Technical Report IR–LP–31–21
September 1989

The Design and Implementation of a Two Process Prolog Debugger

Diomidis Spinellis

Abstract

A Prolog debugger running in the same process as the debugged program presents some problems. In many environments the debugger and the debugged process are separate. We examine how this separation is commonly implemented and present a system abstraction based on a set of primitives for accessing a trace line continuum. We then describe the implementation of such a system, giving how the two processes are created, how they communicate and how the underlying Prolog system meshes with our implementation.

European Computer-Industry
Research Centre GmbH
Arabellastr. 17
D-8000 München 81
West Germany

Contents

List of Figures

Chapter 1

Introduction

Opium [5] is an extensible Prolog debugger. One of its key features is that its command language is Prolog. This allows for sophisticated interaction with the user [6], implementation of *scenarios* — Prolog programs that present to the user a higher level debugging abstraction by filtering and processing raw debugging information — and a flexible user interface.

In this paper we present the design and implementation of a two process version of *Opium*. The design description will be limited to the aspects of transforming the one process *Opium* debugger to a two process debugging system.

1.1 The Previous Implementation of *Opium*

Opium was implemented as a single process debugger. This was made possible by a context switching mechanism. The user program was executing in one context and the debugger scenarios in the other. When information was needed from the user program (such as the next trace line) the context switched from the debugger process to the user process. The context switch involved saving and restoring the Prolog stack and all the state variables of the Prolog interpreter.

1.1.1 Advantages

The approach was relatively easy to implement. As the whole data was accessible from both contexts it was possible for the debugging scenarios to access and modify the dictionary of the debugged session. All the information needed for debugging could be extracted by looking at the appropriate data structures. The communication overhead between the two contexts was minimal as in many cases data structures could be shared.

1.1.2 Problems

The implementation was not without problems however. As the debugger and the user programs were executing in the same process they were very closely coupled. This was undesirable firstly from a pragmatic point of view as there could be interference between them. From a theoretical point of view it was almost impossible to abstract the features needed by the debugger in order to create a portable framework that could be applied to any Prolog system. One additional problem became apparent as the complexity of the scenarios grew. The loop scenario is about 3000 lines long [7] and naturally during its development a debugger was needed. That meant a debugger debugging the debugger, a model which was not supported by the previous implementation.

1.2 The New Implementation

The problems mentioned above can be solved by implementing the system as two processes. Their data areas are totally disjoint and the loose coupling between them helps defining an interface abstraction. By implementing the system in a general way, debugging the debugger is a natural extension.

New problems have been tackled in the new approach:

- How to partition the two processes. Various amounts of responsibility and processing can be delegated between them. The extremes vary from the debugger being hooked to the lowest level of the Prolog interpreter in the debugged process, up to the debugger being a dump terminal programmable in Prolog. The conceptual point where the two processes communicate creates a partition between them that must be crossed by some sort of message passing mechanism. Partitioning too close on the debugged process side will make the system inefficient, making the partition close on the debugger side will make the system inflexible.
- The synchronisation of the two processes. Since both processes can execute concurrently a protocol must be defined both, in order to present to the user a smooth debugging environment and for the processes to stay synchronised during the whole session. This problem is quite common in such applications [19].
- The set of primitives and the communication interface have been redefined for communication between the two processes.

Chapter 2

Related Work

Opium, as far as we know, is the only Prolog debugging system which runs as two separate processes. The idea however of using one process to debug another is quite common in multitasking operating systems. Multics, UNIX [1] and VMS use this approach [14] [2] [3]. The usual procedure is to arrange for a separate process, the debugger, to somehow attach itself to the debugged process and from that point control the execution.

2.1 Requirements of a Debugging System

It has been recognised that debugging systems are a valuable aid to programer productivity [9]. The functionality a programer expects from a debugging tool is outlined in [15]. A debugger should be able to:

1. Start, stop and resume the program execution.
2. Show the execution flow of the program.
3. Call arbitrary pieces of code in the debugged program.
4. Display a *traceback* of procedures called.
5. Keep a history of the program execution.
6. Access variables by their names.
7. Display the program being debugged.
8. Give the user access to some sort of debugging language.
9. Allow the user to customise the interface to the debugger.
10. Provide a user interface to the command language so that commands can be reexecuted.
11. Provide some sort of profiling information.

A lot of this functionality (7, 8, 9, 10) can be addressed at the user level, i.e. no support from the operating system is needed. The rest of the functionality can be built on top of two low level functions provided by the underlying system.

[1]UNIX is a registered trademark of AT&T in the USA and other countries.

1. The ability to execute exactly one instruction of the underlying execution mechanism or, alternatively, to be able to set a breakpoint at an arbitrary place of the code.
2. The ability to examine and modify the complete state of the execution including variables, code (source and executable) and internal variables of the executing machine.

We will examine two approaches for providing this functionality in the UNIX domain. Some general conclusions based on this study will be presented in section 2.4.

2.2 The UNIX *ptrace* approach

The traditional way one process can debug another under UNIX is using the *ptrace* system call [8]. A typical setup used by debuggers such as *adb* [12], *dbx* [18] and *gdb* [16] is to run the program from within the debugger or to start the debugger specifying the process-id of an already running program. Then the user can issue commands to the debugger which are translated to appropriate *ptrace* requests.

Ptrace requests allow the debugging process to:

- Read the debugged process' code, data and user information. (`PT_READ_I`, `PT_READ_D` and `PT_READ_U`).
- Modify the debugged process' code, data and user information. (`PT_WRITE_I`, `PT_WRITE_D` and `PT_WRITE_U`).
- Continue the execution of a stopped process. (`PT_CONTINUE`).
- Terminate the debugged process. (`PT_KILL`).
- Execute a single instruction. (`PT_STEP`).

The two process debugging implementation using the *ptrace* approach has been criticised as inefficient [2] [11]. Three are the main reasons for the inefficiency:

- A single data structure for the exchange of the messages is shared by all processes. Thus every time the data structure is to be used it must be locked in order to ensure mutual exclusion.
- Only a small amount of data (one word) can be transferred by each call.
- Two context switches are needed for every request.

