

**An Extensible Debugging Architecture
Based on a Hybrid Debugging Framework**

A Dissertation

Presented in Partial Fulfillment of the Requirement for the
Degree of Doctor of Philosophy

with a

Major in Computer Science

in the

College of Graduate Studies

University of Idaho

By

Ziad A. Al-Sharif

December 1, 2009

Major Professor: Dr. Clinton L. Jeffery

Copyright © 2008-2009 Ziad Al-Sharif. All rights reserved.

Authorization to Submit Dissertation

This dissertation of **Ziad A. Al-Sharif**, submitted for the degree of Doctor of Philosophy with a major in Computer Science and entitled “**An Extensible Debugging Architecture Based on a Hybrid Debugging Framework**,” has been reviewed in final form. Permission, as indicated by the signatures and dates given below, is now granted to submit final copies to the College of Graduate Studies for approval.

Major Professor: _____ Date: _____
Dr. Clinton L. Jeffery

Committee Member: _____ Date: _____
Dr. Robert B. Heckendorn

Committee Member: _____ Date: _____
Dr. Robert Rinker

Committee Member: _____ Date: _____
Dr. Gregory W. Donohoe

Department
Administrator: _____ Date: _____
Dr. Gregory W. Donohoe

Discipline’s
College Dean: _____ Date: _____
Dr. Donald M. Blacketter

Final Approval
and Acceptance
by the College of
Graduate Studies: _____ Date: _____
Dr. Margrit Von Braun

Abstract

The cost of writing debuggers is very high. Most debuggers are written employing low level operating system and hardware specific code, which is hard to port to new platforms or architectures and to extend with new debugging techniques. Moreover, current debuggers are usually limited in the amount of analysis that they perform and the level of detail that they provide in order to assist with debugging. Most debuggers are well suited for a specific class of bugs. Different bugs call for different debugging techniques, so experimentation is needed in order to develop the features that will someday be widely adopted in debuggers. This dissertation contributes three primary results.

First, it introduces an event-driven debugging framework named AlamoDE (Alamo—Debug Enabled). The role of this framework is analogous to an abstraction layer upon which to build debuggers. AlamoDE 1) provides in-process debugging support with simple communication and no intrusion on the buggy program space, 2) enables debugging tools to be written at a high level of abstraction, and 3) facilitates developers of experimental automatic debugging features in a very high level language. AlamoDE supports construction of a variety of user-defined debugging tools that range from classical source-level debuggers to automated and dynamic analysis techniques.

Second, this dissertation presents an extensible agent-based debugging architecture named IDEA (Idaho Debugging Extension Architecture). IDEA offers novel debugging techniques that break the rigidness, closeness, and inextensibility of most current debuggers. It provides programmers with the ability to easily implement, test, and combine user-defined debugging agents, and offers a simple dynamic and static extension mechanism.

Finally, this dissertation provides a production source-level debugger for the Unicon language named UDB. UDB leverages the classical interactive debugging process with 1) built-in agents employing automatic detection and dynamic analysis techniques, 2) a simple interface to load, unload, enable, and disable separately-compiled dynamically-loaded external debugging agents that work in conjunction with the conventional source-level debugging session and its internal agents, and 3) Dynamic Temporal Assertions (DTA) that assert a sequence of runtime properties. DTAs are introduced for the first time within typical source-level debuggers that targets sequential programs.

While IDEA simplifies a source-level debugger's extensibility and eases its usability, debugging agents add indispensable value with moderate impact on the performance of the debugger. Different agents can work in concert with each other to provide programmers with better understanding of the program's execution behavior and simplify the process of debugging and hunting for elusive and hard to catch bugs.

Curriculum Vita

Ziad A. Al-Sharif
Department of Computer Science
University of Idaho
Moscow, ID, 83844
zsharif@gmail.com

Future Address

Department of Computer Science
Jordan University of Science and Technology
Irbid, Jordan, 22110
P.O.Box 3030

Education

December 2009

Doctor of Philosophy in Computer Science
Department of Computer Science, University of Idaho, Moscow, ID, 83844
GPA: 4.0 out of 4.0

August 2005

Master of Science in Computer Science
Department of Computer Science, New Mexico State University, Las Cruces, NM, 88001
GPA: 3.559 out of 4.0

February 2000

Bachelor of Science in Computer Science
Department of Computer Science, AL al-Bayt University, Mafraq, Jordan
GPA: 81.15% (very good)
Ranked second among my peers in the graduation ceremony of 2000

Publications

Journal Articles

1. Ziad Al-Sharif and Clinton Jeffery, 2010. UDB: An Agent-Oriented Source-Level Debugger. To appear in the *International Journal of Software Engineering (IJSE)*, Vol. 3, No. 1, January 2010.

Conference Papers

1. Ziad Al-Sharif and Clinton Jeffery, 2009. Language Support for Event-Based Debugging. In *Proceedings of the 21st International Conference on Software Engineering and Knowledge Engineering (SEKE 2009)*, Boston, July 1-3, 2009. pp. 392-399. (Acceptance rate 38%).
2. Ziad Al-Sharif and Clinton Jeffery, 2009. A Multi-Agent Debugging Extension Architecture. In *Proceedings of the 21st International Conference on Software Engineering and Knowledge Engineering (SEKE 2009)*, Boston, July 1-3, 2009. pp. 194-199.
3. Ziad Al-Sharif and Clinton Jeffery, 2009. An Agent Oriented Source-level Debugger on Top of a Monitoring Framework. In *Proceedings of the Sixth International Conference on Information Technology: New Generations* (Las Vegas, Nevada, April 27 - 29, 2009). ITNG. IEEE Computer Society. pp. 241-247. (Acceptance rate 29%).
4. Ziad Al-Sharif and Clinton Jeffery, 2009. An Extensible Source-Level Debugger. In *Proceedings of the 2009 ACM Symposium on Applied Computing* (Honolulu, Hawaii, March 9-12). SAC '09. pp. 543-544. (a 2 page abstract with poster).
5. Hani Bani-Salameh, Clinton Jeffery, Ziad Al-Sharif, and Iyad Abu Doush. 2008. Integrating Collaborative Program Development and Debugging within a Virtual Environment. In *Groupware: Design, Implementation, and Use: 14th International Workshop, CRIWG 2008. Omaha, NE, USA. September 14-18, 2008*. Lecture Notes in Computer Science, vol. 5411. Springer-Verlag, pp. 107-120.
6. Ziad Al-Sharif and Clinton Jeffery, 2006. Adding High Level VoIP Facilities to the Unicon Language. In *Proceedings of the Third International Conference on Information Technology: New Generations* (April 10 - 12, 2006). ITNG. IEEE Computer Society. pp.524-529.
7. Clinton Jeffery, Omar El-Khatib, Ziad Al-Sharif, and Naomi Martinez, 2005. Programming Language Support for Collaborative Virtual Environments. In *Proceedings of the International Conference on Computer Animation and Social Agents (CASA'05)*.

Technical Reports

1. Ziad Al-Sharif. Debugging with UDB 2.0: User Guide and Reference Manual. Unicon Technical Report #10, <http://unicon.org/utr/utr10.pdf>. December 2009.
2. Ziad Al-Sharif, A High Level Audio Communications API for the Unicon Language, Master Thesis, Department of Computer Science, New Mexico State University, August 2005.

Acknowledgment

I would like to express my gratitude to all the people who helped me develop this dissertation. It has been a long road involving major research challenges and overcoming critical development obstacles. First, I would like to thank my major professor Dr. Clinton L. Jeffery, whom without this dissertation could not have been written. He did not only serve as my supervisor but also encouraged and challenged me throughout my academic program. He never accepted less than my best efforts. I thank him for all his encouragement, patience, support, and knowledge.

I also would like to thank my committee members: Dr. Robert Rinker, Dr. Robert Heckendorn, and Dr. Gregory Donohoe, who did not save any effort to provide valuable and constructive criticism about the presented research and the formatting of this dissertation. Their significant feedback guided me through the dissertation process; I would like to thank them all.

I also would like to thank Dr. Phillip Thomas from the National Library of Medicine. Dr. Thomas was involved in this research from the beginning. He and his research group at the National Library of Medicine were benevolent to test the UDB debugger on their programs and report back with thoughtful feedbacks and suggestions. Dr. Thomas was kind enough to voluntarily read the early draft of this dissertation. He has provided me with significant enhancements.

This research was supported by the National Library of Medicine Specialized Information Services Division, initially through 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. I would like thank their endless support. Finally, I would like to thank the Department of Computer Science at Jordan University of Science and Technology (JUST) for their generous sponsorship, which allowed me to pursue my MS. and Ph.D. degrees.

Ziad A. Al-Sharif

December 1, 2009

Table of Contents

Authorization to Submit Dissertation	i
Abstract	ii
Curriculum Vita	iii
Acknowledgment	v
Table of Contents	vi
List of Figures	xiv
List of Tables	xviii
List of Equations	xx
Part I Introduction and Background.....	1
Chapter 1 Introduction and Objectives.....	2
1.1. Dissertation Scope.....	2
1.2. Context and Motivation.....	3
1.3. The Problem	4
1.4. The Solution Approach Used in This Research	5
1.4.1. Debugging Framework.....	6
1.4.2. Extension Architecture	7
1.4.3. Very High Level Debugger	8
1.4.4. Extension Agents	9
1.5. The Results	9

1.6. Definitions	10
1.7. Dissertation Outline.....	10

Chapter 2 Background13

2.1. Program Bugs	13
2.2. Runtime Bugs	15
2.3. Debugging Terms	15
2.4. Debugging Tools	19
2.4.1. Architecture	20
2.4.2. Implementation.....	21
2.4.3. Interface	21
2.5. Debugging Process	22
2.6. Debugging Process Architecture	23
2.6.1. Local Debugging	24
2.6.2. Remote Debugging.....	24
2.6.3. Collaborative Debugging.....	25
2.6.4. Debugging Parallel and Distributed Systems	26

Chapter 3 Manual Debugging Tools and Techniques.....27

3.1. In-Code Debugging	28
3.1.1. Print Statements.....	28
3.1.2. Assertions	29
3.2. Dynamic Source-Level Debugging.....	29
3.2.1. Forward Debugging.....	29
3.2.2. Bidirectional Debugging	33
3.2.3. Programmable Debugging.....	35
3.2.4. Trace-Based Debugging	37
3.2.5. IDE-Based Source-Level Debugging	38
3.3. Model-Level Debugging	39
3.4. Summary	39

Chapter 4 Automatic Debugging Tools and Techniques	41
4.1. Static Debugging	42
4.2. Abstract Debugging.....	43
4.3. Dynamic Debugging.....	44
4.3.1. Model Based Software Debugging.....	44
4.3.2. In-Process Debugging (Debugging Libraries).....	46
4.3.3. Dedicated Debuggers.....	47
4.4. Summary	53
Part II Event-Based Debugging Framework	55
Chapter 5 Alamo Monitoring Framework	56
5.1. Unicon's Co-Expression Type	56
5.2. Architecture	57
5.3. Features	58
5.3.1. VM Instrumentation	58
5.3.2. Dynamic Loading	58
5.3.3. Synchronous Execution	59
5.3.4. In-process Execution Model.....	59
5.4. High-Level Execution Monitoring	60
5.4.1. Event Masking.....	60
5.4.2. Loading the Target Program.....	61
5.4.3. Activating the Target Program	61
5.5 Limitations.....	62
Chapter 6 AlamoDE: Alamo's Extensions for Debugging Support	63
6.1. Virtual Machine Instrumentation	63
6.2. Inter-Program Variable Safety	64
6.3. Syntax Instrumentation	67
6.4. High-Level Interpreter Stack Navigation	71

6.5. Signal Handling	73
Chapter 7 AlamoDE: The Debugging Framework	75
7.1. Debugging Events	75
7.2. Event Filtering	77
7.3. Execution State Inspection and Modification	78
7.3.1. Variables	78
7.3.2. Procedures and Stack Frames	80
7.3.3. Executed Source Code	81
7.4. Advanced Debugging Support	81
7.4.1. Multitasking	81
7.4.2. Event Forwarding	81
7.4.3. Custom Defined Debugging Tools	82
Part III Very High Level Extension Mechanism	84
Chapter 8 IDEA: A Debugging Extension Architecture.....	85
8.1. Debugging with Agents	85
8.2. Design	86
8.3. Implementation	86
8.4. Source Code	87
8.5. Extensions	89
8.5.1. Sample Agent	89
8.5.2. External Agents	91
8.5.3. Internal Agents	92
8.5.4. Migration from Externals to Internals	93
8.5.5. Simple Agent Migration Example	94
Chapter 9 UDB: The Unicon Source-Level Debugger	97
9.1. UDB's Debugging Features	97

9.2. Design.....	98
9.3. Debugging Core	99
9.3.1. Console	101
9.3.2. Session	101
9.3.3. Debugging State.	101
9.3.4. Evaluator.....	102
9.3.5. Generators.....	103
9.3.6. Main Debugging Loop	104
9.4. Implementation.....	104
9.4.1. Loading a Buggy Program.....	106
9.4.2. Breakpoints	106
9.4.3. Watchpoints	107
9.4.4. Tracepoints	110
9.4.5. Stepping and Continuing	111
9.4.6. Stack Navigation	113
9.4.7. Data Navigation/Modification.....	113

Part IV Extension Agents.....114

Chapter 10 UDB's Advanced Debugging Agents115

10.1. UDB's Extensibility	115
10.2. Visualization Agent.....	116
10.3. Language-Specific Agents	117
10.3.1. Variable Changing Type (or Domain).....	118
10.3.2. Failed Expressions.....	118
10.3.3. Redundant Conversion	118
10.4. Language-Independent Agents.....	119
10.4.1. Data Related Agents	119
10.4.2. Behavior Related Agents	120

Chapter 11 DTA: Dynamic Temporal Assertions	122
11.1. Temporal Assertions	122
11.1.1. Temporal Logic	123
11.1.2. Temporal Assertions vs. Ordinary Assertions	123
11.1.3. Temporal Assertions vs. Conditional Breakpoints	125
11.2. UDB's DT Assertions	126
11.3. Debugging with DT Assertions	128
11.3.1. Example #1: Loop Invariant	129
11.3.2. Example #2: Sequence of Variable States	130
11.3.3. Example #3: Variables' State from Different Scopes	131
11.4. Design	132
11.4.1. Temporal State	132
11.4.2. Temporal Interval	132
11.4.3. Assertion's Evaluation	134
11.4.5. Evaluation Suite	136
11.4.6. Temporal Assertions & Atomic Agents	137
11.4.7. Evaluation Log	138
11.5. Assertion Language	139
11.5.1. Syntax	140
11.5.2. Past-Time Operators	141
11.5.3. Future-Time Operators	141
11.5.4. All-Time Operators	142
11.6. Implementation	142
11.7. Summary	143
Part V Evaluation and Results	145
Chapter 12 Performance and Evaluation	146
12.1. AlamoDE	146
12.2. Alamo's New Extensions	150

12.2.1. Trapped Variable Assignment	150
12.2.2. Syntax Instrumentation.....	150
12.3. IDEA’s Evaluation	155
12.3.1. Procedure Call vs. Co-Expression Context Switch	155
12.3.2. Extension Agents.....	157
12.3.3. Experiment	160
12.4. UDB’s Evaluation	164
12.5. DT Assertions Evaluation	166
12.5.1. Evaluation.....	167
12.5.2. Challenges	170
Chapter 13 Conclusion and Future Work.....	171
13.1. Conclusion.....	171
13.2. Discussion	174
13.3. Limitations.....	175
13.4. Future Work	176
13.5. Extensibility to Other Languages	178
Appendices.....	179
Appendix A: Dynamic Temporal Assertions	180
A.1. Past-Time Assertions.....	180
A.1.1. Past-Time Temporal Logic Operators	180
A.1.2. Example of Past-Time Assertions	180
A.2. Future-Time Assertions	181
A.2.1. Future-Time Temporal Logic Operators	181
A.2.2. Example of Future-Time Assertions.....	181
A.3. All-Time Assertions	182
A.3.1. All-Time Temporal Logic Operators.....	182
A.3.2. Example of All-Time Assertions.....	182

Appendix B: Evaluation and Performance	183
B.1. Experimental Programs	183
B.2. Experimental Modes	184
B.3. Monitored Events.....	184
B.4. Average Monitored Events (E_Deref, E_Line, E_Syntax, E_Pcall)	192
Appendix C: UDB Command Summary	194
C.1. Essential Commands.....	194
C.2. What to Do after a Crash	194
C.3. Starting UDB	194
C.4. Stopping UDB	195
C.5. Getting Help.....	195
C.6. Executing a Program.....	195
C.7. Breakpoints	195
C.8. Watchpoints	196
C.9. Tracepoints	198
C.10. Program Stack.....	200
C.11. Execution Control.....	200
C.12. Display and Change Data	200
C.13. Source Files and Code Info	201
C.14. Memory Usage.....	202
C.15. Shell Commands	203
C.16. Extension Agents	203
C.17. Temporal Assertions.....	203
Bibliography	205

List of Figures

Figure 1.1. Dissertation's Contributions	5
Figure 1.2. Dissertation Outline	12
Figure 2.1. Sample Semantic Bug	14
Figure 2.2. Debugging Techniques	24
Figure 3.1. Manual Debugging Tools and Techniques	27
Figure 3.2. An Example of Debugging Macros in C++	28
Figure 4.1. Automatic Debugging Tools and Techniques.....	41
Figure 5.1. Alamo's Architecture.....	57
Figure 5.2. Sample Alamo Monitor.....	61
Figure 6.1. Trapped Variable Implementation	65
Figure 6.2. Sample expression where assignment can be violated.....	66
Figure 6.3. The New Data Structure Introduced for the Trapped Variable.....	66
Figure 6.4. The Allocation Macro Introduced for Trapped Variables.....	66
Figure 6.5. Unicon's Line/Syntax/Column Table	67
Figure 6.6. Sample Unicon Program	68
Figure 6.7. Sample <i>ucode</i> Format Before and After the Syntax Instrumentation	69
Figure 6.8. Sample Syntax Monitor	70
Figure 6.9. Sample Stack Trace.....	72
Figure 6.10. Sample Procedure that Backtraces the Current Stack.....	73
Figure 6.11. Sample Monitor Program Using the <i>E_Signal</i> Event	74
Figure 7.1. Sample AlamoDE Debugging Loop	77
Figure 7.2. AlamoDE's Architecture.....	78
Figure 7.3. Sample Monitor Using the <i>event mask</i> and <i>value mask</i>	79
Figure 7.4. Assigning Variables in the Buggy Program.....	80

Figure 7.5. Modifying Procedures in the Buggy Program	80
Figure 7.6. AlamoDE Debugging Capabilities.....	82
Figure 7.7. An AlamoDE Debugging Agent	83
Figure 8.1. IDEA's Architecture.....	87
Figure 8.2. IDEA's UML Diagram	88
Figure 8.3. An IDEA-based Agent Prototype.....	90
Figure 8.4. IDEA's on-the-fly Extensions (<i>External Agents</i>).....	91
Figure 8.5. IDEA's Internal Extensions (<i>Internal Agents</i>).....	92
Figure 8.6. Sample Migrated Agent	94
Figure 8.7. Explicit Agent Registration.....	95
Figure 9.1. Sample UDB Debugging Session	98
Figure 9.2. UDB's Debugging Architecture.....	99
Figure 9.3. UDB's UML Diagram	100
Figure 9.4. UDB's Main Debugging Loop.....	105
Figure 9.5. UDB's Implementation for Breakpoints	106
Figure 9.6. UDB's Implementation for Watchpoints Check	108
Figure 9.7. Initiating a Next Command.....	112
Figure 9.8. Implementing Next within the Evaluator	112
Figure 10.1. UDB's on-the-fly Visual Extensibility	117
Figure 11.1. A DT Assertion over Two Live Procedures.....	124
Figure 11.2. A DT Assertion over Two Sibling Functions	125
Figure 11.3. Sample Factorial Program Written in Unicon.....	127
Figure 11.4. Sample UDB Session that Uses DT Assertions	127
Figure 11.5. Using Temporal Assertions to Check Loop Invariant.....	129
Figure 11.6. Using Temporal Assertions to Validate Infinite Loops	130
Figure 11.7. Using Temporal Assertions to Check Variables from Various Scopes.....	131

Figure 11.8. Temporal Assertions: Scope & Interval	133
Figure 11.9. Temporal Assertions Evaluation	134
Figure 11.10. Sample Temporal Assertion's Evaluation.....	135
Figure 11.11. Sample Evaluation of Various Temporal Assertions	135
Figure 11.12. Sample of Different DT Assertions.....	138
Figure 11.13. UDB's Temporal Assertions Syntax.....	140
Figure 11.14. UDB's Temporal Assertions UML Diagram	144
Figure 12.1. Execution Time- Standalone vs. Monitored Mode	149
Figure 12.2. E_Deref, E_Line, E_Syntax, & E_Pcall Events Ratio to all Other Events	152
Figure 12.3. The Percentage Increase in the Size of the Object Code File	153
Figure 12.4. The Percentage Increase in the Size of the Executable Program	153
Figure 12.5. The Percentage Increase in Compile/Link Times	154
Figure 12.6. Time of Procedure Calls vs. Context Switches	155
Figure 12.7. Sample Unicon Program Measures Procedure Calls vs. Co-Expression	156
Figure 12.8. IDEA's Use of Debugging Agents.....	159
Figure 12.9. Sample Agent Counter	161
Figure 12.10. The Average Time for the Experimental Agent in Seconds	163
Figure 12.11. IDEA's Extension Techniques	164
Figure 12.12. IDEA's Extension Agents vs. Standalone Mode	164
Figure 12.13. The Performance of UDB's Various Debugging Features	166
Figure 12.14. State Based vs. Interval Based Evaluation.....	168
Figure 12.15. Sample Unicon Program Used to Measure Time of Temporal Assertions	168
Figure 12.16. Temporal Assertions Evaluation Time.....	169
Figure 13.1. Dissertation Contributions	173
Figure B.1. rsg Average Execution Time	185
Figure B.2. rsg Reported Events.....	185

Figure B.3. genqueen Average Execution Time	186
Figure B.4. genqueen Reported Events.....	186
Figure B.5. scramble Average Execution Time.....	187
Figure B.6. scramble Reported Events	187
Figure B.7. ichtartp Average Execution Time.....	188
Figure B.8. ichtartp Reported Events	188
Figure B.9. igrep Average Execution Time	189
Figure B.10. igrep Reported Events.....	189
Figure B.11. miu Average Execution Time	190
Figure B.12. miu Reported Events	190
Figure B.13. pargen Average Execution Time	191
Figure B.14. pargen Reported Events	191
Figure B.15. Average Time of all Events.....	193
Figure B.16. Average of E_Deref, E_Pcall, E_Line, and E_Syntax Events.....	193

List of Tables

Table 3.1. Manual Debugging Tools and Techniques	40
Table 4.1. Automatic Debugging Tools and Techniques	54
Table 6.1. Syntax Events and Codes	68
Table 6.2. Unicon's Debugging Related Keywords.....	71
Table 9.1. UDB's Default Monitor Events	103
Table 9.2. UDB's Tracepoints	111
Table 10.1. Atomic Data Related Agents	120
Table 10.2. Execution Behavior Related Agents.....	121
Table 11.1. UDB's DT Assertions Evaluation Action Operators.....	136
Table 11.2. UDB's DT Assertions Evaluation Log.....	138
Table 11.3. UDB's DT Assertions Evaluation Log.....	139
Table 11.4. DTA Temporal Logic Operators	139
Table 12.1. AlamoDE No Mask vs. Event Mask vs. Value Mask	148
Table 12.2. Syntax Instrumentation Effects on Object-Code (ucode) Formats.....	153
Table 12.3. Syntax Instrumentation Effects on Executable (bytecode) Formats	153
Table 12.4. Syntax Instrumentation Effects on Compiling/Linking Time	154
Table 12.5. Performance of IDEA's Extension Agents.....	162
Table 12.6. The Time of Different UDB Debugging Features.....	165
Table 12.7. Evaluation Time of Temporal Assertions.....	169
Table B.1. rsg Execution Time	185
Table B.2. genqueen Execution Time	186
Table B.3. scramble Execution Time	187
Table B.4. ichartp Execution Time.....	188
Table B.5. igrep Execution Time.....	189

Table B.6. miu Execution Time	190
Table B.7. pargen Execution Time	191
Table B.8. The Average Monitoring Time of All Events.....	192
Table B.9. Number of Reported Events	192

List of Equations

Equation 12.1. Number of Context Switches (E Events Reported to n Agents).....	157
Equation 12.2. Cost of Forwarding an Event to n Agents using Context Switches	157
Equation 12.3. Total Cost of Forwarding E Events to n Agents using Context Switches	157
Equation 12.4. Cost of Reporting an Event to an Agent in Standalone Mode	158
Equation 12.5. Total Cost of Reporting an Event to n Agents in Standalone Mode.....	158
Equation 12.6. The Cost of Reporting an Event to n Agents Using Procedure Calls	158

Part I

Introduction and Background

Chapter 1

Introduction and Objectives

The growth in the software industry is rapid and the size of programs is becoming larger and larger. In contrast, the rate of advances in the debugging literature is relatively slow. Most debuggers are well suited for a specific class or set of bugs. Program bugs can be caused by numerous circumstances and revealed long after their root cause. Understanding the source code and the execution behavior of the program is essential to locate and find the cause of most bugs. This understanding can be achieved by different means; one is to employ different debugging tools that capture, depict, analyze, and investigate the state of the program at, and in between, different points of execution. In practice, a programmer often tries more than one debugging tool on the same bug before it is caught.

1.1. Dissertation Scope

Various debugging tools and techniques are available. They range from reviewing the source code and utilizing print statements to using special purpose bug detectors. The research in this dissertation is focused on two kinds of debuggers and their extensibility. First is the typical interactive source-level debugger, which is one of the most valuable debugging tools, but it relies heavily on the user's ability to conduct a live test. It helps programmers locate and find bugs by stepping through the source code and examining the current state of execution. It provides techniques such as breakpoints, watchpoints, single stepping and continuing, and navigating the call 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. For instance, a class variable may be assigned a bad value in a method that is not on the stack when a bug that causes a crash or a core dump is revealed. A user can investigate the current state. If there is no evidence of the bug's root cause, he/she may restart the execution hoping to stop at an earlier point where the cause of the bug is still accessible [1]. These source-level debuggers suffer from various limitations such as:

1. Limited information provided about the execution history
2. Lack of automated and dynamic analysis-based debugging techniques
3. Limited features that are restricted to the commands prescribed in the debugger's manual
4. Closed architecture that provides little or no cooperation with external debugging tools.

Second are the post-mortem reversible debuggers (also known as trace-based debuggers), which provide debugging techniques based on the ability to browse forward and backward through the states of a completed execution. This approach provides outstanding debugging capabilities such as finding where and why some action has happened. For instance, if the buggy program produces an incorrect result, it is possible for the programmer to step backwards on the faulty output and find the improper value at each point until the root cause of the bug is revealed. On the contrary, if the program fails to produce the output, then the programmer has no handle on the bug to trace backwards. Most of these trace-based debuggers incur serious limitations such as:

1. A post-mortem debugging process that requires the ability to trace the completed execution state before they allow a user to investigate
2. Formidable scalability problems that are induced by the huge volume of trace data
3. Good at finding some types of bugs and not others
4. Neglect common debugging techniques such as altering the state of the program being debugged. A debugging process may include modifying the state of the buggy program in order to test “*what if*” kind of hypotheses.

1.2. Context and Motivation

The research presented in this dissertation targets the Unicon programming language [3, 4]. Unicon is an object-oriented dialect of Icon [6, 7], a very high level imperative programming language with dynamic and polymorphic structure types, along with generators and goal-directed evaluation.

This dissertation is motivated by two objectives. First, historically, the Icon language community had no formal debugging tool, only a built-in trace facility. A rationale for this was that the very high level nature of Icon reduces the need for conventional debugging because Icon programs are shorter than they are in conventional languages. However, Unicon programs are often much larger than was common for Icon. Even in a very high level programming language, programmers write bugs and the debugging process is still difficult and time consuming. Also, very high level languages’ advanced features may introduce special kinds of bugs and create special needs for debugging tools and techniques.

Furthermore, the cost of writing debugging tools is typically very high, which plays a significant role in the slow rate of improvement seen in the debugging literature. This motivates the second objective, which is to build underlying debugging facilities that simplify and reduce the cost of

writing debugging tools. Such infrastructure should permit easy experimentation with new debugging techniques that might become standard features in future debuggers. These two objectives take advantage of the Alamo framework, an event-driven monitoring framework developed originally to support visualization of Icon and Unicon program behavior. For this dissertation, Alamo has been extended with new features to enable debugging support. The result of these extensions is called AlamoDE.

1.3. The Problem

Different programmers and bugs require different debugging techniques. Often, bugs are revealed long after their root cause. Whenever a bug is discovered, a programmer tries to find a debugging tool or technique that suits its revealed behavior. However, often a debugger becomes useless when a situation arises that is not supported by its commands. At the same time, it is not feasible to provide all debugging techniques in one tool.

Sometimes, it is useful to utilize various debugging tools on the same bug and compare their outcome in order to better understand a bug's root cause. In these situations, working synchronously with more than one debugging tool on the same bug can speed up the debugging process. Instead of operating different debugging tools side by side, the programmer may benefit from running them all within the same session, allowing simpler interactions and collaborations between the tools. This capability has potential value, but requires underlying support for various debugging tools and techniques to work in concert with each other, which may entail extending a debugging tool with new techniques or integrating the features of one debugging tool into another. This extensibility may result in better debugging tools and reduce the user's search for various techniques. However, this extensibility is rarely supported. When it is applicable, it is difficult and time consuming. It requires sufficient knowledge and high level programming skills.

A solution to this problem can be reached by two levels of support. First, there is a need to simplify the process of developing new debugging tools and techniques, which may aim at improving the value of an existing tool by developing and integrating new techniques, or derive the innovations for new debugging mechanisms. Second, there is a need to simplify the extensibility of various debugging tools and techniques.

1.4. The Solution Approach Used in This Research

Simplifying the experimentation is essential for advances in future debuggers. The research conducted in this dissertation is focused on advancing the debugging state of the art by facilitating the extensibility of a typical interactive source-level debugger with custom-defined trace-based debugging and dynamic analysis techniques, which aim at improving its conventional debugging process.

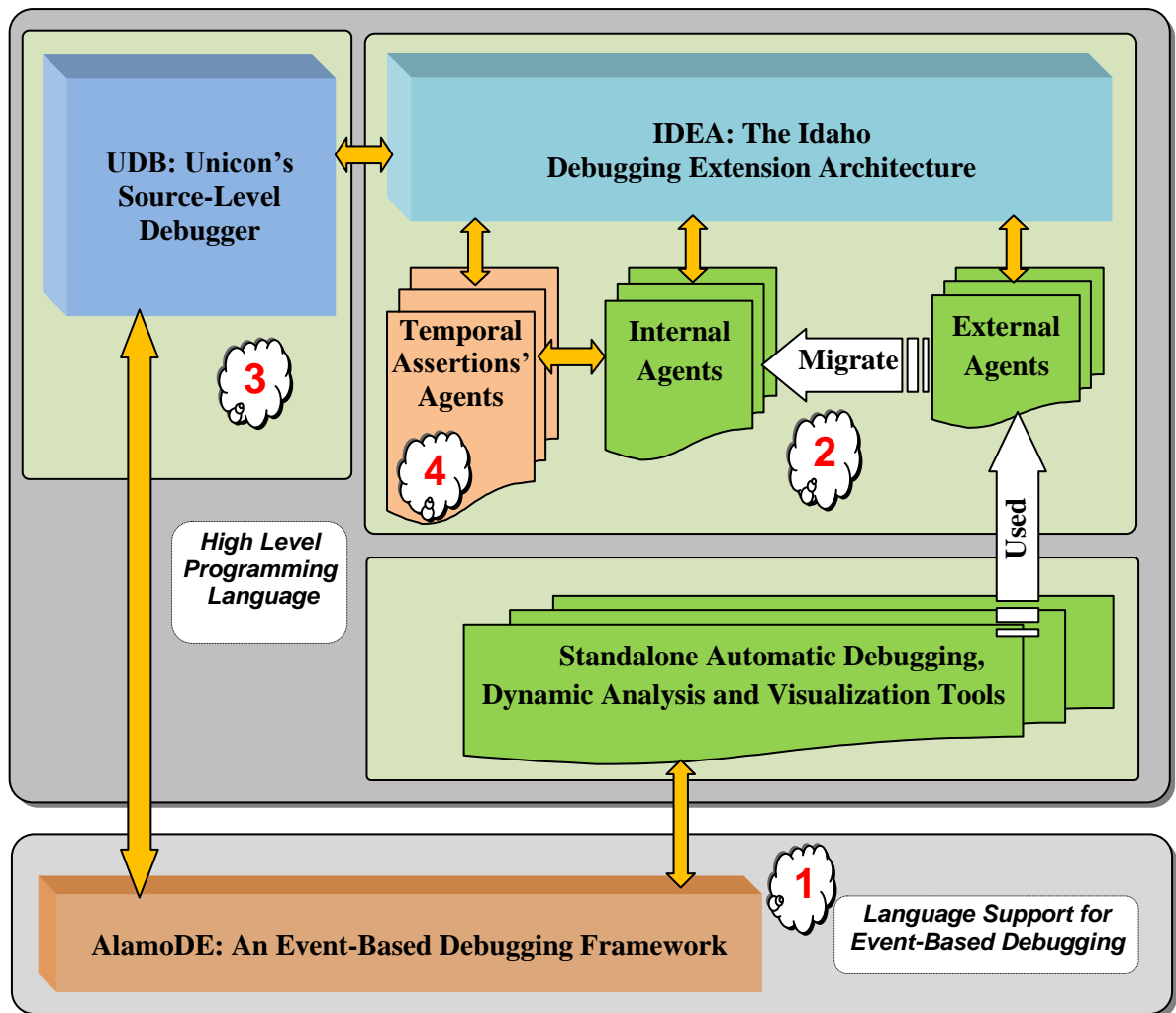


Figure 1.1. Dissertation's Contributions

The approach provides a debugging suite that consists of four primary contributions; see Figure 1.1. First, it provides an underlying high level virtual machine support for various debugging tools. Second, it presents a debugging extension architecture that simplifies the process of extending a source-level debugger with new debugging features, called agents. Third, it introduces a production

grade source-level debugger, named UDB, which utilizes both of the high level debugging support and the extension architecture. Finally, it provides a set of experimental extensions. These extensions introduce temporal assertions for the first time in a typical interactive debugger that debugs sequential programs. These four contributions are introduced in the following four sub-sections and discussed in detail through this dissertation.

1.4.1. Debugging Framework

The first contribution of this dissertation is an event-based debugging framework named AlamoDE (Alamo—Debug Enabled), see Chapters 5-7. This framework provides a high level abstraction mechanism that reduces the cost of writing a variety of debugging tools—including source-level debuggers and custom-defined debugging tools. This simplifies and speeds up the process of experimenting with new debugging techniques such as automatic debugging, dynamic analysis, and visualization tools. It encapsulates goals that include:

1. Reduce the development cost of debugging tools
2. Facilitate all the usual capabilities of classical debuggers
3. Support the creation of advanced debugging features such as automatic debugging, and dynamic analysis techniques
4. Debug novel language features such as generators, goal-directed evaluation, and string scanning
5. Support for runtime information sharing between various debugging tools through execution events

AlamoDE is general enough to support different kinds of debugging tools that range from classical source-level debuggers to automatic and dynamic analysis tools. It is a debugging framework that adds to the original Alamo framework:

1. Debugging-oriented virtual machine instrumentation
2. Additional execution state inspection and source code navigation
3. The ability for debugging tools to safely change the execution state of the buggy program by assigning to its variables and procedures.

1.4.2. Extension Architecture

The second contribution of this dissertation is an agent-oriented debugging extension architecture named IDEA (Idaho Debugging Extension Architecture), see Chapter 8. IDEA sits on top of AlamoDE to further elevate the debugging support through its novel extension mechanism. It simplifies and speeds up the process of experimenting with new AlamoDE-based debugging techniques and visualization tools within a typical interactive debugging session. Experiments are plugged in to the debugger allowing various debugging tools and techniques to work in concert with each other on the same buggy program. Different tools work and hunt for the root cause of a bug simultaneously and under the extended debugging session.

IDEA's extensibility allows different debugging tools to be written and tested as standalone tools and then loaded into a debugger without modification. These debugging tools play the role of agents in a source-level debugger. IDEA supports two types of extensions that distinguish it from other architectures:

1. Dynamic extension on the fly during the debugging session (external agents). This facilitates on the fly debugging extensions and cooperation between various debugging tools
2. Formal steps for migrating and adopting standalone agents as permanent debugging features (internal agents).

IDEA's extensions are in-process debugging agents that are used simultaneously through a mixture of *co-expression* context switches, *inter-program* procedure calls, and *in-program* procedure calls. Those agents are event-driven task-oriented program execution monitors. Each agent monitors a program's execution for custom runtime 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. An agent may employ events, event-sequences, and event-patterns to detect specific execution behaviors. Different agents perform different debugging missions such as detecting a suspicious execution behavior, performing an automatic debugging procedure, or executing a dynamic analysis technique. Each agent receives different runtime events based on its request. IDEA's central debugging core coordinates all agents. Each agent:

1. Provides the debugging core with its set of desired events
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. Presents its analysis results back to the debugging core, another agent, or directly to the end-user. The external debugging agents' standard inputs and outputs are redirected and coordinated by IDEA's debugging core.

1.4.3. Very High Level Debugger

The third contribution of this dissertation is a production source-level debugger for the Unicon [3, 4] programming language named UDB, see Chapter 9. UDB is built on top of AlamoDE framework and utilizes the IDEA architecture. It validates this framework and proves the usefulness of the extension architecture. It combines the capabilities of classical and trace-based debuggers and provides a friendly experimentation environment for various debugging tools and techniques to work in concert with each other. UDB is not limited to classical debugging techniques such as those found in GDB [5]. Extensions can use advanced debugging techniques, such as agents that implement automatic debugging or dynamic analysis techniques that may utilize information prior to the current execution state. Any number of external and internal debugging agents can synchronously assist in locating bugs. UDB suspends all agents whenever a breakpoint or watchpoint is reached, and it resumes them whenever the buggy program is resumed.

Unlike common dynamic analysis tools that have to be linked in advance into the source code of the buggy program, or initialized at the start of the host debugger, UDB's agents can be loaded and managed on the fly during a typical source-level debugging session. The user does not need to restart the debugging session whenever a decision is made to incorporate any of these agents, unless the agent requires information about previously executed properties. Agents that are loaded in the middle of a debugging session are not able to analyze execution properties prior to their loading point.

UDB's extension agents are written and tested as standalone tools, and then incorporated into the debugger via dynamic loading or linked into the debugger executable with almost no source code alterations. Since UDB's extension agents are programmable in the same target language, UDB enables experienced users to write and test their own debugging tools as standalone programs and then use them as externals, or incorporate them as internals—built-in debugging features. Furthermore, UDB's interactive user interface resembles GDB's interface. This provides familiarity and ease of use for programmers who switch between languages frequently. UDB adds a handful of simple but general commands to load, unload, enable, and disable its extension agents. This simplifies the extensibility especially for typical users and novice programmers who may want to benefit from existing agents.

1.4.4. Extension Agents

The final contribution of this dissertation is a set of extension agents that validates the extensibility and the usefulness of the IDEA architecture. These extensions are used within UDB's interactive debugging session, see Chapter 10. The set of extension agents is divided into 1) language dependent agents, 2) language independent agents, and 3) temporal logic operators. The agents of temporal logic operators are used to provide Dynamic Temporal Assertions (DTA), see Chapter 11. These DTA's are logical expressions used to validate relationships (a sequence of execution states) that may extend over the entire execution and check information beyond the current state of evaluation. The temporal logic operators are internal agents used within the IDEA architecture. Those agents can reference other atomic agents. This collaboration between agents can provide a helpful debugging technique and prove the value of the IDEA architecture.

1.5. The Results

An AlamoDE-based debugging tool must use different approaches to implement features found in similar standard debugging tools, and faces potential performance challenges. In compensation, this type of implementation greatly simplifies the process of experimenting with new debugging techniques that probably would not be undertaken if the implementation was limited to the low level approaches found in other debuggers. This dissertation tests the following hypotheses:

1. The AlamoDE event-based debugging framework is sufficient to support various debugging tools and techniques, including typical source-level debugging functionalities, with sufficient performance for production use.
2. AlamoDE's in-process debugging support allows for efficient and complex communication patterns between the debugger and the buggy program. These communications are facilitated by a mixture of event monitoring and high level primitives.
3. An AlamoDE source-level debugger can surpass ordinary debuggers with more debugging capabilities.
4. AlamoDE enables low-cost development of debugging tools, and the IDEA architecture allows AlamoDE tools to be extended easily. AlamoDE-based debugging tools are written in a high level language with no low level or hardware specific code.
5. IDEA simplifies a typical source-level debugger's extensibility with on the fly agents that utilize automatic debugging and dynamic analysis techniques. These extensions require no

special compilation, no source code or object modification, and no pre-initialization or any knowledge of the host debugger internal implementation.

Also, this AlamoDE debugging framework and the IDEA extension architecture, both simplify the process of experimenting with new custom-defined debugging tools and techniques. This experimentation includes:

1. Improvement to traditional techniques such as watchpoints and tracepoints
2. The ability to integrate verification and validation techniques such as dynamic temporal assertions
3. The simplicity to develop, test, and integrate new techniques of debugging agents.

1.6. Definitions

A number of terms and phrases used throughout this dissertation require some explanation. The terms *monitor* and *monitor program* denote a program that performs execution monitoring on another program. In this dissertation, these terms represent a debugger or a debugging tool. The terms *monitored*, *monitored program*, *subject program*, *target program*, and *buggy program* all represent the program being monitored and debugged. The term *agent* represents “a program that performs some information gathering or processing task in the background. Typically, an agent is given a very small and well-defined task” [118]. A *debugging agent* is a special agent that performs an event-driven task-oriented program execution monitor. The term *monitoring framework* represents the underlying support and the public API for event-based execution monitoring. The term *debugging framework* represents the underlying support and the public API for event-based debugging; some runtime events are used to control the execution of the buggy program while others are used to obtain information about its execution state. This framework allows programmers to write a variety of debugging tools in a very high level language. Finally, the term *debugging architecture* represents the structural design of a debugging tool, such as a source-level debugger.

1.7. Dissertation Outline

This dissertation consists of five major parts. Part I gives research background and investigates various debugging tools and techniques. Chapter 2 gives a general introduction to the field of debugging. Chapters 3 and 4 present a debugging literature survey, where different manual and automatic debugging tools and techniques are discussed in terms of their features, pros, and cons.

Part II presents a detailed discussion of the dissertation's first major contribution named AlamoDE; an event-based debugging framework. Chapter 5 presents the Alamo monitoring framework that was originally designed to support software visualization [106,107,108]. Alamo is the related work that constitutes the starting point for this dissertation. This chapter only presents Alamo's features that are adequate for debugging needs. Chapter 6 presents extensions to the Alamo framework developed for this dissertation in order to support debugging. It describes the implementation of these extensions within Unicon's virtual machine and its runtime system in order to facilitate high level debugging support. Chapter 7 presents AlamoDE, a debugging framework that features event-driven debugging support integrated within a high level programming language. This chapter covers existing features from the Unicon language and the Alamo framework along with new extensions. The combination provides very high level debugging support, which enables programmers to develop new debugging tools and techniques in less effort and time.

Part III presents a very high level extension mechanism. Chapter 8 introduces the IDEA architecture, which enables API compliant debugging tools to be loaded under the control of a source-level debugger. Chapter 9 discusses the design and implementation of UDB and its built-in debugging techniques. UDB is an event-driven source-level debugger with exceptional extensibility.

Part IV presents extension agents. Chapter 10 discusses various kinds of agents used in UDB under its IDEA architecture support. Chapter 11 introduces UDB's Dynamic Temporal Assertions (DTA). DTAs are built on top of a set of predefined internal debugging agents that are integrated into UDB using the IDEA architecture. Within UDB, this set of special agents is used implicitly through a high level assertion language that allows users to dynamically assert execution properties across different execution states. DTAs are inserted into the debugging session from within the UDB console based interface.

Part V presents the results found by this dissertation. Chapter 12 provides a performance evaluation for AlamoDE and IDEA architecture, all within the UDB source-level debugger. Chapter 13 presents the conclusion and future work.

This dissertation includes a set of appendices. Appendix A shows a summary of DT assertions introduced in Chapter 11. Appendix B shows detailed information about the evaluation discussed in Chapter 12. Finally, Appendix C shows a command summary for the currently implemented features in UDB.

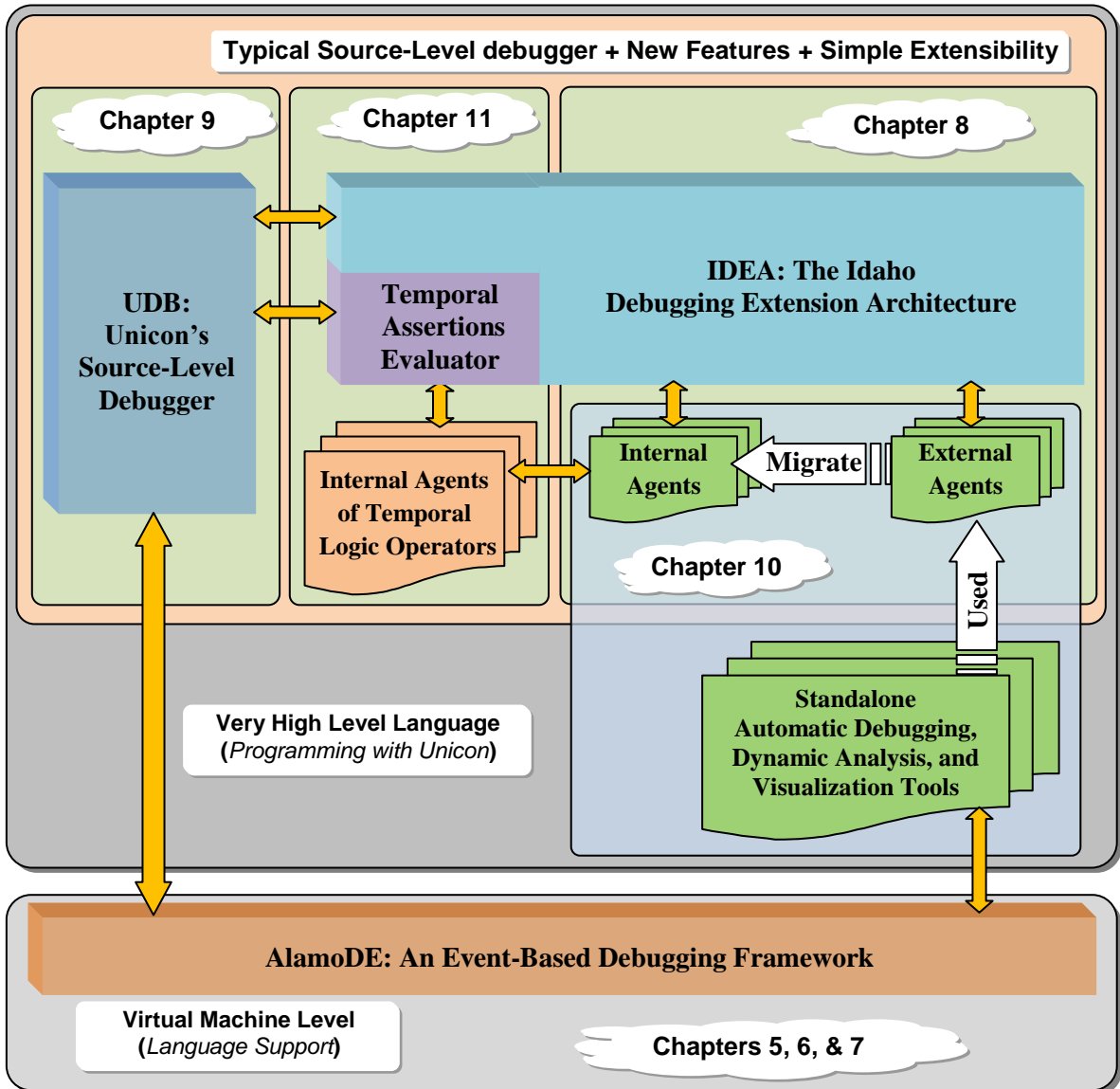


Figure 1.2. Dissertation Outline

Chapter 2

Background

Debugging is essential for every software project; it is part of the *edit-compile-link-debug* development cycle [8]. Different debugging tools are available, each of which may implement different techniques that catch a particular kind or class of bugs. Some consider debugging an art [9], while others consider it a systematic science [10]. In either case, debugging is an active search for the real cause of a known bug [11] and the fixing of program faults. The debugging time depends on the bug, the way it is observed, the employed debugging tool, and the programmer's skill. Some studies have found a significant positive correlation between the debugging rate and the programmer skill [11].

