Valhalla
The Mjølner BETA Debugger
Tutorial and Reference Manual

Mjølner Informatics Report
MIA 92-12(2.0)
August 1996

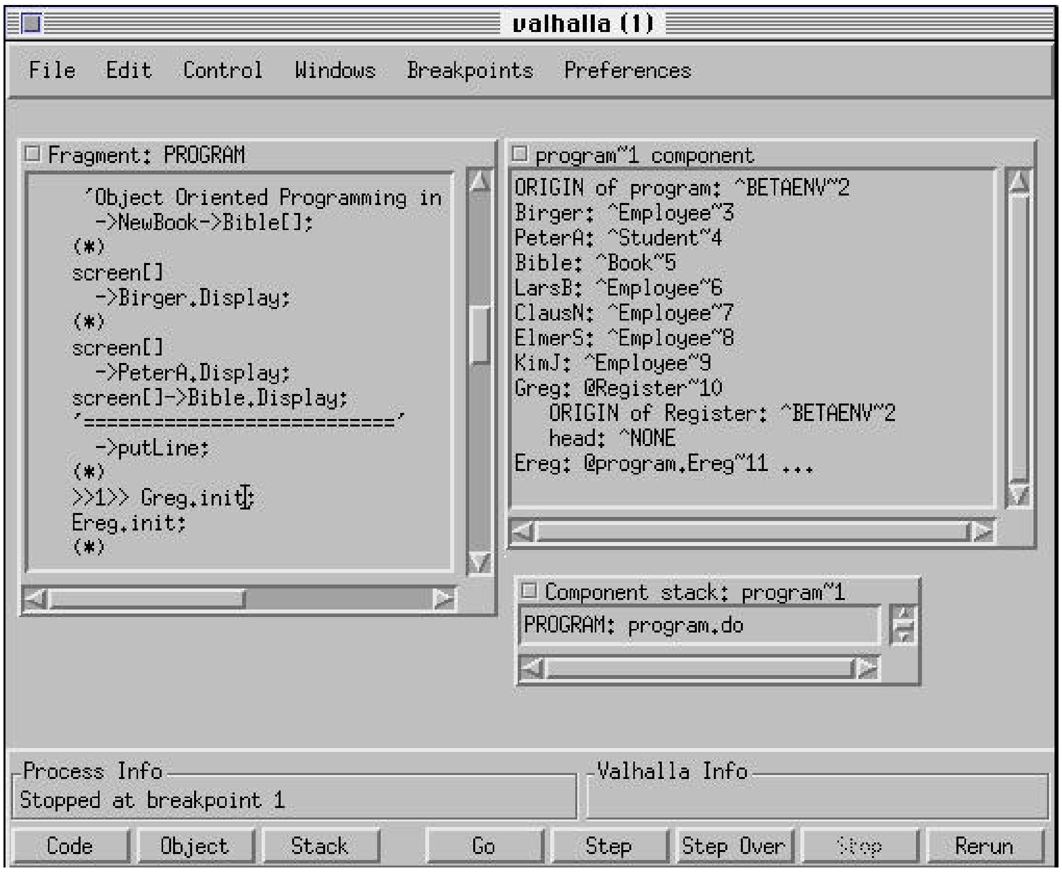
Copyright © 1990-96 Mjølner Informatics ApS.
All rights reserved.
No part of this document may be copied or distributed
without prior written permission from Mjølner Informatics

Contents

Valhalla: The Source Level Debugger

Valhalla[1] is the source level debugger in the Mjølner BETA System. Valhalla offers an object oriented environment for the debugging of BETA programs. Using Valhalla, you are able to control the execution of any BETA program; inspect the state of runtime objects; and trace the execution of the BETA code implementing the functionality of the application. The aim of Valhalla is to help the programmer to locate errors in BETA programs by tracing their execution at the BETA source level.

The user interface is divided in two integrated parts: the source browser (Ymer), and the Valhalla Universe. The source browser enables browsing in the source code of the application being debugged (as well as other source files), and the Valhalla Universe enables the display of runtime objects, execution stacks and source code of the actual execution during the execution of the application.



```
    'Object Oriented Programming in
     ->NewBook->Bible[];
  (*)
  screen[]
     ->Birger.Display;
  (*)
  screen[]
     ->PeterA.Display;
  screen[]->Bible.Display;
  '==========================='
     ->putLine;
  (*)
  >>1>> Greg.init;
  Ereg.init;
  (*)
```

```
ORIGIN of program: ^BETAENV~2
Birger: ^Employee~3
PeterA: ^Student~4
Bible: ^Book~5
LarsB: ^Employee~6
ClausN: ^Employee~7
ElmerS: ^Employee~8
KimJ: ^Employee~9
Greg: @Register~10
   ORIGIN of Register: ^BETAENV~2
   head: ^NONE
Ereg: @program.Ereg~11 ...
```

```
PROGRAM: program.do
```

```
Stopped at breakpoint 1
```

Valhalla Universe

The program execution may be controlled e.g. by setting breakpoints and single stepping at the BETA source level. Runtime errors are caught by Valhalla that will display the offending object and code. From there, program state can be browsed to

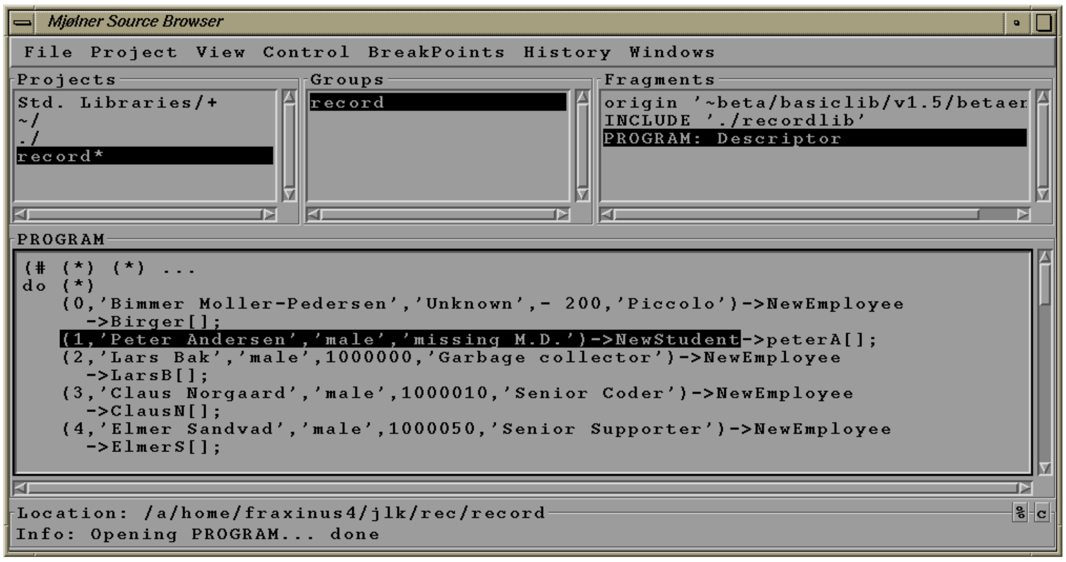
1 Valhalla is the name of the Hall of the Nordic God Odin. Valhalla is the place whereto all the dead warriors are brought when they have fallen as heroes on the battlefield. Odin is the highest ranking God in Asgård.

locate the cause of error. The debugged program executes in a process of its own, being watched by Valhalla, but otherwise unaffected.

Using the Mjølner BETA Debugger you may:

- Control and trace the execution of a BETA program by setting breakpoints and stepping at the level of single BETA source lines, stepping over procedure-calls and even single step at the level of machine code instructions.
- Simultaneously examine the state of any number of objects.
- Examine the execution stack and view code and objects on the stack.
- Examine the program heaps and view objects on the heaps.
- Simultaneously view any number of windows containing source code.

The source browser used by Valhalla is Ymer, and is therefore identical to the source browser used by the other programming tools in the Mjølner BETA System (such as Sif [MIA 90-11], Freja [MIA 93-31] and Frigg [MIA 96-33]), making browsing in the source code of the application of the program being debugged, identical to the browsing facilities during editing, etc. The source browser have been augmented with pull-down and pop-up menus for setting breakpoints and trace points, and for controlling the execution of the application.



```
Std. Libraries/+
~/
./
record*
```

```
record
```

```
origin '~beta/basiclib/v1.5/betaer
INCLUDE './recordlib'
PROGRAM: Descriptor
```

```
(# (*) (*) ...
do (*)
   (0,'Bimmer Moller-Pedersen','Unknown',- 200,'Piccolo')->NewEmployee
     ->Birger[];
   (1,'Peter Andersen','male','missing M.D.')->NewStudent->peterA[];
   (2,'Lars Bak','male',1000000,'Garbage collector')->NewEmployee
     ->LarsB[];
   (3,'Claus Norgaard','male',1000010,'Senior Coder')->NewEmployee
     ->ClausN[];
   (4,'Elmer Sandvad','male',1000050,'Senior Supporter')->NewEmployee
     ->ElmerS[];
```

```
Location: /a/home/fraxinus4/jlk/rec/record
Info: Opening PROGRAM... done
```

Ymer Source Browser

Trace points

In order to make it easy to trace the execution of the application, Valhalla offers trace points. A trace point is a point in the source code, with an associated text string. Each time the execution passes a trace point, the text string is printed on the standard output. The text string is specified as part of the specification of the trace point.

Break points

In order to inspect the state of objects during the execution of the application, Valhalla offers break points. A break point is a point in the source code. Each time the execution passes a break point, control is passed back to Valhalla, enabling you to inspect the state of the execution, the state of runtime objects, etc.

Execution Control

Using Valhalla, you can control the execution of any BETA application. You can start and stop the execution, set break points and trace points, single step at the level of machine code instructions, at the level of BETA imperatives, step over procedure calls, etc.

Runtime Inspection

Valhalla offers extensive support for inspection of the runtime structure of the running application. You can examine the state of objects and runtime stacks. Using the easy-to-use graphical interface, you can navigate through the entire object structure, and locate any object in the application, inspecting its state.

This manual consists of two parts: A tutorial and a reference manual. The reader is assumed to be familiar with the basics of BETA program executions and the fragment system, see for example [MIA 90-2] for a description of the fragment system.

1 Tutorial

The user interface of Valhalla consists of a main window containing a menu and a number of different windows (views) displaying different aspects of the debugged program (called the Valhalla Universe). The *Valhalla Universe* is a top-level window, containing different internal windows for displaying different aspects of the execution state of the debugged process. Below a very short description of the different window types in the Valhalla Universe is given before we go on to describe how to get started using Valhalla.

Window types

- The *code views* display program text from the debugged process.
- The *object views* display the state of BETA objects and components.[2]
- The *stack views* display the runtime stack of the debugged process.

Details on the functionality of the window types follows in the tutorial and in the reference manual following the tutorial.

