ABSTRACT

Standard debuggers are usually limited in the amount of analysis that they perform in order to assist with debugging. This paper presents UDB, an agent-oriented source-level debugger for the Unicon programming language with a novel architecture and capabilities. UDB combines classical debugging techniques such as those found in GDB with a growing set of extension agents. UDB demonstrates the feasibility of a source-level debugger built on top of a very high level event-based monitoring framework. The debugger is easily extended with new debugging agents that can employ a wide range of automatic debugging and dynamic analysis techniques.

Keywords: Debugging, Source-Level Debugger, Extension Agents, Monitoring Agents.

1- INTRODUCTION

Classical debuggers allow developers to observe the state of the program using techniques such as breakpoints and watchpoints, or to step through the source code and examine the execution stack. Such techniques are not always successful or efficient in enabling the programmer to locate or to understand the cause of a bug. Different bugs call for different debugging techniques, so experimentation is needed in order to develop the features that will be widely adopted in future debuggers.

This paper describes an agent-oriented source-level debugger that divides the debugging process over multiple simultaneous agents. The debugger is named UDB, an acronym for Unicon DeBugger [1]. An agent is an event-driven task-oriented program execution monitor; it can be loaded on the fly into the debugging session or migrated into the debugger source code as a permanent feature. Different agents perform different debugging missions such as detecting a suspicious execution behavior, performing an automatic debugging procedure, or executing a dynamic analysis technique.

UDB’s agents can be written and tested as standalone programs and then loaded into the debugger to work in concert with each other. UDB provides a simple interface to load, unload, enable, or disable debugging agents from any point
during conventional source-level debugging sessions, and the user can be selective about which agent(s) to use.

Agents are coordinated by a central debugging core. UDB’s core agent monitors the execution of a program for specific run time events; an event is an action during the execution of the program such as a method being called or a major syntax construct being entered. To extend the debugging core, additional agents can be loaded and active, and each agent receives different runtime events. An agent requests events via an event mask, which is a specification of a set of desired events. Each agent 1) provides the debugging core with its event mask, 2) receives relevant events from the debugging core, 3) performs its debugging mission, which may utilize execution history prior to the current execution state, and 4) sends its results and follow-on requests back to the debugging core.

The rest of this paper presents the design and implementation of UDB. Section two provides background information. Section three gives a sample debugging session. Section four discusses the design and the debugger’s monitoring architecture. Section five provides the implementation of both UDB’s classical and advanced debugging features. Section six describes UDB’s extension architecture. Section seven presents an evaluation of UDB’s performance. Section eight compares UDB’s features with related work. Section nine provides the conclusions and discusses planned future work.

2- BACKGROUND

The implementation context of this research is the Unicon programming language [2]. Unicon is an object-oriented dialect of Icon [3], a very high level imperative programming language with dynamic and polymorphic structure types, along with generators and goal-directed evaluation. Very high level languages’ advanced features may introduce special kinds of bugs and create special needs for debugging tools.

UDB employs the runtime execution events provided by AlamoDE [4], which is an extension of the Alamo monitoring framework [5,6,7]. AlamoDE provides control over the execution of the buggy program with efficient VM runtime system instrumentation and no intrusion on the buggy program space. AlamoDE is integrated in the Unicon language with insignificant cost (other than a modest increase in VM code size) in the production virtual machine. This integration allows the debugger to run on the virtual machine synchronously along with the buggy program. The debugger agents and the buggy program run in separate threads and the buggy program is the only one affected by the instrumentation. Alamo’s load() primitive reads the buggy program, sets it up with its own code, static data, stack and heap, without linking symbols into the current program.

AlamoDE provides a comprehensive set of execution events. Events, event patterns, and event sequences are used by the debugger and its task-oriented agents to investigate the state of the buggy program, and to detect suspicious
behaviors and potential bugs. Alamo’s events are lightweight, in the sense that each event is a (code, value) pair and they are transmitted without involving the operating system, minimizing the processing overhead. Alamo’s support for dynamic event filtering provides the ability to change the set of requested events on the fly by modifying the event mask. Further event filtering based on event values substantially reduces the number of reported events and the resulting number of context switches.