Most debugging literature, old and new, consistently asserts that debugging is a hard problem, which consumes a big percentage of the development time; this percentage often reaches over 50 percent [12]. Additionally, debugging is emotionally difficult because it is a challenging problem, and because people dislike admitting that their program is buggy and requires debugging [9]. Kernighan and Plauger argued that "Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?" [13]. In contrast, some experiments found that programmers can debug their own programs faster and easier than they would debug programs created by others. This may relate to the difference in the debugging process; a programmer usually uses backward reasoning to debug his/her own program, while forward reasoning is often used to debug programs written by others [11]. On the contrary, others think that bugs are not hard to find. In this minority view, if the code is complicated enough that it obscures the bug, then the real bug is the design; this should be enough of an excuse to redesign and rewrite [14].

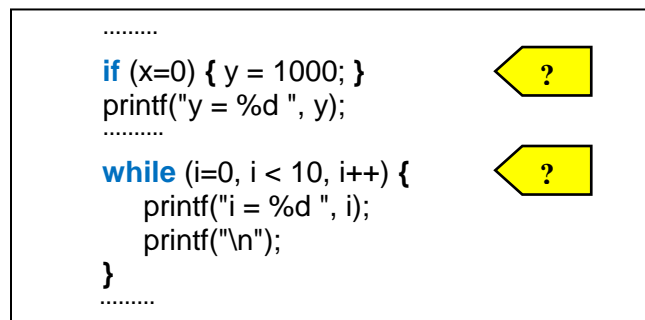
2.1. Program Bugs

A program's source code is composed of 1) syntactic pieces defined by the programming language in use, 2) modules and libraries defined by the design and implementation, and 3) semantics and behaviors that are defined by the requirements and specifications. In general, a bug is a mistake somewhere in the program's development process; it may occur in any one or more of these categories. However, this dissertation considers only those circumstances when the requirements,

specifications, and design are bug free, and the bug is always in the implementation of the program's source code.

First, syntax related bugs are varied based on the language grammar. Each language has a grammar that defines what is valid and what is not. Most bugs in this category are usually a simple omission or duplicate symbol. Since syntax bugs are defined contingently based on each language, any violation of a language syntax rule may result in a syntax bug. The language's compiler or interpreter usually catches those bugs very easily.

Second, linking bugs are related to the program's structure of modules and libraries. In order to debug a linking bug, a programmer needs insight about the relationship between modules and libraries, and how they are incorporated in the main program. Sometimes, the linker discovers these bugs during the linking phase of the build process. However, some programs may utilize dynamic linking, which may not link and validate the linked modules until they are used in the program at runtime. Another example can be seen in some dynamic languages that do not validate the linked modules until they are used. Either one can result in linking bugs during the execution.



```

.....
if (x=0) { y = 1000; }
printf("y = %d ", y);
.....
while (i=0, i < 10, i++) {
    printf("i = %d ", i);
    printf("\n");
}
.....

```

Figure 2.1. Sample Semantic Bug

Finally, in some cases the semantics of a statement is buggy or ambiguous. For example, the if statement in Figure 2.1 is syntactically valid in the C language according to some compilers, but the condition always fails. Furthermore, the `while` loop in Figure 2.1 is syntactically valid, but the condition always fails too—perhaps the programmer intention was to write a `for` loop instead. Static analysis techniques may be used to catch similar suspicious expressions during compilation, which may warn the user about potential misuses or flag semantic errors. In general, semantic bugs are the most difficult to define, find, and locate. They may depend on the program requirements, specifications, design, or implementation. Usually, the program is heavily tested during the development process to reduce the probability of these bugs during actual use in released builds. Besides semantic bugs, this category has many names such as logical bugs, runtime bugs, or even software bugs.

2.2. Runtime Bugs

Runtime bugs are discovered during program's execution. They can be introduced into the software during the: 1) initial implementation, 2) modification of an existing program feature, or 3) repair process of an existing bug [11]. Runtime bugs can be defined by capturing the difference between 1) the computed, observed, or measured values, and 2) the specified, correct, and theoretically true value. Another way to define a runtime bug is by capturing 1) the inconsistency between the base model and the target model, or 2) the inconsistency between the expected behavior and the actual behavior [11].

Runtime bugs can be revealed as an incorrect or missing output, unexpected behavior, or as a program crash. Software bugs can be classified in terms of reproducibility and severity. The ability to reproduce the bug is the first step to debug it. Some bugs are reproducible deterministically while others are not. A deterministic bug is one that is revealed every time the program executes in a specific path; most of the time, those bugs are data dependent (input dependent) such as dereferencing a pointer. In contrast, a non-deterministic bug does not depend on the execution path; those bugs are harder to locate and find and are mostly related to memory corruption [15, 16]. In general, reproducing and finding the root cause of a bug is harder than fixing it.

On the other hand, a bug's level of severity may influence the urgency of fixing it, especially when it is combined with a high recurrence rate. A fatal bug is one that causes the program to crash or freeze. A non-fatal bug is one that may cause a missing or incorrect output or an unexpected behavior. A bug may be so infrequent that the user can afford to live with it, especially if it is non-fatal. In general, the name *functional bug* is given to any bug that may cause an incorrect or missing output, a non terminated execution, or an unexpected termination that might be caused by a crash or a core dump [17]. Often, specific execution behaviors can be checked in relation to the revealed functional bug. For example, many programmers neglect checking function return values, and pointers are often misused causing memory corruptions.

2.3. Debugging Terms

This appendix provides an overview of the most common terms related to debugging. Some of these terms may indicate a category or class of debugging techniques while others may represent a specific debugging tool. The following is in alphabetical order with a brief description for each term:

Abstract Debugging utilizes abstract interpretation to debug programs prior to their execution. It is based on two types of assertions: 1) *invariant assertions*, and 2) *intermittent assertions*. The user

inserts assertions into the buggy program source code and the debugger treats any violation of those assertions as runtime errors [37, 38]. See Section 4.2.

Algorithmic Debugging is a high level means of checking and fixing program correctness. This process may include two algorithms: 1) a *diagnosis algorithm* that identifies the bug in the program based on its incorrect behavior and 2) a *bug correction algorithm* that solves and fixes the already identified bug. It may utilize different static and dynamic analysis tools [11][114].

Automatic Debugging is any debugging tool or technique that employs a computer algorithm to either locate the cause of a bug, reduce the search space for the location of the bug, or reduce the set of inputs that induce the bug. This category contains a variety of debugging techniques, each of which is focused on specific kind or class of bugs [39]. See Chapter 4.

Bidirectional Debugging is a special case of reversible debugging. It reverses or undoes the execution of parts of the buggy program. It may be achieved based on checkpoints taken automatically in correspondence to the debugger commands. The debugger supports two types of commands: forward and backward [1], see Section 3.3.2.

Convergence Debugging is an automatic debugging technique that utilizes a set of test cases. It isolates different test cases based on their convergence on the root cause of the failure by analyzing the internal control and data flow of the failure test case [29], see Section 4.3.3.1.

Declarative Debugging allows users to perform queries on the execution history and specific execution states [40]. It is a specific kind of programmable debugging.

Delta Debugging is an automatic way of narrowing down the differences between a failure run and a successful run [22]. It is a fully automatic debugging technique that finds the simplest test case that generates the failure, and highlights the difference between a passing and failing test case. It consists of two algorithms: a *simplification algorithm* and an *isolation algorithm* [21, 41], see Section 4.3.3.5.

Event-Based Debugging is used to control and obtain information from the buggy program by means of execution events, which are activities during the execution of the target program such as method being called or a variable being assigned. It has been used mostly to debug concurrent and parallel programs. It is also used in debugging sequential programs in debuggers such as Dalek [42, 43] and JPDA [23].

Goal Directed Debugging is a semi-automatic debugger for Microsoft Excel spreadsheets. It allows end-users to report expected cell values and the debugger provides suggestions. It is up to the user to apply, refine, or reject any of these suggestions [44].

Interactive Debugging is a debugging method that allows a user to perform live investigation of the execution states during the debugging session; whether it is before or after the occurrence of the bug [45].

Manual Debugging is any debugging tool or technique that depends heavily on the user's ability to investigate and search for the bug and its location, see Chapter 3.

Model-Based Software Debugging is the process of locating the place of defects in a program based on models generated automatically from the source code or the execution of the program. The generated (observed) model is compared against the intuitive or theoretical model. It is an application of Model Based Diagnosis [46], see Section 4.3.1.

Omniscient Debugging is a post-mortem source-level class of debuggers that provide the ability to go backward in time through the ability to navigate the execution history [47, 48, 49, 50]. See Sections 3.3.4.1 and 3.3.4.2.

On Demand Debugging is the process of starting a debugging session right after encountering a runtime error. The debugger can be attached to the faulty program automatically at the failure point, saving developers' time. A developer does not need to rerun the application and reproduce the situation where the bug should occur again. In some situations, it is possible to fix the bug and resume the program's execution [51].

Performance Debugging is a class of debugging tools that targets the complexity and the efficiency of the program. They are mostly used under the name of profilers. An example of this class of debuggers is gprof [52].

Post-Mortem Debugging is the process of debugging that allows the user to investigate the execution history of states after the occurrence of the bug or after the termination of the program's execution [37, 38].

Record-Replay Debugging provides the ability to reproduce a bug encountered at the end-user site. Recorded information before the occurrence of the crash is sent to the developers, so they may deterministically replay and reproduce the bug in their environment by replaying the last several million instructions before the crash [33, 53], see Section 4.3.3.8.

Relative Debugging is a class of debuggers that target the process of debugging two different versions of the same program. It allows a user to compare the execution of two programs based on expected predefined associations. It may concurrently execute the two programs in order to verify the similarities and find any differences [54, 55], see Section 4.3.3.7.

Reversible Debugging is a general debugging technique that provides the ability to reverse and undo the execution of the buggy program into some previous point. In practice, this kind of debugging encompasses many technical limitations, which may depend on the operating system and the target machine or architecture. Often, it depends on special hardware and operating system support [1, 56], see Section 3.3.2.

Simulation Based Debugging performs live simulation of the execution of the buggy program [57]. It may simulate the hardware of the execution environment by means of virtualization to avoid any modification of the host operating system [115], or provide a synthetic CPU targeting specific predefined anomalies, which is the case in Valgrind [18, 58], see Section 4.3.3.11.

Source-Level Debugging is a class of debuggers that provide the ability to debug a program's execution on the level of its source code. Even though the execution is performed on the program's machine level representation, the debugger should be able to provide the user with information in relation to the source code. It is also known as *symbolic debugging* because it provides users with symbolic information obtained from the source code such as variables and their values [10]. See Section 3.3.

Statistical Debugging is an automatic debugging technique for finding and locating bugs in released software systems. It depends on collecting sparse real samples from large numbers of runs. Sampled data is analyzed to locate and find the cause of different real world bugs [15, 16, 59, 60], see Section 4.3.3.4.

Trace-Based Debugging is the process of debugging the program using a tracing mechanism. The traced data can be collected by different means of instrumentation that can be as simple as print statements or as complex as dedicated instrumentation frameworks [61, 62], see Section 3.3.4.

Visual Debugging is a class of debugging tools that includes visualization and animation for the sake of debugging. Visualization can be used to provide better understanding of complex results provided by the debugger such as a huge amount of traced data [63].

2.4. Debugging Tools

A debugging tool is any tool that is used to assist the user during the debugging process. Those tools include:

1. *Static analysis tools* that check the buggy program source code for potential bugs
2. *Debugging libraries* that are linked into the buggy program to perform some dynamic analysis checking, such as memory leak detection
3. *Tracers* that specifically focus on a specific execution behavior such as a function call or a variable state
4. *Profilers* that target the performance of an execution
5. Interactive and post-mortem *source-level debuggers* that provide users with the ability to investigate the execution state of the buggy program
6. *Algorithmic and automatic debugging tools* that target the program's source code, its execution behavior, or its execution model.

Dynamic debugging tools may change the behavior of the buggy program. This may be intentional, for example when a programmer changes a value in a debugger to see what will happen, or it may be unintentional. In practice, a runtime bug (especially non-deterministic bugs) may behave differently before and after the debugging tool is involved. Generally, debugging tools intrude on the buggy program space as a result of sharing resources such as memory; these effects are minimized in order to preserve the reproducibility of bugs.

Some debugging tools are built as extensions on top of another debugger or debugging architecture. Rob Law [11] classified debugging tools in terms of generations in an analogy similar to the classification of programming languages:

1. *First generation*: low-level debugging tools used to monitor and obtain information about the CPU and its registers and memory. The main drawback of this generation is that the debugger provides little or no resemblance between the source program and the memory instructions. This reduces their usability.
2. *Second generation*: source-level debuggers that provide information in terms of program's source code and the programming language in use. The user can use breakpoints and watchpoints to control the execution of the buggy program.

3. *Third generation*: debuggers that provide information beyond the correlation of execution to the source code. A debugger of this generation can make analysis and logical assumptions about the location and the root cause of the bug. This generation includes tools that may implement static and/or dynamic analysis techniques such as program-slicing algorithms.
4. *Fourth generation*: knowledge-based debugging and intelligent tutoring systems that apply analysis techniques to identify and repair a bug.

This classification provides little or no information about the implementation, advantages, disadvantages, usability, and the kind of analysis in use. A new classification based on the architecture of the debugging process is introduced in Section 2.9. Furthermore, Chapters 3 and 4 present a classification for manual and automatic debugging tools and techniques.

2.4.1. Architecture

The form of communication between a debugging tool and its buggy program may impact the debugging process and limit the capabilities of the utilized debugging techniques. Some debugging tools facilitate their communications through an in-process scheme, while others depend on an inter-process architecture. Each communication method has its own advantages and disadvantages. In particular, debugging tools with in-process communication may intrude on the buggy program space and change the bug behavior. This intrusion may add to the difficulty of the debugging process. However, in-process communication simplifies the implementation of complex interactions between the debugging tool and the buggy program. For example, often in-process debugging architectures provide a debugging tool with direct access to the space of the buggy program that may exclude or reduce the operating system overhead and its implications.

In contrast, pipes, sockets, or even network protocols are used to facilitate inter-process communication between a debugging tool and its buggy program, or a debugging tool front-end and its backend. For example, DDD is a front-end debugger for GDB, DBX, and other console-based debuggers. DDD communicates with the underlying debugger through bidirectional pipes; see Section 3.3.1.7. As another example, the Java Debug Interface (JDI) can communicate with the Java Virtual Machine Tool Interface (JVM TI) using sockets, pipes, or shared memory. Usually, inter-process communication reduces the buggy program intrusion problem. At the same time, it may introduce another layer of overhead, which may add to the debugging time through its delay on the various interactions during the debugging process. In general, the goals and features of a debugging tool justify its architectural design. For example, inter-process architectures are very important to facilitate remote debugging techniques.

2.4.2. Implementation

Current debuggers implement one of three mechanisms for controlling and obtaining debugging information from a buggy program. First, trapped instructions are one of the oldest and most successful techniques found in classical debuggers such as GDB. This trapped instruction mechanism is an efficient breakpoint technique for interactive debugging sessions, but inefficient for conditional or automatic debugging techniques. This means it is efficient as long as the number of trapped instructions is infrequent enough that it does not delay the execution of the buggy program noticeably. This potential performance problem can be seen in GDB in many debugging scenarios [1], see Section 3.3.1.

Second, event-based debugging is one of the most efficient debugging mechanisms for redundant programs that run on different processors in parallel. Most of the time, events are lightweight and easily transferred between different processes as messages or signals. This mechanism is adopted by the Java Platform Debugging Architecture (JPDA) [23] for debugging multi-threaded and sequential programs. JPDA is based on execution events that transfer between different processes through pipes, sockets, or even network protocols; the debugging tool and the buggy program are in different processes. Most of the time, JPDA's events are hidden under a high level API of primitives and methods. In contrast, the AlamoDE debugging framework, presented in this dissertation, transfers lightweight events between in-process threads called co-expressions, see Chapters 5-7. Furthermore, AlamoDE-based debugging tools employ events directly without dealing with extra wrapper functions.

Finally, different algorithmic and automatic debugging tools utilize various algorithms and automated techniques to reduce the human factor and speed up the debugging process. Some automatic debugging tools use static analysis techniques that are applied on the program source code to find potential runtime faults such as static slicing. Others operate on a subset of test cases such as Delta debugging presented in Section 4.3.3.5, or a specific dynamic analysis technique such as Valgrind presented in Section 4.3.3.11.

2.4.3. Interface

Debugging tools vary in their user interface and in the amount and form of information that each provides about the buggy program and its bug. Some tools are interactive and allow live investigation in the execution state, while others are post-mortem and provide execution history navigation mechanisms. Recently, a new debugging interface paradigm emerged that employs natural language questioning during an interactive investigation process. This new debugging interface allows a user to

provide the debugger with questions about the execution of the buggy program, such as *why did?* and *why did not?*. This new interface is introduced in a debugger called Whyline [24, 25]. Whyline has two different implementations: the first targets the Alice framework [26] and the other is for Java programs [27, 28], see Section 4.3.3.9.

In general, some debugging tools provide console-based debugging with character-based commands such as GDB. In GDB, commands are used to control and investigate the buggy program's execution state. In contrast, visual debugging tools provide GUI interfaces where conventional commands are replaced with mouse pointing and clicking. For example, DDD provides a remarkable front-end GUI-based interface for GDB; it hides GDB's commands under GUI buttons. Furthermore, DDD provides dynamic visual data structure representation and a navigation mechanism; see Section 3.3.1.7. Other debuggers provide query-based debugging either from a GUI-based or console-based interface. For example, Coca provides a Prolog-based query interface from a console, see Section 4.3.3.10. In contrast, Omniscient debuggers such as ODB and TOD provide GUI-based queries. Moreover, some debuggers are integrated within IDEs, which may simplify the *edit-compile-link-run-debug* cycle. Another set of debuggers employ a programming approach, allowing the user to write and provide the debugger with some code that may or may not share the syntax and semantics of the target language. See Acid in Section 3.3.3.2 for an example.

Often, GUI-based interfaces simplify some of the tedious interactions needed by the console-based debuggers. For example, GUIs can provide multiple windows within the same screen. This permits the possibility of simultaneously presenting more debugging information such as the call stack, variable states, and source code. More information about the buggy program may simplify the debugging process, especially when it is combined with a convenient navigation mechanism. However, GUI has little benefit when it comes to controlling the process of the buggy program [2].

2.5. Debugging Process

Always questioning the nature of the bug leads to debugging hypotheses. Almost every debugging process starts with a set of hypotheses, which may include the conditions under which the bug is revealed, the bug location, the root cause, the expected behavior, the observed behavior, and how to modify the program in order to fix it [10]. Every hypothesis is validated or refuted by the debugging process, which may employ different debuggers and debugging techniques. The debugging process is an iterative process of verifying, modifying, and changing the set of hypotheses until the bug is fixed [10]. It is important to know that a debugging process is different from the process of finding bugs, which may include testing and verification.

Usually, bugs are revealed long after their failure's root causes [20]. The debugging process may isolate the bug's root cause that produced the failure through understanding the conditions under which the bug occurs. This may include 1) utilizing in-code language features such as print statements and assertions, 2) employing an analysis tool whether it is static, dynamic, or a combination, 3) stepping through the execution with a source-level debugger, or 4) using an automatic debugging tool that may be able to identify the cause or the location of the bug, or at least reduce its search space [29]. See Section 2.4.1.

The debugging process requires experience because bugs are defined based on a combination of different factors such as the program implementation, its running environment, and its design requirements and specifications. Those factors are specific to that particular program and its revealed bug, limiting any generalization that can be made. The debugging process can be seen as a hunting strategy that includes five major categories [11] defined by the level of the program understanding and the debugging context:

1. *Preliminary investigation* is the first step, which is used to ensure that the bug is somewhere in the source code and not in the environment such as the hardware or the operating system. This step may include reviewing user comments, collecting bug reports, and performing preliminary testing.
2. *Static debugging* that includes reviewing the requirements, the design, and the source code.
3. *Runtime observation* of the program's behaviors as a whole. This may include testing and analyzing inputs, processing steps, and outputs.
4. *In-code debugging* through print statements and assertions to verify the program's execution flow of control and validate some critical expression evaluations.
5. *Dynamic debugging* using a dedicated debugging tool such as a source-level debugger that allows the insertion of breakpoints, single stepping, and execution state investigation.

2.6. Debugging Process Architecture

This section presents a new look at a wide range of different debugging tools and techniques, some of which are research prototypes while others are real industrial and open source projects. The result is a taxonomy that mingles different debugging ideas, techniques, and tools, in one place. The classification is presented based on general properties such as pros, cons, techniques, and implementations. It emphasizes the idea that debugging tools share one goal regardless of their scope,

which is to help a user locate the root cause of a bug. This section gives a closer look at different debugging approaches based on their debugging process architecture.

In Figure 2.2, level one divides debugging techniques into four categories presented in the following subsections. Whereas level 2 divides local debugging techniques into manual debugging presented in Chapter 3 and automatic debugging presented in Chapter 4.

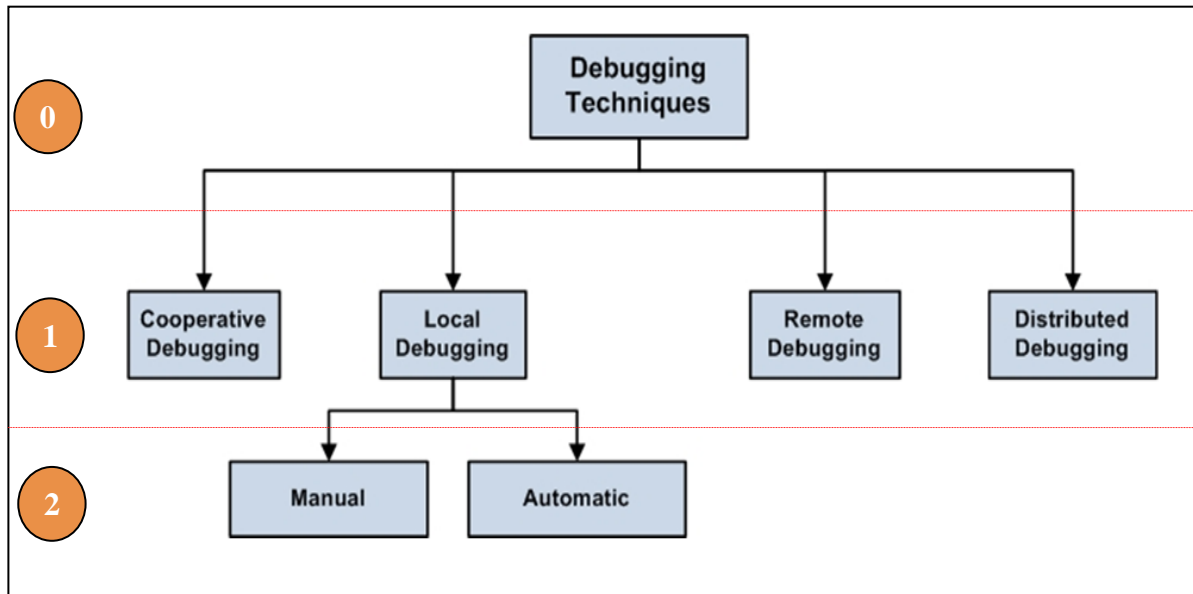


Figure 2.2. Debugging Techniques

2.6.1. Local Debugging

The debugging process is considered local if and only if both the debugging tool and its buggy program live on one machine and only one debugging interface is available. Having the debugging tool and the buggy program on the same machine is not limited to the in-process communication. Inter-process communications between the debugging tool and the buggy program is also considered local debugging as long as they are both on the same machine and same operating system. This category covers the vast majority of debugging tools. The next two chapters present various manual and automatic debugging tools and techniques.

2.6.2. Remote Debugging

Remote debugging is where the debugger and buggy program run on different machines or at least the debugger front-end is at a machine different from the one that is running the debugger

backend and the buggy program. This debugging technique is beneficial in some circumstances where the environment affects, and in some cases alters, the debugging situation. For example, the user can be sitting at a machine that runs the Linux operating system, but the bug occurs only when the program runs on Windows or vice versa. In this situation, the debugger can be running on the user's machine and the target program is running on a remote machine. Remote debugging is useful when the target machine or operating system is not directly accessible to the user who is debugging the program. Common Integrated Development Environments (IDEs) such as Microsoft Visual Studio (MSVS) and Eclipse provide support for remote debugging.

2.6.3. Collaborative Debugging

Large scale programs are hard to manage and debug by one person. Different developers may collaboratively share the process of debugging by dividing known bugs among themselves, where each one works on specific set of bugs, in different sections of the code, independently. This approach is inefficient and it can be misleading, especially when some bugs affect other bugs; this means developers may end up repeating work or overlapping with each other's work. Another collaborative approach is for developers to work cooperatively on the same bug at the same time (by gathering around one screen). This may be inconvenient for some developers for different reasons such as the available space, environment, distance, and differences in reasoning or technique.

A tool that allows different users to collaborate with each other regardless of their location would improve and speed up the debugging process. One of the first collaborative debugging tools over distance locations was web based; the buggy program is posted on a specific website where other developers can look it up and try to resolve bugs. However, real time collaboration would avoid any overlapping or redundant work. Codebugger [30] is one of the first tools to provide real time collaborative debugging. It is a Java debugging tool that allows a group of developers to participate, communicate, and share the debugging session in real time regardless of their physical location [30]. Moreover, some collaborative IDEs such as the IBM's Jazz [31] support a form of real time collaborative debugging.

In contrast, a passive form of collaborative debugging has emerged recently called Cooperative Bug Isolation (CBI) [32]. It targets real world bugs in released software. The information from successful runs as well as failed runs is sent into a central database. Then statistical inference is applied on this collected information to locate the root cause of reported bugs. This form of debugging process is collaborative in the sense of collecting data used to locate the root cause of a bug automatically. See Section 4.3.3.4.

2.6.4. Debugging Parallel and Distributed Systems

Distributed systems and their debugging techniques are beyond the scope of this dissertation. However, this dissertation utilizes the event-driven debugging approach, which is most-common in debugging distributed systems. This section highlights a few of the most common characteristics of distributed systems that makes event-based debugging one of the most used approaches.

Debugging parallel and distributed applications is more complicated than debugging single threaded or sequential programs. Distributed applications have an extra set of potential bugs, which relate to the complexity in communication over multiple simultaneous processes. First, there is lack of global time; each part of a distributed application has its own time-dependent behavior. Time management is a key characteristic of a distributed system that affects any precise query of their global state [9]. A second factor is non-deterministic execution; it is common for distributed programs to behave differently with the same input on different executions [33]. Finally, there are multiple threads of control: complex patterns of communication are imposed by the parallel activities. This adds to the challenge of the debugging process [33, 34]. A debugging tool for distributed systems is better when it is integrated within the system in use. This enables it to play a better role in coordinating the different parts of the debugging tool itself. For example, the distributed Event-Based Behavioral Abstraction (EBBA) tools are adoptable to the behaviors of the local and remote system components [35, 36]. The reader may consult the ACM/ONR Workshops on Parallel and Distributed Debugging, or the more recent workshop on Parallel and Distributed Testing, Analysis and Debugging, for additional information on this subject.

Chapter 3

Manual Debugging Tools and Techniques

A debugging tool is considered manual when its debugging process depends heavily on the user's ability to investigate and search for the root cause of a bug; tools that find bugs without manual investigation are discussed in the next chapter. For example, manual debugging tools may provide the ability to control the buggy program's execution and simplify the user's investigation. One of these tools is the source-level debugger, which provides live execution state investigation by means of breakpoints and watchpoints. This chapter presents an empirical study where different manual debugging tools are evaluated. Detailed information is given about each debugger or class of debuggers such as their goals, usability, utilized implementation techniques, and the pros and cons. This classification is intended to summarize the characteristics of different debuggers, see Figure 3.1 below. The presented tools vary in their user interface, architectural design, implementation, and capabilities.

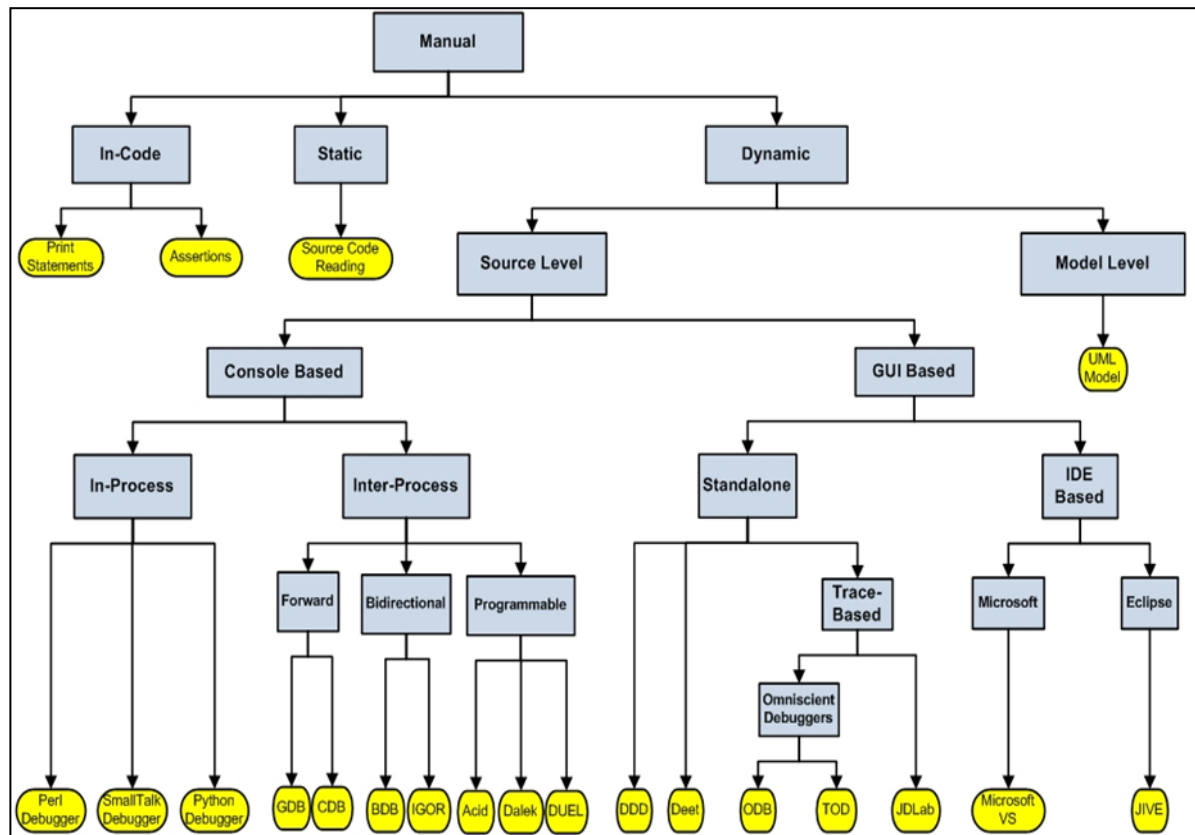


Figure 3.1. Manual Debugging Tools and Techniques

Figure 3.1 shows a tree with the manual debugging tools and techniques presented in this chapter. Rectangular shapes used for internal nodes that represent different classes of manual debugging tools or techniques, whereas circular shapes used for terminal nodes that represent instances or examples of these manual tools and techniques.

3.1. In-Code Debugging

Some programmers may favor in-code debugging using built-in language features such as print statements, assertions, macros, and functions that are used only for debugging purposes. For example, a programmer may have special dedicated functions to traverse some data structure during the debugging process. These in-code debugging techniques may be enabled and disabled using a compiler flag under the developer's control. Figure 3.2 shows a sample debugging macro that can be enabled and disabled using a compiler flag named `DEBUG`.

```
.....  
#ifdef DEBUG  
    #define debug(msg) cout<<(msg)<<endl;  
#else  
    #define debug(msg);  
#endif  
.....  
int main(int args, char **argv)  
{  
    if (args < 2) {  
        debug("There are not enough arguments");  
        exit(-1);  
    }  
    .....  
}
```

Figure 3.2. An Example of Debugging Macros in C++

3.1.1. Print Statements

Print statements are used as a tracing mechanism to ensure that control flow reaches certain execution points with anticipated variable values. Debugging with print statements is considered a bad technique for many reasons, one of which is the amount of overhead associated with inserting, modifying, and removing these statements. Print statements are problematic because they pollute the source code and are always followed with tedious source code cleaning and reorganizing. Dynamic

debugging tools replace print statements with better techniques such as breakpoints and watchpoints, whereas supporting such techniques is applicable in a broad spectrum of debuggers. These dynamic print statements can be implemented as a special case of existing breakpoints and watchpoints. Even though the new techniques provide more capabilities, print statements are still more intuitive especially for novice programmers. In practice, most programmers consider print statements as one of their first debugging choices. When print statements are used for debugging, it is recommended to consider 1) a compiler flag that allows easy enabling and disabling mechanism, or 2) a special debugging macro that wraps print statements inside for cleaner management. For example, the C preprocessor `#ifdef` is used with a compiler flag, such as `DEBUG`, to automate the enabling/disabling process during compilation. See Figure 3.2. Furthermore, different flags can be used for different debugging levels.

3.1.2. Assertions

Assertions are logical expressions that are used to validate pre- and post-conditions and check temporal values of variables and expressions. Assertions are different from exception handling; when an assertion evaluates to false, the execution usually terminates. In contrast, exception handling is often intended to describe how the execution of the program should behave when an unexpected, but valid, event occurs [64]. Like print statements, assertions can be enabled and disabled with a compiler flag. Moreover, languages such as C# and Java provide assertions in the form of modules that are linked into the buggy program during the development process; the .net framework automatically strips assertions out of the released build. Furthermore, assertions are not limited to debugging; they are widely used in program validation and verification, where they are used for reasoning about a program's execution. Sometimes, assertions are inserted into the source code as comments, where the compiler transfers these comments into executable objects— this approach is used in Java Modeling Language (JML) [65] and Temporal Rover [66]. For example, statistical debugging uses assertions to gather real time information about the execution, and unit-testing tools use assertions to decide whether a test succeeded or failed.

3.2. Dynamic Source-Level Debugging

This category includes tools that provide techniques to debug a program based on its execution.

3.2.1. Forward Debugging

Forward or unidirectional debugging includes debuggers with either console-based commands or GUIs that are used to control and navigate the execution of the buggy program. The tools in this category allow only forward execution where the user cannot undo or reverse the execution of a program. If for any reason the user is interested in an execution state before the current one, he/she has to rerun the program and stop it at an earlier execution point. The following are different examples of forward debugging tools.

3.2.1.1. GDB

GDB is a classical example of a typical source-level debugger that provides console-based interface with inter-process debugging architecture. It depends on the `-g` option of compilers such as `gcc`. Using GDB, users can perform debugging by means of breakpoints, watchpoints, single stepping, and execution state investigation. It facilitates what is called *trap based debugging* to control the execution of the buggy program. The debugger dynamically inserts *trap instructions* (illegal instructions) based on the user's interest. For example, the `step` command automatically inserts a trap instruction at the start of the next statement. The `finish` command inserts a trap instruction at the return address of the current function. When a user hits `continue`, the debugger executes the buggy program until it reaches another trapped instruction or it terminates.

Pros: GDB supports a wide range of debugging features that makes it one of the most used source-level debuggers on UNIX platforms. It supports different languages such as C, C++, FORTRAN, and others. GDB has over one hundred basic commands [8]. However, a handful of commands are enough to make effective use of GDB. *Cons:* GDB's trapped instruction mechanism works without any performance problems as long as the number of trapped instructions is relatively small. However, a serious performance problem occurs when the frequency of trap/resume increases before the execution reaches the next stop. For example, consider GDB's counted commands such as `continue 10000`, which can be used to get to the end of a loop. This command may cause the debugger to impose a 10000 trap/resume cycles before it stops. The performance overhead of each trap/resume cycle is roughly around one million processor cycles, most of which is due to the cost of context switching and the system calls used by each trap [1]. A similar situation occurs when the `next` command is used in a recursive function, which may result in a large number of trap/resume cycles before the recursive call is completed [1]. Furthermore, the console-based interface is not easy for novice and inexperienced programmers [8].

3.2.1.2. Perl Debugger

Perl has a built-in debugger named `perl5db.pl`, which is loaded automatically by Perl when the user invokes the script with the `-d` option. It provides console-based interface with in-process debugging architecture. The Perl debugger is an interactive Perl environment with a debugger prompt for user commands. The debugger allows the user to enter arbitrary statements; anything that does not look like an instruction to the debugger is evaluated as Perl code. The Perl debugger provides the user with the classical debugging techniques that are found in GDB [67]. *Pros*: this debugger is fully integrated within the Perl interpreter and can handle arbitrary Perl expressions. *Cons*: it is not really intended for extension or debugging research.

3.2.1.3. PDB

PDB is the standard Python debugger (`pdb.py`). It is a module that defines an interactive source-level debugger for Python programs. It provides a console-based interface with in-process debugging architecture. PDB supports the classical debugging techniques such as breakpoints, stepping and continuing. It also supports post-mortem debugging and it can be called under program control. However, since PDB is a module, it must be imported into the Python program in order to be used; a statement such as `import pdb` must be inserted at the beginning of the Python program. In order to start the debugger a statement such as `pdb.set_trace()` should be inserted into the source code at the point that the user would like to start his/her debugging session; the execution of this statement will start the debugging session with three actions: 1) stop the execution, 2) show the next statement to be executed, and 3) wait for the user input after the `(Pdb)` prompt. At the prompt, the user can perform actions such as 1) execute the next statement with `next` and `step` commands, 2) print the value of a variable, 3) turn off the prompt with the `continue` command, 4) continue to the end of the current sub-routine with the `return` command, or 5) exit the debugger with the `quit` command [68].

Pros: PDB provides a combination of the interactive classical debugging techniques found in GDB and the post-mortem techniques. Furthermore, the interpretive nature and high level of Python make it a good candidate for research experimentation. *Cons*: PDB was not designed with automatic debugging or extension in mind. PDB's module architecture (in-process) suggests that the use of PDB perturbs application behaviors such as garbage collection due to a shared heap.

3.2.1.4. SmallTalk Debugger

The SmallTalk system includes very important tools such as a *browser*, *workspace*, *debugger*, and *inspector*. These tools provide a complete development and testing environment that assist in the

edit-compile-link-run-debug cycle. All SmallTalk objects understand special messages such as `doesNotUnderstand` and `inspect`. The `doesNotUnderstand` message is produced automatically by the SmallTalk runtime system as a result of a runtime error. This message causes the SmallTalk system to provide the user with an *error notification*, which asks the user if he/she is interested in a debugging session. During a debugging session, the programmer is able to modify the program while it is running. In general, SmallTalk runtime errors cause the execution thread to be suspended. In some cases the runtime error can be recoverable. The user may fix the error and continue the execution. This simplifies the process of reproducing the bug. In contrast, the `inspect` message is produced and sent intentionally by the programmer; it allows a user to inspect an object through the inspector window [69].

SmallTalk's debugger has several similarities and important differences compared with UDB presented in Chapter 9. The most important similarity is that both use a thread model of execution, which provides relatively good, high performance access to program state. Another similarity is that most of the debugger is written in the same language as the program that is being debugged. SmallTalk's debugger is less separate from the program being debugged, and relies more on manual instrumentation via subclassing and overriding methods to generate events for dynamic analysis.

3.2.1.5. Deet

Deet is the Desktop Error Elimination Tool, a GUI-based debugger with inter-process debugging architecture. It is a graphical debugger where users can insert breakpoints, watch variables, and navigate the source code and examine data structures all with pointing and clicking. It provides the debugging through nubs, which are small pieces of machine dependent functions inserted into the target program during compilation. Deet debugging commands are performed through communications with these nubs, which allow messages between the debugger and its buggy program to be passed through a pipe or socket. This is an ideal infrastructure for remote debugging. Deet is implemented in tksh, which is an extension to the Korn shell. It utilizes two implementations: one as a layer on top of GDB while the other is based on a modified version of GDB with nub API [70, 71, 72, 73].

Pros: Deet is machine independent, graphical, programmable, distributed, extensible, sits on top GDB. The size of the debugger is less than 1500 lines of shell plus about 1000 lines of C targets machine dependent code for nubs. The small size is attractive because it simplifies the process of understanding, modifying, and extending. *Cons:* Deet's advanced features such as conditional breakpoint extensions are programmable in Tcl or shell. Furthermore, Deet does not provide a match

of GDB. For example, it cannot examine a core dump, evaluate a regular C expression, or debug at the assembly language level.

3.2.1.6. DDD

DDD is the Data Display Debugger, a GUI-based debugger with inter-process debugging architecture [8]. It is a graphical front-end to a set of console-based debuggers such as GDB and DBX. DDD's GUI interface provides the ability to display debugging related data such as program source code, and the ability to perform debugging commands such as breakpoints and watchpoints. DDD does not perform any debugging by itself, commands are forwarded to the underlying debugger and information is displayed and visualized in the GUI interface. For example, DDD runs GDB as a separate process controlled through the traditional GDB command line interface. However, DDD's novelty is based on its ability to visually display and navigate data structures by utilizing a typical debugger. In fact, this feature distinguished it from many GUI extensions to GDB such as XXGDB [74] and CGDB (previously called TGDB) [75]. Furthermore, DDD's design requires no modification of the underlying debugger, which makes it attractive to mainstream developers.

Pros: DDD provides a data visualization and navigation mechanism for simple and complex buggy programs' data structures. It provides a simpler user interface to GDB commands; this interface maybe more attractive especially for novice programmers [8]. Furthermore, DDD implements a clean design that requires no modification of the underlying debugger and no dependencies on particular compilers that may emit distinctive symbol tables. DDD is not tied to local debugging; it can be used in remote debugging facilitated by a long distance remote TTY communication channel, where DDD is on one machine/processor and the underlying debugger is on another machine/processor. *Cons:* DDD endures performance problems entailed by its inter-process communication architecture. DDD's architecture imposes four layers of communications; at one end is the user and at the other end is the buggy program, whereas DDD sits on top of the underlying debugger that runs the buggy program.

3.2.2. Bidirectional Debugging

Typical source-level debuggers are based on forward execution. Naturally, the debugging process develops forward along with the buggy program's execution that moves to the next statement, line, breakpoint, or watchpoint. A user may stop the execution using breakpoints and watchpoints. At each stop, the current state is preserved in global variables, objects, and local variables that still have activation records on the call stack. Returned procedures are part of the execution history that influenced the current state. However, the only way to check their impact is to rerun and stop the

program's execution at a prior point where the procedures' activation records are still on the stack. Moreover, bugs manifest long after their root cause, which is hidden somewhere in the history of execution prior to their revealed time and location. Using conventional forward debuggers, the user can investigate the current state. If there is no evidence about the bug's root cause, the user may restart the execution hoping to stop at an earlier point where the cause of the bug is still accessible [1]. The user may end up investigating incremental modifications on the execution state by stepping program source code line by line.

The ability to go forward and backward in the execution of the buggy program is very useful. It allows the user to undo part of the execution and track the bug backward till its root cause is located. However, reversing the buggy program's execution may require the ability to undo the execution of each statement. Reverse execution requires supporting techniques to keep track of every assignment, save the state after each change, and restore it during the reversal process [37, 38]. This can be a very expensive mechanism, which may need special support from the architecture, the operating system, and the language compiler. For example, it may need different mechanisms such as full interpretation, generation of code with inversion options, or special recompilation [56]. A simpler approach can be achieved by utilizing checkpoints, which are saved images of the execution state at various points. A debugger that uses checkpoints may allow the ability to undo the execution of group of statements up until some previously saved state, which may or may not be the most recent one. Furthermore, the debugger may provide the ability to resume the execution from that reverted point.

3.2.2.1. IGOR

IGOR is an example of a debugger that provides reverse execution [56]. It uses incremental checkpoint facilities called recovery blocks. *Pros*: it needs no code modification and it applies to compiled-languages; no virtual machine is used. It admits to the limitation and the difficulty of being able to reverse every state of the execution such as a network communication or an I/O, with the possibilities of some workarounds [56]. *Cons*: like most reversible debuggers, it suffers from irreversible inputs/outputs such as mouse movements and print statements [56].

3.2.2.2. BDB

BDB is a bidirectional debugger where each conventional forward movement command can be applied backward [1]. The target programming language for this debugger was C and C++ running on Digital/Compaq Alpha based workstation. BDB utilizes a special checkpoint mechanism at each debugger command, which enables it to reverse each command. The implementation tries to follow the user interaction with the conventional debugging by creating checkpoints for every possibly

reversible command. For example, the debugger supports a list of reversible commands such as `next` and `bnext`, `step` and `bstep`, `finish` and `bfinish`, `continue` and `bcontinue`. *Pros*: instead of restarting the execution, a BDB user is able to reverse the execution back to previous points that are saved based on previous reversible debugging commands. It overcomes the common forward debugging problem of overstepping the bug. *Cons*: BDB suffers from platform limitations. The technique depends heavily on the architecture and the operating system. Moreover, the user can only reverse execution based on previous debugging commands. Because it automatically associates checkpoints with reversible commands, if a program crashes before hitting any of these commands, a user will not be able to reverse and undo the execution.