2.3 The UNIX */proc* approach

A better solution than the *ptrace* approach whereby the debugger views the process as a special file has been proposed [10] and implemented under the Eighth Research Unix Edition [1]. Under this approach an image of each executing process can be found in a special file (named by appending its process-id to "`/proc/`"). The processes' code, data and stack are located in documented positions of the file and they can be read or modified by using the normal *read* and *write* system calls. Thus in order to examine the value of a variable the debugger finds the address of the variable from the symbol table of the process and seeks to the appropriate position of the file. Reading from that position gets the value of the variable, writing to it modifies it. Special requests (other than reading or modifying the process' image) are handled by the system *ioctl* call. A set of parameters allows the debugging process to:

- Start, stop or wait for a stop of the debugged process. (`PIOCSTOP`, `PIOCRUN` and `PIOCWSTOP`).
- Examine the *proc* structure where various system parameters of the executing process are stored. (`PIOCGETPR`).
- Specify that specific signals received by the debugged process shall be traced, ignored or received by the debugger. (`PIOCSMASK`, `PIOCSIG` and `PIOCKILL`).
- Cause the process to stop when another program is executed by it. (`PIOCSEXEC` and `PIOCREXEC`).
- Change the process' execution priority and access its executable file. (`PIOCNICE` and `PIOCOPENT`).

One side effect of this interface is that since normal I/O calls are used to communicate with the debugged process, they can be interrupted. As an example the debugger can set an alarm to be interrupted if the process does not reach a breakpoint after a set amount of time.

2.4 Lessons Learned

The (limited) examination of the approaches used for two process debugging systems which we described above supported the design of the two process implementation of *Opium* in a number of ways:

1. The debugger and the debugged process are always separate processes. In all the cases the debugged process runs under the control of the debugger. However provisions exist for the debugger to get control from the debugged process when an abnormal event occurs in it. For example the debugger can be given control when the process accesses an illegal memory location.
2. The interface implementations focus on time efficiency. Latter implementations always try to be more efficient either by addressing issues of the communication efficiency (such as the 8th Edition implementation) or by increasing the quantity and quality of the information that can be passed across them. As an example the latest implementation of *ptrace* under *SunOS* [17] added the following extensions:
   - The ability to read and write blocks of text and data instead of single words. This was done by adding the requests `PTRACE_READTEXT`, `PTRACE_READDATA`, `PTRACE_WRITETEXT` and `PTRACE_WRITEDATA`. This addition increases the *quantity* of information that passes through the two processes in a given amount of time.
   - The ability to stop an execution only whenever a system call is executed (`PTRACE_SYSCALL`) . This addition increases the *quality* of the information that is exchanged and delegates a lot of responsibility to the *debugged* process. It introduces the concept of *filtering* [7] at the context of the process being debugged.

   The approach of searching in the debugged process context as outlined in section 3.1 follows this notion.
3. A general and extensible interface strategy, based on simple primitives (such as the */ptrace* one based on *read*, *write* and *ioctl*) seems to be preferable to a complex and implementation specific one. In our implementation we defined a general and flexible base set of primitives (given in section 3.3) that satisfy all the debugging needs.
4. A two process debugging system will have to address asynchronous and non deterministic events. Although the debugged process is under the control of the debugger a request from the debugger may always lead to an irrelevant and unexpected response from the debugged process. As an

example a request for continuation up to a breakpoint can make the debugged process to signal a segmentation violation. The debugger must be able to handle such events. Section 4.4.3 gives a description on how this has been achieved in our system.

Chapter 3

The Two Process Debugger Design

In this chapter we will try to isolate the key design choices behind this implementation. We hope to distinguish them from technical implementation related decisions and thus present the framework onto which the debugger was built.

3.1 A Debugger Abstraction

A two process debugging system has been implemented by adhering to the following abstraction:

- A process can create another process which is controlling and debugging it.
- The granularity of items that can be traced is a trace line. The trace line is the basic source representation unit of the language being debugged. Associated with it, is its source representation and the relevant state of the execution environment.
- Trace lines form a continuum that starts at the beginning of the execution and ends at the end of it. Single stepping and breakpoints can be implemented by searching through the continuum for a line matching some characteristics. The search can be made either by specifying the characteristics as part of the primitive or referring to characteristics saved on the *slave*.
- The two processes communicate via primitives sent from the debugging process (the *master*) to the original process (the *slave*). Trace lines are accessed by searching forwards or backwards from a point in the execution history called the *current line*. Pragmatic considerations may dictate the need for additional primitives. These may be needed in order to implement a reasonable user interface or limit the amount of storage or processing time needed.
- The only requirement from the *slave* is for it to be able to present to the *master* the traced lines continuum. This can be implemented by a storage mechanism, or by reexecuting up to a point or by any other suitable mechanism.

We claim that a high level, configurable and extensible debugger for sequential Prolog can be built on top of this abstraction. A minimal set of primitives presenting this abstraction would be:

ForwardSearch Search forwards for the first line in the continuum that matches some given characteristics.

BackwardSearch Search backwards for the first line in the continuum that matches some given characteristics.

The characteristics can be things like the file and line number of the trace line, value of some arguments, the name of a procedure to be called or an interesting phenomenon of the target language.

SetEntityLeapable This primitive implements the “characteristics saved in the slave” part of the abstraction. It sets a specified entity such as a procedure or a variable or memory address to satisfy a future leap request.

ForwardLeap Search forwards for the first line that contains an entity that has been set as leapable.

BackwardLeap Search backwards for the first line that contains an entity that has been set as leapable.

GetCurrentLineData Retrieve the data that has been associated with the current line. This can be its file and line number, its source representation, values of variables or arguments.

3.2 Primitives Needed by *Opium*

The primitives needed in order to implement the scenarios of *Opium* are based on the above abstraction. However some more have been added in order to create a real life implementation. They can be classified into four broad categories.

