Debugging with UDB 1.5

User's Guide and Reference Manual

Ziad Al-Sharif

Unicon Technical Report #10

January 8, 2009

Abstract

This report is the primary user documentation for the UDB source-level debugger for Unicon and Icon. UDB is written in Unicon and uses the execution monitoring facilities. It combines a traditional debugging interface with many novel features. By design, most of UDB's commands resemble those of GDB. UDB's underlying event-driven architecture empowers UDB with advanced debugging techniques such as 1) more powerful watchpoints, 2) tracepoints based on suspicious execution behaviors, and 3) an outstanding extensibility provided by the IDEA architecture. Experienced users may write their own custom debugging agents, test them as standalone programs, and use them on the fly during UDB debugging sessions or incorporate them into UDB's source code as permanent features. UDB is still under active development; this report is preliminary but reflects the current implementation in the Unicon language distribution.

University of Idaho

Moscow, ID, 83844

This work was supported in part by an appointment to the National Library of Medicine Research Participation Program. This program is administered by the Oak Ridge Institute for Science and Education for the National Library of Medicine.

Table of Contents

1. About UDB

UDB is the source-level debugger for Unicon [2] and Icon [3]. UDB's main goal is to allow you to see what is going inside your program during its execution or to see what your program was doing when it crashed. In UDB, you can run your program. If it is ever stopped, you will be able to examine what has happened, and you may change things to fix the problem or to affect its behavior.

Icon/Unicon's high level advanced features such as string scanning, dynamic typing, dynamic data structures, generators, and goal directed evaluation introduces new kinds of bugs and needs special debugging techniques. UDB loads your icode executable into the same virtual machine; it analyzes the loaded icode in order to obtain information such as a complete list of source files in use; including library files. When an icode is loaded, UDB knows all global variables, procedures, built-in functions, records, classes and their methods, and all packages and their global variables, classes, and procedures.

UDB features the classical debugging techniques found in a conventional debugger such as GDB [4]. At the same time, it has a rich set of advanced debugging features such as 1) extensive execution behavior tracer, which allows you to trace specific behaviors of procedures, built-in functions, and language operators, and 2) outstanding watchpoints, which are customized to serve the language features such as dynamic typing and string scanning; without ignoring the conventional watchpoints found in other debuggers.

UDB also utilizes an extensible debugging architecture called IDEA (Idaho Debugging Extension Architecture). IDEA gives UDB the ability to employ extensions such as automatic debugging, dynamic analyses, and visualization tools, on the fly from any point during the debugging session. Compatible standalone execution monitoring and debugging tools can be loaded on the fly into the debugging session or migrated to the debugger source code as permanent features.

UDB is written on top Unicon's AlamoDE (Alamo Debug Enabled) facilities. Alamo [1] stands for A Lightweight Architecture for Monitoring; it is a monitoring framework built into Unicon's virtual machine. AlamoDE is a recent extension to provide the event-based debugging needs. AlamoDE provides an in-process debugging architecture with no intrusion on the buggy program space. It is important for a debugger to avoid perturbing applications behavior such as garbage collection due to a shared heap.

2. Getting In and Out of UDB

Once UDB is started, it provides a console as an interface between you and the debugging session until you exit it.

2.1. Invoking UDB

To invoke UDB, run the program udb in one of the following formats:

udb

This starts a UDB session without any loaded program; if you want to load or reload a program into udb after udb is already started, the command load must be used. The command

udb *program*

starts a UDB session and loads the executable named *program* at the same time.

2.2. Quitting UDB

To exit UDB type the command quit at any point during the debugging session.

quit
q

This exits UDB; if your program is running, a question will show up asking your permission to quit UDB, if you are sure you want to exit, jut type y or ENTER, otherwise type n to ignore the quit command.

3. The Console

The UDB console provides an interface between you and the UDB session. You can repeat any UDB command by pressing just ENTER. You can also use the UP/DOWN keys to find a command from the current session command history.

3.1. Command syntax

UDB commands are formed on a single line of input. A command usually starts with a command name, followed by zero or more *arguments*. Arguments are interpreted based on the command itself; each command has a different set of possible arguments. A blank line as input to UDB means to repeat the most recent command.

3.2. Getting help

You can always ask UDB for information on its commands, using the help command.

help
h

This command prints a list of all possible commands in the current debugging situation, the list includes a brief description of each class of commands.

help *class*
h *class*

This prints a complete list of all commands that are supported by this class of commands. The command classes are: break, watch, trace, step, info, and print. MORE

help *command*
h *command*

This prints a complete description of the *command* and its possible arguments.

4. Running Programs under UDB

When you run a program under UDB, no recompilation of the program is needed. You just load your program under UDB and start the debugging session.

4.1. Loading your program

You may load your program when you start UDB, see section 2.1. If you did not, from within UDB you can load/reload your program, at any time, using the load command.

load *program*

Loads the executable program named *program* into the UDB session. If a program is already loaded and running, you will be prompted to see if you are sure you wish to discard the current run.

4.2. Starting your program

Use the run command to start your program under UDB. You may specify the argument list, if any.

run
r

This runs the loaded program under UDB. It can be used also to re-run the program at any time during the UDB session.

The command

run *arguments*
r *arguments*

runs the loaded program with the argument list *arguments*; if the program is running or was running previously, then it will re-run the program with this new list of *arguments*.

4.3. Your program's input and output

By default, UDB switches the terminal to its own terminal modes to interact with you. The program you run under UDB does its inputs and outputs to the same terminal that UDB is using.

5. Stopping and Continuing

One of the main goals of a source-level debugger is to stop and inspect your program at various points before it terminates. While your program is running under UDB, it will stop if it reaches a breakpoint, a watchpoint, a tracepoint, or the right line after the step, next, or finish commands. Stopping the program in the middle of the execution allows you to investigate any trouble and find its cause. You may use any of the UDB commands to examine and/or change the state of your program, and set and/or remove breakpoints, watchpoints, or tracepoints before you continue the execution.

5.1. Breakpoints

Breakpoints are source code locations where you direct the program to stop its execution, so you can investigate. This section shows you how to make the program stop at a source code location such as a *line number* or an *entry to a procedure or method*. Furthermore, it shows you how to manage breakpoints using the enable, disable, clear, and delete commands. Finally, it shows you how to check your breakpoints status.

5.1.1. Setting breakpoints

The break command is used to set breakpoints on line numbers and at the entry of a procedure or method.

break *line*
b *line*

This sets a breakpoint on *line*. If the source line is empty or is commented out, then it prints out an error message. If the program is not running yet, the command selects *line* within the file that has the procedure main(). If the program is stopped, then the command specifies *line* within the current file where the execution is stopped. For example: break *999*; if the program is not running yet, it sets a

breakpoint on line *999* in the file that has the procedure main(), otherwise, it sets a breakpoint on line *999* of the current file where the execution is stopped.

break *file line*
b *file line*

This sets a breakpoint on *line* in the specified *file*; only when that *line* has a valid executable code. For example: break *sort.icn 50*; sets a breakpoint on line *50* of the *sort.icn* file. It is not necessary to include the ".icn" in the file name; the previous command can be applied as follows: break *sort 50*.

break *procedure*
b *procedure*

This searches the source code for the name of *procedure*. If there is one, it sets a breakpoint at the entry of *procedure*, otherwise, it prints out an error message. All source files in use are searched; including library files. For example: break *foo*; sets a breakpoint at the entry to procedure foo().

5.1.2. Enabling/Disabling breakpoints

Breakpoints are enabled by default at insertion time. The user can disable any breakpoint during the session using the disable command. Disabled breakpoints are marked as *disabled* when the info breakpoints command is applied. Disabled breakpoints do not function until they are re-enabled. The user can re-enable any of the disabled breakpoints using the enable command. For example:

disable breakpoints
disable break

disables all breakpoints set during the current session, while

disable break *id*

disables the breakpoint with the number *id*. For example: disable break *3*; disables breakpoint number *3*.

Either of the commands:

enable breakpoints
enable break

enables all breakpoints during the current session, while

enable break *id*

enables the breakpoint with the number *id*. For example: enable break *3*; enables breakpoint number *3*.

5.1.3. Clearing/Deleting breakpoints

Breakpoints can be cleared and removed permanently from the list of breakpoints using the clear command, or they can be marked as *deleted* using the delete command. Unlike deleted breakpoints, cleared breakpoints will not show in the "info breakpoints" command. Unlike cleared breakpoints, deleted breakpoints can be seen as part of the debugging session history. However, deleted and cleared breakpoints cannot be reactivated.

5.1.3.1. Clearing breakpoints

The command clear is used to clear breakpoints permanently. Either of

clear breakpoints
clear break

clears all breakpoints set during the current session. The command

clear break *line*

clears the breakpoint set on *line*. If the execution of the program is stopped, *line* is assumed to be in the current file, otherwise, *line* is assumed to be in the file that has the procedure main(). For example: clear break *50*; clears the breakpoint set on line *50* from the current file where the execution is stopped, or from the file that has the procedure main().

The command

clear break *file line*

clears the breakpoint set on *line* in *file*. For example: clear break *sort.icn 50*; clears the breakpoint set on line *50* in the *sort.icn* file.

clear break *procedure*

Clears the breakpoint set at the entry to *procedure*. For example: clear break *foo*; clears the breakpoint set at the entry to procedure *foo*.

5.1.3.2. Deleting breakpoints

Another way to delete breakpoints is by using the delete command. Unlike cleared breakpoints, deleted breakpoints are not removed but marked as *deleted*; the user will be able look them up using the "info breakpoints" command. Either of

delete breakpoints
delete break

deletes all breakpoint set during the current session.

The command

delete break *id*

deletes the individual breakpoint with the number *id*. For example: delete break *5*; deletes the breakpoint number 5.

5.1.4. Breakpoints info

A user can query about all breakpoints or a specific breakpoint using the info command. The user might be interested in information about a breakpoint such as its id number, source code location that includes the file name and line number, and whether it is enabled, disabled or deleted. Either of

info breakpoints
info break
break
b

prints a complete list of all breakpoints information such as their id, location and status. The command

info break *id*

prints information about the specific breakpoint with the number *id*. The command

info break *file*

prints information about all breakpoints set in *file*.

5.2. Watchpoints

UDB's watchpoints enable you to observe the evaluation of some expressions such as 1) a variable being assigned, 2) read, 3) assigned and the new value is different from the previous value, 4) assigned and the type of the new value is different from the type of the previous value, 5) a keyword explicitly assigned by the program's code, or 6) an implicit string scanning environment (*&subject* and *&pos* keywords) is changed by the string scanning primitives.

Watchpoints may cause the program to stop, or they may work silently collecting information about specific evaluation(s). Silent watchpoints need to be explicitly set using the "*–silent*" flag. Silent watchpoints collect location and value about specific evaluations without pausing program's execution at every incident. The user may review collected information at any point during or after the execution using the command "info watch *id*".

