OnOff : is_member([on, off])
default value : debug_opium(off)
type of parameter : single

Types

is_list_or_conj_of_attribute_constraints_fget
Type which succeeds for list or conjunctions of terms of the form: "AttributeAlias = Term", where AttributeAlias is an alias of a Mercury event attribute and Term is a variable, an exact value, a negated value, a list of values, or an interval (Bottom..Up). Example: fget(chrono=[20, 789] and depth=3..6 and name=foo and arity=not(2)), which can also be typed fget([chrono=[20, 789], depth=3..6, name=foo, arity=not(2)])

is_list_or_conj_of_attributes_current
Type which succeeds for list or conjunctions of terms of the form: "AttributeAlias = Term", where AttributeAlias is an alias of a Mercury event attribute and Term is a variable or a possible value for the corresponding attribute. Example: current(name = Name and decl_module = module1), current([port = call, name = Name]).

is_port
Type which succeeds for a Mercury Port. Mercury ports are call (or 'CALL'), exit (or 'EXIT'), fail (or 'FAIL'), redo (or 'REDO'), then (or 'THEN'), else (or 'ELSE'), disj (or 'DISJ'), switch (or 'SWITCH' or 'SWTC'), first (or 'FIRST' or 'FRST'), later (or 'LATER' or 'LATR'), exception (or 'EXCP' or 'EXCEPTION').

is_port_or_var
Type which succeeds for a Mercury port or a variable (See is_port/1).

is_list_of_ports
Type which succeeds for a sublist of ['CALL', 'EXIT', 'REDO', 'FAIL', 'THEN', 'ELSE', 'DISJ', 'SWITCH', 'SWTC', 'FIRST', 'FRST', 'LATER', 'LATR', 'EXCP', 'EXCEPTION', call, exit, fail, redo, then, else, disj, switch, first, later, exception].

is_port_attribute
Type which succeeds for a port, a negated port (not('CALL')), a list of ports, '-' or a variable.

is_goal_path
Type which succeeds for list of atoms of the form 'e', 't', '?', 'ci', 'si', 'di' where i is an integer > 0.

is_goal_path_or_var
Type which succeeds for a Mercury goal path or a variable (See is_goal_path/1).

is_goal_path_attribute
Type which succeeds for a goal path, a negated goal path, a list of goal path, '-' or a variable.

is_atom_attribute
Type which succeeds for an atom, a negated atoms, a list of atom, a variable or '-'. It is intended to check proc_name def_module and decl_module attributes.

is_proc_type
Type which succeeds for the atoms predicate and function.

is_proc_type_attribute
Type which succeeds for pred or func, not(pred) or not(func), a list of atoms pred or func, '-' or a variable.

is_det_marker
Type which succeeds for a Mercury determinism marker. Mercury determinism are det (or DET'), semidet (or 'SEMI'), nondet (or 'NON'), multidet (or MUL'), cc_nondet (or 'CCNON'), cc_multidet (or 'CCMUL'), failure (or 'FAIL') and erroneous (or 'ERR').

is_det_marker_or_var
Type which succeeds for a Mercury determinism markers or a variable.

is_list_of_dets
Type which succeeds for a sublist of [det, semidet, nondet, multidet, cc_nondet, cc_multidet, failure, erroneous, 'DET', 'SEMI', 'NON', 'MUL', 'ERR', 'FAIL', 'CCNON', 'CCMUL'] (the determinism marker in capital letters are the one use in mdb, the internal Mercury debugger).

is_det_marker_attribute
Type which succeeds for a Mercury determinism marker, a negated determinism (not(nondet)), a list of determinism markers, '-' or a variable.

is_proc
Type which succeeds for terms of the form [ProcType+][Module:]ProcName[/Arity][-ModeNum] where terms betwenn square bracquets are optional, ProcType has type is_proc_type_attribute/1, Module and ProcName have type is_atom_attribute/1, Arity and ModeNum have type is_integer_attribute/1.

is_proc_or_var
Type which succeeds for a Mercury procedure or a variable.

is_arg_attribute
For the time being, you can't perform filtering on arguments i.e. you can only have variables or '-' for that attribute.

is_integer_attribute
Type which succeeds for an integer, a negated integer (not 6), a list of integers ([3, 5, 9]), an interval ('3..11'), a variable or '-'.

is_string_attribute
Type which succeeds for a string, a negated string (not "foo"), a list of strings, a variable or '-'.

9 Scenario "source_M"

Scenario source provides commands to retrieve and display Mercury source (and intermediate) code.

Commands

listing(Module, ProcOrType, Listing) *{ls}*
Retrieves the source code of a Mercury procedure or a Mercury type ProcOrType defined in the module Module and unifies it in Listing. Module can be either a library or a user defined module. Note: to be able to retrieve library procedures or types with listing/3, you need to have the source of the Mercury library somewhere and you need to make sure that the environment variable LIB_MERCURY has been set correctly to the path of the Mercury library in the sh script Opium-M.
Module : is_atom_or_string
ProcOrType : is_mercury_proc_or_type
Listing : is_list_or_var
type of command : opium

listing(Module, PredOrFuncOrType) *{ls}*
listing/2 is the same command as listing/3 except it prints the listing of the code on the standard output.

Example:

1. listing(foo, bar/3, List) unify List with the list of terms defining the Mercury predicate bar/3.
2. listing("~/dir/foo", bar) will display all the predicates bar/n defined in the module foo which is in "~/dir/" directory.
3. listing(io, io__read/3) will display the source of the Mercury library predicate io__read/3 defined in the Mercury module io.

Module : is_atom_or_string
PredOrFuncOrType : is_mercury_proc_or_type
type of command : opium

listing_hlds(Module, PredOrFuncOrType, Listing)
See listing_hlds/2.

Module : is_atom_or_string
PredOrFuncOrType : is_mercury_proc_or_type
Listing : is_list_or_var
type of command : opium

listing_hlds(Module, PredOrFuncOrType)
listing_hlds/2 and listing_hlds/3 are the same commands as listing/2 and listing/3 except they will list the HLDS procedures instead of the source code. To be able to list such HLDS code, you need to compile your module with "-dfinal" option.
Module : is_atom_or_string
PredOrFuncOrType : is_mercury_proc_or_type
type of command : opium

listing_current_procedure *{lcp}*
listing_current_procedure/0 prints the source code of the current procedure on the user window. If the current procedure is defined in a file that is not in the current directory, you need to specify the path of this file with listing_current_procedure/1.
type of command : opium

listing_current_procedure(Path) *{lcp}*
listing_current_procedure/1 is the same as listing_current_procedure/0 except you specify the path of the module of the current procedure.
Path : atom
type of command : opium

Types

is_mercury_proc_or_type
Type which succeeds if its argument is of the form Name or Name/Arity, where Name is an atom and Arity an integer.

is_atom_or_string
Type which succeed for an atom or a string.

