A debugger for the Gobo Eiffel compiler
— User's guide —
Version: 2.0

Wolfgang Jansen

November 22, 2014

Contents

1 Introduction

This manual describes a graphical debugger for the *Gec* compiler. The debugger has two levels of functionality. First, a reduced version of the debugger (called the post mortem analyser) is added to a system if it is compiled not in finalized mode and without the proper debugger (which will be called runtime analyser in the following while debugger means both, post mortem analyser and runtime analyser). The effect is that the post mortem analyser gets invoked if the program crashes for some reason or when the program is interrupted from keyboard. This way, the call stack as well as data on stack and heap can be analysed. The post mortem analyser has been inspired by the `core` file written by *Unix* in case of a program crash: the file can then be used by a debugger to analyse the program state at the point of the crash. So, why not invoking the debugger at once? The post mortem analyser of *Gedb* gives a positive answer. In more detail, the post mortem analyser provides functionality for the following topics:

- moving up and down the call stack (see Sec. 3.8);
- computing and displaying data (see Secs. 3.9, 3.10, 3.11 except 3.11.2);
- listing source code (see Sec. 3.6).

Additionally, the runtime analyser provides functionality for

- defining and manipulating breakpoints (see Sec. 3.5);
- running the program in a stop-and-go manner (see Sec. 3.4);
- assigning values to variables (see Sec. 3.11.2).

In contrast to debuggers like *gdb* that run debugger and debuggee in two different processes, the *Gedb* follows a poor man's approach: at many places of the generated *C* code calls to the debugger are inserted during compilation to check whether the debugger should become active, in particular checking for breakpoints. [1] Depending on the number and complexity of breakpoints, the chosen technique increases dramatically the run time between breakpoints (but does not detectable slow down the debuggee when progressing the system step by step since then most time is consumed by user interactions). The post mortem analyser does not slow down the computation so much: at the places in *C* code where the runtime analyser would be called it simply sets a local variable to the program position. Together, with the meta-information linked into the debuggee this is sufficient for the *Gedb* to do its job after crash.

To integrate the debugger into the *Gec* compiler the compiler had to be modified. But this concerns the code generation and command line analysis only, so the compiler's integrity is preserved. The debuggee's integrity is preseved as well since only additions to the *C* code are made that do not modify the behaviour of the normally generated code (of course, the additions cause the longer runtime).

The debugger uses the *GTK+* graphics library for the graphical user interface (GUI for short). *GTK+* and related libraries as *GLib* etc. are written in *C* and are intended to be used from *C* programs. This is a hard job for an *Eiffel* programmer. But there is the OO programming language *Vala* that is a shell for *GLib* and *GTK+* [2], so the debugger has been written in *Vala*. The result is a library (a shared object under *POSIX* compliant operation systems, a *DLL* under *Windows*) that is to be linked to the *Gec* generated program.

So far it has been described what the debugger is and what it does. Another question is what the debugger does not. On the lower end, some traditional debuggers like the *gdb* allow for low level debugging on the level of assembler code. This is not possible here since the *Gec* compiles to *C* code, not to assembler code, and the necessary meta-information is not available. In the middle, the debugger does not support debugging of multi-threaded *Eiffel* systems: each thread has its own stack and it is not yet possible from

[1]The more intelligent approach of *gdb* and the like which run the debuggee between breakpoints at full speed is not possible. This approach required that information about the generated machine code (among others, the correspondence of machine addresses and program lines) is provided by the compiler to be known to the debugger. But this is not the case here, since the *Gec* compiles to *C* code, not to machine code.

[2]Like *Gec* the *Vala* compiler generates *C* code but unlike *Gec* much of the generated code is calling *GLib* and *GTK+* functions. As legacy from *GLib Vala* also utilizes a different memory management what makes some implementation details rather tricky. Unfortunately, co-operation with the *BDW* garbage collector used by *Gec* could not be tested: even rather simple *Eiffel* systems compiled with garbage collection but without debugger always crash.

the debugger's side to distinguish between the stacks. On the upper end, the *Gedb* is not an *IDE*, i.e. it does not include an editor, nor the *Gec*, nor any kind of project management.

The debugger should help, as the name indicates, to find bugs in an *Eiffel* system, but itself must be bug free. This is the ideal, not the reality. One deficiency is already known and cannot be solved in the given framework: the debugging session may fail if the debuggee uses (via `external` routines) the *GTK+* graphics library as does the debugger. The calls of both programs to this library may interfere in an uncontrolled way causing a crash. Several other debugger errors are caught by exception handling: execution of the actual command is cancelled and an error message is issued. It may happen that even this emergency break does not work and the debugger crashes during command execution. In such a case, please send a bug report to

`mailto:wo.jansen@kabelmail.de`

2 Installation and Usage

2.1 Prerequisites

The debugger is part of our *Gec* distribution. It is shipped as the *C* code generated by the *Vala* compiler, the debugger will be generated during the installation of *Gec*. The debugger's *C* code relies on the graphical library *GTK+* and the prerequisites of that, so the essential prerequisites are

- *GTK+* version 3.6 or higher
  (see `http://www.gtk.org/download`);
- libraries used by *GTK+*, these libraries are in general shipped together with *GTK+*;
- additional *Vala* libraries: *gee-0.8*, *gmodule-2.0*, *gtksourceview-3.0*
  (see `https://github.com/GNOME/libgee`).

It should be emphasized that the *Vala* compiler is not needed but installing it will also install the prerequisites listed. On the other hand, the original *Vala* source code is included in the distribution, too. If the debugger is to be installed from the sources then the following prerequisites are needed additionally:

- the *Vala* compiler
  (see `https://wiki.gnome.org/Projects/Vala`);

Finally, the extended *Gec* and the debugger sources can be downloaded from branch `extra` of the *git* repository at
`https://github.com/wjansen`
(branch `master` provides the original *Gec* compiler).

2.2 Installation

Installation consists of two steps. First, installing the extended *Gec* compiler (this is done the usual way), thereby the debugger is installed, too. The result is, besides the common programs `gec` etc., the library `libgedb.so` (or `gedb.dlls` on *Windows*) that contains the bulk of debugger code. The debuggee is the main program linking `libgedb.so`.

Second, to prepare an *Eiffel* system for debugging operation system specific options are to be provided in files `$GOBO/tool/gec/config/c/*.cfg`. For example, in case of *Linux* using *gcc* the following lines have to be modified or added. In section `Command lines` line

```
link: gcc $lflags -o $exe $objs $libs $gc_libs -lm
```

becomes

```
link: gcc $lflags -o $exe $objs $libs $gc_libs $dg_libs -lm
```

and in section `Variables` (additional flags to link the debugger library):

```
#ifdef EIF_DEBUGGER
dg_libs: -L${GOBO}/bin -lgedb
#else
dg_libs:
#endif
```

These settings have to be adapted for the other configuration files if they are to be used. Moreover, in case of *Unix* like operation systems the environment variable `LD_LIBRARY_PATH` must point (among others) to `$GOBO/bin`, alternatively, you may add the cryptic flag `-Wl,-rpath,$GOBO/bin` (no internal spaces!) to variable `dg_libs`.

Hint

> If the debugger is to be installed form the *Vala* source code switch to directory `$GOBO/tool/gedb` and run `make` before installing *Gec*.

2.3 Usage

The use of the debugger is controlled by *Gec* command line options and/or options in the `.xace` file (corresponding settings in an `.ace` file are ignored). The system is not prepared for debugging (and a traditional system is generated) if it is compiled in finalized mode (option `--finalize` on the command line or `<option name="finalize" value="true"/>` in the`.xace` file). If not in finalized mode then the presence or absence of command line option `--gedb=`*choice* controls which debugger variant will be added to the system:

- option absent: the post mortem analyser is added;

- option present without parameter: the runtime analyser is added;
- option present with parameter: the parameter *choice* must be one of

  none : no debugger is added

  pma : the post mortem analyser is added

  full : the runtime analyser is added

(so, no option or the parameterless option will be sufficient in most cases). It is also possible to generate a debugged system by use of *geant*: add option `-Ddebug` to the command line. This option is originally used to prevent finalizing, now it has the second effect of adding the post mortem analyser. The result of the compilation is an executable program if the post mortem analyser is added but it is a loadable library if the runtime analyser is added (precisely, the runtime analyser is the executable program that loads the library).