AlamoDE offers the ability to investigate the state of the buggy program. It provides both access functions and events that enable monitors to acquire information about the current: 1) source code location of execution such as file name, line number, column number, syntax type, procedure/method name, class name, and package name, 2) stack state and its active and suspended activation records, and 3) global and local variables’ names, values, and types. AlamoDE also provides the ability to change the execution state by assigning to variables, which is important for interactive debugging.

3- SAMPLE SESSION

UDB’s user interface resembles GDB’s console interface. UDB includes most of the commands supported by GDB; it adds a handful of commands to manage extension agents during the debugging session. The user can load, unload, enable, or disable agents from the console based interface. UDB’s external agents are enabled by default at load time, whereas its internal agents need to be explicitly enabled. Fig. 1 shows a sample UDB debugging session in which the loop agent is used to watch for aberrant while loops in the sort program; see Section 5-2-4.

4- DESIGN

UDB uses a hybrid model of execution monitoring. The debugger and buggy program execute as separate threads in a shared address space. Unicon’s threads are called co-expressions. A co-expression is a synchronous thread inside the virtual machine. In the hybrid execution model, agents may either execute within their own co-expression or they may be placed directly within the debugging core.

The debugging core coordinates the operations between multiple simultaneous agents, the buggy program, and the user interaction. The event masks of extension agents are added to the set of events that are requested by default by the debugging core. On the fly, UDB’s debugging core starts asking the buggy program about those extra events. When the debugging core receives an event from the buggy program, it forwards the received event to those extension agents that requested this kind of event in their event mask. For internal agents, this takes the form of a call to a listener method, while for external agents it takes the form of a co-expression switch. Furthermore, an agent may change its event mask during the course of execution. A change on any agent’s event mask triggers a
recalculation of the event mask of the debugging core and alters the set of events forwarded by the debugging core to that agent.

$ udb sort

UDB Version 1.5, June 2009.
sort:loaded 2.5K bytes, 32-bit uncompressed icode
1 Source file is found
Type “help” for assistance

(udb) break BubbleSort
   Breakpoint set successfully in:
   1#  sort.icn(5): BubbleSort( A )

(udb) run
   Starting program sort ...
A = [4,1,8,9,0,6,5,7,2,3]
   Breakpoint #1:
   sort.icn(5): BubbleSort( A )

(udb) load -agent=loop -name=while -iteration=0
   1# Agent loop is activated to watch
      0 iterated while loops

(udb) continue
   Continuing ...
   1# Agent loop: failed while
   test.icn(10): while swapped ~== “true” do{

(udb) print swapped
   swapped = “false”

(udb) disable -agent=loop
   1# Agent loop is disabled

(udb) quit
   sort is running, do you want to quit(Y/n)?:y
   Thank you for using UDB, Goodbye!

$
4-1 ARCHITECTURE

UDB is comprised of five major components: 1) a console that provides a user interface supported by a command interpreter for user control, 2) a session that initializes and manages the state of the debugger and controls the debugging evaluator, 3) an evaluator that provides the main event-driven debugging analysis and monitoring control, 4) an agents interface that facilitates and provides the programming interface for external and internal extensions, and 5) a state that maintains the debugging information between the components. See Fig. 2.

The session and the evaluator are generators; expressions that suspend values to the caller and can be resumed to produce additional values [1, 3]. The evaluator generator provides the ability to suspend the main monitoring loop without losing its state. The session generator provides the ability to maintain
the debugging session and the state of the evaluator generator before handing
the control to the console. This mechanism provides the user with the ability to
continue debugging by resuming the generator of the debugging session at its
previous state and resuming the evaluator at the point that was suspended.

Figure 3 The UDB UML diagram.

4-2 SOURCE CODE

UDB’s source code is organized around its five major components. The user in-
terface component is modeled by the Console class. The session component is
modeled by the Session class. The state component is modeled by the State,
Icode, and srcFile classes. The evaluator component is divided into two
subcomponents: 1) classical debugging that is modeled by the Breakpoints,
Watchpoints, Stepping, Stack, Data, and Tracepoints classes, and 2)
the extension agents interface is modeled by the Extensions, Externals, Internals and Listener classes. See Fig. 3.