1. Primitives that control the execution of the Prolog interpreter. Such predicates may cause the interpreter to abort the execution of the current goal or make the goal under execution fail. None of them are part of the abstraction, but they help to conserve user time as a wrong goal can be terminated immediately.
2. Primitives that can read or modify the state of the Prolog interpreter. These predicates can set the execution limits, control the recording of the lines executed, tracing and setting of spypoints.
3. Primitives that enable access to the Prolog system. These can be used to:
   - Access the source of the predicates. This is only true for this implementation, based on an interpreted Prolog, where the predicates can be trivialy “decompiled”.
   - Access the Prolog dictionary on th debugged process.
   - Implement conditional spypoints.
   - Allow the user to execute goals from within the debugger.
4. Get information about the program execution. This includes accessing all lines that have been or will be executed.

3.3 Primitives Implemented

The following set of primitives was implemented:

3.3.1 Prolog Interpreter Control

`Abort/0` Abort the current execution and returns control to the top level.

`Fail/0` Cause the current goal to fail.

3.3.2 Prolog Interpreter State Control

`VarValueGet/2` Return the value stored in a specified address in the *slave*.

`VarValueSet/2` Set the value stored in a specified address in the *slave*.

These primitives are used to read or modify global C variables of the debugged process. The variables that are currently accessed are:

`LimitDepth` Specifies how deep the execution may nest before it gets interrupted.

`LimitCallNumber` Specifies how many goals may be invoked before the execution gets interrupted.

These two limits can be used to halt non-terminating computations and are used by the loop analysis scenarios.

`sm_in_opium` This variable enables the *master* to see if a goal is under execution or not. It is needed as only a limited number of commands have meaningful semantics when no goal is executed.

`RecordOn` Can be used to enable the recording of the lines executed.

`PredFlagSet/4` Modify the flags associated with a predicate. These flags can specify for the predicate to be hidden or to set a spypoint.

`PredFlagGet/4` Get the value of the flags associated with a predicate.

3.3.3 Prolog System Access

`RemoteExec/2` There is only one primitive used to access the Prolog system. It allows any predicate to be executed on the *slave* and returns the results back to the master. Its generality allows it to be used to access the source of the predicates in the *slave*, or executing goals from the *master* prompt on the *slave*.

3.3.4 Prolog Execution Information

These primitives operate on the notion of the *current line*. Unlike other debugging systems in *Opium* there is no imposed difference between code that has been executed and code that will be executed. They are both viewed in a unified way. The *current line* can be set by asking for a line that has some characteristics. The search starts from the last *current line* and can be specified to proceed either forwards or backwards. Execution may continue if the recorded lines do not contain the line searched for and the search is going in the forward direction. The execution is however more a side effect of the search than an intended action. The abstract model used by the scenarios usually does not take into account the execution.

`CurrArg/1` Return the arguments of the current line.

`CurrChrono/1` Return the chronological number of the current line in the sequence of all trace lines.

`CurrCall/1` Return the call number of the current line.

`CurrDepth/1` Return the execution depth of the current line.

`CurrPort/1` Return the debug port of the current line according to Byrd's box model [4].

`CurrPred/1` Return the name of the predicate executed at the current line.

`CurrArity/1` Return the arity of the predicate executed in the current line.

`Leap/0` Search forward for a call to a predicate on which a spypoint has been set.

`FGet/10` Search forwards for a trace line matching some given characteristics.

`BGet/10` Search backwards for a trace line matching some given characteristics.

The characteristics given to `FGet` and `BGet` are a conjunction of call number, call depth, debug port, predicate name, and chronological number lists or ranges.

Chapter 4

The Two Process Implementation of Opium

4.1 The Structure of the System

A session in *Opium* starts like a session in any Prolog system. When the user wants to start debugging, the command '`tracing(on).`' needs to be given. At that point the Prolog system changes into a two process debugging system. The original process is called the *slave*. A new process is started up by the *slave* and will be used to control the execution of it. The new process, naturally called the *master*, is a fresh invocation of Prolog and appears in a new window. The two processes communicate via pipes. Both processes run the same code, namely the modified Prolog interpreter. At various points of the Prolog execution the *slave* stops and waits for a command from the *master*. The *master* can execute Prolog predicates that send commands to the *slave*. The *slave* receives a command, acts accordingly and sends a reply back to the *master*. Figure 4.1.1 shows the components of the system and the execution of some commands.