10 Scenario "display_M"

Scenario which contains everything related to the display of trace events. In particular the attributes to be displayed can be specified, as well as the way lists and terms are displayed.

Arguments of predicates can be skipped. Many procedures allow you to customize the display.

Commands

print_event *{p}*
Prints the current trace event according to the value of the display parameters. The name of the printed attributes can be get with the command print_displayed_attributes/0.
type of command : opium

print_displayed_attributes
Prints the names of the attributes displayed by print_event/0.
type of command : opium

print_full_event *{pf}*
Prints the current trace event with all the attributes on.
type of command : opium

print_full_displayed_attributes
Prints the names of the attributes printed by print_full_event/0.
type of command : opium

indent(OnOff)
Sets relative indentation on/off. If a tracing process is on, it sets the depth at which the indentation has to start to the current depth. Otherwise the starting depth is 1.
OnOff : is_member([on, off])
type of command : opium

absolute_indent(Depth)
Sets the indentation on and sets the depth at which the indentation has to start to Depth.
Depth : integer
type of command : opium

toggle(AttributeName)
Toggles attribute display of print_event command. For example, if attribute decl_module is off, you can type "toggle(decl_module)" to switch it on. You can list all the attributes you can toggle thanks to list_attribute_aliases/0 command.
AttributeName : atom
type of command : opium

Procedures

write_indent(IndentFlag, IndentValue, IndentDepth, CurrDepth)
Procedure which displays an indentation – if indentation is on – according to the current depth and the indentation starting depth. If IndentFlag is on, it prints N times IndentValue, where N is CurrDepth - IndentDepth if this is positive, 1 otherwise.

write_attribute(AttributeName, AttributeValue)
Procedure which displays an attribute of the trace line. AttributeName is a member of the following list: [chrono, call, depth, port, proc_type, decl_module, def_module, arity, mode_number, args, deter, goal_path, non_arg_var]. To customize the way arguments are displayed, you should rather modify write_arg.

write_arg_attribute(Procedure, ListArg, ArgFlag, TypeFlag)
Procedure which displays the arguments of the trace event when the current procedure is Module:Name/Arity-ModeNum. If only the nth argument of a procedure needs a special treatment, you should customize write_arg/1.

write_nth_arg(Arg, N, Procedure)
Procedure which displays the Nth argument of procedure Procedure in DeclModule.

write_arg(Arg)
Procedure which prints an argument.

write_term(Term)
Procedure which displays a structured term, taking into account the term_display parameter.

write_list(List)
Procedure which displays a list, taking into account the list_display parameter.

write_ersatz
Procedure which writes "..." as a replacement for the hidden parts of the arguments.

write_comma
Procedure which writes ", ".

write_trace(X)
Prints its argument on the trace window.

read_input(Input)
Procedure which reads an input from within the current input stream of Opium-M.

display_stack(Stack)
Procedure that displays the ancestors stack.

display_list_var_names(ListVarNames)
Display the names of the currently live variables given by current_live_var_names_and_types/1.

Parameters

attribute_display(Chrono, Call, Port, Depth, Deter, PredOrFunc, DeclModule, DefModule, Name, Arity, ModeNumber, ListArg, ListNonArgVar, Type, GoalPath)
Parameter which contains the flags for the selective display of attributes. If the value of one argument is "on" then the corresponding attribute is displayed.
Chrono : is_member([on, off])
Call : is_member([on, off])
Port : is_member([on, off])
Depth : is_member([on, off])
Deter : is_member([on, off])
PredOrFunc : is_member([on, off])
DeclModule : is_member([on, off])
DefModule : is_member([on, off])
Name : is_member([on, off])
Arity : is_member([on, off])
ModeNumber : is_member([on, off])
ListArg : is_member([on, off])
ListNonArgVar : is_member([on, off])
Type : is_member([on, off])
GoalPath : is_member([on, off])
default value : attribute_display(on, on, on, on, off, off, off, off, on, off, off, on, off, off, on)
type of parameter : single

arguments_display(Type)
Parameter which tells how arguments shall be displayed. If Type is "simple", then arguments are displayed without taking the list_display and term_display parameters into account.
Type : is_member([normal, simple])
default value : arguments_display(normal)
type of parameter : single

list_display(Type, Range)
Parameter which tells how lists shall be displayed. If Type is "normal", lists are displayed in the standard Prolog way. If Type is "nest", the nested lists are displayed only till level Range (included). If Type is "truncate", only the first Range elements of the lists are displayed.

Type : is_member([normal, nest, truncate])
Range : integer
default value : list_display(normal, 0)
type of parameter : single

term_display(Type, Range)
Parameter which tells how structured terms shall be displayed. If Type is "normal", terms are displayed in the standard Prolog way. If Type is "nest", the nested terms are displayed only till level Range (included). If Type is "truncate", only the first Range elements of the term are displayed.
Type : is_member([normal, nest, truncate])
Range : integer
default value : term_display(normal, 0)
type of parameter : single

indent_display(OnOff, IndentationValue, Depth)
Paramater which tells whether indentation is "on" or "off", what has to be printed as indentation value, and at which depth the indentation has to be started.
OnOff : is_member([on, off])
IndentationValue : atomic
Depth : integer
default value : indent_display(on, ' ', 1)
type of parameter : single

indent_display_limit(IndentLimit)
Parameter which tells up to which depth the trace events shall be indented.
IndentLimit : integer
default value : indent_display_limit(30)
type of parameter : single

arg_undisplay(Name, ArgNo)
Parameter which tells which arguments of which predicates have to be NOT displayed. There must be one "arg_undisplay" clause for each argument which shall not be displayed.
Name : is_proc
ArgNo : integer
default value : none
type of parameter : multiple

11 Scenario "step_by_step_M"

Scenario which provides standard step by step tracing facilities. The tracing commands of this scenario are different from those of the "kernel" scenario. User can use a more simple execution model by setting the "traced_ports" parameter which filters out some of the traced events.