The generation of debugging code may be switched off or on for particular classes or for the classes in chosen clusters. To this end, options like the following have to be added to the `.xace` file:

```
<cluster name="some_cluster">
  <option name="debugger" value="false"/>
     ...
  <class name="SOME_CLASS">
    <option name="debugger" value="true"/>
       ...
  </class>
     ...
</cluster>
```

For each class the class specific option (if set) or that of the narrowest enclosing cluster (if any) is applied, settings in wider clusters have no effect on the class, and if no option has been set at all then the default value `true` applies. If debugging code is not generated for a class then meta-information about its variable attributes and local variables of its routines (including routine arguments and object test variables) is generated as usually, i.e. the may be examined. But the class is treated in the manner of the post mortem analyser: breakpoints cannot be set and the system does not stop within routines of the class (but it can be stopped in routines called from the class). This will improve the runtime behaviour of the system, in particular, if well tested library classes are excluded from debugging. For convenience, if the system is compiled by means of *geant* where classes from kernel libraries are provided in the `.xace` file by tags

```
<mount location="$GOBO/library/library.xace"/>
<mount location="$GOBO/library/free_elks/library.xace"/>
```

then these classes are automatically excluded from debugging. If debugging code for some classes or subclusters should be generated anyway then the appropriate options have to be set in the *geant* generated file `ge.xace`.

The generated program is started as usually. Then after program start (in case of the runtime analyser) or program crash (in case of the post mortem analyser) the debugger GUI is shown waiting for user interactions. Debugger work (and running the debuggee) is terminated by menu item File → Quit or simply by pressing `Control-q` when the debugger's main window has the keyboard focus.

3 GUI

3.1 Overview

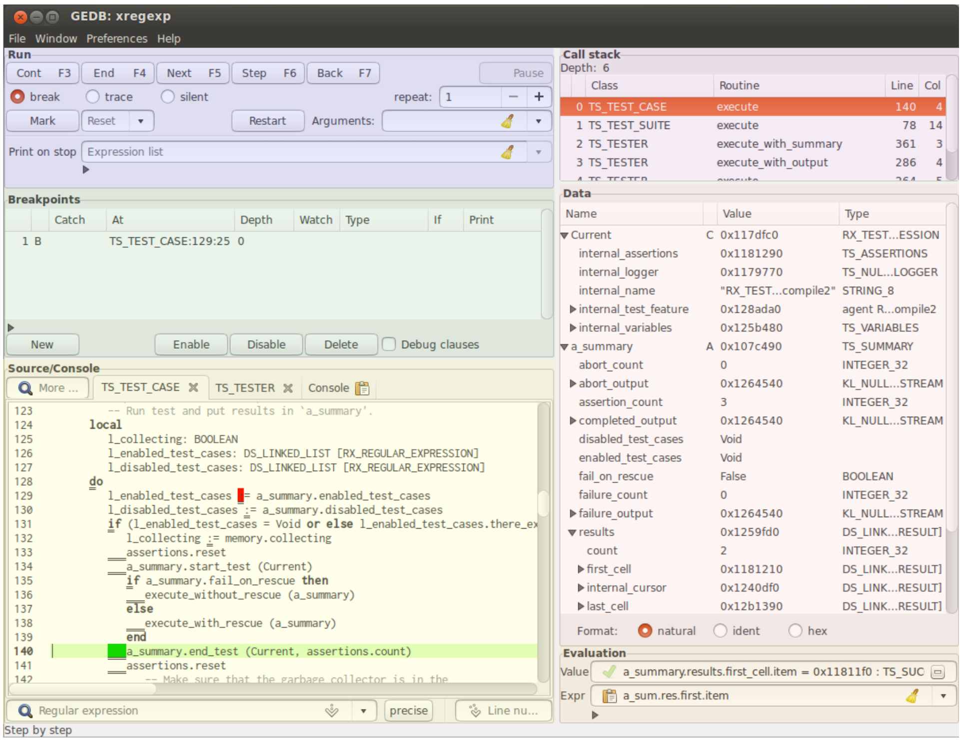

Figure 1: Overview of the main window (here, overlaid by artificial background colors). The *Eiffel* system shown in the screenshots has root class `XREGEXP` from cluster `test/regexp/TESTGEN` of the *GOBO* distribution.

The window contains a menubar on top, a statusbar on bottom and between them the parts named Run, Breakpoints, Source/Console on left half, Call stack, Data, and Evaluation on the right half. There are two more parts: the Source/Console part is actually two parts, Source shown in Fig. 1

and Console not shown, and the SQL select part to be shown separately. More windows can be opened from menu or by specific actions: a poor Source part called Feature window and several additional data windows.

The manual will describe the parts in the in the order from left to right and within each half from top to bottom.

3.2 Specialized fields

Class and type names Parts Breakpoints, Source, and SQL select contain fields to enter a class or type name. This is done via special text entries: a regular expression for the class or type name is to be entered, a combobox opens showing the matching classes or types, one can select from the combobox when this becomes convenient. While class names never include generic parameters a type name may contain actual generic parameters, in this case the opening bracket has to be escaped during input by a backslash since the bracket is a special symbol of regular expressions (the latter meaning takes precedence during input).

Expressions Parts Run, Breakpoints, SQL select, and Evaluation, and the Alias definition window contain special fields for entering expressions. The format of valid expressions is described in Sec. 4, there is also described how the expressions are validated. To most of such fields a restricted history is accompanied storing previously entered expressions. These can be shown and selected from by clicking the button on the field's right end.

If a part contains a field for entering expressions it contains also a so-called Expander widget close to this field (several of those fields in the same part share the Expander), here consisting of two text lines. The first line is generally up to a leading expand/compress icon empty while the second line is hidden (clicking the icon shows and hides the second line). If a validation error occurs then the first line contains an error description and the second line (now made visible) contains the expression in question showing in red the part where the error has been detected. In other words, this is the debugger's way to issue error massages, the heavy way of opening a dialogue window is avoided.

3.3 Menu bar

The menu bar contains the entries File, Window, Preferences, and Help. Menu File has the following items:

Continue the item exists only in the post mortem analyser, it enables to continue the debuggee if it had been interrupted from the keyboard;

Save store the alias name and breakpoint definitions to file *debuggee*`.edg` in the directory where the debuggee is located;

Save as ... store the alias name and breakpoint definitions via a file chooser menu;

Restore from ... restore alias name and breakpoint definitions via a file chooser menu, alias names get merged into the list of existing alias names while breakpoints get added to the breakpoint list (alias name and breakpoint definitions in the standard file *debuggee*.edg in the debuggee's directory are restored after loading, so a particular menu item is not necessary);

Quit quits the debugger session.

Menu Window has the following items to open a new window:

New data opens a new window for displaying data;

Global data opens a window showing global data (constant attributes, values of once functions);

SQL table opens a window for selecting and displaying data analogously to the *SQL* command SELECT;

Alias definition opens a window showing a table of alias definitions, the table entries are editable.

Menu Preferences has the following items:

Appearance opens a window to modify certain properties of the debugger's graphical outlay;

History sizes opens a window to modify the storage sizes of previously entered expressions etc., the storages are associated to various text fields.

Menu Help has the following items:

Manual shows this manual by means of a graphics tool (the choice of the tool depends on the platform);

Colors opens a window explaining the various highlight features of the Source part (colors and underlines);

System opens a window showing some info about the debuggee, e.g. the root class name;

About opens a window showing some info about the debugger itself.

Figure 2: The Run part.

3.4 Run part

The Run part is available only if the debugger is used as runtime analyser (otherwise, the part is hidden). The elements of the part control the stop-and-go behaviour of the debuggee.