3.2.3. Programmable Debugging

Conventional source-level debuggers such as GDB are considered procedural, and not programmable, because conditional breakpoints and watchpoints, are mostly limited to trivial boolean evaluations, and the user ends up stepping through the source code and examining the program state such as variables, objects, and the execution stack. In contrast, programmable source-level debuggers can be scriptable or declarative. This category includes debuggers with commands in the form of a limited language (or sub-language). Programmable debuggers can be either extensible using special syntax or notations, or scriptable using special notations that reduce the human factor during the debugging process. Scriptable debuggers are different from script debuggers that target scripting languages. This section provides three different examples of programmable debuggers.

3.2.3.1. Dalek

Dalek is a scriptable debugger built on top of GDB to debug C programs. It utilizes an event-based data flow approach to debug sequential programs. Dalek supports two types of events: low-level and high-level; high-level events are constructed from low-level events. Events are represented in a graph where the leaves are low-level events and the interior nodes are high-level events. Typical low-level events represent execution activities such as entering a procedure, exiting or returning from a procedure, or hitting a specific execution point. High-level events can be constructed from lower level events by means of pattern matching or programming language constructs. The latter is used by Dalek, which claims that its data flow approach is more flexible and provides more user access than a pattern matching approach. Dalek's events are associated with an event handler called a callback procedure, which may generate other events. An event handler may suspend or resume the execution of the buggy program. Events are provided to the debugger either interactively or through a file. It can be seen as a language extension to GDB. *Pros*: it provides a scriptable debugging interface for GDB.

Cons: the user has to know and provide the debugger with those events interactively or through files [42].

3.2.3.2. Acid

Acid is a language interpreter with specialized primitives for debugging support. It uses an in-process debugging architecture to provide complex interactions between the debugger and the target program. This in-process design is supposed to simplify differences between different hardware architectures. The developers of Acid criticize conventional debuggers for their limited features. It argues that most of the time, it is hard or even impossible to reproduce the state of the program under GDB, especially when the state includes complex data structures. In contrast, Acid debugging commands are implemented as primitives that can be used by the programmer. There is no need to change or extend the debugging core with new functionalities. Users can build their own debugging context and interface by combining their own functions along with the debugger primitives in different ways [2].

Pros: Acid provides programmable debugging techniques that may simplify the process of reasoning about the behavior of the buggy program, all within a rich and flexible debugging environment. Acid's supporting language provides a powerful assertion mechanism that allows a user to assert and validate the logic of the buggy program including its execution state and data structures. *Cons:* an Acid user needs to put some effort in learning a new debugging language. Furthermore, Acid's in-process debugging architecture intrudes on the buggy program space, which alters the behavior of the buggy program and may change the bug behavior.

3.2.3.3. DUEL

DUEL is an interactive debugger that extends GDB with a high level expression evaluator that constitutes a very high level language for debugging. Expressions are a superset of C that includes generators inspired by Icon, loop iterations for data structures, and conditionals to control the evaluation of the expressions. The user can formulate complex state queries through combining expressions. For example, the command "`x[..100] >? 0`" displays the positive elements of the array `x` associated with their indices. DUEL takes the stand that debuggers should not stick to the syntax or semantics of the target language, but must be more expressive to provide easy and more powerful investigation. *Pros:* it provides a simple mechanism to explore program's data, especially complex data structures. *Cons:* it does not provide the ability to control the processes of the buggy program that is required in some debugging situations, especially when the user is interested in stopping the execution at some condition or line number [127].

3.2.4. Trace-Based Debugging

Most trace-based debuggers are not interactive; they create an execution trace history and allow the user to investigate the trace. They are post-mortem debuggers. Often, traced data are searched using query-based techniques, which may be supported with visualization tools that highlight important features or summarize a big picture.

3.2.4.1. ODB

ODB is an omniscient post-mortem trace-based debugger for Java programs. It traces every change within the execution state of a program that includes two types of events: method call/return and assign. Users are able to go backward in time and investigate the history of the execution. ODB provides a GUI interface, which allows users to navigate the execution history and apply query-based breakpoints. ODB performs an execution trace, and then the user is able to investigate. The traced information is collected based on bytecode instrumentation that occurs at load time [47, 48]. *Pros*: it provides the ability to go backward in time to investigate old execution states such as locating where a variable was assigned long before it caused a crash, and finding where a method has been called, even though it has returned. ODB overcomes some of the problems in classical debuggers; there is no *guessing* where to put breakpoints, and no *fatal* mistakes of going far past the root cause of the problem. It makes the debugging process more deterministic; all data can be saved to a file and exchanged between end-users and developers. *Cons*: it suffers from scalability problems induced by the large size of the traced data. For example, a small program (about 300 lines of code) can generate about 2 GB of traced data in 20 seconds [76].

3.2.4.2. TOD

TOD is a scalable omniscient debugger inspired by ODB. TOD utilizes a distributed database in order to facilitate the ability to process a huge volume of traced data and to permit high performance event recording and querying. It was implemented for Java by utilizing the ASM instrumentation framework [50]. Then it was extended to support Aspect Oriented Programming [49]. *Pros*: it handles the scalability problem and the limitations of the ODB by facilitating distributed database. It also provides control flow navigations, and visualization of traced data. It claims an intuitive user interface, which provides the source window of the current event, the state of the stack frames, and the current objects. *Cons*: TOD's implementation of the database solution makes it difficult to deploy. In order to use the debugger, a user needs to know how to setup this distributed database correctly. Furthermore, there is an index for each possible attribute value. For each event that enters the

database it updates the indexes that correspond to its attributes. This generates more indexing data than event data, which increases trace size up to 5x.

3.2.4.3. JDLab

JDLab is the abbreviation for a set of tools named Java Debugging Laboratory. It debugs Java programs by analyzing traced data. Those tools are built on top of the Java Virtual Machine Debug Interface (JVMDI) for acquiring execution events. This simplifies the development process and puts the focus on the target problem instead of being focused on the instrumentation technique. A JDLabAgent generates about 10 bytes of data per event and it needs about *1ms* to store 100 events. It reduces the amount of traced events by utilizing graph algorithms, which makes the JDLabAgent more usable compared with other trace-based debugging tools. Furthermore, unlike other trace-based tools that track only method entries and exit events, JDLab can reconstruct the complete method execution [76, 77]. *Pros*: JDLab 1) provides traces with no source code or bytecode modification, 2) requires no modification for the virtual machine, 3) has selective monitoring points when the source code is not available, 4) employs low overhead events, and 5) supports event analysis for threads, stack traces, methods' arguments, methods' return values, control flow, and exception handling.

3.2.5. IDE-Based Source-Level Debugging

Some debuggers are integrated within an IDE, which packages and simplifies the *compile-edit-debug* cycle. Most of the time, the user is able to navigate and change the source code, place breakpoints and watchpoints directly on the source code with pointing and clicking. Furthermore, single stepping can be watched directly in the source code. Microsoft Visual Studio and Eclipse are two of the most widely used IDE source-level debuggers.

JIVE is a declarative and visual debugging environment integrated within the Eclipse IDE. It utilizes the Eclipse architecture and benefits from the JPDA debugging architecture. It obtains debugging information through queries over the program's execution trace and specific runtime states. Debugging information is presented through a visual mechanism [40]. *Pros*: it makes use of the JPDA and the Eclipse IDE, which give it a robust infrastructure. *Cons*: it handles only small to medium size programs. It runs slowly because of the nature of the event collection mechanism and the visualization views that are updated after each event.

3.3. Model-Level Debugging

Complex software is often designed with the aid of models such as UML diagrams. UML models provide a high level of abstraction, which describes the system behavior through state machines, activities, and interactions. Debugging a system model in an early stage of the development life cycle would save time later and reduce or prevent costly rework [78]. Manual model-level debugging targets UML behavioral models, which can be debugged using dedicated debuggers.

The UML Model Debugger is built to simulate as much as possible the Eclipse code debugger with features such as 1) manually controlling the debugging session, 2) observing current object attributes, and 3) breakpoints placed on behavioral elements. However, debugging at a very high level of abstraction has to sacrifice debugging features found in lower level debuggers and at the same time requires new advanced debugging features. For instance, there is no need for threads and stack investigation, but there is a need for behavioral model elements such as transitions, and actions [78].

3.4. Summary

This chapter presented various manual debugging tools and techniques. Table 3.1 shows the main characteristics of these tools and techniques based on different categories, which include: the debugging process, the user interface, the debugging tool architecture, and the internal technique used to provide the debugging information. Each one of these tools addresses one or more main the debugging techniques. However, the main problem is their lack for an easy extensible mechanism that simplifies the process of adding new debugging features or allows them to collaborate with various debugging tools. This limitation is addressed by this dissertation.

Chapter 4

Automatic Debugging Tools and Techniques

Each bug is associated with a set of symptoms. At some point, the debugging process may aim at reducing the program or its set of inputs into the smallest possible subset that maintains these symptoms. The precision of the simplification process or the amount of reduction in the generated subset may determine the effectiveness of the debugging tool and the debugging process. One of the reduction techniques is *binary search* that repeatedly eliminates some portion of the program until the root cause of the bug is located [79, 80]. However, even with binary search, the manual investigation is very tedious and time consuming. This intensifies the need for automatic debugging techniques; especially for bugs that are difficult to catch using standard tools and techniques.

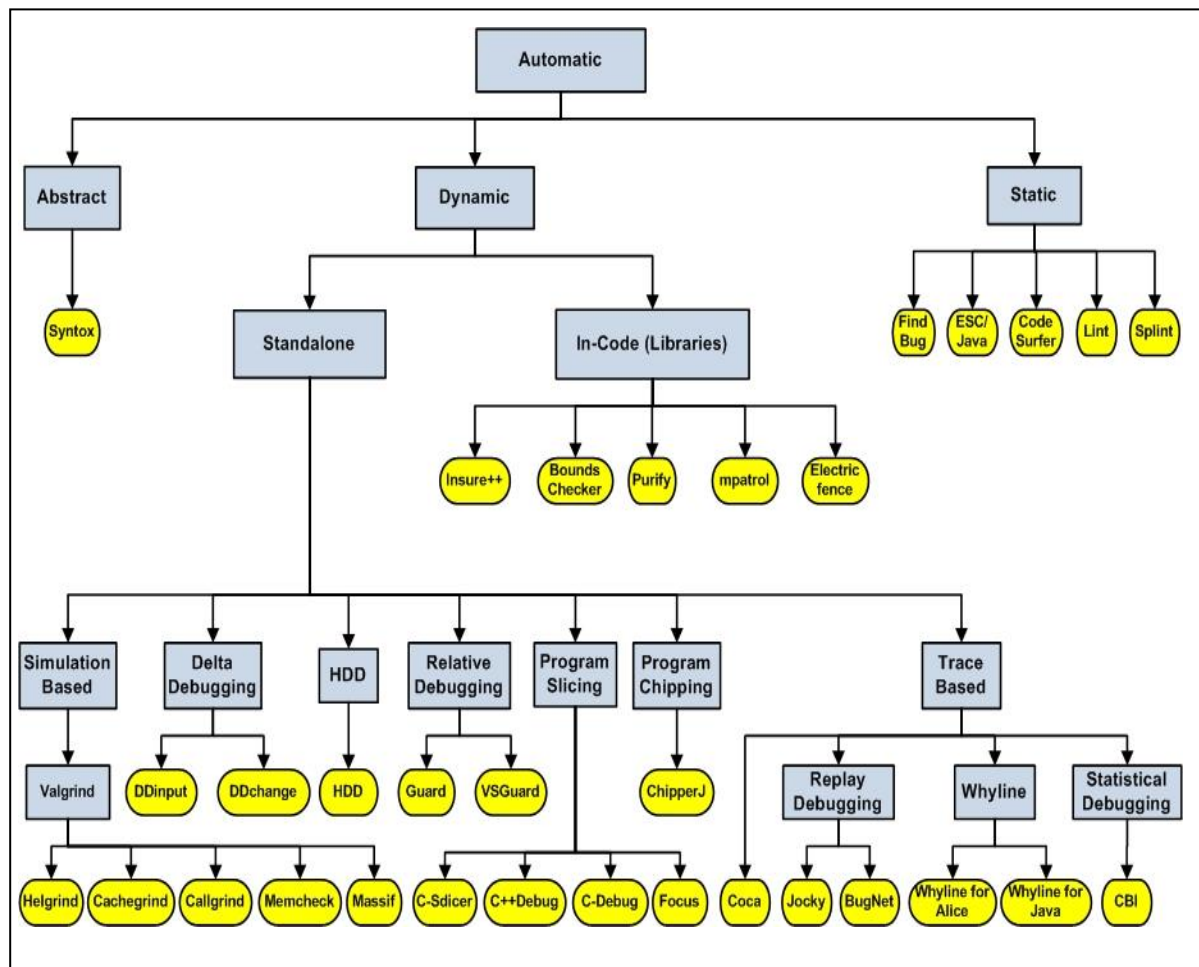


Figure 4.1. Automatic Debugging Tools and Techniques

Automatic debugging aims at improving the efficiency of the debugging process by automating some or all of its boring and time-consuming parts. The ultimate goal is to make the debugging tool smart enough to locate the root cause of a bug. For example, program slicing utilizes algorithms to eliminate some program parts that are irrelevant to a specific test case [29]. In practice, some automatic debugging techniques are still immature. Often, the reduced search space is still too large for the root cause to be located easily. Moreover, the kinds of bugs that are catchable by automatic methods are often uncertain, inconsistent, or perhaps even enigmatic; they can be a bug in one situation while not in another.

This chapter provides a classification for automated debugging tools and techniques. It presents detailed information about each tool's goals, usability, utilized techniques, and the pros and cons. The top level of this classification is divided into two categories based on the kind of analysis in use; whether it is static or dynamic. Each of these techniques is divided into sub-categories. Figure 4.1 shows a tree with the categorization used in this chapter. Internal nodes represent a class of debugging tools or techniques whereas leaves are instances of these tools and techniques.

4.1. Static Debugging

Static analysis is a technique that is used to retrieve valuable information about the program from its source code or object code. They find bugs by analyzing the source code or object code without considering an actual program run. In some situations, the source code is analyzed to find bugs that may occur during the execution of the buggy program. In other situations, the source code is analyzed looking for an already identified bug. For example, static slicing techniques generate a subset of the source code that is responsible for a specific bug.

Static analysis is used in a variety of debugging tools to check semantics, consistent typing, memory allocation, logical statements, and security flaws. Some of the static analysis techniques are standalone tools while others are techniques employed by other tools such as compilers. The following is a list of some of the tools that use static analysis techniques to find potential runtime bugs.

1. **CodeSurfer** is a product of GrammaTech for statically analyzing C programs. It mostly finds bugs by slicing; a slice is a collection of all the code that contributes to the computation of a value. It provides the ability to detect some common language misuses and memory related bugs [86].

2. **Compiler Options** such as the `-Wall` option in the GCC compiler, which enables warnings for many common errors [81].
3. **ESC/Java** and **ESC/Java2** are compile-time program checkers. They find common runtime programming errors in Java programs by static analysis means applied directly on the program source code. Users can control the kind and amount of checking performed through specially formatted comments [88]. ESC/Java2 is an extension that targets JML.
4. **FindBug** is a Java based tool that employs static analysis techniques to find runtime bugs. It works on the object code of the compiled Java program; it does not need the actual source code. This free standalone tool employs the concept of bug patterns, which are common coding practices that are known errors based on a variety of reasons such as misunderstood language features, misused API, and bad use of types and wrong boolean operators. This tool is extendable through its plug-in architecture, but any extension requires expert knowledge of the Java bytecode. In practice, its report of *false warning* falls under 50% [82, 83].
5. **Lint** is a static analysis tool that targets C/C++ programs on UNIX platforms. **Splint** is an open source tool that can be used on a C/C++ program. It is a stronger checker than standard Lint. It checks things such as: unused declarations, type inconsistency, variables used before being assigned, ignored return values, apparent infinite loops [85]. **PC-lint**: is a product of Gimpel Software for statically checking C/C++ programs. It can find suspicious program properties such as uninitialized simple and aggregated variables, unused variables, unused functions, variables that are assigned but not used, and code that is unreachable [87].
6. **PMD** is a Java based standalone static analysis tool that works directly on the source code of the target program. It finds bugs such as dead code, duplicated code, and overcomplicated code [84].

In general, static analysis is usually used to find bugs that do not depend on a specific test run. Because they reason about all possible program runs, static analyzers may perform a deeper analysis than a tool that employs run-time dynamic analysis techniques.

4.2. Abstract Debugging

Abstract debugging is a static semantic-based debugging approach. It uses an abstract interpretation that enables the debugging of programs without their execution. *Abstract debugging* and *abstract interpretation* are different. The former requires precise interpretation in order to find and locate bugs successfully; approximation is not applicable in debugging [37, 38]. In contrast,

abstract interpretation is about finding a safe “*flow-insensitive*” approximation of runtime program properties, which must hold at some point of the execution [38]. Abstract debugging finds the root cause of potential bugs and their conditions through two types of assertions: 1) *invariant assertions*, which they must *always* hold at the predefined control point in a very similar fashion to the assert statement in C, and 2) *intermittent assertions*, which are assertions that must *eventually* hold at a predefined control point. Users can provide the debugger with *invariant* and *intermittent* assertions and the debugger automatically checks for the validity of a program, tests the behavior of certain execution paths, and finds a bug’s root causes instead of their occurrences. Abstract debugging is efficient for higher-order imperative languages as well as logical languages [37].

Syntox is a research prototype that utilizes abstract debugging and targets the Pascal language. It finds bugs related to scalar variables in the program such as array indexing and range sub-types. The user is able to insert assertions into the buggy program source code and the debugger treats any violation of these assertions as runtime errors [37].

4.3. Dynamic Debugging

Dynamic debugging is primarily based on information obtained from the execution of the buggy program. This includes a broad set of debugging tools such as 1) libraries or modules that can be linked into the buggy program, and 2) standalone debugging systems. Some dynamic debugging tools may utilize some information obtained implicitly from a static analysis technique. These tools can be considered hybrid, however, this chapter treats them as dynamic.

4.3.1. Model Based Software Debugging

Automatic Model Based Software Debugging (MBSD) is the process of identifying the location of defects in a program based on models generated automatically from the execution of a program. The generated (observed) model is compared against the intuitive or theoretical model. Model based debugging was first introduced by Console et al [89]. MBSD is an application of Model Based Diagnosis (MBD) [46]. Sometimes, the generated models are criticized for their accuracy, fault assumptions, and their significant computational effort [90]. The difference between the anticipated model and the actual observed model is used to identify components that deviate from the normal behavior and produce the observed behavior. The model is partitioned into sub models based on the actual program source code. In this way, unmatched models can be provided in terms of the actual source code segment.

There are different model based debugging approaches; this section defines only two of them. The first is dependency based models, which are models derived from the dependencies between program statements. Static and dynamic analysis techniques can be employed to find such dependencies. This category includes many sub-models such as: execution trace based dependency model (ETDM), detailed dependency model (DDM), and summarized dependency model (SDM). The differences between models are based on the dependencies themselves and the heap data structures [46, 90].

The second is value based models, which are models derived based on concrete values obtained from the source code of the program using static analysis that utilizes the program's control and data flow. It works by comparing values computed by the program containing faults with values expected by the specification of a test suite. This approach is more precise than the dependency based model. However, the two model approaches implement expensive computations that are proportional to the program size [90].

4.3.1.1. MBD

Model Based Debugging (MBD) [91] is a technique that, instead of analyzing the source code or the execution of the program in order to reduce the search space of the bug location, focuses on how the program should behave according to a behavioral model. MDB can be described as a black-box that takes as input: 1) the buggy program, 2) an extended finite state machine (EFSM) that represents the buggy program's behavioral model, and 3) a failing input sequence. It performs the debugging process by mutating the behavioral model to represent various faulty behaviors. This mechanism reduces the buggy program space, and produces a subset of the behavioral model that can lead to the failure. Furthermore, the subset is ranked as a list of suspicious diagnoses [91].

4.3.1.2. JADE

JADE [92] is a debugger that implements the functional dependency model. It validates models based on the statement level. It combines the standard debugging diagnosis features. The debugging is obtained by representing the program as a dependency model, which is compiled into a logical model. The actual execution of the program is observed to build its behavioral model. Then the observed behavior and the logical model are used to determine potential bugs and the position of their root causes. A bug location is defined based on the statement level [92].

4.3.1.3. EBBA

EBBA is an acronym for Event Based Behavioral Abstraction. It is a high level debugging approach that aims at the debugging of complex distributed systems. EBBA builds an execution model based on the actual behavior of the program and compares it to the expected behavioral model. A set of different behavioral models is constructed from the buggy program to characterize and direct the user with more insight on the investigation. Comparing different models will help characterize and identify faults and their behaviors [36].

4.3.1.4. Ariadne

Ariadne is a post-mortem event-based debugger targeting explicitly parallel languages. It compares the intended program behavior provided by the user with actual program behavior captured by event traces. Events represent 1) low-level inter-process communications, 2) language specific events, and 3) user defined events. Traces are recorded into an execution history graph where nodes represent events and edges represent orderings. The debugger compares the user model and the actual execution history graph looking for a complete or partial match of the sub-graphs [93].

4.3.2. In-Process Debugging (Debugging Libraries)

Dynamic in-code debugging performs dynamic checking on a program's execution properties through libraries or modules that are linked into the buggy program. This category includes different tools, each of which targets a specific execution behavior. Here are some of the most interesting ones:

1. **BoundsChecker** is a product of CompuWare Corporation. It checks memory errors and API calls in C and C++ programs. It instruments the intermediate representation generated by the Microsoft Visual C++ compiler, which may be faster than modifying the original source code [95].
2. **Electric Fence** is a library that can be used to debug memory related bugs in C and C++ programs. It must be linked into the buggy program. It produces warnings for potential memory related bugs such as freeing memory that does not exist [98].
3. **Insure++** is a product of ParaSoft that can find memory bugs such as referencing a null or uninitialized pointer, or an invalid memory location. The program has to be recompiled using this tool instead of one's own compiler [94].

4. **mpatrol** is an open source memory allocation library that targets runtime memory problems in C and C++ programs. It can be used with gcc as a command line option during compilation such as "-fcheck-memory-usage" [97].
5. **Purify** is a product of IBM's Rational Corporation. It checks memory errors and API calls in C and C++ source code and garbage collection problems in Java code. It modifies the object code used to build the target executable [96].

4.3.3. Dedicated Debuggers

Dynamic standalone automatic debuggers include a wide range of automatic debugging tools and systems. Each tool implements a specific debugging technique in favor of a specific class of bugs. This section highlights some of the most distinctive ideas, techniques, and tools.

4.3.3.1. Convergence Debugging

Convergence debugging is an automatic debugging technique that utilizes a set of test cases. It isolates different test cases based on their convergence on the root cause of the failure by analyzing the internal control and data flow of the failed test case. The convergence algorithm selects the test cases that maximize the effectiveness of locating faults by measuring the effectiveness of the test case in isolating the root cause of a failure. It consists of a means for simplifying inputs, internal data, user interaction, and code. The results of these simplifications are analyzed to find the root cause of the failure. It uses the test case that caused the failure to find all *closely-related* and *distantly-related* test cases that also cause the failure. Furthermore, it finds all *distance-successful* test cases. The failure's root cause is generated based on the difference between the failure cause and the converging test cases based on "*tight fault neighborhoods with respect to control and data*". It measures the distance between a set of debug test cases and the actual test case that caused the failure and it finds related failures around an already known fault [29].

Pros: it helps find related failures based on already known faults. In other words, it produces different circumstances that produce the same failure. It provides a tool that is applicable for a wide range of languages such as C, C++, C#, VB, and Java. The tool is based on a commercial tool named Diversity Analyzer [99] used in the Microsoft .net framework. This tool can handle mixed language projects and multiple projects simultaneously, dynamically linked libraries, and multitask code. Its goal is to provide an efficient mechanism to measure the power of a test set to isolate the root cause of a failure [29]. *Cons:* it depends on the programmer to locate the fault based on the convergence debugging data. The tool is limited to the Microsoft .net environment.

4.3.3.2. Program Slicing

Program slicing was introduced in 1979 for debugging. However, it is currently extended to program testing, software measurement, comprehension, and maintenance. It is considered an automated reverse engineering technique. When it is employed for debugging, it extracts some parts of the program based on some computations. The extracted parts can be the good parts or the buggy parts. It includes two major approaches: 1) static slicing that utilizes analysis techniques based on the static information contained in the source code, and 2) dynamic slicing that utilizes analysis techniques based on the program execution such as control flow, data flow, or both. For example, JSlice is dynamic slicing tool for Java programs. Slicing can be 1) forward slicing, which includes all statements related to the slice criterion, 2) backward slicing, which includes all of the program statements that are used in the computation of the slice criterion, 3) hierarchical dynamic slicing [100], or 4) thin slicing [101].

Program slicing is implemented by one of three methods: 1) iterative dataflow equations, 2) relational calculus based on information-flow, and 3) graph reachability by constructing the program's graph dependency followed by implementing graph reachability [102, 103]. *Pros*: slicing tools help users locate source code related to specific conditions such as a relevant variable. Generally, it is more efficient in debugging small programs than conventional tools [27]. *Cons*: slicing tools are unable to point the user at the specific root cause of a bug. Instead, they reduce the search space into a subset that may be as big as one third of the original program. When a user is looking for information relevant to a variable, he/she has to determine what variable is of interest before applying the slicing technique [79, 80].

4.3.3.3. Program Chipping

Program chipping is a simple automatic debugging technique that isolates bugs by chipping away parts of the program based on symptoms. Symptoms might be errors in the output, infinite loops, and unhandled exceptions, for example the specified symptoms are used to reduce the size of the buggy program based on various heuristic techniques including binary search [79, 80]. The goal is to make the programmer focus more on the problematic (symptomatic) part that makes the bad output. It employs simple techniques based on the syntactic structure of the program. This makes program chipping different from program slicing. In program slicing, the user looks for a specific behavior in respect to a variable or set of variables. In contrast, program chipping allows the user to search for a specific behavior in the program as a whole (black-box) and proceed automatically until the bug is found. Program chipping is simpler than slicing and it does not require sophisticated

program analysis techniques, as slicing always does. Program chipping is a specific application of a general technique called *data slicing*, where the buggy program is the data and the chipper is the data debugger used on the syntactic structure of the program [79].

ChipperJ is an application of the program chipping technique. It builds a parse tree for the original program, and then the ChipperJ tool deletes or modifies one or more nodes to generate other parse trees, each with a variant that is validated until the best variant is identified [79]. *Pros*: the results are encouraging with 20% to 35% reduction in the original size of the program. *Cons*: it takes about an hour to perform the reduction of 20% to 35% of a large program such as the Java Compiler. It does not guarantee to find the minimal variant and it does not work on programs that have nondeterministic output such as multithreaded Java programs. Furthermore, the chipper may remove critical read statements from the program that will change the variant [79].

4.3.3.4. Statistical Debugging

Statistical debugging is an automatic debugging technique for finding and locating bugs in released software systems by implicitly collecting real execution samples from real end-users. While other debugging techniques collect information that is limited to a particular failed run, statistical debugging depends on information gathered at all times. It depends on data sampled from a wide range of different actual failed and successful runs. It consists of two phases: 1) sampling the program from real users at a minimal cost, and 2) applying a statistical analysis mechanism to locate and find the root cause of real world bugs [15, 16, 59, 60]. Information is collected from running programs through a sparse sampling mechanism, which is scalable and has little impact on the performance of the system. Information is transmitted into a central database for processing. The gathered information reflects a large number of executions in distant locations [15]. Statistical debugging provides the ability to systematically compare data from failed runs with data from successful runs, and improves the debugging process. Information is gathered sparsely using logic predicates that randomly sample data from released software [60]. *Pros*: its scalability deals with real widely used programs and a wide range of bugs. The sparse sampling mechanism has little impact on the execution of the program. *Cons*: it requires sampling of the program information over millions runs.

4.3.3.5. Delta Debugging

Delta debugging is an automatic way of narrowing down the differences between a failed run and a successful run [22]. It is a fully automatic debugging technique that finds the simplest test case that generates the failure, and highlights the difference between a passing and failing test case. Since reduction of test cases is a human centric process, delta debugging utilizes two algorithms to

automatically minimize a set of test cases. First, a simplification algorithm, which implements a recursive technique to keep examining a smaller set of the input that produces a failure until no smaller set of inputs can be found that can generate the failure. Second, an isolation algorithm that finds a passing set of inputs, which when some elements are added to it, produces a failure. This algorithm finds the biggest passing set of inputs that is a subset of the failing case [21, 41]. For example, DDinput is a plug-in that facilitates delta debugging within the Eclipse IDE. *Pros*: it is an efficient tool for programs that expect structured inputs [41]. *Cons*: the end user must provide the debugger both a successful run and a failed run. Moreover, it takes a considerable amount of time to finish the debugging process. However, this time can be justified by the quality of the results and the precision of the output in pinpointing at the root cause of the bug.

4.3.3.6. Hierarchical Delta Debugging

Hierarchical Delta Debugging (HDD) is a new data debugging technique that is intended to speed up the process of debugging with the delta debugger. It produces a better quality output by minimizing all failure-inducing inputs [41]. HDD focuses only on the simplification algorithm. The technique starts by applying delta debugging to the input data at each level. This excludes a large portion of the input at an early debugging stage. *Pros*: it simplifies the debugging output. It reduces the number of test cases by an order of magnitude over the original general delta debugging simplification algorithm. It also significantly speeds up the simplification time [41]. *Cons*: it is limited to programs that only accept structured inputs such as: 1) a programming language compiler or interpreter that parses input using a context free grammar fed to a compiler or interpreter, 2) an HTML/XML web page that maintains the nested structured inputs, 3) a video codec with limited depth, and 4) a user interface. It works better when there are few dependencies between the input data. It also depends on the programmer to formalize the hierarchy of the input such as building the syntax tree [41].

4.3.3.7. Relative Debugging

Most software programs are in a constant modification process. Often, the modified program produces the same output as the original program does. Relative debugging facilitates the ability to debug two versions of the same program by providing the debugger with the expected similarities between their execution states [55]. Relative debugging is different from delta debugging. The former targets two different executions of two related programs—two different versions of the same program, each with different internal implementation or even different programming language. In contrast, delta debugging targets two different runs, a failed one and a successful one, for the exact

same program. Relative debugging can be used to debug modified programs by comparing their execution states. The bug symptom includes two related programs that generate different outputs or behave differently on the same set of inputs. For example, when a program is ported into a new platform, a relative debugger helps compare data and execution state between the two platforms. This may include environmental changes such as system libraries or new compiler versions. A relative debugger can verify the similarities and find any differences between the two programs [54, 55].

Relative debugging has several implementations. *Guard* is the classical example that supports relative debugging in heterogeneous environments. *VSGuard* is the Microsoft Visual Studio implementation of Guard. It provides a wizard to build one solution for a project that is ported from Microsoft Visual Studio version 6.0 to Visual Studio .net. The user is able to debug the new program by specifying assertions on the related data structures. *Pros*: a relative debugger may run the two programs simultaneously, and it compares them in real time. It finds differences which are associated with the exact line in the source code. Besides the significant relative debugging techniques, it maintains the traditional functionalities of a classical debugger. A relative debugger helps shift the developer's concerns from the actual state of the program into what is the difference and where it starts to happen [54, 55]. *Cons*: relative debugging allows the user to compare the two programs' execution based on expected predefined associations, but it depends on the user who has to specify the points of comparison and anticipate the similarities.

4.3.3.8. Replay Debugging

Replay Debugging (or Record Replay Debugging) is a class of debuggers that provides the developer with a simple mechanism to reproduce a bug that was encountered by the end-user at his site. It is different from recording the final core dump caused by a crash and sending it to the developer that is adopted by many software vendors. A replay debugger may continuously record information from released software. However, only the recorded information before the occurrence of the crash is sent to the developers, so they can deterministically replay and reproduce the bug in their environment; this may include replaying the last several million instructions before the crash [53]. For example, *BugNet* focuses only on the application level events; it does not record any event or instruction from the host operating system. So, it cannot replay the complete system execution [53]. *Jocky* is another example of this kind of debugger [104]. However, *Jocky* is a library that is linked into the program to record invocation of system calls and CPU instructions. It utilizes record replay debugging that targets interactive and distributed systems running on a Linux platform. *Jocky* simplifies tracking complex communication with the operating system. It implements a form of checkpoints that simplifies the management of long-running programs [104].

4.3.3.9. Whyline

Whyline debuggers simplify the debugging process by elevating the human interaction with the debugger to the natural language level. A user is able to ask typical questions about the execution of the program, such as *Why did?* and *Why did not?* It implements a trace-based debugging approach that tracks the complete execution history. The approach analyzes the traced data and provides the user with information in terms of answers to the provided questions. Two different Whyline debuggers have been implemented. The first is for the Alice framework and the other is for Java programs. Whyline for Java instruments the buggy program's bytecode using the `java.lang.instrument` package [27]. *Pros:* Whyline invented a superior debugging interface that provides the ability to ask natural language questions about the program's execution properties; it elevates the debugging process to a new level of interactions. A study has found that Whyline debuggers reduce the debugging time, especially for novice programmers. *Cons:* it faces scalability limitations due to the huge volume of traced data. For example, it is limited to programs that do not execute for more than a few minutes. This prevents its adoption in long running programs.

4.3.3.10. Coca

Coca is an event-based automated debugger for C. It builds a trace of events, where each event has a semantic value and attributes. It provides a Prolog query-based debugging interface driven by the attributes of the runtime event. The searching mechanism combines data and dataflow instead of only one. It differs from most trace-based debuggers in its event manipulation mechanism. Execution events are not stored in any kind of database. Instead, Coca provides an on the fly analysis mechanism executed synchronously along with the trace. It implements breakpoints based on events and language semantics. Coca claims that conventional source-level debuggers such as GDB are missing the semantic part. Coca events are fine-grained, and are used to model the sequential execution of programs written in C. *Pros:* it provides automatic debugging mechanism with on the fly event analysis techniques. *Cons:* it requires the user to master at least a handful of Prolog primitives in order to perform a simple debugging session for a C program [105].

4.3.3.11. Valgrind

Valgrind provides dynamic error detection for runtime bugs such as dangling pointers and memory leaks. It utilizes a simulation-based technique that models the target CPU for debugging and profiling. It provides an automated debugging approach based on synthetic CPU simulation. It analyzes runtime properties and detects specific execution faults such as memory corruptions. This

type of debugging support depends heavily on the host operating system and the target architecture. Its dependency level complicates any attempt at porting its underlying mechanism to new platforms or architecture. For example, regardless of Valgrind's outstanding debugging capabilities, it is still limited to the UNIX based operating systems; in particular Linux. Currently, Valgrind is supported on architectures such as x86, amd64, ppc32, and ppc64. *Pros*: Valgrind is designed with ease of extensibility in mind. For example, new tools can be created without any need for modification of its core structure. It provides exceptional debugging capabilities for C and C++ programs, especially when it is used to debug memory corruptions. Moreover, it requires no modification on the target program with any instrumentation or special compilation. *Cons*: it suffers from a noticeable delay that ranges from a 20x to 50x slowdown during the evaluation of the buggy program. Valgrind's lack of portability for Windows and Mac OS limits its value [18, 19, 58]. Valgrind has many useful extensions that include:

1. *Memcheck* is a memory-management checker that detects memory problems such as leaks and uninitialized memory. The tool monitors critical program activities such as reads, writes, free, delete, new, and malloc.
2. *Cachgrind* is a cache profiler that simulates the CPU cache such as L1, L2, and D1. It detects all cache misses in programs.
3. *Callgrind* is an extension to the Cachgrind that utilizes the caller-callee relationship in reasoning about their role in your cache misses. The generated data from this tool is huge. *KCachgrind* is a KDE visualization tool that can simplify the process of reading this tool's output.
4. *Massif* is a heap profiler that measures the amount of heap memory used by the program along with heap blocks and the stack size.
5. *Helgrind* is a thread synchronization detector that finds synchronization errors in the use of the pthread primitives, potential deadlocks, and data races.

4.4. Summary

Throughout this chapter, various automatic debugging tools and techniques were presented. Table 4.1 shows the main characteristics of these tools and techniques based on different categories, similar to the ones presented in the summary of Section 3.5.

Table 4.1. Automatic Debugging Tools and Techniques

No.	Debugging Tool/Technique	Debugging Process				User Interface			Architecture				Implementation	
		Abstract	Static	Dynamic		IDE-Integrated	GUI-Based	Console-Based	In-Process	Inter-Process	Programmable	Extensible	Trace-Based	Model-Based
				Forward	Reversible									
1	gcc -Wall		X						X					
2	FindBug		X						X					
3	PMD		X						X					
4	Lint/Splint/ PC-Lint		X						X					
5	CodeSurfer		X						X					
6	ESC/Java		X						X					
7	Syntox	X	X											
8	MBD			X										X
9	JADE			X										X
10	EBBA			X										X
11	Ariadne			X										X
12	Insure++			X					X					
13	BoundsChecker			X					X					
14	Purify			X					X					
15	mpatrol			X					X					
16	Electric Fence			X					X					
17	Convergence Debugging			X										
18	Program Slicing			X										
19	Program Chipping			X										
20	Statistical Debugging			X										
21	Delta Debugging			X										
22	HDD			X										
23	Relative Debugging(Gard)			X		X	X			X				
24	Replay Debugging				X									
25	Whyline			X			X			X			X	
26	Coca													
27	Valgrind			X				X	X		X	X	X	

Part II

Event-Based Debugging Framework

Chapter 5

Alamo Monitoring Framework

Alamo stands for **A Lightweight Architecture for Monitoring**. It is a monitoring framework developed originally to support program visualization. Alamo is integrated within the Icon and Unicon virtual machine [108]. A subset of Alamo was implemented for C and Python [106, 107]. This dissertation builds on Alamo's monitoring features to facilitate a high level abstraction layer for Unicon debugging tools. The implementations of the most needed extensions are presented in Chapter 6. The result of these extensions is called AlamoDE (Alamo—Debug Enabled), which is presented in Chapter 7. This chapter presents Alamo's most important features, within Unicon's virtual machine, that is used as foundations for the new AlamoDE.

5.1. Unicon's Co-Expression Type

Unicon's threads are called *co-expressions*. Co-expressions provide synchronous, but not simultaneous, expression evaluation mechanism within Unicon's virtual machine. Unicon's co-expressions are similar to *co-routines* found in other languages. Co-routines are procedure calls where the state of local variables and execution control are saved to be resumed at the next entrance to that procedure. In contrast, Unicon's co-expressions are independent threads of control extended to include arbitrary expression evaluation. This capability of synchronous co-expressions inside the virtual machine provides the ability for different expressions (statements) to be evaluated in a synchronous fashion within the same procedure.

Unicon's co-expressions are in-process threads that are hidden from the operating system. The evaluation of a co-expression requires both an interpreter stack and a C stack that are separate from the stacks of the main program. This independent evaluation mechanism provides clean intercommunication facilities within the same address space. This makes co-expressions suitable for very high level fast communication techniques with no intrusion of one co-expression into another. Furthermore, a co-expression context switch does not include any operating system calls. Because they are synchronous, co-expression switches are much faster than typical thread switches such as those provided by the *pthread*s library [116].

5.2. Architecture

Alamo's architecture is based on the thread model of execution monitoring, where a monitor program and its target program are separate threads in a shared address space. Alamo extended the co-expression facility with the ability to load a program. Each loaded program is set up with its own code, static data, stack, and heap, but without linking symbols into the current program. This capability allows a program to load another program and execute it in a controlled environment. Standalone programs can be loaded and executed as if they were co-expressions of simple expressions or procedures. Switching between 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.

Figure 5.1 shows a monitor program and its target program all within the same Unicon's virtual machine. The monitor program resides in the main co-expression (thread #0) whereas the target program resides in a different co-expression (thread #1). Whenever the monitor program requests an event from the target program, a lightweight co-expression context switch transfers control to the target program. Then the target program resumes its execution until its interpreter encounters a runtime event of interest to the monitor. Then another co-expression context switch transfers the control back to the monitor program.

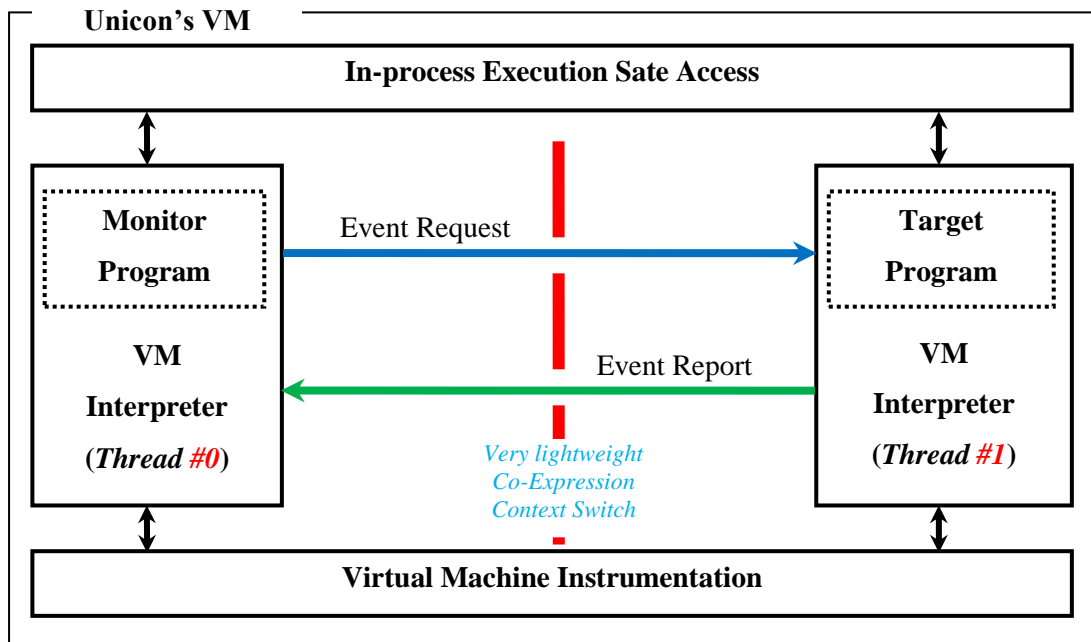


Figure 5.1. Alamo's Architecture

5.3. Features

Alamo is a monitoring framework adequate for passive observation of program execution, suitable for high level sophisticated software visualization tools. The framework provides execution monitoring through a variety of tools and techniques that are integrated within Unicon's virtual machine and its runtime system.

5.3.1. VM Instrumentation

One of the most difficult parts of writing execution monitors is the instrumentation task. Software instrumentation can be manual or automatic. Specific instrumentation frameworks can be employed to instrument the target program with specific instrumentation points, or *sensors*, that monitor its execution properties. Most automatic instrumentation tools modify the program's source or object code. For example, most Java monitoring tools instrument the target program's bytecode at load time [24,25,47,48]. In contrast, Alamo facilitates a third kind of instrumentation mechanism, which instruments the virtual machine itself where the target program is to be executed. Specific locations in the source code of Unicon's virtual machine and its runtime system include conditions that test for specific execution events. Even though this approach has some potential performance overhead especially for unmonitored programs, it provides a seamless instrumentation mechanism, which requires no special compilation and no source or object code modification. This makes it very attractive for experimentation and rapid prototyping of high level visualization tools and techniques.

5.3.2. Dynamic Loading

Dynamic loading is the process of loading modules into an application at runtime rather than at compile time; the dynamic loading does not include merging names of the loaded modules with the host program. It is a subset of dynamic linking. Dynamic linking is the process of linking a module into an application at runtime; it includes loading and merging names. The linking process merges the executable and the linked module(s) [117]. The result is a shared name space, which may cause some naming conflicts between the application and the linked module(s) and perturbs the application behavior due to the shared memory such as stack and heap.

Ideally, the monitor program should have no intrusion on the target program. Achieving this clean behavior is not always feasible; especially when a monitor program is designed to depict precise execution properties and behaviors. Often, the accuracy and reliability of these monitors increases whenever their intrusion on the execution of the target program decreases.

For Alamo and its extension AlamoDE, standalone target programs are dynamically loaded into a separate execution environment within the virtual machine process, see Figure 5.1 above. This type of in-process but separate execution environment is needed for many reasons:

1. No intrusion on the target program space. Each loaded program has its own execution environment
2. No naming conflicts between the variables of the monitor and target programs
3. Simple loading and unloading mechanism that allows the ability to load more than one program within the same monitoring session
4. Simple and fast access features—no operating system is involved and no system calls are used. Often, in-process is much faster than inter-process communication even when processes reside on the same machine

Alamo's dynamic loading and execution model minimizes the prospect that a depicted behavior is due to the act of monitoring instead of the actual target program behavior. This provides an ideal model for debugging.

5.3.3. Synchronous Execution

As it was mentioned in Section 5.1, Alamo provides synchronous and not simultaneous execution mechanism. This allows easy manipulation of the information obtained from the target program. The interpreter of the target program suspends before it reports execution events to the monitor program based on the monitor's current request. This allows the monitor program to utilize access functions to further investigate the execution state of the target program after every reported event. For example, the monitor program is able to look up the target program's state such as global and local variables, keywords, and data structures. Furthermore, while the target program is stopped, the monitor program is safely able to modify the set of monitored events before the next resumption of the target program's execution.

5.3.4. In-process Execution Model

Alamo's event-driven monitoring provides a shared address space, but independent execution model where the monitor program is separate from the target program. Each program has its own control flow; only one of the two programs is running at any point in time. This execution model has several advantages:

1. It provides a simple communication mechanism that allows the monitor program full access to the target program space. This simplicity allows for complicated communication patterns that otherwise would not be considered for reasons of performance
2. It simplifies the process of writing general execution monitors that can be applied to different programs easily
3. It allows simple management for the monitored events and state of the monitored program, because of the synchronization between the monitor program and the target program.
4. It provides each program with its own memory region. Memory allocations in the monitor program do not affect memory in the target program and vice versa. This memory independence also affects the garbage collector behavior
5. The in-process execution model excludes any operating system involvement that might slow down any related operation

5.4. High-Level Execution Monitoring

Alamo provides a programmer with high level primitives that make programming a monitor as simple as any other programming task. This following three sub-sections discuss Alamo's features such as event masking, loading the target program, and activating the target program.

5.4.1. Event Masking

Alamo supports a total of 118 kinds of events. Alamo's events consist of a tuple that pairs a code with an associated value. The *event code* represents the kind of action occurring in this execution of the target program, whereas the *event value* represents a relevant value related to that action. For example, if the event code is `E_Assign`, then the event value represents the string name of the variable that is to be assigned. If the event code is `E_Line`, then the event value represents the actual source code line number that is just about to execute.

The monitor program may receive millions of events from the execution of a small program. An event filtering mechanism is needed to optimize the monitoring process. Alamo introduced the notion of *event mask*, a dynamic set of event codes used to filter the target program events before they are reported back to the monitor program. The event mask provides a simple but dynamic control over the execution of the target program and its prospective events. It reduces the huge volume of reported events to the ones that are of interest to the monitor program. This helps build more efficient task-oriented monitor programs.