1.1 Getting Started

The example program

In this tutorial, the program `record` will be used as an example. This program consists of the files

```
~beta/debugger/v2.1/demo/record
~beta/debugger/v2.1/demo/recordlib
```

and uses

```
~beta/containers/v1.5/hashTable
```

The files `record` and `recordlib` are listed in appendix A. To get hands-on experience using Valhalla, you should copy `record` and `recordlib` to a directory of your own, compile them, and then read the tutorial while strolling along on your own workstation:

```
> cd ~beta/debugger/v2.1/demo
> cp record* myDir
> cd myDir
> beta record
```

Starting Valhalla

In general, Valhalla is started simply by typing:

```
valhalla
```

at the UNIX prompt. Valhalla will then open the Valhalla Universe, and you can now use the **New...** menu item in the **File** menu to specify the application, you wish to debug. You can also specify this application directly when invoking Valhalla by giving its name as argument to Valhalla by typing:

2 Components are BETA objects that have their own thread of control.

```
valhalla <application-name>
```

If the application demands some command line arguments, you can specify these directly when invoking Valhalla by typing:

```
valhalla <application-name> <arg1> arg2> …
```

Irrespectively of how the application is specified, you can specify additional command line arguments before the application is invoked, since Valhalla will disply a Command Line Editor before the application is started. From that point, the user controls the execution of the debugged process.

1.2 An Example Usage

This section assumes the `record` program has been compiled as described in the previous section. Running record results in a runtime error:

```
> cd myDir
> record
# BETA execution aborted: Reference is none
# Look at 'record.dump'
```

Now let's use Valhalla to locate the cause of the error. As `record` does not take command line parameters, simply start Valhalla by typing:

```
> valhalla record
```

Valhalla will initialize and open the Valhalla Universe as shown[3] in Figure 1.

Valhalla Universe

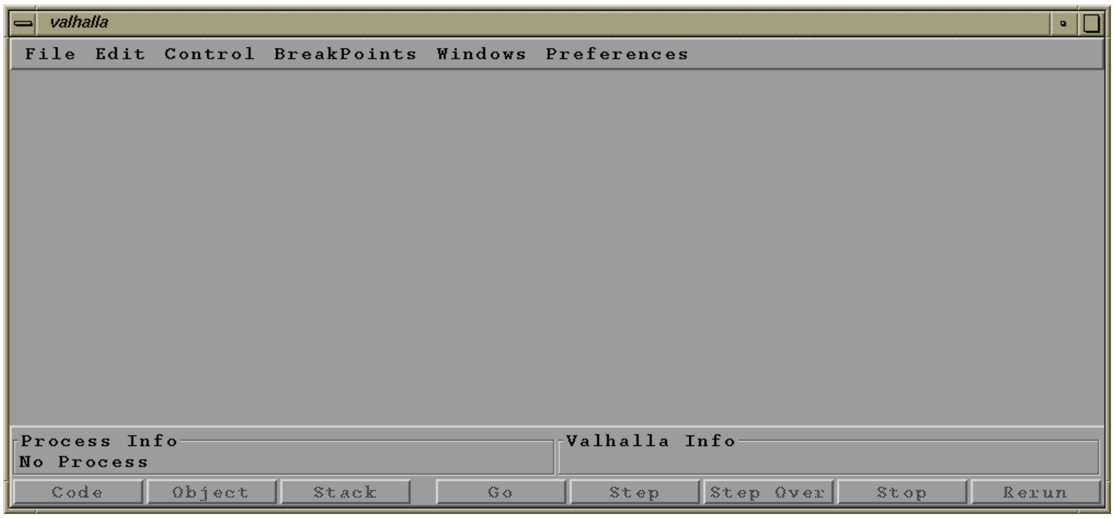


Figure 1: The Valhalla Universe

Immediately after the Valhalla Universe have been opened, the Command Line Editor will be displayed, as seen in Figure 2. Since record does not use any command line arguments, we just press the **Cancel** button (otherwise, we could have entered the command line arguments in the text field, and then pressed the **OK** button).

---

3 Screen dumps shown in the tutorial shows the views as they would have been if you had not copied the example program to a directory of your own, but simply compiled them in the `~beta/debugger/v2.1/demo` directory.

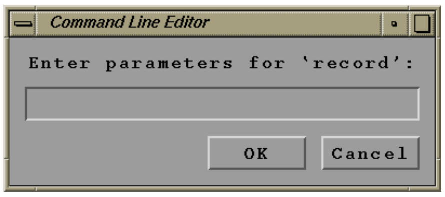


Figure 2: The Command Line editor

Command Line Editor

Contents of the Valhalla Universe

Menus

The Valhalla Universe defines a number of menus. Most commands in these menus operate on the state of the debugged process, or enables control over the debugger process.

The views area

The middle pane of the Valhalla Universe contains a view area in which the different local views will be displayed.

Finally the bottom pane contains three areas: the Process Info Area, the Valhalla Info Area, and the Buttons Area.

The Process Info Area

In the Process Info area, you will see different messages related to the debugged process. In Figure 1, you see the message: No Process, indication that no process is being debugged at this point.

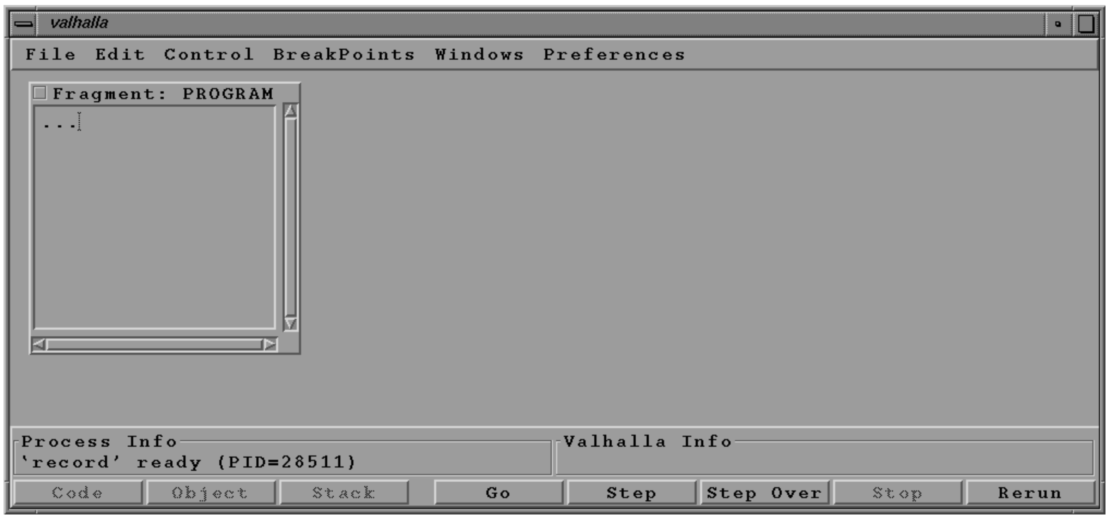
The Valhalla Info Area

In the Valhalla Info Area, you will find different messages related to the operation of Valhalla (status messages, error messages, etc.). In Figure 1, no Valhalla messages are displayed.

The Buttons Area

In the Buttons areas, you find a number of buttons, which are short-cuts to often used commands, also found in the menus of the Valhalla Universe.

After we now have presented the Valhalla Universe, we continue the record example. After having dealt with the Command Line Editor, we will be presented with the following contents in the Universe:



```
...
```

'record' ready (PID=28511)

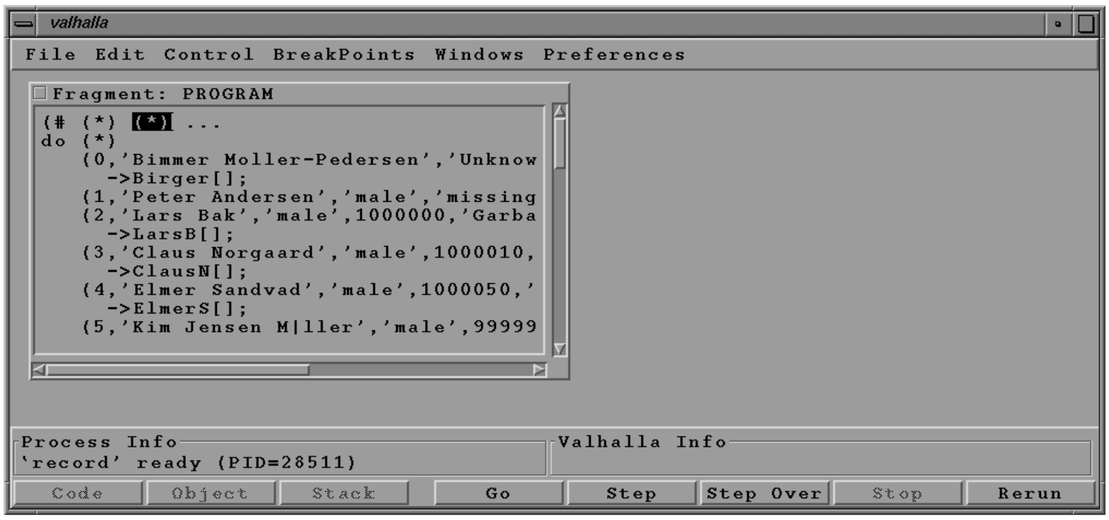
Figure 3: Universe with ready process

Code View

The Universe at this point contains one inner window with the title: Fragment: PROGRAM. This view is a code view, displaying the contents of the PROGRAM slot of the source code of the process, being ready to be debugged. This code view is identical to the code views, you find in the other Mjølner BETA System tools (e.g. Sif [MIA 90-11], Freja [MIA 93-31], and Frigg [96-33]), and you can interact with the code in the window exactly as described in the Sif manual [MIA 90-11], except that only browsing is possible - editing in the source code is not possible through code views in Valhalla. This implies that semantic browsing are available for browsing in the source code, shown in code views.

As you can see, abstract presentation is also available in the code view. After having detailed the **...** in the code view, you will be able to see some of the source code in the code view in the Universe:

Contraction


valhalla
File Edit Control BreakPoints Windows Preferences
Fragment: PROGRAM
Code Object Stack Go Step Step Over Stop Rerun

```
(# (*) (*) ...
do (*)
   (0,'Bimmer Moller-Pedersen','Unknow
     ->Birger[];
   (1,'Peter Andersen','male','missing
   (2,'Lars Bak','male',1000000,'Garba
     ->LarsB[];
   (3,'Claus Norgaard','male',1000010,
     ->ClausN[];
   (4,'Elmer Sandvad','male',1000050,'
     ->ElmerS[];
   (5,'Kim Jensen M|ller','male',99999
```

Process Info
'record' ready (PID=28511)