Furthermore, regardless of the watchpoint being silent or not, the user is able to set a watchpoint for a limited number of satisfied incidents. An integer *count* flag is used as a counter that empowers watchpoints with control over the number of times an action should be observed. If *count* is positive, the watchpoint observes only the first *count* number of satisfied incidents. However, if *count* is negative, the

watchpoint observes every incident, but keeps track of the last *count* number of them (the very recent *count* number of incidents), which they also can be looked up, by the user, during or after the execution. Most watchpoints support relational operations such as (=, ~=, <, >, <=, >=), which they can be applied on the value or type of the observed variable or keyword.

Watched variables can be mangled with their scope (procedure name). UDB utilizes special characters, which are used by the Alamo framework to determine the scope and the kind of local parameters. The "–" is used for normal locals, the "^" is used for parameters, and the ":" is used for statics. Global variables always can be distinguished using the "+" character attached to the end of the variable name; i.e. g+ is the global variable g. For example: a-main: is the normal local variable a in procedure main, p^foo: is the parameter p to procedure foo, and s:bar is the static variable s in procedure bar. Watchpoints on mangled variables do not have to be active somewhere on the stack.

If the program is NOT running (not stopped), the user can set watchpoints on valid keywords, global variables, and local variables that are mangled with their scope. If the program is stopped, the user can set watchpoints on valid keywords, locals that are mangled with their scope name, locals that are not mangled but live in the currently selected stack frame, and of course global variables. UDB is able to verify locals that are mangled and their procedures are active on the calling stack. However, if the procedure is not active, UDB trusts the user's intuition assuming the scope is right. If the variable is not mangled, UDB automatically resolves the scope based on the currently selected stack frame and the current execution state. By default, when a plain variable is specified by the watchpoint, UDB checks whether it is a keyword, if not UDB looks it up in the currently selected stack frame, if not, then UDB looks it up in the global variables, if not, then it complains with an error message.

UDB provides five types of watchpoints: 1) assignment watchpoints; awatch, 2) read watchpoints; rwatch, 3) value change watchpoints; vwatch, 4) type change watchpoints; twatch, and 5) string scanning watchpoints; swatch.

5.2.1. Assignment watchpoints

Assignment watchpoints are set on variables or keywords whenever assigned explicitly in the program's source code. The awatch command is used to set this type of watchpoints. The awatch command has the following syntax:

awatch [-*silent*] [*count*] *variable* [[=|~=|>|<|>=|<=] *value*]

The command:

awatch *variable*

sets a watchpoint on *variable* whenever it is assigned. For example: awatch *a*; sets an assignment watchpoint on variable *a*.

The command

awatch *variable >= value*

sets a watchpoint on *variable* whenever it is assigned and the new *assigned-value >= value*. For example: awatch *x >= 1*; sets a watchpoint on variable *x* that is triggered whenever it is assigned with a *value >= 1*. Another example: awatch *k == "hi this is udb"*; sets a watchpoint on variable *k* that is triggered whenever it is assigned the value *"hi this is udb"*. The command

awatch *count variable*

sets a watchpoint on *variable* for the first *count* number of times it is assigned. For example, awatch *5 a*; sets a watchpoint on variable *a* for the first *5* times it is assigned. In contrast, the command

awatch *-count variable*

sets a watchpoint on *variable* and keeps track of the last *count* number of times it is assigned. For example,
awatch *-5 a*; sets a watchpoint on variable *a* whenever it is assigned and keeps information about the last *5* assignments. The command

awatch *–silent variable*

sets a silent watchpoint on *variable* whenever assigned. Silent watchpoints collect information about every incident without stopping the execution. The user can use the "info awatch *variable*" or "info awatch *n*" to look up whatever information that watchpoint has collected. For example: awatch *–silent &pos*; sets a silent watchpoint on the keyword *&pos*; the execution will not stop at every incidents *&pos* get assigned explicitly in the code, but the user can use the command "info awatch *&pos*" to look up the collected information. The command

awatch *–silent –count variable*

sets a silent watchpoint on *variable* whenever it is assigned and keeps information about the last *count* number of assignments. For example: awatch *–silent -10 x*; sets a silent watchpoint on the variable *x* that is triggered whenever it is assigned and it keeps information about the most recent *10* incidents. Note: the execution will not stop at every incident, but the user can use the command "info awatch *x*" to look up collected information.

5.2.2. Read watchpoints

In situations where you are only interested in a variable or keyword being referenced or read; the rwatch command is very helpful. It allows you to set a *normal* or *silent* watchpoint on various reading conditions. The rwatch command has the following syntax:

rwatch [*-silent*] [*count*] *variable* [[=|~=|>|<|>=|<=] *value*]

For example, the command

rwatch *variable*

sets a read watchpoint on *variable* whenever it is read (dereferenced). For example: rwatch *x*; sets a *read watchpoint* on variable *x* that stops the execution whenever it is read. The command

rwatch *variable* >= *value*

sets a read watchpoint on *variable* whenever it is read and *its value* >= *value*. For example, rwatch *x* <= *100*; sets a read watchpoint on variable *x* that stops the execution whenever *x* is read and *x-value* <= *100*.

The command

rwatch *count variable*

sets a read watchpoint on *variable* for the first *count* number of times it is read. For example: rwatch *5 x*; sets a read watchpoint on variable *x* for the first *5* times it is read that observes and stops the execution of the first 5 times *x* is read. In contrast, the command

rwatch -*count variable*

sets a read watchpoint on *variable* and keeps track of the last *count* number of times it is read. For example, rwatch -*5 x*; sets a read watchpoint on variable *x* that stops the execution whenever *x* is read and it keeps information about the last *5* read incidents; the user can look up these most recent *5* incidents any time during or after the execution using the "info rwatch *x*" command.

The command

rwatch –*silent variable*

sets a silent read watchpoint on *variable* whenever it is read. Silent read watchpoints keeps information about every read incident without stopping the execution. The user can use the "info rwatch *variable*" or "info rwatch *n*" to look up whatever information that read watchpoint has collected. For example: rwatch –*silent x*; sets a silent read watchpoint on variable *x*, this watchpoint will NOT stop the exaction whenever *x* is read, instead it collects information that the user can look up any time during or after the execution using the "info rwatch *x*" command.

5.2.3. Value change watchpoints

In some situations the traditional watch on assignment is not good enough; sometimes the user is interested in the variable when the value has really changed after the assignment. The vwatch command provides easy normal or silent value watchpoints on variables whenever they are assigned and the new value is really different from the previous values.

The syntax of the vwatch command looks exactly like the awatch command:

vwatch [-*silent*] [*count*] *variable* [[=|~=|>|<|>=|<=] *value*]

For example, the command

vwatch *variable*

sets a value watchpoint on *variable* whenever it is assigned and the *variable*'s value is really been changed.

The command

vwatch *variable >= value*

sets a value watchpoint on *variable* whenever it is assigned, the assignment has changed its value, and its *new-value >= value*. For example, vwatch *a ~= 100*; sets a watchpoint on variable *a* whenever its value is changed to something *~= 100*.

The command

vwatch *count variable*

sets a watchpoint on *variable* for the first *count* number of times it is assigned a changed value. For example, vwatch *5 a*; sets a value watchpoint on variable *a* for the first *5* times its value is changed. In contrast, the command

vwatch *-count variable*

sets a value watchpoint on *variable* and keeps track of the last *count* number of times its value is changed. For example, vwatch *-5 a*; sets a value watchpoint on variable *a* whenever its value is changed and keeps information about the last *5* assignments. The user can inquire about these very recent *5* incidents during or after the execution using the "*info vwatch a*" command.

The command

vwatch *–silent variable*

sets a silent watchpoint on *variable* whenever assigned its value changes. Silent watchpoints keep information about every incident without stopping the execution. The user can use the "info vwatch *variable*" or "info vwatch *n*" to look up whatever information that watchpoint has collected. For example vwatch *–silent x*; sets a value watchpoint on variable *x*. This watchpoint will not stop the execution and the user can query the collected information using the "info vwatch *x*" command.

5.2.4. Type change watchpoints

In a dynamically typed language such as Unicon, a variable can be assigned with any type of values during the course of execution. It can be very important to watch variables that change type. UDB provides the twatch command specifically for this purpose.

The syntax of twatch looks exactly like the awatch command, but the optional condition is limited to the = and ~= operations. Valid Unicon types are integer, real, string, list, table, set, cset, record, function, and procedure.

twatch [-*silent*] [*count*] *variable* [[=|~=] *type*]

For example, the command

twatch *variable*

sets a type watchpoint on *variable* whenever it is assigned a value that changes *variable*'s type. For example: twatch *x*; sets a type watchpoint on variable *x* whenever it changes its type. i.e. if the source code has something equivalent to *x := 10; …. ; x := 10.5*; then this will fire a type watchpoint if twatch *x* is enabled. The command

twatch *variable = type*

sets a watchpoint on *variable* that is triggered whenever its type is changed to *type*. For example: twatch *a = integer*; sets a watchpoint on variable *a* whenever it is assigned with a value from the *integer* type. Another example: twatch *x ~= string*; sets a type watchpoint on variable *x* whenever assigned with a value that is not from the *string* type.

The command

twatch *count variable*

sets a watchpoint on *variable* for the first *count* number of times it is assigned a change in type. For example: twatch *5 a*; sets a type watchpoint on variable *a* for the first *5* times it changes its type. In contrast, the command

twatch -*count variable*

sets a watchpoint on *variable* and keeps track of the last *count* number of times its type is changed. For example: twatch *-5 a*; sets a watchpoint on variable *a* whenever its type is changed and keeps information about the last *5* incidents. The user can query those last *5* incidents using the command "info twatch *a*".

The command

twatch *–silent variable*

sets a silent type watchpoint on *variable* that is triggered whenever its type changes. Silent watchpoints keep information about every incident without stopping the execution. The user can use the "info twatch *variable*" or "info twatch *n*" to look up whatever information that watchpoint has collected. For example: twatch *–silent -10 x*; sets a silent type watchpoint on *x* whenever changed its type and keeps track of the last *10* times that watchpoint is satisfied. The user can inquire those very recent *10* incidents during or after the execution using the "info twatch *x*" command.

5.2.5. String scanning watchpoints

Unicon/Icon richness with the string scanning simplicity may require special type of watchpoints dedicated to string scanning environment. The swatch command provides simple interface to set a watchpoint on the string scanning environment. The syntax of the swatch is as follows:

swatch [*-silent*] [*count*]

For example, the command

swatch

sets a watchpoint on the string scanning environment, in particular the keywords *&subject* and *&pos* and their implicit change. The command

swatch *–silent*

sets a silent watchpoint on the string scanning environment, in particular the keywords *&subject* and *&pos* and their implicit change. The user will not be notified by every change incident, instead he/she is able to inquire collected information during or after the execution using the command "info swatch".

The command

swatch *count*

sets a watchpoint on the string scanning environment for the first *count* number of times an implicit change has been made to the keywords *&pos* and *&subject*. In contrast, the command

swatch *-count*

sets a watchpoint on string scanning environment and keeps track of the last *count* number of times the keyword *&pos* and *&subject* have been implicitly changed. The keyword *&subject* is changed implicitly whenever the execution of the program initiates new string scanning environment. On the other hand, the keyword *&pos* is implicitly changed by any of the string scanning environment primitives such as move(), and tab().