5.4.2. Loading the Target Program

An execution monitoring task starts by loading the target program. A target program is loaded and initialized within Unicon's virtual machine using the `EvInit()` library primitive, which loads and initializes the target program. It takes the target program name along with its list of arguments, and an optional stack, string-heap, and block-heap sizes. When this function executes, it sets the keyword `&eventsource` with a pointer to the loaded program space—a structure that maintains the loaded program execution state. Figure 5.2 shows a monitoring template that monitors procedure and method activations of call and return.

```

1  $include "evdefs.icn"
2  link evinit
3  procedure main(args)
4    local eventmask
5
6    EvInit(args)
7    eventmask := cset(E_Pcall || E_Pret)
8    while event := EvGet(eventmask) do {
10     case event of {
11       E_Pcall: { ..... }
12       E_Pret: { ..... }
13     }
14   }
15 end

```

Figure 5.2. Sample Alamo Monitor

5.4.3. Activating the Target Program

Alamo's execution and monitoring control is *event-driven*, a programming model that captures the execution properties of the target program using events or sensors. The primary primitive in the activation process is `EvGet()`, which resumes the execution of the target program. This primitive allows the monitor program to specify the set of requested events before the next resumption of the target program. The `EvGet()` primitive activates the target program up until the next available event. Internally, it activates the co-expression currently pointed at by the keyword `&eventsource`. When a target program is activated, it runs until an event is encountered that is of interest to the monitor program. The interpreter of the target program reports the next available event code to the monitor program as a return value from the `EvGet()` primitive, and it fails when there are no more events and the program terminates. This simple function call interface allows even novice programmers to write

simple execution monitors. A programmer can always access the last reported event code and value using the keywords `&eventcode` and `&eventvalue` respectively.

The monitor program can resume the execution of the target program millions of times and the user may interact with the monitor program to control the execution of the target program. When the monitor program receives a specific event, it may evaluate the received event and perform one or more of the following activities:

1. Reactivate the target program for the next event: perform its next call to the `EvGet()` primitive
2. Modify the set of requested events
3. Inspect further the state of the target program through high level state access primitives,
4. Forward the received event into one or more external monitors
5. Terminate the target program.

5.5 Limitations

Utilizing the Alamo monitoring framework as a debugging framework showed that it endures some limitation. These limitations are:

1. Did not allow a monitor to change local variables in the target program
2. Did not support syntax monitoring—needed for some of the automatic debugging needs
3. Did not handle signals gracefully
4. Frequent context switches; lightweight plus high occurrence rate accumulates to performance problem.
5. More filtering needed before the event is reported
6. Did not take full advantage of the in-process architecture, for example stack trace

These issues were addressed in Chapters 6 and 7.

Chapter 6

AlamoDE: Alamo's Extensions for Debugging Support

AlamoDE is an extension to the Alamo framework that enables debugging tools and techniques to be written at a high level of abstraction. AlamoDE adds to the original Alamo framework new features that:

1. Include debugging-oriented virtual machine instrumentation
2. Support additional execution state inspection and source code navigation, and
3. Provide debugging tools with the ability to change the execution state by safely assigning to a buggy program's variables and procedures

This chapter provides an overview of the implementation of the most important underlying extensions. Some of these extensions are general additions to the Unicon virtual machine and its runtime system; in favor of the AlamoDE, while the rest are extensions to the Alamo monitoring framework. All sections, except section 6.1, are implemented for this dissertation. The work described in Section 6.1 was originally done by Griswold and Townsend, and later adapted and extended by Jeffery [108]. This chapter is based on material from [119].

6.1. Virtual Machine Instrumentation

Event-based debugging support needs instrumentation, which can be inserted into the source code, the bytecode, or the virtual machine itself. Implicit instrumentation within the virtual machine and its runtime system provides a simple mechanism for getting execution events out of a running program. However, instrumentation always incurs overhead in space and processing time. Unicon has a small virtual machine (about 700KB with the instrumentation). A top priority for Unicon's implicit instrumentation is to minimize the processing time overhead cost, especially for unmonitored execution.

Originally, Alamo was an optional extension to the Icon virtual machine, because Alamo's instrumentation imposed a cost even when monitoring was not being performed. In Unicon, a means was developed to include Alamo at very low cost (other than code size) in the production VM. This integration allows the debugger to run on the virtual machine synchronously along with the buggy program, which is the only one affected by the instrumentation.

AlamoDE maintains two versions of 30 runtime functions in the binary executable VM that contain instrumentation. One version is uninstrumented and used in any unmonitored execution; the other version is instrumented and used when a program is monitored. Not all of the instrumented functions are used when the program is under monitoring; a dynamic binding associates the instrumented or uninstrumented function with the current execution state based on the current event mask, which is specified by the debugger. A table maps event codes into their instrumented functions. Whenever an event is added to the monitored events (event mask), the related instrumented function is used. If an event is removed from the event mask, the original uninstrumented version of the function is restored.

Inside the Unicon virtual machine source code, the name of the instrumented function uses the suffix “_1”, whereas the name of the uninstrumented version of the same function uses the suffix “_0”. Functions that contain instrumentation use macros to maintain one copy of the source code, which simplifies the maintenance effort. Using this method of dynamic binding, the instrumentation imposes no cost on the execution time of the virtual machine until the program is debugged or monitored, and the only instrumented functions used are the ones relevant to the currently monitored events, which are specified by the event mask and customized by the debugging tool and the programmer.

6.2. Inter-Program Variable Safety

In order for a debugging tool to be able to change the value of a variable inside a buggy program, the tool must have access to the state of the buggy program. The debugging tool and its buggy programs are loaded into different co-expressions inside the same virtual machine. It is possible for one of the co-expressions to obtain a reference for a variable that is either on the stack, in the static data section, or in the heap of the other co-expression. While the first co-expression is trying to change a variable in the second co-expression, a context switch may allow control to be transferred to the second co-expression. A memory violation might occur if the second co-expression executes further while the first co-expression has a reference to a local variable; a reference to a variable that lives on the stack might become invalid. For example, this can happen if the procedure returned and its activation record is popped off the stack. Since co-expressions are synchronous this is admittedly an unlikely occurrence that would only be caused by a deliberate adversary.

The implemented solution is a trapped variable technique [109]. Trapped variables are not new to the Icon and Unicon implementation. For example, some keywords such as `&trace` require special checking before they are assigned. However, this dissertation presents the implementation of a special

case of trapped variables between simultaneously executed co-expressions. Whenever one co-expression obtains a reference to the state of another co-expression, a trapped variable block is allocated and the assignment uses a reference to this trapped variable to ensure that no context switch ever occurs between the time the reference is obtained and the variable new value is assigned.

Figure 6.1 illustrates how trapped variables are used. The first co-expression contains a reference to a trapped variable block, which references the actual variable in the second co-expression. This new block holds information about the current number of context switches between the two co-expressions, see Figure 6.3. This number is compared to the very recent one just before writing to that variable. If there is any difference between the number of context switches when the reference was obtained and when the reference is written, then this technique detects the invalid assignment and issues a runtime error. This newly introduced trapped variable block is allocated using a new macro described in Figure 6.4.

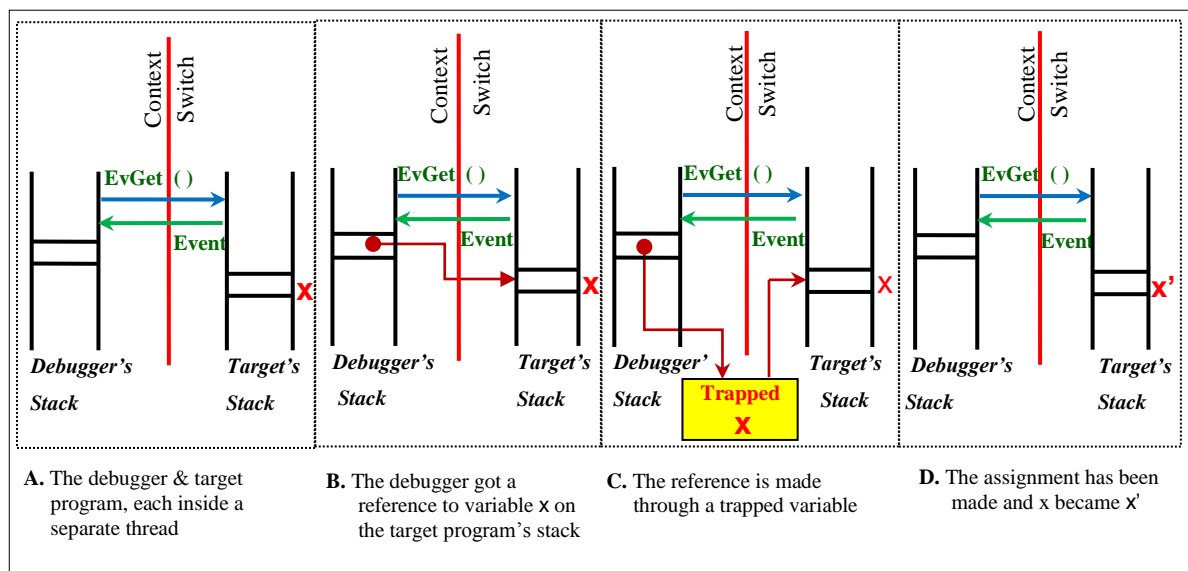


Figure 6.1. Trapped Variable Implementation

This new technique produces a runtime error if a monitor deliberately invokes the subject program, which can only happen if a context switch occurs in the middle of an assignment to a monitored trapped variable. Figure 6.2 shows that this critical section can occur inside an Alamo monitor in unlikely scenarios. The statement calls `EvGet()` and transfer control to the buggy program between the time the variable *x* is referenced and its assignment, but it is not easy. Not surprisingly, the code for a normal debugger does not do any such thing. The safety feature was added to the language to extend the `variable()` function to produce references to local variables while a program is paused.

```
1(variable("x", &eventsourc, 1), EvGet()) := 5
```

Figure 6.2. Sample expression where assignment can be violated

```

1  #ifdef EventMon
2  struct b_tvmonitored {           /* Monitored variable block */
3      word title;                 /* T_Tvmonitored */
4      word cur_actv;             /* current co-expression activation */
5      struct descrip tv;         /* the variable in the other program */
6  };
7  #endif                          /* EventMon */

```

Figure 6.3. The New Data Structure Introduced for the Trapped Variable

```

1  #ifdef MultiThread
2      alctvtbl_macro(alctvtbl_0,0)
3      alctvtbl_macro(alctvtbl_1,E_Tvtbl)
4  #else                               /* MultiThread */
5      alctvtbl_macro(alctvtbl,0)
6  #endif                               /* MultiThread */
7
8  #ifdef EventMon
9  #begdef alctvmonitored_macro(f)
10 /*
11  * alctvmonitored - allocate a trapped monitored variable block in the block
12  * region. no need for event, unless the Monitor is a TP for another Monitor.
13  */
14
15 struct b_tvmonitored *(register dptr tv, word count)
16 {
17     tended struct descrip vref = *tv;
18     register struct b_tvmonitored *blk;
19
20     AlcFixBlk(blk, b_tvmonitored,T_Tvmonitored);
21     blk->tv = vref;
22     blk->cur_actv = count;
23     return blk;
24 }
25 #enddef
26
27 alctvmonitored_macro(alctvmonitored)
28 #endif                               /* EventMon */

```

Figure 6.4. The Allocation Macro Introduced for Trapped Variables

6.3. Syntax Instrumentation

Unicon's source code information, such as line numbers and file names, is already built into a sparse table compiled by the linker as part of the executable binary format—named bytecode or icode. The tables associate each Interpreter Program Counter (IPC) with its corresponding line number and file name. These tables are employed by Alamo to obtain source code information based on the current IPC, which points to the current virtual machine instruction at which the target program is stopped.

Unicon's bytecode executes as a sequence of virtual machine instructions. Like most binary code formats, the bytecode formerly contained no information about the actual syntax of the source code. However, some automatic debugging facilities need information about the syntax of the running program. For example, an automated debugging technique that locates frequently failed loops needs to know when the execution of the buggy program enters and leaves a loop and what type of loop it is; such as a `while` loop.

The solution is to add a new pseudo instruction that is managed by the translator and the linker. The new `Op_Synt` syntax pseudo instruction extends the `line#/column#` table with information about the syntax. It is a reasonable solution because the only cost is a small increase in the size of the table. The cost of retrieving the syntax information from the table is paid for only when a program is monitored and that information is needed.

32-bit Interpreter Program Counter (IPC)	16-bit Column Number	16-bit Line Number	
A. Original table field layout			
32-bit Interpreter Program Counter (IPC)	11-bit Column Number	5-bit Syntax Code	16-bit Line Number
B. Modified table field layout			

Figure 6.5. Unicon's Line/Syntax/Column Table

The `line#/column#` table was transformed into a `line#/column#/syntax` table without altering its position in the bytecode files. See Figure 6.5. The table entry is a 32-bit integer; the 16 most significant bits were for the column number and the 16 least significant bits were for the line number. The maximum possible line/column number is 65535, which is more than is needed for a column

number. AlamoDE changes the column number bits to be the 11 most significant bits, and the remaining 5 bits are used for syntax information. The new design reduces the maximum possible column number to 2048, which is still more than enough for a column number in handwritten source code. The new 5-bit syntax code can hold up to 32 different syntax indicators. Table 6.1 includes the currently supported syntax codes in the Unicon virtual machine.

Table 6.1. Syntax Events and Codes

Syntax	String Code	Integer Code
unidentified syntax	any	0
entering case expression	case	1
exiting case expression	endcase	2
entering if expression	if	3
exiting if expression	endif	4
entering if/else expression	ifelse	5
exiting if/else expression	endifelse	6
entering while loop	while	7
exiting while loop	endwhile	8
entering every loop	every	9
exiting every loop	endevery	10
entering until loop	until	11
exiting until loop	enduntil	12
entering repeat loop	repeat	13
exiting repeat loop	endrepeat	14
entering suspend loop	suspend	15
exiting suspend loop	endsuspend	16




The newly added pseudo-instruction only appears in the human readable object files (named *ucode*) and is used by the linker while generating the executable bytecode. Figure 6.7 shows the automatically generated ucode of the program presented in Figure 6.6. Part A is the ucode file before the syntax instrumentation, whereas part B is the ucode file after the syntax instrumentation.

```

1  procedure main(arg)
2      local i := 1
3      while i < 10 do{
4          write("Hello World !!!")
5          i += 1 }
6  end

```

Figure 6.6. Sample Unicon Program

1	version U9.0.00	1	version U9.0.00
2	impl local	2	impl local
3	global 1, 0,000005,main,1	3	global 1, 0,000005,main,1
4	proc main	4 5	proc main
5	local 0,001000,arg	6	local 0,001000,arg
6	local 1,000000,i	7	local 1,000000,i
7	local 2,000000,write	8	local 2,000000,write
8	con 0,002000,2,10	9	con 0,002000,2,10
9	con 1,010000,15,110,145,154,	10	con 1,010000,15,110,145,154,
10	154,157,040,127,157,162,		154,157,040,127,157,162,
	154,144,040,041,041,041		154,144,040,041,041,041
	con 2,002000,1,1	11	con 2,002000,1,1
11	declend	12	declend
12	filen test.icn	13	filen test.icn
13	line 1	14	line 1
14	coln 11	15	coln 11
15	mark L1	16	synt any
16	lab L2	17	 mark L1
17	line 2	18	lab L2
18	coln 5	19	line 2
19	mark0	20	coln 5
20	pnull	21	synt while
21	var 1	22	 mark0
22	int 0	23	pnull
23	line 2	24	var 1
24	coln 13	25	int 0
25	numlt	26	line 2
26	unmark	27	coln 3
27	mark L2	28	synt any
28	mark L5	29	 numlt
29	var 2	30	unmark
30	str 1	31	mark L2
31	line 3	32	mark L5
32	coln 13	33	var 2
33	invoke 1	34	str 1
34	unmark	35	line 3
35	lab L5	36	coln 13
36	pnull	37	synt any
37	var 1	38	invoke 1
38	dup	39	unmark
39	int 2	40	lab L5
40	line 4	41	pnull
41	coln 10	42	var 1
42	plus	43	dup
43	asgn	44	int 2
44	lab L3	45	line 4
45	unmark	46	coln 10
46	goto L2	47	synt any
47	lab L4	48	plus
48	line 2	49	asgn
49	coln 5	50	lab L3
50	unmark	51	unmark
51	lab L1	52	goto L2
52	pnull	53	lab L4
53	line 6	54	line 2
54	coln 1	55	coln 5
55	pfail	56	synt endwhile
56	end	57	unmark
57		58	lab L1
		59	pnull
		60	line 6
		61	coln 1
		62	synt any
		63	pfail
		64	end
A. Sample <i>ucode</i> without Syntax Info		B. Sample <i>ucode</i> with Syntax Info	

The while loop (condition + body)

Figure 6.7. Sample *ucode* Format Before and After the Syntax Instrumentation

Part B of Figure 6.7 shows the new pseudo-instruction `synt` added to the ucode after the originally implemented `line` and `colm` (column#) pseudo-instructions. The binary executable is assembled from one or more ucode files using the linker. The linker processes the `synt` pseudo-instruction by inserting the syntax code specified by its operand into the `line#/column#/syntax` table.

AlamoDE presents syntax information as a new selectable event code `E_Syntax` and its related event value is the syntax code, see Table 6.1. A monitor program can inquire the current syntax name at any time using the newly added keyword `&syntax`. This keyword's presence is limited to the monitored program, where it is accessed using the Alamo `keyword()` function.

Figure 6.8 shows a sample monitor program that uses the new `E_Syntax` event. This monitor prints the line number and syntax name for every executed source line. The `E_Line` event code is reported whenever the execution changes into a new source line. This program also prints the line number and syntax name for every modified syntax structure. The `E_Syntax` event code is reported whenever the execution changes into a new syntax construct.

```

1  $include "evdefs.icn"
2  link evinit
3  link syntname # needed for the library primitive syntax()
4
5  procedure main(args)
6      local eventmask, synt_code, synt_name, line
7      EvInit(args)
8      eventmask := cset(E_Line || E_Syntax)
9      while EvGet(eventmask) do{
10         case &eventcode of {
11             E_Line:{
12                 synt_code := keyword("&syntax", Monitored)
13                 synt_name := syntname(synt_code)
14                 write("line # : ", &eventvalue, ", syntax name is :", synt_name)
15             }
16             E_Syntax:{
17                 line := keyword("&line", Monitored)
18                 synt_name := syntax(&eventvalue)
19                 write("syntax change at line # : ", line, " new syntax: ", synt_name)
20             }
21         }
22     }
23 end

```

Figure 6.8. Sample Syntax Monitor

6.4. High-Level Interpreter Stack Navigation

The Unicon language provides some reserved global names prefixed with ampersand (&) called *keywords*. Some keywords are introduced by Alamo for monitoring needs. For example, the keyword `&eventsource` contains a reference to the currently monitored program. Other keywords are used for error reporting and debugging. For example, the keywords `&file`, `&line`, and `&column`, report the currently executed file name, line number, and column number respectively. See Table 6.2 for more monitoring and debugging related keywords [4,5,6]. These keywords can be inserted directly in the source code of the buggy program for debugging with print statements and assertions.

Table 6.2. Unicon's Debugging Related Keywords

#	Keyword	Description
Source Code Related Keywords		
1	<code>&file</code>	Reports the currently executed source file name
2	<code>&line</code>	Reports the currently executed line number
3	<code>&syntax</code>	Reports the currently executed syntax name
Interpreter Stack Level		
4	<code>&level</code>	Reports the current number of procedure frames on the interpreter stack
Memory Allocation Related Keywords		
5	<code>&allocated</code>	Reports the total allocations in heap, static, string, and block regions
6	<code>&regions</code>	Reports the current size of static, string, and block regions
7	<code>&storage</code>	Reports the currently used memory in the static, string, and block regions
Garbage Collection Related		
8	<code>&collections</code>	Reports the number of collections in heap, static, string, and block regions
Monitoring Related Keywords		
9	<code>&eventcode</code>	Reports the code of the last reported execution event
10	<code>&eventvalue</code>	Reports the value of the last reported execution event
11	<code>&eventsource</code>	Denotes to the currently monitored program
Error Reporting Related Keywords		
12	<code>&errornumber</code>	Reports the current runtime error number
13	<code>&errortext</code>	Reports the current runtime error message
14	<code>&errorvalue</code>	Report the current runtime error value

The `keyword()` primitive is used by Alamo to access keywords that belong to the execution state of the target program. This primitive is used to take two parameters: 1) the string name of the keyword that a monitor is hunting for, and 2) the target program's co-expression handle, from which the keyword value is obtained. For example, the current file name can be obtained using `keyword("&file", &eventsourc)`, whereas `keyword("&line", &eventsourc)` is used to obtain the current line number.

Originally, this primitive did not provide the ability to obtain the value of the keywords `&file` and `&line` for procedures that have active frames on the interpreter stack, other than the top one. For example, a source-level debugger requires this feature to facilitate a back tracing mechanism. It is needed to provide connections between the activation records on the stack and the source code location, file name and line number, which initiated each of these records. A mechanism that is supported by the runtime system can avoid a huge monitoring overhead.

The extension mechanism utilizes the level of the activation record on the stack. This level is used to obtain the activation record and read its Interpreter Program Counter (IPC), which can be used to identify the file name and line number using a binary search algorithm, taking advantage of the fact that these tables are already sorted based on their IPCs. This mechanism extends the `keyword()` primitive with a third optional parameter, which is the level number of that procedure on the interpreter stack. This new feature is very useful in traversing the execution stack in UDB's `backtrace` (or `where`) command. The default level is zero, which is the level of the most recent procedure frame currently at the top of the stack.

For example, `keyword("&file", &eventsourc, 10)` returns the file name that contains the call to the 10th outermost activation record on the interpreter stack. Similarly, `keyword("&line", &eventsourc, 10)` looks up the buggy program's call stack, and returns the line number of the statement for which the tenth outer most activation record was instantiated. Figure 6.10 shows a sample Unicon procedure that backtraces the stack. This procedure is called from the debugging tool (monitor program). Figure 6.9 shows a sample output for this procedure.

1	0 # Current location is in procedure DD, test.icn:25
2	1 # procedure DD was called from procedure CC, test.icn:19
3	2 # procedure CC was called from procedure BB, test.icn:12
4	3 # procedure BB was called from procedure AA, test.icn:5
5	4 # procedure AA was called from procedure main, test.icn:29

Figure 6.9. Sample Stack Trace

```

1  procedure backtrace()
2      local frame, level, fname, line, curpname, pname
3
4      frame:=0,
5      level := keyword("&level", Monitored)
6
7      curpname := image(proc(Monitored, 0))
8      fname := keyword("&file", Monitored)
9      line := keyword("&line", Monitored)
10     write(frame||" # Current location is in "|| curpname||", "||fname||":"||line)
11
12     frame += 1
13     while frame < level do{
14         pname := image(proc(Monitored, frame))
15         fname := keyword("&file", Monitored, frame)
16         line := keyword("&line", Monitored, frame)
17         write(frame||" # "||curpname||" was called from "||pname||", at "||fname ||":"|| line)
18         frame += 1
19         curpname := pname
20     }
21 end

```

Figure 6.10. Sample Procedure that Backtraces the Current Stack

6.5. Signal Handling

Signals are interrupts sent to the process by the operating system. Some signals are fatal, while others can be ignored or handled by the process using dedicated signal handlers. In Alamo's thread-based monitoring model, the operating system treats the virtual machine as one process, but at any point in time, only one of the multiple loaded programs is running. This means receiving a signal and handling it depends on which program is holding the execution control when the signal is issued and whether the signal is trapped by the signaled program or not.

If the virtual machine has only one loaded program, then the signal is handled only if that program already has a trap for it. Otherwise, the signal's default action is performed. This was the original design consideration by the Unicon virtual machine and its runtime system. However, a new design is needed for multiple programs running synchronously within the same virtual machine. This new design is based on whether the signaled program is a parent or a child. If a child program received a signal that is not handled or trapped, then this child program is terminated and execution

control is transferred back to its parent. If the parent program is the one who received the unhandled signal, then the signal's default action is performed regardless of its children. If this parent is the root program in the current VM, then the complete process is terminated.

In order for the monitor program to observe signals received by the target program, a new event code `E_Signal` reports whenever a child program receives a signal that is not trapped or handled. The value of this event is the string name of the received signal. For example, Figure 6.11 shows a monitor program using the `E_Signal` event in its event mask. This program prints the string name of the signal whenever it is reported.

```
1  procedure main(arg)
2      local eventmask
3
4      eventmask := cset(E_Signal || E_Pcall)
5      while EvGet(eventmask) do
6          if &eventcode === E_Signal then
7              write("The child program received the signal : ", &eventvalue)
8          end
```

Figure 6.11. Sample Monitor Program Using the `E_Signal` Event

Chapter 7

AlamoDE: The Debugging Framework

AlamoDE is the result of recent extensions to the Alamo framework. AlamoDE provides high level facilities for event-based debugging tools that *observe, inspect, analyze, control, and change* the state and behavior of a buggy program. Its goals include:

1. The ability to write debugging tools at a high level of abstraction,
2. All the usual capabilities of classical debuggers,
3. Support for the creation of advanced debugging features such as automatic debugging and dynamic analysis techniques,
4. The ability to debug special language features such as generators, goal-directed evaluation, and string scanning, and
5. Extensibility that allows different standalone debugging tools to work in concert with each other. It allows one debugging tool to forward events into another tool, whether it is real execution and runtime events or pseudo events.

This debugging framework has been tested and refined within a multi-agent debugging architecture called IDEA presented in Chapter 8, and an extensible source-level debugger called UDB presented in Chapter 9. This chapter introduces various AlamoDE features, some of which are a result of the extensions presented in Chapter 6 while others are original Alamo and Unicon features used within the context of AlamoDE for debugging needs.

7.1. Debugging Events

Originally, Alamo provided a programmer with a wide range of events. Some events are explicitly related to the evaluation of source code expressions, while others are implicit runtime system events. Explicit events include:

1. Execution source code location such as line and column numbers
2. Procedure, built-in function, and operator activities such as call, return, fail, suspend, and resume, and
3. String-scanning activities that include scanning environment creation, and position change.

Implicit events include:

1. Memory allocation activities, which are distinguished based on the target region (there are string, block, and static regions), and based on the size and type of the allocated blocks,
2. Garbage collection activities, which are distinguished based on the region being collected and the program, which has the activity that triggered the collection process,
3. Type conversions performed on parameters to functions and operators, and
4. Virtual machine instructions executed by the Icon virtual machine.

Debugging and visualization serve many common goals. For AlamoDE, the underlying instrumentation was extended with three event types that are needed for debugging. The new events are:

1. `E_Deref` reports when a variable is read (dereferenced). This event is needed to implement watchpoints on specific variables. This event was implemented prior to the state of this dissertation.
2. `E_Signal` reports when a target program receives a signal that is not trapped or handled. See Section 6.5 for the implementation.
3. `E_Syntax` reports when a major syntax construct such as a loop starts or ends. This event was inspired by the needs of automatic debugging systems [12, 110] and required that syntax information be added to the Unicon virtual machine bytecode executable format. See Section 6.3 for more syntax instrumentation details.

Figure 7.1 shows a sample debugging loop where `EvGet()` is used inside the `while` condition. This function keeps activating the monitored program reporting the next available event until the monitored program is terminated. In this `while` loop, each reported event is filtered. The `E_Line` event is used for implementing breakpoints and single stepping. It is reported whenever the execution jumps to a new line number in the actual source code of the monitored program. `E_Assign` event is reported whenever a variable is assigned. This event code is always followed by the `E_Value` event that represents the assigned value. `E_Deref` event is reported whenever a variable is being read, `E_Spos` and `E_Snew` relate to the string scanning environment. And finally `E_Error` and `E_Exit` are reported whenever the target program is terminated. `E_Error` shows that a runtime error caused the termination whereas `E_Exit` shows a normal program termination.

```

1  # Template of an AlamoDE event-based debugging loop
2  EvInit("buggy program name and its arguments")
3  while event := EvGet(eventmask, valuemask) do {
4      case event of {
5          E_Line           : { } # handle breakpoints, stepping, etc
6          E_Assign | E_Value : { } # Handle assignment watchpoints
7          E_Deref          : { } # Handle read watchpoints
8          E_Spos  | E_Snew  : { } # Handle string scanning environments
9          E_Error          : { } # Handle a runtime error
10         E_Exit           : { } # Handle buggy program normal exit
11     }
12     # Handle other debugging features such as tracing,
13     # profiling and internal and external debugging tools
14 }

```

Figure 7.1. Sample AlamoDE Debugging Loop

7.2. Event Filtering

Considering the many millions of events produced by AlamoDE's detailed VM instrumentation, which provides 121 kinds of events, an efficient filtering mechanism is needed to reduce the monitoring time. Alamo originally used a simple bit vector called an event mask to specify event types of interest. Later and before the start of this dissertation, the filtering was extended so that each event type of interest could have an associated *value mask*, a set of event values of interest, which further restricts whether an event is reported. Instrumentation in the virtual machine checks for execution events in two levels. First, it checks whether the encountered runtime event is part of the event mask. Then, it checks if there is a value mask associated with this kind of event. If so, only those events that have values in the value mask are reported. See Figure 7.2.

The dynamicity of event mask and value mask allow a debugging tool to change and customize the monitored events on the fly during the course of execution; any change on either of the two masks will immediately change the set of reported events. For example, placing a breakpoint on one or more line numbers requires the `E_Line` event to be a member of the event mask. The value mask provides the ability to limit the reported `E_Line` events to those line numbers that have breakpoints on them. To clear a breakpoint, a tool removes the line number from the value mask. The `E_Line` event is removed from the event mask only if there are no more breakpoints and no other requests for `E_Line` events by the debugging tool or any of its cooperative tools. Figure 7.3 shows a monitor program that asks the user for a line number that is to be monitored during the execution of the target program. It uses the value mask table with the `E_Line` key.

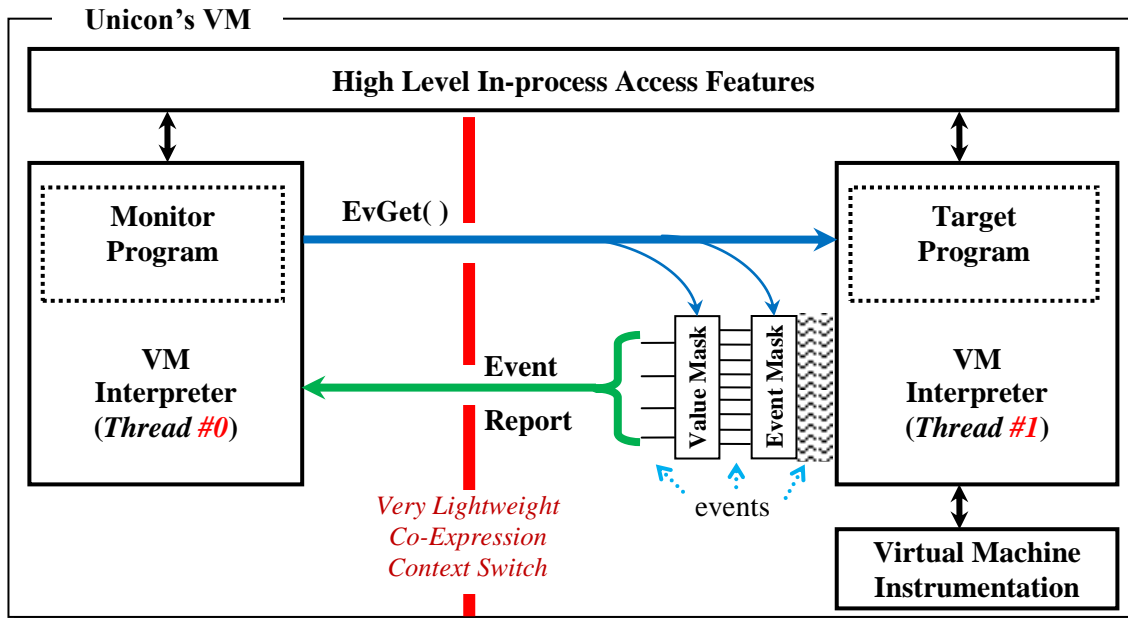


Figure 7.2. AlamoDE's Architecture

7.3. Execution State Inspection and Modification

AlamoDE provides facilities to inspect the execution stack, check a variable state, and acquire information about the source code of the buggy program. It allows the monitor to control and change the state of the buggy program by assigning to variables and redirecting procedures and functions. The target program's execution state and data are accessible by the monitor program. Alamo provides two kinds of features that allow a monitor program to inquire about the execution state of a target program. First, events that are reported based on different actions during the evaluation of the target program. Second, a monitor program is always able to look up further information about the target program's state of execution using high level primitives.

7.3.1. Variables

Alamo provides several built-in functions for execution monitors. Monitor programs use these primitives for further investigation of the target program execution state. A variable is either global, or local including static and parameter variables. Variable names can be obtained using dedicated primitives such as `globalnames()`, `localnames()`, `paramnames()`, and `staticnames()` [4]. A local variable value can be obtained using the built-in function `variable(name, &eventsources, level)`, which returns the current value of the variable `name` in the frame number `level` of the buggy program's call stack. If `name` is a global variable or a keyword, the same function is used without the `level` parameter (i.e. `variable(name, &eventsources)`).

```

1  $include "evdefs.icn"
2  link evinit
3
4  procedure main (args)
5      local fname, eventmask, valuemask, line, ans, flag
6
7      EvInit(args) | stop(" *** cannot initialize monitored program *** ")
8      eventmask:= cset(E_Line)
9      valuemask:= table()
10
11     write("Please enter a line number you want the execution to stop at:")
12     line := integer(read())
13     valuemask[ E_Line ] := set(line)
14
15     while EvGet(eventmask, valuemask) do {
16         fname := keyword("&file", Monitored)
17         write(" ==> reaching line number || &eventvalue || " in file ", fname)
18         write("would you like to stop at another line (Y/n):")
19         if /flag & (*(ans:=read())=0 | not(ans[1] == ("n"|"N"))) then {
20             write("Please enter a line number :")
21             line := integer(read())
22             insert(valuemask[E_Line], line)
23         }
24         else
25             flag := 1
26     }
27 end

```

Figure 7.3. Sample Monitor Using the *event mask* and *value mask*

The `variable()` function is also used by debugging tools that modify the target program's state by assigning to variables in the buggy program. When it is used to assign a new value to a variable in the target program, a fourth parameter is used as a flag, see Figure 7.4. This flag is an integer value introduced by the implementation of *inter-program variable safety* that is presented in Section 6.2. When this flag is present with a value other than zero, it allows the monitor program to safely assign to a variable in the monitored program. Otherwise, the assignment is ignored. This behavior prevents the monitor program from modifying target program execution properties that are not valid. If a context switch occurred between the time the variable reference is obtained and the time the assignment is complete, this assignment will produce a runtime error and terminate the execution.

```
variable(name, &eventsourc, level, flag) := value
```

Figure 7.4. Assigning Variables in the Buggy Program

7.3.2. Procedures and Stack Frames

Activation records (frames) on the stack are distinguished by a positive integer called *level*; the most recent stack frame is at level zero, whereas the highest level value is for the activation record of procedure `main()`. The `proc()` built-in function was extended for AlamoDE to allow the debugging tool to identify which procedure is currently active on a specific stack level. For example, `proc(&eventsourc,7)` returns a pointer to the procedure or method, which lives on the seventh outer most level of the buggy program's call stack. The depth of the call stack can be checked using the keyword `&level`. The keyword `("&level", &eventsourc)` returns the number of frames currently on the buggy program's interpreter stack.

Furthermore, the Unicon language allows programmers to replace a procedure with another procedure during the execution. This feature is very useful for some debugging tools. For example, if the buggy program contains two versions of a sorting algorithm, in different procedures, the debugger can replace one by the other on the fly during the execution.

The procedure or method pointer obtained by the `proc()` function allows a debugging tool to place a call to that procedure as an *inter-program procedure call*. This mechanism is very useful for interactive source-level debuggers. For example, the buggy program may contain a procedure that prints the elements of a linked list, which is being debugged by the user. The debugger can place a call to that procedure, from any point during the debugging session, without modifying the buggy program source code. Moreover, a source-level debugger may incorporate general service procedures that can be plugged in to the buggy program source code on the fly during the debugging session.

For example, the assignment in Figure 7.5 replaces the buggy program's procedure `sort1()` with the debugger service procedure `qsort()`. Of course, the two procedures' formal parameters must be compatible in their order and type.

```
variable("sort1", &eventsourc) := proc("qsort", &current)
```

Figure 7.5. Modifying Procedures in the Buggy Program

7.3.3. Executed Source Code

Unicon's executable bytecode contains information about the linked source files including any used library modules. For AlamoDE, a class library was developed to analyze the bytecode and generate a list of its source file names, and their static source code properties such as packages, classes, global variables, and user defined functions. Another class library maintains a list of all source files in use. Those library classes provide a debugging tool with the buggy program's source code static information. Furthermore, the debugging tool can inspect the currently executed source code using runtime events and high level functions such as the `keyword()` function discussed in Section 7.3.2. For example, the `E_Line` and `E_Syntax` events report the currently executed line number, and source code syntax construct respectively.

7.4. Advanced Debugging Support

AlamoDE provides underlying infrastructure for automatic debugging, dynamic analysis, profiling, and visualization.

7.4.1. Multitasking

AlamoDE provides a multitasking mechanism for various debugging tools and techniques. A debugging tool runs as the main co-expression inside the virtual machine. A buggy program and secondary standalone debugging tools can be loaded into different co-expressions controlled by the debugger. A debugger transfers control to the buggy program using the `EvGet()` function. Then the buggy program executes until there is some event that is of interest to the debugger. `EvGet()` requests the next event by resuming the program that is denoted by `&eventsource`. An AlamoDE-based debugging tool can debug multiple buggy programs in one session. This can be used to perform advanced debugging techniques such as relative debugging [54] or delta debugging [21]. Switching between different programs is accomplished by changing the value of `&eventsource` before the next call to `EvGet()`.

7.4.2. Event Forwarding

A monitor coordinator allows different debugging tools to work in concert during the same monitoring activity, playing the role of a central server for other debugging tools. The debugger and its loaded tools work synchronously on the same buggy program. One debugging tool can use the `EvSend()` primitive to forward an event to another tool running on the same virtual machine. This primitive forwards an event code and its corresponding value into another tool. The forwarded event

is received by the other debugging tool as it requests the events using the typical `EvGet()` primitive. Moreover, this forwarded event code and value do not need to be any real event reported from the target program. Sometimes, this primitive is used to provide other monitoring tools with artificial or pseudo events, which can be used as a communication protocol between various debugging tools.

Additionally, the primitive `eventmask()` provides the ability for a monitoring tool to read or modify the set of events requested by another monitoring tool. This feature is important for performance reasons. The main debugging tool, which is also called the debugging coordinator, can utilize monitoring information from secondary tools to optimize the number of monitored events applied on the target program. Furthermore, this main debugging tool needs to know the kinds of events that are requested by each of the secondary tools in order to forward them.

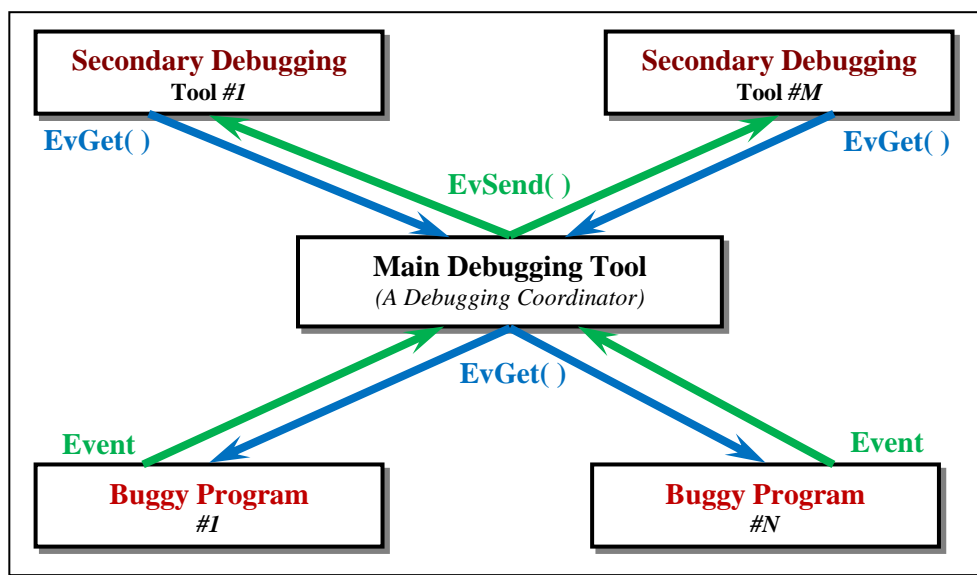


Figure 7.6. AlamoDE Debugging Capabilities

7.4.3. Custom Defined Debugging Tools

AlamoDE puts execution events in the hands of programmers, who can use events, event sequences, and event patterns to write their own automated debugging and dynamic analysis tools. For example, the code in Figure 7.7 shows a toy example of an AlamoDE-based debugging tool. It captures the number of garbage collections that happen during the execution of a buggy program, and finds the total and average of collected data from the string and block regions. This provides a rough measure of whether the buggy program is mostly doing string processing or not. This example program can be used as a standalone tool, or loaded into another debugging tool on the fly without any source code modification at all.

```

1  $include "evdefs.icn"
2  link evinit
3  class Example (
4      eventMask, gc, lastStr, lastBlk, collectedStr, collectedBlk, avgStr, avgBlk
5  )
6  method handle_E_Collect()
7      local Storage := [ ]
8      gc += 1
9      every put(Storage, keyword("storage", Monitored))
10     lastStr := Storage[2]; lastBlk := Storage[3]
11 end
12 method handle_E_EndCollect()
13     local Storage := [ ]
14     every put(Storage, keyword("storage", Monitored))
15     collectedStr += lastStr - Storage[2]; collectedBlk += lastBlk - Storage[3]
16 end
17 method analyze_data()
18     if gc = 0 then return 0
19     avgStr := collectedStr / gc; avgBlk := collectedBlk / gc
20 end
21 method write_data()
22     write(" # Garbage Collections : ", gc)
23     write(" Collected Strings : ", collectedStr, " Avg : ", avgStr)
24     write(" Collected Blocks : ", collectedBlk, " Avg : ", avgBlk)
25 end
26 initially()
27     eventMask := cset(E_Collect || E_EndCollect)
28     gc := 0; collectedStr := collectedBlk := 0.0
29 end
30 procedure main(arg)
31     EvInit(arg)
32     obj := Example()
33     while event := EvGet(obj.eventMask) do {
34         case event of {
35             E_Collect:    { obj.handle_E_Collect()    }
36             E_EndCollect: { obj.handle_E_EndCollect() }
37         }
38     }
39     obj.analyze_data()
40     obj.write_data()
41 end

```

Figure 7.7. An AlamoDE Debugging Agent

Part III

Very High Level Extension Mechanism

Chapter 8

IDEA: A Debugging Extension Architecture

This chapter presents the Idaho Debugging Extension Architecture (IDEA). IDEA's extensions are called *agents*. IDEA's agents are event-driven task-oriented program execution monitors that embody lightweight dynamic analyses. IDEA's agents are written and tested as standalone programs, after which they can be loaded and used on the fly from within the IDEA-based source-level debugger (*external agents*), or integrated as permanent features into the debugging core (*internal agents*). The IDEA-based source-level debugger provides a simple interface to load, unload, enable, or disable debugging agents on the fly, and the user can be selective about where, when, and which agent(s) to use. This chapter is based on material from [120].

8.1. Debugging with Agents

Conventional debuggers allow users to explore their debugging hypotheses using manual investigation. Debugging with agents leverages the conventional debugging process by empowering the user with more tools to inspect the state of the buggy program. For example, many functions return a specific value when they encounter an error or fail to accomplish their job. An agent can automatically catch any of these failed functions and save the user the time that can be spent during a manual inspection. In general, IDEA's agents may retain information beyond the current state of execution and perform automatic debugging and dynamic analysis techniques that could be supported by trace-based debuggers such as ODB [47, 48]. Using IDEA, it is easy to incorporate debugging agents that capture specific execution behaviors such as:

1. Loops that iterate N times, for some $N \geq 0$
2. Variables that are read and never assigned or assigned and never read during a particular execution
3. Expressions such as subscripts that fail silently in a context where failure is not being checked
4. Variables that change their type during the course of execution
5. A trace of variable states, which allows users to trace backward and see where a specific variable was assigned long before it is involved in a crash

8.2. Design

IDEA features novel properties that distinguish it from other debugging architectures. First, it provides two types of extensions: *dynamic extension* on the fly during the debugging session and *static extension* supported by formal steps to migrate and adopt standalone agents as permanent debugging features—statically linked into the source code of the debugger. Second, it simplifies the extensibility of a source-level debugger, which provides an interactive user interface. This interface allows simultaneous agents to be loaded and managed during a debugging session. Finally, this simple extensibility may encourage users to write their own agents and incorporate them into a typical source-level debugging session.

IDEA's agents are able to analyze execution properties and behaviors based on runtime information, which they collect while they sit enabled in the background of the debugging session. Sometimes, the user may limit an agent to a specific part of the execution by manually enabling and disabling it between different execution points. Other times, the agent is programmed to automatically trigger information gathering based on some specific runtime properties. For example, an agent can automatically watch all **while** loops or just the one within a specific procedure or method.

In general, unless an agent depends on information prior to its load time, the user does not need to rerun the program whenever a decision is made to incorporate the agent into the debugging session. In contrast, most static and dynamic analysis tools and libraries must be linked in advance into the source code of the buggy program, or initialized at the start of the host debugger. Moreover, often these static and dynamic analysis tools provide no means for the user to control the part of the program or the execution interval where the information should be collected or analyzed.

8.3. Implementation

IDEA extends the debugging core of a source-level debugger with two major components:

1. An evaluator that provides the main event filtering and forwarding mechanism, and
2. An agent interface that facilitates and provides the programming interface for external and internal extensions.

IDEA's evaluator is comprised of two components that make the source-level debugger an event coordinator for the debugging agents; **Internals** and **Externals**. These components are abstracted by objects, which serve as **Proxies** for external agents or as **Listeners** in the case of internal extensions. These objects are plugged in to the main debugging loop as extra listeners on the runtime events.

IDEA manages and coordinates all extension agents and forwards received events from the buggy program into active agents based on their interest. Each agent:

1. Provides the evaluator with its set of desired events in the form of an event mask
2. Receives relevant events from the evaluator
3. Performs its debugging mission, which may utilize execution history prior to the current execution state, and
4. Presents its analysis results back to the user.

Figure 8.1 shows IDEA's architecture. When the evaluator receives an event from the buggy program, it forwards the received event to those agents that are enabled and requested this 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 context switch, which the agent sees as a return from its `EvGet()` function. `EvGet()` is the AlamoDE primitive that resumes the buggy program until the next available event. In case of external agents, `EvGet()` resumes IDEA's evaluator.

