An Extensible Source-Level Debugger

Ziad Al-Sharif
Computer Science Department
University of Idaho
zsharif@vandals.uidaho.edu

Clinton Jeffery
Computer Science Department
University of Idaho
jeffery@cs.uidaho.edu

ABSTRACT
Standard debuggers are limited in the amount of analysis that they perform in order to assist with debugging. This paper presents UDB, a 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 automatic debugging extensions. UDB tests the hypotheses that a debugger built on top of a high-level framework enables better debugging capabilities as well as easier and more efficient extension than ordinary debuggers.

Categories and Subject Descriptors
D.2.5 [Software Engineering]: Testing and Debugging – code inspections and walk-throughs, debugging aids, monitors

General Terms
Design.

Keywords
Automatic Debugging, Debugging Agent, Source level Debugger, Monitoring Framework.

1. INTRODUCTION
Classical debuggers allow developers to observe the state of the program using techniques such as breakpoints, 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. Experimentation is needed in order to develop the features that will someday be widely adopted in future debuggers.

This paper introduces a debugger named UDB for the Unicon language. Unicon [4] is an object-oriented dialect of Icon [1], 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 such as UDB.

UDB is a source-level debugger that: 1) monitors the execution of a program for 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, 2) provides new debugging techniques, which utilize execution history prior to the current state, techniques that are more often found in trace-based debuggers such as ODB[5], 3) is written in a very high level language, which reduces its maintenance cost and simplifies its extension with new debugging techniques, 4) integrates built-in automatic detection techniques, and 5) provides a simple interface to load, unload, enable, or disable separately compiled dynamically loaded external debugging agents to work in conjunction with the conventional interactive source-level debugging session. In this paper, a debugging agent is an event-driven task-oriented program execution monitor.

External agents are written and tested as standalone tools and then loaded and managed on the fly by the debugging core during a UDB debugging session. Active agents are paused whenever the debuggee is paused and resumed when the debuggee resumes. Successful external agents may be promoted to internal built-in monitors within the debugging core for improved performance. Internal agents do not pay the cost of the context-switch communication between the debugging core and the external agents. UDB provides smooth migration from external agents to internal. Migration entails structuring as a callback-style event listener architecture in place of the main() procedure that an external agent uses from a separate thread.

UDB employs agents of standard and automated techniques to locate numerous bugs and potential bugs based on symptoms such as 1) variables that change type during execution, 2) variables that are read and never assigned or assigned and never read during a particular execution, 3) expressions that fail silently in contexts where failure is not checked, 4) loops that consistently execute zero times, and 5) redundant implicit type conversion. UDB provides commands to watch such suspicious behaviors, and look up interesting source locations such as where a specific variable was assigned prior to causing a crash.

2. UDB SESSION
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. UDB’s user interface resembles GDB’s console based interface, with a handful of extra commands to handle and manage its agent extension support. See Figure 1.

```
$ udb sort
..........
(udb) run
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)
```

Figure 1. Sample UDB session

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SAC ’09, March 8-12, 2009, Honolulu, Hawaii, U.S.A.
Copyright 2009 ACM 978-1-60558-166-8/09/03...$5.00.
3. DESIGN

UDB employs the thread model of execution monitoring, where a debugger and a debuggee program run in separate threads in a shared address space. UDB is built atop a monitoring framework oriented towards program visualizations called Alamo [3], which was extended with capabilities called AlamoDE (Alamo Debug Enabled). UDB monitors the debuggee program for execution events. It customizes events 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.

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 debugging session that initializes and manages the state of the debugger and controls the debugging evaluator, 3) a debugging evaluator that provides the main event-driven debugging analysis and monitoring control, 4) an agent interface that facilitates and provides the programming interface for external and internal extensions, and 5) a debugging state that maintains the debugging information between the debugging components.

The debugging session and the debugging evaluator are generators, expressions that suspend values to the caller and can be resumed to produce additional values [1,4]. The diagram in Figure 2 shows the control flow inside UDB and how its generators are related to each other. The evaluator generator provides the ability to suspend the main monitoring loop without losing its state. The session generator provides implicit ability to maintain the debugging session and the state of the evaluator generator before handling the control to the console. This mechanism provides the capacity 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. EVALUATION

An AlamoDE based debugger uses different approaches to implement features found in standard source-level debuggers, and faces potential performance challenges. In compensation, this type of implementation greatly simplifies the process of experimenting with new advanced debugging techniques that probably would not be undertaken if the implementation was limited to one of the low-level approaches found in other debuggers.

While evaluating UDB’s advanced debugging techniques, a suite of six different debugging agents 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 debuggee program over an un-instrumented execution mode.

In general the slowdown depends on the algorithms used by the agent’s dynamic analysis. To place this in perspective, a debugger such as valgrind [6] imposes a 20 to 50 times slowdown, and it does not provide the interactive debugging environment that UDB provides, where the user can selectively enable or disable agents between steps in a debugging session. In practice programmers only have to pay for expensive features when they need them.

5. CONCLUSION AND FUTURE WORK

An elusive bug can be characterized as having symptoms such as: 1) being a bug in one situation but not in another, 2) causing a logical error and/or an incorrect result instead of a core dump or a runtime error, and 3) being revealed long after its root cause.

Elusive bugs are difficult to catch using standard techniques. UDB provides the user with agent-based automated techniques to catch those bugs based on suspicious execution behaviors. In some cases the debugger is confident that it has found a bug and in others it reports an appropriate warning. Either way, the combination of valgrind-style [6] dynamic analysis within an interactive debugger provides more effective debugging.

Subsets of the Alamo framework used by UDB have been implemented for C and Python. Future work may extend UDB’s debugging facilities to those languages, or port UDB to run on other debugging platforms such as JPDA [2].

6. 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.

7. REFERENCES