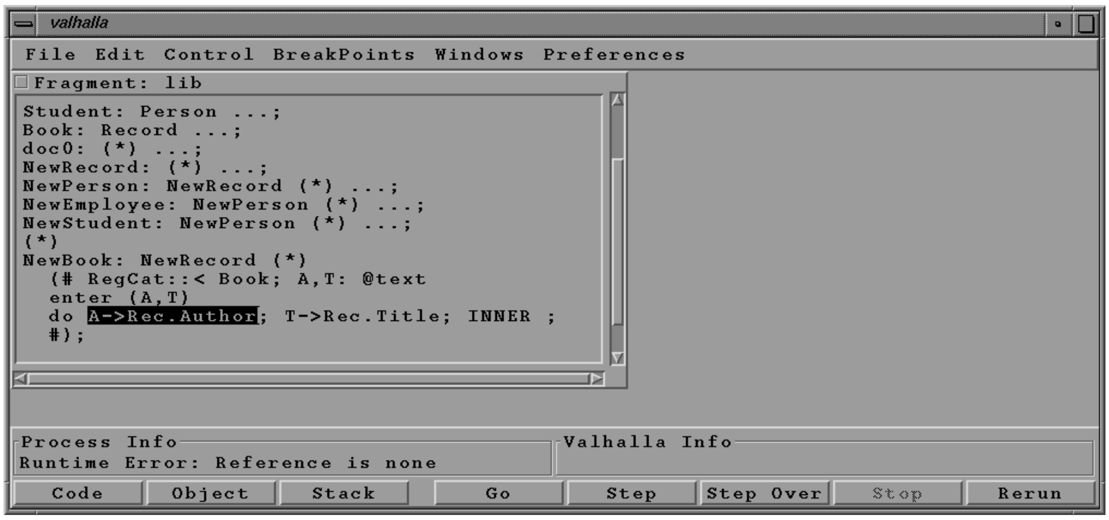
Valhalla Info

Figure 4: Detailing the contractions

Start execution

We are now ready to start execution of the record application. We do this by pressing the **Go** button in the Buttons Area. The application will now begin execution, and since there in this case is a runtime error, the debugged process will not complete. Since the application is being debugged, the application does not terminate as usual, but will signal the error to Valhalla, who will then present the following Universe:

Runtime error caught by Valhalla


valhalla
File Edit Control BreakPoints Windows Preferences
Fragment: lib
Code Object Stack Go Step Step Over Stop Rerun

```
Student: Person ...;
Book: Record ...;
doc0: (*) ...;
NewRecord: (*) ...;
NewPerson: NewRecord (*) ...;
NewEmployee: NewPerson (*) ...;
NewStudent: NewPerson (*) ...;
(*)
NewBook: NewRecord (*)
  (# RegCat::< Book; A,T: @text
  enter (A,T)
  do A->Rec.Author; T->Rec.Title; INNER ;
  #);
```

Process Info
Runtime Error: Reference is none

Valhalla Info

Figure 5: code view opened at time of error

Note that a new code view is now visible in the Universe (in this picture, this new code view is placed entirely on top of the previous code view). This new code view displays the code being executed at the time of the runtime error, and highlights the exact source code that gave rise to the runtime error.

The name of the code view opened is Fragment: `lib` and the pattern in the source code is `newBook`, implying that the newBook pattern is defined in a fragment, called lib. From here, you can now use the semantic browsing facilities described in the Sif manual [MIA 90-11] to find the definitions of the different names in the displayed source code. If the semantic links refer to source code, not in the current code view, new code views are created, displaying the proper source code (similar to Open Separate in Sif). If, during browsing, you forget what imperative caused the error, or cannot find the window containing that imperative, just press the **Code** button, and Valhalla will raise (and wriggle) the window with the offending imperative selected.

Runtime Information

A number of other informations about program state at time of error would be useful in order to decide what caused the error:

Information about program state

· What is the state of these objects.

· What was the call chain at time of error.

In the following sections we will consider how to obtain these informations.

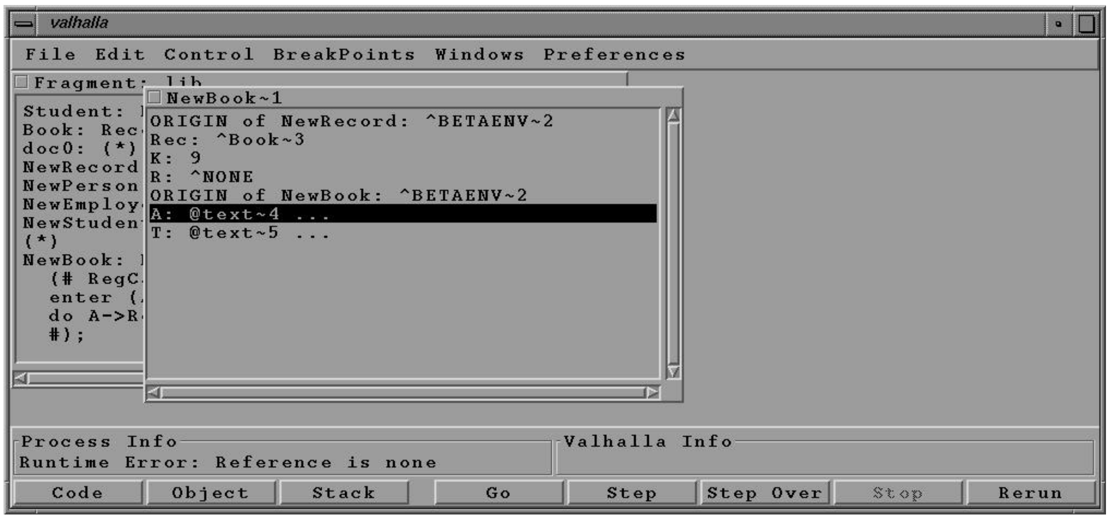
1.2.1 Inspecting object state

Object view

Above we found that Valhalla automatically displayed the source code containing the offending imperative. Usually, in order to determine the cause of the error, one has to consider the state of the objects in the executable to understand what caused the error. To browse the state of the objects, Valhalla implements so-called object views.

The immediately most interesting object, related to the error is the so-called Current Object, which is the object that executed the offending imperative. We can gain access to the current object by pressing the **Object** button in the Buttons Area. This will result in an object view being displayed in the Universe. This object view will display the state of the Current Object at the time of the error.

More generally, whenever the debugged process is stopped, pressing the **Object** button in the Buttons Area of the Valhalla Universe opens an object view displaying the current object if already open in the Universe, the object view will be raised (and wriggled). If we press the **Object** button, we get the following contents of the Universe:



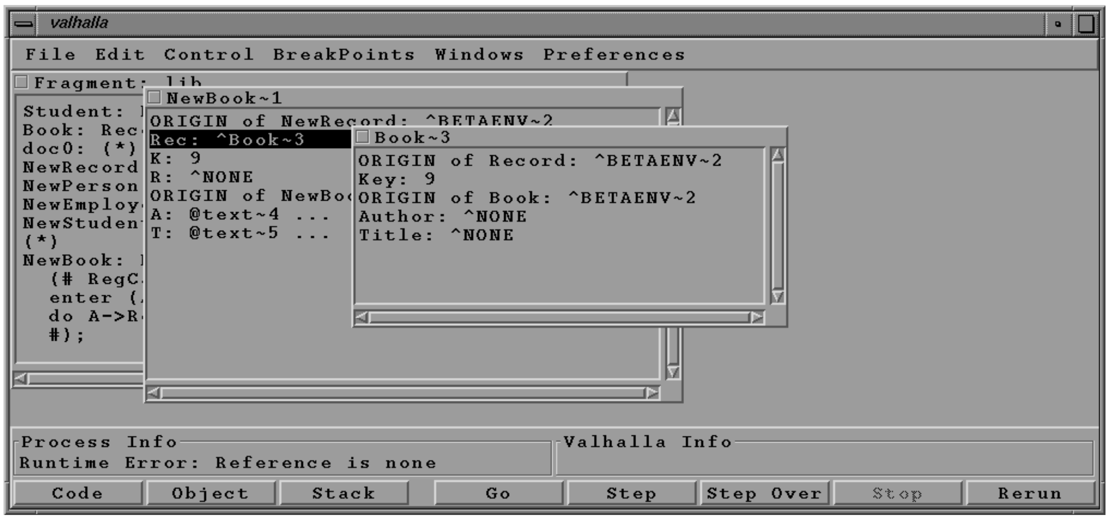
```
Student: 
Book: Rec
doc0: (*)
NewRecord
NewPerson
NewEmploy
NewStuden
(*)
NewBook: 
  (# RegC
  enter (
  do A->R
  #);
```

```
ORIGIN of NewRecord: ^BETAENV~2
Rec: ^Book~3
K: 9
R: ^NONE
ORIGIN of NewBook: ^BETAENV~2
A: @text~4 ...
T: @text~5 ...
```

```
Runtime Error: Reference is none
```

Figure 6: Object view opened at time of error

From the object view in Figure 6, we see that the current object at time of error was an instance of the pattern `NewBook`. The state of the object is displayed in the window. The `Rec` attribute is a reference to an object and is therefore described only by the name of its pattern, followed by ~ and a number. If we double-click the line ***Rec: ^Book~3, a new*** object view is opened, displaying the state fo the object referred to by Rec (as shown in Figure 7):



```
Fragment: lib
Student: 
Book: Rec
doc0: (*)
NewRecord
NewPerson
NewEmploy
NewStuden
(*)
NewBook: 
  (# RegC
  enter (
  do A->R
  #);
```

```
NewBook~1
ORIGIN of NewRecord: ^BETAENV~2
Rec: ^Book~3
K: 9
R: ^NONE
ORIGIN of NewBo
A: @text~4 ...
T: @text~5 ...
```

```
Book~3
ORIGIN of Record: ^BETAENV~2
Key: 9
ORIGIN of Book: ^BETAENV~2
Author: ^NONE
Title: ^NONE
```

```
Process Info
Runtime Error: Reference is none
Valhalla Info
```

Figure 7: Object view displaying the `new` attribute of the failing object

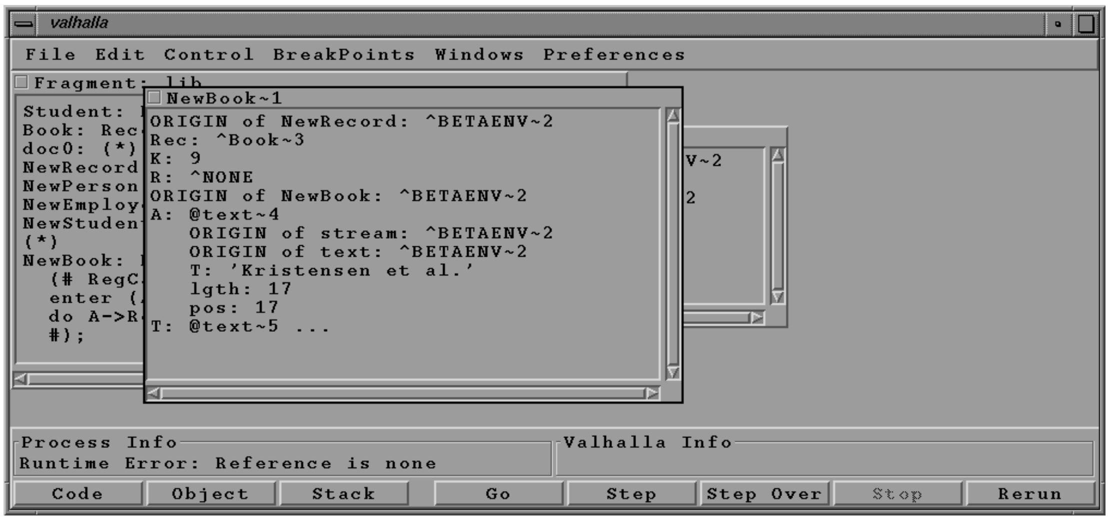
As it can be seen, the `author` attribute of `Rec` is `NONE`. This is actually the reason for the runtime error in record. As you probably remember, the failing imperative was `A->Rec.Author` and because `Rec.author` is `NONE`, this results in a runtime error. If you do not remember, simply press `the` **`Code`** button in the Buttons Area of the Universe.