Commands

next *{n}*
Command which moves forward to the next trace event according to the "traced_ports" parameter. This is the same command as step/0 ("next" is the name used in the Prolog version of Opium, "step" is the name used in the internal Mercury debugger).
type of command : trace

step
See next/0.
type of command : trace

det_next
Command which does the same thing as step/0, but it is not backtrackable.
type of command : trace

det_step
See det_next/0.
type of command : trace

next(N) *{n}*
Command which prints the N next trace events according to the "traced_ports" parameter.
N : integer
type of command : opium

step(N)
See next/1.
N : integer
type of command : opium

finish *{f}*
Command which makes the execution continuing until it reaches a final port (exit or fail) of the goal to which the current event refers. If the current port is already final, it acts like

a step/0. It is the same command as skip/0 ("skip" is the name used in the Prolog version of Opium, "finish" is the name used in the internal Mercury debugger).
type of command : trace

skip *{sk}*
See finish/0.
type of command : trace

Primitives

next_np *{n_np}*
Primitive which does the same as command next except printing a trace line.

step_np
Primitive which does the same as command step except printing a trace line.

det_next_np
Primitive which does the same as command det_next except printing a trace line.

det_step_np
Primitive which does the same as command det_step except printing a trace line.

finish_np *{f_np}*
Primitive which does the same as command finish except printing a trace line.

skip_np *{sk_np}*
Primitive which does the same as command skip except printing a trace line.

Parameters

traced_ports(PortList)
Parameter which tells which events (w.r.t. ports) are to be traced by commands "next" and "step".
PortList : is_list_of_ports
default value : traced_ports([call, exit, fail, redo, then, else, switch, disj, first, later])
type of parameter : single

12 Scenario "help"

Scenario which provides the user with on-line help. There is also the facility to get a printed version of the Opium manual.

Commands

opium_help
Command which shows the help commands.
type of command : opium

show_all(ObjectType)
Command which shows all the Opium objects of a certain type, together with their arguments and their abbreviations if these exist.
ObjectType : is_member([modules, scenarios, commands, primitives, procedures, parameters, types, demos])
type of command : opium

show_all(ObjectType, Scenario)
Command which shows all the Opium objects of a certain type related to Scenario if Scenario is visible in the current module.
ObjectType : is_member([commands, procedures, primitives, parameters, types, demos])
Scenario : is_opium_scenario
type of command : tool

show_all_in_module(ObjectType, Scenario, Module)
Command which shows all the Opium objects of a certain type related to Scenario loaded in a given module.
ObjectType : is_member([commands, procedures, primitives, parameters, types, demos])
Scenario : is_opium_scenario
Module : is_opium_module
type of command : opium

show_abbreviations *{abbrevs}*
Command which shows all the abbreviations of Opium commands and primitives.
type of command : opium

show_abbreviations(Scenario) *{abbrevs}*
Command which shows all the abbreviations of commands and primitives related to Scenario if Scenario is visible in the current module.

Scenario : is_opium_scenario
type of command : tool

show_abbreviations_in_module(Scenario, Module) *{abbrevs}*
Command which shows all the abbreviations of commands and primitives related to Scenario in a given module.
Scenario : is_opium_scenario
Module : is_opium_module
type of command : opium

man(Name)
Command which describes a scenario, a command, a primitive, a procedure, a parameter, or a type. For a scenario it gives the corresponding commands, parameters, primitives, procedures, and types. For the other objects it gives the corresponding scenario.
Name : is_opium_object_or_var
type of command : opium

manual(File)
Command which shows all the scenarios, their commands and the corresponding explanations in the file "File" (in LaTeX format). It also does some fixes in the LaTeX file. The LaTeX file will then be called <File>.tex afterwards. In order to get a printable <File>.dvi, use command latex_manual/1.
File : atom
type of command : opium

latex_manual(File)
Command which applies the Unix command "latex ; makeindex ; latex" to File, where File has been generated by command manual/1. File has to be the name of the LaTeX file without extension '.tex'.
File : atom
type of command : opium

Procedures

print_header(Device, Type, Name, Module)
Procedure which prints (for on-line and paper manuals) the type of the object and whether it is global or local. Essentially used for scenarios and in many places.

print_syntax(Device, Name, ArgList, Abbrev, Type)
Procedure which prints (for on-line and paper manuals) the syntax of an object, i.e. the list of arguments, and the abbreviation if existing.

print_man(Device, ArgList, ArgType, Message, DefaultValue, ObjType)
Procedure which prints (for on-line and paper manuals) the help message of an object, the type of the arguments, the default value if a parameter and the type of the object. If you want to customize it beware that there is a patch for demos.

13 Scenario "scenario_handler"

The scenario handler manages all the Opium objects: scenarios, commands, parameters, primitives, types and demos. It also handles an error recovery mechanism and the autoload of inactive scenarios.

Commands

make(Scenario)
Command which loads a single scenario as active, traceable, and local into the current module. The declaration of a scenario must be done in a file named <scenario>.op. Only those files of the scenario are loaded which are not up-to-date. The scenario source files must be in the current directory. The object files will be put in a directory called "opiumfiles".
Scenario : atom
type of command : tool

make(Scenario, Module)
Command which loads a single scenario as active, traceable, and local into the given module. The declaration of a scenario must be done in a file named <scenario>.op. Only those files of the scenario are loaded which are not up-to-date. The scenario source files must be in the current directory. The object files will be put in a directory called "opiumfiles".
Scenario : atom
Module : atom
type of command : opium

make(Scenario, Module, OptionList)
Command which loads a single scenario into a given module. The options are active/inactive, traceable/untraceable, and global/local; they have to be given in this order. The declaration of a scenario must be done in a file named <scenario>.op. Only those files of the scenario are loaded which are not up-to-date. The scenario source files must be in the current directory. The object files will be put in a directory called "opiumfiles".
Scenario : atom
Module : atom

OptionList : is_option_list
type of command : opium

make(Scenario, Module, OptionList, SrcDir, ObjDir)
Command which loads a single scenario into a given module. The source files are taken from SrcDir, the object files are taken from, resp. written to, ObjDir. The options are active/inactive, traceable/untraceable, and global/local; they have to be given in this order. Only those files of the scenario are loaded which are not up-to-date.
Scenario : atom
Module : atom
OptionList : is_option_list
SrcDir : is_absolute_dir
ObjDir : is_absolute_dir
type of command : opium

get_parameter(Parameter, ValueList)
Command which gets the value of the parameter visible in the current module.
Parameter : is_opium_parameter
ValueList : is_list_or_var
type of command : tool

get_parameter_in_module(Parameter, ValueList, Module)
Command which gets the value of the parameter in a given module.
Parameter : is_opium_parameter
ValueList : is_list_or_var
Module : is_opium_module
type of command : opium

set_parameter(Parameter, ValueList)
Command which sets the value of the parameter visible in the current module. It automatically prompts the user for the values using the types given in the declaration of the parameter.
Parameter : is_opium_parameter
ValueList : is_list
type of command : tool

set_parameter_in_module(Parameter, ValueList, Module)
Command which sets the value of the parameter in a given module. It automatically prompts the user for the values using the types given in the declaration of the parameter.
Parameter : is_opium_parameter
ValueList : is_list
Module : is_opium_module
type of command : opium

