NeD: The Network Extensible Debugger

Paul Maybee – Solbourne Computer, Inc.

ABSTRACT

NeD is a debugging server with a programmable network interface. NeD is designed to be flexible and extensible enough to support a wide range of debugging needs. Debugging clients communicate with NeD by sending it programs to execute. The programming language, NeDtcl, is tcl[1] extended with 30 debugging specific functions. NeD can be used as a traditional debugger with a textual interface, but the user would find the language cumbersome. It is designed to be convenient for communication between programs, rather than between program and user. As a demonstration of NeD's viability as a debugging server, the pdb[2] debugger has been retargeted to use NeD as its server.

1.0 Introduction

The nature of UNIX software development constantly changes. Over the past several years workstations with graphics displays have largely replaced ascii terminals, resulting in ubiquitous windowing systems, and the enormous growth in user interface development. The increasing acceptance of object oriented programming has resulted in the widespread use of new programming languages to support it. Multi-process programming has always been common on UNIX machines, but its use has taken on new dimensions with networks and the ability to easily create and communicate with remote processes. Multi-threaded programming is supported now on many vendors' platforms, with more to come in the near future. All of these developments have resulted in a steady increase in the size and complexity of software systems. Yet the scope of changes in debugging technology over the same period of time has been narrow, focusing primarily on user interface issues. Several products, e.g., dbxtool [3] and SaberC (now CodeCenter) [4], have placed window system interfaces over essentially command driven debuggers. Pi [5] and pdb pushed this technology further by completely eliminating the command interface. SaberC made an initial attempt at data structure visualization, and FIELD [6] advanced this technology by the addition of layout methods and iconic structure representations.

FIELD also developed an inter-tool communication system (now available in commercial guise) that allows various other software development and visualization tools to interact directly with the debugger [7]. Tool-interconnection appears to be the next major step in software development environments, with several vendors developing products along the lines of the FIELD work.

Several debuggers now support debugging C++ programs, at various levels. One of the primary reasons for less than complete support in the debuggers has been inadequate compiler generated symbol tables. This problem should gradually disappear as better compilers become available.

The ability to debug multiple processes, even on the same machine, has typically not been supported, and UNIX implementations often have prevented debuggers from even attaching to newly forked process. Large program debugging seems to be an issue that is often ignored in the construction of debugging systems (with the exceptions of pi [12] and pdb). It is common for users go away and come back later when a program has finished loading.

NeD addresses all of these issues. NeD does not have a user interface to speak of; it is a server designed to work with client user interface tools. NeD was built specifically for the support of the C++ language, although it is suitable for use with other languages. However, its capabilities to debug C++ in a completely native mode still suffers somewhat for want of compiler support. NeD supports the debugging of child processes as they fork from debugged parents as well as debugging over the network and debugging multithreaded processes. NeD is extensible; a client may customize NeD's remote interface to more easily and efficiently provide needed services. Thus NeD will continue to be useful in the face of change by allowing clients to expand the power of the debugging server at run time. Finally, because of the use of lazy symbol evaluation, NeD is very efficient, especially in its use of memory.

In the following discussion, "NeD" will refer to the NeD server, "client" will refer to a client of the NeD server, "pdb/NeD" will refer to the version of pdb implemented using NeD, and "subordinate program" will refer to a program being debugged by NeD.

2.0 Features

2.1 Client-Server Debugging

NeD implements the server side of a client/server debugging system. The typical client provides a user interface, the NeD server provides debugging services. NeD executes on the same host as the subordinate program, and uses the operating system provided interface to query and control the process. NeD contains an extended tcl[1] command interpreter. Tcl provides an embedded, list structured, command language that provides "mechanisms for variable, procedures, expressions, etc."; the extension consists of a set of additional embedded functions that provide debugging capability. This extended language is NeDtcl (see Appendix A). The clients execute remote procedure calls that send NeDtcl statements to the interpreter. For example, a call may contain a direct invocation of an embedded debugger function, or contain statements defining new NeDtcl procedures, or it may contain code invoking these previously defined procedures.

The execution of the NeDtcl statements produces results that are forwarded back to the client as the return value of the remote procedure call. In addition , the execution of some statements causes the subordinate to change state, for example to begin execution. When an execution state change occurs NeD generates an event. The event is passed to the NeDtcl interpreter which may handle it, or send it on in a message to the client. NeD generates several classes of events automatically. Clients can also cause additional, user defined, events to be generated. A client handles the event when the message is received, or it can download procedures to the interpreter to catch and process events.