5- IMPLEMENTATION

UDB implements classical debugging techniques as well as advanced agents by monitoring the buggy program for execution events at run time. This implementation faces potential performance challenges. In compensation, this type of implementation greatly simplifies the process of experimenting with new agents that utilize advanced debugging techniques. This experimentation probably would not be undertaken if the implementation was limited to one of the low-level approaches found in mainstream debuggers. This section discusses UDB’s implementation of several classical debugging techniques, and introduces some of its agents.

5-1 CLASSICAL DEBUGGING TECHNIQUES

This section presents the implementation of UDB’s classical debugging techniques by means of monitored execution events.

5-1-1 BREAKPOINTS AND WATCHPOINTS

UDB’s breakpoints are implemented by monitoring the line number event \( E_{\text{Line}} \) only when there is at least one breakpoint in the debugging session. Furthermore, UDB processes a line number event only when that line number has a predefined breakpoint on it. This implementation is approximated by utilizing the value mask of the Alamo framework. A value mask is a set of values of interest for a given kind of event; each event in the event mask can have its own value mask. Using the value mask, the line numbers of break points are associated with the line number event code \( E_{\text{Line}} \). In this way, there is only a context switch for those line number events whose source code lines are in the value mask. For the sake of better monitoring consistency, a breakpoint on a procedure/method is converted internally into a breakpoint on the line number of the procedure’s header.

UDB’s watchpoints are implemented by monitoring the assignment event \( E_{\text{Assign}} \). Assignment events are very frequent and therefore expensive to monitor. The watchpoint implementation needs to be selective, and monitor assignments only for variables being watched. This is approximated by utilizing the value mask for \( E_{\text{Assign}} \) events to specify variables of interest.

5-1-2 STEPPING AND CONTINUING

UDB implements the basic step and next commands using the line number event \( E_{\text{Line}} \). The implementation of the next command must ensure that if the line number change is preceded by a procedure call event \( E_{\text{Pcall}} \), UDB ignores line number changes until the program returns from all of the procedures that were called on that line where the next command was applied.
On the other hand, the `continue` command resumes the buggy program at its full speed. Its implementation is accomplished by removing the line number event `E_Line` and the procedure call event `E_Pcall` from the monitoring event mask unless they are needed by another currently enabled debugging feature.

5-2 ADVANCED DEBUGGING AGENTS

UDB employs agents with automated techniques to locate a growing set of suspicious behaviors. In some cases the agent can be confident from its analysis that it has found a bug and in others it issues an appropriate warning. Either way, the combination of valgrind-style [8] dynamic analysis within an interactive debugger makes both methods more effective. This section describes a representative number of simple example agents.

5-2-1 VARIABLE CHANGING TYPE (OR DOMAIN)

Unicon is a dynamically typed language; variable declarations are not required and a variable can be assigned values of different types. Such type changes are not a good programming practice; a type change can indicate a logical error or complicate a reading of the source code. UDB offers an agent that catches such dynamically typed variables by monitoring every assignment and checking whether it produces any type change on the assigned variable. This detection is based on two consecutive events: `E_Assign` and `E_Value` which are the event code of the assignment, and assigned value respectively. This implementation is costly, but it is worth paying for when it is needed, see Section 7. Appropriate static analysis (such as type inferencing) is outside the scope of this paper but can reduce the cost of this dynamic analysis, for example by reducing the runtime monitoring to that set of variables for which type inferencing determines that they might change their types.

5-2-2 UNINITIALIZED AND DEAD VARIABLES

Uninitialized and dead variables are variables that are read before they are assigned or assigned and not subsequently read during a particular execution. In mainstream languages, detecting uninitialized and dead variables can be achieved using static analysis techniques. However, this agent detects variables that are theoretically live according to static analysis, but observed to be dead in a particular program run. Even if such variables do not introduce a bug, they are still a bad programming practice; it is good to warn the user about them.

This agent tracks referenced variables based on their scope. For example, local variables are monitored over several calls before they are considered frequently uninitialized or dead. The primary monitored event code is `E_Deref`, which is reported when a variable is read.