set_parameter(Parameter)
Interactive command which helps to set the value of the parameter which is visible in the current module. It automatically checks the type of the values according to the type given in the declaration of the parameter.
Parameter : is_opium_parameter
type of command : tool

set_parameter_in_module(Parameter, Module)
Interactive command which helps to set the value of the parameter in a given module. It automatically checks the type of the values according to the type given in the declaration of the parameter.
Parameter : is_opium_parameter
Module : is_opium_module
type of command : opium

unset_parameter(Parameter, ValueList)
Command which unsets a value of a parameter which may have multiple values. For a parameter of type "single" or "c" you can use set_parameter.
Parameter : is_opium_parameter
ValueList : is_list
type of command : tool

unset_parameter_in_module(Parameter, ValueList, Module)
Command which unsets the value of a parameter which may have multiple values, in a given module. For a parameter of type "single" or "c" you can use set_parameter_in_module.
Parameter : is_opium_parameter
ValueList : is_list
Module : is_opium_module
type of command : opium

set_default_parameters
Command which sets or resets all the parameters of all the scenarios to their default values.
type of command : opium

set_default_parameters(Scenario)
Command which sets or resets the parameters of a scenario visible in the current module to their default values.
Scenario : is_opium_scenario
type of command : tool

set_default_parameters_in_module(Scenario, Module)
Command which sets or resets the parameters of a scenario to their default values in a given module.

Scenario : is_opium_scenario
Module : is_opium_module
type of command : opium

set_default(Parameter)
Command which sets or resets the default value of Parameter visible in the current module.
Parameter : is_opium_parameter
type of command : tool

set_default_in_module(Parameter, Module)
Command which sets or resets the default value of Parameter in a given module.
Parameter : is_opium_parameter
Module : is_opium_module
type of command : opium

show_parameters(Scenario)
Command which shows the values of all the parameters related to a scenario visible in the current module.
Scenario : is_opium_scenario
type of command : tool

show_parameters_in_module(Scenario, Module)
Command which shows the values of all the parameters related to Scenario in a given module.
Scenario : is_opium_scenario
Module : is_opium_module
type of command : opium

show_parameters
Command which shows the values of all the parameters of all scenarios.
type of command : opium

initialization(Goals)
When called as a compiled goal in a scenario file, initialization/1 asserts a clause which ensures that the goals given as argument of initialization/1 will be called whenever a new trace is started. NOTE: it has to be ensured that these goals refer to either global or exported predicates!
Goals : is_term
type of command : tool

set_default(ObjectType, Pred, Scenario, Module)
Commands which sets the object Pred (e.g. next/1) of type ObjectType (e.g. command) in Scenario and Module to its default value. If used with variables it will set to default the matching objects on backtracking. For parameters use set_default/1.

ObjectType : is_customizable_type_or_var
Pred : is_pred_or_var
Scenario : is_opium_scenario_or_var
Module : is_opium_module_or_var
type of command : opium

rebuild_object(ObjectType, Pred, Implementation, Scenario, Module)
Commands which links Pred (e.g. next/1) of ObjectType (e.g. command) in Scenario and Module to the given Implementation. Pred must be the name of an existing object with same arity. Implementation must be the name of a predicate (e.g. mynext). This predicate must have the same arity as the object to rebuild (except for tools commands where the implementation must be of arity +1). The existence of such a predicate is not checked by Opium.
ObjectType : is_customizable_type
Pred : is_pred
Implementation : atom
Scenario : is_opium_scenario_or_var
Module : is_opium_module_or_var
type of command : opium

Primitives

get_opium_filename(File, FileName)
Primitive which gives the full file name including suffix of an Opium file. If the file does not exist it fails and gives an error message.
File : atom
FileName : var

get_parameter_info(Parameter, Scenario, Module, ArgList, ArgType, Default, CurrentValues)
Primitive which gives information about parameter Name related to Scenario in a given module. CurrentValues is instantiated to the list of all current values.
Parameter : is_opium_parameter
Scenario : is_opium_scenario
Module : is_opium_module
ArgList : var
ArgType : var
Default : var
CurrentValues : var

opium_module(M)
Primitive which succeeds if its argument is an opium module. It can also be used to generate all the opium modules on backtracking.
M : is_opium_module_or_var

implementation_link(ObjectType, Pred, DefaultImpl, Module)
Primitive which retrieves the link between Pred, an Opium objects (e.g. next/0) of ObjectType and its default implementation visible in Module. This is useful when you want to customize an object and you want to re-use the default implementation. Only commands, primitives, procedures and types can be customized. For parameters see set_parameter.
ObjectType : is_customizable_type_or_var
Pred : is_pred_or_var
DefaultImpl : is_atom_or_var
Module : is_opium_module_or_var

Procedures

opium_scenario_in_module(Scenario, Module)
Procedure which succeeds if Scenario is declared in Module.

opium_command_in_module(Command, Module)
Procedure which succeeds if Command is declared in Module.

opium_primitive_in_module(Primitive, Module)
Procedure which succeeds if Primitive is declared in Module.

opium_procedure_in_module(Procedure, Module)
Procedure which succeeds if Procedure is declared in Module.

opium_parameter_in_module(Parameter, Module)
Procedure which succeeds if Parameter is declared in Module.

opium_type_in_module(Type, Module)
Procedure which succeeds if Type is declared in Module.

opium_demo_in_module(Demo, Module)
Procedure which succeeds if Demo is declared in Module.

modify_time(File, Time)
Procedure which returns the Time when File has been modified. If the file does not exist, it returns 0.

check_arg_type(ArgValList, ArgNameList, ArgTypeList, NewValList, Module)
Procedure which checks the types of a list of arguments. If the type of an argument is not correct the user will be prompted for another value. If ArgVal is [] but ArgTypeList is not [] then the procedure will prompt the user for proper values. The types have to be visible in Module.

check_arg(ArgValue, ArgName, ArgType, NewValue, Module)
Procedure which is called to check the type of a single argument. If the type of an argument is not correct the user will be prompted for another value until the new argument has the proper type. The type has to be visible in Module.