NeD supports a special kind of breakpoint that is placed only on a fork system call. When this breakpoint is hit, NeD modifies the program text following the fork call such that executing the new code will cause the process to suspend. NeD then allows the process to execute the fork. The new child process that is created will immediately suspend. NeD takes control of the parent again, loads the pid of the child, and opens a new socket. NeD then forks. The child copy of NeD attaches to the new subordinate child and accepts connections on the socket. The parent copy of NeD generates a "new_proc" event containing the pid and socket id of the child NeD. At this point the NeD client can connect to the new NeD and debug the child subordinate process (see Figure 1).

NeD clients can extend this picture by attaching NeD servers to multiple processes, even when those processes are on different machines. Pdb/NeD,

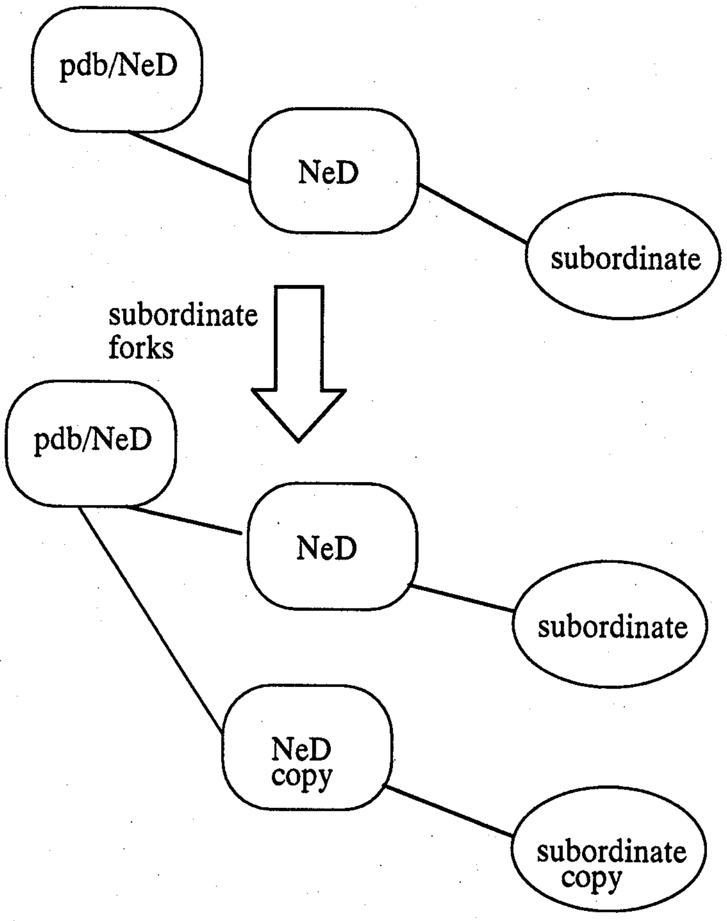


Figure 1: Debugging Processes that Fork

for example, will allow any number of processes to be debugged simultaneously through one set of windows. The user can select the process to view via a menu selection. In addition, a process being debugged will automatically come into view when it hits a breakpoint.

Using a client-server architecture has efficiency drawbacks when more sophisticated tools such as event stream recognizers or program monitors are being used [14,15]. Network latency makes events out-of-date by the time they arrive. Data collection may take up a great deal of network and machine resource. NeD alleviates these problems through its extensiblity. Debugging tools can implement event recognizers or data filters in the server itself (section 2.3) or link additional functionality into the subordinate process(section 2.4).

2.2 Displaying Data

All communication between NeD and its clients is textual, including values returned by expression evaluation requests. NeD supports displaying data structure values through "templates". Templates describe how fields are to be displayed, including conditional display based upon the program state. If an integer variable "i" with a current value of 10 is evaluated the result returned by NeD would be "SIMPLE NUMBER 10", indicating that a simple value was returned. A template for this expression would have a form similar to this result. The template "SIMPLE {hex tim}" would direct NeD to return the value in hexidecimal and as a system time value. Also allowed are octal, decimal, binary, ascii, unsigned decimal, floating, and special formats. The special option allows the user to customize the display of values in a way that makes sense in the context of the subordinate program. This option requires that the name of a function to be found in the subordinate be supplied, e.g., "SIMPLE {hex {spl *myfunction*}}". When this value is to be displayed, myfunction is invoked with the value as input. It returns a pointer to a static text string representing the value.

