An Agent-Oriented Source-Level Debugger on Top of a Monitoring Framework

Ziad Al-Sharif, Clinton Jeffery
Computer Science Department
University of Idaho
zsharif@ieee.org, jeffery@cs.uidaho.edu

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. The paper tests three hypotheses: 1) a source-level debugger built on top of a monitoring framework can surpass ordinary debuggers with more debugging capabilities, 2) a debugger based on a high-level monitoring framework allows easier and more efficient agent-based extension, and 3) an agent-oriented debugger is easier to extend on the fly with new agents that utilize automatic debugging and dynamic analysis techniques.

Key Words: 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 good, but they are not always successful 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 someday be widely adopted in future debuggers.

This paper describes an agent-oriented source-level debugger named UDB, which divides the debugging process over multiple simultaneous agents. An agent is an event-driven task-oriented program execution monitor; it can be loaded on the fly 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 the conventional source-level debugging session, and the user can be selective about which agent(s) to use.

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 the conventional source-level debugging session, and the user can be selective about which agent(s) to use.

UDB’s debugging 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. Different agents can be loaded and active, and each agent receives different runtime events based on their own request, which is defined as an event mask; a detailed event filter for the current set of desired events. Agents are coordinated by a central debugging core. 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 decision 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 discusses the monitoring framework’s suitability for debugging. Section eight presents an evaluation of UDB’s performance. Section nine compares UDB’s features with related work. Section ten provides the conclusions and discusses planned future work.

2. Background

Unicon [7] is an object-oriented dialect of Icon [5], 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 is an event-driven debugger that facilitates the runtime execution events provided by AlamoDE, which is a recent extension to the Alamo monitoring framework [6,8,16]. The general concept of UDB is 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 [3] for Java, or PIN[14] and Atom[15] 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.

3. UDB sample session

UDB’s user interface resembles GDB’s console based interface. UDB includes most of the commands supported by GDB, it also adds a handful of commands to handle and manage extension agents from any point during the debugging session. The user can provide a command to load, unload, enable, or disable agents in the console based interface. UDB’s external agents are enabled by default at load time, whereas its internal agents need to be explicitly enabled. Figure 1 shows UDB’s sample debugging session where the loop agent is used to watch zero iterated while loops in the sort program.

```
$ udb sort
  sort: loaded 2.5K bytes of 32-bit uncompressed icode
  1 Source file(s) found
  Type “help” for assistance
(udb) break BubbleSort
  Breakpoint set successfully in:
    # sort.icn(5): BubbleSort( A )
(udb) run
  A = [4,1,8,9,0,6,5,7,2,3]
  Breakpoint: 
    sort.icn(5): BubbleSort( A )
(udb) watch -agent=loop -name=while -iteration=0
(udb) cont
  loop: failed while
    test.icn(10): while swapped ~== "true" do{
(udb) quit
  sort is running, are you sure you want to quit,(Y/n)?y
  Thank you for using UDB, Goodbye!
```

Figure 1. Sample UDB session

4. Design

UDB is based on the thread model of execution monitoring, where a debugger and a buggy program are in 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. Switching between any two co-expressions is done through a small piece of assembler code that performs a lightweight context switch. The state of the program is saved and the control is transferred into the other program without the involvement of the operating system. Because they are synchronous, co-expression switches are much faster than typical thread switches such as those provided by the pthreads library.

Different debugging agents run in different co-expressions. 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 these 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.
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.

The session and the evaluator are *generators*; expressions that suspend values to the caller and can be resumed to produce additional values [5,7]. 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 handling the control to the console. This mechanism provides the user with the ability to continue the debugging by resuming the generator of the debugging session to its previous state and resuming the evaluator to the point that was suspended.

4.2. Source code

UDB’s source code is organized around its five major components. The user interface 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 BreakPoint, WatchPoint, Stepping, Stack, Data, and Tracing classes, and 2) the extension agents interface is modeled by the Agent,Externals, Internals and Listener classes. See Figure 3.

5. Implementation

UDB implements its classical debugging techniques as well as its 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 of advanced debugging techniques. This experimentation probably would not be undertaken if the implementation was limited to one of the low-level approaches found in other debuggers. This section discusses UDB’s implementation approaches of classical debugging techniques, and introduces some of its agents.

5.1. Classical debugging techniques

5.1.1. Loading a buggy program. Alamo’s `load()` primitive reads a program, sets it up with its own code, static data, stack and heap, without linking symbols into the current program. With instrumentation located in the VM and runtime system, this behavior provides no intrusion on the buggy program space, which is ideal for debugging.