When the debugger gets invoked then the expression list (if any) in text field Print on stop will be evaluated and its values will printed on the Console part (see Sec. 3.7). The expressions are evaluated in the routine of the topmost call stack level. If the routine changed after a stop then the expression list disappears (names of local variables in the expressions are now invalid, maybe also query names) and reappears as soon as the routine is entered again with the same target type.

3.4.1 Stop-and-go behaviour

Control switches forth and back between debugger and debuggee. More precisely, the debugger regains control for any of the following reasons:

- just after loading the debuggee;
- when the program reaches the target condition of a dynamic command (see below);
- when a `debug` instruction is reached (and stopping there is enabled, see Sec. 3.5.3);
- when the debuggee is interrupted by pressing the Stop button (same effect as interrupt from keyboard);
- after program end.

When the debugger got the control the debuggee may be continued by means of the buttons on the top of the part. In particular they have the following effect (where n is the value of `repeat`):

Button	Key	Meaning
Cont	`F3`	continue to n-th occurrence of a breakpoint
End	`F4`	continue to the end of the n-th nested compound instruction, in particular leave `loop` clauses
Next	`F5`	continue to the n-th instruction without stepping into called routines
Step	`F6`	do n steps, step also into called routines
Back	`F7`	continue to the selected stack level
Stop	`Pause`	interrupt the debuggee

If some classes have been compiled without debugger information (see Sec. 2.3) then the continuation commands stop at the first possible position for which debugger information is available, in particular, the Step command stops at the first nested routine call for which debugging information has been generated (if any) or simply jumps over routine calls of the class. Nonetheless, even if debugger information is not generated the actual program positions are monitored, so the contents of the Call stack part is correct.

The counter n is determined by the spin button repeat. Moreover the effect of Cont is modified by the radio buttons

break treat breakpoints and tracepoints as defined;

trace treat breakpoints as tracepoints, i.e. do not stop;

silent ignore breakpoints and tracepoints at all.

The effect of button Stop is complementary to that of the other buttons. Quite naturally, also its sensitivity is complementary: if the debuggee has stopped for some reason then Stop is insensitive while the other "dynamic" buttons are sensitive, but if the debuggee is running then Stop is sensitive while the dynamic buttons (and most GUI elements) are insensitive. [3]

Some remarks seem to be in order to explain when the Next and Step commands actually stop. Command Next stops before the next instruction. In particular, it stops at the keywords and special symbols `do`, `once`, `from`, `until` (each iteration), `across`, `if`, `inspect`, `create` (if an instruction), `retry`, `:=`, and `?=`, as well as before a procedure call (qualified or not).

Hint

These program points are those where a breakpoint can be set.

The stop of command Step is more involved. If stopped at an instruction (e.g. as the result of a Next command) then the program continues to the next routine call or instruction; but if it stopped at (not before!) a routine call then the program steps into the routine and stops there at the `do` or `once` keyword (or at the `end` keyword in case of an already initialised once

[3] The button label Stop is suppressed on some platforms in favour of the keyboard accelerator Pause. You may need to read Pause for Stop.

routine) what is considered stopping at an instruction. The two stages of the Step command have been introduced to be able to stop before nested function calls, and to make those stops visible. For example, consider the procedure call `some_proc (some_func (arg))` (where for simplicity `arg` is not a function call) and suppose that a Next command stopped before the call:

```
 some_proc (some_func (arg))
```

(the stop point is indicated in the displayed source code by green background, here by a boxed character). A following Step command continues to the call of `some_func` but does not yet enter the function (that's for visibility):

```
some_proc (some_func (arg))
```

The next Step command jumps into the function, and after return (e.g. by command Back) the program is ready to enter the procedure by one more Step command (again for visibility not yet in the procedure):

```
some_proc (some_func (arg))
```

A similar role as the call instructions play for stepping into their argument expressions do the keywords `create`, `if`, `inspect`, `until` play for the expressions following them.

3.4.2 Mark and reset

If the program stopped at a point where a breakpoint can be set (underlined in the Source part) then the program state (variables on heap and stack, once function values, and program point) may be stored to a file for later restoring during the same debugger session. The stored system state remains valid as long as the current routine is not left to its caller, otherwise, the stored state is silently discarded. Storing and restoring the system state may be used to approximate a critical program point more and more closely.

Clicking button Mark stores the program state and marks the program point to be recoverable. The program point is added to the combobox to its right. A program point in this combobox is recovered when selected. Then also marked program points not in the call stack up to the selected program point are discarded but marked points at the same call stack level remain valid. So, it is possible to jump forward and backward in the same routine call.

Warnings

- Non-Eiffel data, e.g. file contents, are not stored and restored. This may cause the program state to be incorrect after Reset.

- Store and restore make a deep traversal through the objects reachable from the current routine, so these actions may take some time.
- The program state is written to a binary file *debuggee*`.m`*n* in a temporary directory (n is a counter). This file should not be viewed or printed and must not be modified (it is deleted at end of the debugging session).
- Debugger settings such as breakpoint definitions are not stored and restored. So, any breakpoint definitions are preserved across the Reset command.

A special variant of restoring a previous program state is given by button Restart. The program start is marked automatically (but not shown in the Reset list) and restored, as the button name indicates, by clicking Restart. This form of restoring does even a bit more: the string in text field Arguments becomes the new command line arguments. After editing this field its contents does not present the actual arguments as long as the Restart button has not been clicked.

Hint

Pressing the right mouse button in this text field pops up a menu showing how the string is parsed into separate arguments. The menu is for visualization only, clicking an item has no effect.

Restriction

Argument parsing works very well under *Posix* compliant operating systems, otherwise it is rather poor.

3.5 Breakpoints part

The Breakpoints part is available only if the debugger is used as runtime analyser (otherwise, the part is hidden). The part consists of a table showing all defined breakpoints and a row of buttons.

Figure 3: The Breakpoints part.

3.5.1 Breakpoints table

The columns of the table have the following meaning, most of them denote restricting conditions: the breakpoint becomes active when all specified conditions are satisfied (an empty field means `True`).

(no title) a number identifying the breakpoint (no condition);

(no title) one of the values `B` or `T` for a breakpoint or a tracepoint, respectively;

Catch one of the values for the following run time errors to be caught (in decreasing priority):

- `void` call on void target
- `memory` out of memory
- `check` failing check condition
- `failure` routine failure (i.e. leaving `rescue` clause without `retry`)
- `when` inconsistent inspect value
- `signal` signal from the operation system
- `eiffel` other *Eiffel* error, particularly a catcall
- `io` I/O error
- `raise` developer exception (i.e. call of routine `raise`)
- `all` any error

At location in source code composed of class name, line and column number;

Depth minimum depth of call stack to be entered (an empty field is technically not possible, so "`0`" means not set);

Watch address of a memory location that is watched for modification of its contents;

Type type of `Current`;

If a boolean expression to be satisfied;

Print no condition but an expression list to be computed and printed when all conditions are satisfied.

The Type condition may be used to make the expressions in the If and Print columns valid (the `Current` object is the starting point for evaluation). More complex is the interaction of the Catch and At conditions. If both are specified then they work together in the following manner: the condition is satisfied if the rescue clause of the routine enclosing the At location will catch the exception. In any case (At specified or not), the program point shown in the Source and Call stack parts is the point where the exception occurs, not that of a `rescue` clause catching the exception, in particular, not the At location.

3.5.2 Editing

Breakpoints may (and should!) be edited. To this end, clicking the right mouse button on a field in the table starts an editing cycle for this field. According to the column type the "editor" is as follows.

(no name) a combobox with the values `Breakpoint`, `Tracepoint`;

Catch a combobox with the possible values;

At a combobox with values `set from source` and `clear`, in the first case the new value is the cursor position in the Source part (see Sec. 3.6.5 item `Control-b` for details);

Depth a spinbox with the possible values;

Watch a combobox with values `set from data` and `clear`, in the first case the new value is the address of the variable selected in the Data part;

Type a text entry for entering a type name (see Sec. 3.2 for details), there is also the entry `clear` to remove an already present type;

If a text field to be filled manually (lazy evaluation, see Sec. 3.2);

Print a text field to be filled manually (lazy evaluation, see Sec. 3.2).