For aggregate values the format matches that of the data structure. Given the following structure definition:

```
struct s {
    int i;
    struct s *ptr;
};
```

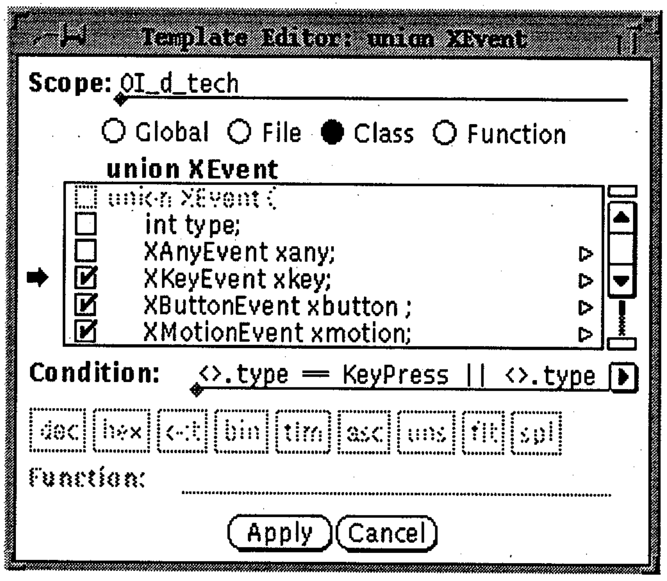
when a variable of this type is evaluated NeD returns:

```
AGGREGATE s {
    {i {SIMPLE NUMBER 10}}
    {ptr {SIMPLE PTR 0xf7fffb00}}
}
```

Each field of an aggregate template contains a three element list:: first the field name, then a conditional expression, and then the field's template. The template that matches "struct s", that displays "i" as above, and displays "ptr" only when "i" is non-zero looks like:

```
set_template {unsigned int} {CLASS foo} {SIMPLE oct}
```

Figure 2: Displaying unsigned integers in class "foo"



Template Editor: union XEvent

Scope: OI_d_tech

Global File [✓ Class] Function

union XEvent

```
☐ union XEvent {
☐     int type;
☐     XAnyEvent xany;
☑     XKeyEvent xkey;
☑     XButtonEvent xbutton ;
☑     XMotionEvent xmotion;
```

Condition: <>.type == KeyPress || <>.type

Function:

Figure 3: Pdb/NeD template editor

```
AGGREGATE s {
    {i {} {SIMPLE {hex tim}}}
    {ptr {<>.i != 0} {SIMPLE hex}}
}
```

Any boolean expression that can be evaluated in the context of the original expression can be used as an evaluation condition. The angle bracket pair (<>) in the example condition refers to the expression being evaluated, "exp", the brackets will be replaced by "(exp)" before evaluation. This is especially useful when displaying union types with tag fields. Each variant of the union can carry a condition such as "tag == mytag". Then when the union is evaluated only the correct union variant appears.

A template can be specified on an individual expression or for all expressions of a given type (in various scopes). For example, all unsigned integers in the scope of class "foo" can be displayed in octal by executing the NeDtcl command shown in Figure 2. Variables of type "unsigned int" evaluated in locations other than member functions of class "foo" will not be affected. Scope designations can include GLOBAL, FILE, CLASS, or FUNCTION scopes. When an expression is evaluated the templates will be searched for a matching type in the inner most matching scope. If no custom template is found then a default global template is used.

The set of custom templates that are in effect can be retrieved from NeD and saved. NeD can then be reloaded with these the next time that the program is being debugged. Templates for entire classes of debugging can be prepared a priori and used by groups of programmers. For example, the X window system [5] defines a union called an XEvent with thirty-three variants. Thirty-one of the variants are the different X event types and one of the variants is the tag field (i.e., the tag is an overlay of the first field in each of the events), the remaining variant is padding. A NeDtcl initialization file could be prepared for X programmers that would include a template for the XEvent; displaying the proper variant depending upon the tag field. Figure 3 shows the pdb/NeD user interface for editing type templates

2.3 Extensibility

NeDtcl's embedded functions provide, more or less, traditional debugger functionality. There are functions that, for example, set and delete breakpoints, advance execution, and evaluate expressions. The interface is stateless. There is no concept of a current function, or current file, as a more traditional debugger would supply. Commands tend to take additional parameters with which to specify program locations or options. For example, in dbx the user can type "stop at 10" to set a breakpoint at line 10 of the current file. When using NeDtcl this operation requires entering code as shown in Figure 4. This makes the language unacceptably cumbersome for human use, but not for computer use. The NeD interface is designed for use by a NeD client that presents an alternate, most likely graphical, user interface. This is not to say that NeD cannot be used as a textual debugger, only that the interface, by default, does not support it well. Since NeDtcl is a complete programming language a "dbx-like" user interface could be written in NeDtcl and loaded into the interpreter. In fact, to facilitate testing NeD, this was done.