Types

is_opium_scenario
Type which succeeds for an active opium scenario.

is_opium_scenario_or_var
Type which succeeds for an active opium scenario or a variable.

is_opium_parameter
Type which succeeds for a parameter of an opium scenario.

is_opium_object_or_var
Type which succeeds for an object declared in an opium scenario, or a variable. An Opium object is a scenario, a command, a primitive, a procedure, a parameter, or a type.

is_option_list
Type which succeeds for a list of options for a scenario: [active/inactive, traceable/untraceable, global/local].

is_absolute_dir
Type which succeeds for an atom starting with "/" and ending with "/".

is_opium_module
Type which succeeds for the name of a module which contains an opium scenario, or a module which has been initialized interactively by calling an Opium command or primitive.

is_opium_module_or_var
Type which succeeds if the argument is an opium module or a variable.

is_customizable_type_or_var
Type which succeeds for a type of Opium object which is customizable, or a variable. Customizable types are command, primitive, procedure, type.

is_customizable_type
Type which succeeds for a type of Opium object which is customizable. Customizable types are command, primitive, procedure, type.

is_list_of_vars_or_empty_list
Type which succeeds for a list of variables, or the empty list.

is_list_of_atoms_or_empty_list
Type which succeeds for a list containing atoms, or an empty list.

is_opium_declaration
Type which succeeds for an opium object (eg. opium_command/9). The arity of the predicate must be the arity of the declaration in the source.

is_pred
Type which succeeds for a predicate id of the form P/A or M:P/A. The default module is the toplevel module in the traced session.

is_pred_id
Type which succeeds for a predicate id which consists of P/A only.

is_goal_or_var
Type which succeeds for a term which is either a var, a compound term or an atom.

is_goal
Type which succeeds for a term which is either a compound term or an atom.

is_list
Type which succeeds for any list.

is_term
Type which succeeds for any Prolog term.

is_pred_or_var
Type which succeeds for a predicate id of the form P/A, or M:P/A, or a variable. The default module is the toplevel module in the traced session.

is_integer_or_var
Type which succeeds for an integer or a variable.

is_atom_or_var
Type which succeeds for an atom or a variable.

is_string_or_var
Type which succeeds if its argument is a string or a variable.

is_string_or_integer_or_var
Type which succeeds if its argument is a string or a variable.

is_string_or_integer
Type which succeeds if its argument is a string or an integer.

is_list_or_var
Type which succeeds for a list or a variable.

is_list_of_atoms
Type which succeeds for a list of atoms.

is_atom_or_list_of_atoms
Type which succeeds for a single atom or a non-empty list of atoms.

is_list_of_preds
Type which succeeds for a list of predicate ids of the form P/A or M:P/A. The default module is the toplevel module in the traced session.

is_pred_or_list_of_preds
Type which succeeds for a predicate id, or for a list of predicate ids of the form P/A or M:P/A.

is_list_of_integers
Type which succeeds for a list of integers.

is_list_of_integers_or_var
Type which succeeds for a list of integers or a variable.

Part III

Appendices

A Listing of qsort.m

```
%   qsort
%
%   adapted from a Prolog program written by David H. D. Warren
%
%   quicksort a list of integers

:- module qsort.

:- interface.

:- import_module list, int, printlist, io.

:- pred main1(list(int)).
:- mode main1(out) is det.

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

:- pred main3(list(int), io__state, io__state).
:- mode main3(out, di, uo) is det.

:- implementation.

main --> main3(_).

main1(Out) :-
        data(Data),
        qsort(Data, Out, []).

main3(Out) -->
        { main1(Out) },
        print_list(Out).
```

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

:- pred qsort(list(int), list(int), list(int)).
:- mode qsort(in, out, in) is det.

:- pred partition(list(int), int, list(int), list(int)).
:- mode partition(in, in, out, out) is det.

data([3, 1, 2]).

qsort([X|L], R, R0) :-
        partition(L, X, L1, L2),
        qsort(L2, R1, R0),
        qsort(L1, R, [X|R1]).
qsort([], R, R).

partition([], _P, [], []).
partition([H|T], P, Lo, Hi) :-
        ( H =< P ->
                partition(T, P, Lo1, Hi),
                Lo = [H|Lo1]
        ;
                partition(T, P, Lo, Hi1),
                Hi = [H|Hi1]
        ).


%-----------------------------------------------------------------------------%
:- module printlist.

:- interface.

:- import_module list, int, io.

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

:- implementation.

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)
        ).
```

B Listing of the buggy mastermind program

```
%-----------------------------------------------------------------------%
%
% Author : Erwan Jahier
% File   : mastermind.m
%
%-----------------------------------------------------------------------%

:- module mastermind.
```

```
:- interface.
:- import_module io, list.

:- type color
        --->    blue
        ;       white
        ;       black
        ;       yellow
        ;       gray
        ;       red
        ;       brown
        ;       orange
        ;       hole.


:- type mastermind
        ---> mastermind(color, color, color, color, color).

:- type store ---> store(mastermind, int, int).

:- type database == list(store).

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

%-----------------------------------------------------------------------%

:- implementation.
:- import_module mastermind_generator, mastermind_checker, display, list, int,
        require, std_util.


main -->
        ask_user_for_a_mastermind(Mastermind),
        solve_mastermind(Mastermind).


%-----------------------------------------------------------------------%
:- pred solve_mastermind(mastermind, io__state, io__state).
:- mode solve_mastermind(in, di, uo) is det.
solve_mastermind(Code) -->
        { generate_all_possible_guess(ListofGuesses) },
        { solve_mastermind_step(Code, 0, NumberOfGuesses, [], Database,
```

```
                ListofGuesses, _) },
        display_mastermind(Code, Database),

%       Display the results when tcl/tk display is broken.
        write_string("The solution was found with "),
        write(NumberOfGuesses),
        write_string(" guesses.\n"),
        write_string("The successive guesses were :\n"),
        print_database(Database),
        nl.

%-----------------------------------------------------------------------------%
:- pred solve_mastermind_step(
        mastermind::in,         % Code to break
        int::in,                % Guess counter
        int::out,               % New guess counter
        database::in,           % List of previous attemps
        database::out,          % New list of attemps
        list(mastermind)::in,   % List of untried attemps
        list(mastermind)::out   % New list of untried attemps
                ) is det.
solve_mastermind_step(Code, GuessNumberIn, GuessNumberOut, DatabaseIn,
        DatabaseOut, ListofGuessesIn, ListofGuessesOut) :-
        (
                propose_a_guess(DatabaseIn, ListofGuessesIn, ListofGuesses2,
                        Guess),
                check_mastermind(Guess, Code, Bulls, Cows),
                ( Bulls = 5 ->
                        DatabaseOut = [ store(Guess, Bulls, Cows) | DatabaseIn],
                        GuessNumberOut is GuessNumberIn + 1,
                        ListofGuessesOut = ListofGuesses2
                ;
                        solve_mastermind_step(Code,
                                GuessNumberIn + 1, GuessNumberOut,
                                [ store(Guess, Bulls, Cows) | DatabaseIn],
                                DatabaseOut,
                                ListofGuesses2,
                                ListofGuessesOut )
                )
        ;
                % Should never occur
                 error(" in solve_mastermind_step/6:
```

```
                I have tried all the possible solutions and I was not able
                to break that code !\n")
        ).