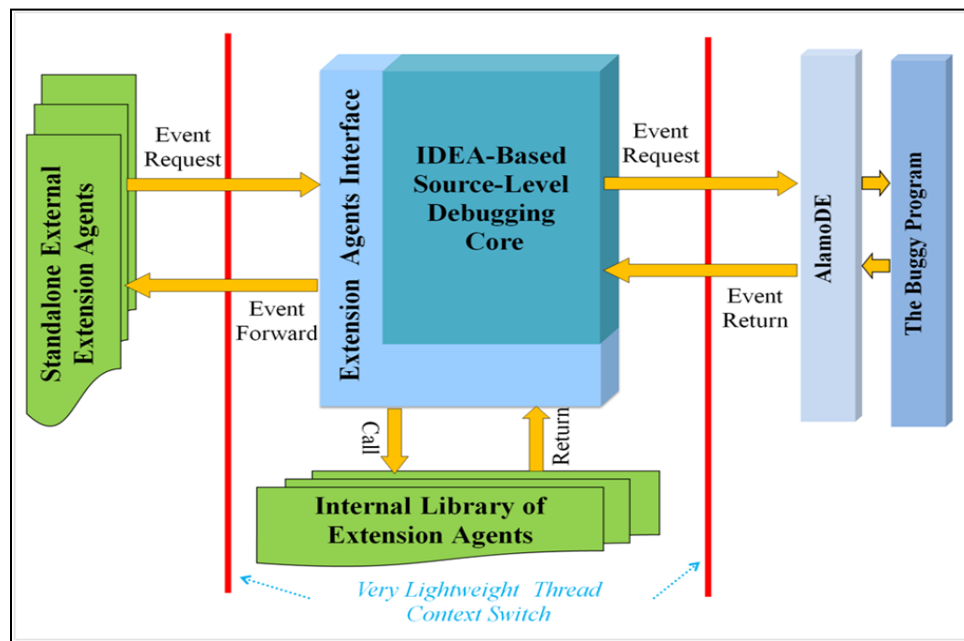


Figure 8.1. IDEA's Architecture

8.4. Source Code

IDEA's debugging core is comprised of five basic classes. One class is general for all extension agents, two classes are dedicated for external extensions, and the other two classes are dedicated for internal extensions. Figure 8.2 shows IDEA's UML diagram.

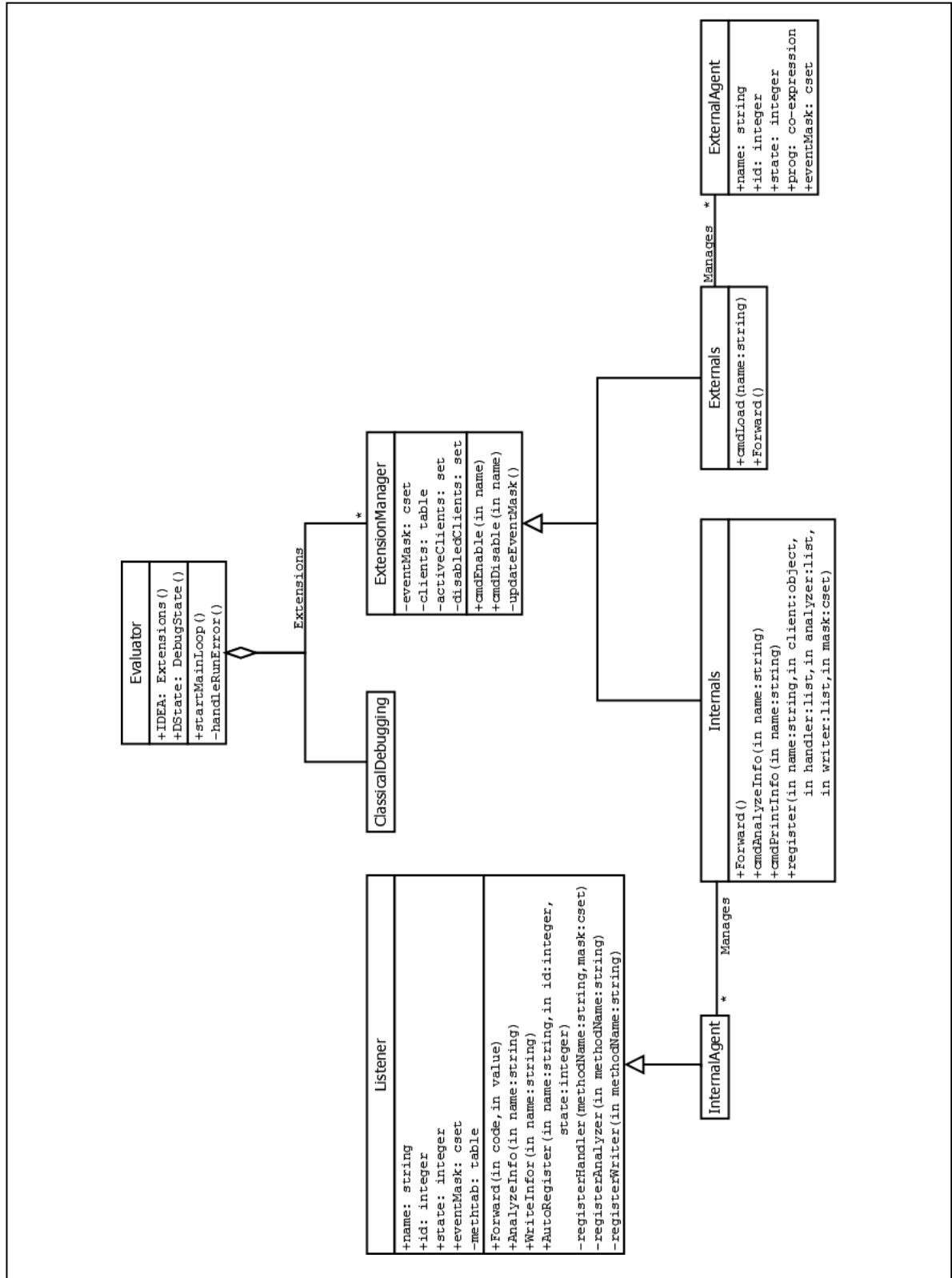


Figure 8.2. IDEA's UML Diagram

1. **Agent** class handles all extension agents' basic features such as enabling, disabling, and constructing their event mask. It provides public methods such as `disableAgent(name)` that disables an agent, `enableAgent(name)` that enables an agent, and `updateMask()` that checks and updates the event mask of the target debugger whenever an agent is enabled or disabled.
2. **Externals** class handles the separately-compiled dynamically-loaded external debugging tools that are loaded on the fly. It provides two public methods. The method `cmdLoad()` loads an external agents' executable and registers it on the fly under its name (the name of the executable). The method `Forward()` checks all active external agents and forwards the received event to those that acquire this event in their mask.
3. **ExternalClient** class handles information about external agents. Each of the currently loaded agents has its own object, which is saved into the active clients list.
4. **Internals** class handles the debugging tools that have migrated to internals. It provides the `Forward()` method just like the **Externals** class. It also provides the `register()` method that is used to manually register agents as internal debugging features.
5. **Listener** class handles the entire migrated agents interface. An external agent must be subclassed from this **Listener** class before it can be registered as internal built-in feature. This class automatically analyzes and registers the agent's features.

8.5. Extensions

Different agents can be loaded and active, and each agent receives different runtime events based on their own event mask. For every received event, IDEA's evaluator checks for any enabled internal and external agent; it forwards events to the enabled ones based on their event mask. Newly activated agents start receiving relevant events right after their activation. Disabled agents receive no events until they are enabled explicitly by the user. An extension agent may change its event mask during the course of execution. A change on any agent's event mask immediately triggers an update to the event mask of the debugging core and alters the set of events received by the debugging core and forwarded to the agents.

8.5.1. Sample Agent

The code provided in Figure 8.3 shows an example IDEA-based 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 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 8.5.4. In contrast, external agents require no special formatting and no pre-registration.

```

1  $include "evdefs.icn"
2  link evinit
3  class Example(
4      eventMask, pcalls, fcalls, prate, frate )
5      method handle_E_Pcall( )
6          pcalls += 1
7      end
8      method handle_E_Fcall( )
9          fcalls += 1
10     end
11     method analyze_info( )
12         total := real(pcalls + fcalls)
13         prate := pcalls / total * 100
14         frate := fcalls / total * 100
15     end
16     method write_info( )
17         write(" # pcalls = ", pcalls, " at rate :", prate)
18         write(" # fcalls = ", fcalls, " at ratio :", frate)
19     end
20     initially()
21         eventMask := cset(E_Pcall || E_Fcall)
22         pcalls := fcalls := 0
23     end
24     procedure main(args)
25         EvInit(args)
26         obj := Example()
27         while EvGet(obj.eventMask) do
28             case &eventcode of {
29                 E_Pcall:{ obj.handle_E_Pcall() }
30                 E_Fcall:{ obj.handle_E_Fcall() }
31             }
32             obj.analyze_info()
33             obj.write_info()
34     end

```

Type 1: Event Handler #1

Type 1: Event Handler #2

Type 2: Information Analyzer

Type 3: Information Writer

Figure 8.3. An IDEA-based Agent Prototype

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()`).

8.5.2. External Agents

External agents can be written and tested as standalone tools, and subsequently loaded on the fly and used together during a debugging session—no special registration and no pre-initialization is needed. External extensions are managed by three classes: `Agent`, `Externals`, and `ExternalClient`. IDEA's external agents are loaded and controlled by its debugging core. Separately-compiled dynamically-loaded external agents receive their information from IDEA's evaluator, which controls them. The external debugging agents' standard inputs and outputs are redirected and coordinated by IDEA's debugging core. IDEA's evaluator receives runtime events from the buggy program based on the current debugging context, which includes the event masks of enabled external agents. The `Externals` component multiplexes the received events between different external agents. Events are forwarded to related active agents.

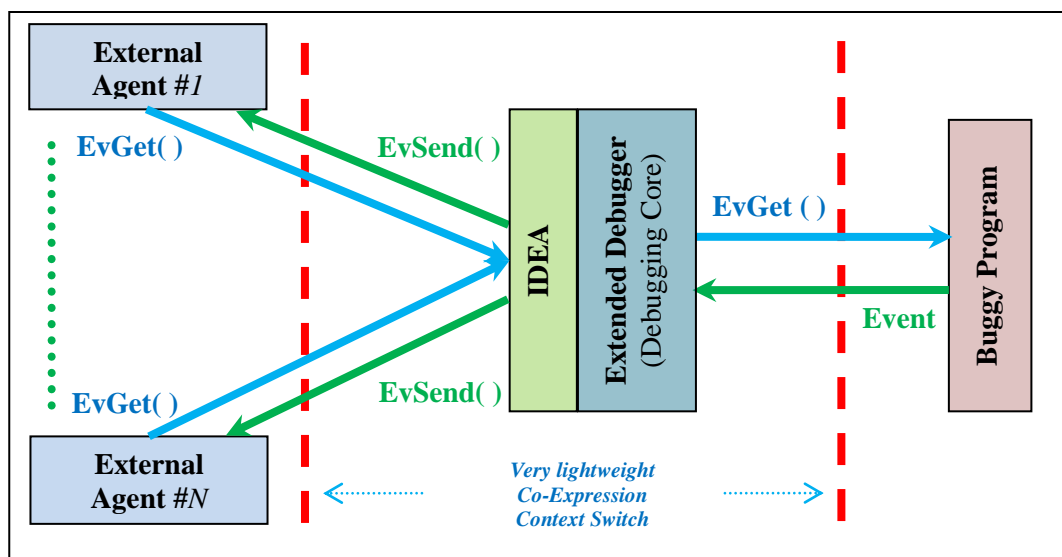


Figure 8.4. IDEA's on-the-fly Extensions (*External Agents*)

An external agent requests events from the IDEA evaluator using the `EvGet()` primitive, which transfers control from the external agent to the extended debugger, see Figure 8.4 above. `EvGet()` is the same primitive that transfers control and acquires events from the buggy program when the agent is used in a standalone mode. The `Externals` component forwards events to any of the external agents using the `EvSend()` primitive, which is used to send the last event received by the evaluator to the external agent. 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 itself, which means the external agent needs no modification to be loaded and used by IDEA's core.

8.5.3. Internal Agents

Besides support for whole programs as external agents, IDEA supports insertion of dynamic analyses into the debugging core as listener agents that implement a set of callback methods. IDEA's debugging core implements different built-in agents for different classes of bugs. For performance reasons, each agent has its own implementation based on the kind of events that the debugging core must monitor in the buggy program. The `Internals` component handles the built-in agents. Internal agents are called from the main debugging loop with a call to the `Forward()` method of the `Internals` component, where internal agents are registered during initialization. The `Internals` component checks which agents are active and calls the related underlying method(s) based on the event code that is received by the debugging core. See Figure 8.5.

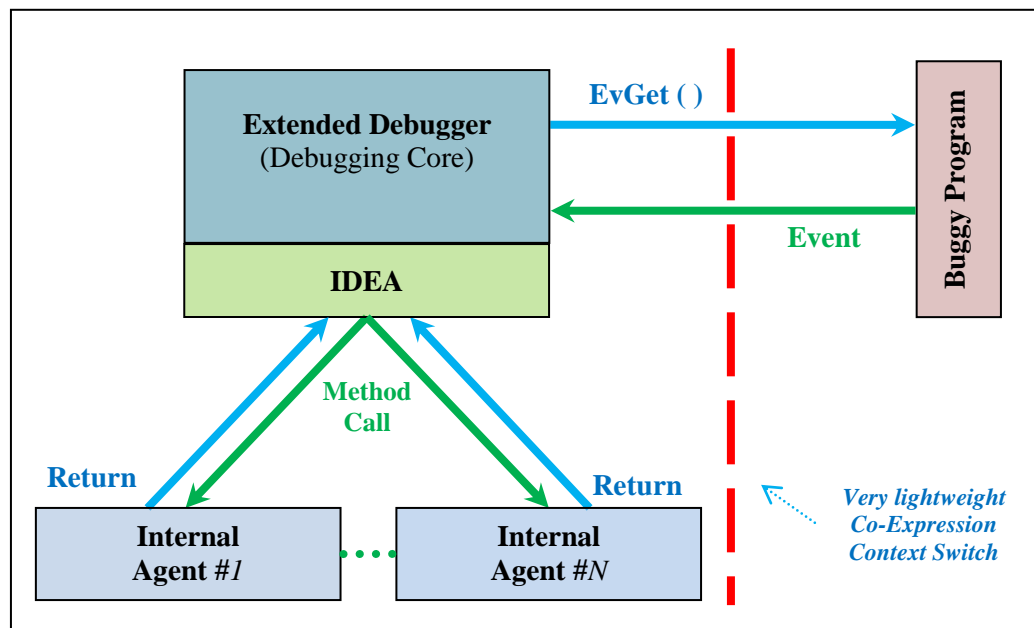


Figure 8.5. IDEA's Internal Extensions (*Internal Agents*)

8.5.4. 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—built-in monitors within the debugging core for improved performance. Internal agents do not pay the (lightweight, but still painful) cost of the context-switch communication between the debugging core and the external agents. IDEA 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. IDEA provides an abstract class called `Listener`, which must be subclassed within the agent before the external can become an 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 event types.

The agent prototype discussed in Section 8.5.1 and Figure 8.3 can be used as a standalone program or as an external agent under IDEA without any modification. In order to move such an external agent to an internal one, the user must derive this `Example` class from IDEA's `Listener` abstract class and register it in the `Internals` class. Whenever its own event mask changes, this abstract class helps the `Internals` class rebuild the event mask for the internal agents and the debugging core using the `updateMask()` method in IDEA's `Agent` class. This method updates the extended debugger with the new event mask obtained from the internal agent.

An object of the newly migrated internal agent must be instantiated and inserted into the list of clients in the `Internals` class. This can be done through the method `register()` from the `Internals` class. For example, to register the prototype `Example` agent in Figure 8.3 as an internal agent, the programmer has to place a call to the method `register()` in the `Init()` method of the `Internals` class where the first parameter associates the agent with a formal name as a string ID during the debugging session, and the second parameter is an object of that agent class (i.e. `register("calls", Example())`).

This is the simple automatic registration that applies for agents who follow the sample agent convention shown in Figure 8.3 and discussed in Section 8.5.1. To register a complex agent that does not follow this sample convention, the method `register()` can be called with four extra parameters to register the method handlers, the analyzers, and the writers respectively along with agent event mask.

The new internal agent must be stripped of its `main()` procedure before it is linked into the debugging core. `Alamo's EvInit()` and `EvGet()` are no longer needed as it is already performed by the debugging core. The mapping of events such as `E_Pcall` to their listener methods (`handle_E_Pcall`)

is constructed automatically when the `Example()` class is subclassed derived from the `Listener` class provided by IDEA, see Figure 8.6.

<pre> 1 \$include "evdefs.icn" 2 link evinit 3 4 class Example (5 eventMask, pcalls, fcalls, prate, frate 6) 7 method handle_E_Pcall() 8 pcalls += 1 9 end 10 method handle_E_Fcall() 11 fcalls += 1 12 end 13 method analyze_info() 14 total := pcalls + fcalls 15 prate := pcalls / total * 100 16 frate := fcalls / total * 100 17 end 18 method write_info() 19 write(" # pcalls = ", pcalls, " at rate :", prate) 20 write(" # fcalls = ", fcalls, " at ratio :", frate) 21 end 22 initially() 23 eventMask := cset(E_Pcall E_Fcall) 24 pcalls := fcalls := 0.0 25 end 26 27 procedure main(args) 28 EvInit(args) 29 obj := Example() 30 while EvGet(obj.eventMask) do 31 case &eventcode of { 32 E_Pcall:{ obj.handle_E_Pcall() } 33 E_Fcall:{ obj.handle_E_Fcall() } 34 } 35 obj.analyze_info(); obj.write_info() 36 end </pre>	<pre> 1 \$include "evdefs.icn" 2 link evinit 3 4 class Example : Listener (5 eventMask, pcalls, fcalls, prate, frate 6) 7 method handle_E_Pcall() 8 pcalls += 1 9 end 10 method handle_E_Fcall() 11 fcalls += 1 12 end 13 method analyze_info() 14 total := pcalls + fcalls 15 prate := pcalls / total * 100 16 frate := fcalls / total * 100 17 end 18 method write_info() 19 write(" # pcalls = ", pcalls, " at rate :", prate) 20 write(" # fcalls = ", fcalls, " at ratio :", frate) 21 end 22 initially() 23 eventMask := cset(E_Pcall E_Fcall) 24 pcalls := fcalls := 0.0 25 end 26 27 procedure main(args) 28 EvInit(args) 29 obj := Example() 30 while EvGet(obj.eventMask) do 31 case &eventcode of { 32 E_Pcall:{ obj.handle_E_Pcall() } 33 E_Fcall:{ obj.handle_E_Fcall() } 34 } 35 obj.analyze_info(); obj.write_info() 36 end</pre>
A. Standalone External Agent	B. Migrated to Internal Agent

Figure 8.6. Sample Migrated Agent

8.5.5. Simple Agent Migration Example

Figure 8.3 showed a simple IDEA-based extension agent. Figure 8.6 shows the migration of that agent from standalone program to IDEA internal agent. Each monitored event is mapped, in a one-to-

one relation, into a single method. This conventional format allows the IDEA-based 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 the IDEA's architecture. 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 extended 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 Internal 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.

8.4.6. Complex Agent Migration Example

Complex agents do not follow the naming convention discussed in Section 8.4.1. 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 user has to 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. Figure 8.7 shows the call to the register() method that manually registers the Example class as shown in Figure 8.3.

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

Figure 8.7. Explicit Agent Registration

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.

Chapter 9

UDB: The Unicon Source-Level Debugger

This chapter presents the design and implementation of UDB [121,122], a source-level debugger for the Unicon [3, 4] and Icon [6, 7] programming languages. UDB is an event-driven agent-oriented extensible source-level debugger. It is written in Unicon on top of the AlamoDE debugging framework presented in Chapter 7, and the IDEA architecture presented in Chapter 8. UDB combines classical debugging techniques such as those found in GDB with a growing set of extension agents. Unlike ordinary debuggers, which are usually limited in the amount of analysis that they perform in order to assist with debugging, UDB's design and implementation proves three hypotheses:

1. A source-level debugger built on top of an event-driven debugging framework can surpass ordinary debuggers with more debugging capabilities
2. A debugger based on a high-level framework allows an easy and 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.

This chapter is based on material presented in [121, 122].

9.1. UDB's Debugging Features

UDB provides typical debugging techniques such as breakpoints, watchpoints, single stepping and continuing, and stack navigation. At the same time, it has a rich set of advanced debugging features. The underlying event-driven architecture empowers UDB with advanced debugging techniques. First, it features more powerful watchpoints that support advanced language features such as dynamic typing and string scanning. Second, it provides tracepoints that allow the ability to trace specific execution behaviors of procedures, built-in functions, and language operators. Finally, it supports outstanding extensibility provided by the IDEA architecture. This allows experienced users to 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 debugging features.

9.2. Design

UDB employs AlamoDE's thread model of execution monitoring, where the debugger and its buggy program are in separate threads in a shared address space. The IDEA architecture allows UDB to provide advanced debugging features through multiple simultaneous agents. As it is stated in Chapter 8, UDB's extension agents are written and tested as standalone tools and then loaded and managed on the fly by its IDEA architecture during a typical debugging session. Successful external agents may be promoted to internal built-in features within the debugging core for improved performance. Agents are suspended whenever a breakpoint or watchpoint is reached, and they are resumed when the buggy program is resumed. UDB provides smooth migration from external agents to internals. See Section 8.4 for more details on the migration procedure.

```

1  $ udb sort
2      UDB Version 1.5, January 2009.
3      sort : loaded 2.5K bytes of 32-bit uncompressed icode
4      1 Source file(s) found
5      Type "help" for assistance
6  (udb) break BubbleSort
7      Breakpoint set successfully in:
8      1# sort.icn(5): BubbleSort( A )
9  (udb) run
10     A =[4,1,8,9,0,6,5,7,2,3]
11     Breakpoint:    sort.icn(5): BubbleSort( A )
12  (udb) enable -agent failedloop
13     The agent failedloop is enabled
14  (udb) cont
15     loop: failed while
16     sort.icn(10): while swapped ~== "true" do{
17  (udb) quit
18     sort is running, are you sure you want to quit,(Y/n)?:y
19     Thank you for using UDB, Goodbye !
20  $

```

Figure 9.1. Sample UDB Debugging Session

By design, most of UDB's commands resemble those of GDB. This provides familiarity and ease of use for programmers who switch between languages frequently. In addition to GDB's command set, UDB adds a handful of simple but general commands that load, unload, enable, and disable its extension agents. This simplifies the extensibility, especially for typical users and novice programmers who may want to benefit from existing agents.

Figure 9.1 shows a sample UDB debugging session. The target program sorts an array of integers using the bubble sort algorithm, which uses a `while` loop. Line 14 in the figure enables the internal agent named `loop` to watch for `while` loops that iterate zero times. Then lines 17 and 18 show the `loop` agent printed a message about a failed `while` loop detected at line 10 of the target program's source file named `sort.icn`.

9.3. Debugging Core

UDB's debugging core provides the main debugging features and user interface. It is comprised of four major components: 1) a console, 2) a debugging session, 3) an evaluator, and 4) a debugging state. UDB's debugging core manages all of the built-in classical debugging techniques, and coordinates the operations between the extension architecture, the buggy program, and the user interaction. Figure 9.2 shows UDB's architecture and Figure 9.3 shows UDB's UML diagram.

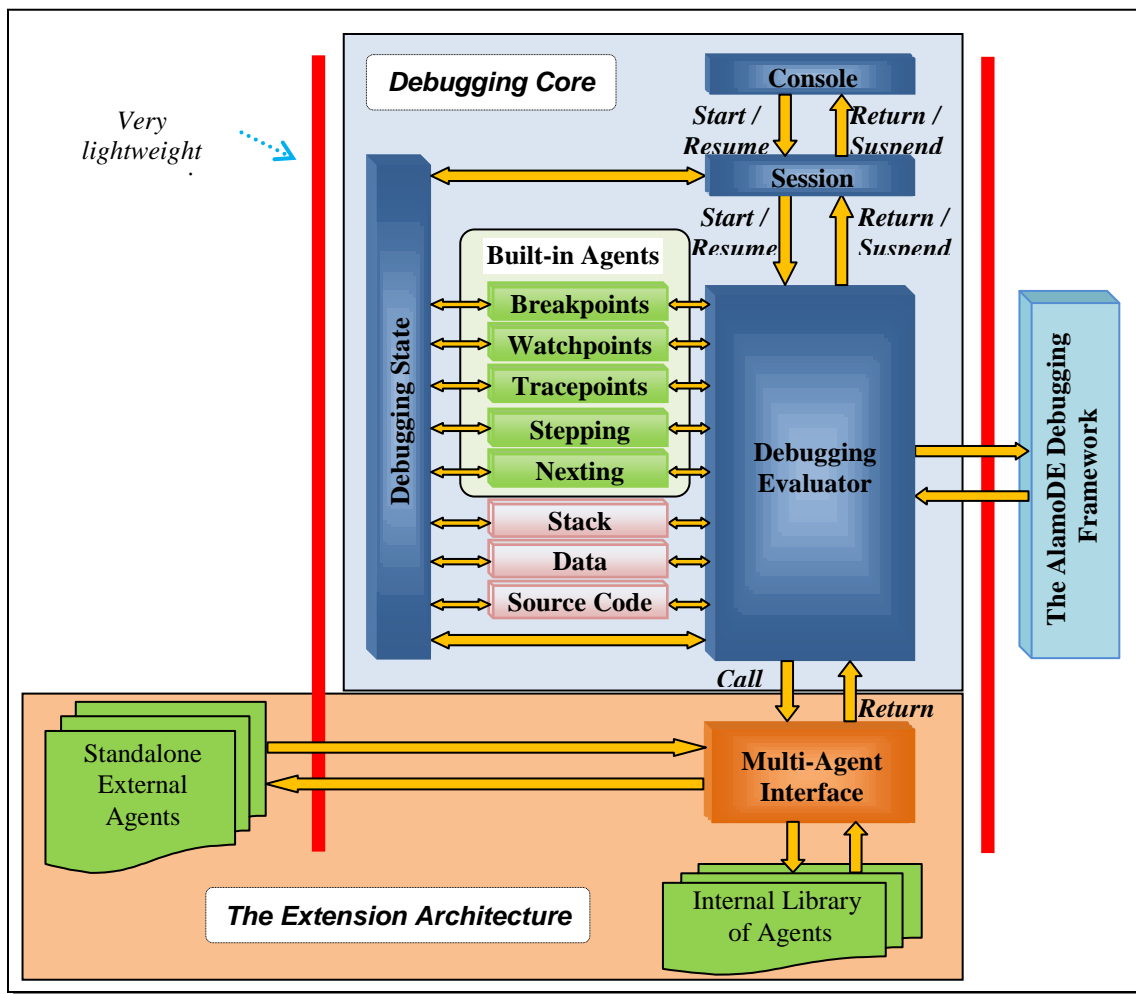


Figure 9.2. UDB's Debugging Architecture

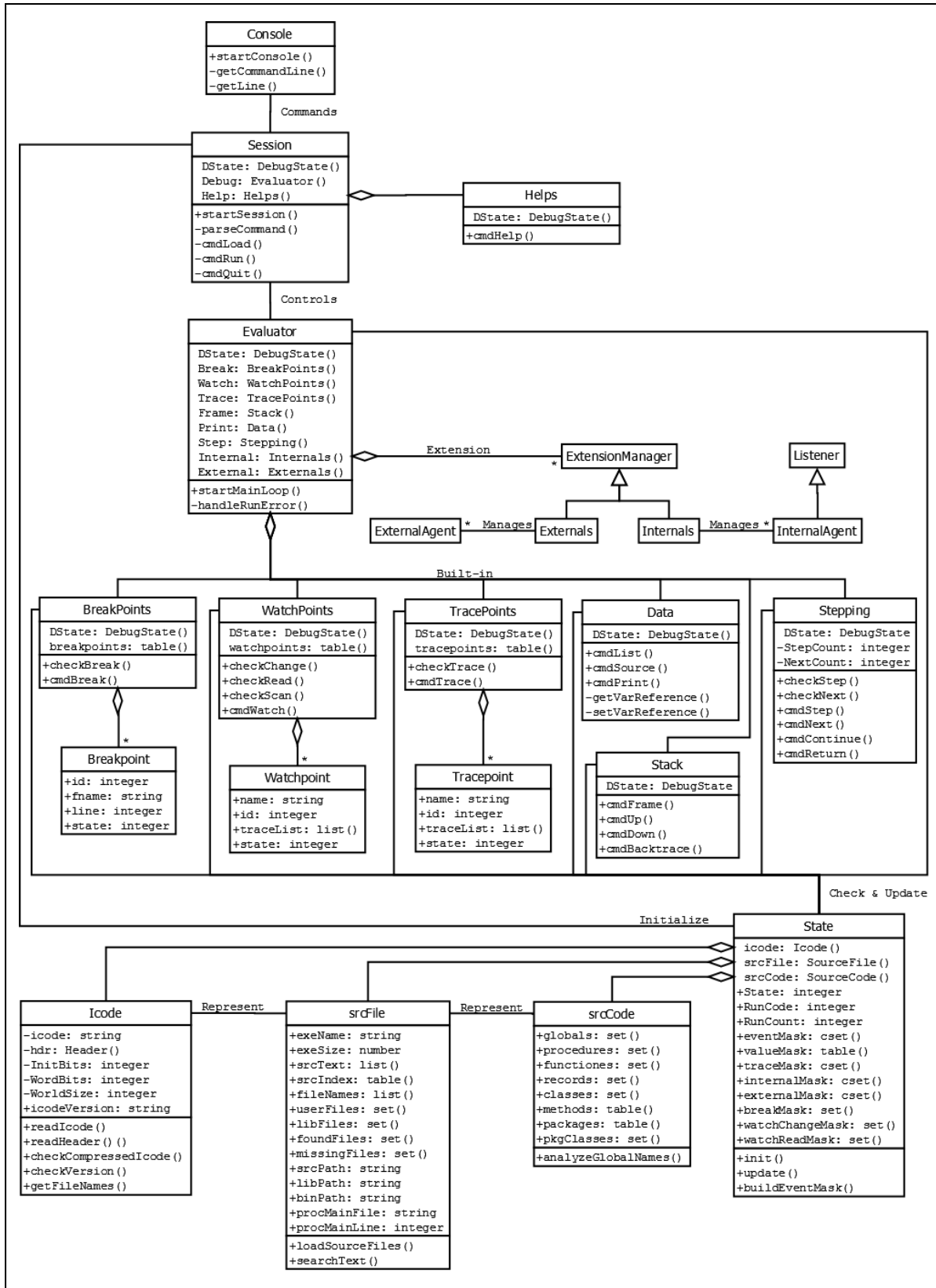


Figure 9.3. UDB's UML Diagram

9.3.1. Console

The top level component in UDB's debugging core is the console. It provides a user interface supported by a command interpreter for user control. It receives a command line from the user and parses it into a list of tokens. The first element in the list is the UDB command followed by its arguments. This command list is passed into the second component which is the debugging session. The source code of this component is modeled by one class named `Console`.

9.3.2. Session

The second component in UDB's debugging core is the debugging session. It initializes and manages the state of the debugger and controls its debugging evaluator. At the start of every debugging session, it loads the target program and analyzes its bytecode. The source code of this component is modeled by two classes:

1. The `Session` class initializes the debugging state and loads the subject program. It interprets all of the commands that are received from the console
2. The `Helps` class provides the basic in-line help functionalities. It reads the debugging state and provides information about commands based on the current debugging context.

9.3.3. Debugging State.

The third component in UDB's debugging core is the debugging state. Initially, the session component initializes this debugging state. At the start of every debugging session, the debugging state is loaded with the available source files and the executable's symbol table. It analyzes the semantic properties of global variables, packages, classes, methods, procedures, and built-in functions. This information assists the debugger and the user with execution state inspection and source code information. This debugging state is updated and maintained during the debugging session based on the user interaction and the state of the buggy program. The source code of this component is modeled by four classes:

1. The `DebugState` class encapsulates the entire debugging state; it has attributes and flags to control the debugger such as what event codes and values the debugging evaluator should be receiving from the buggy program.
2. The `Icode` class opens and analyzes the program's executable virtual machine binary in order to obtain static information about the executable program. The most important information

obtained is a list of all the names of the source files that are contributed to the executable, including library files.

3. The `SourceFile` class opens and organizes the source files that were compiled and linked to construct the program. It provides the ability to search and locate source lines and procedures.
4. The `SymbolTable` class analyzes global names found in the executable. This class maintains the executable's semantic properties such as global variables, procedures, packages, class, and methods.

9.3.4. Evaluator

The final component in UDB's debugging core is the debugging evaluator. This evaluator provides the main event-driven debugging analysis and monitoring control. Built-in debugging features such as breakpoints, watchpoints, tracepoints, stepping and nexting are implemented by internal monitors that are built into UDB's debugging core. By default, UDB's evaluator monitors the `E_Error`, `E_Exit`, and `E_Signal` events, see Table 9.1. The event masks of enabled built-in features and extension agents are added to this set of events. On the fly, UDB's evaluator starts asking the buggy program about those extra events. When the evaluator receives an event from the buggy program, first it checks whether any classical action is needed such as a breakpoint, or watchpoint. Then it checks the extension architecture, which checks its enabled internal and external agent; it forwards events to the enabled ones based on their event mask. The source code of this component is modeled by seven classes; see UDB's UML diagram presented in Figure 9.3 above:

1. The `Evaluator` class implements the main core of the debugging control as an AlamoDE execution monitor. It performs many activities such as: 1) activating the subject program, 2) collecting events out of the subject program, 3) filtering and analyzing according to the debugging context, 4) and forwarding events to all of the classical and advanced debugging facilities.
2. The `BreakPoints` class implements an event-driven breakpoint mechanism by monitoring the `E_Line` execution event. Each breakpoint is represented by an instance of the `Breakpoint` class. See Section 9.4.2.
3. The `WatchPoints` class implements a software watchpoint mechanism by monitoring different kinds of events for different watchpoints. Each watchpoint is represented by an instance of the `Watchpoint` class. See Section 9.4.3.

4. The `TracePoints` class implements an event-driven execution behavior trace by monitoring various execution events based on the requested trace behavior. Each requested trace represented by an instance of the `Tracepoint` class. See Section 9.4.4.
5. The `Stepping` class provides single stepping, nexting, and continuing related commands. It monitors the `E_Line` execution event for some commands, while for others it monitors the level (`&level`) of the stack, especially for commands such as `return` and `next`. See Section 9.4.5.
6. The `Stack` class provides facilities to explore the execution stack. It implements basic stack related commands such as `up`, `down`, `frame`, and `backtrace`. See Section 9.4.6.
7. The `Data` class provides facilities to explore execution data and modify it. It also provides static and semantic information about the executable. For example, it implements basic UDB commands such as `print`, `list`, and `src`. See Section 9.4.7.

Table 9.1. UDB's Default Monitor Events

#	Event Code	Description
1	<code>E_Exit</code>	Reports when the target program terminates normally
2	<code>E_Error</code>	Reports when the target program terminates abnormally
3	<code>E_Signal</code>	Reports when the target program receives an unhandled signal
4	<code>E_MXevent</code>	Reports when a GUI event is handled properly in any of the GUI loaded based external agents

9.3.5. Generators

The debugging `Session` and the debugging `Evaluator` are generators, expressions that suspend values to the caller and are resumed to produce additional values [4, 6]. The diagram in Figure 9.2 shows the control flow inside UDB and how its generators are related to each other. The evaluator generator provides the ability to suspend its main monitoring loop without losing its state. Then control is transferred into its caller, which is another generator called the session generator. The session generator is where the state of the debugging session is saved and later resumed when the user resumes the execution of the buggy program after some investigation in the console interface. This session generator provides implicit ability to maintain the debugging session and the state of the evaluator generator before handing control to the console. This mechanism provides the capacity to

continue debugging by resuming the generator of the debugging session, which continues from its previous state and resumes the evaluator at the point that was suspended.

Those generators allow a clean design that includes two nested loops. First loop is the main console based interface. This loop interprets commands from the user and passes them to the debugging Session. Second loop is the main debugging session loop that iterates until the buggy program is terminated. The implementation of these generators has little impact on UDB's overall performance. They are only resumed after a command line used by the user and they are suspended based on some debugging context that has to be presented to the end user. So, the time complexity of these generators is not noticeable by the end user.

For example, if the received event represents a runtime error `E_Error`, then the generator of the debugging evaluator terminates, returning control to the debugging session. The debugging session saves its state and transfers control back to the console, where the user can investigate. However, if the received event represents any other action such as a breakpoint or a watchpoint, or a bug has been detected, then the generator of the debugging evaluator suspends, thereby saving its state, and transfers control to the debugging session. The debugging session transfers control back to the console with the right message based on the current debugging context and the debugging state. In the console, the user may choose to investigate or resume execution, at which point the generators of the session and the evaluator are resumed.

9.3.6. Main Debugging Loop

The code in Figure 9.4 starts with the `while` loop. In each iteration, `EvGet()` activates the buggy program looking for an event. Events received by the debugger are further filtered, and additional state inspection performed, as part of the execution monitor's analysis. The main debugging action is maintained in both the `State` and the `RunCode` attributes of the `DebugState` class. At the end of the loop and before reactivating the buggy program for a new event, the debugging `State` is checked to decide if the loop has to be suspended, or returned, or just has to look for another event.

9.4. Implementation

UDB implements its classical debugging techniques as well as its advanced agents by monitoring the buggy program for runtime execution events. This section provides implementation details about UDB's various debugging techniques.

```

1  while EvGet(DState.eventMask, DState.valueMask) do {
2      case &eventcode of{
3          E_Line:{
4              if *DState.breakMask > 0 then Break.checkBreakpoint()
5              if DState.RunCode = NEXT then{ Step.checkNext() }
6              if DState.RunCode = STEP then{ Step.checkStep() }
7              }
8          E_Assign | E_Value :{
9              if *DState.watchChangeMask > 0 then Watch.checkWatchChange()
10             }
11         E_Deref:{
12             if *DState.watchReadMask > 0 then Watch.checkWatchRead()
13             }
14         E_Spos | E_Snew:{
15             if *DState.watchChangeMask > 0 then Watch.checkWatchScan()
16             }
17         E_Exit:{
18             DState.State := END
19             }
20         E_Error:{
21             DState.State := PAUSE
22             DState.RunCode := ERROR
23             handleRunError()
24             }
25         E_Signal:{
26             DState.State := PAUSE
27             DState.RunCode := SIGNAL
28             }
29     } # end of case ecode
30
31     if *DState.traceMask > 0 & member(DState.traceMask, &eventcode) then
32         Trace.TraceBehavior()
33     if Internal.enabled > 0 & member(Internal.eventMask, &eventcode) then
34         Internal.forward()
35     if External.enabled > 0 & member(External.eventMask, &eventcode) then
36         External.forward()
37
38     if Dstate.State = PAUSE then suspend
39     if Dstate.State = END then return
40 }# end of while

```

Figure 9.4. UDB's Main Debugging Loop

9.4.1. Loading a Buggy Program

UDB dynamically loads the program's binary executable, see Chapter 5. At load time, UDB analyzes the code in order to obtain a complete list of source file names in use; including library files. When a program is loaded, UDB builds its related symbol table with fields including: all global variables, procedures, built-in functions, records, classes and their methods, and all packages and their global variables, classes, and procedures.

9.4.2. 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. Utilizing the value mask of the AlamoDE framework approximates this implementation. This value mask provides an extra condition, applied on the event value. It limits the line number event code `E_Line` to those values provided by the value mask, see Figure 9.5.

```

1  method checkBreakpoint()
2    local cur_file, cur_line, L, x, id
3
4    cur_file := keyword("file", Monitored)
5    if L := member(breakPoints, cur_file) then{
6      cur_line := &eventvalue
7      every x := !L do{
8        if cur_line = x.line & x.state = ENABLED &
9          id := isBreakExist(cur_file,cur_line) then{
10         DState.State := PAUSE
11         # Temporarily remove the breakMask set from the valueMask table
12         # until the "continue" command is applied. This allows the "next" and "step"
13         # commands to operate
14         delete(DState.valueMask,E_Line)
15         msg := "\n Breakpoint # "||id||" at : "|| cur_file||":"||cur_line||".
16         DState.Write(msg)
17         return
18       }
19     }
20 end

```

Figure 9.5. UDB's Implementation for Breakpoints

In this implementation, there is only a context switch if the line number event and its value are satisfied. For example, when the user applies a breakpoint command, UDB checks its location within the program's source code, to ensure the line number is within the file and it has an executable statement. It inserts the `E_Line` event code to the set of monitored events; the event mask. At the same time, it associates this event mask with the line number of this breakpoint. So, the internal instrumentation will only report the `E_Line` event for those lines that match one of the lines found in the value mask. For the sake of better monitoring consistency, a breakpoint on a procedure or method is converted internally into a breakpoint on a line number, which is the line number of the procedure's header. Figure 9.5 shows the `checkBreakpoint()` method, which is called by UDB's evaluator when the `E_Line` event is reported. In line 4 of the figure, UDB inquires the current executable file name, using the built-in function `keyword("file", Monitored)`, to ensure the reported line number event is from the right file name. Alamo's `E_Line` filtering mechanism of event mask and value mask does not check for the program source file. This induces a false positive report of the `E_Line` event when its value is satisfied regardless of the source file.

9.4.3. Watchpoints

UDB's watchpoints are implemented by monitoring the assignment event `E_Assign` only for those variables that have predefined watchpoints on them. Utilizing the value mask over the `E_Assign` event approximates this implementation in a technique similar to the one discussed in Section 9.4.2. This implementation takes advantage of the dynamic event masking and value masking provided by AlamoDE. For example, when the user applies a watchpoint command, UDB resolves its scope and inserts the `E_Assign` event code to the set of monitored events; the event mask. At the same time, it associates this event mask with the name of this watched variable. So, the internal instrumentation will only report the `E_Assign` event for those assignments that are related to that exact variable.

After the watchpoint command is applied, UDB starts monitoring this `E_Assign` event. The method `checkWatchChange()` provided in Figure 9.6 shows UDB's implementation for typical watchpoints that check whenever a variable is assigned. The event value of the `E_Assign` event is the name of the variable to be assigned. The event is reported right before the assignment is accomplished. In order to obtain the new value to be assigned, AlamoDE provides the `E_Value` event. This event is always associated with the `E_Assign` event and reports the assigned value. For performance reasons, UDB's evaluator monitors the minimum number of events based on the current debugging context. After UDB validates that the reported `E_Assign` has a currently enabled

watchpoint on it, it obtains this variable value by updating the event mask with a new event code `E_Value` before re-activating the target program, see line 12 of Figure 9.6. This allows the target program interpreter to report the value of the assigned variable as soon as that variable is assigned. After the watchpoint is reached, this event code `E_Value` is removed from the set of monitored events, see line 21 of Figure 9.6.

```

1  method checkWatchChange( )
2    static var, viv, hit := 0, evaluate := 0
3
4    if &eventcode == E_Assign &
5      member(DState.watchChangeMask,&eventvalue) &
6      (viv := varInfo[&eventvalue]) & viv.state = ENABLED then{
7      var := &eventvalue
8      if /viv.hitMax | viv.hitMax < 0 | (viv.hitCount < abs(viv.hitMax)) then{
9        hit := 1
10       if not member(DState.eventMask, E_Value) then{
11         evaluate := 1
12         DState.eventMask ++:= cset(E_Value) # adds E_Value to the value mask
13       }
14       return
15     }
16   }
17 else if &eventcode == E_Value & hit=1 then{
18   hit := 0
19   if evaluate = 1 then{
20     evaluate := 0
21     DState.eventMask --:= cset(E_Value) # removes E_Value to the value mask
22   }
23   viv.oldValue := viv.curValue
24   viv.curValue := &eventvalue
25   if \viv.catchValue then
26     checkCatchValue(var)
27   else
28     printWatchedVarInfo(var)
29   return
30 }
31 end

```

Figure 9.6. UDB's Implementation for Watchpoints Check

9.4.3.1. Advanced Watchpoints

Besides typical watchpoints that observe variables being assigned, UDB supports special watchpoints that deal with advanced language features such as string scanning environments and implicit type changing. UDB's watchpoints are capable of observing expressions such as:

1. Variable assignment. See `awatch` command in Appendix A.8.
2. Variable read (dereferenced). See `rwatch` command in Appendix A.8.
3. Variable assigned a value different from the previous value. See `vwatch` command in Appendix A.8.
4. Variable assigned and the new type is different from the previous type. See `twatch` command in Appendix A.8.
5. A keyword explicitly assigned by the program's code, and
6. An implicit string scanning environment is changed by the string scanning primitives; mainly the `&subject` and `&pos` keywords. See `swatch` command in Appendix A.8.

For example, the command `swatch` observes every operation on a string scanning environment and shows a window of the scanned string along with other information such as the old position, the new position, and the delta between them.

UDB's watchpoints may cause the program to stop, or they may work silently collecting information about specific evaluation(s). Silent watchpoints collect location and value about specific evaluations without pausing the program's execution. The user may review collected information at any point during or after the execution. Regardless of whether the watchpoint is silent or not, the user is able to set a watchpoint for a limited number of satisfied incidents. See Appendix A.8.

9.4.3.2. Watched Variables

When reported in an Alamo event value, variable names are mangled with *scope code* characters that identify the scope of the reported variable. The characters “-”, “^”, and “:” are appended along with the procedure name to distinguish normal local, parameter, and static variables respectively. Global variables are distinguished using the “+” character attached to the end of the variable name [108]. The potential value mask associated with the `E_Assign` event is a set consisting of the watched (monitored) variables. This value mask eliminates the `E_Assign` event from being reported for similar variable names found in other procedures.

UDB uses these scope codes while watching variables. When the buggy program is loaded but not running yet, the user can set watchpoints on valid keywords, global variables, and local variables that are provided by the command as mangled variables (i.e. `watch a-main`). When the program is stopped for a breakpoint, 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.

Locals that are mangled and their procedures are currently active on the call stack are verified based on dynamic information from the current execution state. Otherwise, UDB uses the static information collected from the buggy program at load time to ensure that those variables are valid. 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 command, UDB checks whether it is a keyword. If it is not a keyword, then UDB looks it up in the currently selected stack frame. If this variable name is neither a keyword nor a local variable, then UDB looks it up in the global variables. Otherwise, UDB complains with an error message.

9.4.4. Tracepoints

UDB's tracepoints are another extension that goes beyond the capabilities of 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. For example, often programmers write their functions and procedures to return a specific value as an error code, which may describe an unfinished or failed job. UDB's tracepoints allow a user to place a tracepoint on a specific procedure returned value. The command `"trace bar return <= 1"` sets a tracepoint on procedure `bar` whenever it returns a value `<= 1`.

This type of tracing provides additional flexibility in order to simplify and speed up the process of discovering bug locations. The user can check the traced info from any point during or after the execution. Traced execution 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`. For example, the command `"trace 10 bar resume"` sets a tracepoint on procedure `bar` for the first 10 times it

resumes, see Appendix 1.9. Behaviors are associated with the semantics of the Unicon/Icon language, see Table 9.2.