The editors have been chosen such that most input errors can be avoided. Unfortunately, in case of At and Watch conditions one has to work in two parts of the graphics, first in the Source or Data part to select the interesting location or variable and then in the Breakpoints part to apply it.

Hint

> For convenience, there is also the possibility to set and remove breakpoints directly in the Source part, see Sec. 3.6.5.

Recommendation

> The check for breakpoint matches is very time consuming if there are breakpoints without any of the Catch, At, and Depth conditions. So it is recommended that all breakpoints have one of these conditions set and, for reasons of implementation, that there are no more than 20 breakpoints with the At condition. Unfortunately, this excludes watchpoints.

Mode	Breakpoints	Watchpoints	Time [sec]
`--gedb=none`	N.A.	N.A.	0.62
`--gedb=pma`	N.A.	N.A.	2.21
`--gedb=full`	0	0	10.93
`--gedb=full`	2	0	11.68
`--gedb=full`	0	1	1751.98
estudio	2	N.A	35.42
Hand crafted *C*	N.A.	N.A.	0.58

Table 1: Ackermann 3 11

To give some expression on time consumption the *Ackermann* function has been run on the same computer with the same parameters but various debugging facilities. The results are shown in Table 1, for comparison also the *ISE* compiled program and a hand crafted *C* system has been included. The breakpoints and watchpoints had been set so that they did never match but had to be checked for matching during run.

3.5.3 Create and remove

A new breakpoint without any conditions is created by clicking button New. One or more breakpoints may be selected by singly left clicking (or selected and deselected when also the `Control` key is pressed). The set of selected breakpoints will be enabled, disabled, or deleted when clicking the corresponding buttons. Finally, the Debug clauses check button controls whether debug clauses in the *Eiffel* code are to be treated like breakpoints.

3.6 Source part

The Source part consists of the main text field that is the overlay of several source texts (and the Console part, see Sec. 3.7) and of the bottom fields to enter a search string or a line number. The text field is not editable to preserve the consistence of shown class texts with the ones the debuggee was compiled with (of course, the source texts must also not be edited by an external editor as long as the debugging session runs).

3.6.1 Highlighted text

Within the displayed source code a few small parts may be highlighted (besides syntax highlighting) as follows:

green background actual stop position if the class is that of the actual stack level or belongs otherwise to the call stack (see remark in Sec. 3.6.5);

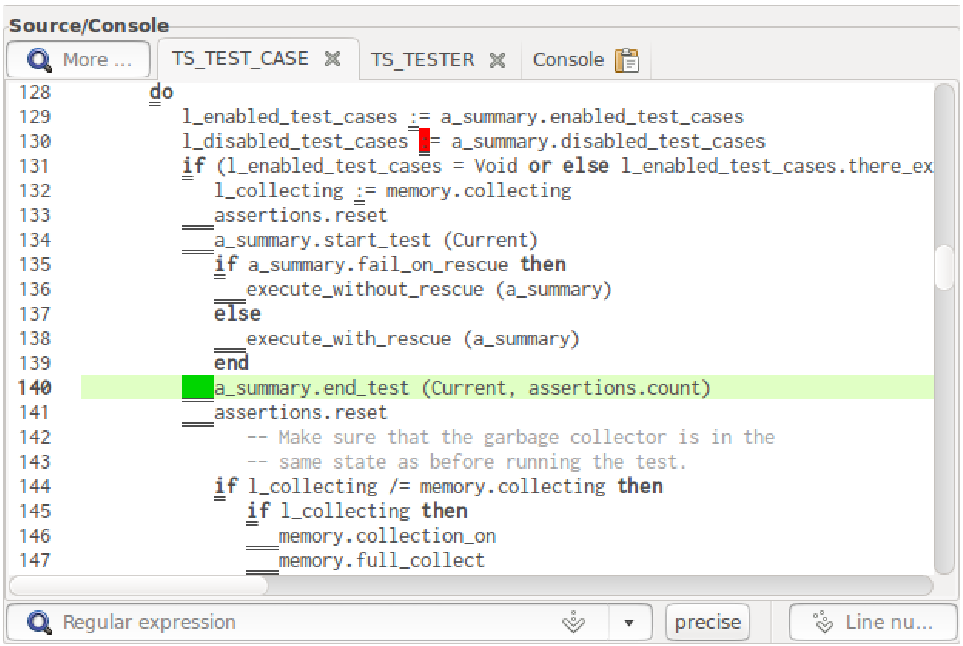

Figure 4: The Source part showing a breakpoint (red) and the actual stop point (green). The underlined characters represent possible positions for breakpoints.

light green background line of actual stop position (to make the stop position better detectable);

red background the position of the At condition of a breakpoint;

underlined possible positions for the At condition of breakpoints (classes not compiled for debugging or routines without generated *C* code don't have those positions);

orange background string found by a search (see Sec. 3.6.4);

yellow background other matching strings (see Sec. 3.6.4);

light blue background name of the feature under the mouse pointer whose definition in source code can be displayed, may be also a class name (see Sec. 3.6.5);

light pink background name of the feature under the mouse pointer that can be displayed and computed, the enclosing routine must belong to the call stack (see Sec. 3.6.5);

light gray background line of the insertion cursor (to make the cursor position better detectable).

The various background colors take the priorities: orange, yellow, red, green, light pink, light blue, light green, light gray.

3.6.2 Class selection

The class is selected (and added to the tag fields bar if not yet there) by

- choosing a tag from the tag fields bar (the tag can also be selected by right clicking somewhere in the tag fields bar: a pop-up menu opens from which the class can be chosen);
- choosing a class from the text entry at the left of the tag field bar (see Sec. 3.2 for details on class name selection);
- choosing a stack level in the Call stack part (see Sec. 3.8), in this case the source code around the actual stop point is displayed;
- the class of the additional Feature window when clicking To source there (see Sec. 3.6.5).

If a class is selected that has been excluded from debugging then this fact will be indicated by striking through the class name in the tag fields bar.

3.6.3 Cursor movement

Within the text field a special location is indicated by the insertion cursor (not to be confused with the mouse pointer): a small bar that sits between two characters of a line (or at the very begin or end of a line). It can be moved by

`left click` the cursor jumps to the position of the mouse pointer;

`arrow` keys move one position in the indicated direction;

`Home, End` keys move to the line start or end;

`Page Up, Page Down` keys move up or down as many lines as are shown in the source text.

`Control-l` select a line directly (see below);

`Control-f` search for a string (see Sec. 3.6.4).

A specific line can be selected by entering a number into the entry on bottom right of the Source part. Entering of the number may be initiated by clicking in the entry to make it sensitive for keyboard input, or by `Control-l` in the text field (in this case the text field remains sensitive); entering is finished by pressing the `Enter` key. The entered number n may be positive the n-th line from start is selected (the one with line number n);

negative the n-th line from the text end is selected;

zero the line of the previous show line action is selected.

3.6.4 Searching

The text field is not writeable but searchable: two methods are provided. First, enter the string (generally: a regular expression) to be searched into the search field at the part's bottom, close input by pressing `Enter`. Repeatedly search of the same pattern continues the search starting after the previously found pattern.

Second, fast processing incremental search can be performed from within the text field. Entering `Control-f` starts a search cycle, then forthcoming entered printable keys are accumulated to a string that is searched immediately. The sequence of key presses may include `BackSpace` to remove the last key from the search string (and search goes back to the previously found pattern), and it may include `Control-f` or `Control-r` to search the same string again. All other keys and moving the mouse pointer outside the text field terminate the search cycle, in particular, `Esc` also clears the highlighted strings. The search direction is forward after extending the search string or pressing `Control-f`, the direction is backward after reducing the search string or pressing `Control-r`. After termination the same string may be searched by initiating another search cycle with two `Control-f` presses. In any case, the search is case insensitive and regular expressions are not supported.

3.6.5 Other actions

The actions of the Source part described so far work completely inside the GUI without interacting the debuggee and these actions are available even when the debuggee is running. By contrast, if not the contrary is not stated explicitly, the following actions are available only if the debuggee is not running.