Since `author` is a dynamic reference to a `Text` object, we could correct the error by changing `author` to become a static instance of `Text` (exchanging `^` with `@`).

Eventhough we have now found the source of error, there is still a number of Valhalla features that have not yet been demonstrated. As a consequence we just continue this tutorial to introduce you to more of the Valhalla functionality.

Detailing static objects

Some attributes are shown contracted (shown by the **...**). These **...** indicate that these objects are complex objects, with inner structure. You can see the inner structure (and state) by double clicking on the attribute. If we do this on e.g. the A attribute, we get the following screen:



```
NewBook~1
ORIGIN of NewRecord: ^BETAENV~2
Rec: ^Book~3
K: 9
R: ^NONE
ORIGIN of NewBook: ^BETAENV~2
A: @text~4
   ORIGIN of stream: ^BETAENV~2
   ORIGIN of text: ^BETAENV~2
   T: 'Kristensen et al.'
   lgth: 17
   pos: 17
T: @text~5 ...
```

```
Process Info
Runtime Error: Reference is none
Valhalla Info
```

Figure 8: Detailing static objects

Static vs. Dynamic Object References

Note, that in contrast to when we followed the Rec attribute in Figure 7, no new object view is opened, but the state of the A attribute is shown inline. This is due to the fact, that A is a static object, whereas Rec is a dynamic object reference.

View updates automatically

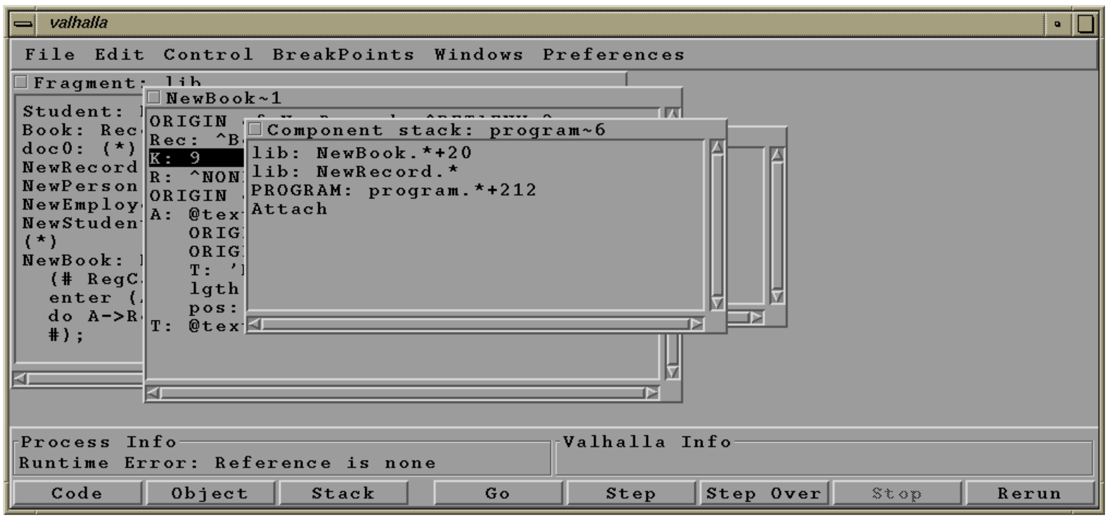
Object views are updated every time the program stops by hitting a breakpoint, receiving a signal or in case of runtime error.

1.2.2 Inspecting the call chain

Now we know where and why the error happened. But how did we get there? To answer this question we use the stack view: We can get a stack view by pressing the **Stack** button in the Button Area.

Stack view

The stack view shows the process stack at time of error. Each line in the stack view refers to a stack frame, with the most recent as the top-most line:



```
lib: NewBook.*+20
lib: NewRecord.*
PROGRAM: program.*+212
Attach
```

```
Runtime Error: Reference is none
```

Figure 9: The stack view

By double-clicking on the lines in the stack view, code views are opened, displaying the code related with this stack frame (with the active imperative selected). By holding the right mouse button down on a line, you get a small menu from where you can create object views, displaying the state of the corresponding stack frame.

1.2.3 Rerunning the program

Rerun

If it during a debugging session becomes necessary to restart the debugged process (e.g. to try to trace the location of the error after having inserted some breakpoints), we can rerun the debugged process by pressing the **Rerun** button in the Buttons Area. The debugged process will then be reinitialized and prepared to start from the beginning again. All existing program state is cleared by the rerun, but existing breakpoints are maintained.

1.2.4 Setting a Breakpoint

Until now, Valhalla has decided when to interrupt the debugged process, doing so because of a “Reference is `NONE`” error. We have not yet seen an example of controlling the program execution in greater detail. To do this we are now going to restart the debugged process and trace the program flow until time of error

We press the **Rerun** button, and Valhalla now restarts the process and makes it ready to be restarted.[4]

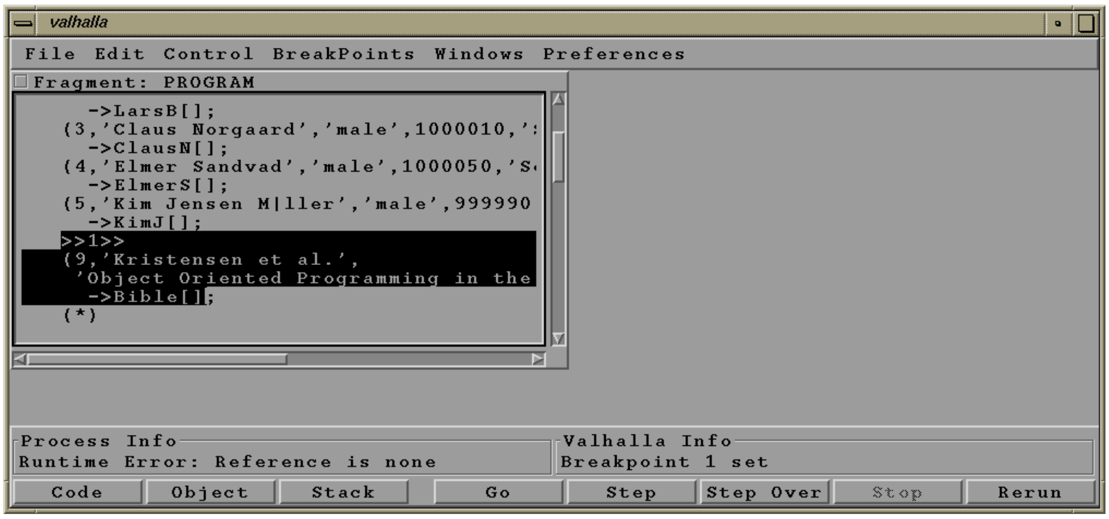
We now examine the code view, locating the invocation of the erronious `NewBook` pattern. We now want to make the process continue execution until it is about to execute `NewBook`. We can do this by setting a breakpoint immediately before the invocation.

Setting a breakpoint

We click at the BETA imperative containing the generation and execution of a *`NewBook`* object. Then select **Set Breakpoint** from the **Breakpoints** menu in the Universe. An

4 All breakpoints set before rerunning the program will continue being set.

breakpoint marker (>>1>>) appears in front of the imperative to mark the presence of a breakpoint (the 1 indicates that this is the first breakpoint). The process will thus be interrupted just before this imperative is about to be executed. Figure 10 displays the look of the code view at this point.



```
      ->LarsB[];
    (3,'Claus Norgaard','male',1000010,':
      ->ClausN[];
    (4,'Elmer Sandvad','male',1000050,'S
      ->ElmerS[];
    (5,'Kim Jensen M|ller','male',999990
      ->KimJ[];
    >>1>>
    (9,'Kristensen et al.',
     'Object Oriented Programming in the
      ->Bible[];
    (*)
```

```
Process Info
Runtime Error: Reference is none
```

```
Valhalla Info
Breakpoint 1 set
```

Figure 10: Your first breakpoint

Go

Having set the breakpoint we make the debugged process begin execution by pressing the **Go** button. The process now runs until the breakpoint is hit. Then Valhalla updates all open code and object views to display the current state of the debugged process.

Step Over

Alternatively, we could have selected the **Step Over** button a number of times. **Step Over** executes the imperatives one by one, returning control to Valhalla after each imperative. This would bring the debugged process to exactly the same imperative, but by executing an imperative at the time in the `PROGRAM` fragment.[5]

Now we would like to continue until the first imperative executed by `NewBook`.

Step

From here we might want to trace the execution more closely. We can do this by using the **Step** button. **Step** is a single step facility, which executes one single BETA imperative at a time. This implies, that **Step Over** executed entire patterns in a single step, since **Step Over** is intuitively single stepping at the imperative level of the visible code in the code view, whereas **Step** will stop execution e.g. immediately after a pattern invocation have been initiated, setting a breakpoint before the first imperative in the invoked pattern.

If we now press the **Step** button repeatedly, we can now follow the execution closely, until we reach the point immediately before the offending imperative.

1.2.5 Browsing code

Source browser

We have above described the source code browsing, based on the semantic linking facilities. However, this is often not sufficient, and Valhalla is therefore integrated with the source browser (Ymer), just as other tools in the Mjølner BETA System.

You can gain access to this source browser by selectiong the **Source Browser** menu enty in the **File** menu of the Universe. This will bring up a standard source browser, with the dependency graph of the debugged program as a socalled root project. We will refer to the Sif manual [MIA 90-11] for more information on the browsing facilities of the Source Browser.

5 **Step Over** sets a breakpoint at the imperative following the current imperative in the current pattern and thus skips procedure calls that might be embedded in the current imperative.

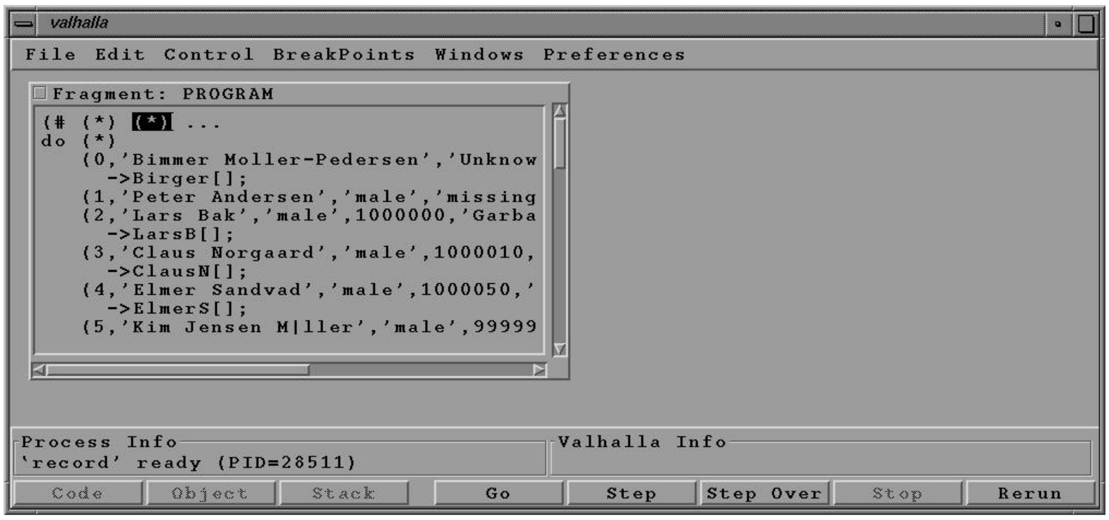
1.2.6 The End

You have now concluded a tour of the most important Valhalla facilities. To get a more detailed description of these facilities as well as others not covered in this tutorial, please consult the reference part of this manual.