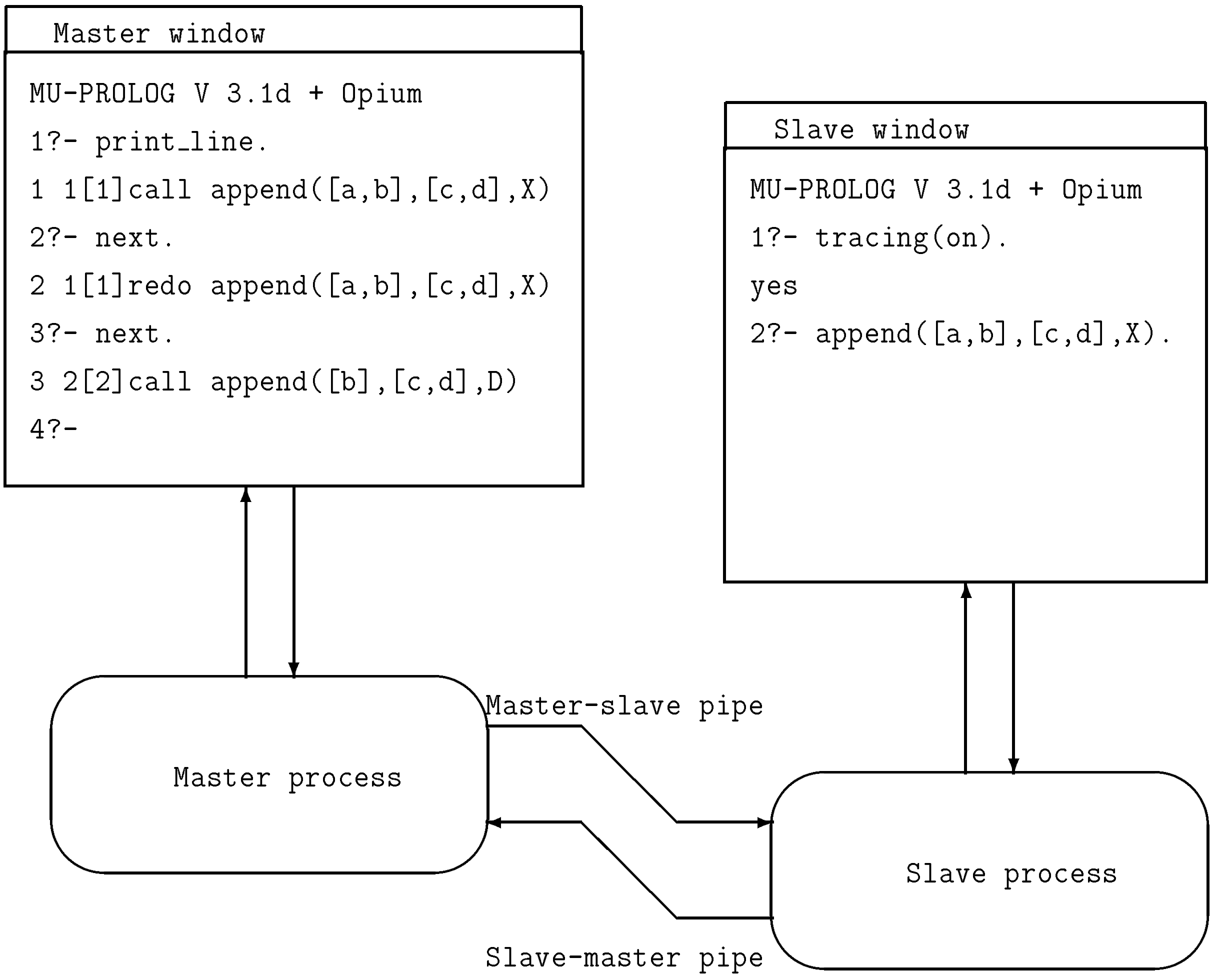
4.2 A Tale of Two Processes

The real life of the debugging system starts when the *master* process is started up. At that point the system can function as shown in Figure 4.1.1. We will examine what the two processes have in common, how they differ and then show how each one of them acquires its properties.

4.2.1 Things the Processes Have in Common

Both processes are an invocation of a modified *MU-Prolog* [13] interpreter. The modifications are mainly:

- Additional predicates which form the core of the *Opium* debugging system. These predicates form the *extraction module* [7].
- A recording mechanism for saving the history of the computation in the form of trace lines.
- A modified Prolog main loop with hooks that can divert the execution flow into the *Opium* system. This is used for handling recoring and some of the predicates.



```
MU-PROLOG V 3.1d + Opium
1?- print_line.
1 1[1]call append([a,b],[c,d],X)
2?- next.
2 1[1]redo append([a,b],[c,d],X)
3?- next.
3 2[2]call append([b],[c,d],D)
4?-
```

```
MU-PROLOG V 3.1d + Opium
1?- tracing(on).
yes
2?- append([a,b],[c,d],X).
```

Figure 4.1.1: The two process implementation of *Opium*

- The module that handles the mechanics of the communication between the two processes.

4.2.2 Differences of the *slave*

The differences of the *slave* from a normal Prolog session are the following:

- It has the C variable `master_exists` set to true. This is used so that the *slave* will not try to communicate with a non existing *master*. The sequence that starts up a master also checks this variable in order to avoid the creation of two masters.
- The C variable `master_process_id` contains the process id of the *master*. It is used for sending signals to the *master*.
- It has the *stdio* streams `fslavein` and `fslaveout` connected via pipes to the *master*. These are used for reading commands from the *master* and sending replies to it.
- It has arranged to kill the *master* when it finishes execution.
- It has arranged to be notified when the *master* finishes execution, so that it can revert to being a normal Prolog session. This is done by undoing the differences described above, that is closing

the two files and setting `master_exists` and `master_process_id` to 0.

4.2.3 Differences of the *master*

The differences of the *master* from a normal Prolog session are the following:

- The C variable `master` indicates its level on the master hierarchy. For a *master* that has been fired up by a normal Prolog session this variable has the value of 1. If the *master* was fired up in order to debug Prolog code in a lower level *master* (e.g. in order to debug a complex scenario) then the variable has the value of 2. This can go on to higher levels, although we don't think that there will be often more than two masters in the hierarchy. This variable can be examined by Prolog predicates to see if the process where the predicate is executed in is a *master*. The variable can also be displayed to help the user tell the processes apart.
- It has the *stdio* streams `fmasterin` and `fmasterout` connected via pipes to the *slave*. They are used for sending commands to the *slave* and getting results back.

4.2.4 The Creation of a Master

The *master* is created by the `create_master_c/0` predicate, which in turn calls the `OpCreate-Master` C function. This C function after checking that no master already exists, creates two pipes to be used for communication and then *forks*. This means the original process splits in two. The original is called the *parent* and the new one is called the *child*. Both continue the execution from the point after the call to the *fork*. All open files and variable values exist in both processes.

The original process is the *slave*. It does the following actions:

1. Closes the pipe ends which the *master* will be using.
2. Reopens its own pipe ends as buffered *stdio* streams called `fslavein` and `fslaveout`.
3. Sets the function `kill_master` to be called when it exits using the `on_exit` function. The function `kill_master` just sends a `SIGKILL` signal to the master process. This is needed so that the *master* will terminate together with the *slave*.
4. Arranges for the function `master_is_dead` to be called when a `SIGCHLD` signal is received. This signal, sent whenever something happens to a child process, would indicate that the *master* has terminated. Since *master* is not the only possible child of the slave (the slave can invoke other UNIX commands using the *system* predicate of Prolog) the action to this signal is temporarily suspended each time another process is executed from the *slave*. This creates a small window in which the death of a *master* would go unnoticed. The way to eliminate this window would be to recode all uses of the C library *system* function to an elaborate system of *wait*s and return value checks.
5. Finally the predicate returns `SUCCEED` and the normal Prolog execution resumes.