The command

swatch –*silent*

sets a silent watchpoint on string scanning environment and keeps track of ever incident any of the keywords *&pos* and *&subject* have been implicitly changed. The user can use the "info swatch" to look up whatever information this watchpoint has collected. For example, swatch –silent *10*; sets a silent watchpoint on string scanning environment and keeps track of the first *10* incident where any of the keywords *&pos* and *&subject* have been implicitly changed. In contrast, swatch –silent *-10*; sets a silent watchpoint on string scanning environment and keeps track of the most recent *10* incident where any of the keywords *&pos* and *&subject* have been implicitly changed.

5.2.6. Enabling/Disabling watchpoints

The user can manage watchpoints by enabling, disabling, deleting, or clearing them. The info command allows the user to check the state of watchpoints. In situations where watchpoints are not needed for a while, the user can put the watchpoint to sleep by disabling it. Disabled watchpoints can be enabled by the user at any point during the execution. All newly set watchpoints are enabled. The commands disable and enable are used to disable and enable watchpoints respectively.

5.2.6.1. Disabling watchpoints

The command disable is used to disable watchpoints. Either of

disable watchpoints
disable watch

disables all watchpoints of all kinds set during the current session. The commands

disable awatch, disable rwatch, disable twatch, disable vwatch, and disable swatch

disable all assignment watchpoints, read watchpoints, type change watchpoints, value change watchpoints, and string scanning watchpoints respectively. The command

disable watch *id*

disables the watchpoint with the number *id*. For example: disable watch *2*; disables the watchpoint with the *id* number *2*.

5.2.6.2. Enabling watchpoints

The command enable is used to enable disabled watchpoints. Either of

enable watchpoints
enable watch

enables all disabled watchpoints set during the current session. The commands

enable awatch, enable rwatch, enable twatch, enable vwatch, and enable swatch

enables all assignment watchpoints, read watchpoints, type change watchpoints, value change watchpoints, and string scanning watchpoints respectively. The command

enable watch *id*

enables watchpoint with the number *id*. For example: enable watch *2*; enables the watchpoint with the *id* number *2*.

5.2.7. Clearing/Deleting watchpoints

Unlike disabling watchpoints, deleting watchpoints marks the watchpoint as *deleted*, which means it cannot be enabled after that. However, the user is still able to see those deleted watchpoints using the "info watch" command. On the other hand, the command clear is used to delete watchpoints and never see them again.

5.2.7.1. Clearing watchpoints

The command clear is used to clear watchpoints permanently. Either of

clear watchpoints
clear watch

clears all watchpoints set during the current session; all watchpoints will be removed and cannot be seen again. The command

clear awatch, clear rwatch, clear twatch, clear vwatch, and clear swatch

clears all assignment watchpoints, read watchpoints, type change watchpoints, value change watchpoints, and string scanning watchpoints respectively. The command

clear watch *id*

It clears the watchpoint with the number *id*. For example: clear watch *7*; clears the watchpoint with the *id* number *7*.

5.2.7.2. Deleting watchpoints

The command delete is used to mark a watchpoint as *deleted*. Either of

delete watchpoints
delete watch

deletes all (*enabled* and *disabled*) watchpoints set during the current session. The commands

delete awatch, delete rwatch, delete twatch, delete vwatch, and delete swatch

deletes all assignment watchpoints, read watchpoints, type change watchpoints, value change watchpoints, and string scanning watchpoints respectively.

The command

delete watch *id*

deletes the watchpoint with the number *id*. For example: delete watch *7*; deletes the watchpoint with the *id* number *7*.

5.2.8. Watchpoints info

The user can inquire about the watchpoints set during the current session. Either of

info watchpoints
info watch
watch

prints a list of all watchpoint set during the current session and not cleared. The commands

info awatch, info rwatch, info twatch, info vwatch, and info swatch

prints a complete list of all assignment watchpoints, read watchpoints, type change watchpoints, value change watchpoints, and string scanning watchpoints respectively. The command

info watch *id*

prints detailed information about the watchpoint with the number *id*. For example: info watch *6*; shows the watchpoint with the *id* number *6*. The command

info watch *variable*

prints a list of all watchpoints set on *variable*. For example: info watch *x*; shows all watchpoints on variable *x*.

5.3. Tracepoints

Tracepoints can be seen as an extension over traditional breakpoints and watchpoints found in conventional debuggers. Using execution behavior tracing, a user is able to stop the execution based on potential behaviors such as the type of the returned value from a user-defined procedure, built-in function, and language operator. This type of tracing provides an additional lightweight tool and flexibility in order to simplify and speed up the process of discovering bug locations.

Behaviors are divided into two categories: 1) general behaviors, which are described by the words start and end, and 2) detailed behaviors, which are used to describe more details about the start and end. The

start behavior can be broken down into call and resume, whereas the end behavior is broken down into return, suspend, fail, and remove. Behaviors are associated with the semantics of the Unicon/Icon language:

- call: represents normal procedure, built-in function, or operator call
- resume: represents the behavior of resuming a suspended procedure, built-in function, and operator
- return: represents the behavior of exiting a procedure with the language keyword return. For built-in functions and operators return: represents the behavior of finishing a successful call
- fail: represents the behavior of exiting a procedure with the language keyword fail or reaching the end of the procedure. For built-in functions and operator fail: represents the behavior of failing to accomplish the intended job
- suspend: represents the behavior of suspending with the language keyword suspend
- remove: represents the behavior of removing a suspended procedure, built-in function, or operator as a result of exiting a parent procedure, built-in function, or operator
- start: represents the general call or resume of a procedure, built-in function, or operator
- end: represents the general return, fail, suspend, and remove of a procedure, built-in function, or operator.

In particular, the return behavior is applicable for extra operations such as =, ~=, <, >, <=, and >=. Those operations are used to specify an extra condition on the returned value from all of user-defined procedures, built-in functions and operators. If the –silent flag is provided, the tracepoint does not stop the execution, but the user will be able to check the traced info from any point during or after the execution. When *count* is provided; if *count > 0*, the tracepoint traces the first *count* number of times the trace condition is satisfied, otherwise, if *count < 0*, it traces every satisfying incident, but keep track of the most recent *count* number of them.

5.3.1. Procedure tracepoints

Procedures and methods that are defined by the programmer or the one that are provided by the Unicon/Icon library are source code that can be debugged by UDB. Very often during the debugging process, a user may become interested in the behavior of a set of procedures such as their execution behavior and their returned or suspended values. In such cases, UDB provides simple tracing facilities that are easy to use. The syntax of the procedure tracing command is as follows:

trace [*-silent*] [*count*] *procedure* [*behavior* [*op value*]]

The trace command sets a tracepoint on *procedure* whenever the specified *behavior* is satisfied. If *behavior* is not specified, all *behavior*s are traced. If *–silent* flag is not provided, the tracepoint stops the execution at every incident, otherwise, the tracepoint is silent and the user is able to look up all previous values during or after the execution.

The command

trace *procedure*

sets a tracepoint on all *procedure* activities (*behaviors*), such as *call*, *resume*, *suspend*, *return*, *remove*, and *fail*. The execution will be stopped at every incident to notify the user about the behavior. For example: trace *foo*; sets a tracepoint on *procedure foo* whenever it is *called*, *resumed*, *suspended*, *returned*, *removed*, and *failed*.

The command

trace *procedure fail*

sets a tracepoint on *procedure* whenever it is *failed*. For example: trace *bar fail*; sets a tracepoint on *procedure bar* whenever it is failed. The command

trace *procedure return < value*

sets a tracepoint on *procedure* whenever it returns a value < *value*. For example: trace *bar return <= 0*; sets a tracepoint on *procedure bar* that is triggered whenever it returns a numeric *value <=0*. The command

trace *count procedure return = value*

sets a tracepoint on *procedure* for the first *count* number of times it returns a value = *value*. For example:
trace *10 bar return ~= 0*; sets a tracepoint on *procedure bar* for the first *10* times it returned a *value ~= 0*. Another example: trace *-10 bar return <= 0*; sets a tracepoint on *procedure bar* that is triggered whenever it returns a *value <=0*; the *-10* indicates that the user will always be able to check the last *10* incidents using the "info trace bar" command.

The command

trace *–silent -count procedure return > value*

sets a silent tracepoint on *procedure* that is triggered whenever it returns a value > *value*. For example: trace *–silent -100 bar return < 0*; sets a silent tracepoint on *procedure bar* that is triggered whenever it returns a value < *value*. This tracepoint keeps track of the most recent *100* incidents. The trace is silent, which means the user will not be notified when an incident is satisfied and the user will be able to look up those traced incidents during or after the execution using the command "info trace bar".

5.3.2. Function tracepoints

UDB's tracepoints go beyond the user-defined procedures and methods. The user is able to set tracepoints on built-in functions that are supported by the runtime system. Built-in functions can be traced in a way similar to procedure tracing.

The syntax of the function tracing is as follow:

trace [*-silent*] [*count*] *function* [*behavior* [*op value*]]

For example, the command

trace *function*

sets a tracepoint on all *function* activities (*behaviors*) such as *call*, *resume*, *suspend*, *return*, *remove*, and *fail*. The execution will be stopped at every incident to notify the user about the behavior. For example: trace *foo*; sets a tracepoint on *function foo* that is triggered whenever it is *called*, *resumed*, *suspended*, *returned*, *removed*, and *failed*. The command

trace *function return*

sets a tracepoint on *function* that is triggered whenever it *return*s. The command

trace *count function return < value*

sets a tracepoint on *function* for the first *count* number of times it a value *< value*. For example, the command trace *10 cos return < 0*; sets a tracepoint on the *cosine* built-in function *cos()* for the first *10* times it returns with a *value < 0*.

The command

trace *–silent -count function return < value*

sets a tracepoint on *function* whenever it a value *< 0*. This tracepoint will keep track of the last *count* number of incidents. The trace is silent, which means the user is not notified when an incident is occurred. The user is able to look up those traced incidents during or after the execution using the "info trace *function*" command. For example: trace *-10 cos return < 0*; traces the *cosine* built-in function *cos()* and keeps track of the last *10* times it returns a *value < 0*. The user is able to look the most recent *10* incidents during or after the execution using the command "info trace cos".

5.3.3. Operator tracepoints

In a goal directed language such as Unicon, operators play a very important role in the semantic of the language and the evaluation of its expressions. For example, the expression (*a < b < c < d*) may succeed and generate the value of *d*, or fail and generate the value of either *a*, *b*, or *c*. In conventional debugging, the user may step at this expression and check the value of each of the involved variables. However, UDB allows a user to place a tracepoint on the < operator that is triggered whenever it fails and finds automatically which variable made this expression to fail. The syntax of the trace operator is as follows:

trace [*-silent*] [*count*] *function* [*behavior* [*op value*]]

For example, the command

trace *operator*

sets a tracepoint on all *operator* activities (*behaviors*) such as *call*, *resume*, *suspend*, *return*, *remove*, and *fail*. The execution will be stopped at every incident. For example, the command trace <; sets a tracepoint on all < activities. Another example: trace < *fail*; sets a tracepoint on the operator < that is triggered whenever it fails.

The command

trace [] *fail*

sets a tracepoint on [] (subscript operator) that is triggered whenever it *fails*.