NeDtcl does not contain functions that are not directly debugger related (e.g., dbx's "make") or functions that produce information that can otherwise be computed (e.g., dbx's "where"). Non-debugging functionality, such as recompiling objects, is

```
set_break [addr_of {thisfile.c 10 {}}] {} {} stop
```

Figure 4: A breakpoint in NeDtcl

```
### definition of traceback(stack_num, max_depth)

proc traceback {{stack_num 0} {max_frames 100}} {          ### compute traceback
    set d [min $max_frames [depth stack_num]]              ### how many frames?
    set trace ""                                           ### initialize list
    for {set i 0} {$i < $d} {set i [expr {$i + 1}]} {      ### for each frame
        set pc [get_pc $stack_num $i]                      ### get the pc
        set loc [location_of $pc]                          ### get source location
        ### concat the pc, location, and evaluated parameter list
        lappend trace [list $pc $loc
                [eval_list $loc $stack_num $i
                    [list_of_params $loc]]]
    }
    return $trace                                          ### return completed list
}
```

Figure 5: A traceback

assumed to be provided, if necessary, by the NeD client. Complex functionality, such as displaying a stack traceback, can be loaded into the interpreter. In the pdb/NeD implementation, the first operation pdb directs NeD to do at start-up is load a file of NeDtcl procedure definitions into the server's interpreter; one of which is "traceback"; see Figure 5.

The information returned by *traceback* is a list with the same information as is contained in a stack traceback obtainable from any source level debugger. *Traceback* traverses the execution stack returning a description of each stack frame. Each description is a list containing the pc, source location (including function name), and parameter list for the frame. A parameter list contains the name and current value of each parameter. In these examples itialics are used for names (e.g., *traceback*, *get_pc*, *eval_list*, and *list_of_params*) defined in the pdb/NeD start-up file; names in bold (e.g., **depth** and **location_of**) are embedded NeDtcl functions. The remainder is tcl code.

Getting a traceback requires only one remote procedure call. The previous version of pdb required one procedure call to get the location and parameter names and then another call for the evaluation of each parameter. An alternate way to decrease the remote traffic would have been to modify the former interface to include all the information returned by this traceback function. However, the problem with a fixed interface is that it does not provide enough flexibility. It may be that a future graphical debugger interface would require much less, or much more information for each stack frame (e.g., no parameter evaluation or values for all local variables). NeD provides this flexibility.

NeDtcl also provides support for the interception and generation of NeD events. Figure 6 implements a conditional step function. The procedure *step_until1* takes an expression (and some context information) and steps the program one source line at a time until the expression evaluates to true.

Figure 6 implements a function similar to a "watchpoint", i.e., break execution on a memory location content change rather than on a memory location being executed. At each execution of **step** the process changes state twice; it starts executing and then it stops executing. Each of these state changes will result in an event being sent to the client. If the client normally processes these events to update a display then this may result is a great deal of unwanted screen painting and network traffic. This procedure can be easily modified to intercept the events.

When NeD generates an event it invokes the NeDtcl function *process_events*. *Process_events* cycles through a list of event processor functions looking for one that has expressed an interest in the event. If one is not found, the event is sent to the client via the **event_pass** function. So to intercept the process state change messages, the "step until" procedure must be modified to add an event processor. The function *on_event* is used to post or remove an event processor. The first parameter indicates the event class to catch, the second parameter is a command to execute. If the command returns non-zero then the event has been handled. *Step_until2* installs a simple handler for all "proc_state" events before invoking *step_until1*, and then removes it afterwards. *Step_until2* also sends a "proc_state running" event to the client before

```
### step_until1(location stacknum expression)
###                 : no event interception

proc step_until1 {loc stacknum exp} {
    while {![proc_eval $loc $stacknum 0 $exp {} 0]} {
      step 0 {}
    }
}
```

Figure 6: A watchpoint

```
### step_until2(location stacknum expression)

proc step_until2 {loc stacknum exp} {
    event_pass {proc_state running}            ### program set running
    on_event proc_state {return 1}             ### handle event
    step_until1 $loc $stacknum $exp
    on_event proc_state                        ### remove handler
    checkstate          1                      ### generate final event
}
```

Figure 7: A more complex example

beginning execution and causes NeD to generate a "proc_state" event after stepping has completed. This allows the client's normal event processing to see the "step until" operation as it would a single step; see Figure 7.

The previous examples demonstrate NeD's ability to display the state and control the execution of a subordinate process, the following examples show that NeD can also perform non-trivial data queries on subordinates. Procedure *list_length1* will calculate the length of a linked list. It assumes that the list structure contains a next field, by default named "next", pointing to a structure of the same type. Given the head of the list, the function iterates until it discovers a next field with a special value, by default 0x0. Each succeeding next value is computed by adding ".next" onto the previous expression and reevaluating. See Figure 8 for the listing.

This implementation will work, but the size of the expression being evaluated grows with the length of the list, making it a less than perfect solution. An alternate implementation makes use of the expression evaluator's "typeof" intrinsic function to keep the length of the expression constant. Instead of using the previous expression to get the next expression to evaluate, *list_length2* uses the value of the previous expression. The next expression is generated by casting the value to a pointer to the type of the structure and then adding "->next". See Figure 9.