4.2.5 The Creation of a Slave

The child of the *fork* is the *master*. The *master* does the following actions in order to set up an new fresh Prolog process:

1. Closes the pipe ends which the slave will be using.

2. Increment the C variable `master` to pass it to the new process.

3. Create a suitable command line for executing the new master. This is done by reading the `OPIUM_WINDOW` environment variable, which is set by the user to the command that can be used on the system to fire up an new window running a given command. A typical value under X-11 might be:

   ```
   xterm -T Opium_%d -n Opium_%d -e
   ```

   The `%d` in the variable is replaced by the value of the `master` variable. In the case above it is used so that the *master* level will be shown on the title bar of the window.

4. Modify the command line so that the new process starts up as a *master* and gets the information needed. That is:

   - The fact that the process is a *master* and not a normal Prolog session.
   - The file descriptor numbers to be used for writing commands and reading results.
   - The level of the *master* in the hierarchy.

   This is done by giving to the new process the argument `-master` followed by three numbers, them being the ASCII representations of the hierarchical level of the *master-to-be* and the two file descriptors. Thus at the end the command to be executed would be something like:

   ```
   xterm -T Opium_1 -n Opium_1 -e Opium -master 4 5 1
   ```

   In the above line 4 is the file descriptor used for writing commands to the slave, 5 the one of reading the results and 1 the *master* level.

5. Execute the new process. The old process is overlaid by the new one. The name of the process to execute is taken to be the zeroth command line argument. The only link remaining between the two processes are the two file descriptors.

6. As the new process starts up a check is made for the first argument being `-master`. Since that is the case the next three arguments are processed in a special way.

7. The C variable `master` is set to be the value of the first argument.

8. Two *stdio* buffered streams are opened on the values of the file descriptors passed as seconds and third arguments.

9. The normal Prolog initialisation starts and the main loop is entered.

4.3 Method of Communication

In this section we explain the way the two processes communicate. We examine the way the two processes communicate and the structure of the messages exchanged. The synchronisation of the two processes will be examined in section 4.4.3.

4.3.1 Communication Protocol

A simple *request-reply* protocol — with the *master* initiating the requests — has been implemented. All communication from the *master* to the *slave* goes through the `write_command_to_slave` C function. This function receives a command number, the types of the arguments and the arguments to send to the *slave* assembles them into a message, sends the message to the *slave* reads the results back and returns. On the *slave* side the command, the types of the arguments and the arguments themselves are read by the `read_command_from_master` function, the command is executed and the (possibly modified) arguments are passed back to the *master*.

4.3.2 Structure of the Messages

A message exchange sequence is done in the following steps:

- The *master* sends a command number to the *slave*. This is the number assigned to an executable function via an enumeration. The slave calls the appropriate function by indexing an array of pointers to functions using this number.
- The *master* sends the types of the arguments. Argument passing is not implemented as a general mechanism. Rather, all the different possible combinations were found and hard coded into a switch statement. Although this approach is not flexible (it requires a change each time a function with different arguments is added) it is efficient and portable. A general solution would be machine specific as we do not know of a portable way to call an arbitrary function with a variable number of arguments.
- The *master* sends the actual arguments. Since the types of the arguments known, the actual arguments can be read by the *slave* into the correct variables. Strings are passed by first giving their length as an integer and then the actual string data. One more special case is the passing of Prolog terms. This will be discussed in section 4.4.5. At this point it is enough to note that the appropriate Prolog printing and parsing functions are called on each end.
- The *slave* sends the modified arguments back to the *master*.
- The *slave* sends the return code of the execution (`SUCCESS`, `FAIL`, `ERROR` etc.) to the *master*.

4.4 Interface with the Prolog System

4.4.1 Predicate Splitting

In order to present a consistent interface abstraction and as a result of the previous implementation of the system all commands are just the back ends of Prolog predicates. In fact the commands started their life as simple C function implementations of Prolog predicates. Such a C function will typically examine the type of its arguments, perform some action on them and then bind any free Prolog variables to the appropriate values.

In the two process implementation each C implementation of a predicate gets split into two parts, one half is executed in the *master* and the other in the *slave*. The part in the *master* is the one actually called from Prolog. It unbundles and checks the arguments, calls `send_command_to_slave` with the command number, the types of the arguments and the arguments, binds the modified arguments to the appropriate Prolog variables and returns to Prolog. The *slave* halve is called by the `read_command_from_master`

```
MasterOpCurrArity(t, l)
        Ptr t;
        levtype l;
{
        long arity;

        /* Send the command to the slave.  Arity will be set to the value. */
        write_command_to_slave(sm_CurrArity, sm_argument_types_l, &arity);
        /* Unbundle the argument */
        findbind((Ptr)targ(1, t), l, &t, &l);
        switch (tagtype(t)) {                     /* Check its type and act */
        case TAGNUM:                              /* It is a number */
                if(tnum(t) == arity)              /* Check if they match */
                        return(SUCCEED);          /* Yes, succeed */
                else
                        return(FAIL);             /* No, fail */
        case TAGVAR:                              /* It is a variable */
                nbind(t, l, ConsInt(arity));      /* Bind it to arity */
                return(SUCCEED);                  /* succeed */
        default:                                  /* Wrong type */
                plerror(EUFUNC);                  /* Signal an error */
                return(ERROR);
        }
}
```

Figure 4.4.1: The implementation of the *master* part of `CurrArity`

.

function. The function called is the one pointed to by the array of pointers to functions, indexed by the number sent by the *master*. It is called with pointer to the arguments sent by the *master*. After executing its part, it sets the variables to the correct values and returns.

As an example let us consider the sequence of events when a typical split predicate is called. `Curr_arity_c` is a predicate which unifies its argument with the arity of the current predicate in the slave. Its C implementation for the *master* is shown in Figure 4.4.1 and for the *slave* in Figure 4.4.2.