The command

trace *count operator return < value*

sets a tracepoint on *operator* for the first *count* number of times it returns a value < *0*. For example, the command trace *10 = return < 0*; traces the *equal* operator = for the first *10* times it returns with a *value < 0*. Another example, the command trace *-10 = return < 0*; traces the *equal* operator = every time it returns a *value < 0*, however, the user is able to check the most recent 10 incidents during or after the execution using the command "info trace =". The command

trace *–silent -count operator return = value*

sets a tracepoint on *operator* that is triggered whenever it returns a value equal to *value*. This tracepoint keeps track of the most recent *count* number of incidents. The tracepoint is silent, which means the user is not notified when an incident is occurred. However, the user is able to look up those traced incidents during or after the execution. For example, the command trace *-10 = return < 0*; traces the built-in *equal* operator = and keeps track of the most recent *10* times it returns a value < *0*. The user will be able to look those *10* most recent incidents during or after the execution using the command "info trace =".

5.3.4. Enabling/Disabling tracepoints

The user can manage tracepoints by enabling or disabling them. The info command allows the user to check the state of tracepoints. In situations where a tracepoint is not needed for a while, the user can put the tracepoint to sleep by disabling it. Disabled tracepoint can be enabled by the user at any point during the execution. All newly set tracepoints are enabled. The commands disable and enable are used to disable and enable tracepoints respectively.

5.3.4.1. Disabling tracepoint

The command disable is used to disable tracepoints. Either of

disable tracepoints
disable trace

disables all tracepoints set during the current session.

The command

disable ptrace

disables all procedure tracepoints set during the current session. The command

disable trace *procedure*

disables the specific tracepoint set on *procedure*. For example, disable trace *bar*; disables the tracepoint set on *procedure bar*.

The command

disable ftrace

disables all function tracepoints set during the current session. The command

disable trace *function*

disables the specific tracepoint set on *function*. For example, disable trace *foo*; disables the tracepoint set on *function foo*.

The command

disable otrace

disables all operator tracepoints set during the current session. the command

disable trace *operator*

disables the specific tracepoint set on *operator*. For example, disable trace ==; disables the tracepoint set on *operator* ==.

The command

disable trace *id*

disables the tracepoint with the number *id*. For example, disable trace *3*; disables the tracepoint with the *id* number *3*.

5.3.4.2. Enabling tracepoint

The command enable is used to enable an already disabled tracepoint. Either of

enable tracepoints
enable trace

enables all disabled tracepoints set during the current session. The command

enable ptrace

enables all disabled procedure tracepoints set during the current session.

The command

enable trace *procedure*

enables the specific tracepoint set on *procedure*. For example, enable trace *bar*; enables the disabled tracepoint set on *procedure bar*. The command

enable ftrace

enables all disabled function tracepoints set during the current session. The command

enable trace *function*

enables the specific tracepoint set on *function*. For example, enable trace *foo*; enables the disabled tracepoint set on *function bar*. The command

enable otrace

enables all disabled operator tracepoints set during the current session. The command

enable trace *operator*

enables the specific tracepoint set on *operator*. For example, enable trace ==; enables the disabled tracepoint set on *operator* ==.

The command

enable trace *id*

enables the tracepoint with the number *id*. For example, enable trace *3*; enables the disabled tracepoint with the *id* number *3*.

5.3.5. Clearing/Deleting tracepoints

Tracepoints can be cleared and removed permanently from the list of tracepoints using the clear command, or it can be marked as *deleted* using the delete command. Cleared breakpoints will not show by the "info tracepoints" command.

5.3.5.1. Clearing tracepoints

The clear command is used to clear tracepoints permanently. Either of

clear tracepoints
clear trace

clears all tracepoints set during the current session.

The command

clear ptrace

clears all procedure tracepoints. The command

clear trace *procedure*

clears the specific tracepoint set on *procedure*. For example, clear trace *bar*; clears the tracepoint set on *procedure bar*.

The command

clear ftrace

clears all function tracepoints set during the current session.

The command

clear trace *function*

clears the specific tracepoint set on *function*. For example, clear trace *foo*; clears the tracepoint set on *function foo*.

The command

clear otrace

clears all operator tracepoints set during the current session.

The command

clear trace *operator*

clears the specific tracepoint set on *operator*. For example, clear trace ==; clears the tracepoint set on operator ==. The command

clear trace *id*

clears the tracepoint with the number *id*. For example, clear trace *3*; clears the tracepoint with the *id* number *3*.

5.3.5.2. Deleting tracepoints

Another way to delete tracepoints is by using the delete command. Unlike cleared tracepoints, deleted tracepoints are not removed, instead they are marked as *deleted*; the user will be able look them up using the "info tracepoints" command.

Either of

delete tracepoints
delete trace

deletes all tracepoints set during the current session. The command

delete ptrace

deletes all procedure tracepoints set during the current session. The command

delete trace *procedure*

deletes the specific tracepoint set on *procedure*. For example, delete trace *bar*; deletes the tracepoint set on *procedure bar*. The command

delete ftrace

deletes all function tracepoints set during the current session. The command

delete trace *function*

deletes the specific tracepoint set on *function*. For example, delete trace *foo*; deletes the tracepoint set on *function bar*. The comamnd

delete otrace

deletes all operator tracepoints set during the current session. The command

delete trace *operator*

deletes the specific tracepoint set on *operator*. For example, delete trace *[]*; deletes the tracepoint set on operator *[]*. The command

delete trace *id*

deletes the tracepoint with the number *id*. For example, delete trace *3*; deletes the tracepoint with the *id* number *3*.

5.3.6. Tracepoints info

A user can inquire about tracepoints using the info command. Either of

info tracepoints
info trace
trace

prints a complete list of all tracepoints set during the current session. The command

info ptrace

prints a complete list of all procedure tracepoints set during the current session. The command

info trace *procedure*

prints information about the tracepoint set on *procedure*. For example, info trace *bar*; prints information about the tracepoint set on *procedure bar*.

The command

info ftrace

prints a complete list of all function tracepoints set during the current session. The command

info trace *function*

prints information about the specific tracepoint set on *function*. For example, info trace *foo*; prints information about the tracepoint set on *function foo*.

The command

info otrace

prints a complete list of all operator tracepoints set during the current session. The command

info trace *operator*

prints information about the specific tracepoint set on *operator*. For example, info trace >=; prints information about the tracepoint set on the operator >=.

The command

info trace *id*

prints information about the tracepoint with the number *id*. For example: info trace *3*; prints information about the tracepoint with the *id* number *3*.

5.4. Stepping and continuing

Stepping using the command step means executing just one line of your program source code. Stepping using the command next means continue to the next source line in the current stack frame; next is similar to step but without going inside any procedure or function calls that may appear within the next line of source code. In next, execution stops when control reaches a different line of source code at the original stack level that the control was in when you gave the next command. Continuing using the continue command means resuming program execution until your program completes normally or reaches another breakpoint, watchpoint, or tracepoint that may stop the execution. The commands

step
s

steps one source line in the execution of the program. The commands

step *count*
s *count*

steps *count* number of lines in the source program. The default value of *count* is one.

The commands

next
n

steps one source line of the source program. If that source line has any procedure or function call, the control will not step anywhere inside that call; the called procedure or function will be evaluated without showing any stepping inside. The commands

next *count*
n *count*

steps *count* number of lines in the source program. Stepping each source line will not show any control detail for any procedure or function call, which may occur in these lines.

The commands

continue
cont
c

resumes the program's execution at its normal speed until it terminates or reaches a breakpoint, watchpoint, or tracepoint that may stop the execution.

The commands

```
finish
return
ret
```

completes the execution of the current procedure and returns back to the caller; it steps on the next statement to be executed after the call.

6. Examining the Stack

When a program performs a procedure call, information about the call is generated and saved in the execution stack in a block called the procedure frame. Each frame includes the level, the arguments, and the local variables of the called procedure. The procedure frame is saved on the stack until the procedure is returned. However, when the program is stopped, you may need to know where and/or how it got there; UDB commands help you do that.

By default, when the program stops, UDB points implicitly to the current procedure frame, which is the last frame on the execution stack. When the program is stopped, the current farm is fame number *0*, which is the innermost frame, the oldest frame on the stack has the biggest frame number, which is the frame for `procedure main()`. You can explicitly jump to any frame by making the current frame whichever frame number you are interested in. This is important because there is some UDB commands refer implicitly to the currently selected frame. In particular, whenever you ask UDB for the value of a variable, the variable is supposed to be found in the currently selected frame.

6.1. Information about a frame

You can obtain information, about the current state of the execution stack and its frames, using one of the following commands:

```
frame
f
```

This prints a brief description of the currently selected stack frame plus a list of all formal parameters. When it is used without any argument, this command does not change the currently selected frame. If you want to know how many frame levels your program execution currently has, the command

```
print &level
```

prints the number of levels (number of frames) that are in the execution stack.

6.2. Selecting a frame

You can select a frame from the current execution stack using one of the following commands

frame *n*
f *n*

This prints a brief description of the frame number *n* and changes the currently selected frame by making it pointing at the frame number *n*. Note that *n* must be within the range of the current stack levels. The command

up

moves the current frame pointer one frame up in the execution stack. Assuming the stack is growing downward, if the current frame pointer points at the most oldest stack frame, which is the stack frame of procedure main(), this command will print an error message because there is no more frames to go up on the stack. Furthermore, the command

up *n*

moves the current frame pointer *n* number of frames up on the execution stack. The default value of *n* is one. Note that *n* must be within the range of the current stack levels. The command

down

moves the current frame pointer one frame down on the execution stack. Assuming the stack is growing downward, if the current frame pointer points at the most recent stack frame, this command will print an error message because there is no more frames to go down on the stack. Furthermore, the command

down *n*

moves the current frame pointer *n* number of frames down on the execution stack. The default value of *n* is one. Note that *n* must be within the range of the current stack levels.

6.3. Backtracing

Backtracing allows you to investigate the entire execution stack. It is commonly used after a runtime error. The main command name is backtrace, however, the command has two more synonym: where and bt. Either of

backtrace
where
bt

prints information about every procedure frame on the current execution stack.

The commands

backtrace *n*
where n
bt *n*

prints information about the *n* innermost procedure frames; in other words, it prints the most recent *n* frames currently on the stack. However, if *n* is negative, the printed frames are the oldest *n* fames. The commands

backtrace –*n*
where –n
bt –*n*

prints information about the *n* outermost procedure frames; in other words, it prints the most oldest *n* frames currently on the stack.

7. Examining Data

UDB provides the ability to examine and change data during the execution. The print command is used to either *get a peek or change* any of the variables, keywords, or data structures in the current execution state.

7.1. Getting a peek at the value of an expression

Usually, the print command is used. The target variable can be global, keyword, or local including static, and parameter variables; if the variable is not a keyword or one of the locals within the currently selected stack frame, it is assumed to be global. Either of

print *expr*
p *expr*

prints the value of *expr*. The printed value of *expr* is different based on its type. If *expr* is a variable with an *Atomic Type* such as null, integer, real, cset, or string, then the printed value is the variables current value. Otherwise, if *expr* is a variable with a *Structured Type* such as list, table, record, set, procedure, or window, then the printed value is an *image* or *ximage* of that variable. An *image* of a structure is the internal name of that structure associated with its serial number and the number of elements or fields inside. On the other hand, the *ximage* of a structure is a detailed print of the entire elements in that structure and its substructures.