Queries such as this can be very useful in building advanced graphical user interfaces, for instance structure browsers. Network latency can be reduced by filtering data at the server rather than requiring network transmission. As with the templates, groups of users working on similar problems can share modifications to enhance a project's debugging environment.

2.4 Subordinate Extensibility

NeD allows the subordinate program to be extended at run time. NeD will load dynamic libraries into the executing subordinate and then allow functions in the library to be invoked through the debugger interface. This can be useful for supporting debugging aids, such as tools that traverse program data structures looking for inconsistencies, or performance monitoring functions. For example, the list_length function of section 2.3, or a free space analysis routine could be supplied by the user in a shared library. One of the first uses this was put to at Solbourne was to provide a file descriptor status utility. A function that 'stat's all the processes file descriptors is linked in and invoked. The function dumps a report to a file; the utility reads the report and displays the information to the user.

Newly linked library code can also be directly called by the subordinate process by inserting calls into the program text. This feature provides the facilities needed for less intrusive debugging and monitoring tools, such as Parasight [13]. The Parasight paradigm involves inserting new functions into a running process to collect performance and trace data. It is orders of magnitude faster to have the program collect its own data, than to have the debugger stop it continually to do so.

NeD causes the subordinate process to execute dynamic linker calls in order to load new libraries. Under SunOS, dynamic linker routines are always resident with any dynamically linked process. NeD exposes the 4 dynamic library routines necessary

```
### list_length1(loc stacknum head field lend): length of list
proc list_length1 {loc stacknum head {field next} {lend 0x0}} {
    set next [simplevalof [proc_eval $loc $stacknum 0
    $head {} 0]] set exp $head for {set i 0} {$next != $lend} {set i [expr
    $i+1]} {
      set exp ($exp).$field set next [simplevalof
      [proc_eval $loc $stacknum 0 $exp {} 0]] } return $i }
```

Figure 8: Calculating length of a list

```
### list_length2(loc stacknum head type field lend): length of list
proc list_length2 {loc stacknum head {field next} {lend 0x0}} {
    set next [simplevalof [proc_eval $loc $stacknum 0
    $head {} 0]] set type [simplevalof [proc_eval $loc
    $stacknum 0 typeof($head) {} 0]] for {set i 0} {$next!= $lend} {set i
    [expr $i+1]} {
      set exp (($type *)$next)->$field set next [simplevalof
      [proc_eval $loc $stacknum 0 $exp {} 0]] } return $i }
```

Figure 9: An alternate list length calculator

(dlopen, dlclose, dlsym, and dlerror) by placing symbols for them into the symbol table. Expressions can then reference the symbols to access the functionality they provide. On systems where the dynamic libraries are not always present in the subordinate the debugger must either use an alternate mechanism or require the image to be linked specially. Figure 10 is a sample NeDtcl function that invokes an arbitrary function in an arbitrary shared library.

3.0 Implementation

NeD is more than a demonstration of concept; it will form the basis for the next generation of Solbourne's debugger products. As such, it must be efficient and provide a full range of debugging features. NeD is still under development but is already robust enough and functional enough to support the pdb debugger. NeD is implemented for the SPARC architecture under SunOS 4.1.1. It can

```
proc dyncall {library func args} {
    set l simplevalueof
        [proc_eval {{} 0 {}} 0 0 dlopen($library,1) {} 0]
    set f simplevalueof
        [proc_eval {{} 0 {}} 0 0 dlsym($l, $func) {} 0]
    proc_eval {{} 0 {}} 0 0 ((int ())$f)($args) {} 0
    proc_eval {{} 0 {}} 0 0 dlclose($l) {} 0
}
```