When `Curr_arity_c` is called from Prolog in the *master*, the following things happen:

1. `Curr_arity_c` calls the master part of the C function `OpCurrArity`.
2. `OpCurrArity` calls `write_command_to_slave` with its special code and the arguments it was given.
3. `Write_command_to_slave` writes the special code for `OpCurrArity` and the arguments passed to the *slave*.
4. The *slave* reads the function code and the arguments. This is done by the `read_command_from_master` C function.
5. The `read_command_from_master` C function calls the slave part of the `OpCurrArity`.
6. The slave part does any processing needed and returns to `read_command_from_master`.

```
SlaveOpCurrArity( arity )
        long *arity;
{
        /* Check the current predicate */
        if (IsAtom(Curr.Pred))
                /* It is an atom.  Arity is 0 */
                *arity = 0;
        else
                /* It had parameters.  Extract its arity from the dictionary */
                *arity = d_nargs(tdict(* Curr.Pred)) - 1;
}
```

Figure 4.4.2: A (slightly simplified) implementation of the *slave* part of `CurrArity`

.

7. `Read_command_from_master` passes the modified arguments and the return code back to the *master*.
8. The *master* `write_command_to_slave` function passes the results back to the master part of the `OpCurrArity`.
9. `OpCurrArity` passes the results back to `Curr_arity_c`.

4.4.2 Arbitration

After the *master* is created, the *slave* has to respond to commands from the *master* as well as to normal user requests. As an example, a user might first set a breakpoint by executing the `spy` predicate on the *master*. A command, `PredFlagSet`, would be sent to the *slave* to set a spy point on the predicate. The the user would enter a goal on the *slave* and a `leap` command on the *master*. Clearly the *slave* must be able to accept commands from both the user and the *master*. If the *slave* was to try to read from the terminal it would block in the *read* call until something was entered. No commands from the *master* could be accepted during that time. Specifying for the *read* call not to block and continuously loop trying to read from the terminal and from the *master* would be horrendously inefficient.

The problem is solved by the `arbitrate` function. Whenever Prolog tries to read from a file or device the `arbitrate` function is first called. This function uses the *select(2)* system call to multiplex between reading a command from the *master* and reading user input from the terminal. *Select* receives a bit vector of file descriptors which the program is interested in reading, writing or checking for abnormal conditions. It then blocks until one of the descriptors given is ready for the action specified. At that point the vector is updated to show which descriptor is ready. In our case we specify for *select* to block until either input is available from the terminal or in the descriptor associated with `fmasterin` — the descriptor where commands from the *master* are received.

The pseudo-code for the `arbitrate` function is shown in Figure 4.4.3. If data is available from the terminal `arbitrate` simply returns to the caller — the Prolog function reading from the terminal. That function will now read without blocking (since input is available). This cycle is repeated for every character read. As Prolog uses buffered *stdio* streams for input the `arbitrate` function first checks if there is any buffered data available before calling *select*. In the case where data form the *master* is available `read_command_from_master` is called which reads and executes the command. Then the `arbitrate` function will loop, calling *select* again.

```
arbitrate()
{
        if (buffered_data_available_on_input())
                return;
        for(;;) {
                select();
                if (data_available_on_input())
                        return;
                if (data_available_from_master())
                        read_command_from_master();
        }
}
```

Figure 4.4.3: Pseudo-code for the `arbitrate` function

.

There are two more special cases that have to be handled by the arbitrate code. The *select* call might get interrupted and in that case it will return with an error indication. The reasons for the interrupt could be:

- The tracing of the program from a debugger.
- The termination of the *master* with a corresponding signal being sent to the *slave*.
- The termination of the *master* making one of the file descriptors which *select* is examining invalid.

One might wonder why there are two possible outcomes of a *master* termination. The reason for this is a race condition. When the *master* terminates *stdio* closes the open descriptors. If the *slave* gets run by the operating system after that time and before the *master* actually terminates then a bad descriptor error (`EBADF`)will be returned from *select*. If on the other hand the *master* fully terminates before the *slave* gets a chance to run then the *select* will be interrupted by the signal of the *master* death(`EINTR`).

4.4.3 Synchronisation

Having examined all the components of the system we will now focus our attention on how they interact to orchestrate the intended result. The *slave* can be at any moment in one of five different states. The state transition diagram is shown in Figure 4.4.4. The states are:

1. *Top level loop*
   This is the state where the *slave* starts. In this state a prompt is printed, commands are read from the screen and executed as normal Prolog predicates.

2. *Arbitrate*
   This state is entered when a Prolog *read* predicate is encountered in state 1. At this state the `arbitrate` function is checking for input from the *master* or from the user. If input comes from the user, then the *slave* moves back to state 1.

3. *Top level read command from master*
   This state is entered when a *master* command appears while the *slave* is in state 2. A command from the *master* is read and executed, the result is written back immediately. Only a subset of commands can be meaningfully executed here, as no program is yet running.