%-----------------------------------------------------------------------%
:- pred is_guess_sensible(mastermind, database).
:- mode is_guess_sensible(in, in) is semidet.
is_guess_sensible(_, []).
is_guess_sensible(Guess, [ store(OldGuess, Bulls, Cows) | DatabaseTail]) :-
        % The guess is sensible if it produces the same number of cows and
        % bulls with the previous attemps.
        check_mastermind(OldGuess, Guess, Bulls, Cows),
        is_guess_sensible(Guess, DatabaseTail).


:- pred propose_a_guess(database, list(mastermind), list(mastermind),
        mastermind).
:- mode propose_a_guess(in, in, out, out) is semidet.
propose_a_guess(Database, ListofGuessesIn, ListofGuessesOut, ProposedGuess) :-
        ListofGuessesIn = [ Guess | TailGuesses ],
        ( is_guess_sensible(Guess, Database) ->
                ListofGuessesOut = TailGuesses,
                ProposedGuess = Guess
        ;
                propose_a_guess(Database, TailGuesses, ListofGuessesOut,
                        ProposedGuess)
        ).


:- pred generate_all_possible_guess(list(mastermind)).
:- mode generate_all_possible_guess(out) is det.
generate_all_possible_guess(ListOfGuesses) :-
        solutions(generate_a_guess, ListOfGuesses).


:- pred generate_a_guess(mastermind).
:- mode generate_a_guess(out) is multi.
generate_a_guess(Guess) :-
        Guess = mastermind(C1, C2, C3, C4, C5),
        select_color(C1),
        select_color(C2),
```

```
        select_color(C3),
        select_color(C4),
        select_color(C5).


:- pred select_color(color).
:- mode select_color(out) is multi.
select_color(blue).
select_color(white).
select_color(black).
select_color(yellow).
select_color(gray).
select_color(red).
select_color(brown).
select_color(orange).


%-----------------------------------------------------------------------%
:- pred print_database(database, io__state, io__state).
:- mode print_database(in, di, uo) is det.
print_database(Database) -->
        { reverse(Database, DatabaseReversed) },
        print_database2(DatabaseReversed, 1),
        nl.


:- pred print_database2(database, int, io__state, io__state).
:- mode print_database2(in, in, di, uo) is det.
print_database2([], _) -->
        nl.

print_database2([store(mastermind(C1, C2, C3, C4, C5), Bulls, Cows)|Tail], N) -->
        write_string("    "),
        write(N),
        write_string(": "),
        write(C1),
        write_string(" "),
        write(C2),
        write_string(" "),
        write(C3),
        write_string(" "),
        write(C4),
```

```
        write_string(" "),
        write(C5),
        write_string(" ("),
        write(Bulls),
        write_string(" Bulls and "),
        write(Cows),
        write_string(" Cows).\n"),
        print_database2(Tail, N + 1).

%-----------------------------------------------------------------------%
%
% Author : Erwan Jahier
% File   : mastermind_checker.m
%
%
% This module implements the check_mastermind/4 predicate.
% This predicate takes in input a Code and a Guess and outputs the number of
% Bulls (colors that appears in identical position in the Guess and in the Code)
% and Cows (colors that appears in the Guess and in the code but at wrong
% positions).

:- module mastermind_checker.

:- interface.
:- import_module mastermind, int.

:- pred check_mastermind(mastermind, mastermind, int, int).
:- mode check_mastermind(in, in, out, out) is det.

:- implementation.
:- import_module int, list, mastermind.


%-----------------------------------------------------------------------%
check_mastermind(Guess, Code, Bulls, Cows) :-
         count_bulls(Guess, Code, Bulls, Guess2, Code2),
        % Code2 and  Guess2 is the same as Code and Guess except that
        % we have removed the well placed whatsits.
         count_cows(Guess2, Code2, Cows).


:- pred count_bulls(mastermind, mastermind, int, mastermind, mastermind).
```

```
:- mode count_bulls(in, in, out, out, out) is det.
count_bulls(Guess, Code, Bulls, Guess2, Code2) :-
        Guess = mastermind(G1, G2, G3, G4, G5),
        Code = mastermind(Color1, Color2, Color3, Color4, Color5),
        Guess2 = mastermind(
                        remove_bulls(G1, Color1),
                        remove_bulls(G2, Color2),
                        remove_bulls(G3, Color3),
                        remove_bulls(G4, Color4),
                        remove_bulls(G5, Color5)
                                ),
        Code2 = mastermind(
                        remove_bulls(Color1, G1),
                        remove_bulls(Color2, G2),
                        remove_bulls(Color3, G3),
                        remove_bulls(Color4, G4),
                        remove_bulls(Color5, G5)
                                ),
        count_hole(Guess2, Bulls).


:- func remove_bulls(color, color) = color.
:- mode remove_bulls(in, in) = out is det.
remove_bulls(G, Color) = Out :-
        % We remove the well placed whatsits
        ( (G = Color) ->
                Out = hole
        ;
                Out = G
        ).


:- pred count_hole(mastermind, int).
:- mode count_hole(in, out) is det.
count_hole(mastermind(C1, C2, C3, C4, C5), N) :-
        Counter0 = 0,
        count_hole2(C1, Counter0, Counter1),
        count_hole2(C2, Counter1, Counter2),
        count_hole2(C3, Counter2, Counter3),
        count_hole2(C4, Counter3, Counter4),
        count_hole2(C5, Counter4, N).
```

```
:- pred count_hole2(color, int, int).
:- mode count_hole2(in, in, out) is det.
count_hole2(Color, Cin, Cout) :-
        ( (Color = hole) ->
                Cout is Cin + 1
        ;
                Cout = Cin
        ).


:- pred count_cows(mastermind, mastermind, int).
:- mode count_cows(in, in, out) is det.
count_cows(Guess, Code, Cows) :-
        % Outputs the number of cows in the Guess.
        Guess = mastermind(G1, G2, G3, G4, G5),
        Code = mastermind(Color1, Color2, Color3, Color4, Color5),
        List1 = [G1, G2, G3, G4, G5],
        remove_cows_from_list(Color1, List1, List2),
        remove_cows_from_list(Color2, List2, List3),
        remove_cows_from_list(Color3, List3, List4),
        remove_cows_from_list(Color4, List4, List5),
        remove_cows_from_list(Color5, List5, List6),
        list__length(List6, M),
        Cows is 5 - M.