`Control-.` (dot) If the displayed class is not the base class of a routine in the call stack then the base class of the selected stack routine is selected first. Then in any case the source code around the stop point is displayed.

`Control-b` Set a breakpoint next to the insertion cursor; the breakpoint has only the `At` condition defined, the other parts may be specified later by manipulation in the Breakpoints part.

`Control-k` Delete the breakpoint(s) next to the insertion cursor.

`Control-c` If the routine containing the insertion cursor belongs to the call stack then continue debuggee execution to the point next to the insertion cursor (or to the routine's caller if the point is not hit), i.e. define a temporary breakpoint and issue a Cont command with repeat=1, other breakpoint settings remain active.

`Control-C` Like `Control-c` but ignore all other breakpoints (i.e. run in silent mode).

`single right click` or `Control-x` If the feature name under the mouse pointer is highlighted light blue or light pink then the feature definition is shown in an additional window. This window has two buttons:

Close : close the window

To source : close the window and display its contents in the Source part.

Similarly, if the class name under the mouse pointer is highlighted light blue then the class text is shown in the Source part (no detour through the additional window).
Hint: This action is always available.

`double right click` or `Control-e` If the feature name under the mouse pointer is highlighted light pink then the feature (in any case a query) is inserted in the Evaluation part's Expr field at its insertion cursor where it can be manipulated further.

`triple right click` or `Control-E` If the feature name under the mouse pointer is highlighted light pink then the Evaluation part's Expr field is cleared, the feature is inserted in the field, and the evaluation is done immediately.

The various control characters may be read as "show actual", "Breakpoint", "Kill breakpoint", "Come here", "Come here in any case!", "teXt of", "Evaluate", and "Evaluate now!", respectively.

Some final remarks on the phrases "routine in the call stack" and "next to the insertion cursor". A routine belongs to the call stack if it is the routine of the selected level in the Call stack part (see Sec. 3.8) or the routine of a higher level (for implementation reasons, routines on lower levels are not accessible). If the routine occurs several times then action `Control-.` and background colouring for the actual point select the one closest to the selected level.

The location for the breakpoint is the first underlined text position left or above the insertion cursor, it is at the same time a position where a Next command stops.

3.7 Console part

The Console part collects the *Eiffel* system's I/O. It is available only if the debugger is used as runtime analyser (otherwise, it stays hidden). In case of the post mortem analyser the Console part is meaningless: before crash the debugger graphics is not shown and all I/O is done in the usual console window, after crash further I/O won't happen.

The Console part is a text field of (nearly) black background containing the debuggee's output by `io.output` (white), `io.error` (light pink), and the user input read by `io.input` (light blue). Additionally, the continuation commands (i.e. Cont etc., Mark, Reset) and stop reasons are recorded there (the last includes also the output of Print fields of the Run part or of a matching breakpoint). The colors follow the traffic light metaphor: green for activating the debuggee by Cont etc., red for the stop reason, and, possibly, yellow for recording Mark and Reset commands.

The Console part is integrated into the Source part for practical reasons only. The default size of the Source part is chosen such that as many source code can be shown as is visible in a terminal window. The console window has naturally the same size and to save space of the debugger outlay, the Console part has been made an alternative "source text". Its tag name is Console and it is always located at the right end of the tag fields bar. The different roles of source code and console are distinguishable by their complementary backgrounds: white versus black. Nevertheless, if there is enough space on screen then the Console part may be become a separate window: click on the icon of its tag. Clicking the Close button of that window puts the Console part back into the Source part.

If the Console part is not displayed in a separate window then the following show/hide policy applies. The part is shown (i.e. automatically selected among all sources of the Source part) as soon as the debuggee requests user input, it is hidden when the debugger regains control to show the source text of the actual stop point. Output to `io.output` or `io.error` has no effect on the visibility of the Console part.

Finally, if the Console part is shown within the Source part then it is searchable as any source text, if it shown as a separate window then incremental search is still available (this is possible because this search mode relies only on key presses in the window itself, i.e. interactions with the main window are not involved). In any case, the background colour of the string found is, adapted to the window's dark background, different from that used in the Source part.

3.8 Stack part

The Call stack part simply contains a table of the nested routine calls leading to the actual stop point.

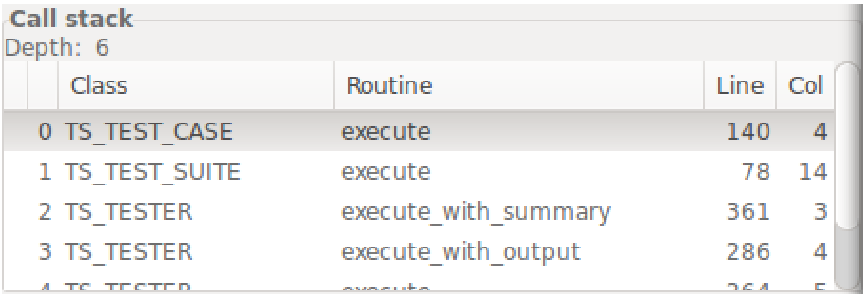

Figure 5: The Call stack part.

The table has the following columns:

(no name) empty or an indicator that the routine has a rescue clause;

(no name) the stack level;

Class name of the class where the level's current routine is defined;

Routine name of the level's current routine;

Line line number of the level's stop point;

Col column number of the level's stop point.

If the class of a level has been excluded from debugging (i.e. there is only very poor information about the class and the routine) then the class and routine names are struck through. This has no effect on actions available and row, and column numbers are correct anyway.

Left click on a table row selects it: the source code of the class is displayed in the Source part and the values of `Current` and of the local variables is displayed in the Data part.

3.9 Data part

The Data part consists of two subparts: a tree displaying objects and their fields and a row of buttons.

3.9.1 Data representation

The tree displays the result of expressions in hierarchical order: fields of an object may be displayed as subtrees. The nodes of the tree contain the name of the object or field, its category (see below), its value, and its type. The category is a single letter meaning

* the `Current` object;

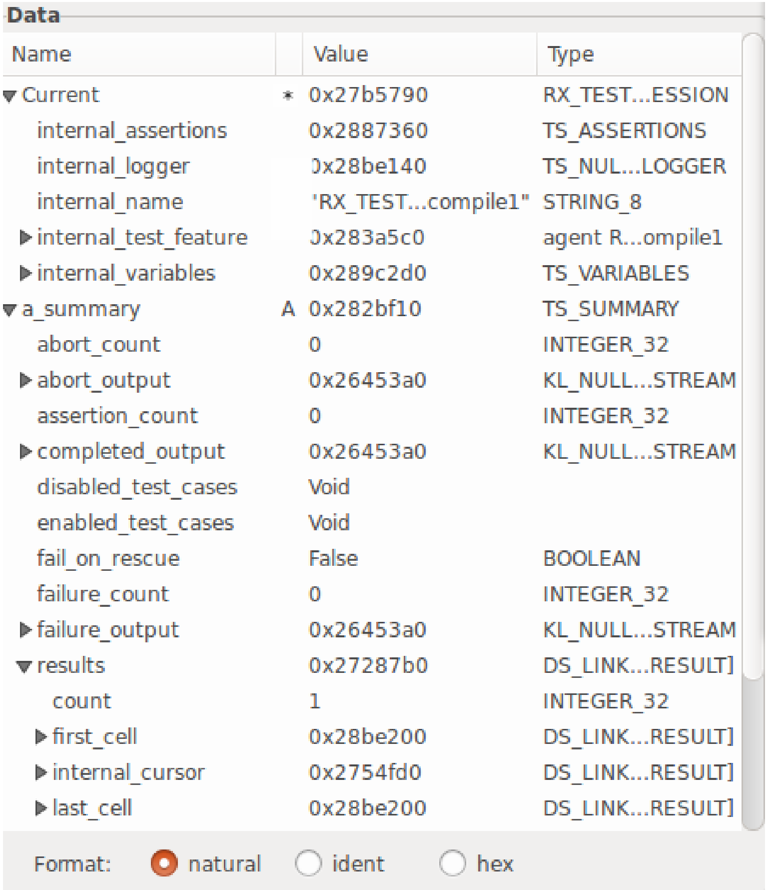

Figure 6: The Data part. The * in the second column denotes the `Current` object and `A` denotes a routine argument, the other contents should be clear.