2 Reference Manual

When finished initialising, Valhalla opens its Valhalla Universe containing a number of menus. Figure 11 shows the look of the Valhalla Universe when using Valhalla on the example program from the tutorial.

Valhalla Universe



```
(# (*) (*) ...
do (*)
   (0,'Bimmer Moller-Pedersen','Unknow
     ->Birger[];
   (1,'Peter Andersen','male','missing
   (2,'Lars Bak','male',1000000,'Garba
     ->LarsB[];
   (3,'Claus Norgaard','male',1000010,
     ->ClausN[];
   (4,'Elmer Sandvad','male',1000050,'
     ->ElmerS[];
   (5,'Kim Jensen M|ller','male',99999
```

```
'record' ready (PID=28511)
```

Figure 11: The Valhalla Universe

The top pane of the Valhalla Universe contains the menus. The middle pane contains the different views, and the bottom pane contains two info areas and a number of buttons.

Source Browser

Please note, that this reference manual only describes the Valhalla Universe, not the source browser. We refer to the Sif Tutorial and Reference Manual [MIA 90-11] for a reference manual on the Source Browser.
However, it should be noted, that the Source Browser as integrated with Valhalla have been augmented with one global menu, namely the **Breakpoints** menu. The same menu is also available as pop-up menu in all code views, instantiated from the Source Browser. For details on the **Breakpoints** menu, see later.

2.1 Command Line Arguments to the Debugged Process and Command Line Editor

Some applications demand command line arguments in order to execute properly, and others allow command line arguments when being executed. In order to support debugging of such applications, Valhalla have facilities for handling command line arguments to the debugged process.

There are essentially two different ways to specify command line arguments to the debugged process:

- Through the command line arguments to Valhalla
- Through the Command Line Editor

Specifying Command Line Arguments

When invoking Valhalla, it is possible to supply a number of command line arguments to Valhalla itself (see later). Following these Valhalla command line arguments, you can specify the name of the application to be debugged. If any command line arguments are given following the name of the application to be debugged, these are interpreted as command line arguments to the debugged process.

During the initialization process of Valhalla, the user is given yet another chance to specify the command line arguments to the debugged process:

- either as additional command line arguments to the debugged process (if command line arguments were given in the Valhalla command line)
- or edit the command line arguments given in the Valhalla command line
- or specify the command line fully (if no command line arguments were given in the Valhalla command line)

This is done by displaying the Command Line Editor:

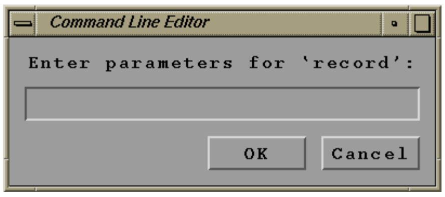


Figure 12: The Command Line Editor

Edit Command Line Arguments

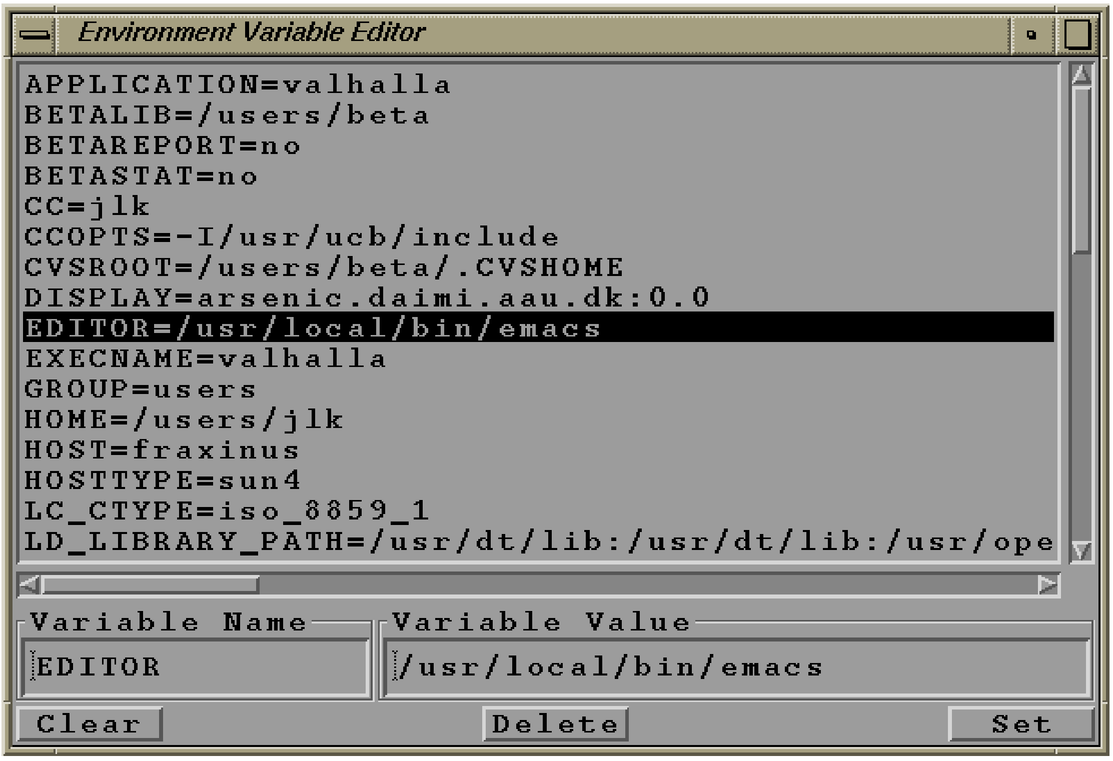
Here you can edit and specify the command line arguments to the debugged process before they are handed to the debugged process.

At any time during the debugging of the application, you can invoke the Command Line Editor to edit the command line arguments (by selecting **Command Line...** in the **Edit** menu). Editing the command line arguments does, however, not directly affect the process being debugged. The changes are only reflected to the debugged process when it is rerun.

2.2 Environment Editor

The debugged process is executed as a separate process, and is executed in an environment that is a copy of the environment, that Valhalla is executing in. However, in some cases, the debugged process demands special values in some of the environment variables, or it may demand some additional environment variables to be set.

In order to operate on the environment of the debugged process, the Environment Editor is available (by selecting **Environment...** in the **Edit** menu):



```
APPLICATION=valhalla
BETALIB=/users/beta
BETAREPORT=no
BETASTAT=no
CC=jlk
CCOPTS=-I/usr/ucb/include
CVSROOT=/users/beta/.CVSHOME
DISPLAY=arsenic.daimi.aau.dk:0.0
EDITOR=/usr/local/bin/emacs
EXECNAME=valhalla
GROUP=users
HOME=/users/jlk
HOST=fraxinus
HOSTTYPE=sun4
LC_CTYPE=iso_8859_1
LD_LIBRARY_PATH=/usr/dt/lib:/usr/dt/lib:/usr/ope
```

Variable Name: EDITOR
Variable Value: /usr/local/bin/emacs

Figure 13: The Environment Editor

Editing Environment Variables

The upper pane is a scrolling list containing a name-value pair for each environment variable in the environment of the debugged process. By selecting one such pair, it is also visible in the lower pane, in the two edit fields. To the left is the name, and to the right the value. By editing the value, you can specify a new value to the environment variables. By pressing the Set button, the new value is defined.

Defining Environment Variables

If you wish to define a new environment variable, you can just specify its name in the left text field, and its value in the right text field, and it will then become defined when the **Set** button is pressed.

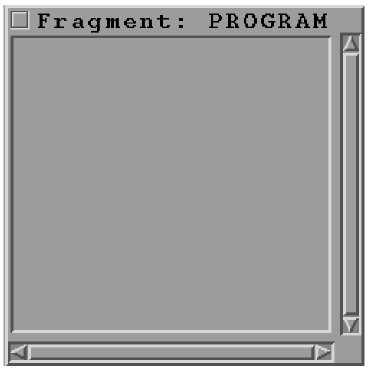
If you wish to remove the definition of an environment variable, you can select the variable from the list and press the **Delete** button.

Pressing the **Clear** button will remove any contents in the two edit fields.

Changes to the environment are not directly reflected in the debugged process, but will be reflected when you rerun the debugged process.

2.3 Interaction with Views

Views are nested inside the Valhalla Universe. Each view has a header consisting of an iconify button and a name field.



View

The different views share many interaction properties, that are described in this section.

Clicking in the name field puts the view on top of all other views. Clicking and dragging the name field moves the view around.

The view may be resized by placing the mouse cursor on the borders of the view (the cursor changes to a cross), and then click-and-drag to resize.

View Menu

Each view has an associated menu, that is popped up by clicking the right mouse-button in the name field of the view. The contents of that menu depends on whether the type of the view. However, the menu contains in all cases **Close** and **Iconify**. **Close** closes the view, and **Iconify** iconifies the view. By selecting **Deiconify** in the menu popped up by clicking the right mouse-button on the icon, the view is displayed in full again.

Closing Multiple Views

Closing Views

By dragging a rectangle in the middle part of Universe, multiple views are selected (all views contained in the selected rectangle). If you then choosing **Close** in the **Edit** menu of the Universe (Shortcut: **Ctrl-W**), the selected views are all closed in a single operation.

View Short-cuts

View Short-cuts

- leftmouse-click on the iconify button in the upper left corner, iconifies the view. This corresponds to selecting **Iconify** in the name field popup menu.
- leftmouse-double-click on the icon, displays the view in full again.
- shift-control-leftmouse-click in the name field or on the icon closes the view.
- rightmouse-press on the name field or on the icon results in the view menu being popped up.
- leftmouse-drag on the name field or on the icon makes it possible to move the view or icon.
- leftmouse-click in the view name field or icon selects the view.
- Rightmouse-press on the individual lines inside the view pops up the item menu (different in the different view types).

View Outline

Outlining a View

If **Outline On Open** is enabled, an outline is shown initially before a new view is opened in the Universe.

While outlining a view, the following actions are possible:

- leftmouse-drag inside the outline makes it possible to move the outline around.

- leftmouse-press outside the outline moves the closest corner of the outline to the position of the mouse (resizing the outline). Further dragging will continue the resize. When the mouse is released, the size and position of the outline is defined, and the new view will be opened in that size, and positioned accordingly.

2.4 Controlling the Execution

When an application is being debugged by Valhalla, it is executed as a separate process, under the control of Valhalla. This gives the user of Valhalla extensive control over the debugged process.

2.4.1 Interrupting the debugged process

Stop

If the debugged process executes, and you wish to interrupt the execution, you can always hit the **Stop** button in the Buttons Area of the Universe (or select the **Stop** item in the **Control** menu of the Universe). This will immediately interrupt the debugged process, making it possible to inspect it (the state of objects, the current stack, etc.).

2.4.2 Setting Breakpoints

Breakpoints are naturally associated with the imperatives of the BETA program, and acts as points in the debugged process, where Valhalla is given some control over the execution of the debugged process. Valhalla supports three types of break points:

Break

Break: If a Break point is reached during the execution of the debugged process, the process will be interrupted, and Valhalla is now able to inspect the current state of the application.