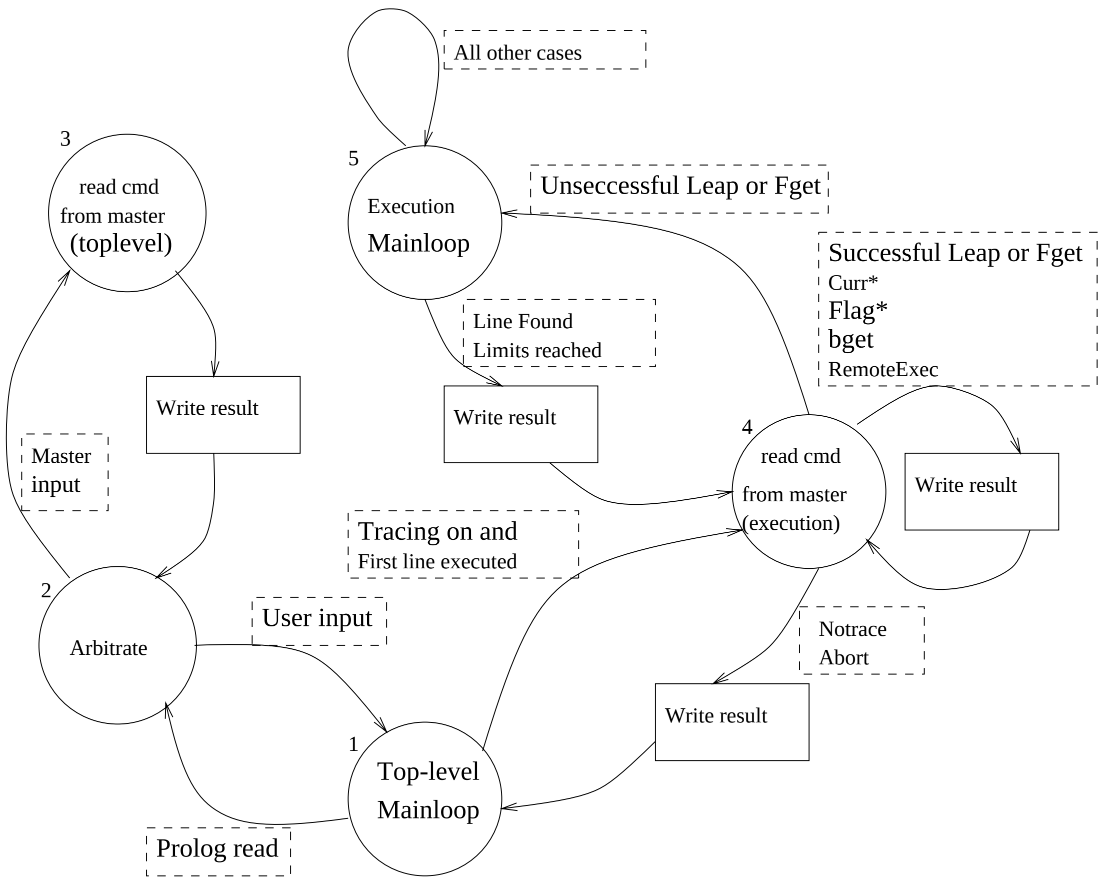


Figure 4.4.4: State diagram for the *slave*.

4. *Execution read command from master*
   This state can be entered in two ways:

   (a) When the first goal of a command is executed in state 1 and tracing is switched on.

   (b) When, while the *execution main loop* is running a line which has been previously requested is found, or some execution limits are reached.

   Depending on the command received and the recorded history this state can be affected in three different ways:

   (a) When a command that can be executed immediately and does not alter the Prolog execution is requested from the *master* then a reply is given immediately and the state does not change. These commands are:

   - `Fail`
   - `PredFlagSet`
   - `PredFlagGet`
   - `CurrArg`
   - `CurrChrono`
   - `CurrCall`
   - `CurrDepth`
   - `CurrPort`

    - `CurrPred`
    - `CurrArity`
    - `VarValueGet`
    - `VarValueSet`
    - `RemoteExec`
    - `BGet`

    In addition to the above commands an `fget` or `leap` that can be satisfied by the recorded history are also handled in the same way.

   (b) If a `Notrace` or `Abort` command is received from the *master* then control is given back to state 1.

   (c) When an `fget` or `leap` are received from the *master* and they cannot be satisfied from the recorded history then the system moves to state 5 until a suitable line is found.

5. *Execution main loop*
   This state is entered when the *master* requests a line that does not exist in the recorded history via an `fget` or a `leap`. In this case Prolog execution has to continue in order to satisfy the *master* request. When the suitable line is found, or the execution limits are reached the *slave* enters again state 4.

The manifestation of the various states in the source code is unfortunately not very clear. The reason for this is the structure imposed by the preexisting system. States 1 and 5 are the *Opium* `mainloop` function. In state 1 tracing is off. If tracing is turned on then the system is in state 5 when `mainloop` is executed. In state 5 after every goal execution the `TraceDisplay` function is called, which in turn calls `ExecOpiumProlog` if a suitable line has been found or limits have been reached. `ExecOpiumProlog` writes the result for the last `fget` or `leap` to the *master* and then calls `read_command_from_master`.

`Read_command_from_master` is used for states 3 and 4. It is called with a parameter which specifies whether it should loop on some commands (as it should do in state 4) or just return. When called from `arbitrate` no looping is specified, when called from `ExecOpiumProlog` looping is enabled.

4.4.4 Modification of C Variables

During the design of the system it was found that a number of split predicates were needed in order to check or set a C global variable on the *slave*. These variables control the tracing, recording, the state of the execution etc. Having to write two wrapper functions for each of these variables and add the corresponding enumerations and entries into the function pointer array was a repetitive and mundane process. A more general solution should offer increased flexibility. For this reason the predicate `var_value_in_slave/2` was written. The primitives `VarValueGet` and `VarValueSet` described in section 3.3.2 are the back ends of this predicate. This predicate unifies its second parameter with the name of a C global variable given as its first parameter. In order to do so it finds the location of the system executable, and looks the address of the variable in the symbol table. It then sends a command to the *slave* to set or get the value at specified address.

The location of the executable (whose name is given in `argv[0]`) is found by looking in all the places specified in the `PATH` environment variable. This search is only done if the name is not an absolute path. After the search is performed the result is saved, so this process is only preformed once. Also for performance reasons a small cache is maintained for all the variable addresses found in the symbol table. When the cache fills up a warning is printed and the symbol table is searched every time when variables are not found in the cache.

4.4.5 Passing Terms Between the Processes

Passing a Prolog term from one session to another is not trivial. Terms are stored in the form of a tree data structure inside the Prolog system. Clearly the pointers in the one session had no relation to the pointers in the other. For this reason the term tree needs to be traversed, its items converted into some portable form, transmitted to the other process and then reassembled at that end. This was almost equivalent to the `pwrite` and `pread` C functions that can write the ASCII representation of a term into a file or transform a string into a term. Thus whenever a term needs to be sent from the *master* to the *slave* the output of `pwrite` on the *master* is redirected to `fmasterout` and the input of `pread` on the *slave* is changed to `fslavein`. This approach is a bit slower than a system specifically constructed for this purpose as for example the integers are redundantly transformed to and from ASCII, but is more portable as all Prolog systems have predicates that can read and write terms.