(empty) an attribute of the node's parent object;

A an argument of the current routine;

L a local variable;

S scope dependent variable: variable of an object test or iterator of an `across` clause;

X the result of an expression evaluation (see Sec. 3.11).

The tree is filled the following ways:

- by the values of `Current` and the local variables when a stack frame has been selected, in particular, after stop of the debuggee;
- additionally by the value of the expression within the Evaluation part;

The format of the data values is controlled by the radio buttons on the bottom of the Data part:

natural the "natural" representation of the value if it is of a basic expanded type or is a `STRING_*`, the address (hexadecimal) of other reference types, empty of other expanded types (the value is given by the values of the object's attributes);

ident objects of a reference type are represented by a unique integer number, the number has a leading underscore to distinguish it from common integer numbers (see Sec. 3.10 how the numbers are obtained);

hex values of basic expanded types are shown by their hexadecimal representations and values of any reference type by their address.

The fields of a value of a `SPECIAL` type get a *C* inspired name: the field index enclosed in brackets. Also the type name is *C* inspired: the field type name and the value's count enclosed in brackets. For practical reasons the arrangement of both parts has been reversed, first the count then the item type name. The reason is that the count at the end would often not be visible in case of a long type name, so it would not be clear that the value is of a `SPECIAL` type.

The fields of a value of a `TUPLE` type get the name of their label if the object itself has been declared as `TUPLE` type with labels, otherwise the names are `item_1` etc.

Values of `agent` objects are more peculiar. They are considered to be of an own anonymous type that conforms to `ROUTINE`, `FUNCTION`, or `PREDICATE`. The fields of such a type are its closed arguments and (in case of a `FUNCTION`) its result value, the field names are the argument names of the agent's routine for the closed arguments or `Result` for agent's result value. Since the type is anonymous, the type name in the table is automatically generated such that it is guaranteed a unique and otherwise invalid type name: it is composed like

`agent` *type*`.`*routine*`(`*args*`)`

where *type* is the name of the agent's base type (enclosed in braces if the target is an open argument), *routine* denotes the feature called (the routine name in case of a call agent, otherwise another automatically generated name), and *args* is a comma separated list of question marks and underscores indicating open and closed arguments, respectively. The *routine* of an inline agent looks like *_integer_routine* where *routine* is the name of the routine declaring the agent and *integer* is simply a counter (observe the leading underscore of the resulting name). The *args* include both the open and closed arguments to show how open and closed arguments are distributed in the resulting feature call, this choice makes also the generated type name unique.

A final remark on the presentation of data whose underlying class is excluded from debugging. If a special treatment of the data is provided as described above (i.e. for basic expanded types, `SPECIAL`s, `TUPLE`s, and agents) then this treatment applies (the associated meta-information is not discarded), otherwise, no attributes are shown. Similarly, if the routine of the selected call stack level belongs to such a class then arguments, local variables, object test variables, and across iterators are not shown, only `Current` is shown (that in turn has no attributes).

3.9.2 Actions

A single row in the data tree (provided its category is an attribute or a local variable) can be selected by left clicking. The associated value may then become the source of a breakpoint's Watch condition or the target of an assignment from the Evaluation part.

A right click on a row (columns Name ... Value) opens a pop-up menu showing first an empty line for the value itself then all queries of its dynamic type. Selecting a menu item inserts it into the Expr field of the Evaluation part. There the query may be treated as any other expression. Arguments of functions are indicated as bullets, they have to be replaced by meaningful expressions before the resulting expression can be computed. In particular, the right operand of infix operators is indicated by a bullet while its left operand is the data clicked, and so is the unique operand of prefix operators.

A right click on a row in column Type opens a pop-up menu showing the typeset of the row's data (i.e. types of source expressions that can be assigned to the data such that forthcoming resolution of dynamic dispatch will work) if assignment is supported and if this set is not trivial (e.g. not for expanded types). The menu is for information display only, selecting an item has no effect.

3.9.3 More data windows

More Data windows can be opened from menu Window → New data, they contain the data tree and additionally a combobox on top to select a stack level and the buttons manually/automatic, Update, Close on bottom. These windows and the Data part on the main window work independently of each other. After stopping the debuggee the main Data part is automatically updated, not the additional Data windows if manually is selected (in case of automatic they are updated like the main Data window). This way, it is possible to have at the same time a look at the actual values and at previous values. An additional Data window is synchronized to the main Data window when its Update button is clicked. As long as the window is not synchronized manipulation of data and stack is not possible, the window simply shows what was shown in the last synchronized state. Moreover, a right mouse click works similarly as in the Data part if the stack level is not below the Data part's stack level (and, of course, if the window is synchronized): the selected query is inserted into the Evaluation part with the necessary leading ^ characters added (see Sec. 4.1 for the meaning of these characters).

A similar structure has the additional window opened from menu Window → Global data. It shows the values of constant attributes (indicated by `C`), once functions (if already computed), and the initialization status of any once routine (indicated by `O`); the latter are struck through if not yet initialized. The data are arranged class by class, and the window's table has an additional column showing the name of the defining class. The table can be searched for items: pressing `Control-f` opens a text field where a string can be entered, then any constant attribute and once routine whose name starts with this string will be highlighted.

3.10 SQL part

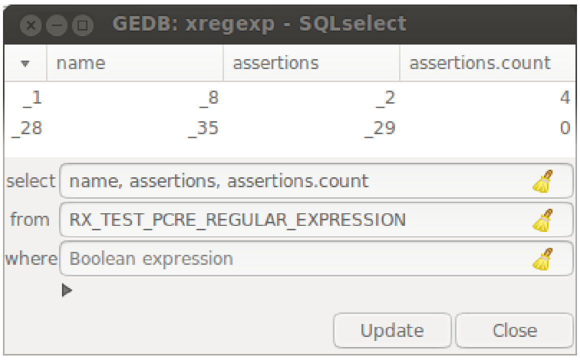

Figure 7: An SQL select part. _1, . . . , _35 are object idents while 4 and 0 in the last column are the values themselves.

Another data window can be opened from menu Window → SQL select. Its purpose is to simulate the `SELECT` command of *SQL*. The debugger maintains for each reference type an *SQL* table (conceptually, not really). The table columns are the type's attributes, and the table rows are all objects of this type that are reachable from the current routine (i.e. from target, arguments, and local variables) by deep traversal. During the traversal consecutive numbers are assigned to each reference object encountered (starting with "1" for `Current`). These numbers (with a leading underscore) are then shown in the window and are at the same time the numbers shown in the main Data window when the ident formatting is chosen. A special meaning have the idents "0" and "?": "0" denotes a void reference and "?" a reference to an object that has no ident since it has been created during the call of a function. Data of basic expanded types are shown in the natural format and data of other expanded types are (recursively) expanded to their fields. The idents remain valid as long as the debuggee is not running and the stack level is not changed. The idents are refreshed after run and stop of the debuggee and when the stack level has been changed (i.e. at the same events when the idents of the Data part are refreshed).

Issuing a `SELCET` command to this table consists of filling one to three text fields in the window and then clicking button Update (a separate update action is necessary to inform the debugger that all three input fields have got their values, the click on a single input field may be misleading; input fields select and where need to be clicked by their own to accept the entered string). The entries are:

select (optional) queries of the table's type as expression list (empty means all attributes);

from (mandatory) a type name as the table name;

where (optional) a boolean expression to be satisfied by the objects of the table (empty means `True`)

(see Sec. 3.2 for entering a type name or expressions). The expressions of any nested level may refer to the matching object by a leading placeholder "?". This is particularly important if a name clash of a from type's query with a local variable occurs: the local variable takes precedence and the query should be written like "`?.`*query*". In any case, the type name should be entered first to ensure that expression validation of the two other entries can be successful (the arrangement of the entries is for tradition's sake). Further, the expression list in the select field is always extended by a leading element (i.e. a leading column in the table shown): the ident of the row's object.