Oneshot

Oneshot: which is a Break point, that is only effective once (will automatically be removed, when it have been reached the first time). Otherwise it functions exactly as a Break point.

Trace

Trace: is used to trace the execution without actually interrupting. A trace point have associated a text string. Each time a trace point is reached during the execution of the debugged process, the associated text will be printed on standard output, and the execution will be continued immediately.

Valhalla supports that break points are positioned in two positions, relative to an imperative:

Before

Before: when a break point is positioned before an imperative, it will be activated immediately before the imperative will be executed.
A Before break is shown visually in the code views by **>>n>>**, where **n** is a sequence number of the break point.

After

After: when a break point is positioned after an imperative, it will be activated immediately after the imperative have been executed. The only places, where using After position is the only possible solution, is when one wish to interrupt the application immediately before it reaches the terminating #) of a descriptor. In all other cases, we could just as well have placed the break point before the imperative following the one selected.
An After break is shown visually in the code views by **<<n<<**, where **n** is a sequence number of the break point.

Automatic Update of Views

An important aspect of Oneshot and Break break points is, that the object- and stack views that are visible in the Valhalla Universe, are automatically updated to reflect the current state of the objects and the stack at the time of the break. Should the object shown in an object view have become inaccessible (and therefore reclaimed by the garbage collector), this fact is reflected in the object view by clearing the object view, and changing the label of the view.

Unsetting Breakpoints

Unset Breakpoints

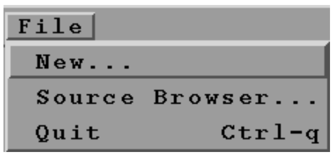
Breakpoints may be unset essentially in the same way as they have been set.

2.5 The Universe Menus

The Valhalla Universe defines a number of menus, which will be described in this section. The buttons in the Buttons Area are merely short-cuts for some of these menu items, and will therefore not be described separately.

2.5.1 File menu

File Menu



New…

This menu item will open a file dialog in which it is possible to select the application to be debugged. You may select both the executable, the source file or the AST file. In all cases, the executable will be loaded by Valhalla that will prepare to debug the application. The application will be forked by Valhalla as a separate process, but execution will not be initiated (before the user asks Valhalla to do so). This is because the user in this way is able to specify breakpoints before initiating the execution. Execution is started by selecting **Go** (see later).

Source Browser…

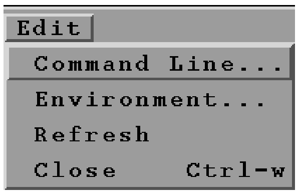
This menu item will result in a source browser being displayed (or, if a source browser is already opened, it will be raised (and wriggled).

Quit Ctrl-q

Selection this menu item will terminate the debugged process, and Valhalla will thereafter be terminated too.

2.5.2 Edit menu

Edit Menu



Command Line…

Selecting this menu item will open the Command Line Editor. Using this Editor, you can edit the command line arguments, that will be handed to the debugged process when it is forked as a separate process by Valhalla. Note, that changes to the command line arguments will not have any effect on the debugged process, if it have been forked. However, by rerunning the debugged process, changes to the command line arguments will be reflected to the debugged process.

Environment…

Selecting this menu item will open a Environment Editor. This editor makes it possible to edit the values of the environment variables (and define new environment variables) for the debugged process. As for command line arguments, these changes will only affect the debugged process after rerunning it.

Refresh

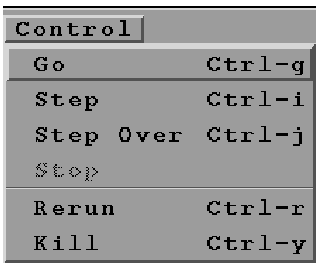
If the Universe for some reason seems corrupted, it might help to refresh the Universe. This will totally redraw the Universe, including the views, and the contents of these views.

Close Ctrl-w

This will close the currently selected view(s).

2.5.3 Control menu

Control Menu



Below the items of the **Control** menu of the Valhalla Universe are described. These are concerned with controlling the execution of the debugged process.

Go Ctrl-g

The debugged process resumes execution until it hits a breakpoint, receives a signal, terminates or a runtime error is detected by the BETA runtime system.

Also available as **Go** button in the Buttons Area.

Step Ctrl-i

The **Step** command makes the debugged process resume execution until it have either executed one single BETA imperative, or it have stepped into some routine. This is the basic step of the debugger (single stepping).

Also available as **Step** button in the Buttons Area.

Step Over Ctrl-j

The **Step Over** command makes the debugged process continue execution after setting a temporary breakpoint at the next BETA imperative in the code currently being executing. **Step Over** is only available if the debugged process is currently stopped in some BETA code (as opposed to being stopped during the execution of code written in C or some other language), as 'next BETA imperative' has no local meaning otherwise..

Also available as **Step Over** button in the Buttons Area.

Stop

Kills the debugged process.

Also available as **Stop** button in the Buttons Area.

Rerun Ctrl-r

Kills the debugged process and restarts the program in a new process. Breakpoints already set continues to be set. Otherwise the state of the debugged process will be as if it just started execution. The command line arguments and the environment variables will be identical to the previous execution, unless they have been edited since last (re)run.

Also available as **Rerun** button in the Buttons Area.

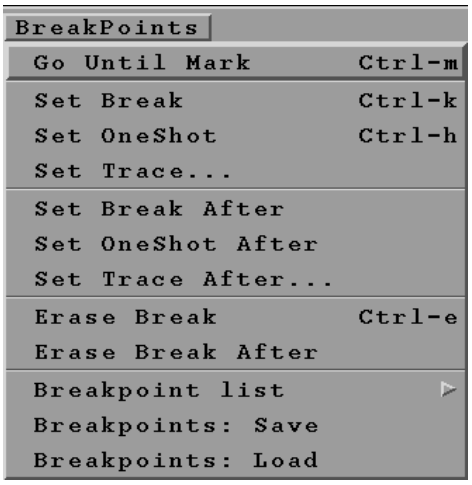
Kill Ctrl-y

Kills the debugged process.

2.5.4 Breakpoints menu

Breakpoints Menu

This menu is the ‘bread and butter’ in controlling the debugged process. It is by means of the entries in this menu, that you can make the debugged process stop at specific places in the code (or output trace information at these points). You can also gain access to all defined breakpoints through this menu, and you can save the breakpoints, to be able to load them into the debugger in a later debugging session. There are three types of breakpoints: Break, Oneshot, and Trace, and breakpoints may be places either before or after an imperative.



Go Until Mark Ctrl-m

If you have selected an imperative in a code view, and select this menu item, then Valhalla will instruct the debugged process to resume execution until the execution reaches the selected imperative.

Set Break Ctrl-k

This will set a break point before the selected imperative in the code view. Each time execution reaches this point, the debugged process will be suspended, and all views open in the Universe will be updated. This implies that the stack view and all open object views will display the current state of the stack (respectively objects) at the points of the break.

Set OneShot Ctrl-h

This will also set a breakpoint before the selected imperative in the code view, but a Oneshot break point will be removed automatically when it is reached the first time (i.e. the debugged process will only be suspended at this point one single time).

Set Trace…

Selecting this menu item will display a small dialog, in which you can specify a text string to be associated with the trace point. Valhalla will then insert a trace point before the selected imperative in the code view. When execution reaches a trace point, the associated text string will be printed, and execution immediately resumed.

Set Break After

Similar to **Set Break**, but will set the break point after the selected imperative in the code view.

Set OneShot After

Similar to **Set OneShot**, but will set the Oneshot point after the selected imperative in the code view.

Set Trace After…

Similar to **Set Trace…**, but will set the Trace point after the selected imperative in the code view.

Erase Break Ctrl-e

If a break point is placed before the selected imperative in the code view, it will be remove.

Erase Break After

If a break point is placed after the selected imperative in the code view, it will be remove.

Breakpoint List

Through this menu item, you can gain access to all break points, that are currently set in the debugged process. These breakpoints are all accessible through the submenu, attached to this menu item. By selecting a given break point from the submenu, the source code with the selected break point will be displayed in a code view.

Breakpoints: Save

This menu item enables you to save the current set of break points onto a file (specified through a file dialog).

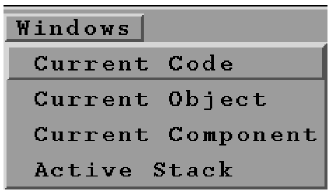
Breakpoints: Load

This menu item enables you to load a previously save set of breakpoints into the debugged process from onto a file (specified through a file dialog). This is only a safe operation, if first of all, it is the same application, and secondly no changes have been made to any of the source files in which any saved breakpoints appear. If one of these two conditions are not satisfied, this operation is not guaranteed to give any sensible results. Currently no check are implemented in Valhalla to test the legality of this **load** operation, and it should therefore be used with care (obeying the above conditions).

2.5.5 Windows menu

Windows Menu

Through this menu, you can open views into the current state of the debugged process, i.e. inspecting state and source code of the currently executing object and component, and inspect the runtime stack of the debugged process.



Current Code

This will open a code view, displaying the source code, that was executed at the point of the break.

Also available as **Code** button in the Buttons Area.

Current Object

This will open an object view, displaying the state of the object that was executing at the point of the break.

Also available as **Object** button in the Buttons Area.

Current Component

This will open a component view into the component, that was executing at the point of the break.

Active Stack

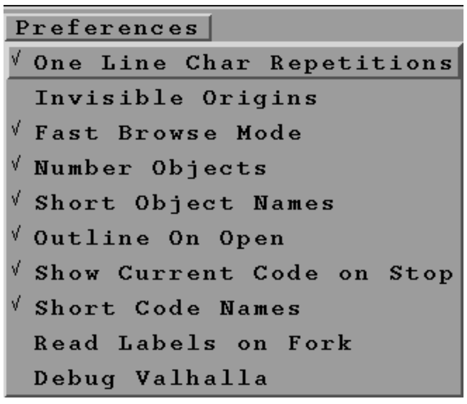
This will display the contents of the execution stack at the point of the break.

Also available as **Stack** button in the Buttons Area.

2.5.6 Preferences menu

Preferences Menu

This menu gives a number of possibilities for setting preferences, defining the behaviour of the different parts of Valhalla. All menu items are toggle items, such that the preference is enabled if a check mark is visible to the left of the corresponding menu item.



One Line Char Repetitions

If enabled, char repetitions will be displayed as a text string. Otherwise, char repetitions will be displayed as other repetitions (i.e. on multiple lines with one index and the corresponding value on each line).

Invisible Origins

If enabled, Origin fields of objects will not be displayed.

Fast Browse Mode

If enabled, following dynamic object references will display the state of the referenced object in the same object view (replacing the existing contents). If disabled, following dynamic object references will create new object views for the referenced object.

Number Objects

If enabled, object references in object views will be specified with both the name of the pattern from which the object is instantiated and a sequence number. The sequence number helps in identifying easily that two object references refer to the same object. If disabled, the sequence numbers are not displayed.

Short Object Names

If enabled, object names in object views will be given in short form, leaving out information on the location of the pattern, from which they are instantiated. If disabled, the object names will contain this information.

Outline on Open

If enabled, opening a view in the Universe will be done interactively by the user by dragging an outline, defining the size and position of the new view. If disabled, Valhalla will define the size and position of the new view.