5-2-3 OUT OF BOUNDS SUBSCRIPTS

Unicon's lists are dynamic in size. If the program tries to access an element beyond the list's actual number of elements, the operation fails silently. This
semantic is useful in conditional expressions, but in ordinary code it usually indicates a bug. In UDB, users can request notification about unchecked failed subscripts, and they can decide for themselves whether it is a bug or not. A similar check might benefit several other expressions that can fail.

This agent performs the out of bounds subscript check by monitoring every failed subscript and reports where and when that failure happened, and how far is the index beyond the actual size. This detection is based on three different events: E_Ref, E_Sub and E_Ofail which are the event code for the referenced list, the index, and the failed operation respectively.

5-2-4 ZERO ITERATED LOOPS

Code in the program must be executed under some circumstances; otherwise it is dead code. Dead code in routines that are not called is common, especially in programs under construction or with many optional modules. However, sometimes a loop in live code may execute zero times because the loop condition is not valid. If such a loop fails constantly, it may indicate a bug. Unfortunately, using classical debugging techniques, it is difficult to observe loops that exhibit this suspicious behavior.

This agent checks zero iterated loops by monitoring both the E_Syntax and E_Efail event codes, which are the current syntax and the current failed expression events respectively.

5-2-5 REDUNDANT CONVERSION

A program’s poor performance might go unexplained, especially if the complexities of the algorithms do not indicate that performance should be slow. This slowdown might be caused by any number of reasons related to bad programming practice. In Unicon, one of the common performance bugs results from frequent redundant type conversions. This agent automatically detects such potential performance bugs. It starts by tracking implicit type conversions at every location and analyzes the frequent conversions and their locations. This detection is based on three events: E_Loc, E_Sconv, and E_Tconv, which report the location, the successful conversion, and the result type of the conversion respectively.

5-2-6 TRACE VARIABLES’ STATE: ACTUAL CAUSE OF CRASH

Most bugs are revealed long after their actual cause. A variable might be assigned early in the execution, and that value may cause a bug far from that last assigned place. This agent provides automated techniques to detect and trace back variable information such as when and where in the program that variable’s last assignment occurred. The main monitored event is E_Assign, which is reported at every assignment. If a variable’s value causes a later crash or an incorrect result, then UDB provides the user with the last line number and file name where that variable was assigned.
6- EXTENSIONS

UDB supports two types of extension agents: 1) standalone agents that can be loaded on the fly, from any point during the debugging session, and 2) debugging agents that are incorporated into the debugging core as permanent features.

6-1 SAMPLE AGENT

Fig. 4 shows an example agent that captures the number of calls of user-defined procedures, methods, and native built-in functions, and finds the ratio for each call type. This provides a rough measure of the degree of VM overhead for a particular application. The class `Example()` contains three kinds of methods that summarize the potential functionalities provided by a debugging agent. Agents that follow this method naming convention can be registered automatically with the library of internal agents. Otherwise, agents can be registered manually. For more information about the migration process see Section 6-4. In contrast, external agents require no special formatting and no pre-registration.

1. Event handler methods whose names start with the prefix `handle_` followed by the handled event name. Each method processes one event, (i.e. `handle_E_Pcall()`). The agent's event mask is constructed automatically based on those handler methods. They may collect or analyze information based on the received events.

2. Information analyzer methods whose names start with the prefix `analyze_` followed by any name (i.e. `analyze_info()`). This method analyzes the collected information.

3. Information or result writer methods whose names start with the prefix `write_` followed by any name. This method should write information based on the agent analyses (i.e. `write_info()`).

6-2 EXTERNAL DEBUGGING AGENTS

External agents can be written and tested as standalone tools and then loaded and managed on the fly during the debugging session. UDB’s external agents are loaded and controlled by its debugging core. Active agents are paused whenever the buggy program is paused and they resume when it resumes.
The debugging core receives runtime events from the buggy program based on the current debugging context, and based on the event mask of the external agents. The Externals component filters the received events between different external agents. Events are sent to related active agents. A context switch occurs whenever control transfers between the debugging core and either a buggy program or an external agent. Event forwarding is accomplished without the knowledge of the external agent, which means the standalone external agent needs no modification to be loaded and used by UDB.