Restrictions

- The expressions in the select field must not contain ranges (precisely, they are silently ignored): the number of matching indices may vary from row to row and no appropriate table width can be obtained in advance.
- The expressions in the select and where fields must not contain object idents: the idents are newly computed and previous settings have become invalid.

The SQL select part allows for two policies to filter objects. The policy is chosen by clicking the precise/conform button:

precise the object type is equal to the one specified in the from field, this must be an alive type;

conform the object type conforms to the one specified in the from field, this may be a not alive (or even a deferred) type, additional attributes of descendant types are not shown and additional queries are not available in the select, where expressions.

Known deficiency

In case that a descendant type renames attributes or other queries then the output does not follow the renaming, the output may look very spurious. The reason is that the meta information generated by the compiler for use of the debugger does not (not yet?) include information about renamings. No such problems occur for the precise type policy.

The SQL select part provides the following actions:

`left click` if the value in the field under the mouse pointer is of a reference type then a data window is opened (composed like the one of the Data part) showing all the qualifiers that lead from a root of the persistence closure to the selected object;

`right click` the value in the field under the mouse pointer is inserted into the Expr field of the Evaluation part (similarly to the Data part).

A final remark. It was not intended to implement a full database, so the debugger's `SELECT` query is rather poor: only one table is supported at one time (no `JOIN`), no grouping (no `GROUP BY`, `HAVING`), and no explicit sorting (no `ORDER BY`, but there is implicit sorting by GUI manipulation).

3.11 Evaluation part

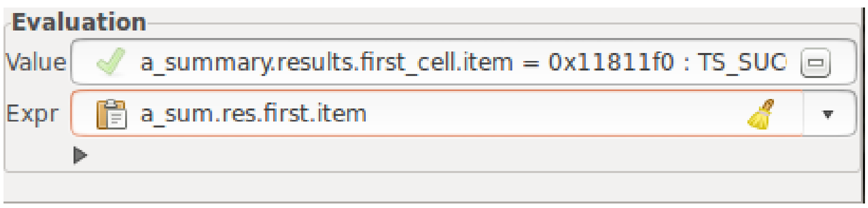

Figure 8: The Evaluation part: an expression list entered in the Expr field and its value in the Value field. Observe that the abbreviated query names entered into the Expr field have been correctly expanded in the Value field.

3.11.1 Evaluation

The Evaluation part consists of two text fields: Expr and Value. Field Expr may be filled

- manually with an expression list;
- by right click on a tree node in the various data parts: a pop-up menu is opened whose entries are the query names of the node, and selecting a query inserts this near the insertion cursor;
- by multiple right click when the mouse pointer is in Source part over a light pink highlighted feature name (in any case a query): the query together with its targets and arguments is inserted near the insertion but not yet computed (double click) or the field is first cleared and then the query is computed immediately (triple click).

The term "near the insertion cursor" needs an explanation. If there are bullets (denoting not yet written function arguments that may come from a data window) right to the insertion cursor then the first one is replaced by the selected query, if there is no bullet to the right then the query is appended to the existing string. Then the cursor moves rightward to the next bullet, if any, or to the string end. This way the natural left-to-right writing of expressions is supported and cursor positioning on bullets to be replaced is rather automatic. The insertion cursor is not visible if the text field has not the input focus. To make the target position of a clicked query visible, the bullet right to the insertion cursor is larger than the others.

The expression is computed when `Enter` is pressed (except for the triple click just mentioned). The value is shown in the non-editable field Value. Precisely, the part of the expression list is shown where the insertion cursor of the Expr field resides (the expression nearest to the left of the cursor up to its leftmost target). This way, values of even very long expression lists can be shown in the one line field. Not shown are expression ranges and their details since a specific index value is not determined, but the rule applies to the other expression details.

Clicking the left icon in the Expr field copies the values of the items of the expression list to the tree of the Data part thereby expanding expression ranges and and their details. There the expressions will indicated by an `X`.

3.11.2 Assignment

Clicking the left icon in the Value field (if sensitive) assigns the shown value to the selected tree item of the Data part. The icon is sensitive only if the item is assignable from the value. In case of numeric basic expanded types: any `INTEGER_*` and `NATURAL_*` values can be assigned to `INTEGER_*`, `NATURAL_*`, or `REAL_*` items, any `REAL_*` values can be assigned to `REAL_*` items; the conversion is done along the lines of the *C* type casting. In all other cases the item is assignable if its type set contains the type of the expression in the Value field (see Sec. 3.9.2 for displaying the typeset of an entity).

3.12 Status bar

The intention of the status bar is to show whether the debuggee is running or not, i.e. whether commands can be issued. The status bar is not exhausted by this purpose, so more information is displayed here depending on the location of the mouse pointer. In particular, the contents of fields of the tables or trees is displayed when the mouse hovers over the entries since there may not be enough space while the status bar is much wider (concerns parts Breakpoints, Call stack, Data, SQL select). For some columns even more information is shown. In case of the Watch column of the Breakpoints part the status bar does not only show the address but also its current contents (that is never shown in the table). In case of the Name column of the Data part the name is expanded by the names of its the qualifiers. Finally, in case of an ident in the Data and SQL select parts the qualified expression (starting at its root in the current routine) for the related object is shown.

The mouse pointer has a similar effect in the Source part. When the mouse hovers over a feature name whose definition can be displayed it is not only highlighted with light blue or light pink background (see Sec. 3.6.1), also the name of its defining class is shown in the status bar if this class is not the actual class. Moreover if the feature can be computed (highlighted light pink) and if the evaluation does not involve function calls then the value is shown additionally.

If the status bar is not occupied by the information just mentioned (this happens always when the mouse pointer is in the Run part) then it displays the status of the debuggee: `Running` if the debuggee is running or else the actual stop reason.

4 Expressions

Eiffel expressions have to be entered in several debugger fields. The expressions can be rather complex and are by far the most complex keyboard inputs required, so they may need a special explanation. The first section describes their general format, the following one describes how to format `SPECIAL` objects or to select entries.

4.1 General objects

The debugger distinguishes three levels of complexity:
expression, *detailed_expression*, and *expression_list*.
They are related to each other by the Backus-Naur-form given in Sec. 4.2.

First, an *expression* is, with a few restrictions and additions, any *Eiffel* expression.

Restriction

`create`, `agent`, `Precursor`, address `$`, and object test expressions are not supported.

An expression is written as follows: a (un-)qualified expression like `x` or `x.y.z`, or the composition of calls to infix and prefix operators where the target is a (un-)qualified expression either. Expressions may include function calls (the arguments are, recursively, *expression*s as well). The first item of a qualified expression (or the expression itself if not qualified) may be one of

- a query of the current class;
- an argument or local variable of the routine at the chosen stack level;
- the predefined entities `Current`, `Result`, `Void`, `True`, `False`;
- a manifest constant of `BOOLEAN`, `INTEGER`, `REAL`, `CHARACTER`, or `STRING` (more precisely, a numerical manifest is first of type `REAL_64` if it contains a decimal point, if not then it is `INTEGER_64` if negative and `NATURAL_64` else, then the manifest is turned into the expected type when used as an routine argument), manifest numericals may not include underscore characters, integers/naturals may be decimal or hexadecimal, reals decimal only;
- a `TUPLE` manifest provided that instances of the type can be created;
- an already initialised once function or a constant definition written as {*CLASS_NAME*}.*feature_name*;
- an alias name (see Sec. 4.4);
- in specific contexts: a question mark `?` or an exclamation mark `!`.

Restriction

Evaluation of functions and creation of objects (here, `STRING` and `TUPLE` objects) uses the corresponding routines in the generated *C* code, similarly evaluation of attributes uses the entries of *C* `struct`s corresponding to *Eiffel* types. Since the *Gec* does a good job in dead code removal, many routines and even some attributes do not occur in the *C* code. If an expression relies on removed routines etc. then they are indicated as "unknown" the same way as routine names etc. not occurring in the *Eiffel* code.