Figure 10: An arbitrary function evaluator

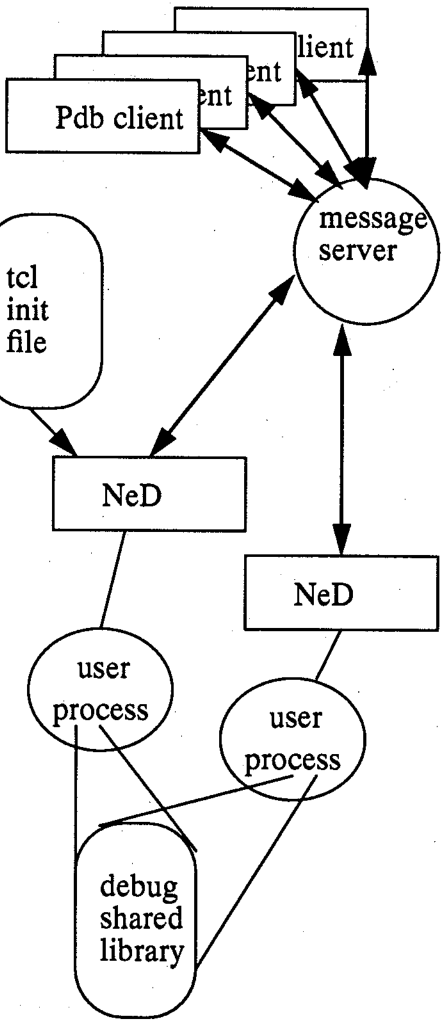


Figure 11: pdb/NeD architecture

debug C, C++, and Fortran programs at the source level, understanding proper target language syntax. Initial tests indicate that NeD uses considerably fewer resources than conventional debuggers such as dbx and gdb. Table 1 shows a comparison of the time and memory required to load a C++ program containing approximately 144,000 symbols. The tests generating these results were run on a Solbourne S4000 with 40 meg of memory.

resource	NeD	gdb	dbx
real time (sec)	33	29	140
memory (megs)	4	16	39

Table 1: Comparison of NeD, gdb, and dbx

4.0 Futures

4.1 Pdb/NeD Architecture

Pdb/NeD clients currently communicates directly with NeD servers. However they will eventually communicate through a FIELD-like message server [7,9]. Pdb/NeD is currently a single client, and that too will change. The pdb user will be able to invoke multiple, independent clients that communicate through the message server. Clients will be specialized, for example to display and edit annotated source, to evaluate expressions, to graph data structures, or to monitor communications. NeD servers will respond to queries from any client. Clients will extend the servers by downloading procedure definitions, or by having the servers load initialization files, or by mapping shared libraries that provide additional functionality into the subordinate's address space (see Figure 11).

4.2 Multi-threaded Process Debugging

NeD contains support for debugging multi-threaded programs, but the current implementation base (i.e., SunOS 4.1.1) does not implement multi-threaded processes, thus there is never more than one thread per process. NeD will be ported to a multi-threaded version of the SVR4 operating system in the near future. The port will also result in a switch from using the ptrace debugging interface to using the /proc interface [10]. A benefit of the move to the /proc interface is that it greatly simplifies debugging forked processes.

4.3 Grafting NeDtcl to Other Debuggers

A debugger for a modern software development environment cannot succeed without associated tools providing graphical user interfaces. In fact, most debuggers offer very similar underlying capabilities, with the main distinction between them coming in the interface or higher level analysis capabilities. User interface software written for the X window system, in our experience, has been easy to port between platforms. Porting debuggers is a much more difficult task since the debugger is dependent upon the operating system's debugging interface, the compiler-debugger interface, and machine architecture. Projects to standardize UNIX and to standardize the symbols generated by compilers, e.g., DWARF [11], will help make the task easier if adopted. But the task will not become as easy as porting X library dependent code, or other application software, in the foreseeable future.

The NeD viewpoint is that the investment in graphical tools can be most easily capitalized on if the debugger doesn't have to be ported. If the graphical tools are written to use a defined debugger interface such as NeDtcl, then only the interface must be ported to a new debugger. There are debuggers available, e.g., the Free Software Foundation's gdb, that have already been ported to many platforms. If such a debugger were to understand the NeDtcl interface then new tools could more rapidly be available across different platforms.

The NeDtcl portion of the NeD server amounts to about 1600 lines of C++ code, most of which is a straightforward translation of the parameters to, and results generated by, the embedded functions from character to internal format. The special formats required for expression value output (see "value" in Appendix A, Data Types) is not such a large change to the traditional value display procedure in any debugger. This would typically be a recursive routine that traversed an internal structure and prints out the contents of each field. The modifications consist mainly of printing some additional brackets and typing information that must already be available to such a module.

5.0 Acknowledgments and Availability

Andrew Gerber implemented the graphical user interface for NeD, making sophisticated testing possible. The members of the OI group served as guinea pigs for early versions of pdb/NeD. Their suggestions and comments have helped enormously.

NeD is still under development but will be available from Solbourne Computer's software business unit later this year. The initial version will be targeted for SPARC compatible computers running UNIX SVR4.