6-3 INTERNAL DEBUGGING AGENTS

Besides support for whole programs as external agents, UDB supports insertion of dynamic analyses into the debugging core as listener agents that implement a set of callback methods. UDB’s debugging core implements different integrat-
ed agents for different classes of bugs. The Internals component handles the integrated agents; it registers internal agents during initialization and checks which agents are active and calls the related underlying method(s) based on the event code that is received by the debugging core.

6-4 MIGRATION FROM EXTERNALS TO INTERNALS

External agents allow automatic debugging techniques based on various dynamic analyses to be developed and tested easily in the production environment. Selected external agents may become internal. Internal agents do not pay the lightweight cost of the context-switch communication between the debugging core and the external agents.

UDB provides smooth migration from external agents to internal. The first issue in migration is to accept a callback-style event listener architecture in place of the more general main() procedure that an external agent uses from a separate thread. UDB provides an abstract class called Listener, which must be subclassed within the external agent before the external can be used as internal. The Listener class allows the debugging core to acquire the event mask of the migrated internal agents, and to determine which listener methods to use for the various events.

An object of the newly migrated internal agent must be instantiated and inserted into the list of clients in the Internals class. A simple automatic registration can be done through the Internals class method register(); where its first parameter associates the agent with a formal name as an ID during the debugging session, and the second parameter is an object of that agent class. Furthermore, the method register() can be used with three extra parameters to register user-defined agent features such as handlers, analyzers, and writers respectively.

The new internal agent must be stripped of its main() procedure before compilation and linking into the debugging core. Alamo primitives found in the external agent are no longer needed when it becomes an internal agent. For example, Alamo’s EvInit() primitive is discarded automatically, it is needed once per each buggy program and it is already performed by the debugging core.
Fig. 4 showed a simple UDB extension agent. Fig. 5 above shows the migration of that agent from standalone program (external agent) to internal agent. Each monitored event is mapped, in a one-to-one relation, onto a single method. This convention allows the debugger to provide automatic registration for the event callback methods and the agent’s event mask. The agent’s class has three kinds of methods that are recognized by the automatic registration process. Agents are registered automatically with four simple steps:

1. Derive the agent class from the Listener class provided by UDB. This abstract class analyzes the derived class looking for the three kinds of event handlers. It builds a table that maps each prospective event into its handler method, and builds the agent’s event mask and updates the core of the ex-

```cpp
#include "evdefs.icn"
link evinit

class Example : Listener
    mask, pcalls, fcalls, prate, frate
method handle_E_Pcall()
    pcalls += 1
end
method handle_E_Fcall()
    fcalls += 1
end
method analyze_info()
    total := pcalls + fcalls
    prate := pcalls / total * 100
    frate := fcalls / total * 100
end
method write_info()
    write(" # pcalls = ", pcalls, " at rate : ", prate)
    write(" # fcalls = ", fcalls, " at ratio : ", frate)
end
initially()
    mask := cset(E_Pcall || E_Fcall)
    pcalls := fcalls := 0.0
end
procedure main(args)
    EvInit(args)
    obj := Example()
    while EvGet(obj.mask)
        case &eventcode of
            E_Pcall: obj.handle_E_Pcall();
            E_Fcall: obj.handle_E_Fcall();
        end
        obj.analyze_info();
        obj.write_info()  
end
```

Figure 5 Sample migrated agent.

6-4-1 MIGRATING AUTOMATICALLY REGISTERED AGENTS

A. Standalone External Agent

B. Migrated to Internal Agent
tended debugger to request those events from the execution of the buggy program.

2. Place a call to the `register()` method in the `Init()` method of the `Internals` class as follows: `register("calls", Example())`

3. Strip the agent’s `main()` procedure, and

4. Compile and link the migrated agent into the extended debugger executable.

When the process of migration has completed successfully, users can use their own agents from within the host debugger as internal agents during the debugging session. Agents are distinguished by their names. The user can enable or disable the agent facilities on the fly during the debugging session by referring to their names.