Furthermore, *expr* may reference some elements or fields of a structure such as an element of a list, or table, a field of record, or a sub-string. The *expr* can utilize unary operators such as * and ! to be used

with applicable variables and keywords. For example, the ! operator can be used to generate and print the complete list of all values generated by a keyword such as &features; (i.e. print ! &features).

The command

print *expr n*
p *expr n*

prints the value of *expr* based on information obtained from the stack frame number *n*. Frame *0* is the innermost frame, frame *1* is the direct parent of the innermost frame, and frame *n* is the frame number *n* away from the currently selected frame. The procedure main() always has the highest frame number.

7.1.1 Simple variables

Simple variables are those with *Atomic Types* such as integer, real, null, and string.

print *variable*

It prints the value of *variable*. For example, print *a*; prints the value of variable *a*.

7.1.2. Lists

UDB provides the ability to print the whole list or to print an element or some contiguous elements in the list. For example, the command

print *L*

prints an *ximage* of the list *L*. The *ximage* includes all elements and sub-elements in *L*. The command

print *L[2]*

prints whatever in the element number *2* in the list *L*. If *L[2]* is a reference of a structure, then this command prints its *image* or *ximage*. The command

print *L[2][10]*

prints the element number *10* from the position *2* of *L*. The command

print *L[i][j]*

prints the element *L[i][j]* after evaluating both of *i* and *j*, note that *i* and *j* must be valid variables in the current execution context. The command

print *L[1:5]*

prints an *image* or *ximage* of all the elements between *1* and *5* of the list *L*.

The command

print *L*

prints the current size of list *L*. The command

print *!L*

prints all elements in the list *L*; the *ximage* of *L*.

7.1.3. Tables

Table subscription is very similar to the list subscription with the extension that tables are not ordered nor restricted to the integer keys. For example, the command

print *T*

prints an *ximage* of the table *T*. The *ximage* includes all elements and keys in the table *T*. The command

print *T["one"]*

prints whatever in the table *T* under the key "*one*". The command

print **T*

prints the size of table *T*. The command

print *!T*

prints all the elements in the table *T*; the *ximage* of *T*.

7.1.4. Strings

To print a string value or a substring, you just refer to that element in the source code. For example, the command

print *S*

prints the string contained in *S*. The command

print *S[5]*

prints the character in the position *5* of the string *S*. The command

print *S[2:8]*

prints all characters between the position *2* and *8* of the string *S*.

The command

print *S

prints the size of string *S*. The command

print !S

prints all the characters in the string *S*.

7.1.5. Records

To print any field of a record, you just refer to that field using the *dot* operator. For example, the command

print R

prints an *ximage* of the record *R*. The *ximage* includes all field names and their values (*image* or *ximage*) of the record *R*. The command

print *R.fname*

prints whatever in the field *fname* of the record *R*.

7.1.6. Global variables

The print command provides the ability to print a complete list of all global variable names and their current type. Either of

print –*global*
p –*g*

prints a sorted list of all global variable *names* and their *types* based on the current program state. The command "info global" is another way to produce this list.

7.1.7. Local variables

The print command provides the ability to print all local variable *names* and their *types* based on the currently selected stack frame. The user may use this command to check all local variable names that are in the currently selected stack frame. Either of

print –*local*
p –*l*

prints a sorted list of all local variable *names* and their *types* based on the currently selected stack frame. The command "info local" is another way to produce this list. This command excludes the *statics* and *parameters* from the local variable.

7.1.8. Static variables

The print command provides the ability to print all static variable *names* and their *types* based on the currently selected stack frame. The user may use this command to check all static variable names that are in the currently selected stack frame. Either of

print *–static*
p *–s*

prints a sorted list of all static variable *names* and their *types* based on the currently selected stack frame. The command "info static" is another way to produce this list.

7.1.9. Parameter variables

The print command provides the ability to print all parameter variable *names* and their *types* based on the currently selected stack frame. The user may use this command to check all static variable names that are in the currently selected stack frame. Either of

print *–parameter*
p *–param*

prints a sorted list of all parameter variable *names* and their *types* based on the currently selected stack frame. The command "info parameter" is another way to produce this list.

7.1.10. Keywords

Icon/Unicon keywords are varied in their types; some of which are simple integers such as &pos, or strings such as *&subject*. Others are generators such as *&features*, *&storage*, *&allocated*, and *&collections*. The print command is used to print a single value or generate a list of values out of a valid keyword. The command

print *keyword*

if *keyword* is a single value, it prints the value of *keyword*. Otherwise, if *keyword* is a generator, it prints the very first value generated by *keyword*. The command

print *&pos*

prints the value of keyword *&pos*. The command

print *&subject*

prints whatever string currently in the keyword *&subject*. The command

print **&subject*

prints the size of the sting currently in the keyword *&subject*. The command

print *&features*

prints the very first value generated by the keyword *&features*. However, the command

print *!&features*

prints all the values generated by the keyword *&features*.

7.2. Changing the value of an expression

The same print command that is used to get a peek at the value of an expression, it can be used also to change the value of a variable or keyword. In order to change a variable value in your program during the debugging session, the assignment operator ":=" or "=" must be used. When the print command is used to change the value of an expression, its syntax is as follows:

print *expr1* [:=|=] *expr2*
p *expr1* [:=|=] *expr2*

The *expr1* is either a *variable*, *keyword*, or an *element of a data structure*. Where as *expr2* is either a *literal* value, such as numeric or string, or an *expression*. The print command evaluates *expr2* first and assigns the result to *expr1*.

7.2.1. Simple variables

Simple global and local variables can be changed using the print command. Either of

print *variable := expr*
p *variable = expr*

evaluates *expr* based on the current execution state and the currently selected stack frame and assigns the result to *variable*. For example, the command

print *a = 10*

assigns variable *a* with the integer value *10*. The command

print *x = 10.5*

assigns variable x with the real value *10.5*. The command

print *name = "my test program"*

assigns variable *name* with the string value *"my test program"*. And the command

print *x = y*

assigns the value of *y* to the variable *x*.

7.2.2. Lists

UDB allows you to change list elements by assign to an element of a list.

For example, the command

print *L[2] = 3*

assigns the element number *2* of the list *L* with the literal integer *3*. The command

print *L[5] = b*

assigns the element number *5* of the list *L* with the value of the variable *b*. And the command

print *L[i][j] =10*

assigns the element *L[i][j]* with the literal integer *10*; both *i* and *j* must be valid variables in the current context. Note that in the current parsing mechanism; *L[i][j]* must has no spaces in it.

7.2.3. Tables

Assigning to an element of a table is similar to assigning to an element of a list, the only the difference is that a table key is not limited to the integer keys, and table elements can be created on the fly using the assignment operator. On the other hand, list elements are contiguous and must be created at the initialization time or with any of the built-in functions such as push and put. For example, the command

print *T[1] = 1000*

assigns the table element of *T[1]* with *1000*. And the command

print *T["one"] = 10*

assigns the table element of *T["one"]* with *10*.

7.2.4. Strings

A user can assign to a string or substring from within UDB. For example, the command

print *S = "abcdefg"*

assigns the literal string *"abcdefg"* to *S*. The command

print *S[1] = "A"*

assigns the string *"A"* to the first character in *S*. And the command

print *S[2:6] := "ab"*

replaces the characters between *2* and *6* of *S* with the substring *"ab"*.

7.2.5. Records

You can refer to any field of a record using the dot operator.

For example, the command

print *R.fname = 10*

assigns the field member *fname* of the record *R* with the literal integer *10*. The command

print *R.fname = person.id*

assigns the field *fname* of the record *R* with the field *id* of the record *person*. And the command

print *person.id = L[2]*

assigns the field *id* of the record person with the element number *2* of the list *L*.

7.2.6. Keywords

Some of the Icon/Unicon keywords are variables that can be assigned explicitly by the programmer during the course of execution. For those variable keywords, a UDB user is able to change their value too. For example, the command

print *&keyword = expr*

evaluates *expr* and assigns its value to the valid *&keyword*. The command

print *&pos = 10*

assigns the literal value *10* to the keyword *&pos*. And the command

print *&subject = "abcdefg"*

assigns the keyword *&subject* with the literal string value of *"abcdefg"*.

8. Object-Oriented Support

UDB's object oriented support includes both classes and packages. UDB provides the ability to place breakpoints on method names, watch and investigate class variables as well as method's local variables. If these properties are encapsulated within a package, UDB allow users to watch and investigate any of these properties based on their package.

8.1. Breakpoints on Methods

UDB uses the **::** scope character to separate between the class name and its method name. For example,

```
break Car::door
```

places a breakpoint on method door of the class Car.

8.2. Class Variables

When the currently selected a stack frame is for a method, the user can look up and modify the state of local variables as well as class variables. Class variables are accessed using the self class reference. For example, the command

```
print  self.x
```

displays the value of variable x in the current class. And the command

```
print slef.y := 10
```

assigns the current class variable y with the value 10.

9. Examining Source Code and Files

UDB knows about the source code and what files are involved in building the executable. At load time, UDB tries to open all of the related source files including library files; it builds a list of what source files are available and what are not. It builds such information by analyzing the icode of the executable itself. UDB allows you print and investigate parts of your program's source code. At any point during the debugging session UDB spontaneously points to the line where it stopped. Likewise, when you backtrace or select a stack frame, UDB prints information about the file and the line where the execution in that frame has stopped. In general, you can print any part of a source file or query successfully opened files and the executable static properties such as packages, procedures, classes and methods.

9.1. Check source lines

The command list allows you to print a window of your source code lines. The default number of lines is ten. The default source file name is the current source file that the program is stopped in. Otherwise, if the program is not running yet, the default source file name is the one that has the procedure main() in it and the default source line is pointing at the header of the procedure main().

For example, the commands

list
l

prints *10* lines of the source code surrounding the current execution point. Another list command will show the next *10* lines that follow based on the previous command. If The program is not running yet, the printed *10* lines will be centered around the procedure main().

After the first list command, the commands

list
l
list +
l +

shows the next *10* lines of the current source file based on a previous list command. The command

list –
l –

shows the previous *10* lines of the current source file based on a previous list command. The command

list *line*

shows *10* lines of the current source file surrounding *line*. The command

list *–line*

It shows *10* lines of the current source file surrounding the line number *line* from the end of the file.

list *procedure*

shows *10* lines of the source code surrounding the header of *procedure*. And the command

list *fname line*

shows *10* lines of the source file *fname* surrounding the line number *line*.

9.2. Check source info

UDB analyzes the loaded icode and builds information about its global names. When the program is loaded, UDB knows all global variables, records, classes, packages, procedures, and built functions in use. The user can query any of them using the info command.

For example, the commands

info package, info class, info record

print a sorted list of all package, class, and record names in use respectively. The commands

info procedure, info function

print a sorted list of all user defined procedure and built-in function names in use respectively. The command

info global

print a sorted list of all global variable names in use. And the commands

info local, info static, info parameter

print a sorted list of all local, static, and parameter variable names in the currently selected stack frame.