References

[1] Ousterhout, J. K. *Tcl: An Embeddable Command Language*, Proceedings of the Winter 1990 USENIX Conference, Washington D. C. (January 22-26, 1990)

[2] Maybee, P. *pdb: A Network Oriented Symbolic Debugger*, Proceedings of the Winter 1990 USENIX Conference, Washington D. C. (January 22-26, 1990)

[3] Adams, E. and Muchnick, S. S. *Dbxtool: A Window-Based Symbolic Debugger for Sun Workstations*, Software - Practice and Experience, V16 N7, July 1986, pp.653-669

[4] Kafer, S. Lopex, R. and Pratap S. *Saber-C An Interpreter-based Programming Environment for the C Language*, Proceedings of the Summer 1988 USENIX Conference, San Francisco, CA (June 20-24, 1988)

[5] Cargill, T. A. *The Feel of Pi*, Proceedings of the Winter 1986 USENIX Conference, Denver, CO (January 15-17, 1986)

[6] Reiss, S. P. *Interacting with the FIELD Environment*, Brown University Department of Computer Science Technical Report CS-89-51 (May, 1989)

[7] Reiss, S. P. *Integration Mechanisms in the FIELD Environment*, Brown University Department of Computer Science Technical Report CS-88-18 (October, 1988)

[8] Scheifler, R. W., Gettys, J., and Newman, R. *X Window System C Library and Protocol Reference*, Digital Press, 1988

[9] *ToolTalk Programmers Guide*, SunSoft, Inc. Revision A of September 1991

[10] Faulkner, R., and Gomes, R. *The Process File System and Process Model in UNIX System V*, Proceedings of the Winter 1991 USENIX Conference, Dallas, TX (Jan. 21-25,1991)

[11] *DWARF Debugging Information Format*, UNIX International Programming Languages Special Interest Group, October 21, 1991, DRAFT

[12] Cargill, T. A. *Pi: A Case Study in Object-Oriented Programming*, C++ Workshop Proceedings, Santa Fe, NM, USENIX Assoc. (Nov 9-10, 1987)

[13] Aral, Z. and Gertner, I., *High-Level Debugging in Parasight*, Proceeding of Workshop on Parallel and Distributed Debugging, published as SIGPLAN Notices, V24, No. 1, pp151-162, January 1989.

[14] Bates, P., *Debugging Heterogeneous Distributed Systems Using Event-BAsed Models of Behavior*, Proceeding of Workshop on Parallel and Distributed Debugging, May 5-6, 1988, published as ACM SIGPLAN Notices, V24, No. 1, pp 11-22, January 1989.

[15] Spezialetti, M. *An Approach to Reducing Delays in Recognizing Distributed Event Occurrences*, Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, May 20-21, 1991, published as ACM SIGPLAN Notices, V26, No. 12, December 1991.

Author Information

Paul Maybee is a software engineer with Solbourne Computer, Inc. He is the principle designer of Solbourne's debugger products including pdb and NeD. Prior to joining Solbourne, Paul attended the University of Colorado where he received a Masters degree in Computer Science. Reach him via U.S. Mail at Solbourne Computer, Inc., 1900 Pike Rd. Longmont, CO 80501, or electronically at paulm@solbourne.com.

Appendix A: NeDtcl grammar

This grammar defines the NeDtcl extensions to tcl. All non-chain rule productions correspond to tcl lists. Thus a production with n terms on the right hand side refers to a tcl list of length n (unless n = 1). Terminals appear in uppercase, non-terminals in lowercase. All non-terminals ending in "_name" are chained to STRING, these productions do not appear. Functions can complete successfully or fail. Functions are listed with the parameters types each expects. Each function is followed, optionally, by the type of value each returns when it completes successfully and the type of event it generates.

Embedded Functions

```
command = proc_cmd | scope_cmd | obj_cmd | expr_cmd | sig_cmd | inst_cmd
                  | exec_cmd | info_cmd

proc_cmd =                                  ### process/stack commands
        "stacks"                            # returns stackid_list
  |     "depth" stackid                     # returns INT
  |     "location_of" ADDRESS               # returns location
  |     "addr_of" location                  # returns ADDRESS

scope_cmd =                                 ### scope query commands
        "list_of_variables" context         # returns var_list
  |     "list_of_types" context             # returns type_list
  |     "list_of_funcs" context             # returns func_list
  |     "list_of_files"                     # returns file_list
  |     "list_of_classes"                   # returns class_list

obj_cmd =                                   ### object command
                "load_obj" file_name        # generates object_load

expr_cmd =                                  ### expression evaluation commands
        "proc_eval" location stackid stackdepth expression template async
                                            # returns value, generates expr_eval
  |     "set_template" type_name context template # returns BOOLEAN
  |     "get_template" type_name context    # returns template

sig_cmd =                                   ### signal processing commands
        "handle" signal_name sighow         #
  |     "signal" signal_name                #
  |     "sigstate"                          # returns signal_list

inst_cmd =                                  ### instrumentation commands
        "set_break" ADDRESS condition expression break_action
                                            # returns break_id
  |     "set_watch" ADDRESS size condition expression break_action
                                            # returns break_id
  |     "catchfork"                         # returns break_id
  |     "delete_break" break_id             #
  |     "get_breaks"                        # returns break_list

exec_cmd =                                  ### process execution commands
        "attach" pid                        #
  |     "coreattach" corefile_name objectfile_name# returns INT
  |     "detach"                            #
  |     [ "Stepi" | "Nexti" | "Step" | "Next" | "Cont" | "Finish" ]
                    stackid signal_name# generates proc_state

info_cmd =                                  ### informational commands
        "config" config_kind                # returns config_list
  |     "checkstate" async                  # returns proc_desc,
                                            # generates proc_state
  |     "event_pass" event                  #
  |     "interp" interp_kind address n_a n_b  # returns interp_list
  |     "set_param"      param_name      param_value
```