By contrast, arithmetical operations, logical operations, and comparisons of basic expanded types (i.e. the standard operators of these types) are available even if the *Gec* compiler did not generate *C* code for them because the debugger uses its own routines. Thus, the debugger may be used as a pocket computer.

Each following item of a qualified expression has to be a query of the (static or dynamic) type of its left neighbour. In particular, if the left neighbour is of a `TUPLE` type then its fields are `item_1`, `item_2` etc., but the labels are also accepted if the left neighbour's declaration specifies `TUPLE` labels. If the left neighbour is an agent object then the fields are the closed operands and their names are those of the corresponding routine arguments.

In general, the evaluation starts at `Current` of the shown stack level but in case of immediate evaluation (see Sec. 4.5) it is also possible to start at a stack level above. To this end, the first item of a (un-)qualified expression is to be preceded by one or more ˆ characters (or by ˆnˆ meaning n ˆ characters): the evaluation starts n stack levels above the shown one. This way, an expression may combine variables from different stack levels.

4.2 Expression lists

The more sophisticated expressions are composed as follows:

$$
\begin{aligned}
\textit{expression\_list} &::= \{\, \textit{detailed\_expression} \text{ “}\texttt{,}\text{” } \}^{+} \\
\textit{detailed\_expression} &::= \textit{expression}\, [\, \text{“}\texttt{[[}\text{” } \textit{index\_range} \text{ “}\texttt{]]}\text{” }]\, [\, \textit{details}\,] \\
\textit{details} &::= \text{ “\{” } \textit{expression\_list} \text{ “\}”} \\
\textit{index\_range} &::= first \text{ “}\texttt{:}\text{” } \textit{last} \\
&\quad\; \mid first \text{ “}\texttt{\$}\text{” } \textit{count} \\
&\quad\; \mid \text{ “}\texttt{all}\text{”} \\
&\quad\; \mid \text{ “}\texttt{if}\text{” } \textit{expression} \\
\textit{expression} &::= \texttt{--}\ \text{Eiffel expression as described in Sec. 4.1}
\end{aligned}
$$

The *index_range* will be discussed in Sec. 4.3. The expressions within the braces of *details* may start (besides the start item of arbitrary *expression*s) with one or more place-holders `?` referring to the value of the *expression* in front of the braces (if just one `?`) or in front of the p-th nested braces (if p placeholders). In contrast to *expression*s, *detailed_expression*s may not be further referenced.

Whether a simple expression is required on input or an expression list is possible depends on the context, the manual precisely distinguishes between "expression", "detailed_expression", and "expression list".

4.3 `SPECIAL` objects

Single items of a `SPECIAL` object are accessible in a C like notation: appending the index (once again an *expression*) in brackets. The result is an *expression* and may be further referenced.

Multiple array items may be selected by specifying an *index_range* within double brackets as follows:

- two integer *expressions* separated by a colon (say *first* and *last*) select all elements from index *first* to index *last* including the limits;
- two integer *expressions* separated by a dollar sign (say *first* and *count*) select *count* elements starting at index *first*;
- keyword `all`: select all items;
- keyword `if` followed by a boolean *expression*: select the items satisfying this *expression* where the running array item and index are referred to by `?` and `!`, respectively.

Each of these notations define a *detailed_expression* that cannot be further referenced, but *details* in braces may be added. If so then a placeholder `?` in the *details* part refers to the running array element, and the placeholder `!` (precisely, the leading placeholder of a sequence of placeholders) denotes the index itself.

Example

If query `table` is of a `HASH_TABLE` type whose keys are of a reference type then

```
table.keys [[if ? /= Void]] { table.content[!] }
```

selects key and item pairs of all elements of `table`, empty fields in the underlying arrays will be skipped.

4.4 Alias names

To ease the repeated typing of the same complicated expression, alias names for expressions can be defined. To this end, menu Window → Alias definition opens a window containing an editable table in the style of key/value pairs. The table holds a row for each alias name definiton where column Name contains the alias names and column Expression holds the *expression* the row's name stands for. A new alias name is created (i.e. a new row is added to the table) by clicking button New of the window, it starts with bullets as name and expression. An alias definition is discarded (and the row will be removed from the table) by setting its name to the empty string. An alias name is valid if it is a valid *Eiffel* identifier, an expression is valid if it is a *detailed_expression* (i.e. it may contain an *index_range* and *details* but there is merely a syntax check). Alias name definitions may refer to other alias names but cyclic dependencies are not allowed. If an invalid name or expression has been entered then it will be indicated by striking it through. In particular, the bullets of a newly created alias definition are invalid and need to be edited.

Extension

> By contrast to normal expressions, an alias name definition may contain placeholders that point outside the expression. They are resolved when the alias name is used within an expression, i.e. this expression must have sufficiently many nested details in front of the alias name.

An alias definition may be used when composing a *detailed_expression*: the alias name is to be prefixed by an underscore; the leading underscore ensures that the alias name cannot be misunderstood as the feature or variable name. Within the *detailed_expression* to be composed the alias name is replaced by a copy of its associated *detailed_expression*. The resulting *detailed_expression* is in fact an *expression* if the defining *detailed_expression* does not contain an *index_range* or *details*. In this case it may be further referenced or a *range* or *detail* may be added. The expansion of an alias name will be checked for validity in the given context.

Example

(continuation of example in Sec. 4.3): Define the alias name `pairs` as

```
?.keys [[if ? /= Void]] { ??.content[!] }
```

and suppose that variable `table` is of a `HASH_TABLE` type whose keys are of a reference type. Then expression `table { _pairs }` expands to

```
table { ?.keys [[if ? /= Void]] { ??.content[!] } }
```

where the first `?` and the `??` refer to `table` (in front of one or two nested details, respectively, as seen from the placeholders) while the second `?` and the `!` refer to the actual item of `keys` and `content` (range and innermost details specification).

In fact, the rather complex syntax of *detailed_expression* has been motivated by the need for displaying key/value pairs of a hashtable, and to do this repeatedly motivated the introduction of alias names.

4.5 Validation

Expressions and expression lists entered are checked for correctness. Any expression is first checked for syntactic correctness (balancing parentheses etc.) then for consistence of its constituents. Depending of the context this is done in one of two ways. In any case, the context provides a class or type and, possibly, a routine of that class or type.

Lazy compution The type, if given, is only used to provide the class. The first target of a qualified expression may be a query of the given class or (if also a routine is provided) it may be an argument or local variable of the routine. Any forthcoming fields in the qualified expression must be a query of the class of its target (the left neighbour), any polymorphism is ignored.

If the expression includes function calls the number of actual arguments must be the same as the number of formal arguments. The actual argument expressions are checked for correctness, too, and their declared type must conform to the formal arguments' type.

The specification of alphanumeric feature names (not operators!) is relaxed so that only as many characters have to be entered as are necessary to make the name unique within its target class. Moreover, names attributes and local variables take precedence over the predefined entities `Current`, `Result`, `Void`, `True`, `False`. For example, if there is a unique attribute starting with `t` then entering just a `t` means this attribute, not `True`.

Immediate evaluation Type and routine must be provided, additionally the address of an object of this type (the object's dynamic type). Starting with the object as first target the expression is computed and at each stage of evaluation forthcoming fields of a qualified expression must be queries of the dynamic type of its (already computed) target. The other rules are as before with "class" and "declared type" replaced by "type" and "dynamic type", respectively.

The "lazy evaluation" is applied if the expression is specified now but computed later. A typical situation is the Print option of a breakpoint: the expression is entered when the debuggee does not run and it is computed later during running when the breakpoint matches. In this case the address of an object (all the more its dynamic type) for immediate evaluation cannot be provided when the breakpoint is declared. An expression checked for lazy evaluation is checked again when it will be computed (of course, now for "immediate evaluation"). Finally, "immediate evaluation" is applied in the first place for expressions entered in the Expr field of the Evaluation part. Here, the shown stack level determines the routine and the type as the type of `Current`.