Show Current Code on Stop

If enabled, Valhalla will automatically display the source code, that was being executed at the time of the break.

Short Code Names

Similar to **Short Object Names**, just for code views.

Read Labels on Fork

If enabled, Valhalla will read the labels in the executable at the time of the fork. Otherwise, reading the labels will be postponed.

Debug Valhalla

Internal debugging facility for Valhalla.

2.6 Browsing through BETA Code

2.6.1 Inspecting the current code

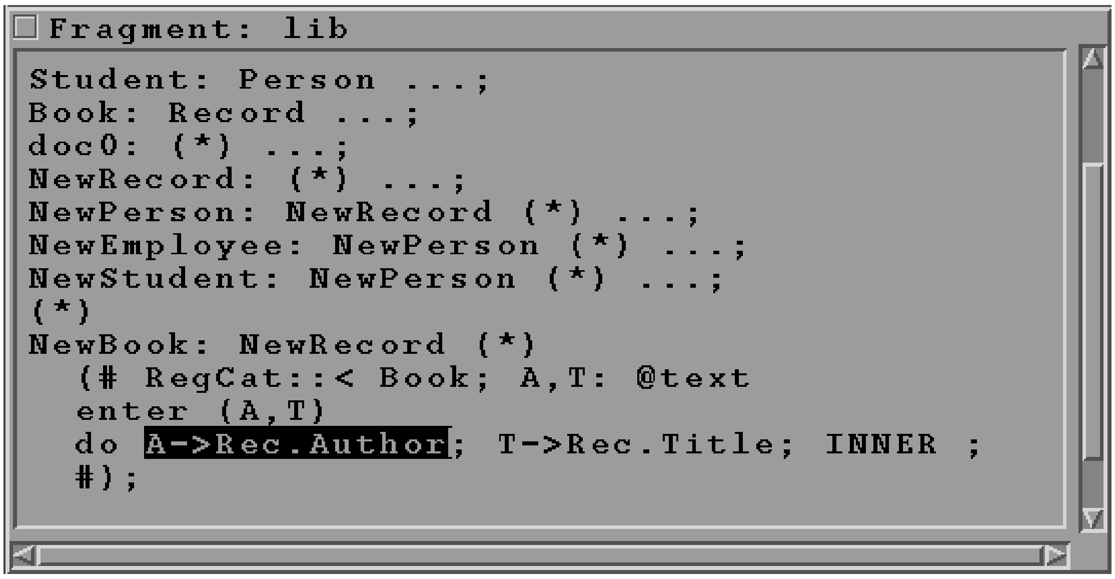
Current Code

When the debugged process is stopped in some BETA code, and no code view is opened on that code, selecting **Current Code** from the **Windows** Menu of the Valhalla Universe opens a code view with the current BETA imperative selected. If a code view on that code is already opened, it is raised (and wriggled), and the current imperative selected.

2.6.2 The code view

Code View

A code view displays an abstract textual description of a BETA object descriptor. Abstract means that nested object descriptors are displayed as three dots. Using the facilities of a code view it is possible to set breakpoints, perform expression searches, or simply inspect the textual description of the pattern.



```
Student: Person ...;
Book: Record ...;
doc0: (*) ...;
NewRecord: (*) ...;
NewPerson: NewRecord (*) ...;
NewEmployee: NewPerson (*) ...;
NewStudent: NewPerson (*) ...;
(*)
NewBook: NewRecord (*)
  (# RegCat::< Book; A,T: @text
  enter (A,T)
  do A->Rec.Author; T->Rec.Title; INNER ;
  #);
```

Figure 14: The Code View

A code view is a standard code viewer as in Sif, Freja, and Frigg, and we will refer to the Sif Reference Manual [MIA 90-11] for more information on the different presentation and browsing facilities of the code view.

Breakpoints Menu in Code Views

Anywhere inside the code view, you can get access to the **Breakpoints** menu, simply by holding the right mouse button down. This will pop-up the **Breakpoints** menu.

2.6.3 Source Browser

Using the **Source Browser...** menu item in the **File** menu of the Valhalla Universe, you can invoke the standard Ymer source browser.

Source Browser

When invoked from Valhalla, the source browser will automatically have a root project defined, containing the entire dependency graph of the debugged program. Using this project, you can browse in all source code, contributing to the debugged process. The source browser is enhanced with facilities for setting breakpoints. This is done by a global **Breakpoints** menu, and by all code views having a **Breakpoints** pop-up menu. These **Breakpoints** menus are identical to the **Breakpoints** menu presented later. These breakpoint menus makes it possible to use the source browser to specify breakpoints anywhere in the source code that is contributing to the debugged process.

We refer to the Source Browser Reference Manual in the Sif manual [MIA 90-11] for further details of the source browser facilities.

2.7 The Object View

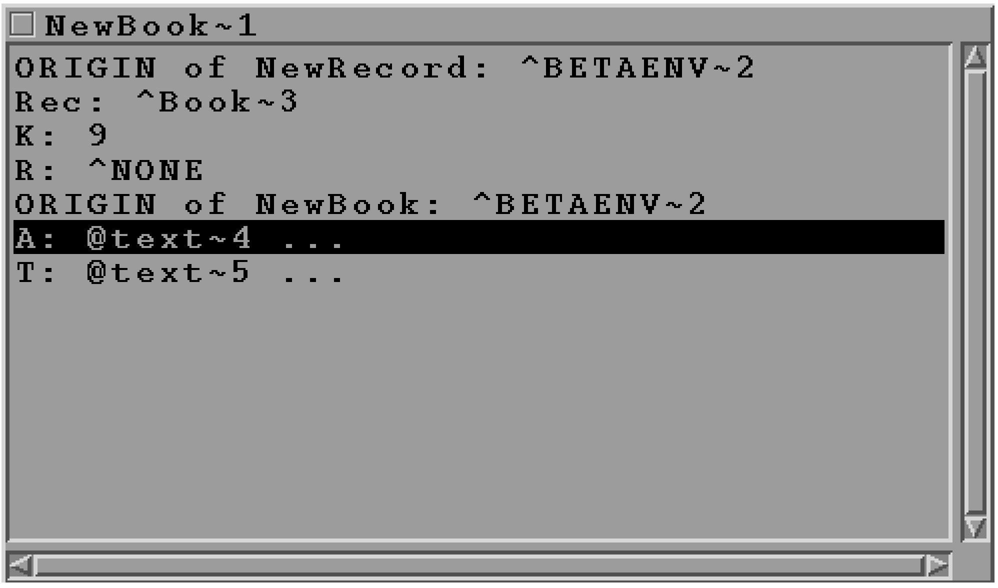

```
NewBook~1
ORIGIN of NewRecord: ^BETAENV~2
Rec: ^Book~3
K: 9
R: ^NONE
ORIGIN of NewBook: ^BETAENV~2
A: @text~4 ...
T: @text~5 ...
```

Figure 15: The Object View

Object View

The state of complex objects is displayed in object views as the result of selecting **Current Object** from the **Windows** menu, double-clicking an object reference in a stack view or double-clicking a line in another object view.

The name of the pattern of which the object is an instance is displayed as the name of the object view window. The pattern name is prefixed by the origin chain from the pattern to the fragment in which the pattern is declared (if **Long Object Names** is enabled). For example, an object view having the name **`Object: lib.x.y`** displays an instance of the pattern `y`, declared nested in the pattern `x` contained in the fragment `lib`.

Each line in an object view corresponds to either a static or dynamic reference contained in the object. Chars, Booleans, Integers and Reals are displayed directly by value whereas attributes of more complex type are described by their pattern name.

Browsing objects:

Browsing Objects

The object browser uses abstract presentation of the objects presented. This means that nested part objects are initially shown contracted, i.e. as three dots. By double-clicking a line of the object view ending in '...', the hidden details will be shown. By double-clicking the same line again, the details are hidden.

Each line in the object view corresponds to some attribute of the object. Simple attributes (@Char, @Integer, ...) cannot be further detailed, whereas other kinds of attributes can. Default when double-clicking some attribute is as follows:

Dynamic References

1. *Dynamic references*: A new object view is opened on the referred object. If some object view on that object is already open, visual feedback (wriggling and highlighting) will signal this fact. The same behaviour goes for origin references as well.

Static References

2. *Static references*: If the attribute is contracted, double-clicking shows an extra level of detail in the same window. Otherwise the attribute is contracted.

3. *Repetition references*: If the repetition is contracted, double-clicking unfolds the repetition by showing a line for each index in the repetition. Otherwise the repetition is contracted. **Repetition References**

4. *Pattern references*: Currently a more detailed view on pattern references is not implemented. **Pattern References**

Object View menus:

In addition to the default menu items for all views (described above), an object view has a number of additional items in the name field popup menu, i.e. the menu popped up by clicking the right mouse-button in the name field of an object view. These are described in turn below:

Object View Menu

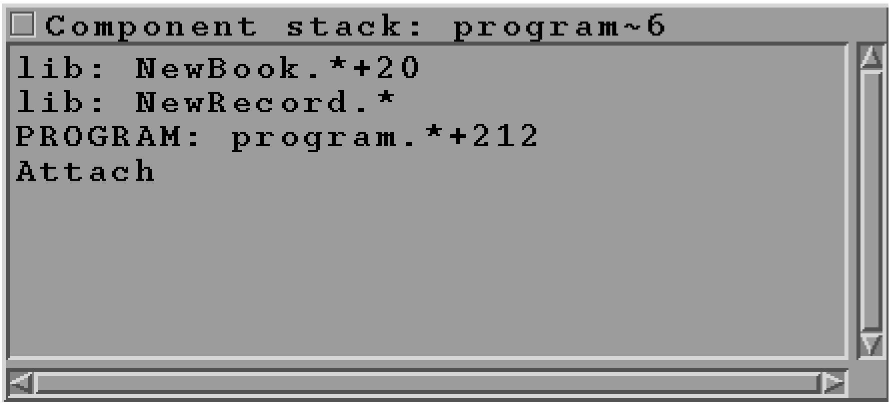
1. **Fit to contents**: Tries to resize the object view to a reasonable size.
2. **Show attribute**: This nested menu is a list of object attributes that are not currently visible. By default this list includes the origin attributes. By selecting an entry in this menu, the corresponding attribute is made visible. It may later be re-hidden as explained below.
3. **Fast Browse Mode**: Default when double-clicking dynamic references is to open a new object view showing the object referred. By selecting "**Fast Browse Mode**", the default is changed into showing the object referred in the same window, replacing the previous contents. This allows for fast browsing without opening unnecessary object views. Default may be reset by choosing "**Fast Browse Mode**" again.
4. **Go Back**: The browser maintains a stack of objects visited during Fast Browse Mode. By selecting "**Go Back**", the previous object on that stack in reshown in the current window.