6-4-2 MIGRATING MANUALLY REGISTERED AGENTS

Arbitrary agents do not follow the naming convention discussed in Section 6-1. The method names of these agents have no restriction. However, the user has to classify the agent’s methods into `handlers`, `analyzers`, and `writers`. This kind of agent is registered in a similar way to the simple agents discussed in the previous section. However, the programmer must place a call to the `register()` method of the `Internals` class with four extra parameters, which are used to register the handler methods, analyzer methods, writer methods, and the agent’s event mask. Fig. 6 shows the call to the `register()` method that manually registers the `Example` class as shown in Fig. 4.

This type of registration provides users with enough freedom to write their own standalone agents in the way they want, and allow them to integrate those as internals with the least possible modifications. Moreover, this explicit registration does not disable the automatic registration; the automatic registration is always applied. If there is any method that is following the naming convention introduced earlier, they are automatically registered. This explicit registration provides an addition on top of the automatic registration, and removes the restriction of one handler per-event required in the automatic registration.

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

Figure 6 Explicit agent registration.
7- EVALUATION

One of the biggest considerations in the design of an event-based source-level debugger is the performance in terms of space and time. Most event-driven debuggers suffer from a scalability problem because they must handle a huge volume of trace data. UDB and its agents handle events on the fly; some events are used to control the execution of the target program, while other events supply information about the execution state. UDB’s extension agents are task-oriented with specific debugging and analysis missions. Those agents often deal with a relatively small volume of traced data that does not have big performance related problems in terms of storing or handling. Most UDB extension agents maintain traced data in memory without any problem.

UDB spends a considerable amount of time processing events from the instrumentation provided by AlamoDE. The primary cost of these events is due to the context switches between UDB and the buggy program. Additional time is spent on the event filtering and processing inside UDB’s debugging core (the debugging evaluator).

In order to find how much slowdown is imposed on a program running under UDB, the time is measured for a Unicon program running on a Linux machine without UDB, then the time is measured for the same program running on the same machine but under UDB; different runs were performed and in each run one of the debugging techniques was enabled, the measured time in Table 1 is the average of three different runs. The program is the Unicon translator itself, translating a large (1200 LOC) Unicon module named idol.icn. This translator size is 609KB and the virtual machine size is 790KB. The time, in this experiment, is measured using the Linux command `time` on a laptop with Genuine Intel CPU T2050, 1.60 GHz, Centrino Duo, RAM 2.0 GB. The machine runs openSUSE 10.3 with Linux kernel 2.6.5-31-default.

The time command gives timing statistics about a specific program (in this case the Unicon virtual machine running UDB). The result is divided into three categories: 1) real time, which represents the elapsed time between the start of the VM and its termination; this time includes loading the debugger and its target program, 2) the user time, which is the total number of CPU-seconds that the process spent in user mode, and 3) the system time, which is the system time or the total number of CPU-seconds that the process spent in kernel mode. For the sake of this experiment and in order to reduce the human factor, UDB is automated to load the target program and the debugging extension agents without the user interaction.

Table 1 shows that UDB at present provides reasonable performance for ordinary debugging operations. The AlamoDE architecture has been shown viable for debugging, but it will become more attractive with further tuning. Additional VM support for trapping instructions and variables will benefit the performance of breakpoints and watchpoints using known techniques. More serious performance slowdowns are associated with various automatic debugging techniques,
which employ dynamic analyses of varying complexity in order to function. In practice programmers can enable/disable individual techniques between breakpoints or between steps in the debugger. Programmers only have to pay for expensive features when they need them, but further reducing the cost of the various automatic debugging features is a key to making them practical.

Table 1 Execution time of different UDB debugging features.