5.1.2. Breakpoints. UDB’s breakpoints are implemented by monitoring the line number event `E_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. Using the value mask, an extra condition, which is the value of the event, can be associated with the line number event `E_Line`. In this implementation, there is only a context switch if the line number event and its value are satisfied. For the sake of better monitoring consistency, a breakpoint on a procedure/method is converted internally into a breakpoint on a line number which is the line number of the procedure’s header.

5.1.3. Watchpoints. UDB’s watchpoints are implemented by monitoring the assignment event `E_Assign` only when that assignment has a predefined watchpoint on it. This implementation is approximated by utilizing the value mask over the `E_Assign` event in a technique similar to the one discussed in section 5.1.2.

5.1.4. Stepping and continuing. UDB implements the basic `step` and `next` commands using the line number event `E_Line`. However, the implementation of the `next` command ensures that the event from the line number change `E_Line` is never preceded by any procedure call event `E_Pcall`. If a procedure call event occurs, 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 either one of them is 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 is confident that it has found a bug and in others it issues an appropriate warning. Either way, the combination of valgrind-style [11] dynamic analysis within an interactive debugger makes both methods more effective.

### 5.2.1. Variable changing type (or domain)

Unicon is a dynamically typed language; no variable declaration is needed and a variable can be assigned values of different types. Such type changes are not a good programming practice; it can indicate a logical error and/or complicate any reading of the source code. This agent 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 8.

### 5.2.2. Uninitialized and dead variables

Uninitialized and dead variables are variables that are read and never assigned or assigned and never 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 enough to warn the user about them.

This agent tracks referenced variables based on their scope. For example, local variables are monitored over different calls before they can be 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_Lref`, `E_Lsub` 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. However, sometimes a loop 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 be unexplainable, 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 most common performance bugs result 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 assigned value happened. The main monitored event is `E_Assign`, which is reported at every assigned variable. If that variable value caused a 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. External debugging agents

Multiple simultaneous 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.

6.2. Internal debugging agents

Besides support for whole programs as external agents, UDB supports insertion of dynamic analyses into the debugging core as a listener agent that implements a set of callback methods. UDB’s debugging core implements different integrated 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.

6.3. 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.

7. Monitoring framework

AlamoDE provides control over the execution of the buggy program with efficient instrumentation and no intrusion on the buggy program space. AlamoDE is integrated in the Unicon language with no measurable cost (other than 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 and the buggy program run in two different co-expressions and the buggy program is the only one affected by the instrumentation.

AlamoDE provides a comprehensive set of execution event. 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 pair of code and value, which minimizes the processing overhead. Alamo’s support for dynamic event customization provides the ability to change the set of inquired events on the fly by adding/removing events’ codes to/from the event mask. Event filtering based on events’ values substantially reduces the amount of reported events and the number of context switches.

AlamoDE offers the ability to investigate the state of the buggy program. It provides both primitives and events to inquire information about the current: 1) 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 frames, and 3) global/local variables state; their names, values, and types. Furthermore, it provides the ability to change
the execution state by assigning to variables, which is important for interactive debugging.

8. Evaluation

In order to evaluate the current state of the 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. 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 [11] 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/disable them from within a breakpoint based debugging session.

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.

9. Related work

Trace-based debuggers such as the ODB [10,13] and the WhyLine debugger [9] provide advanced debugging techniques by recording the whole program’s history of execution in order to provide an answer for almost any question that can be asked. Coca [4] 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 [1,2], and delta debugging [17] provide automation for specific kinds of bug hunts.

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 [10,13]. 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 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 [9, 10,13].

10. Conclusion and future work

An event-driven debugger with agent-oriented extensions provides many advantages over traditional source-level debuggers such as: 1) it simplifies the design by breaking the debugging task into small task-oriented agents, 2) it provides more debugging features through 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; it provides a simple interface to load, unload, enable, or disable agents on the fly from any point during the interactive debugging session, and 4) 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 with 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 and watch for specific behaviors that may indicate a bug. This has the advantage 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 (i.e. duplicated control logic, wrong operator, and aliasing-related bugs), 2) improve performance by adding more support for debugging inside the Unicon virtual machine, and 3) investigate the performance of 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 [8,16]. Future work may extend UDB’s debugging facilities to these languages. Building an Eclipse plug-in for UDB after supporting other languages will improve its usability.

11. Acknowledgment

This research 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.

12. References