9.3. Check source files

UDB provides command to get information about the executable binary and what files, packages, classes, and procedures are used. When an executable is loaded, UDB analyses its icode, finds out what user and library files are in use, and what packages, classes, and procedures are used. For example, the command

info source

prints a detailed summary about the executable such as the number of files that are in use, and which is a user defined file and which is library file. The command

info files

prints a sorted list of all files used in the binary. The command

info found, info missing

prints a sorted list of all source files UDB was able to open and all file names UDB was not able to open respectively. The command

info user

prints a sorted list of all user-defined source file names in use.

The command

info lib

prints a sorted list of all library file names in use.

info icode

prints information about the current icode binary such as its version and its word size.

10. Examining Memory Usage

UDB provides the ability to investigate the memory usage of the buggy program at any point during the debugging session. A program running in the Unicon's virtual machine has two different memory regions that are allocated and managed separately: 1) string region is used by the string scanning facilities, and 2) block region is used by other programming activities. Each region is cleaned by the Garbage Collector separately. UDB's no intrusion on the buggy program space includes separate memory regions for each loaded program. If frequent contiguous thrashing is occurred, each region can grow automatically to prevent useless garbage collections.

10.1. Regions

The user can investigate the size of each region allocated for the loaded program. The command

print *&regions*

prints the total memory allocated for the loaded program and the size of the current maximum size of each region.

10.2. Allocated storage

The user can investigate the current memory in use. The command

print *&storage*

prints the total used memory by the buggy program, and how much memory is currently in each of the string and block regions.

10.3. Total allocations

The user can find how much is the total memory allocated by the buggy program during up until some point during or after the execution.

The command

print *&allocated*

prints the total amount of memory allocated by the buggy program; this shows the amount of memory allocated for each of the string region and the block region. Note that this command shows all allocated memory even those that may be collected during a garbage collection.

10.4. Total collections

The user can investigate the garbage collector activities in each region. The command

print *&collections*

prints the total number of garbage collections occurred during the execution of the buggy program up until this point. The command provides the total number of garbage collections, and the number of times it is been triggered by each of the string region and the block region.

11. Shell Commands

UDB console provides only two shell commands:

ls

This is equivalent to the UNIX ls command. And the command

pwd

is equivalent to the UNIX pwd command.

cd

is equivalent to the UNIX cd command.

12. Extension Agents

GDB style of debugging is not always good enough. Bugs vary in their root causes and their revealed behavior; some may cause a crash or a core dump, while others may cause an incorrect or missing output or an unexpected behavior. Breakpoint based debugging provides the ability to control the execution of the buggy program by stepping and continuing, and the ability to investigate the current execution state. Users have to keep their mentalities active and full with heuristic information such as when and where a variable was assigned with its current value.

UDB's debugging philosophy tries to overcome the typical debuggers' limitations. It provides the ability to facilitate additional external or internal debugging tools (agents). UDB is built on top of an open debugging architecture that allows the debugging core to cooperate with standalone event-driven Alamo-based debugging and monitoring agents. UDB's extensible architecture named IDEA (I*daho* D*ebugging* E*xtension* A*rchitecture*). IDEA is powerful enough to allow seamless incorporation of external and internal event-based monitoring agents to be loaded, enabled, or disabled on-the-fly during the debugging session. An agent may perform algorithmic and automatic debugging techniques through dynamic

analysis and visualization means. Different agents may vary in their goals; some may trace specific event while others may capture a specific execution behavior that is described through a sequence or a pattern of events. For example, users may utilize agents that would allow them to inquire information prior to the current execution state by tracing variable states over the execution; such an agent would allow the user to locate where a variable was assigned long before it caused the crash, which is hard or impossible to be captured using traditional debugging facilities of stepping and continuing.

12.1. Internal Agents

UDB has a library of different agents, which serve different behaviors such as memory allocations, garbage collections, failed loops, failed subscripts, variable profiles, dead variables, loops time, procedure times, etc. Internal agents are disabled by default, in order to be used; the user has to enable each one of them explicitly.

12.1.1. Enabling internal agents

An internal debugging agent can be enabled using the enable command. For example, the command

enable internal *agent*

enables the internal debugging agent named *agent*. For example, enable internal *memory*; enables the internal debugging agent named *memory*.

12.1.2. Disabling internal agents

An internal debugging agent can be disabled using the disable command. For example, the command

disable internal *agent*

disables the internal debugging agent named *agent*. For example, disable internal *memory*; enables the internal debugging agent named *memory*.

12.1.3. Analyzing information in the internal agent

The internal agent might collect information and not analyze it until the end of the program; however, a user can force the internal agents to analyze its information from any point during the debugging session. The command analyze is used to do so.

For example, the command

analyze internal *agent*

forces the internal debugging agent named *agent* to analyze its collected information. For example, analyze internal *memory*; it analyzes the information collected by the internal debugging agent named *memory*. This command is valid whenever the internal debugging agent is enabled. However, some internal debugging agents may not have this facility; for such agents, this command will *fail silently*.

12.1.4. Printing information from the internal agent

The internal agent collects information in the form of execution events. The user can force the internal agent to print its analyzed collected information using the command

print internal *agent*

This prints the analyzed information collected by the internal debugging agent named *agent*. For example,
print internal *memory*; enables the internal debugging agent named *memory*. This command is valid whenever the internal debugging agent is enabled. However, some internal debugging agents may not have this facility; for such agents, this command will *fail silently*.

12.1.5. Internal agents' info

The user can inquire about internal debugging agents using the info command.

info internal

This provides the current state of all internal debugging agents that are available in the system and whether each one is *enabled* or *disabled*.

The command

info internal *agent*

provides the current state of the internal debugging agent named *agent* such as whether it is *disabled* or *enabled*. For example, info internal *memory*; prints the status of the internal debugging agent named *memory*.

12.2. External Agents

The IDEA architecture opens the door for compatible standalone debugging programs to be loaded and used on the fly, as debugging agents, during a UDB debugging session. Any event-driven Alamo-based standalone agent can be loaded under UDB's control. Loaded external agents are enabled by default. External agents are loaded under the control of UDB's debugging core; it means that external agents are paused when the buggy program is paused and resumed when the buggy program is resumed after a breakpoint, watchpoint, tracepoint, or even in between stepping and continuing.

12.2.1. Loading external agents

Any compatible standalone program can be incorporated as a debugging agent within a UDB debugging session. The program can be loaded using the load command.

For example, the command

load external *agent*

loads the executable of the standalone program named *agent* as external debugging agent on the fly during the debugging session. The loaded agent is enabled by default. A standalone program's executable can be loaded from any point during the debugging session. However, the external agent will start collecting information from its loading point and as long as it is kept enabled. So, the agent information will be based on its watching time. This logic is good at some debugging situation where the user is only interested in a portion of the buggy program's execution time. Otherwise, the user has to load its debugging agent at the beginning of the debugging session.

12.2.2. Enabling external agents

An external debugging agent is enabled by default when it is loaded. If any of the external agents is disabled, it can be re-enabled using the enable command. For example, the command

enable external *agent*

enables the external debugging agent named *agent*. For example, enable external *memory*; enables the external debugging agent named *memory*. The command is valid from any point during the debugging session. The only restriction is that the enabled external agent must be previously loaded and disabled.

12.2.3. Disabling external agents

Any previously-loaded external agent can be disabled from any point during the debugging session. However, disabled agents receive no events and provide no information about the execution period when they were disabled.

The external agent can be disabled using the disable command.

disable external *agent*

This command disables the external debugging agent named *agent*. For example, disable external *memory*; disables the external debugging agent named *memory*. The command is valid from any point during the debugging session. The only condition is that the disabled agent must be loaded previously.

12.2.4. External agents' info

The user can inquire about loaded external debugging agents using the info command.

info external

This command provides the current state of all external debugging agents that are loaded into the debugging session and whether each one is *enabled* or *disabled*. The command

info external *agent*

provides the current state of the loaded external debugging agent named *agent* such as whether it is *disabled* or *enabled*.

12.3. Migration from externals to internals

External monitors impose an extra level of communication overhead in the form of *co-expression* context switch (Unicon lightweight threads are called co-expressions). For better performance external debugging agents can be migrated to internals. UDB's extensible architecture is open for standalone external tools that are: 1) event-driven Alamo-based monitors, and 2) written in either Icon or Unicon (Supporting C agents is on the TO DO list).

12.3.1. Simple agent

The code in Figure 1 is a very simple Alamo-based execution monitor (agent), where each event is mapped, in a one-to-one relation, into a single method. This conventional format allows UDB to provide automatic and easy registration for the used methods and the agent *event mask*. There are three types of methods inside the agent class that the automatic registration can recognize:

1) **Event handler methods**: that starts with the prefix "*handle_*" followed by the event name. Each method supposed to handle one event, (i.e. handle_E_Pcall()). The agent' *event mask* is constructed automatically based on those handler methods.
2) **Information analyzer method**: that starts with the prefix "*analyze_*" followed by any name. This method supposed to analyze the collected information, (i.e. analyze_info()).
3) **Information writer method**: this method starts with the prefix "*write_*" followed by any name. This method supposed to write information collected by the agent, (i.e. write_info()).

Agents that follow the naming convention used in Figure 1 are registered automatically through the two simple steps. First, derive the agent class from the Listener() class provided by the UDB's architecture. Second, place a call to the register() method provided by the Internal class:

```
register("name", object)
```

The first parameter provides a formal name to the agent, and the second parameter is the actual object of that agent. For example, in order to register the following simple Example() agent in Figure 1, you need to place a call to the register() method in the init() method of the Internal class as follows:

```
register("calls", Example())
```

This suggested formatting described in Figure 1 is intended to simplify the registration process. Users follow this format, all they have to do is to strip the monitor from its main procedure, use the automatic registration method, compile their agents and link them into the source code of UDB, see Figure 2. At that point users are able to use their own agents from within UDB and during the debugging session. Different

agents are distinguished by their names. The user is liable to enable or disable and use the agent facilities on the fly during the debugging session.

12.3.2. Complex agent

Complex agents are those ones that do not follow the conventional formatting discussed in the previous section. Complex agents have no restriction on how to handle events, how to analyze information, or how to display the analysis. Such agent methods are free from the naming convention. However, in order to register this type of agents, the user has to place a call the register() method with four extra parameters, which are used to register three types of method: 1) event handler, 2) information analyzer, 3) information or result writer, and 4) the agent event mask.

```
register( "name", object, [handlers], [analyzers], [writers],mask);
```

1) [handlers]: is a list of the string names of the methods used to handle events in the agent
2) [analyzers]: is a list of the string names of the methods used to analyze the collected information in the agent
3) [writers]: is a list of the string names of the methods used to write or print information from the agent
4) mask : is the set of events that the agent is monitoring in the buggy program.