<table>
<thead>
<tr>
<th>Feature</th>
<th>real</th>
<th>user</th>
<th>sys</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal (No UDB)</td>
<td>1.17</td>
<td>0.99</td>
<td>0.12</td>
</tr>
<tr>
<td>Under UDB</td>
<td>1.39</td>
<td>1.17</td>
<td>0.11</td>
</tr>
<tr>
<td>Breakpoint</td>
<td>3.68</td>
<td>3.45</td>
<td>0.14</td>
</tr>
<tr>
<td>Watchpoint</td>
<td>7.98</td>
<td>6.51</td>
<td>1.38</td>
</tr>
<tr>
<td>Detect Variable Type Change</td>
<td>14.17</td>
<td>11.24</td>
<td>2.77</td>
</tr>
<tr>
<td>Detect Subscript Fails</td>
<td>14.91</td>
<td>11.41</td>
<td>3.33</td>
</tr>
<tr>
<td>Detect Zero Time Loops</td>
<td>15.01</td>
<td>12.17</td>
<td>2.66</td>
</tr>
<tr>
<td>Tracing Type Conversion</td>
<td>8.99</td>
<td>6.74</td>
<td>1.98</td>
</tr>
<tr>
<td>Tracing All Procedure Activities</td>
<td>3.37</td>
<td>2.79</td>
<td>0.49</td>
</tr>
<tr>
<td>Tracing String Scanning Activities</td>
<td>2.38</td>
<td>1.98</td>
<td>0.27</td>
</tr>
<tr>
<td>Tracing Procedure Calls</td>
<td>2.19</td>
<td>1.77</td>
<td>0.28</td>
</tr>
<tr>
<td>Tracing Built-in Function Calls</td>
<td>4.61</td>
<td>3.75</td>
<td>0.79</td>
</tr>
</tbody>
</table>

In order to evaluate the current state of UDB’s advanced debugging techniques, a suite of six different debugging agents, which are discussed in Section 5-2, were loaded and tested as external agents under UDB, and then migrated to become part of UDB’s library of internal agents. The slowdown imposed by the external agents was at most 3 times slower than the standalone agent mode, and the slowdown imposed by the migrated internal agents was at most 2 times slower than the standalone agents, see Fig. 7.

The entire suite of agents imposes at most 20 times slowdown on the execution of the buggy program over an uninstrumented execution mode. In the general case, the slowdown depends on the algorithms used by the dynamic analysis. To place this in perspective, a debugger such as valgrind [8] imposes a 20 to 50
times slowdown, and it does not provide the interactive debugging environment that UDB provides, where the user can be selective about which agents to use and where to enable them in a debugging session. In practice programmers only have to pay for expensive features when they need them.

Figure 7 Agent monitoring time.

8- RELATED WORK

Standard source-level debuggers such as GDB [9] and its graphical front end DDD [10] provide convenient debugging and tracing facilities. But using a conventional source-level debugger is still time consuming; it is largely based on forming a hypothesis and guessing where to place break points. One way to reduce the debugging time is to automate the debugging process. Automated debugging is a challenging problem that goes back to the 60’s [11] and mid 70’s [12]. Its most challenging part is the reasoning about the information supplied by the debugging tool, which is still heavily depends on the human factor.

Many static and dynamic analysis tools have been built in order to address the limitations of the conventional source-level debuggers. C/C++ static analysis tools such as Splint [13], the \texttt{-Wall} option in gcc compiler [14], and the Code-
Surfer [15] provide the ability to detect some common language misuses and memory related bugs. ESC/Java [16] is a compile-time program checker that finds common runtime programming errors. Dynamic analyses tools such as BoundsChecker [17], Electric fence [18], mpatrol [19], Purify [20], and valgrind [10] provide dynamic error detection for some typical runtime bugs like dangling pointers or memory leaks.

Each automated tool addresses one or more of the most common hard to detect bugs. Some tools work as a standalone tool while other tools work as part of a compiler or a source-level debugger. However, there is a third group of tools that are general enough to be considered debuggers, consisting of tools based on tracing techniques such as the Omniscient Debugger (ODB) [21, 22], which provides the debugging facilities by tracing the complete program history of states. In this debugger, the program must be traced first before the debugging techniques can be used, but it provides the ability to investigate backward in the execution history. The WhyLine debugger [23] traces every statement in the program history to be able to answer questions of the type why did? and why did not?. Coca [24] provides automatic debugging for C; it maintains a data base and provides Prolog primitives as its user interface. Many automated and advanced debugging techniques such as algorithmic debugging, event grammars [25, 26], and delta debugging [27] provide automation for specific kinds of bug hunts. Another trace-based tool is the IBM Jinsight [28], which combines tracing and visualization techniques. It depends on a special modified (instrumented) version of the Java Virtual Machine, and generates a huge amount of traced data in a short period of time. In general, the most common problem of trace-based debuggers is the huge amount of traced data that limits the scalability and raises the level of complexity. An example of a recent tool is JDLab [29], which applies the graph theoretical algorithms to reduce the amount of traced events.