When clicking the right mouse-button inside an object view, a sligthly different menu than the name field menu pops up. The entries of this menu, the "attribute menu" are as follows:

Attribute Menu

1. **Open Separate**: Default when double-clicking static references is to detail the corresponding attribute in the current window. Alternatively one may single-click the static attribute and then select "**Open Separate**". This opens a new object view on the part object.
2. **Open inline**: If a static reference attribute is double-clicked in order to detail the view, and another object view on that part object is already open, the existing object view is highlighted, and the attribute *not* detailed. If one wants to detail in the current window anyway, one may either close the existing object view, or single-click to select the attribute and then choose "**Open inline**" from the object state popup menu.
3. **Contract attribute**: Has the same effect as double-clicking a complex attribute that is already detailed. I.e., the attribute is contracted.
4. **Hide Attribute**: By single-clicking an attribute in the object view and then selecting "**Hide Attribute**", the attribute is hidden. Note that "contracted" and "hidden" is not the same thing. Contraction replaces a complex attribute by a single line ending with '...'. Hiding an attribute means moving it completely out of sight.
5. **Show Attribute**: This entry corresponds to the menu entry with the same name in the name field popup menu. However, instead of showing a list of hidden attributes for the main object, a list of attributes hidden in the currently selected complex attribute is shown. I.e., to show a hidden attribute of a nested part object, single-click that part-object and then choose "**Show Attribute**" from the object state popup menu.

2.8 The Stack View



```
Component stack: program~6
lib: NewBook.*+20
lib: NewRecord.*
PROGRAM: program.*+212
Attach
```

Figure 16: The Stack View

Stack View

A stack view is opened by choosing **Current Stack** from the **Display** menu of the Valhalla Universe. The stack view shows the current runtime stack of the debugged process.

- BETA stack frames displayed as **frahmentName: PatternName**. Double-clicking on a BETA stack frame opens a code view showing the code referred.
- C stack frames. If an external procedure calls back to BETA, the part of the runtime stack used between the external call from BETA to C and the callback from C to BETA is shown abstractly as **[C Stack Frame]**.

Stack Frame Object Menu

By double-clicking on the BETA stack frame lines, a code view is opened, displaying the specific code of this stack frame (with the executed imperative being selected). By right mouse button hold, you gain access to the stack frame object menu. By selecting the entry, you gain access to the object executing the stack frame.

2.9 Valhalla Command Line Options

Valhalla accepts the following command line options:

-**EXEC** executable
-**EXECNAME** executable
executable is the name of the BETA program executable to be executed and debugged.

-**ENV** name value
-**ENVIRONMENT** name value
Used to set environment variables for the debugged process. Default is to hand over the environment. However, using the ENVIRONMENT command-line parameter, the default values may be overridden.

-**BOPT** name value
-**BOOLEANOPTION** name value
Used to set the boolean option named name. Value should be TRUE or FALSE. See Section BOOLEAN OPTIONS later for details on available boolean options.

-CL parameter
-COMMLINE parameter
Used to give command line parameters to the process being debugged One COMMLINE is needed for each command line parameter to be given to the debugged process.

-PID number
Used to inform valhalla that it is supposed to debug the process with process id number. Used if valhalla is started after the debugged process.

BOOLEAN OPTIONS

The following boolean options can be set (or unset) in the above mentioned BOOLEANOPTION option. All boolean option names below are non-case sensible.

ShortCodeNames
Whether to use truncated imperative names for code expressions shown on the stack.
Default: TRUE.

ReadLabelsOnStartUp
Whether to read labels from executable at startup instead of on demand.
Default: FALSE.

DebugValhalla
Print out Valhalla debugging information.
Default: FALSE.

AlwaysCurCode
Show current code on each process stop, even if no view is currently open on that code.
Default: TRUE.

CharRepOnALine
Whether char repetitions should be shown on a single line.
Default: FALSE.

InvisibleOrigins
Whether origins should initially be invisible in object views.
Default: TRUE.

NumberObjects
Whether objects should be given a serial number.
Default: TRUE.

ShortObjectTitles
Whether to use truncated pattern names (max 2 pattern names in path) for object descriptions.
Default: TRUE.

DragOutlineOnOpen
Whether a nested window is dragged on open, or is simply given a default size and position.
Default: TRUE.

2.10 Valhalla Environment Variables

Valhalla recognises the following environment variable:

VALHALLAOPTS
May contain command line options to Valhalla as defined above. Command line

options given on the command line are given priority over command line options specified in the **VALHALLAOPTS** environment variable.

2.11 Other Issues

This section contains a number of issues not yet touched upon.

2.11.1 The BETART environment variable

The BETART environment variable is described in [MIA 90-4]. It may be used to control the behavior of a BETA program in several ways. To set BETART for a BETA program being debugged without affecting Valhalla[6], set the BETART environment variable either through the Valhalla command line option **ENVIRONMENT**, or using the Environment Editor.

2.11.2 Tracing garbage collections

Trace Garbage Collection

If you wish to trace the garbage collection in the debugged process, you can do this by setting the BETART environment variable for the debugged process. The Compiler Manual [MIA 90-4] defines the possible garbage collection traces.

2.11.3 Known bugs and inconveniences

- In some cases (especially on SUN 4) the stack view does not display the full runtime stack, forgetting a number of entries. Likewise, the stack view may get confused if the debugged process is stopped while executing C code. This also has implications for Valhalla when deciding where a runtime error occurred, and might result in the wrong place being pointed out.
- When a complex attribute has been detailed, the scrolllist implementing the view may enter an "auto-scroll" mode. Click some attribute to switch off this mode.

6 Valhalla is a BETA program itself.

Bibliography

[MIA 90-2]	Mjølner Informatics: *The Mjølner BETA System: BETA Compiler Reference Manual* Mjølner Informatics Report MIA 90-2.
[MIA 90-4]	Mjølner Informatics: *The Mjølner BETA System: Using BETA on UNIX Systems*, Mjølner Informatics Report MIA 90-4.
[MIA 90-11]	Mjølner Informatics*: Sif - Mjølner BETA Source Browser and Editor - Tutorial and Reference Manual*, Mjølner Informatics Report MIA 90-11
[MIA 93-31]	Mjølner Informatics: *Freja - An Object-Oriented CASE Tool - Tutorial and Reference Manual*, Mjølner Informatics Report MIA 93-31.
[MIA 96-33]	Mjølner Informatics: *Frigg - User Interface Builder - Tutorial and Reference Manual*, Mjølner Informatics Report MIA 96-33.

Appendix A

This appendix contains the source code for the BETA program used as example in the tutorial. The source consists of the files `record.bet` and `recordlib.bet`.

```
ORIGIN '~beta/basiclib/v1.5/betaenv';
(* record.bet
 *
 * COPYRIGHT
 *                     Copyright Mjølner Informatics, 1989 - 1994
 *                     All rights reserved.
 *)
INCLUDE 'recordlib'
--- PROGRAM:descriptor ----
(#      Birger, PeterA, Bible, LarsP, ClausN, ElmerS, KimJ:  ^Record;
 Greg: @Register;
 Ereg: @Register
 (# element::< Employee;
    Display::< (#do '/Employee '->putText; INNER #);
 #);
do      (0 ,'Bimmer Moller-Pedersen','Unknown',- 200,'Piccolo')
        -> &NewEmployee->Birger[];
 (1,'Peter Andersen','male','missing M.D.')
        -> &NewStudent->peterA[];
 (2,'Lars Bak','male',1000000,'Garbage collector')
        -> &NewEmployee->LarsP[];
 (3,'Claus Norgaard','male',1000010,'Senior Coder')
        -> &NewEmployee->ClausN[];
 (4,'Elmer Sandvad','male',1000050,'Senior Supporter')
        -> &NewEmployee->ElmerS[];
 (5,'Kim Jensen M|ller','male',999990,'Painter')
        -> &NewEmployee->KimJ[];
 (9,'Kristensen et al.','Object Oriented Programming in the BETA language')
        -> &NewBook->Bible[];
 Birger.Display; PeterA.Display; Bible.Display;
 '========================='->putText; newline;
 Greg.init; Ereg.init;
 Birger[]->Greg.insert; Bible[]->Greg.insert; PeterA[]->Greg.insert;
 ClausN[]->Ereg.insert; LarsP[]->Ereg.insert; ElmerS[]->Ereg.insert;
 KimJ[]->Ereg.insert;
 Greg.display; Ereg.display;
 (if LarsP[]->Ereg.has then
     'LarsP in employee register'->puttext
  else
    'LarsP not in employee register'->puttext
 if);
 newline;
 (if LarsP[]->Greg.has then
     'LarsP in general register'->puttext
  else
    'LarsP not in general register'->puttext
 if);
 newline;
#)
```

```
ORIGIN '~beta/basiclib/v1.5/betaenv';
(* recordlib.bet
 *
 * COPYRIGHT
 *  Copyright Mjølner Informatics, 1989 - 1994
 *  All rights reserved.
 *)
INCLUDE '~beta/containers/v1.5/hashTable'
--- lib:attributes ---
Record:
  (# key: @Integer;
     Display:< (* declaration of a virtual (procedure) pattern  *)
       (#
       do newline;
          '-------------------'->putText; newline;
          'Record:      Key  = '->putText; Key->putInt; newline;
          INNER
       #);
  #);
Person: Record
  (# name,sex: @Text;
     Display::<  (* a further binding of Display from Record *)
       (#
       do 'Person:      Name = '->putText; name[]->putText; newline;
          '             Sex  = '->putText; sex[]->putText; newline;
          INNER
       #);
  #);
Employee: Person
  (# salary: @Integer; position: @Text;
     Display::<
       (#
       do 'Employee:    Salary = '->putText;   salary  ->putInt; newline;
          '             Position = '->putText; Position[]->putText; newline;
          INNER
       #);
  #);
Student: Person
  (# status: @Text;
     Display::<
       (#
       do 'Student:     Status = '->putText; Status[]->putText; newline;
          INNER
       #)
  #);
Book: Record
  (# author, title: ^Text;
     Display ::<
       (#
       do 'Book:        Author = '->putText; Author[]->putText; newline;
          '             Title  = '->putText; Title[]->putText; newline;
          INNER
       #)
  #);
NewRecord:
  (# resultType:< Record;
     new: ^resultType;
     key: @Integer;
  enter key
  do &resultType[]->new[];
     key->new.key;
     INNER;
  exit new[]
  #);
NewPerson: NewRecord
  (# resultType::< Person;
     N,S: @Text
  enter (N,S)
  do N->new.Name; S->new.Sex;
```

```
      INNER;
   #);
 NewEmployee: NewPerson
   (# resultType::< Employee;
      S: @Integer; P: @Text
   enter (S,P)
   do S->new.Salary; P->new.Position;
      INNER;
   #);
 NewStudent: NewPerson
   (# resultType::< Student;
      S: @Text
   enter S
   do S->new.Status;
      INNER;
   #);
 NewBook: NewRecord
   (# resultType::< Book;
      A,T: @Text
   enter (A,T)
   do A->new.Author; T->new.Title;
      INNER;
   #);

 Register: HashTable
   (# element::< Record;
      (* Virtual class specifying the (generic) element
       * type of the hashtable. *)

      hashfunction::< (# do e.key->value #);
      (* Specialization of the hashfunction to use on
       * elements of the hashtable. *)

      Display:<
        (* Display all elements of the table. *)
        (#
        do newline; '############ Register Display '->putText;
           INNER;
           newline;
           scan (# do current.display #);
           '############ End Register Display #######'->putText; newline
        #);

      Has:
        (* Check if an element is present in the table. *)
        (# e: ^element;
           found: @Boolean;
        enter e[]
        do
           e[]->hashfunction->findIndexed
           (# predicate::< (# do current.key = e.key->value #);
              notFound::< (# do false->found #);
           do true->found;
           #);
        exit found
        #);
   #)
```

Index