Table 9.2. UDB's Tracepoints

Behavior	Description
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
call	represents normal procedure, built-in function, or operator call
resume	Represents the resumption of a suspended procedure, built-in function, and operator
return	Represents exiting a procedure with the language keyword <code>return</code> . For built-in functions and operators represents the behavior of finishing a successful call
fail	Represents exiting a procedure with the language keyword <code>fail</code> or reaching the end of the procedure. For built-in functions and operators represents the behavior of failing to accomplish the intended job
suspend	Represents suspending with the language keyword <code>suspend</code>
remove	Represents removing a suspended procedure, built-in function, or operator as a result of exiting a parent procedure, built-in function, or operator

9.4.5. Stepping and Continuing

UDB implements the `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 they are needed by another currently enabled debugging feature.

Figure 9.7 and 9.8 show the implementation of the two methods responsible for the `next` command in UDB. First, the implementation of the `next` command is provided by the `cmdNext(cmd)` method in Figure 9.7. It receives a command from the user and updates the debugging state with the `NEXT` flag. It checks and saves the current stack level using `keyword("&level", Monitored)` function. This level is used by the second method named

checkNext() provided in Figure 9.8. This method is called at every line number change E_Line event after the next command is applied. It checks for the appropriate line where to stop after the next command. It also inquires the keyword &level of the buggy program and compares its value against the level obtained when the command was applied.

```

1  # starts the next command
2  method cmdNext(cmd)
3    local count
4    if DState.State = PAUSE & DState.RunCode ~= ERROR then {
5      if count := integer(cmd[2]) then nextCount := count
6      else nextCount := 1
7      nex_level := keyword("level", Monitored)
8      DState.Update(NEXT)
9      DState.Write(" Nexting.")
10   }
11  else {
12    DState.State := ERROR
13    msg := "\n The program is not being run._
14          \n Try \"run\", or Type \"help\" for assistance"
15    DState.Write(msg)
16  }
17  end

```

Figure 9.7. Initiating a Next Command

```

1  # checks for the appropriate line where to stop after the next command.
2  method checkNext()
3    local level
4    level := keyword("level", Monitored)
5    if level > nex_level then
6      nextCount +=1
7    if level = nex_level then{
8      if nextCount > 1 then { nextCount -= 1 }
9      else if next_count = 1 then{
10     nextCount := 0
11     stepCount := 1
12     DState.State := PAUSE
13     DState.RunCode := STEP
14   }
15 }
16 end

```

Figure 9.8. Implementing Next within the Evaluator

9.4.6. Stack Navigation

When a program performs a procedure call, information about the call is generated and saved in the execution stack in blocks called procedure frames, which are distinguished by their level. Each frame includes arguments and 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, UDB provides the ability to investigate where and/or how the program's execution got to this point. By default, when the buggy 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 frame is frame number 0. In contrast, the oldest frame on the stack has the biggest frame number, which is the frame for procedure `main()`. UDB allows a user to explicitly jump to any other frame. UDB associates procedure frames with the exact statements in the source code that instantiated them by utilizing the AlamoDE `keyword()` primitive to find relevant keyword values such as `&file`, `&line`, and `&level`.

9.4.7. Data Navigation/Modification

UDB provides the ability to examine and change data during the execution. Specific variables, keywords, and data structures can be looked up and modified during the debugging process using the `variable()` primitive. When a variable is looked up, its type is checked implicitly to assist the user better. If the target variable has an *Atomic Type* such as `null`, `integer`, `real`, `cset`, or `string`, then the presented value is the variables current value. Otherwise, if the target variable has a *Structured Type* such as `list`, `table`, `record`, `set`, `procedure`, or `window`, then UDB provides the user with a string representation of the referenced structure. 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. In contrast, the `ximage` of a structure is a string containing the elements of that structure and its substructures.

Part IV

Extension Agents

Chapter 10

UDB's Advanced Debugging Agents

Bugs vary in their root causes and their revealed behaviors; some may cause a crash or a core dump, while others may cause an incorrect or missing output or an unexpected behavior. Moreover, 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 often requires users to manually track heuristic information over different execution states. This information may include a trace of specific variables' values and their assigned locations, procedures and their returned values, and detailed execution paths.

10.1. UDB's Extensibility

UDB's 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. This style of debugging is not always good enough. Augmenting UDB with various agents is one way to improve its standard debugging process. For UDB, agents may retain information beyond the current state of execution and perform automatic and dynamic analysis techniques. 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 [18] dynamic analysis within an interactive debugger makes both methods more effective.

Taking advantage of the IDEA architecture presented in Chapter 8, UDB allows various agents-based tools and techniques to be used during a debugging session. Any event-driven AlamoDE-based standalone program can be loaded and used on the fly during a debugging session. External agents are enabled at load time but may be explicitly disabled and re-enabled by the user at will. Originally, some of UDB's agents were written and tested as standalone programs, used as external debugging agents under UDB, then migrated to internals for reasons that include:

1. Better performance. External agents are used through context switches whereas internal agents are used through a relatively faster procedure call mechanism
2. Better availability. Unlike external agents that must be located and loaded for every debugging session, internals are always available and the user has only to enable them when they are needed. Internal agents are distributed with the source code of UDB

3. Agent collaboration. Some internal agents can be used from within the temporal logic operators, which are another set of internal agents. So, dynamic temporal assertions allow some agents to implicitly use other agents, see Chapter 11 for more information.

Currently, UDB has a library of different internal agents, which monitor different behaviors such as memory allocations, garbage collections, loop iterations, loop times, and procedure times. Internal agents are disabled by default; the user has to enable them explicitly during the debugging session. This chapter presents three kinds of UDB's potential extension agents.

10.2. Visualization Agent

Visualization tools are standalone graphical tools used to summarize and depict execution properties and present their analyzed data by visual means. Figure 10.1 shows an example of two visualization tools loaded on the fly at the beginning of a UDB debugging session. These tools are not interactive; they work simultaneously in the background of the debugging session.

The debugging session in Figure 10.1 shows a moment during an execution of the Unicon translator under UDB (a preprocessor that translates Unicon down to Icon). During this debugging session, the translator was given a relatively large Unicon module (`idol.icn`, 1235 lines) as input. The upper tool shows an incremental view of the total memory allocations in both of the string and block regions. The lower tool presents the allocation and usage of various data types. Each allocation type is coded in a different color. The upper row of pie charts shows the percentage of total allocations, number of allocations, string allocations, block allocations, and number of created structures respectively. The second row shows the usage of each one of those data structures starting with list, tables, sets, records, and the usage percentage of all data structures in use.

These visualization tools are used as external agents and have not been migrated to internals. However, these tools being loaded and used within a typical interactive session, the user is able to manipulate the target's program control and data flow through means of breakpoints and watchpoints. For example, the user can place a breakpoint on some line number and stop the program. This allows the user to check the tools' results up until that point. It is also possible to single step the execution and simultaneously check the incremental progress in the visualization process. Moreover, if any of these tools are used in a standalone mode, the user will have no control over the execution of the monitored program unless it is already supported by the tool's user interface.

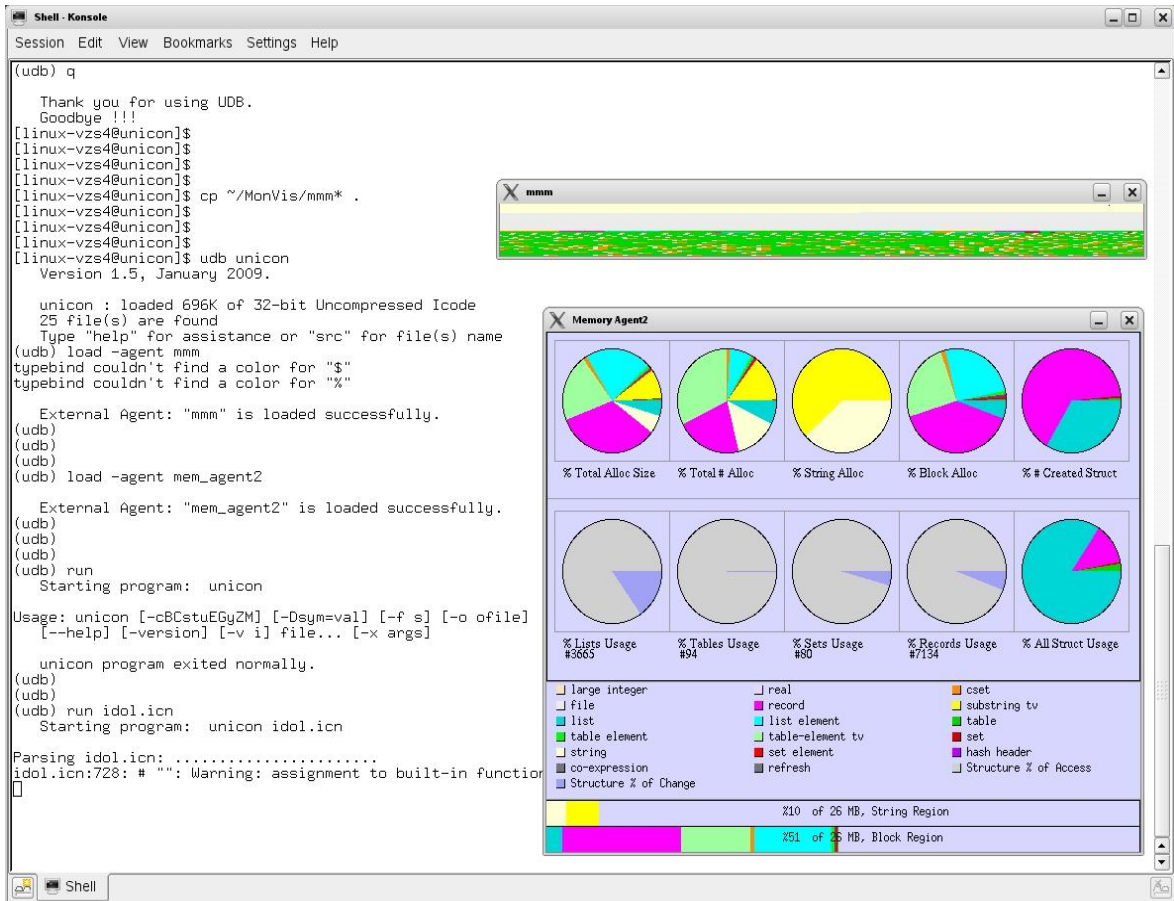


Figure 10.1. UDB's on-the-fly Visual Extensibility

10.3. Language-Specific Agents

UDB employs a set of internal agents to locate numerous potential bugs associated with the semantics of the Icon and Unicon languages. UDB's IDEA support provides commands to enable and disable such agents, see Appendix A.16. Those agents monitor execution behaviors looking for specific symptoms such as:

1. Variables that may change their type during the course of execution
2. Expressions that may fail silently in contexts where failure is not being checked, and
3. Redundant implicit type conversion which may hurt the execution performance.

The following subsections provide a discussion for each one of these agents.

10.3.1. 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; they usually 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 agent's implementation is very expensive. It can be enhanced with information from the static type inference used originally by the Icon's compiler named `iconc` [128] used in Unicon when the `-C` option is used on the command line [129, 130]. This type inference output may limit the detection process to a much smaller subset of suspicious variables.

10.3.2. Failed Expressions

Unicon's logic programming flavor of failure and success has its advantages. But, if it is not used properly, it will induce side effects into the execution of the program. In practice, not all failures are intentional. Sometimes, a failure can point at a potential cause of a bug. For example, 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 expressions, and they can decide for themselves whether it is a bug or not. This agent performs the suspicious failure check by monitoring failures in various expressions and built-in operators and reports where and when that failure was happened. An example, of the monitored events is `E_Efail`, `E_Ofail`, and `E_Ffail` which they report after expression failures, operator failures, and built-in function call failures respectively.

10.3.3. 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 slow down might be caused by any number of bad programming practices. In Unicon, one common performance bug 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 two events: `E_Sconv` and `E_Tconv`, which report the successful conversion and the result type of the conversion respectively.

10.4. Language-Independent Agents

Finally, UDB is extended with two sets of general agents: data and behavior related agents. This set of agents is called *Atomic* for their tiny sizes and outcome results that produce a value, which may be boolean, numeric, string, or even a structure. They facilitate simple but common operations during the debugging process and provide extra heuristic information with easy processing. These two sets of agents can be used as standalone internal agents or as atomic operations applied by UDB's dynamic temporal assertions presented in Chapter 11. These extension agents expand the usability of UDB's built-in features with the ability to validate more specific data and behavioral aspects of the execution properties.

10.4.1. Data Related Agents

This category of extension agents is used to retain and process data in relevance to the execution state. Data agents are used to utilize advanced on demand specific data tracing techniques. The user can enable and disable those agents to work on different variables; each agent can be enabled several times, each on a different variable provided by the user at enabling time. See Table 10.1 for a list of all UDB's atomic data extension agents.

For example, depicting a variable's initial, previous, current, or next value can be critical in understanding the evaluation of an expression and the execution of the program. Normally a user can place a watchpoint on specific variable to inspect its value during the execution. This watchpoint notifies the user whenever the target variable is changed or even read. Then the user has to write down or memorize these values trying to understand how the evaluation develops during the execution. Sometimes, the user may end up doing some calculations on those written down data such as finding the minimum, maximum, sum, and average. Or even carefully watch those values to find when a new maximum or new minimum is reached. UDB provides various agents that can be employed to semi-automate such a process and save the user's time and effort. These data related agents can allow the user to automatically collect various properties about specific variables and retain them whenever it is needed.

Table 10.1. Atomic Data Related Agents

Agent Name	Return Type	Description
initial(x)	Any	The initial value of x
final(x)	Any	The final value of x
old(x)	Any	The previous value of x
current(x)	Any	The current value of x
new(x)	Any	The next value of x
max(x)	Numeric / String	The maximum of all x values
min(x)	Numeric / String	The minimum of all x values
newmax(x)	True/False	Evaluate True if x has new max, False otherwise
newmin(x)	True/False	Evaluate True if x has new min, False otherwise
sum(x)	Numeric	The sum of all x values
avg(x)	Numeric	The average of all x values

10.4.2. Behavior Related Agents

During a debugging process, a user may try to understand the behavior of the execution by watching specific runtime properties such as the number of times a loop has been iterated or the number of times a procedure has been called, a variable being assigned, read, referenced, or even initialized, and how many aliases a data structure has and what they are. UDB employs behavior related agents that will reduce the manual inspection that could be done using traditional breakpoints and watchpoints. See Table 10.2 for a list of UDB's atomic behavior related agents.

This set of extension agents is intended to facilitate users' ability to validate and check specific execution behaviors. They provide advanced on demand specific behavior tracing techniques. Some of those agents are focused on the call/return behavior of procedures; which either counts the number of times a procedure has been called or what value is returned. Other agents are focused on the read/write and aliasing of variable behaviors.

For example, 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. The agent

iteration(**while**) finds the actual number of iterations for the last **while** loop. This agent checks loops iteration by monitoring both the **E_Syntax** and **E_Efail** event codes, which report the current syntax and failed expression respectively.

Table 10.2. Execution Behavior Related Agents

Agent Name	Returns	Description
call(proc)	Integer	The number of times proc is been called
return(proc)	Variable	The current value returned by proc
initialized(x)	True/False	True if x was assigned at first reference, False otherwise
dead(x)	True/False	True if x is never referenced at least once, False otherwise
reference(x)	Integer	The number of times x is been read + written
assign(x)	Integer	The number of times x is been assigned
read(x)	Integer	The number of times x is been read only
alias(x)	List	All current x aliases
iterations(loop)	Integer	The number of actual iterations of loop

Another example targets uninitialized and dead variables, which are variables 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, the **dead(x)** agent detects variables that are theoretically live according to static analysis and the user's expectations, 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 helps 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.

Chapter 11

DTA: Dynamic Temporal Assertions

This chapter introduces the idea of Dynamic Temporal (DT) assertions into the conventional source-level debugging session. It extends UDB with on the fly DT assertions that are separate from the program source code. Each assertion is capable of:

1. Validating a sequence of execution states, named temporal interval
2. Referencing out-of-scope variables, which may not be live in the execution state at evaluation time
3. Employing a growing set of user defined atomic agents (internal extension agents).

These new assertions are not bounded by the limitations of ordinary in-code assertions such as *locality*, *temporality*, and *static hardwiring* into the source code. Furthermore, they advance typical interactive debugging sessions and their conditional breakpoints and watchpoints. UDB's DT assertions serve three purposes:

1. Extend the usability of conventional source-level debuggers' conditional breakpoints and watchpoints. This simplifies the ability to validate relationships that may extend over the entire execution and check information beyond the state of evaluation
2. Reduce the number of times a user has to stop and single step the execution for state-based investigation
3. Augment a traditional breakpoint-based debugging session with testing and verification capabilities. It introduces testing and verification features into traditional source-level debugging sessions [125]. For example, it allows users to verify loop invariants.

11.1. Temporal Assertions

Assertions are logical expressions that are inserted into the source code of a program. When execution reaches the asserted statement, it asserts that some property holds in the program's current state. If the asserted expression does not hold, the assertion will abort and terminate the program's execution. In contrast, Temporal Assertions are logical expressions that use Temporal Logic (TL) in order to validate, not one state, but a *sequence of execution states*, such as a sequence of variable values changed within a block of code [126].

11.1.1. Temporal Logic

Temporal logic is a special branch of modal logic. It emphasizes the notion of time and order [123, 124]. Linear-time Temporal Logic (LTL) is a special branch of temporal logic that extends propositional logic with a new set of operators such as:

1. **Next:** the property must hold in the next step
2. **Previous:** the property must hold in the previous step
3. **Finally, Eventually, Sometime:** the property will hold at some state in the future
4. **Globally, Always:** the property must hold at every state on the execution path
5. **Until:** the property has to hold until some other property holds
6. **Since:** the property should hold since another property was held

LTL is mostly used to measure program correctness. Metric Temporal Logic (MTL) extends LTL to support real-time and relative-time constraints [123, 124]. MTL has two models 1) a *point-based* model that observes the target program at every instant in time, and 2) an *interval-based* model, that observes the target program over an interval of time. This chapter utilizes a combination of these two models within an interactive debugging session. The extended debugger provides Temporal Assertions that allow programmers to check real time execution properties over both temporal-state and temporal-interval and during interactive debugging sessions.

11.1.2. Temporal Assertions vs. Ordinary Assertions

Standard in-code assertions are inserted into the source code to validate *pre-* and *post-*conditions or to check the value of some variables and expressions. In general, typical assertions suffer from three limitations: locality, temporality, and lack of dynamicity within the source code of the buggy program. The following three sub-sections discuss these limitations in detail.

11.1.2.1. Locality

An ordinary assertion is bounded by its location (scope); it cannot reference a variable from another scope even if it is live based on the current execution state. Assertions live in one of the functions; each can reference local and global variables. If the scope is a method, it can reference any of the class variables. In fact, typical assertions cannot check or validate local variables in other functions or methods, even if that foreign local is static or still live somewhere on the stack of the current program's execution state.

For the sake of simplicity, suppose that a variable in the calling function must be checked against another variable in the callee. A typical assertion cannot reference both of them at once. DT assertions provide a simple solution for such situations. See Figure 11.1 where a DT assertion can be virtually inserted into line 27. The assertion would refer to the local variable `y` in function `bar()` and compares its value against the variable `x` from function `foo()`. Furthermore, even though this assertion is inserted at line 27, it will evaluate the asserted expression (`y >= foo:x`) whenever the value of `y` is changed within `bar()`. That is because it is a Temporal Assertion, not a typical assertion, and its scope is procedure `bar()`. This particular assertion asserts that always the value of this expression (`y >= foo:x`) must hold (evaluate to true) for every evaluation, whether it is on the temporal-state or temporal-interval level. When this assertion is triggered for evaluation, by entering procedure `bar()`, the assertion agent has already obtained the last value of variables `x` of procedure `foo()`. Variable `y` is local to where the assertion is virtually located, so the assertion agent will inquire this value at evaluation time. Of course, this particular example can become more interesting than its current version when `foo()` is a recursive function and `x` is a parameter to `bar()`.

```

10 procedure foo( )
11   local a, b, c
12   x := 10
13   .....
14   .....
15   .....
16   bar()
17   .....
18   .....
19 end
20 procedure bar( )
21   local a, b, c
22   y := 20
23   .....
24   .....
25   .....
26   y := ( y * a ) / b - c
27   // virtually assert always() { y >= foo:x }
28   .....
29   .....
30 end

```

Figure 11.1. A DT Assertion over Two Live Procedures

11.1.2.2. Temporality

An ordinary assertion is bounded by the current state of execution. It can check only the current value of the referenced variables. In Figure 11.2, line 75 of procedure `baz()`, both `foo()` and `bar()` are not on the stack any more. What if a user needs to check the value of variable `x` from procedure `foo()` against variable `y` from function `bar()`? Ordinary assertions are found to be useless once more.

```

10 procedure foo( )
11     static x := 0
12     x += 1
13     .....
30 end
31 procedure bar( )
32     static y := 0
33     y += 1
34     .....
60 end
61 procedure baz( )
62     foo()
63     .....
74     bar()
75     // virtually assert always() { bar:y >= foo:x }
76     .....
90 end

```

Figure 11.2. A DT Assertion over Two Sibling Functions

11.1.2.3. Source Code Location

An ordinary assertion is bound to the source code, where it is written and compiled in the executable; any change or modification requires the ability to recompile and rebuild the executable. If the ordinary assertion evaluates to false, it may provide a warning statement or terminate the execution. If the user wants to investigate, he/she may modify the assertion by tightening or loosening the condition, or adding nearby assertions. In addition, the user may consider loading the buggy program under a source-level debugger, which allows single stepping and provides the ability to investigate the execution state. Thus, DT assertions work with the source-level debugger, where the user is interactive with the execution and able to insert, delete, and modify DT assertions on the fly without source or object code modification.

11.1.3. Temporal Assertions vs. Conditional Breakpoints

Conditional breakpoints and watchpoints are dynamically inserted during the debugging session. They can check execution properties and stop the execution whenever a condition is satisfied. Even though such breakpoints may have the advantage of being conditional and dynamic with on the fly insertion, deletion, and modification, they are still bounded to their locations; the exact line number in the source code of the target program and the state of the referenced variables and objects in that location.

In general, most source-level debugging techniques rely heavily on the user's ability to investigate when the program is stopped. For example, in reference to the instance provided in Figure 11.1 of Section 11.1.2.1, tackling similar problems from inside a conventional source-level debugger will take one of two approaches. The first approach is to insert two different breakpoints, where the user has to investigate at each stop and memorize or write down the value of variable `x` from the first breakpoint in order to compare it against the value of `y` at the second breakpoint. The second approach takes advantage of function `foo()` being on the stack. The user may insert one breakpoint in function `bar()`, find the value of `y`, navigate the stack to find the value of `x`, and compare them. In contrast, a DT assertion is able to reference the out-of-scope variable `x` and compare it directly against the in-scope variable `y` and notify the user only when the assertion evaluates to false.

Unlike breakpoints that stop the execution only when the condition evaluates to true, this DT assertion default action is to stop the execution when the condition is violated. The evaluation action `hide` is the default action for any temporal assertion that evaluates to true. However, the user can change this default action to `pause`, `show`, or `stop`, see Section 11.4.5. Furthermore, DT assertions are able to reference variables that are not accessible (not active in the current execution state) at evaluation time. This feature solves the problem provided in Figure 11.2 of Section 11.1.2.2, which shows that procedure `foo()` and `bar()` are siblings in `baz()`.

11.2. UDB's DT Assertions

Figure 11.4 provides a simple UDB debugging session with an example of a DT assertion used to validate that a recursive function is going in the right direction. The debugged program is shown in Figure 11.3. The debugging session example shows a recursive implementation of the factorial calculation and illustrates how a DT assertion is used from within UDB's console-based interface. The assertion is inserted using the `assert` command, which dynamically anchors the body of the assertion in the executable of the buggy program, at line 3 of Figure 11.3, with no executable or source code modification; more details about the notation are discussed in Section 11.7.

During a debugging session, a user is always able to adjust and move assertions manually. However, if for any reason, the target program source code has to be changed and reloaded under the debugger, currently, the debugger has no means to maintain those ad-hoc assertions within their virtual source code related location. A future work is planned to maintain these assertions in a debugging session configuration file that will allow the user to reload the session. When the assertion is found to be misplaced, if the debugger failed to automatically adjust any of the affected assertions, the debugger may ask the user to manually relocate it.

```

1  # The factorial sample test program
2  procedure fact(n)
3      if n <= 1 then
4          return 1
5      else
6          return n * fact(n-1)
7  end
8  procedure main(arg)
9      write("The factorial of ", &programe," is ",fact(arg[1]) )
10 end

```

Figure 11.3. Sample Factorial Program Written in Unicorn

```

1  $ udb factorial
2      UDB Version 2.0, December 1, 2009.
3      factorial: loaded 2.5K bytes of 32-bit uncompressed icode
4      1 Source file(s) found
5      Type "help" for assistance
6
7  (udb) assert factorial.icn:3 alwaysp() { old(n) > current(n) }
8      Assertion 1#: assert factorial:3  alwaysp() { old(n) > current(n) }
9      is set successfully.
10
11 (udb) run 5
12     running factorial ...
13 The factorial of 5 is 120
14     The factorial program exited normally.
15 (udb) quit
16     Thank you for using UDB, Goodbye !
17 $

```

Figure 11.4. Sample UDB Session that Uses DT Assertions

11.3. Debugging with DT Assertions

DT assertions, within a typical source level debugger, provide an extension of conditional breakpoints and watchpoints. They employ agents that implement temporal logic operators, each with an automatic tracing mechanism. Traced data are assertion-driven; relevant information is gathered and analyzed in real time. Different DT assertions can be applied on different execution properties with dynamicity and flexibility. Each assertion is capable of validating program properties that may extend over a sequence of execution states. UDB's DT assertions have the following features:

1. Dynamic insertion, deletion, enabling, disabling, and modification. Assertions are managed on the fly during the debugging session without source or executable code alteration
2. A non destructive way of programming supported by an *assertion free source code*. In general, debugging information is needed only during program development, testing, verification, validation, and debugging
3. Assertions are virtually inserted and evaluated as part of the buggy program source code. All assertions live in the debugging session configuration; each is evaluated by the debugger in the debugger execution space. The debugger automatically maintains *state-based* techniques to determine what information is needed to evaluate each assertion, and it uses *event-based* techniques to determine when and where to trigger each assertion evaluation process. Some program state-based information is collected before assertion evaluation, while other information is obtained during the evaluation process; see Section 11.6.1 and 11.6.2. All DT assertions are evaluated as if they were part of the target program space
4. Optional evaluation suite, where a user can specify an evaluation action such as **stop**, **pause**, **show**, and **hide**. Both **pause** and **show** actions enrich assertions with the sense of in-code tracing and debugging with print statements, where a user can ensure that the evaluation has reached some points and the referenced variables satisfy the condition, see Section 11.4.5.
5. The ability to log the assertion's evaluation result. This lets the user review the assertion evaluation history for a specific run. Evaluated assertions are marked with **True** or **False**. Some DT assertions may reference data in the future; those assertions are marked with **Not Valid** for that exact state-based evaluation. Assertions' intervals are marked with a counter that tracks their order in the execution. If an assertion has never been reached, it is distinguished by its counter value, which is zero in this case; see Section 11.4.7. Log comparison of different runs is considered in future works.

6. Most importantly, DT assertions can go beyond the scope of the inserted location. Each assertion may refer to variables or objects that were living in the past during previous states, but not at evaluation point, and each assertion may compare previous variable values against current or future values. Each DT assertion implicitly employs various agents to traces referenced objects and retains their relevant state information in order to be used at evaluation time.

11.3.1. Example #1: Loop Invariant

Checking a loop invariant is one of the most difficult tasks during conventional interactive debugging. A programmer might end up with several breakpoints, watchpoints, and single stepping. However, utilizing UDB's temporal assertions allows a programmer to check a specific loop invariant with one simple DT assertion. Figure 11.5 shows a selection sort algorithm that violates the loop invariant marked with #1, whereas #2 is where the invariant might get violated. A UDB user can check the invariant for this loop using the `assert` command provided in #3. This assertion can be inserted at any line within this procedure. It will notify the user at the very first incident that the asserted expression is violated.

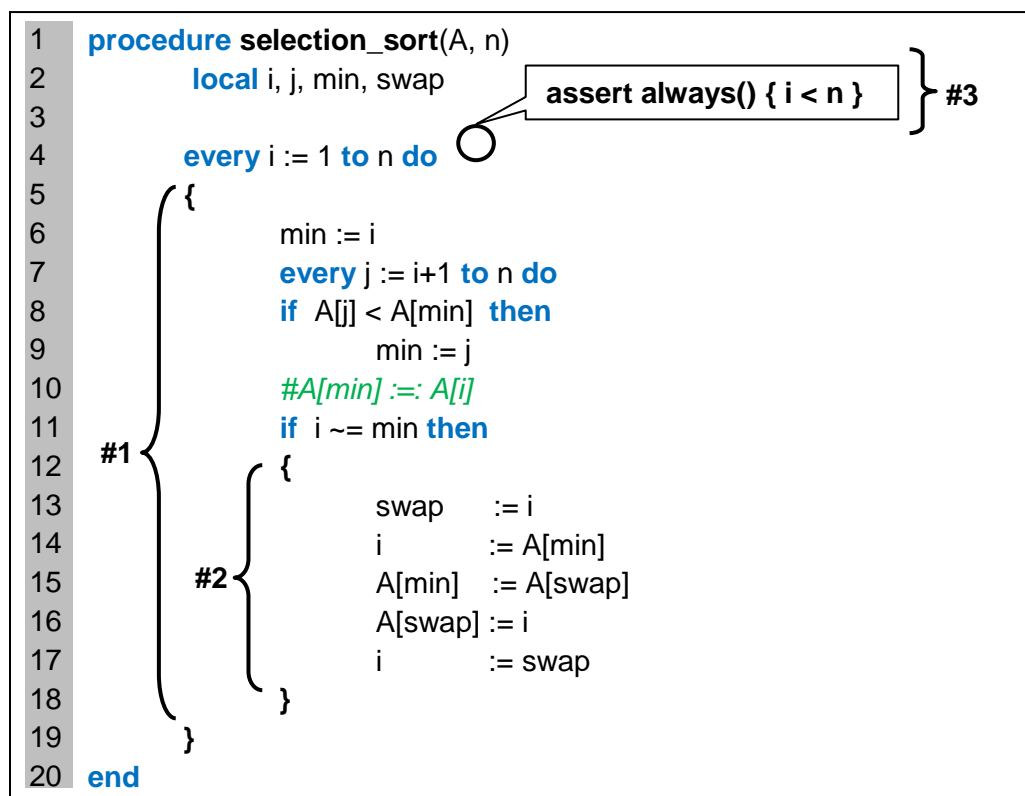


Figure 11.5. Using Temporal Assertions to Check Loop Invariant

11.3.2. Example #2: Sequence of Variable States

During a debugging session, sometimes, it is useful to check a variable value against its old value (last or previous value). Programmers might accomplish this using a typical technique such as breakpoints and single stepping. Figure 11.6 shows an iterative binary search algorithm. At the first look, the implementation of this algorithm gives the impression that it is perfect. It does what a typical binary search algorithm is supposed to do, checking for a mid value and dividing the searched list into two parts until the searched item is found or the searched sub-list has no more elements to be searched. However, in this particular implementation, marking the target sub-list might introduce an infinite loop as a result of integer division shown in line #10. UDB allows a user to insert a temporal assertion about any of these involved variables. This assertion may utilize the atomic agent named `old`, which traces the last value of the target variable. The temporal assertion checks the agent's value against the current variable value during the entire execution of this procedure. It will break execution at the very first incident found to violate the asserted expression.

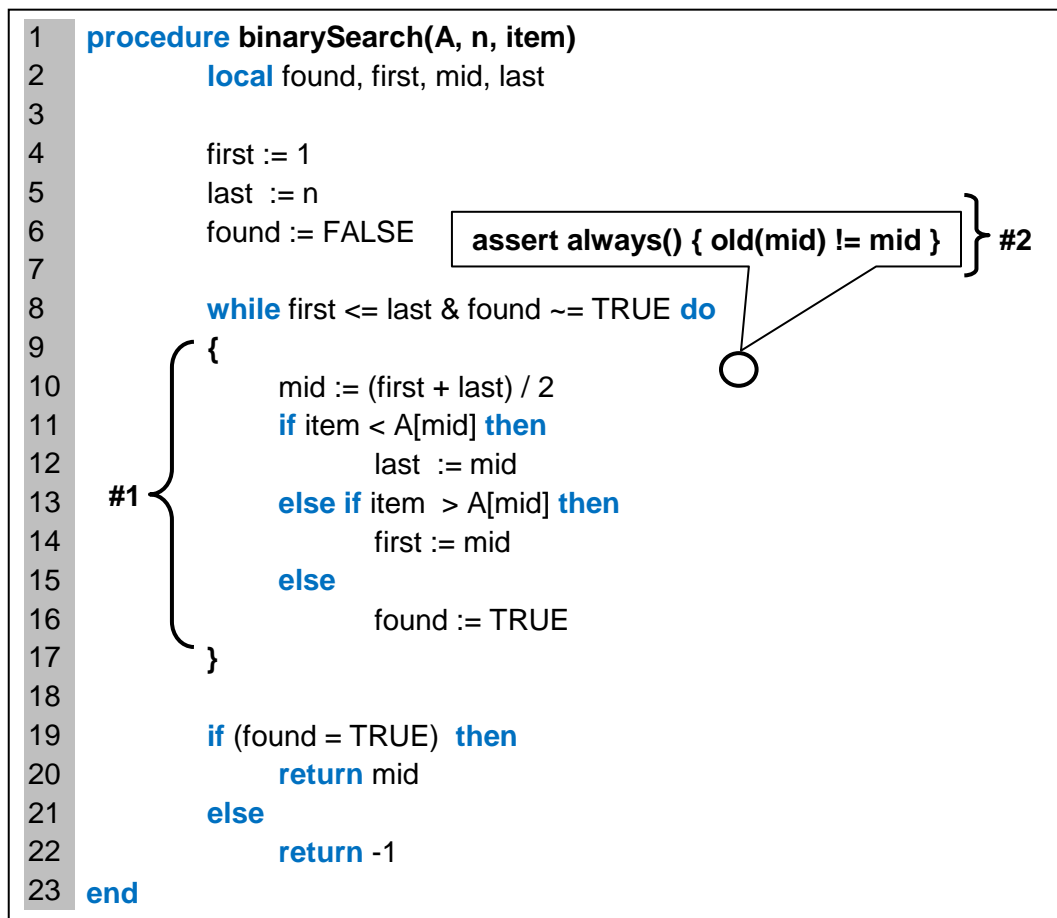


Figure 11.6. Using Temporal Assertions to Validate Infinite Loops

11.3.3. Example #3: Variables' State from Different Scopes

A debugging process may include checking variable values from different scopes, see Section 11.1. Figure 11.7 shows a program that prints out the prime numbers from 1 to some x . Function `main()` calls `isPrime()`, which returns true when the passed argument is a primary number. The temporal assertion provided in #1 shows how to check the current local value of variable `i` against the last value of variable `i` of function `main()` (`main:i`). This assertion assumes that the value of parameter `i` should not change during the execution of `isPrime()`. However, because the programmer is modifying the value of `i`, this assertion will evaluate to false at every change (temporal-state) to `i` in this `isPrime()` function, and it will evaluate to false at every return (temporal-interval) from this `isPrime()` function.

```

1  procedure main()
2      local x, i
3
4      writes(" Please enter a positive integer number : ")
5      x := read()
6
7      write("\n The following are the primary numbers <= ", x)
8      every i := 1 to x do
9          if isPrime( i ) then
10             write( i , " is a primary number ")
11      end
12
13  procedure isPrime( i )
14      local k
15
16      k := i
17      i -= 1
18
19      while ( i > 1 ) do
20          {
21              if k % i = 0 then
22                  fail
23              i -= 1
24          }
25      return k
26  end

```

} #1

assert always() { i == main:i }

Figure 11.7. Using Temporal Assertions to Check Variables from Various Scopes

11.4. Design

Temporal assertions do not replace traditional breakpoints or watchpoints, instead they provide a technique to reduce their number, which means they are used to reduce the number of execution stops and improve the overall process of investigation. These temporal assertions advance breakpoints with agents of temporal logic operators, see Section 11.5. At a stop, besides the source-level debugging functionalities, the user can delete, enable, disable, and modify existing assertions, or even insert new assertions at any location in the buggy program source code; all without the need to recompile the target program source code or to reload it under the debugger. UDB supports three *kinds* of Dynamic Temporal Assertions (DTA):

1. Past-Time Temporal Assertions
2. Future-Time Temporal Assertions, and
3. All-Time Temporal Assertions.

Each of these three kinds has its own temporal interval, see Section 11.4.1. These DTAs can reference execution properties and other internal extension agents such as the atomic data and behavioral agents discussed in Section 10.4 of the previous Chapter.

11.4.1. Temporal State

Each reached assertion has at least one temporal interval. This interval consists of a sequence of temporal states. Temporal states are defined based on the referenced execution object, which may reference execution behaviors, data flow, and control flow. For example:

1. Variables' temporal states are defined based on their assignments and/or references
2. Procedures' temporal states are defined based on their behavior such as call and return
3. Loops' temporal states are defined based on their number of iterations
4. Data structures' temporal states are defined based on their activities. For instance, a stack's temporal states are defined by its basic operations of `pop()` and `push()`, and a queue's temporal states are defined by its basic operations of `add()` and `remove()`.

11.4.2. Temporal Interval

Temporal interval is defined by the assertion *scope* and kind. Assertion's scope is defined based on the source code location provided in the `assert` command. This scope is the procedure or method surrounding the assertion location. Figure 11.8 shows the temporal interval for all three kinds of

temporal assertions in reference to the provided location. Together, the assertion's scope and kind define the temporal interval. In particular:

1. Temporal Intervals of Past-Time temporal assertions start at entering the assertion scope (calling the scope procedure) and end at reaching assertion's source code location for the very first time after entering the scope.
2. Temporal Intervals of Future-Time temporal assertions start at reaching assertion's source code location for the very first time after entering the assertion scope and ends at exiting the assertion scope (returning from the scope procedure). In this kind of temporal assertions, the source code location can be hit more than once before the interval is closed.
3. Temporal Intervals of All-Time temporal assertions start at entering assertion's scope and ends at exiting that scope; regardless of the provides source code location.

Figure 11.9 compares temporal intervals between all three kinds of temporal assertions and shows how these intervals relate to each assertion's source code location. Part A shows an example of a Past-Time temporal assertion, which has different temporal intervals in each hit. Parts B and C shows the temporal intervals for this assertion when it is used as a Future-Time assertion and All-Time assertion respectively. Each temporal interval consists of one or more temporal states; see Section 11.4.2. During a debugging session, it is possible for a user to have multiple assertions, each with multiple temporal intervals, and each interval with multiple temporal states. See Figures 11.9, 11.10, and 11.11.

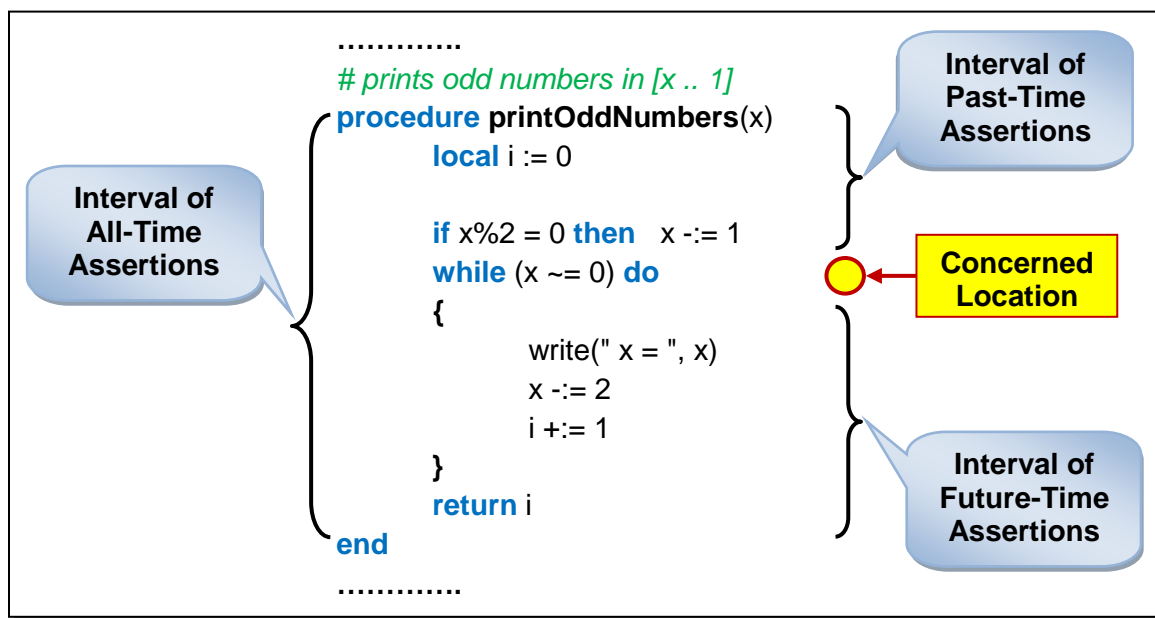


Figure 11.8. Temporal Assertions: Scope & Interval

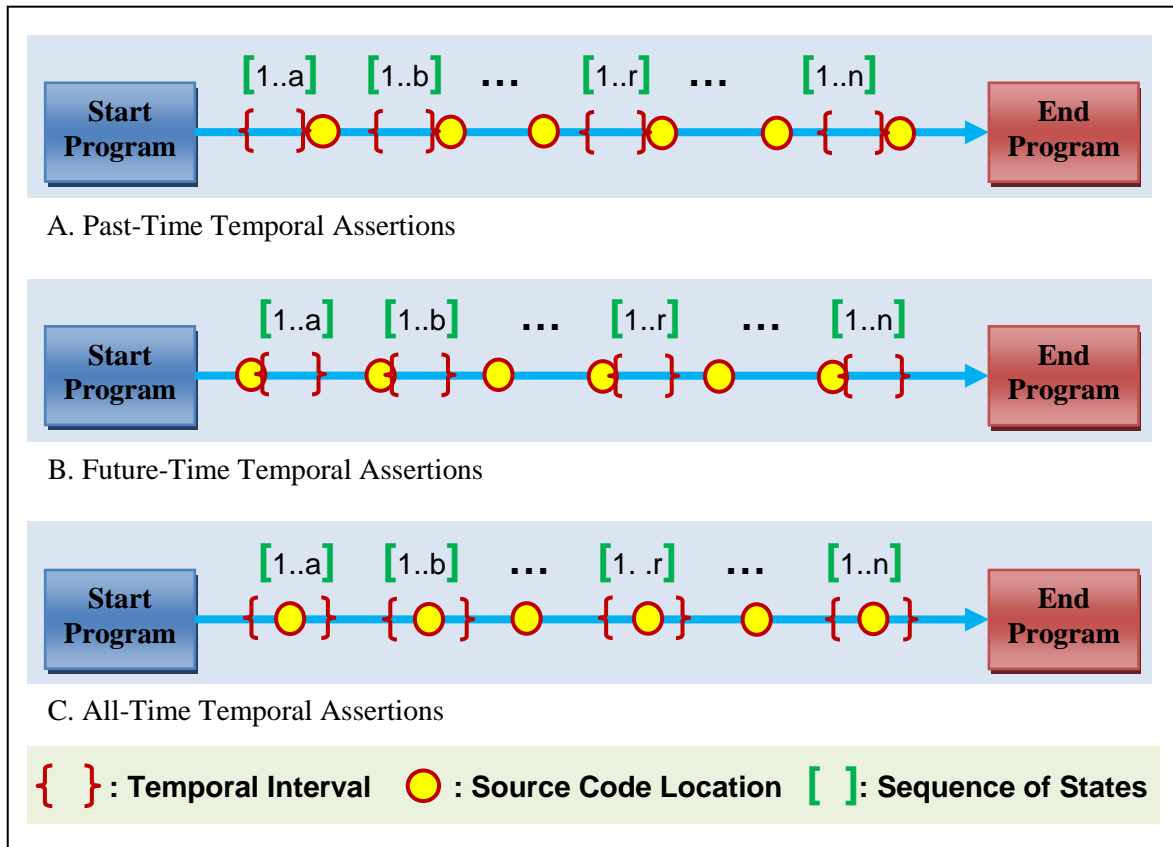


Figure 11.9. Temporal Assertions Evaluation

11.4.3. Assertion's Evaluation

UDB's temporal assertions are evaluated in the debugger as if they were part of the buggy program source code. By default, whenever an assertion evaluates to false, the source-level debugger stops execution in a manner similar to a breakpoint. The debugger transfers control to the user with an evaluation summary. Furthermore, the assertion's log gives the user the ability to review the evaluation behavior of each assertion. Each temporal assertion runs through three levels of evaluations:

1. State-based temporal level (single state change)
2. Interval-based (a sequence of consecutive states), and
3. Overall execution-based (a sequence of consecutive intervals)

Each assertion is evaluated based on 1) its temporal-state, which is triggered by any change to the assertion referenced objects, and 2) its temporal-interval, which is triggered by reaching the end of assertion's temporal interval. See Figures 12.10 and 12.11.