Events

```
event = object_load | expr_evaluated | proc_state | trace
object_load = "objload" time                     ### object loading is complete
expr_evaluated =                                 ### expression eval is complete
        "expreval" location stackid stackdepth expr_value type_name
proc_state =                                     ### process state change
        [ "stopped" | "running" | "terminated" | "unattached" ]
new_process = "newproc" pid host_name port ### new process to debug
trace =                                          ### tracepoint executed
        "tracemsg" location stackid stackdepth expr_value type_name
```

Data Types

```
aggregate_temp = "AGGREGATE" field_temp    ### struct/union template
aggregate_value =                          ### struct/union expression value
        "AGGREGATE" agg_name field_value*
async = BOOLEAN                            ### asynchronous notification flag
break_action = "STOP" | "GO"               ### stop==breakpoint, go==tracepoint
break_desc =                               ### breakpoint description
        [       "BREAKPOINT" normal_break_desc |
                "WATCHPOINT" watch_desc |
                "FORKPOINT" ] break_id
break_id = INT                             ### instrumentation identifier
break_list = break_desc*                   ### breakpoint description list
class_list = class_name*                   ### class name list
condition = expression                     ### conditional expression
config_kind = "registers" | "signals"      ### configuration types
context =                                  ### scope description
        scope_type scope_name | "LOCATION" location
expr_value = expression value              ### result of evaluating an expression
expression = STRING                        ### C/C++/Fortran expression
field_temp =                               ### struct/union field template
        field_name print_condition template
field_value = field_name value             ### struct/union field expr value
file_list = source_file*                   ### list of source files
func_list =                                ### list of functions in source
        function_name file_name path_name line_num ADDRESS
interp_desc = intruction_desc              ### memory interpretation
interp_kind = "INSTR"                      ### how memory should be interpreted
interp_list = interp_desc*                 ### list of memory interpretations
instruction_desc =                         ### instruction description
        location instr_tranlation symbolic_name
instr_tranlation = STRING                  ### instruction disassembly
line_number = INT                          ### source line number
location =                                 ### source location
        file_name line_number function_name
n_a = INT                                  ### number of translations
                                           ### after address
n_b = INT                                  ### number of translations
                                           ### before address
normal_break_desc =                        ### breakpoint description
        ADDRESS condition expression break_action
pid = INT                                  ### UNIX process id
port = INT                                 ### inet port number
print_condition = expression               ### conditional expression for
                                           ### templates
print_kind =                               ### simple print formats
        "dec" | "hex" | "oct" | "uns" | "bin" |
        "tim" | "asc" | "ins" | "flt" | "spe"
proc_desc = STRING                         ### description of process being
```

```
regclass_list = register_class_name*        ### list of register classes
regname_list = register_name*               ### list of register names
sig_state = signal_name sighow              ### how signal will be handled
sighow = "IGNORE" | "CATCH"                 ### signal handling types
signal_list = sig_state*                    ### list of signal handlings
simple_temp = "SIMPLE" print_kind*          ### simple type template
simple_value =                    "SIMPLE"  ### simple type value
        ["NUMBER" | "PTR" | "MSG" | "ERROR" |
        "VOID" | "ENUM" | "COMPLEX" | "STRING"] STRING
size = INT                                  ### number of bytes
scope_type =                                ### kinds of scopes
        ["GLOBAL" | "CLASS" | "FILE" | "FUNCTION" ]
source_file = file_name path_name           ### source file description
stackdepth = INT                            ### number of frames on call stack
stackid = INT                               ### identifier for call stack
storage_class =                             ### variable storage classes
                "local" | "global" | "static" | "param" | "refparam" |
                "resparam" | "register" | "regparam" | "unknown" |
                "classmem" | "common"
template = simple_temp | aggregate_temp     ### value print templates
time = INT                                  ### object file time stamp
type_list = type_name*                      ### list of program types
value = simple_value | aggregate_value      ### expression value
var_list = variable*                        ### list of program variables
variable = variable_name storage_class      ### variable name and class
watch_desc =                                ### description of a watchpoint
        ADDRESS size condition expression break_action
```