:- pred remove_cows_from_list(color, list(color), list(color)).
:- mode remove_cows_from_list(in, in, out) is det.
remove_cows_from_list(_, [], []).
remove_cows_from_list(C, [X | Xs], ListOut) :-
        (
                C \= hole,
                C = X
         ->
                ListOut = Xs
        ;
                remove_cows_from_list(C, Xs, List),
                ListOut = [C | List]    %  Bug !
                % ListOut = [X | List]  % Correct code.
        ).
```

```
%--------------------------------------------------------------------------%


%--------------------------------------------------------------------------%
%
% Author : Erwan Jahier
% File   : mastermind_generator.m
%
% This module provides  mastermind generators:
%  - ask_user_for_a_mastermind/1
%  - generate_random_mastermind/3.
%
% The first one will prompt the user to enter a mastermind problem. The second
% one will pseudo-randomly generate mastermind problems.

:- module mastermind_generator.

:- interface.
:- import_module mastermind, io, random.

:- pred ask_user_for_a_mastermind(
        mastermind::out,
        io__state::di,
        io__state::uo)is det.

:- pred generate_random_mastermind(
        mastermind::out,
        random__supply::mdi,
        random__supply::muo) is det.

:- pred convert_string_into_color(string, color).
:- mode convert_string_into_color(in, out) is semidet.
:- mode convert_string_into_color(out, in) is semidet.

:- implementation.
:- import_module mastermind, random, term, term_io, io, list, int.


%--------------------------------------------------------------------------%
ask_user_for_a_mastermind(Mastermind) -->
        write_string("Enter a mastermind problem :\n"),
        term_io__read_term(ReadTerm),
```

```
        (
                { convert_readterm_into_mastermind(ReadTerm, Mastermind1) }
        ->
                { Mastermind = Mastermind1 }
        ;
                write_string("\nYou should write something like:\n"),
                write_string("mastermind(blue, white, black, yellow, gray)."),
                write_string("\nPossible colors are: "),
                write_string("blue, white, black, yellow, gray, red, brown, "),
                write_string("orange.\n"),
                ask_user_for_a_mastermind(Mastermind)
        ).


:- pred convert_readterm_into_mastermind(read_term, mastermind).
:- mode convert_readterm_into_mastermind(in, out) is semidet.
convert_readterm_into_mastermind(ReadTerm, Mastermind) :-
        ReadTerm = term(_Varset, Term),
        Term = functor(
                atom("mastermind"),
                [
                        functor(atom(ColorStr1), _, _),
                        functor(atom(ColorStr2), _, _),
                        functor(atom(ColorStr3), _, _),
                        functor(atom(ColorStr4), _, _),
                        functor(atom(ColorStr5), _, _)
                ],
                context("<standard input>", _)
                        ),
        convert_string_into_color(ColorStr1, Color1),
        convert_string_into_color(ColorStr2, Color2),
        convert_string_into_color(ColorStr3, Color3),
        convert_string_into_color(ColorStr4, Color4),
        convert_string_into_color(ColorStr5, Color5),
        Mastermind = mastermind(Color1, Color2, Color3, Color4, Color5).


% :- pred convert_string_into_color(string, color).
% :- mode convert_string_into_color(in, out) is semidet.
% :- mode convert_string_into_color(out, in) is semidet.
convert_string_into_color("red", red).
convert_string_into_color("blue", blue).
```

```
convert_string_into_color("white", white).
convert_string_into_color("yellow", yellow).
convert_string_into_color("gray", gray).
convert_string_into_color("brown", brown).
convert_string_into_color("orange", orange).
convert_string_into_color("black", black).


%-----------------------------------------------------------------------%
        % random__init(0, RS)
        % generate_random_mastermind(Mastermind, RS, RSout)
        % ...
generate_random_mastermind(mastermind(C1, C2, C3, C4, C5), RSin, RSout) :-
        generate_one_color(C1, RSin, RS1),
        generate_one_color(C2, RS1, RS2),
        generate_one_color(C3, RS2, RS3),
        generate_one_color(C4, RS3, RS4),
        generate_one_color(C5, RS4, RSout).


:- pred generate_one_color(color, random__supply, random__supply).
:- mode generate_one_color(out, mdi, muo) is det.
generate_one_color(Color, RSin, RSout) :-
        random__random(Num,  RSin, RS1),
        random__randmax(RandMax, RS1, RSout),
        NNum is Num * 8,
        (
                NNum < RandMax
        ->
                Color = blue
        ;
                NNum < 2*RandMax
        ->
                Color = white
        ;
                NNum < 3*RandMax
        ->
                Color = black
        ;
                NNum < 4*RandMax
        ->
                Color = yellow
```

```
        ;
                NNum < 5*RandMax
        ->
                Color = gray
        ;
                NNum < 6*RandMax
        ->
                Color = red
        ;
                Color = brown
        ).

%------------------------------------------------------------------------%
%
% Author : Erwan Jahier
% File   : display.m
%
% Display a mastermind board via a tcl/tk script

:- module display.

:- interface.
:- import_module io, mastermind.


:- pred display_mastermind(mastermind, database, io__state, io__state).
:- mode display_mastermind(in, in, di, uo) is det.

%------------------------------------------------------------------------%

:- implementation.
:- import_module mastermind, mastermind_generator, mastermind_checker, list, int,
        require, std_util, string.


display_mastermind(Code, Database) -->
        { generate_tcltk_command(Code, Database, TclTkCmd) },
        io__call_system(TclTkCmd, _Res).


:- pred generate_tcltk_command(mastermind, database, string).
:- mode generate_tcltk_command(in, in, out) is det.
```

```
generate_tcltk_command(Code, Database, TclTkCmd) :-
        Code = mastermind(C1, C2, C3, C4, C5),
        CodeList = [C1, C2, C3, C4, C5],
        list__length(Database, Card),
        int_to_string(Card, CardStr),
        transform_list_into_command(CodeList, CodeCmd2),
        wrap_with(CodeCmd2, " \" ", " \" ", CodeCmd),
        transform_database_into_command(Database, DatabaseCmd),
        append("display_mastermind ", CodeCmd, A1),
        append(A1, DatabaseCmd, A2),
        append(A2, CardStr, TclTkCmd).


:- pred transform_database_into_command(database, string).
:- mode transform_database_into_command(in, out) is det.
transform_database_into_command(Database, DatabaseCmd) :-
        transform_database_into_command2(Database, SubmitionCmd, ResponseCmd),
        wrap_with(SubmitionCmd, " \" ", " \" ", SubmitionCmdNew),
        wrap_with(ResponseCmd, " \" ", " \" ", ResponseCmdNew),
        append(SubmitionCmdNew, ResponseCmdNew, DatabaseCmd).