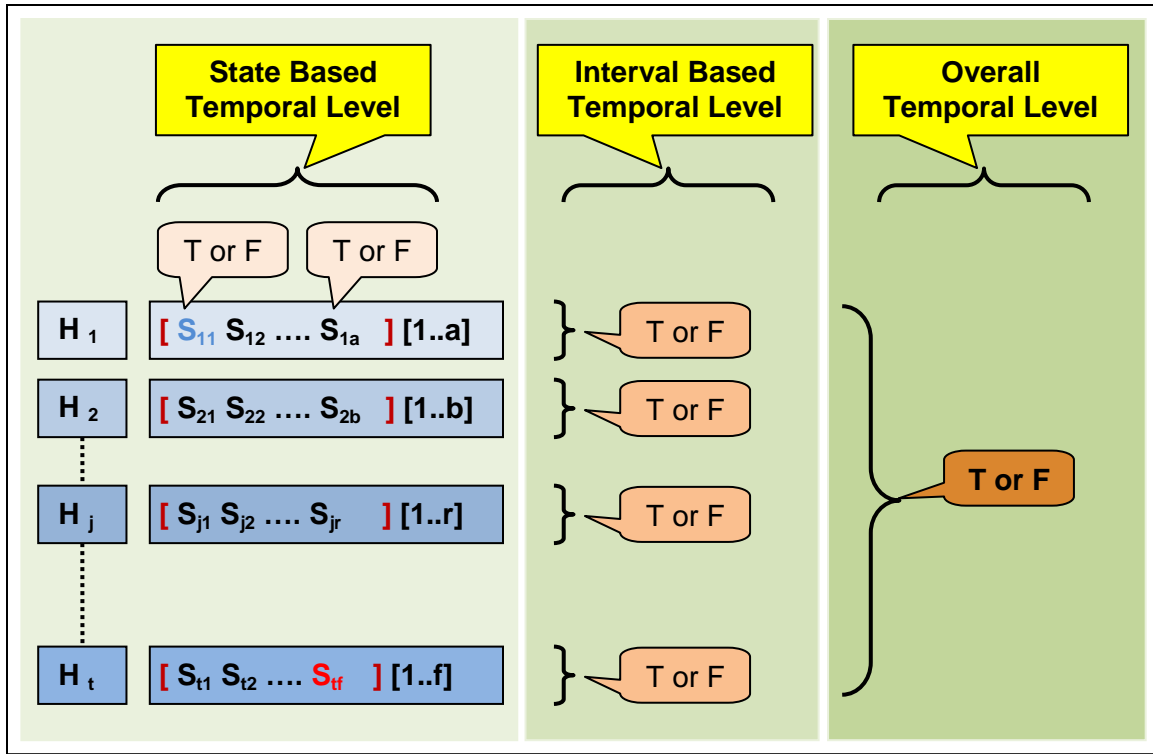


Figure 11.10. Sample Temporal Assertion's Evaluation

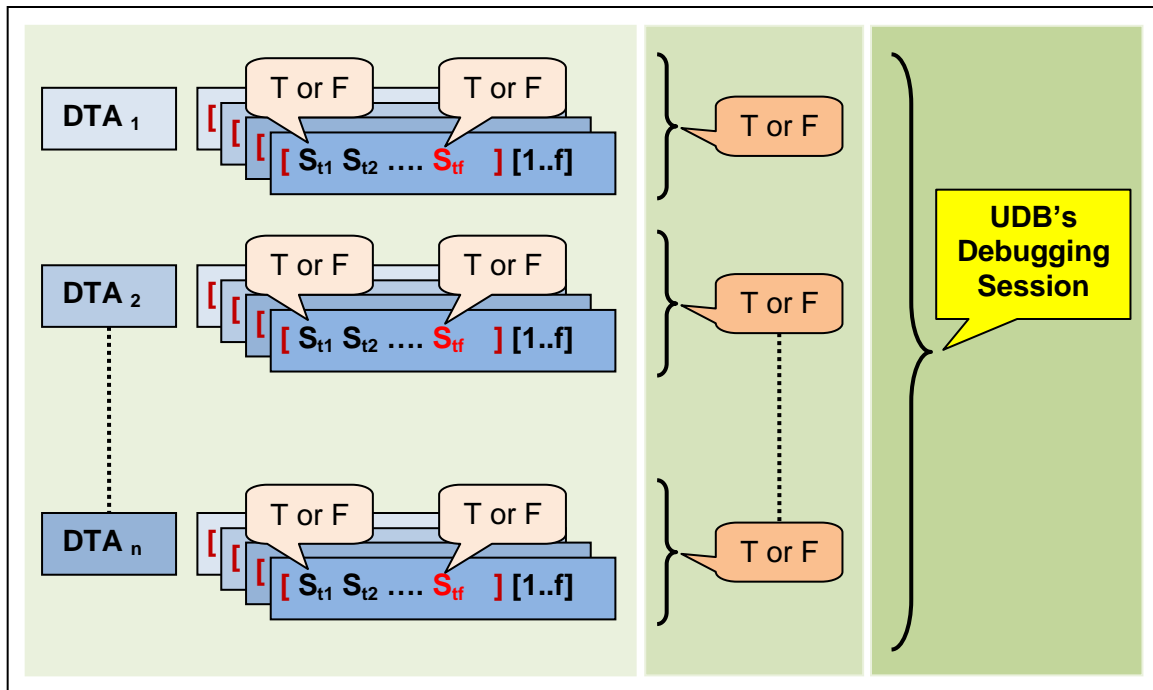


Figure 11.11. Sample Evaluation of Various Temporal Assertions

11.4.4. Temporal' Cycles and Limits

A temporal *cycle* is an integer that defines the maximum number of temporal intervals or maximum number of temporal level evaluation times. The default value of cycle is `&null`, which means to have unlimited evaluation. Temporal *limit* defines the maximum number of temporal states considered in each temporal interval. The definition of temporal limit is changed based on the kind of temporal assertion in reference. In particular:

1. In Past-Time temporal assertions, limit defines the maximum number of consecutive states before reaching assertion's source code location and after entering the assertion's scope
2. In Future-Time temporal assertions, limit defines the maximum number of consecutive states after assertion's source code location is reached and before exiting the assertion's scope
3. In All-Time temporal assertions, limit defines the maximum number of states before and after assertion's source code location is reached, all within the assertion's scope.

The Default limit is defined by whatever temporal states (temporal interval) are encountered during the execution of assertions' scope and based in its temporal interval. The user can reduce the number of temporal states considered in each temporal interval by setting this limit using the `limit` command.

Table 11.1. UDB's DT Assertions Evaluation Action Operators

Evaluation Action	Descriptions
hide	The evaluation will be hidden as long as it is <code>True</code> , otherwise it will break (default)
pause	The evaluation will be paused as long as it is <code>True</code> , otherwise it will break
show	The evaluation will notify the user with a printed message as long as it is <code>True</code> , otherwise it will break
stop	The evaluation will be stopped every time it evaluates, whether it is <code>True</code> or <code>False</code>

11.4.5. Evaluation Suite

UDB's DT assertions are supported with an evaluation suite of actions, see Table 11.1. These actions add automation and tracing flavors. The default evaluation action is to hide the evaluation result as long as it is true and only stop the execution when the evaluation result is false. However, the user can change the default evaluation action that is performed when the assertion evaluates to true. He/she may choose to `pause` after each evaluation for `N` seconds, or just to `show` that the assertion is

evaluated without pausing by printing an appropriate message. By default, the `pause` period is 5 seconds; however it can be changed by the user. During the `pause`, the user may hit a key that will change the `pause` to `stop` (breakpoint behavior). These evaluation suites provide two advantages:

1. The user will know that the program has reached the assertion point and the assertion's evaluation result is true; similar to the semantics of tracing with in-code print statements, and
2. Without the need to `stop` and `continue` the execution manually, the execution will resume automatically, unless the user interrupts and stops the execution for more investigation.

11.4.6. Temporal Assertions & Atomic Agents

Atomic agents are a special kind of internal extension agents. They expand the usability of DT assertions and facilitate the ability to validate more specific data and behavioral relationships over different execution states. When an atomic agent is used within an assertion, it retains and processes data and observes behaviors in relevance to the used assertions. The assertion scope is what determines when the agent should start to work and what range of data it should be able to retain and process. For example, if the assertion uses the `max(variable)` or `min(variable)` agents, the agent always retains the maximum or minimum respectively over the assertion temporal interval.

For more information, see Table 10.1 and Table 10.2 of Section 10.4 of the previous Chapter. Those atomic agents add more advancement and flexibility to the usefulness of DT assertions and their basic temporal logic operators. For example, DT assertions that reference atomic agents can easily check and compare data obtained by these atomic agents, which encapsulate simple data processing such as finding the *minimum*, *maximum*, *sum*, *number of changes*, or *average*.

In particular, suppose that a static variable is changed based on a conditional statement where it is incremented when the condition is true and decremented when the condition fails. What if the user is interested in the point at which this variable reaches a new maximum or minimum? DT assertions provide a simple solution for such situations. The assertion number 1 of Figure 11.12 will pause the execution when variable `x` becomes greater than or equal to `y`.

As another example, suppose the user is interested in the reasons behind an infinite recursion; perhaps a key parameter in a recursive function is not changing. DT assertions provide a mechanism to retain the parameter value from the last call and compare it with the value of the current call, see assertion number 2 of Figure 11.12. If `old(x) == current(x)`, the assertion will stop the execution and hand control to the debugger where the user can perform further investigation. Of course, there are

other reasons that may cause infinite recursion, such as the key parameter value is changing in the opposite directions on successive calls.

Moreover, DT assertions simplify the process of inserting assertions on program properties such as functions' return values, and loops' number of iterations. For example, a user may insert a breakpoint inside a function in order to investigate its return value, or place an in-code assertion on the value of the returned expression. A DT assertion provides a simpler mechanism; see assertion number 3 of Figure 11.12. Assertion number 4 of Figure 11.12 states that the `while` loop at line 50 in `test.icn` file always iterates less than 100 times. Finally, assertion number 5 of Figure 11.12 shows how to place a DT assertion on the number of calls to a function; the assertion will stop execution at call number 1000. This particular assertion is hard to accomplish using conventional source-level debugging features such as breakpoints and watchpoints.

1. (udb) **assert** test.icn:50 sometimep() { x < y }
2. (udb) **assert** test.icn:50 alwaysp() { old(x) != current(x) }
3. (udb) **assert** test.icn:50 alwaysf() { return(foo) > 0 }
4. (udb) **assert** test.icn:50 always() { iteration(while) < 100 }
5. (udb) **assert** test.icn:50 always() { call(baz) < 1000 }

Figure 11.12. Sample of Different DT Assertions

11.4.7. Evaluation Log

An assertion log allows a user to review the evaluation history. The debugger maintains a hash table for each assertion. It maps assertion's intervals into lists with information about their temporal state base evaluation. Each list reflects a temporal interval, which maintains the evaluation order and result for each temporal state. Each list reflects one temporal interval, which they are maintained based on their order too. Completely evaluated intervals are tagged with `True` or `False`. If the evaluation process is already started, but the final result is still incomplete, perhaps the end of the interval is not reached yet, these intervals are tagged with `Pending` until they are complete. This will convert `Pending` into `True` or `False`. See Table 11.2. However, some assertions may never be triggered for evaluation; this may occur because the execution never reached the assertion's insertion point during a particular run. These assertions have the hit counter set to zero.

Table 11.3. UDB's DT Assertions Evaluation Log

Evaluation Action	Descriptions
True	The evaluation is finished and the result is True (state-based & interval-based)
False	The evaluation is finished and the result is False (state-based & interval-based)
Pending	The evaluation is triggered but the interval is not complete yet, future information is still possible.

11.5. Assertion Language

DT assertions are applied using the `assert` command. Different assertions may utilize different Temporal Logic operators, each of which may utilize out scope objects and atomic agents. In order to evaluate the value of DT assertions within a typical source-level debugger, UDB has been extended with three kinds of internal agents performing various temporal logic operations. These agents work as temporal logic operators. The assertion language is comprised of a set of ten temporal logic operators. These agents are divided into three categories based on their temporal time and scope. Table 11.3 categorizes these temporal logic operators based on the three kinds of temporal assertions. The extended UDB enables programmers to add, delete, and modify ad-hoc DT assertions in the buggy program source code during debugging sessions. These assertions are capable of referencing variables beyond the scope of the assertion location and utilize information beyond the current state of execution.

Table 11.4. DTA Temporal Logic Operators

A. Past-Time Operators	Source Code Location	B. Future-Time Operators
alwaysp (cycle) { expr }		alwaysf (cycle) { expr }
sometimep (cycle) { expr }		sometimef (cycle) { expr }
since (cycle) { expr }		until (cycle) { expr }
previous (cycle) { expr }		next (cycle) { expr }
C. All-Time Operators		
always (cycle) { expr }		
sometime (cycle) { expr }		

11.5.1. Syntax

The syntax of DT assertion language consists of ten temporal logic operators, see Table 11.3. Each Temporal Logic operator consists of an integer cycle parameter and a body. The body may reference program variables, and objects, combined with any of the data and behavioral extension agents described in Table 10.1 and Table 10.2 of Section 10.4. Variables and atomic agents can be combined with any of the relational and propositional logic operators. See Figure 11.13.

DT assertions can reference execution properties such as variables, objects and their attributes, functions, methods, and control structures such as loops. If the referenced property is a variable from another function scope, it must be prefixed with the name of its function (i.e. `foo:variable`), whereas if that function is a method, the variable is prefixed with the method name prefixed with its class name (i.e. `class::foo:variable`). Of course, if that variable is a field of a record or an object, the dot operator is used (i.e. `object.variable`). Moreover, execution properties can be passed into any of the data and behavioral extension macros provided in Table 10.1 and Table 10.2 of Section 10.4.

(udb) assert location temporal-agent [(cycle, limit)] { expression } : true_behavior	
location	::= file-name:line-number procedure-name
temporal-agent	::= all-time-agent past-time-agent future-time-agent
all-time-agent	::= always(cycle) sometime(cycle)
past-time-agent	::= since(cycle) previous(cycle) alwaysp() sometimep()
future-time-agent	::= until(cycle) next(cycle) alwaysf() sometimef()
expression	::= agent operator agent
agent	::= literal variable atomic-agent
variable	::= variable-name procedure :: variable-name
operator	::= relational-operator propositional-logic-operator
relational	::= < > <= >= = !=
propositional-logic	::= &&(and) (or) ==> (implies)
cycle	::= integer
limit	::= integer
file-name	::= string
line-number	::= integer
true-behavior	::= hide show pause stop

Figure 11.13. UDB's Temporal Assertions Syntax

11.5.2. Past-Time Operators

This category consists of four Past-Time Temporal Logic Operators (agents); see Table 11.3, part A above. These operators utilize information retained between an *entering assertion's scope* and a *reaching assertion's source code location*. At insertion time, the debugger starts retaining relevant information to be used during the assertion's evaluation. When the execution reaches the virtual execution point, where the assertion is hooked in the buggy program space, the assertion temporal interval is evaluated. If the evaluation is not able to complete due to some missing information—maybe out-of-scope referenced data is never used during assertion's lifetime, the assertion evaluation is tagged with **Not Valid**. This category consists of four temporal assertions:

1. `alwaysp() { expression }` : asserts that expression must always hold (evaluate to true) for each, temporal state, temporal interval, and during the whole execution
2. `sometimep() { expression }` : asserts that expression must hold at least once for each temporal interval, and during the whole execution.
3. `previous() { expression }` : asserts that expression must hold right at the last state before the end of the temporal interval.
4. `since() { condition ==> expression }` : asserts that expression must hold right after condition is true up until the end of the temporal interval and for each interval.

11.5.3. Future-Time Operators

This category consists of four Future-Time operators; see Table 11.3. Part B. These operators utilize information retained between a *reaching of assertion's source code location* and a *leaving of assertion's scope*. The agents of those operators start watching for referenced objects when the evaluation is triggered, where the debugger starts retaining relevant information until assertion's temporal interval is evaluated completely. If the execution is terminated before assertion's interval is complete, the user is able to check temporal states in that incomplete temporal interval. This category consists of four temporal assertions:

1. `alwaysf() { expression }` : asserts that expression must always hold (evaluate to true) for each, state, temporal interval, and during the whole execution.
2. `sometimef() { expression }` : asserts that expression must hold at least once for each temporal interval, and during the whole execution.

3. `next() { expression }` : asserts that `expression` must hold right at the very first state in the temporal interval
4. `until() { condition ==> expression }` : asserts that `expression` must hold from the beginning of the temporal interval up until `condition` is true or the end of the temporal interval and for each interval.

11.5.4. All-Time Operators

This category consists of two All-Time operators; see Table 11.3, Part C. These two operators are based on the time interval between an *entering assertion's scope* and an *exiting assertion's scope*. When the assertion scope is entered, the assertion starts retaining relevant information and evaluates its temporal states. When the execution exits the assertion scope, the assertion temporal interval is evaluated. This category consists of four temporal assertions:

1. `always() { expression }` : asserts that `expression` must always hold (evaluate to true) for each, state, temporal interval, and during the whole execution
2. `sometime() { expression }` : asserts that `expression` must hold at least once for each temporal interval, and during the whole execution

11.6. Implementation

DT assertions are virtually inserted into the buggy program source code on the fly during the source-level debugging session. UDB's static information is used to assist the user and check the syntax and the semantic of the inserted assertion. Each assertion is associated with two sets of information 1) event-based and 2) state-based. The debugger automatically analyzes each assertion at insertion time in order to determine each set. It finds the kind of agents that are required to be encountered in the evaluation process. If any extension agent is used, the debugger establishes an instance of that agent and associates it with its relevant object.

The host debugger maintains a hash table that maps each assertion source code location into its related object (agent). The string format *file_name:line_number* is used as a key to access the assertion object in the hash table. The assertion object is responsible for maintaining and evaluating its assertion. It contains information such as 1) the parsed assertion, 2) a list of all referenced variables 3) a list with all temporal intervals and their temporal states, and 4) the assertion event mask, a set of event codes to be monitored for each assertion; this event mask includes the events mask for any of the referenced agents.

Execution events are acquired and analyzed in real time. Some events are used to control the execution whereas others are used to obtain information in support of the state-based technique. For example, an assertion is anchored in the execution state based on the `E_Line` event code, which is associated with the actual line number (event value). Other events such as the `E_Assign` and `E_Value` are used to watch and trace an assertion's referenced variables.

Each assertion has its own event and value masks, which are constructed automatically based on the assertion. A union set of all enabled assertion event masks is unified with the debugging core event mask. The result is a set of events requested by the debugging core during the execution of the buggy program. This set is recalculated whenever an assertion is added, deleted, enabled, or disabled. On the fly, UDB's debugging core starts asking the buggy program about this new set of events. A change on any assertion event mask alters the set of events forwarded by the debugging core to that assertion object.

Temporal logic agents automatically obtain the buggy program state-based information to evaluate DT assertions. Each agent automatically watches assertion referenced variables and retains their information in the debugger space. Some of the information is obtained through the values associated with the reported event code while others are obtained using AlamoDE high-level primitives.

UDB's Temporal Assertions are designed as extensions that utilize the IDEA architecture. There is an abstract class named `Assertion`, it is inherited by all of the ten other temporal logic operators. See Figure 11.14 for the UDB's class diagram for Temporal Assertions.

11.7. Summary

DT assertions bring an extended version of in-code assertion techniques, found in mainstream languages such as C/C++, Java, and C#, into a source-level debugging session. These temporal assertions help users test and validate different relationships across different states of the execution. Furthermore, assertion evaluation actions such as `show` and `pause` provide the sense of debugging and tracing using print statements from within the source-level debugging session. They give the user a chance to know that the execution has reached that point and the asserted expression evaluated to true; it also gives the user the ability to interrupt and stop the execution for more investigation. The ability to log the assertion evaluation result provides the user with the ability to review the evaluation process. A user can check a summary result of what went wrong and what was just fine.

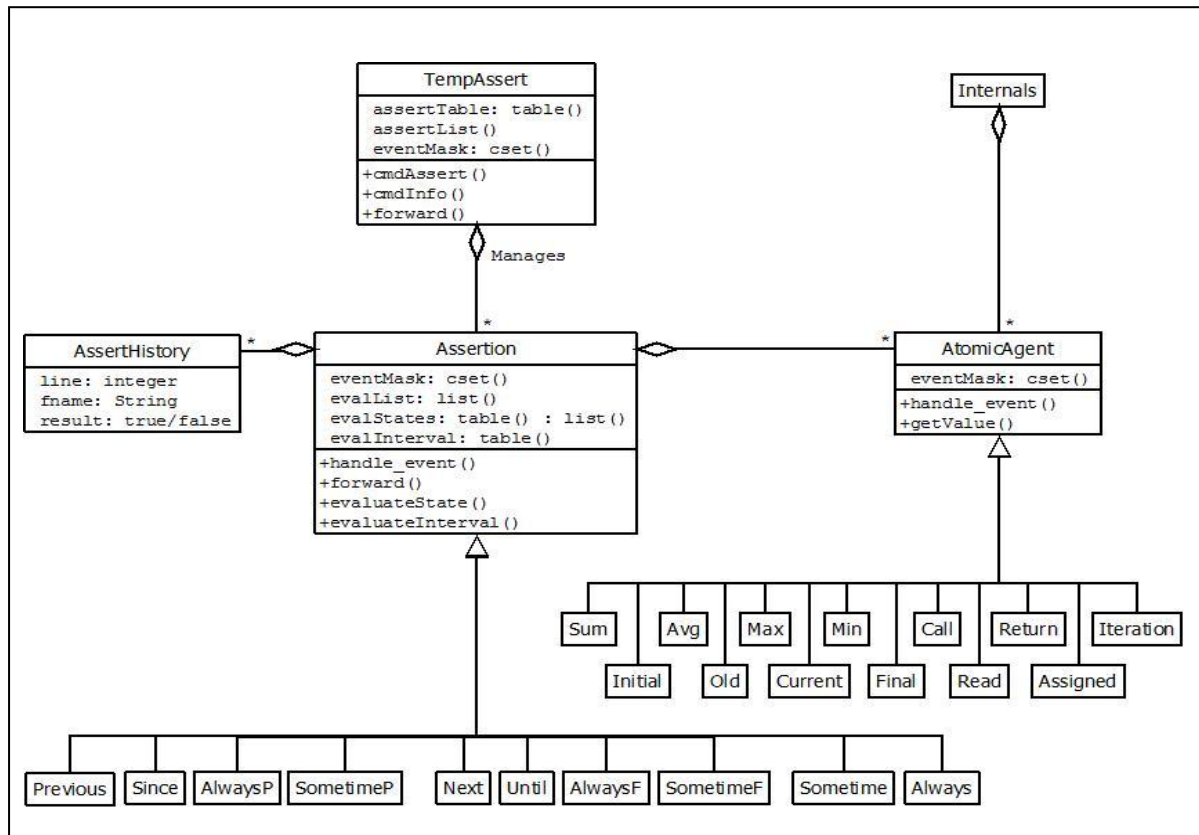


Figure 11.14. UDB's Temporal Assertions UML Diagram

Source-level debuggers provide the ability to conditionally stop the execution through different breakpoints and watchpoints. At each stop, a user will manually investigate the execution by navigating the call stack and variable values. Source-level debuggers require a user to come up with assumptions about the bug and let him/her manually investigate those assumptions through breakpoints, watchpoints, single stepping, and printing. In contrast, DT assertions require the user to come up with logical expressions that assert execution properties related to bug's revealed behavior and the debugger will validate these assertions. Asserted expressions can reference execution properties from different execution states, scopes, and over various temporal intervals. Furthermore, unlike conditional breakpoints and watchpoints, which only evaluate the current state, DT assertions are capable of referencing variables that are not accessible at evaluation time (not active in the current execution state).

DT assertions do not replace traditional breakpoints or watchpoints, but they offer a technique to reduce their number and improve the overall investigation process. DT assertions reduce the amount of manual investigation of the execution state such as the number of times a buggy program has to stop for investigation.

Part V

Evaluation and Results

Chapter 12

Performance and Evaluation

This chapter evaluates the primary contributions of this dissertation. First, before evaluating major extensions included in AlamoDE, it highlights Alamo's features and their advantages for event based debugging tools. Then, it discusses a new set of extensions that are needed to facilitate some of the debugging features. This chapter measures the effects of these extensions on both debugging tools and the Unicon language. Second, this chapter evaluates IDEA's features within UDB, including its internal and external agents. It highlights the simplicity of these extensions and their advantages and measures their performance. Finally, this chapter ends the evaluation discussion with a look at dynamic temporal assertions, which are introduced for the very first time in a typical source-level debugger for sequential programming.

The evaluation discussion is focused on capabilities and performance, which are considered a major step toward practicality. Part of the performance evaluation includes measuring the affected execution time. Unless indicated otherwise, seven different programs were considered during these experiments. These programs are `rsg`, `scramble`, `genqueen`, `ichartp`, `igrep`, `miu`, and `pargen`. See Appendix B for more details about these programs.

Moreover, unless indicated otherwise, the execution time is measured using the UNIX `time` command. It gives timing statistics about a specific program. The time is shown in three categories: 1) real time, which represents the elapsed time between the start of the process and its termination, 2) user time, which is the total number of CPU-seconds that the process spent in user mode, and 3) system time, which is the total number of CPU-seconds that the process spent in kernel mode. Moreover, these experiments were performed on Unicon version 11.6 running on a 32-bit Intel machine. This machine runs Linux open SUSE 11.0. It has the 1.6 T2050 core 2 Duo CPU and a 2 GB of RAM.

12.1. AlamoDE

AlamoDE inherits Alamo's implicit virtual machine instrumentation, which requires no special compilation and no source code or bytecode modification. It also provides an underlying mechanism to forward an event into another debugging tool that is loaded into the same virtual machine. This allows various debugging tools to share execution events. AlamoDE is used to build the extensible source-level debugger called UDB presented in Chapter 9. UDB integrates new automatic detection

techniques that can be found in trace-based debuggers such as ODB [10,13]. One measurement of AlamoDE's effectiveness is that UDB is a working prototype source-level debugger; it imitates most of GDB's functionalities with less than 10K lines of source code, including its IDEA architecture. This contributes as a proof by example to the value of AlamoDE as a debugging framework.

AlamoDE's 121 kinds of events and their relevant values provide ample information about the execution of the monitored program. AlamoDE's form of in-process debugging does not intrude into the execution of the target program space. At the same time, it has the advantage of providing direct access to the target program space through a set of high level primitives (built-in functions). If the monitor program requires information beyond the reported event code and value, it can employ these primitives. For example, variable values can be obtained using the `variable()` primitive, keyword values can be obtained using the `keyword()` primitive, and procedure values can be checked using the `proc()` primitive, see Chapters 5-7. This mixture of monitored events and direct access features allows complex communication patterns between the monitor program and the target program. For example, a monitor program may decide to further investigate the execution state of the target program based on a particular event code and value.

Experiment

Most monitored programs generate millions of execution events, which affect the scalability of the monitoring task in both time and space. Often, event-based monitors have to provide their own application level filtering mechanism. AlamoDE provides high level facilities to dynamically customize monitored events. For example, UDB's monitored events are adapted on the fly to the current active debugging features including its extension agents. AlamoDE provides two levels of event filtering mechanisms, the *event mask* and *value mask*. These masks are applied on event codes and their values respectively. They are checked by the instrumentation before events are reported to the monitor program [106, 107, 108].

Table 12.1 shows three programs `rsg`, `genqueen`, and `scramble`. Each of these programs is monitored for three different modes. The monitored modes are 1) all kinds of events; no event mask or value mask is used, 2) one kind of events specified by the event mask, but without utilizing the value mask; the monitored event is `E_Deref`, and 3) one kind of events with a specific monitored event code and value; this utilizes both of the event mask and value mask, the monitored event is `E_Deref` and its value is one of the dereferenced variables. Table 12.1 shows the number of reported events from each program and its monitoring mode. It also provides the corresponding average running time, each monitoring mode is observed for five different runs and the average time of these five times is calculated. The time is measured using the UNIX time commands. Figure 12.1 shows

the execution time of these three programs and compares it against the unmonitored version (standalone mode).

Table 12.1. AlamoDE No Mask vs. Event Mask vs. Value Mask

Program	Monitoring Mode	Number of Events	Real/S	User/S	Sys/S
rsg 0	No Monitoring	<i>none</i>	0.060	0.028	0.027
rsg 1	all events — <i>No Mask</i>	744817	2.765	1.756	0.977
rsg 2	E_Deref — <i>Event Mask</i>	61611	0.385	0.211	0.126
rsg 3	E_Deref — <i>Event Mask + Value Mask</i>	75	0.137	0.097	0.023
genqueen 0	No Monitoring	<i>none</i>	0.295	0.130	0.026
genqueen 1	all events — <i>No Mask</i>	5926941	22.072	13.287	8.611
genqueen 2	E_Deref — <i>Event Mask</i>	537410	2.425	1.600	0.751
genqueen 3	E_Deref — <i>Event Mask + Value Mask</i>	75546	0.876	0.718	0.136
scramble 0	No Monitoring	<i>none</i>	0.249	0.156	0.045
scramble 1	all events — <i>No Mask</i>	2125740	7.762	4.914	2.809
scramble 2	E_Deref — <i>Event Mask</i>	182162	0.962	0.658	0.288
scramble 3	E_Deref — <i>Event Mask + Value Mask</i>	15288	0.422	0.348	0.063
ichartp 0	No Monitoring	<i>none</i>	0.691	0.653	0.029
ichartp 1	all events — <i>No Mask</i>	31622681	58.934	11.519	47.277
ichartp 2	E_Deref — <i>Event Mask</i>	2108995	9.202	6.354	2.818
ichartp 3	E_Deref — <i>Event Mask + Value Mask</i>	110921	2.807	2.571	0.206
igrep 0	No Monitoring	<i>none</i>	0.108	0.070	0.015
igrep 1	all events — <i>No Mask</i>	1790572	6.441	4.120	2.301
igrep 2	E_Deref — <i>Event Mask</i>	195307	0.902	0.570	0.306
igrep 3	E_Deref — <i>Event Mask + Value Mask</i>	1006	0.280	0.229	0.027
miu 0	No Monitoring	<i>none</i>	1.340	0.672	0.042
miu 1	all events — <i>No Mask</i>	4550772	17.002	10.591	6.349
miu 2	E_Deref — <i>Event Mask</i>	212421	1.999	1.242	0.358
miu 3	E_Deref — <i>Event Mask + Value Mask</i>	24	1.549	0.874	0.065
pargen 0	No Monitoring	<i>none</i>	0.028	0.011	0.015
pargen 1	all events — <i>No Mask</i>	27193	0.163	0.098	0.066
pargen 2	E_Deref — <i>Event Mask</i>	1351	0.044	0.020	0.018
pargen 3	E_Deref — <i>Event Mask + Value Mask</i>	16	0.033	0.012	0.018

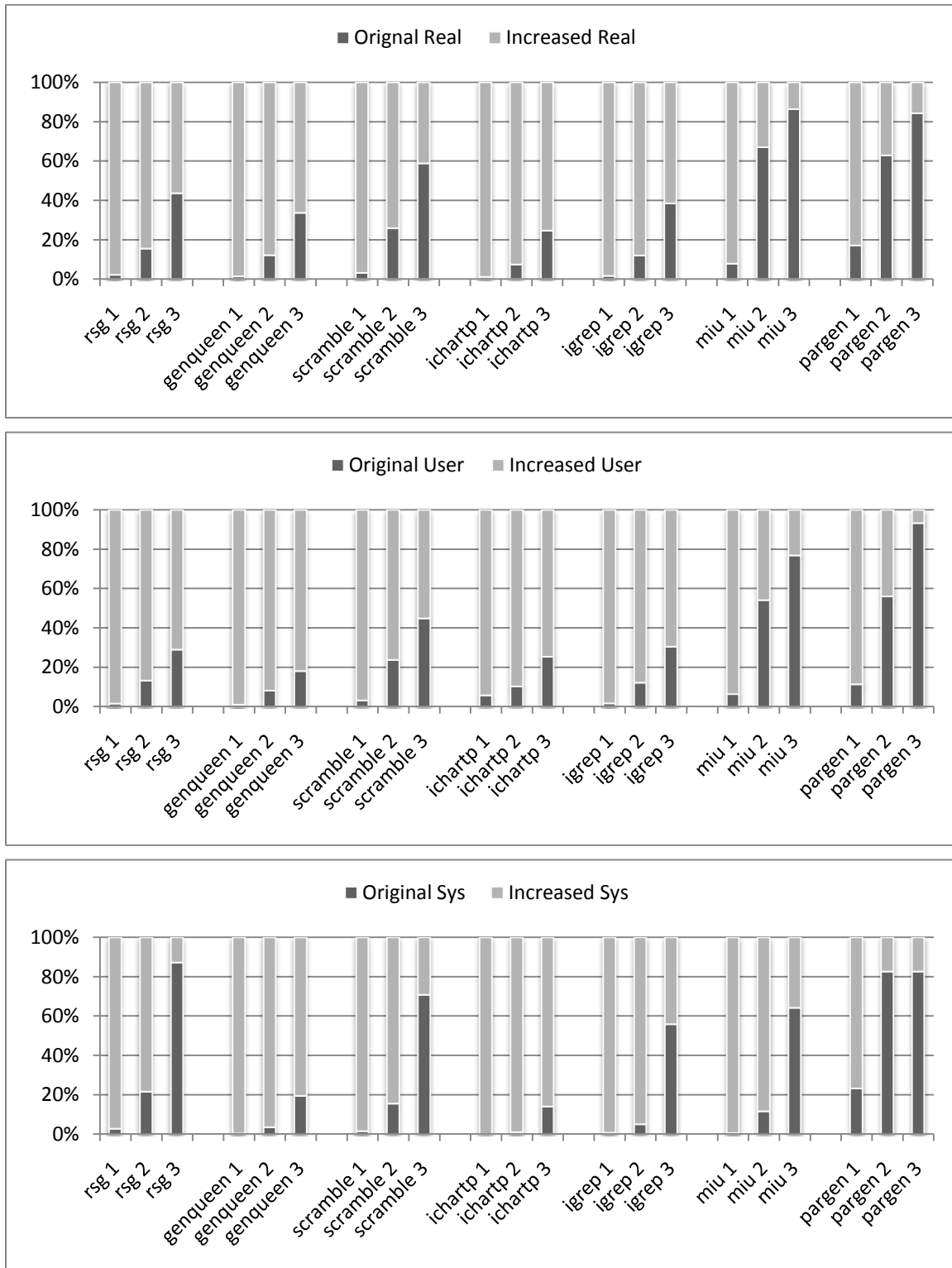


Figure 12.1. Execution Time- Standalone vs. Monitored Mode

12.2. Alamo's New Extensions

This section provides an evaluation for some of the new extensions implemented as addition to the original Alamo framework for debugging support. It evaluates two of the most important additions that have their effects on the Unicon language and the debugging process. The evaluation targets the performance of trapped variable implementation and compares it to original Alamo's variable reading mechanism. It also evaluates the syntax instrumentation and its impact on the size of both the compiled object program and the executable binary format.

12.2.1. Trapped Variable Assignment

Since both the monitor and target program share the same address space, the monitor program has almost direct access to the space of the target program. This access method is facilitated through high level primitives supported by Alamo and its AlamoDE support within Unicon's virtual machine. The `variable()` function can be used by the monitor program to read and write target program's variables—local and global names. When this `variable()` function is used to read a value, there is no overhead induced on the space by this trapped block, it only acquires a copy of the variable value being read. But, if this function is used to change the value of the target program's variable, then for security reasons, instead of directly referencing this target variable, a block of trapped variable is implicitly allocated.

This block points to the target variable and contains two more fields; one contains the title of the block to be distinguishable by the internal implementation of the virtual machine, and the other is an integer counter used to validate the number of context switches between the time at which the reference is obtained and the time at which the final value is written. This block is allocated with every assignment to the target program space. The size of this block depends on the target machine. For example, on an Intel 32-bit machine, the size of this block is 12 bytes, whereas it is doubled on an AMD 64-bit machine. After the assignment is complete, this block becomes garbage and is cleaned by the language garbage collector.

12.2.2. Syntax Instrumentation

Syntax instrumentation is implemented to provide syntax information to the monitor program upon its request. This syntax information is inquired through monitoring the `E_Syntax` event or through a direct access to the `&syntax` keyword. Direct access to a keyword allows the monitor program to request the currently executed syntax name at any point, whether the syntax event is being monitored or not. In contrast, monitoring the `E_Syntax` event entails that the virtual machine

instrumentation should include a new event that is reported to the monitor program at the start and finish of each major syntax construct.

The implemented technique requires the ability to make the syntax information available to Unicon's runtime system. This was achieved based on an already available line and column number table that is included in the executable bytecode. Each entry in this table is based on two special object code commands `line` and `colm` introduced by the translator **icont** and written into the compiled object code file (ucode). Then these two commands are assembled by the linker based on two pseudo virtual machine instructions `Op_Line` and `Op_Colm` respectively. The result is one sparse table that maps Interpreter Program Counters (IPCs) into relevant line and column numbers found in the actual compiled source code. See Section 6.3.

Originally, this table was only used to provide source code information for runtime errors and tracing facility. The first part in syntax instrumentation is augmenting this table with additional syntax information. The layout of this table is not changed except for the 5 bits taken from the original 16 bits column, reducing it to 11 bits. However, since this table is sparse, it does not provide a one-to-one map from each source code location to its relevant IPCs. These original entries were found unable to provide sufficient and precise syntax information. A set of new entries were added to mark monitored syntax constructs such as major control statements and loop structures. The implementation of this syntax instrumentation affects Unicon programs in four ways.

1. The effect on the size of the object code as a result of the new `synt` object code command instruction used to mark the syntax code in the compiled object code.
2. The effect on the linking time used to assemble various object files into one bytecode executable. The Unicon linker is extended with new pseudo virtual machine instruction named `Op_Synt`, which is used to read the `synt` command and insert the syntax code along with the already provided line and column number in to the table.
3. The effect on the size of the executable as a result of new entries in the line/column number table since the original entries were error oriented and not intended to monitor syntax information, which required inserting new entries surrounding major syntax constructs. In addition to what is already in the table, few numbers of entries were added to mark entering and exiting major syntax constructs such as control statements.
4. The effect on the execution time of the monitored program as a result of the new `E_Syntx` event that occurs whenever a major syntax constructs starts or finishes.

The first three problems are general ones. They affect all Unicon programs unless the virtual machine is built with the `NoSrcSyntaxInfo` defined in `define.h`. This macro disables syntax instrumentation support—i.e. `#define NoSrcSyntaxInfo`. The fourth point affects only the monitored programs, especially when they include the `E_Syntax` event in their event mask.

The reporting frequency of this new `E_Syntax` event depends on the target program. However, it is been monitored along with the `E_Line` and `E_Deref` events that are reported for new lines and variable dereferencing respectively. These events were monitored in `rsg`, `scramble`, `genqueen`, `ichartp`, `igrep`, `miu`, and `pargen`. The ratio of these reported events is shown in Figure 12.2.

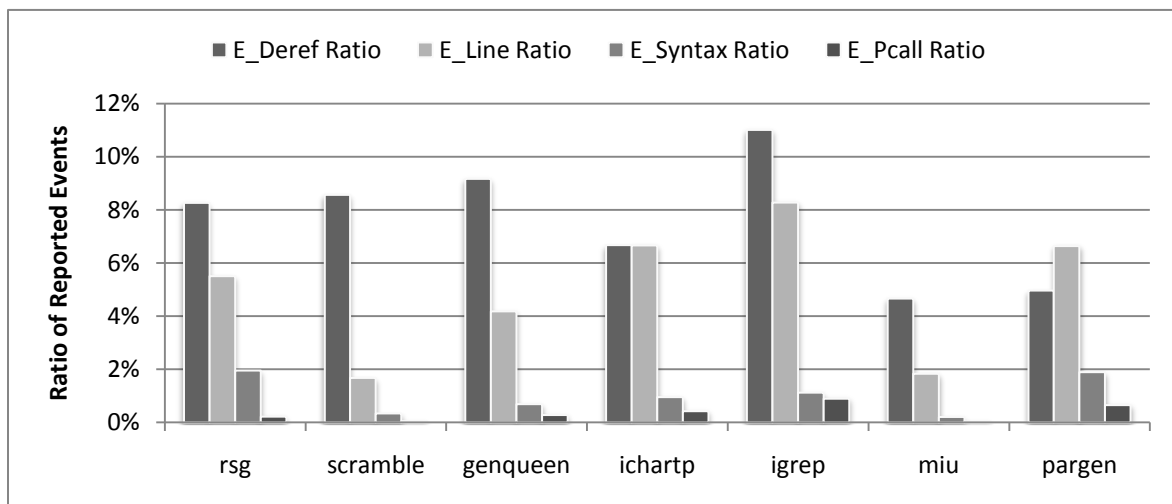


Figure 12.2. E_Deref, E_Line, E_Syntax, & E_Pcall Events Ratio to all Other Events

In order to evaluate these effects, six different programs were measured, before and after the syntax instrumentations, based on three categories: 1) the size of the bytecode, 2) the size of the executable, and 3) the compiling and linking time. Table 12.2 shows the impact of syntax instrumentation on the compiled object code. It measures the size of six object files before and after the syntax instrumentation on an Intel 32-bit machine. Then, it shows the amount of increase in the object code size imposed on each one of these files in kilobytes. Table 12.3 shows the impact of this new syntax instrumentation on the binary executable. It measures the sizes of six different executables before and after the syntax instrumentation and finds the difference in kilobytes. In Table 12.2, the first three object files are linked directly into the executable provided in Table 12.3 without any other user or library files being involved. The last three files are big programs; they were linked from several ucode files. Figures 12.3 and 12.4 shows the percentage of these increases and compares them to the original sizes. Table 12.4 shows the compiling/linking time before and after the syntax instrumentation, whereas Figure 12.5 shows the percentage increase in these times.

Table 12.2. Syntax Instrumentation Effects on Object-Code (ucode) Formats

File Name	Size Before Syntax/KB	Size After Syntax/KB	Difference/KB
rsg.u	30.28	35.77	5.49
scramble.u	4.97	5.96	1.00
genqueen.u	4.20	5.01	0.81
unicon.u	89.23	103.79	14.56
ivib.u	319.26	365.79	46.52
ui.u	77.08	84.96	7.88

Table 12.3. Syntax Instrumentation Effects on Executable (bytecode) Formats

Program Name	Size Before Syntax/KB	Size After Syntax/KB	Difference/KB
rsg	17.03	17.23	0.20
scramble	3.28	3.29	0.01
genqueen	2.86	2.88	0.02
unicon	678.90	684.44	5.54
ivib	392.42	398.70	6.28
ui	590.15	600.87	10.72

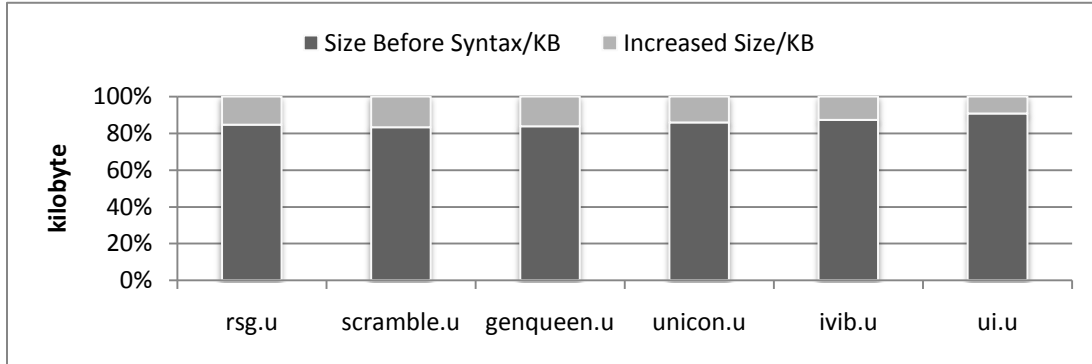


Figure 12.3. The Percentage Increase in the Size of the Object Code File

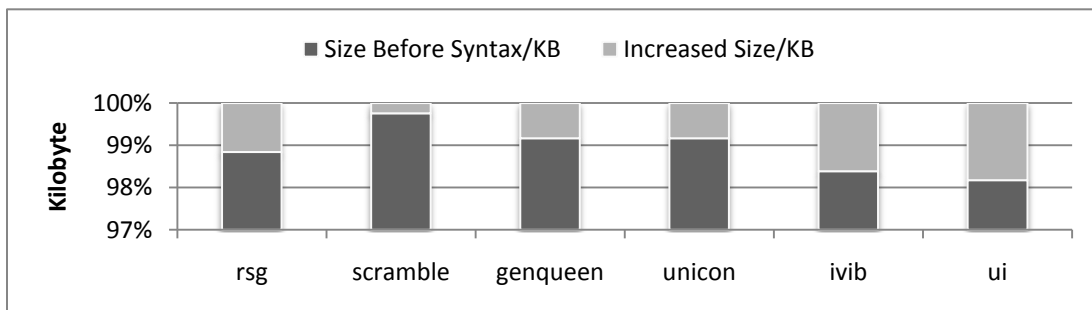


Figure 12.4. The Percentage Increase in the Size of the Executable Program

Table 12.4. Syntax Instrumentation Effects on Compiling/Linking Time

Program Name	Before Syntax Instrumentation			After Syntax Instrumentation		
	Real	User	Sys	Real	User	Sys
rsg	0.035	0.0272	0.0072	0.0378	0.0288	0.0088
scramble	0.015	0.0088	0.0048	0.0156	0.0096	0.0056
genqueen	0.0148	0.0048	0.0072	0.015	0.008	0.0076
unicon	0.198	0.176	0.0176	0.2224	0.1912	0.024
ivib	3.3344	3.2368	0.072	3.5148	3.4096	0.076
ui	3.8236	3.6384	0.16	3.905	3.6776	0.1784

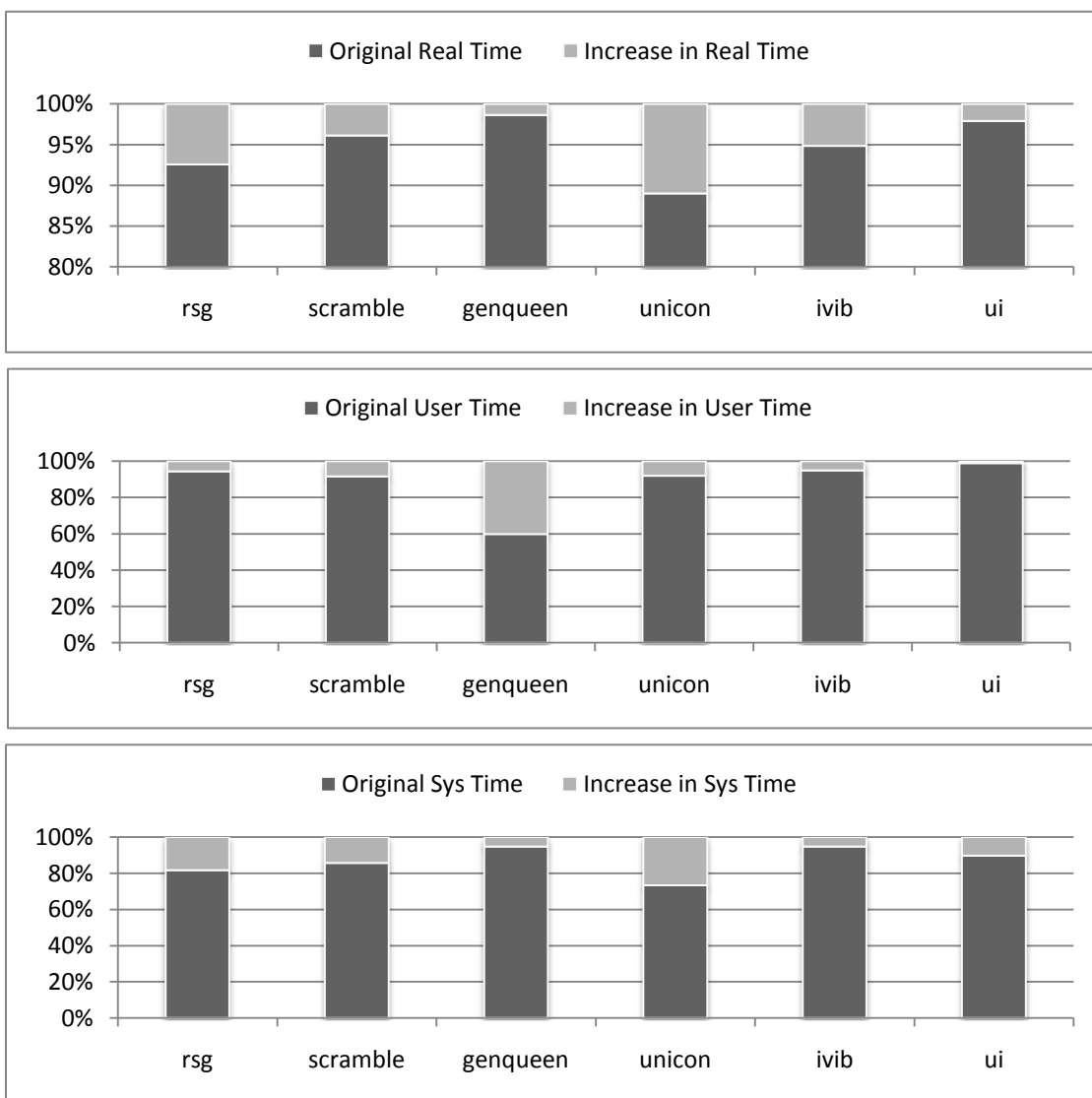


Figure 12.5. The Percentage Increase in Compile/Link Times

12.3. IDEA's Evaluation

The ability to easily extend a debugging tool is very important, because there is no debugging tool good enough to debug all kinds of bugs. Extensibility simplifies the task of improving these tools with new techniques. An IDEA-based debugger allows different debugging tools (extension agents) to simultaneously debug a program during the same debugging session (same run). IDEA's core is a mediator that coordinates various extension agents. However, one of the biggest considerations in this type of design is performance. In IDEA, a considerable amount of time is spent on:

1. Processing the instrumentation in the target program
2. Performing context switches between the target program and IDEA's debugging core
3. Filtering the received events in the monitor program (the main debugging tool)
4. Forwarding events from IDEA's core to extension debugging agent(s).