For example, in order to explicitly register the Example() agent provided in Figure 1, the user can place a call to the register() method of the Internals class as follows:

```
register("calls", Example(),
                  ["handle_E_Pcall()",”handle_E_Fcall()”],
                  ["analyze_Info"],
                  ["write_Info"],
                  cset(E_Pcall || E_Fcall) )
```

This way of registration provides an extra layer over the simple automatic registration. It performs the simple automatic registration and adds to it those specified methods. This type of registration intended to provide users with enough freedom to write their own standalone agents in the way they want, and in the same time users will be able to integrate those as internals with the least possible modifications.

```
$include "evdefs.icn"
link evinit

 # A simple monitor that counts the number of method/procedure and built-in function calls
 class Example(eventMask,  pcalls, fcalls, pcalls_rate, fcalls_rate)

  # This method counts the number of method/procedure calls
  method handle_E_Pcall(code, value)
    pcalls +:= 1
  end

  # This method counts the number of built-in function calls
  method handle_E_Fcall(code, value)
    fcalls +:= 1
  end

  # This method do some calculations and analyze the collected information
  method analyze_info()
    local total

    total := pcalls + fcalls
    pcalls_rate := pcalls/total * 100
    fcalls_rate := fcalls/total * 100
  end

  # This method prints out some information
  method write_info()
    write(" # pcalls = ", pcalls, "  and its ratio is :", pcalls_rate)
    write(" # fcalls = ", fcalls, "  and its ratio is :", fcalls_rate)
  end

 # Constructor
 initially()
     eventMask := cset(E_Pcall||E_Fcall)
     pcalls := fcalls := 0
 end

 procedure main(args)
   local obj
   EvInit(args)
   obj := Example()
   while EvGet(obj.eventMask) do{
     case &eventcode of{
       E_Pcall:{
          obj.handle_E_Pcall(&eventcode, &eventvalue)
          }
       E_Fcall:{
          obj.handle_E_Fcall(&eventcode, &eventvalue)
          }
       }
     }
   obj.analyze_info()
   obj.write_info()
 end
```

Figure 1. Sample External Debugging Agent

```
$include "evdefs.icn"
link evinit

# A simple monitor that counts the number of method/procedure and built-in function calls
class Example : Listener (eventMask,  pcalls, fcalls, pcalls_rate, fcalls_rate)

  # This method counts the number of method/procedure calls
  method handle_E_Pcall(code, value)
    pcalls +:= 1
  end

  # This method counts the number of built-in function calls
  method handle_E_Fcall(code, value)
    fcalls +:= 1
  end

  # This method do some calculations and analyze the collected information
  method analyze_info()
    local total

    total := pcalls + fcalls
    pcalls_rate := pcalls/total * 100
    fcalls_rate := fcalls/total * 100
  end

  # This method prints out some information
  method write_info()
    write(" # pcalls = ", pcalls, "  and its ratio is :", pcalls_rate)
    write(" # fcalls = ", fcalls, "  and its ratio is :", fcalls_rate)
  end

 # Constructor
 initially()
     eventMask := cset(E_Pcall||E_Fcall)
     pcalls := fcalls := 0
 end

 procedure main(args)
    local obj

    EvInit(args)
    obj := MyMon()
    while EvGet(obj.eventMask) do{
        case &eventcode of{
            E_Pcall:{
                obj.handle_E_Pcall(&eventcode, &eventvalue)
                }
            E_Fcall:{
                obj.handle_E_Fcall(&eventcode, &eventvalue)
                }
            }
        }
    obj.analyze_info()
    obj.write_info()
 end
```

Figure 2. Sample Internal Debugging Agent

13. Future Work

UDB is under active development; which includes adding new advanced debugging features such as automatic debugging, dynamic analysis, and visualization features. Advances on UDB will be added into this TR incrementally as it is developed and stabilized. Eventually, a future version of UDB will be running with every Unicon program silently as an observer, if there is something interesting such as a runtime error, it will take the lead putting the user in no time in the debugging process. This will save the user the hassle of reproducing the bug once again inside the debugger. Hence, UDB as an observer in the back of every program does not cost the execution time of the program anything noticeable.

14. Appendix: UDB Command Summary

This appendix provides a summary of UDB commands with more examples. It can be used as a quick command reference.

A. Essential Commands

The most common commands that a user has to know in order to execute and control a program under UDB.

udb *program*	Starts UDB and loads the executable *program* into it.
run [*arglist*]	Starts the already loaded program [with *arglist*].
b *procedure*	Sets a breakpoint at the entry of *procedure*.
bt	backtrace: displays the current program stack ; where is an alias of this command.
p *expr*	print: displays the value of *expr*.
c	continue: resumes the running of the program.
n	next: executes the next line and steps over any procedure call in it.
s	step: executes the next line and steps into any procedure call in it.

B. What to Do After a Crash

The following commands are good enough to start an investigation after a crash on the buggy program.

where	displays the current execution stack
frame	provides information about the currently selected stack frame
up	moves the currently selected stack frame one frame up on the execution stack (current frame + 1).
down	moves the currently selected stack frame one frame down on the execution stack (current frame - 1).
print	allows you to print variable values.

C. Starting UDB

Different ways to start UDB and to load a program into it.

udb	Starts UDB with no executable.
udb *program*	Starts UDB and loads the executable *program* into it.

D. Stopping UDB

One command that is needed in order to exit UDB from any point.

quit	Exits UDB. q and Ctrl-C are aliases.

E. Getting Help

An important command to inquire and get help about other UDB commands.

help	Lists all classes of commands. h and ? are aliases.
help *class*	Provides a specific description for a *class* of commands.
help *command*	Provides a detailed description about a specific *command*.

F. Executing a Program

How to start the execution of a loaded program.

run *arglist*	Starts the currently loaded program with *arglist*. r *arglist* is an alias.
run	Starts the currently loaded program without arguments. r is an alias.
load *program*	Loads the *program* executable into UDB; if a program is already loaded, this command replaces it with a new *program*.

G. Breakpoints

Important commands on how to make the program stop at certain points; source code locations such as a line number or an entry to a procedure or method.

break *line*	If execution is stopped, it assumes *line* within the *current file*, otherwise, *line* is assumed to be within the *file* that contains procedure main(). b *line* is an alias.
break [*file*] *line*	Sets a breakpoint at *line* number [in *file*]. b [*file*] *line* is an alias, i.e. b *test.icn 15*.
b *procedure*	Sets a breakpoint at the entry of *procedure*.
info break [*id*]	Shows a complete list of all breakpoints and their status. If *id* is provided it shows only the breakpoint with the number *id*. info breakpoints [*id*] is an alias.
info break [*file*]	Shows a complete list of all breakpoints and their status; if *file* is provided, it shows only breakpoints from that [*file*]. info breakpoints [*file*] is an alias.
clear	Removes all breakpoints.
clear break	Removes all breakpoints.
clear break [*file*] *line*	Removes the breakpoint at *line* [in *file*].
clear break *proc*	Removes the breakpoint at the entry to procedure *proc*.
???????????	
delete break [*n*]	Deletes all breakpoints, if [*n*] is provided, it only deletes the breakpoint with the id number [*n*]; deleted breakpoints are still seen by the command info break, but marked as *deleted*.
enable break [*n*]	Enables all disabled breakpoints, if [*n*] is provided, it only enables the breakpoint with the id number [*n*].
disable break [*n*]	Disables all breakpoints, if [*n*] is provided, it only disables the breakpoint with the id number [*n*].

H. Watchpoints

Techniques to observe certain variable activities such as a variable being assigned, read, changed value, or changed type. Watchpoints may cause the program to stop at specific action(s), or they may work

silently collecting information about specific action(s). Most watchpoints supports relational operations such as =, ~=, <, >, <=, >=, which they can be applied on the *value* or *type* of the observed variable or keyword.

awatch [-*silent*] [*count*] variable [[=|>|<|<=|>=|~=] *value*]
: Sets an assignment watchpoint on *variable* whenever assigned, with an optional condition on the assigned value. watch is an alias.
If *–silent* is provided, the watchpoint does not notify the user at every incident.
If *count* is provided and *count > 0*, it observes the first *count* number of incidents.
If *count* is provided and *count < 0*, the user is able to trace back the last *count* number of incident's locations and values.

watch *–silent variable*
: Sets a *silent* watchpoint on *variable* whenever assigned.

watch *count variable*
: Sets a *normal* watchpoint on *variable* on the first *count* number of assignments.

watch *-count variable*
: Sets a *normal* watchpoint on *variable* and keeps track of the last *count* number of assignments.

watch *variable = value*
: Sets a *normal* watchpoint on *variable* whenever assigned with *value*.

watch *variable > value*
: Sets a *silent* watchpoint on *variable* whenever assigned and the assigned *value > value*.

watch *–s n variable*
: Sets a *silent* watchpoint on *variable* on the first *n* number of assignments.

rwatch [*–silent*] [*count*] *variable* [[=|>|<|<=|>=|~=] *value*]
: Sets a watchpoint on *variable* whenever read. Other arguments are similar to the watch command.

vwatch [*–silent*] [*count*] *variable* [[=|>|<|<=|>=|~=] *value*]
: Sets a watchpoint on *variable* whenever assigned and the new value is different from the old one (changed value). Other arguments are similar to the watch command.

twatch [*–silent*] [*count*] *variable* [[=|~=] *type*]
: Sets a watchpoint on *variable* whenever assigned and the type of new value is different from the type of the old one (changed type). Other arguments are similar to the watch command.

swatch [*–silent*][*count*]
: Sets a watchpoint on string scanning environment; in particular the explicit and implicit change of *&pos* and *&subject* keywords.

info watchpoints
: Shows a complete list of all watchpoints; info watch and watch are aliases.

info awatch
: Shows a list of all *assignment* watchpoints.

info rwatch
: Shows a list of all *read* watchpoints.

info vwatch
: Shows a list of all *value change* watchpoints.

info twatch
: Shows a list of all *type change* watchpoints.

clear watch — Clears all watchpoints; watchpoints with different types are cleared. If watch is replaced with any of awatch, rwatch, twatch, vwatch, or swatch, it clears only the specified type of watchpoints.

delete watch [*n*] — Deletes all watchpoints, if [*n*] is provided, it only deletes the watchpoint with the id number [*n*]. If watch is replaced with any of awatch, rwatch, twatch, vwatch, or swatch, it deletes only the specified type of watchpoints.

enable watch [*n*] — Enables all disabled watchpoints; if [*n*] is provided, it only enables the watchpoint with the id number [*n*]. If watch is replaced with any of awatch, rwatch, twatch, *vwatch*, or *swatch*, it enables only the specified type of watchpoints.

disable watch [*n*] — Disables all enabled watchpoints; if [*n*] is provided, it only disables the watchpoint with the id number [*n*]. If watch is replaced with any of awatch, rwatch, twatch, vwatch, or swatch, it disables only the specified type of watchpoints.