:- pred transform_database_into_command2(database, string, string).
:- mode transform_database_into_command2(in, out, out) is det.
transform_database_into_command2(Database, SubmitionList, ResponseList) :-
        (
                Database = [],
                SubmitionList = " ",
                ResponseList = " "
        ;
                Database = [Store | Tail],
                Store = store(mastermind(C1, C2, C3, C4, C5), Bulls, Cows),
                List = [C1, C2, C3, C4, C5],
                transform_list_into_command(List, Submition),
                wrap_with(Submition, " { ", " } ", Submition2),
                generate_response_command(Bulls, Cows, Response),
                transform_database_into_command2(Tail, SubmitionTail, ResponseTail),
                append(Submition2,  SubmitionTail, SubmitionList),
                append(Response, ResponseTail, ResponseList)
        ).
```

```
:- pred generate_response_command(int, int, string).
:- mode generate_response_command(in, in, out) is det.
generate_response_command(Bulls, Cows, Response) :-
        generate_response(Bulls, " black ", BullsCmd),
        generate_response(Cows, " white ", CowsCmd),
        append(BullsCmd, CowsCmd, BullsCowsCmd),
        wrap_with(BullsCowsCmd, " { ",  " } ", Response).


:- pred generate_response(int, string, string).
:- mode generate_response(in, in, out) is det.
generate_response(N, Color, Cmd) :-
        (
                N = 0
        ->
                Cmd = " "
        ;
                % N > 0,
                NN is N - 1,
                generate_response(NN, Color, Cmd2),
                append(Color, Cmd2, Cmd)
        ).


:- pred transform_list_into_command(list(color), string).
:- mode transform_list_into_command(in, out) is det.
transform_list_into_command(List , Cmd) :-
        (
                List = [],
                Cmd = ""
        ;
                List = [C | Tail],
                transform_list_into_command(Tail, TailCmd),
                ( convert_string_into_color(CStr, C) ->
                        append(CStr, " ", A1),
                        append(A1, TailCmd, Cmd)
                ;
                        Cmd = TailCmd
                )
        ).
```

```
:- pred wrap_with(string, string, string, string).
:- mode wrap_with(in, in, in, out) is det.
wrap_with(StringIn, Before, After, StringOut) :-
        append(Before, StringIn, A1),
        append(A1, After, StringOut).
```

C Listing of Step_by_step_M scenario

```
%-------------------------------------------------------------------------------%
% Copyright (C) 1999 IRISA/INRIA.
%
% Author : Erwan Jahier <jahier@irisa.fr>
%

opium_scenario(
        name            : 'step_by_step_M',
        files           : ['step_by_step_M'],
        scenarios       : [],
        message         :
"Scenario which provides standard step by step tracing facilities. The tracing \
commands of this scenario are different from those of the ``kernel'' scenario. \
User can use a more simple execution model by setting the ``traced_ports'' \
parameter which filters out some of the traced events."
        ).


%-------------------------------------------------------------------------------%
opium_command(
        name            : next,
        arg_list        : [],
        arg_type_list   : [],
        abbrev          : n,
        interface       : button,
        command_type    : trace,
        implementation  : next_Op,
        parameters      : [traced_ports],
        message         :
"Command which moves forward to the next trace event according to the \
```

```
``traced_ports'' parameter. This is the same command as step/0 (``next'' is \
the name used in the Prolog version of Opium, ``step'' is the name used in the \
internal Mercury debugger)."
        ).


opium_command(
        name            : step,
        arg_list        : [],
        arg_type_list   : [],
        abbrev          : _,
        interface       : hidden,
        command_type    : trace,
        implementation  : next_Op,
        parameters      : [traced_ports],
        message         :
"See next/0."
        ).

next_Op :-
        traced_ports(PortList),
        fget_np(port = PortList).


%-----------------------------------------------------------------------------%
opium_command(
        name            : det_next,
        arg_list        : [],
        arg_type_list   : [],
        abbrev          : _,
        interface       : menu,
        command_type    : trace,
        implementation  : det_next_Op,
        parameters      : [traced_ports],
        message         :
"Command which does the same thing as step/0, but it is not backtrackable."
        ).

opium_command(
        name            : det_step,
        arg_list        : [],
        arg_type_list   : [],
```

```
        abbrev          : _,
        interface       : hidden,
        command_type    : trace,
        implementation  : det_next_Op,
        parameters      : [traced_ports],
        message         :
"See det_next/0."
        ).

det_next_Op :-
        next_np,
        !.


%-------------------------------------------------------------------------------%
opium_command(
        name            : next,
        arg_list        : [N],
        arg_type_list   : [integer],
        abbrev          : n,
        interface       : menu,
        command_type    : opium,
        implementation  : next_Op,
        parameters      : [traced_ports],
        message         :
"Command which prints the N next trace events according to the ``traced_ports'' \
parameter."
        ).

opium_command(
        name            : step,
        arg_list        : [N],
        arg_type_list   : [integer],
        abbrev          : _,
        interface       : hidden,
        command_type    : opium,
        implementation  : next_Op,
        parameters      : [traced_ports],
        message         :
"See next/1."
        ).
```

```
next_Op(N) :-
        N =< 0,
        !.
next_Op(N) :-
        setval(next_counter, N),
        next,
        decval(next_counter),
        getval(next_counter, 0),
        !.


%-------------------------------------------------------------------------------%
opium_command(
        name            : finish,
        arg_list        : [],
        arg_type_list   : [],
        abbrev          : f,
        interface       : button,
        command_type    : trace,
        implementation  : skip_Op,
        parameters      : [],
        message         :
"Command which makes the execution continuing until it reaches a final port \
(exit or fail) of the goal to which the current event refers. If the current \
port is already final, it acts like a step/0.\n\
It is the same command as skip/0 (``skip'' is the name used in the Prolog \
version of Opium, ``finish'' is the name used in the internal Mercury \
debugger)."
).

opium_command(
        name            : skip,
        arg_list        : [],
        arg_type_list   : [],
        abbrev          : sk,
        interface       : hidden,
        command_type    : trace,
        implementation  : skip_Op,
        parameters      : [],
        message         :
 "See finish/0."
        ).
```

```
skip_Op :-
        current(port = Port),
        skip_int(Port).

skip_int(Port) :-
        is_quit_port(Port),
        !,
        det_next_np.
skip_int(_) :-
        current(call = Call),
        fget_np(call = Call and port = [exit, fail]),
        !.

is_quit_port(exit).
is_quit_port(fail).


%-------------------------------------------------------------------------------%
opium_parameter(
        name            : traced_ports,
        arg_list        : [PortList],
        arg_type_list   : [is_list_of_ports],
        parameter_type  : single,
        default         : [[call, exit, fail, redo, then, else,
                                switch, disj, first, later]],
        commands        : [next],
        message         :
"Parameter which tells which events (w.r.t. ports) are to be traced by \
commands ``next'' and ``step''."
        ).
```

D Index

Index

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

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