UDB’s approach is to provide an extensible source-level debugger that allows different separately-compiled dynamic analysis tools to be dynamically-loaded and used synchronously in a typical debugging session without modification. UDB preserves the debugging techniques found in classical debuggers such as GDB, while providing a simple extension interface to load standalone external agents on the fly during a debugging session or incorporate them as permanent features in the debugger core. UDB utilizes agents with new automatic detection techniques that could be found in trace-based debuggers such as ODB [21,22]. In contrast, ODB only provides a navigation tool for execution history. When it comes to changing the state of the running program during the debugging session, ODB forces the user to trace the complete program first, before the user is able to trace-back and re-start the execution from some middle point with new value assigned to a variable. UDB provides reasonable performance for ordinary debugging operations, with additional VM support needed for classical debugging features such as breakpoints and watchpoints.

UDB customizes runtime monitoring of the buggy program on the fly based on
the current debugging context; an event is not reported if it is irrelevant to the current set of active debugging facilities. It makes the monitoring cost relatively low most of the time, and it avoids dealing with a huge volume of traced data. In contrast, some automatic debugging tools incur much higher complexity and sacrifice scalability by storing every single state of the program in order to be able to answer the user’s questions and trace back the execution [21,22,23].

9- CONCLUSIONS AND FUTURE WORK

UDB is an event-driven debugger with an agent-based extension mechanism that provides many advantages over traditional source-level debuggers, such as: 1) it simplifies the design of automatic debuggers by breaking the debugging task into small task-oriented agents, 2) it provides more debugging features than a conventional debugger by means of agents performing dynamic analysis and automatic debugging missions, 3) it supports the ability to employ agents, in the conventional debugging session, only when they are needed, 4) it provides a simple interface to load, unload, enable, or disable agents on the fly from any point during the interactive debugging session, and 5) it enables users to write their own custom user-defined agents and use them as external agents or incorporate them as internal permanent debugging features.

UDB’s agents are used to locate a growing set of suspicious behaviors from within a conventional debugging interface. Instead of recording the complete program state and letting the user investigate, UDB’s agents monitor the execution of the program while it runs and watch for specific behaviors that may indicate a bug. This has the dual advantages of better scalability and providing answers on the fly.

Compared with the slowdown of many automatic debugging techniques, the performance of UDB is very good. However, the true test of UDB’s performance will be whether it enables debugging agents that justify their time cost by the value they provide to programmers.

Future work is divided into different categories; first, improve the process of debugging using UDB. This can be achieved by means such as: 1) add more agents that utilize automatic debugging techniques for classes of bugs that are difficult to catch using standard techniques, for example duplicated control logic, the use of the wrong operator, and aliasing-related bugs, 2) improve performance by adding more support for debugging inside the Unicon virtual machine, and 3) develop a concurrent monitoring framework to offload the cost of automated debugging onto additional processor cores.

Second, subsets of the Alamo framework used by UDB for Unicon debugging have been implemented for monitoring C and Python [6,7]. Future work may extend UDB’s debugging facilities to these languages. The concepts introduced in UDB are independent of its AlamoDE implementation. After UDB is tested and validated for Unicon, it may be applied to other languages. For example, UDB
might be implemented for other languages by utilizing a sophisticated instrumentation framework such as ASM [30] for Java, or PIN [31] or Atom [32] for C/C++. The long-term goal is for UDB and the analyzers developed for it to support other mainstream languages after they have been validated on the Unicon platform, whose high level makes it amenable to experimental work. Building an Eclipse plug-in for UDB after supporting other languages will improve its usability.

ACKNOWLEDGMENT

This research was supported by the National Library of Medicine Specialized Information Services Division and 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.

REFERENCES