I. Tracepoints

Techniques to observe execution behavior of *potential suspicions activities* such as the type of the returned value from a user-defined *procedure*, built-in *function*, and language *operator*. It is intended to provide more lightweight flexibility to simplify and speed up the process of discovering bug locations. Behaviors can be general such as *start* or *end*, or detailed such as *call* and *resume* as specific details for the *start* behavior, and *return*, *suspend*, *fail*, and *remove* as specific details for *end* behavior. In particular, the *return* behavior is applicable for extra condition on the returned value. If the flag –silent is provided, the tracepoint will not stop the execution but the user will be able to check the traced info from any point during or after the execution.

trace [–*silent*] [*count*] *procedure* [*behavior* [*op value*]]

Sets a tracepoint on *procedure* whenever the provided *behavior* is satisfied.
If *behavior* is not provided, all behaviors are traced.
If *–silent* is provided, the tracepoint does not notify the user at every incident.
If *count* is provided and *count > 0*, it traces the first *count* number of incidents.
If *count* is provided and *count < 0*, the user is able to trace back the last *count* number of incidents.

trace *bar* — Sets a tracepoint on all valid behaviors of the *procedure bar*.

trace *bar call* — Sets a tracepoint on *procedure bar* whenever it is *called*. Action is very similar to the break bar command.

trace *bar return* — Sets a tracepoint on *procedure bar* whenever it is *returned*.

trace *bar return <= 1* — Sets a tracepoint on *procedure bar* whenever it *returns* a *value <= 1*.

trace *10 bar resume* — Sets a tracepoint on *procedure bar* for the first *10* times it *resumes*.

trace *bar fail* — Sets a tracepoint of *procedure bar* whenever it is failed.

trace *–silent bar* — Sets a *silent* tracepoint on all valid behaviors of the *procedure bar*; this tracepoint will not stop the execution at every traced behavior incident.

trace [–*silent*] [*count*] *function* [*behavior* [*op value*]]

Sets a tracepoint on built-in *function* whenever the provide *behavior* is satisfied. If *behavior* is not provided, all behaviors are traced. Other arguments are similar to the procedure trace command.

trace *abs call* — Sets a tracepoint on the *function abs()* whenever it is *called*.

trace *write fail* — Sets a tracepoint on the *function write()* whenever it is *failed*.

trace *cos return < 0* — Sets a tracepoint on the *function cos()* whenever it is *returns* a *value < 0*.

trace [–*silent*] [*count*] *operator* [*behavior* [*op value*]]

Sets a tracepoint on a built-in *operator* whenever the provided *behavior* is satisfied. If behavior is not provided, all behaviors are traced. *operator* is one of the following: (+, -, *, /, \, =, ~=, ==, ~==, ===, ~===, <, <=, <<=, >, >=, >>=, ++, --, **, !, ?, []).

trace *[] fail* — Sets a tracepoint on *[]* (*subscript operation*) whenever it is *failed*.

trace *! suspend* — Sets a tracepoint on *!* (*Bang operator*) whenever it is *suspended*.

trace = *fail* — Sets a trace point on = whenever it is *failed*.

trace ~== — Sets a trace point on ~== whenever any of its behaviors is satisfied (occurred).

trace ~== *return* — Sets a tracepoint on ~== whenever it *returns* (the operation succeeded because both sides are lexically not equal).

trace ~== *return = "ab"* — Sets a tracepoint on ~== whenever it *returns* (the operation succeeded because both sides are lexically equal to "*ab*").

info tracepoints — Prints a complete list of all tracepoints; info trace and trace are aliases.

info trace [*n*] — Prints detailed information about the tracepoint with id number [*n*].

info trace [*name*] — Prints detailed information about the tracepoint set on [*name*].

info trace *enabled* — Prints a complete list of all *enabled* tracepoints.

info trace *disabled* — Prints a complete list of all *disabled* tracepoints.

info trace *deleted* — Prints a complete list of all *deleted* tracepoints.

clear trace [*n*] — Clears all tracepoints, if [*n*] is provided, it only clears the tracepoint with id number [*n*].

clear trace [*name*] — Clears all tracepoints, if [*name*] is provided, it only clears the tracepoint set on [*name*].

delete trace [*n*] — Deletes all tracepoints, if [*n*] is provided, it only deletes the tracepoint with id number [*n*].

delete trace [*name*] — Deletes all tracepoints, if [*name*] is provided, it only deletes the tracepoint set on [*name*].

enable trace [*n*] — Enables all tracepoints; if [*n*] is provided, it only enables the tracepoint that has the id [*n*].

enable trace [*name*] — Enables all tracepoints; if [*name*] is provided, it only enables the tracepoint set on [*name*].

disable trace [*n*] — Disables all tracepoints; if [*n*] is provided, it only disables the tracepoint that has the id number [*n*].

disable trace [*name*] — Disables all tracepoints; if [*name*] is provided, it only disables the tracepoint set on [*name*].

J. Program Stack

Techniques to investigate the interpreter stack (execution stack). When the execution stops at any point, the currently selected frame points at the frame of the currently executing procedure, a user may change the currently selected frame or traceback all stack frames.

backtrace [n]	Prints a trace of all frames in the current stack. If [*n*] is provided, it prints the n^{th} innermost frames when *n>0*, and it prints the n^{th} outermost frames when *n < 0*. where [*n*] and bt [*n*] are aliases; i.e. where, where *10*, where -*10*, bt , bt *10*.
frame [n]	Selects and displays information of frame number [*n*]; if [*n*] is not provided, it displays information about the currently selected frame. f [*n*] is an alias.
up [n]	Moves the selected frame [*n*] frames up; if [*n*] is not provided, it moves the currently selected frame one frame up.
down [n]	Moves the selected frame [*n*] frames down; if [*n*] is not provided, it moves the currently selected frame one frame down.

K. Execution Control

Includes commands to step and resume the execution of the program.

continue	Resumes program's execution. cont and c are aliases.
step [*count*]	Executes the program until a new line is reached; if [*count*] is specified, it repeats the command *count* more times. s and s [*count*] are aliases.
next [*count*]	Executes the next line and steps over any procedure call; if [*count*] is specified, it repeats the command *count* more times. n and n [*count*] are aliases.
return	Completes the execution of the current procedure and returns back to the place of calling to step on the next statement after the call. ret and finish are aliases.

L. Display and Change Data

Ways to examine and change data in the current execution state; change can be done by assigning to variables or keywords.

print *variable*	Prints the value of *variable*; if *variable* is a reference to a structure, then it displays its ximage, otherwise it displays its simple value. p is an alias.
print *&keyword*	Prints the value of *&keyword*; For example: print *&pos*
print *expr*	Prints the evaluation of the *expr*. For example: p *L[5]* : prints the contents of *L[5]*. p *S[i : 10]* : prints the characters between *i* and *10* of string *S*. print *r.a* : prints the contents of failed *a* of record *r*.

print *variable = expr* Evaluates *expr* and assigns its value to *variable*. For example:
print *x = 10*
print *L[1] = 1000*
print *T["one"] = "First"*
print *S[4] = "K"*
print *S[5:10] = "insert a string"*
print *r.a = 4.5*
print *x = y*; where *y* is another variable.

print *&keyword = value* Assigns a value to a *&keyword*; For example:
print *&pos = 1*
print *&subject = "ABCcba"*

print **variable* Prints the *size* of *variable* whenever it is applicable; i.e. print **L*, or print **S*.

print *!variable* Generates and prints the values of *variable*; i.e. print *!L*, or print *!S*.

print *&features* Prints the *first generated* value out of the keyword *&features*.

print *! &features* Prints *all generated* values out of the keyword *&features*.

info local Shows all local variable names in the currently selected frame. print –local is an alias.

info static Shows all static variable names in the currently selected frame. print –static is an alias.

info parameter Shows all parameter variable names in the currently selected frame. print –param is an alias.

M. Source Files and Code Info

Commands to look up source files and code. UDB tries to open *user* and *library* files, which are used to build the executable. A user can navigate source files and source code based on the executable.

list Displays *ten lines* of source code; if execution is paused, the printed lines are from the current line and file, otherwise, the printed lines are from the file that has the procedure main(). l is an alias.

list + Displays the *next ten lines* of source code. l + is an alias.

list - Displays the *previous ten lines* of source code. l - is an alias.

list *procedure* Displays ten source lines surrounding *procedure*.

list [*file*] *line* Displays ten source lines surrounding *line* [in *file*]; if line is positive, counts will starts from the top of the file, otherwise, count starts from the bottom of the file. i.e. l -25: shows ten lines surrounding the line number *25* counting backward from the end of *file*.

info source Prints a detailed summary about the loaded executable. source is an alias.

info file Prints a list of all source files in use including library files. source file is an alias.

info found	Prints a list of all loaded source files including library files. source found is an alias.
info missing	Prints a list of all not loaded used source files. source missing is an alias.
info user	Prints a list of all user-defined source file names in use. source user is an alias.
info lib	Prints a list of all library file names in use. source lib is an alias.
info package	Prints a list of all package names in use. source package is an alias.
info class	Prints a list of all class names in use. source class is an alias.
info record	Prints a list of all record names in use. source record is an alias.
info procedure	Prints a list of all procedure names in use. source procedure is an alias.
info function	Prints a list of all built-in function names in use. source function is an alias
info global	Prints a list of all global variable names in use. source global is an alias.
info icode	Prints information about the current icode binary such as its version. source icode is an alias.

N. Memory Usage

Important commands to look up the memory usage

print *&regions*	Prints a summary of the total available memory an how mach in each region.
print *&storage*	Prints a summary of the total currently used memory and how much is currently allocated in each region.
print *&allocations*	Prints a summary of the total allocations up to that point of execution. Memory that cleaned up by the GC is still count.
print *&collections*	Prints a summary of the total number of Garbage Collections occurred up to that point of execution.

O. Shell Commands

Some of the most needed shell commands during a UDB session.

ls	Equivalent to the Unix ls shell command.
pwd	Equivalent to the Unix pwd shell command.
cd	Equivalent to the Unix cd shell command.

P. Extension Agents

How to load and manage external standalone debugging agents on the fly during the debugging session.

enable internal *agent*	Enables the internal agent named *agent* on the fly during the debugging session
disable internal *agent*	Disables the internal agent named *agent* on the fly during the debugging session

info internal	Prints information about all internal agents available in the session and the system
info internal *agent*	Prints information about the internal agents named *agent*
load –agent *agent*	Loads the standalone external agent named *agent* on the fly during the debugging session
enable external	Enables all external agent that are loaded and disabled in the current session
enable external *agent*	Enables the external agent named agent that is loaded and disabled in the current session
disable external	Disables all external agent that are loaded in the current session
disable external *agent*	Disables the external agent named agent that is loaded and enabled in the current session
info external	Prints information about all external debugging agents available in the session
info external *agent*	Prints information about the external agent named *agent*

15. References

1. Jeffery, C. L., 1999. *Program Monitoring and Visualization: an Exploratory Approach*, Springer New York.
2. Jeffery, C. L., Mohamed, S., Pereda, R., and Parlett, R. 2004. *Programming with Unicon*. http://unicon.org/book/ub.pdf.
3. Griswold, R. E., and Griswold, M. T.1997. *The Icon Programming Language*. Peer-to-Peer Communications, Inc., San Jose, California.
4. Stallman, R. M., Pesch, R., Shebs, S., et al. 2002. *Debugging with GDB: the GNU Source Level Debugger*. http://sourceware.org/gdb/documentation.
5. Ziad Al-Sharif and Clinton Jeffery, *An Extensible Source –Level Debugger*, To be published by the ACM SAC, Honolulu, Hawaii, march 8-12, 2009.
6. Ziad Al-Sharif and Clinton Jeffery, *An Agent Oriented Source-Level Debugger on Top of a Monitoring Framework*, To be published by the IEEE ITNG, Las Vegas, Nevada, April 27-29, 2009.