4.4.6 Remote Execution of Predicates

As noted in section 3.3.3 a predicate is needed that can executed Prolog predicates on the *slave* from the *master* section. This is needed for example so that scenarios on the *master* can access the Prolog database on the *slave*. This predicate was implemented by having the *slave* save all its internal state variables, create a new environment, read the predicate to be executed into that environment, execute it, print the resulting term back to the *master* and restore the old state. One particular difficulty in writing this predicate was that the routines used for input and output were not reentrant and thus could not be called in the middle of a Prolog execution. A special routine was created that saved all the static variables of the I/O module in a stack every time a new level of nesting was needed.

4.4.7 Handling of Asynchronous Events

In this section we examine how the handling of asynchronous events could be implemented. This work has not been done, but looks like an interesting, valuable and straightforward extension. In many debuggers the user is able to interrupt the execution of the program by sending a signal from the keyboard. This should be also possible in *Opium*. The user should be able to type `^ C` at the master while the slave is executing and the master is waiting for a result. The effect should be the same as if execution limits had been reached — the user should receive a prompt at the *master* and control should return to `read_command_from_master` at the *slave*. This feature can be implemented as follows:

- The *master* arranges to propagate the interrupt signal to the *slave* by setting up a signal handler which then sends the signal to the slave by using in the *master* some code like:

```
master_sigint_handler()
{
        kill(getppid(), SIGINT);
}

master_setup()
{
        /* ... */
        signal(SIGINT, master_sigint_handler);
        /* ... */
}
```

- On the *slave* side a signal handler is set up to indicate via a global variable that a signal has been received. The Prolog main loop checks the value of that variable every time a new goal is executed, at the same place where a check for execution limits is made. If it is found that a signal has been received then a result is written back to the *master* and the system moves from state 5 to state 4.

Chapter 5

Conclusions

The system as implemented performed remarkably well. After the six month design and implementation period all the scenarios developed on the old system were used under the new one. The ability to debug scenarios was extensibly used and was a welcome addition. There were no complaints on the performance of the system, which indicates that the way the two processes were partitioned was acceptable. The whole functionality of the old system was retained. Implementing a Prolog debugger as two processes is possible without any disadvantage and might be the best way to implement advanced debugging tools.

Chapter 6

Acknowledgements

I would like to thank Mireille Ducassé and Anna-Maria Emde for their support during the whole project. Mireille spent countless hours trying to make me understand the ideas behind *Opium* and was extremely patient with my difficulty in comprehending her vision of debugging. Anna implemented the remote execution command, surely the most demanding and difficult primitive and gave me valuable and dependable help on all sorts of Prolog problems I had. Working with them was fun as well as a privilege. Finally I want to thank all the members of ECRC for making it possible for me to work in an exciting, productive and very pleasant environment.

Bibliography

[1] AT&T Bell Laboratories, Murray Hill, New Jersey. *UNIX Time-Sharing System, Programmer's Manual, Research Version*, February 1985. Eighth Edition.

[2] Maurice J. Bach. *The Design of the UNIX Operating System*, page 376. Prentice Hall, 1985.

[3] Bert Beander. VAX DEBUG: An interactive, symbolic, multilingual debugger. In M.S. Johnson, editor, *Proceedings of the Software Engineering Symposium on High-Level Debugging*, pages 173–179. ACM SIGSOFT/SIGPLAN, March 1983.

[4] L. Byrd. Understanding the control flow of Prolog programs. In *Logic Programming Workshop*, Debrecen, 1980.

[5] M. Ducassé. Opium, an extensible tracer for Prolog, prototype description, further specifications. Technical Report LP-14, ECRC, January 1987.

[6] M. Ducassé. Opium+, a meta-debugger for Prolog. In *Proceedings of the European Conference on Artificial Intelligence*, pages 272–277, Munich, August 1988. ECCAI.

[7] M. Ducassé and A-M. Emde. Automated debugging of real Prolog programs using symptom-driven abstraction. The non-termination analysis. Technical Report IR-LP-41, ECRC, July 1989.

[8] Computer Systems Research Group. *UNIX Programmer's Reference Manual*. Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California 94720, April 1986. 4.3 Berkeley Distribution.

[9] S.J. Hanson and R.R. Robinski. Programmer perceptions of productivity and programming tools. *Communications of the ACM*, 28(2):180–189, February 1985.

[10] T. S. Killian. Processes as files. In *Proceedings of the USENIX Summer 84 Conference*, pages 203–207. USENIX Association, 1984.

[11] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD Unix Operating System*, page 104. Addison-Wesley, 1988.

[12] J. F. Maranzano and S. R. Bourne. A tutorial introduction to adb. In *UNIX Programmer's Supplementary Documents, Volume 1*. Computer Systems Research Group, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California 94720, April 1986. 4.3 Berkeley Software Distribution.

[13] Lee Naish. *Mu-prolog 3.1db Reference Manual*. Melbourne University, 1984.

[14] Elliot I. Organick. *The Multics System: An Examination of Its Structure*, chapter 7.2.4, page 284. The MIT Press, 1972.

[15] R. Seidner and N. Tindall. Interactive debug requirements. In M.S. Johnson, editor, *Proceedings of the Software Engineering Symposium on High-Level Debugging*, pages 9–22. ACM SIGSOFT/SIGPLAN, March 1983.

[16] Richard M. Stallman. The GNU source-level debugger. Distributed by the Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, January 1989.

[17] Sun Microsystems Inc., Mountain View, California. *SunOS Reference Manual*, 1988. Release 4.0.

[18] Bill Tuthill and Kevin J. Dunlap. Debugging with dbx. In *UNIX Programmer's Supplementary Documents, Volume 1*. Computer Systems Research Group, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California 94720, April 1986. 4.3 Berkeley Software Distribution.

[19] R. Winder and J. Nicolson. Jdb: An adaptable interface for debugging. *Software-Practice and Experience*, 18(3):221–238, March 1988.