12.3.1. Procedure Call vs. Co-Expression Context Switch

Although co-expression context switches are lightweight and managed in-process without the knowledge of the operating system, they are still costly; one of the reasons is their high occurrence rate. Migrate extension agents to internal or use them in a call-back mode will enhance the overall performance. In order to measure the gained processing speed for migrated agents, this dissertation starts with an experiment that measures the basic difference in time between procedure calls and co-expression context switch.

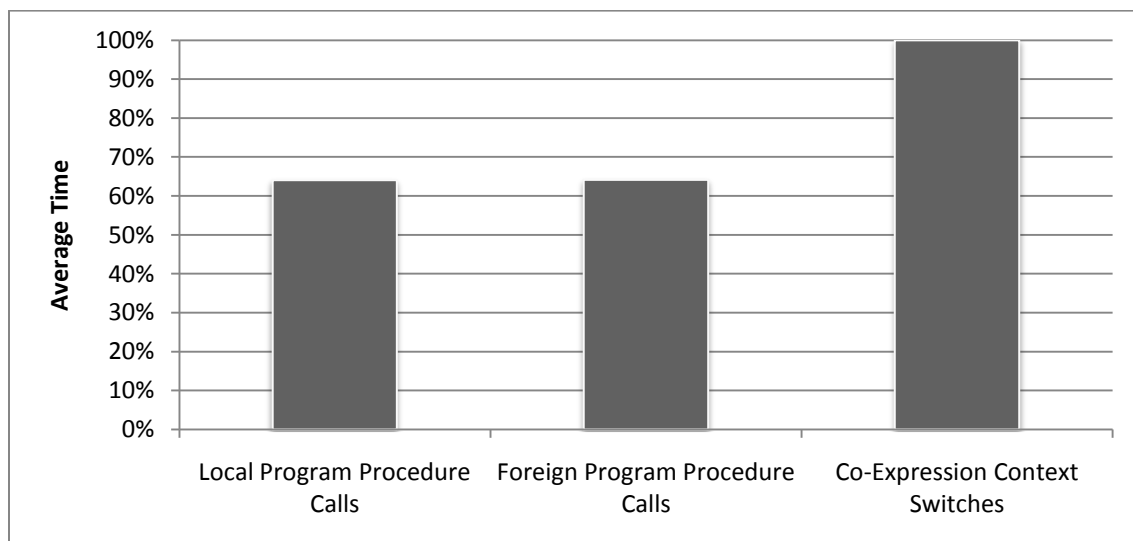


Figure 12.6. Time of Procedure Calls vs. Context Switches

Figure 12.6 compares the difference in time between local procedure calls, foreign procedure calls, and co-expression context switches. The comparison shows that procedure calls (both local and foreign) reduces the processing time by about one third. Data is collected from a very simple Unicon program provided in Figure 12.7. This program is used in 10 different runs, and the average time of these runs is used in Figure 12.6. The Unicon keyword `&time` is used to measure the elapsed CPU time in milliseconds.

```

1  # A simple program test the basic difference in time between local procedure calls,
2  # foreign procedure calls, and co-expression context switches.
3  $define NUM 10000000
4  procedure main(argv)
5      local t1, t2, t3, ce, foreign_prog, foreign_proc
6
7      t1 := &time
8      every i := 1 to NUM do      p()
9      write("The cost of local procedure calls : ", &time - t1, "ms.")
10
11     foreign_prog := load(argv[1])
12     foreign_proc := variable("pp", foreign_prog)
13     t2 := &time
14     every i := 1 to NUM do      foreign_proc()
15     write("The cost of foreign procedure calls : ", &time - t2, "ms.")
16
17     ce := create | 1
18     t2 := &time
19     every i := 1 to NUM do      @ce
20     write("The cost of context switches : ", &time - t3, "ms.")
21 end
22 procedure p()
23     return 1
24 end

```

Figure 12.7. Sample Unicon Program Measures Procedure Calls vs. Co-Expression

12.3.2. Extension Agents

Originally, external extension agents were used through context switches. For events that are forwarded to external debugging agents, two extra context switches are added to the debugging cost. There is a total of four context switches. The first two are between IDEA's debugging core and the target program, and the second two are between the debugging core and the external agent. If more than one agent requests the same reported event code, then each one of those agents will add another two context switches.

For instance, if there is a total of m agents loaded under the IDEA based source-level debugger, and n of these m agents request a specific event code; note that $n \leq m$. The total number of context switches for this particular event is $2*n+2$. This number is repeated each time this event is reported. During a debugging session (monitoring task), if this event is reported E number of times, then there is a total of N_c context switches during this session, see Equation 12.1.

$$N_c = E * (2*n + 2)$$

Equation 12.1. Number of Context Switches (E Events Reported to n Agents)

If E_c is the cost of each one of these context switches, where little c stands for the context switch, then the cost of reporting this event to these n agents is C_c , and the total cost of reporting this event during the session is TC_c , both of which are shown in Equations 12.2 and 12.3 respectively. TC_c depends on three variables E , E_c and n . UDB and its IDEA architecture utilizes AlamoDE's direct access and event filtering mechanisms to reduce these two factors to the least minimum possible based on the current state of AlamoDE.

$$C_c = E_c * (2*n+2)$$

Equation 12.2. Cost of Forwarding an Event to n Agents using Context Switches

$$TC_c = E * C_c \rightarrow TC_c = E * (2E_c*n + 2E_c) \rightarrow TC_c = (E * 2E_c * n) + (E * 2E_c)$$

Equation 12.3. Total Cost of Forwarding E Events to n Agents using Context Switches

In contrast, when these n agents are used in the standalone mode (directly monitoring the target program—IDEA is not involved), the user must run them separately; one at a time. This eliminates the ability to compare their outcomes from within the same debugging session and precludes their potential for collaborations. However, the cost of reporting this event to an agent runs in the

standalone mode is $2E_c$. That is because reporting an event requires a total of two context switches. The total cost of reporting this event for each agent is C and the total cost of reporting this event to all n agents is TC , see Equations 12.4 and 12.5 respectively.

$$C = E * 2E_c$$

Equation 12.4. Cost of Reporting an Event to an Agent in Standalone Mode

$$TC = (E * 2E_c) * n$$

Equation 12.5. Total Cost of Reporting an Event to n Agents in Standalone Mode

Comparing TC_c and TC from Equations 12.3 and 12.5 respectively, this comparison shows that TC is less than TC_c and the difference is C . This means that running these n agents in the standalone mode saves a total of $E*2E_c$, which is the cost of reporting an event to one of these standalone agents. If you would consider UDB as an extra tool, not just a coordinator, then its cost is justified. However, even though utilizing these agents under a source-level debugger cost the performance a total of C , the user gains the advantage of synchronous execution and the ability for these agents to collaborate with each other during the same debugging session.

The previous discussion was about these extension agents that are loaded on the fly during the debugging session and used through co-expression context switches. When an agent migrates to internal and is used through the inter-program procedure calls, the user gains extra performance. Section 12.3.1 shows that replacing context switches with local or foreign procedure calls lowers the overhead imposed by these context switches by approximately one third. In other words, the cost of 2 context switches is equivalent to the cost of 3 procedure calls (local or foreign).

This means, replacing context switches with procedure calls will reduce E_c to E_p , where little p stands for procedure calls. $E_p \approx 2/3 E_c$. This will change the formula presented in Equation 12.3. The new formula is presented in Equation 12.6. This equation retains the second part ($2E_c * E$) because there is one remaining context switch between the target program and UDB. Figure 12.8 shows these notations in the three scenarios and compares them. Part A shows the use of these agents in the standalone mode, whereas parts B and C compare these agents when they are used in context switches vs. procedure calls.

$$C_p = (2E_p * n + 2E_c) * E \rightarrow C_p = E * 2(E_p * n + E_c) \rightarrow C_p = E * 2(2/3E_c * n + E_c)$$

Equation 12.6. The Cost of Reporting an Event to n Agents Using Procedure Calls

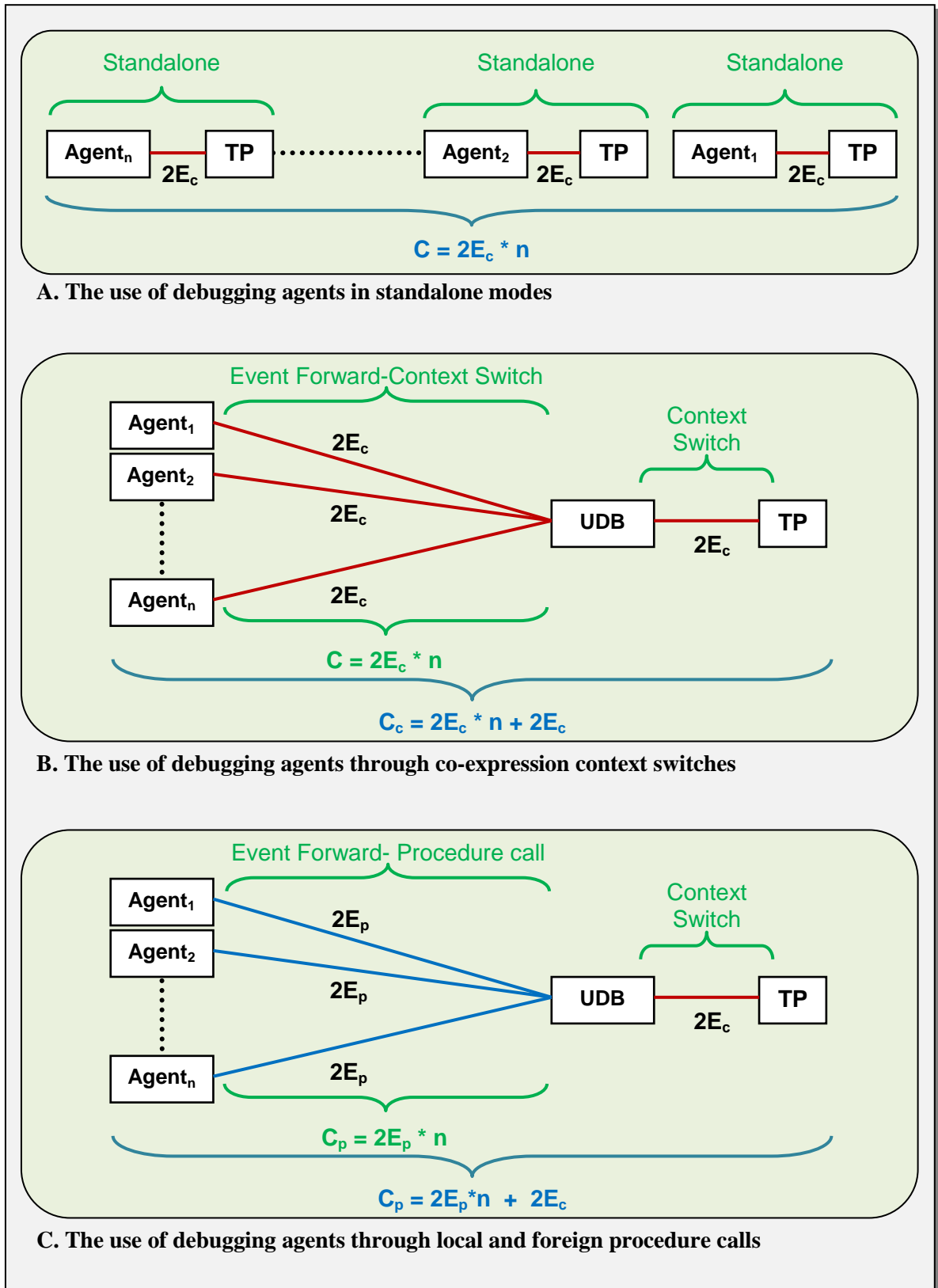


Figure 12.8. IDEA's Use of Debugging Agents

12.3.3. Experiment

Earlier in Section 12.3.2, the Equations 12.3, 12.5, and 12.6 theoretically compared the use of extension agents through context switches, in the standalone mode, and through procedure calls respectively. Now, in order to find in practice the amount of slowdown imposed by a debugging agent that is running as a separate tool, or as an extension agent under IDEA, a simple agent is used to monitor a very frequent event, `E_Deref`. This event occurs whenever a variable is dereferenced inside the target program. The agent is kept simple; its computation is limited to counting the total number of monitored events, see Figure 12.9. This allows us to measure the event forwarding technique without worrying about the algorithm implemented within the agent itself.

In order to find the effect of this agent on the debugging process, six different experiments were performed. In each experiment the same three target programs are used; each program is monitored for five different runs and the average time of these five runs is measured. The first program is `rsg`. The second program is `scramble`. The third program is `genqueen`. Table 12.5 shows the measuring time of these six experiments. The time is measured using the UNIX command `time`, which produces the *real*, *user*, and *sys* times. These times are presented in seconds. These six experiments were performed on the same machine used in Section 12.3.1.

The first experiment is used to find the running time of these three programs when they are used without any monitoring; see Table 12.5 row #1. The second experiment is used to find the impact of UDB on these three programs. Each of these programs was loaded and run under UDB; the session neither enables any of the extension agents nor does it apply any of the classical debugging commands, see Table 12.5 row #2. The third experiment is to find how much time the simple standalone agent slows down the execution of these programs. The experiment runs the agent in the standalone mode, which loads, runs, and monitors these three programs, see Table 12.5 row #3.

The fourth experiment is to find how much time the same agent takes if it is used in the method call approach under IDEA. The same three programs were used as target programs under UDB, see Table 12.5 row #4. The fifth experiment is to find how much time the same agent takes if it is used as external agent under UDB through the inter-program procedure call. The same three programs were used as target programs under UDB, see Table 12.5 row #5.

```

1  $include "evdefs.icn"
2  $ifndef StandAlone
3  class EventCounter : Listener(
4  $else
5  class EventCounter(
6      eventMask, # an event mask
7  $endif
8      count    # count the number of events
9  )
10 method handle_E_Deref()
11     count += 1
12 end
13 method handle_E_Exit()
14     write("Total # of events is : ", count)
15 end
16 initially(name, state)
17 $ifndef StandAlone
18     self.Listener.initially(name, state)
19 $endif
20     count := 0
21     eventMask := cset (E_Deref || E_Exit)
22 end
23
24 # StandAlone is defined when this tool is used as a standalone monitor.
25 # otherwise, this tool can be statically linked into the main udb source code
26 $ifdef StandAlone
27 link evinit
28
29 # This main() procedure is only used in the standalone mode
30 # or udb's external co-expression mode
31 procedure main(tp)
32     local mask, obj
33     EvInit(tp) | stop(" cannot initalize target program" || tp[1])
34     obj := EventCounter()
35
36     while EvGet(obj.eventMask) do
37         if &eventcode == E_Deref then obj.handle_E_Deref()
38         else                          obj.handle_E_Exit()
39     return
40     handle_Events()
41 end
42
43 # This procedure is only used by the inter-program procedure calls
44 procedure handle_Events(code, value)
45     static obj
46     initial{ obj := EventCounter()
47             return obj.eventMask }
48     &eventcode := code
49     &eventvalue := value
50     if &eventcode == E_Deref then obj.handle_E_Deref()
51     else                          obj.handle_E_Exit()
52 end
53 $endif

```

} *The Agent class*

} *Agent's procedure main() used in the standalone mode*

} *Agent's interface for inter-program procedure calls*

Figure 12.9. Sample Agent Counter

Finally, the sixth experiment is to find how much time the same agent takes if it is used in the external approach through the context switch event forwarding method. In each run, the method `cmdLoad()` is called once to load that agent. Then the method `Forward()` is called frequently in IDEA's evaluator loop to forward events to this agent. This method's underlying implementation checks whether the received event is in the event mask of every one of those loaded agents before utilizing the `EvSend()` primitive, which forwards this event to the agent. The same three programs are used as target programs under UDB, see Table 12.5 row #6. In this final experiment, in order to check the maximum overhead for this approach, the loading time of the external debugging tool is already included and added to the overhead imposed by updating the external agent with the received event. Figure 12.10 shows the average time for these six experiments in seconds.

Table 12.5. Performance of IDEA's Extension Agents

Approach		rsg			scramble			genqueen		
#	Time	Real/s	User/s	Sys/s	Real/s	User/s	Sys/s	Real/s	User/s	Sys/s
1	No Monitoring Involved	0.076	0.047	0.022	0.244	0.178	0.041	0.333	0.154	0.029
2	Directly Under UDB	0.108	0.078	0.026	0.253	0.203	0.042	0.328	0.166	0.044
3	Directly Under the Agent	0.421	0.295	0.119	1.153	0.860	0.282	2.939	2.197	0.708
4	Internal Procedure Call	1.064	0.878	0.134	2.735	2.412	0.305	7.710	6.781	0.782
5	External Procedure Call	1.106	0.896	0.131	2.860	2.504	0.327	8.027	7.000	0.852
6	External Context Switch	1.239	0.972	0.209	3.226	2.641	0.547	8.907	7.353	1.491

In Figure 12.11, the chart compares the total number of events that can be processed by an extension agent during one second. It compares IDEA's three event forwarding mechanisms: 1) internal procedure calls used by internal extension agents, 2) external procedure calls used by external

extension agents that utilize inter-program procedure calls, and 3) external co-expression used by external extension agents that utilize co-expression context switches. The chart in Figure 12.12 extends the comparison to compare agents running under IDEA against the same agent used in the standalone mode.

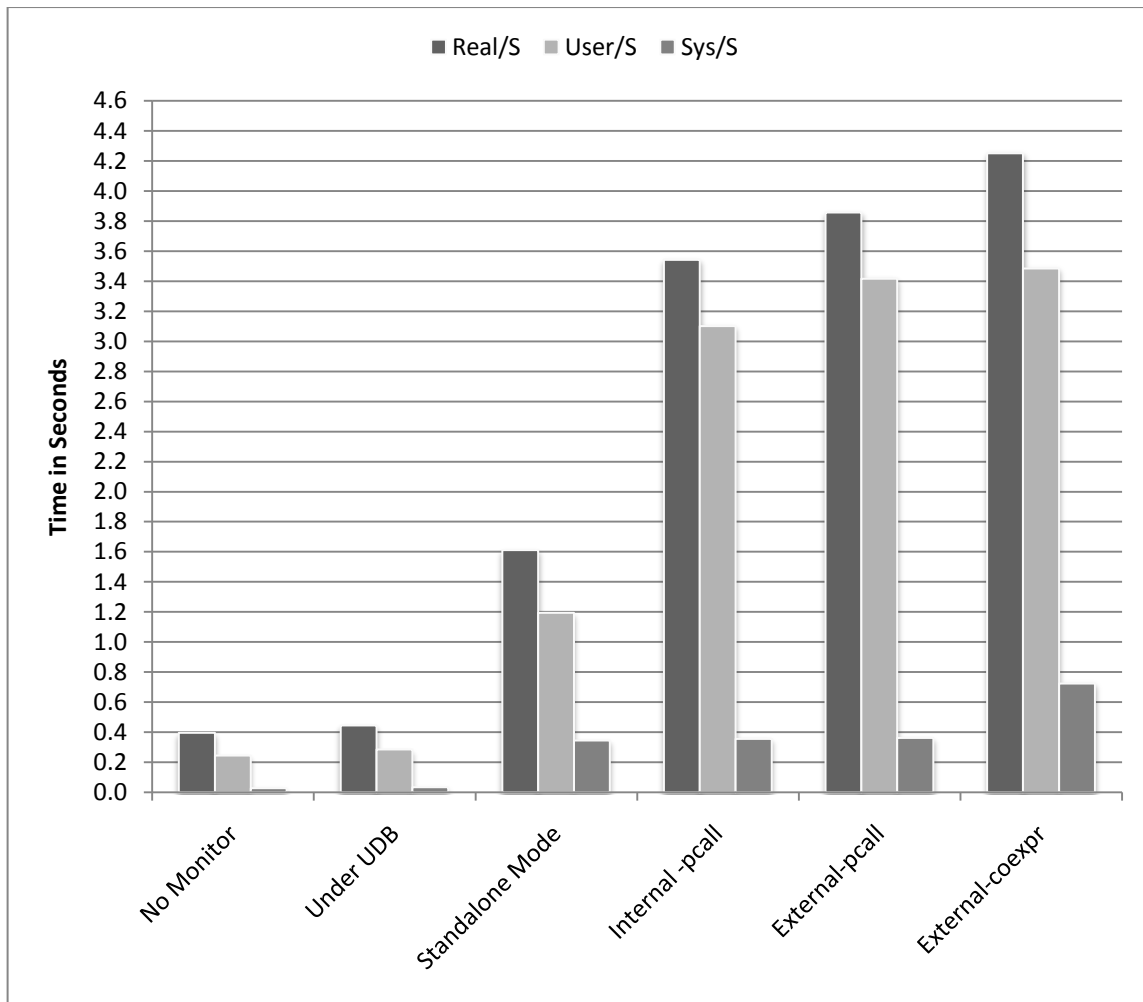


Figure 12.10. The Average Time for the Experimental Agent in Seconds

The standalone approach relatively proves its speed and efficiency, where it handles more than twice the number of events that are handled by the external agent under UDB. Using an external agent that is running under UDB provides a worse time performance than either the standalone approach or the internal (built-in) to UDB approach. However, this approach is still a valuable technique because of its flexibility and usability; especially for testing purposes. It allows users to write their own debugging and dynamic analysis tools and uses them on the fly from the inside of a typical source-level debugging session.

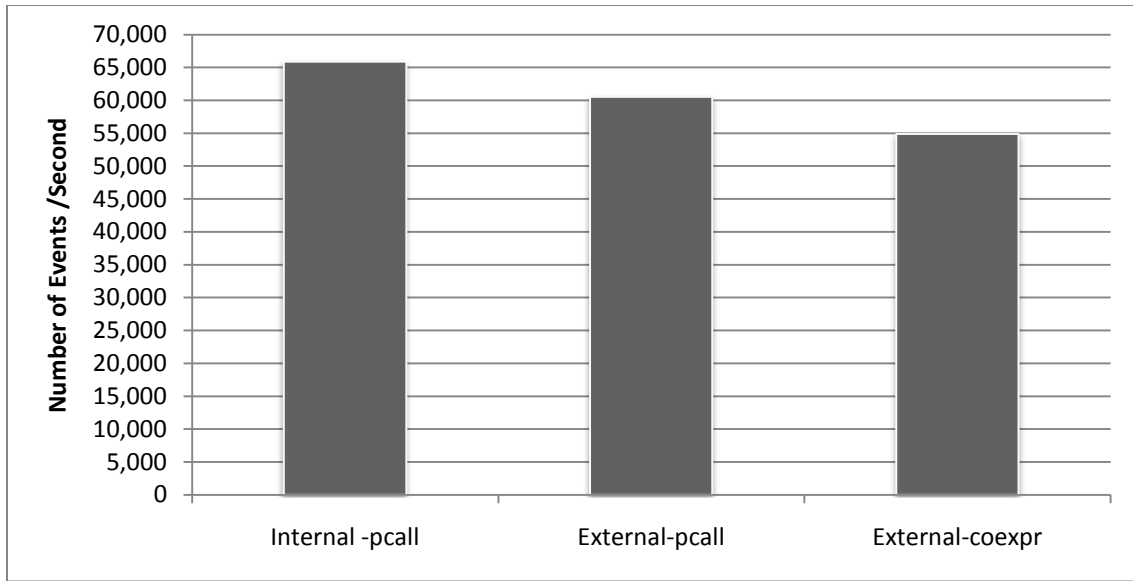


Figure 12.11. IDEA's Extension Techniques

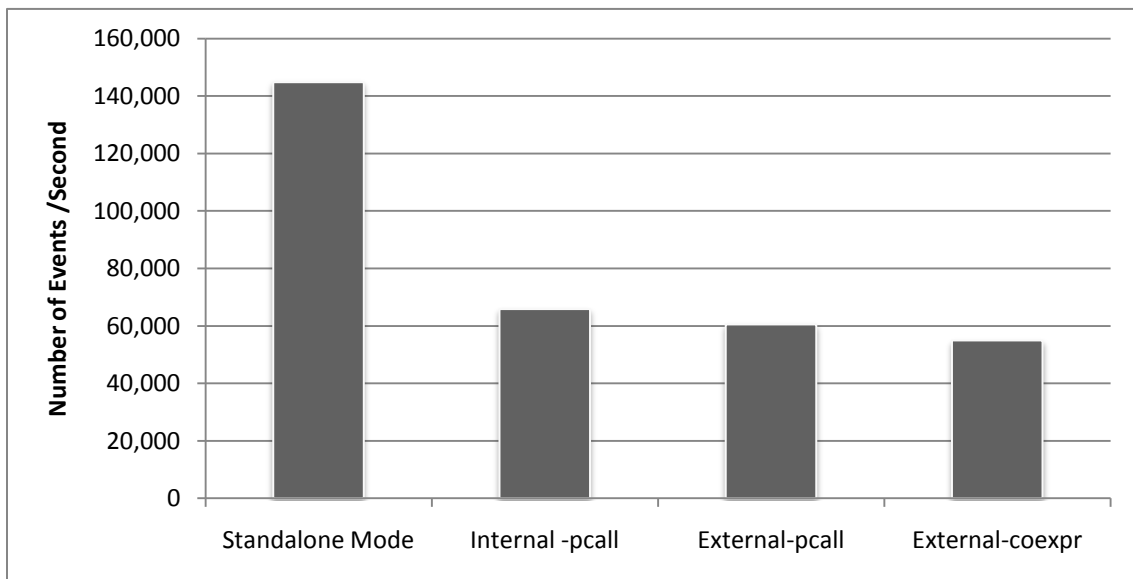


Figure 12.12. IDEA's Extension Agents vs. Standalone Mode

12.4. UDB's 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 scalability problem

because they must handle a huge volume of trace data. In regard to the processing time of events, UDB spends a considerable amount of time processing the instrumentation provided by AlamoDE, on the context switches between UDB and the buggy program, and on the event filtering and processing inside UDB's main debugging core (the evaluator). In this regard, the most frequent events are organized to be checked first.

Experiment

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 12.6 is the average of three different runs. The program is the Unicon translator itself, its size is 609KB and the virtual machine size is 790KB.

Table 12.6. The Time of Different UDB Debugging Features

#	Feature	Real/s	User/s	Sys/s
1	Normal (No UDB)	1.17	0.99	0.12
2	Under UDB	1.39	1.17	0.11
3	Tracing Procedure Calls	2.19	1.77	0.28
4	Tracing String Scanning Activities	2.38	1.98	0.27
5	Tracing All Procedure Activities	3.37	2.79	0.49
6	Breakpoint	3.68	3.45	0.14
7	Tracing Built-in Function Calls	4.61	3.75	0.79
8	Watchpoint	7.98	6.51	1.38
9	Tracing Type Conversion	8.99	6.74	1.98
10	Detect Variable Type Change	14.17	11.24	2.77
11	Detect Subscript Fails	14.91	11.41	3.33
12	Detect Zero Time Loops	15.01	12.17	2.66

Table 12.6 shows that UDB at present provides acceptable performance for ordinary debugging operations, with additional VM support needed for breakpoints and watchpoints. The AlamoDE architecture has been shown viable for debugging, but it will become more attractive with further tuning. More serious performance slowdowns are associated with various automatic debugging techniques, which may introduce complex dynamic analyses in order to function, see Figure 12.13. 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 for the mainstream languages.

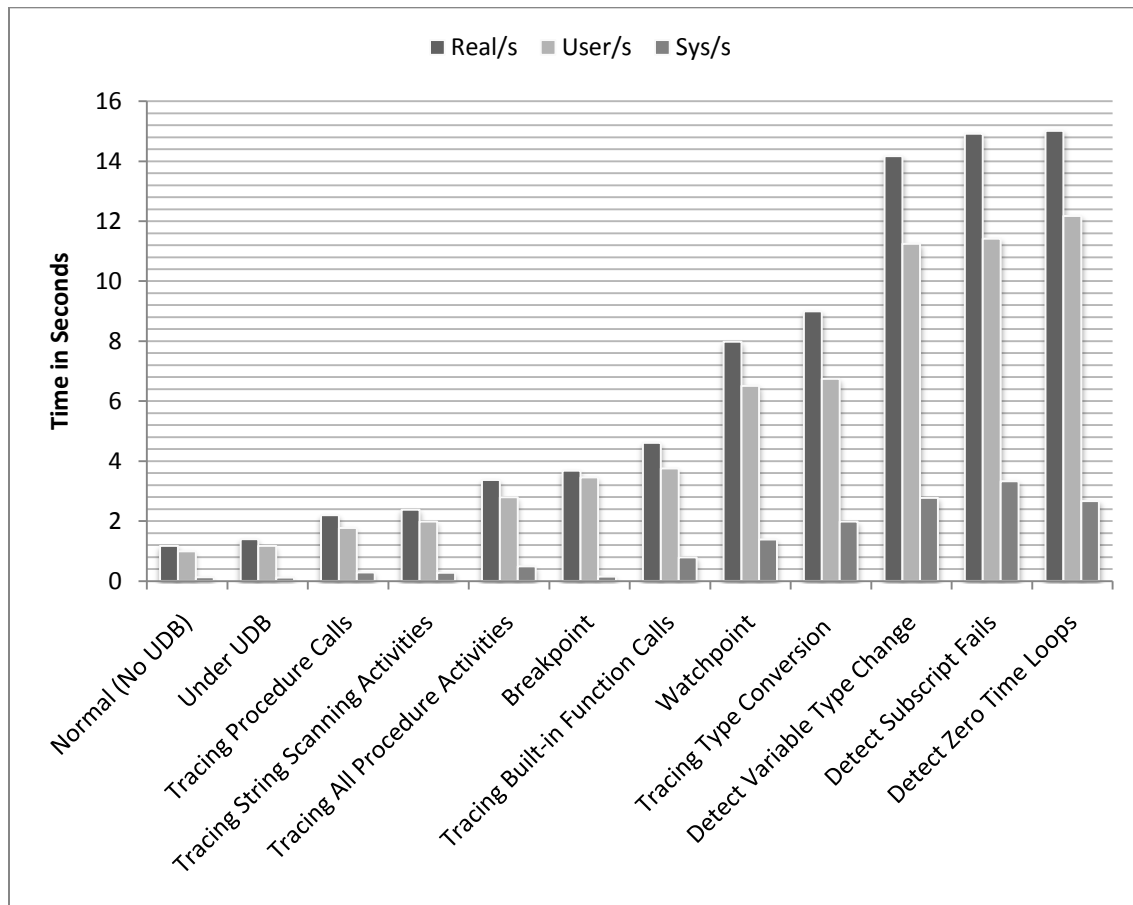


Figure 12.13. The Performance of UDB's Various Debugging Features

12.5. DT Assertions Evaluation

DT assertions provide the ability to validate relationships that may extend over the entire execution and check information beyond the current state of evaluation. DT assertions' temporal logic operators are internal agents used within the IDEA architecture. Those agents can reference other atomic agents, which provide access to valuable execution data and behavior information. This collaboration between agents can provide a helpful debugging technique and prove the value of the IDEA architecture. However, the design and implementation of DT assertions encounters some challenges and limitations discussed in the following subsections.

12.5.1. Evaluation

In consideration of the performance in terms of time, the implementation of temporal assertions utilizes a conservative assertion-based event-driven tracing technique. It only monitors relevant events; the event mask and value mask are generated automatically for each assertion at insertion time. Temporal assertions are evaluated in three levels. First is the state based level, which depends on any change to the referenced execution property. Second is the interval based level, which is determined by the assertion scope and kind. Third is the overall evaluation level, which occurs once per each execution. Different assertions can reference different execution properties. For this reason various assertions will differ in their cost.

However, in order to generally assess the role of the three evaluation levels in the complexity of these temporal assertions, let us assume that E_s is the maximum cost of monitoring and evaluating a state change within a temporal assertion. Furthermore, let us assume that n is the maximum number of state changes during a temporal interval and m is the maximum number of temporal intervals during an execution, see Figure 12.14. This means, the maximum cost of evaluating a temporal interval for this assertion is $E_s * n$ and the maximum cost of an assertion during the whole execution is $(E_s * n) * m$ which is equal to $E_s * n * m$. E_s includes the cost of event forwarding presented during evaluating the IDEA architecture in Section 12.3.2. This means that part of E_s is $(2E_p + 2E_c)$, where E_c is the cost of reporting an event to UDB and E_p is the cost of forwarding an event to the temporal logic agent (internal agent). This means the E_s dominates both n and m ; state change is the main performance issue in temporal assertions.

Furthermore, retained information is limited and driven by assertions' referenced execution properties. Assertions are virtually evaluated because they are in another execution space. The evaluation occurs in the debugger space with data collected and obtained from the buggy program space. The assertion log gives the user the ability to review the evaluation behavior of each assertion. Temporal assertions use in-memory tracing. A table is allocated for all assertions; it maps each assertion source code location to the instance object of the actual assertion. Another table is allocated for each assertion; it tracks temporal intervals, each of which is a list (stack) with each of the state based evaluation result. A third table is used to map assertion temporal intervals with their evaluation result, each of which is one value True, False, or Not Valid. Then one variable is holding the up to the point result which is either true or false. The dominating part in the used space is the number of state changes, E_s . Each state base evaluation is tracked with a record that keeps information about the line number, file name, and the result.

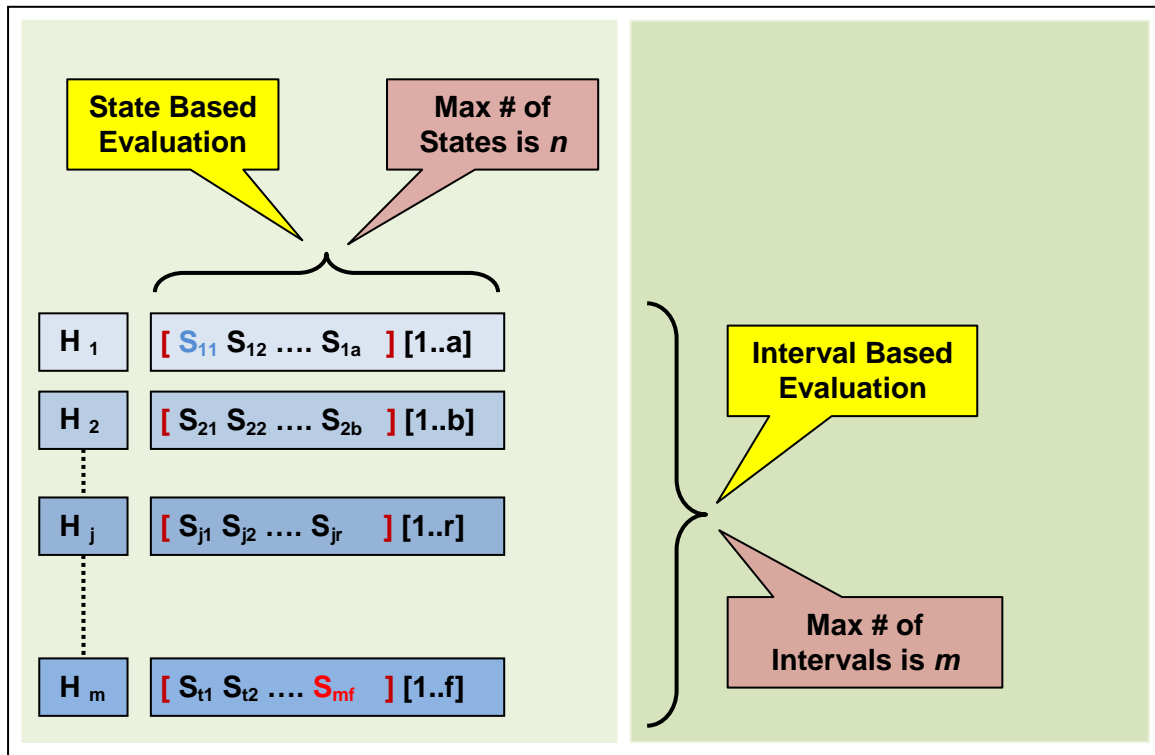


Figure 12.14. State Based vs. Interval Based Evaluation

Experiment

In order to find the impact of temporal assertions on the execution of the target program and the debugging time, a simple temporal assertion is applied on a simple program. The program prints numbers between 1 and 100,000; see Figure 12.15. The temporal assertion is applied with various sizes of temporal intervals. These intervals start at size 1, 100, 1000, 10,000, 50,000 and 100,000. Table 12.7 shows eight kinds of runs, each is observed for five times and the average time of these times measured. These kinds of runs range from measuring the time for the program in the standalone mode (no monitoring is involved), monitored under UDB with no assertion applied, then with an assertion that has various intervals. Figure 12.16 shows the impact of these temporal assertions of the execution time.

```

1  # A program that prints the numbers from 1..100000
2  procedure main(argv)
3      local n := 100000
4      every i := 1 to n do write(i)
5  end

```

Figure 12.15. Sample Unicon Program Used to Measure Time of Temporal Assertions

Table 12.7. Evaluation Time of Temporal Assertions

#	Feature	Real/s	User/s	Sys/s
1	standalone	2.1	0.1	0.2
2	under UDB	2.2	0.2	0.2
3	sometime { $i < n$ }	2.2	0.2	0.2
4	always (limit=100) { $i < n$ }	2.7	0.2	0.2
5	always (limit=1000) { $i < n$ }	2.7	0.3	0.2
6	always (limit=10000) { $i < n$ }	3.9	1.5	0.2
7	always (limit=50000) { $i < n$ }	8.1	6.2	0.5
8	always (limit=100000) { $i < n$ }	13.6	12.4	0.9

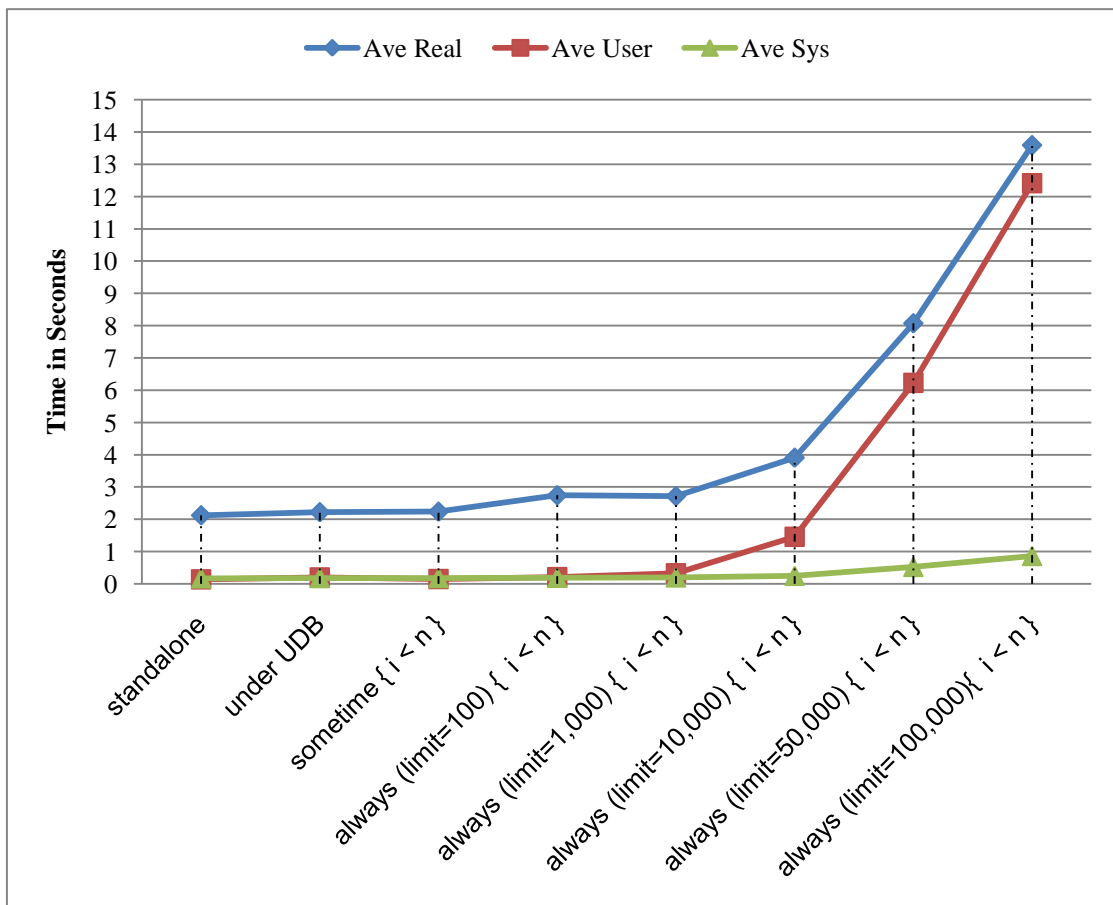


Figure 12.16. Temporal Assertions Evaluation Time

12.5.2. Challenges

Debugging with DT assertions provides advantages over typical assertions and conditional breakpoints and watchpoints. At the same time, it faces many challenges, some of which are based on associating assertions with the executable's source code, evaluating assertions in the debugger, and the source-level debugger's ability to obtain and retain relevant event-based and state-based information with reasonable performance.

First, if an assertion makes a reference to a variable, which is not accessible from within the assertion's scope, the debugger should automatically trace those variables and retain their relevant state information to be used at the assertion evaluation time. This allows a DT assertion to access data that is not live at the assertion's evaluation time.

Second, what if the assertion source code location is overlapping with a statement? Which one should be evaluated first, the assertion or the statement? A conservative approach may consider the assertion evaluation after the statement only if the statement has no variables referenced by the assertion, or if the statement does not assign to any of the assertion referenced variables. However, if the statement will assign to any of the assertion referenced variable, the assertion can be evaluated before and after the statement evaluation. If the two evaluations are different such as one is true and the other is false, or both are false, the assertion will stop the execution and hand the control to the debugger and the user to investigate. This dissertation, takes the simplest approach which is to evaluate the assertion before the statement. Furthermore, if an assertion is not overlapping with an executable statement, the AlamoDE framework cannot report a line number event from a non executable line. A line number event is only reported when a statement in that line number is fetched to be executed. This is reached by checking the assertion source code location before confirming that the assertion is inserted successfully. It checks whether the line number is empty or it is commented out.

Finally, if a referenced variable is an object or a data structure such as a list, this can cause two problems. First, the object is subject to changes under other names because of aliasing. Second, if the object is local, it may get disposed by the garbage collector before the evaluation time. The implementation could be extended to implement trapped variables that would allow us to watch an element of a structure or utilize an aliasing tracing mechanism to retain all changes that may occur under different names. The implementation of temporal assertions presented in this dissertation does not go after heap variables.

Chapter 13

Conclusion and Future Work

13.1. Conclusion

This dissertation presented three primary results. The first contribution is AlamoDE, which facilitates the ability to build various custom-defined debugging, dynamic analysis, and visualization tools. These tools can be written and tested as standalone programs, which utilize execution event patterns to detect suspicious execution behaviors and potential bugs. AlamoDE provides high level 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 very low cost (other than code size) in the production virtual machine. This integration allows the debugging tool 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's support for dynamic event customization provides the ability to change the set of requested 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. This dissertation proved that:

1. AlamoDE is sufficient to support various event-based debugging tools and techniques, including typical source-level debugging functionalities, with sufficient performance for production use
2. A high level event-based framework reduces the development cost of debugging tools and simplifies their extensions
3. AlamoDE's in-process debugging support allows for more efficient and complex communication patterns between the debugging tool and its target program.

The second contribution is IDEA extension architecture, which facilitates a source-level debugger with an extension mechanism. It provides the ease to simultaneously run those custom-designed tools (or agents) in conjunction with the typical source-level debugging session. The combination of AlamoDE and IDEA simplify the experimenting process with new custom-defined debugging tools and techniques, which may include:

1. Improvement to traditional techniques such as watchpoints and tracepoints
2. The ability to integrate verification and validation techniques such as dynamic temporal assertions
3. The simplicity to develop, test, and integrate new automated and dynamic analysis techniques of debugging agents

The final contribution of this dissertation is UDB, which is an event-driven source-level debugger that utilizes the IDEA architecture in its debugging core. See Figure 13.1. Under UDB, a user can easily load standalone debugging tools as external agents or incorporate them as internals within the source code of the debugger, all with no source code modification. It allows programmers to run a chosen suite of dynamic analysis agents (internals and externals) on the fly from within its typical console-based interactive debugging session. Furthermore, this event-driven agent-oriented implementation provides many advantages over traditional source-level debuggers, such as simplifying the process of extending the debugger with new debugging features. UDB's implementation and ease of extensibility demonstrate the value of AlamoDE and IDEA, and prove that:

1. A source-level debugger built on top of a high level event-driven debugging framework can surpass ordinary debuggers with more debugging capabilities, and it is easier to extend and maintain than a conventional debugger. For example, UDB is imitating GDB's functionalities with less than 10K lines of source code
2. It simplifies applying common source-level debugging functionalities such as breakpoints, watchpoints, stepping and continuing.
3. It facilitates complex debugging techniques in a simpler design that breaks the debugging process into small task-oriented agents. These agents allow for more debugging features with dynamic analysis and automatic debugging techniques
4. It provides the ability to employ agents, in the source-level debugging session, only when they are needed. The debugger can easily provide simple commands to load, unload, enable, and disable any number of external extension agents on the fly during a source-level debugging session
5. It enables custom-defined debugging tools to be used as external agents or registered as internal permanent debugging features. This encourages users to write their own high level standalone debugging agents.

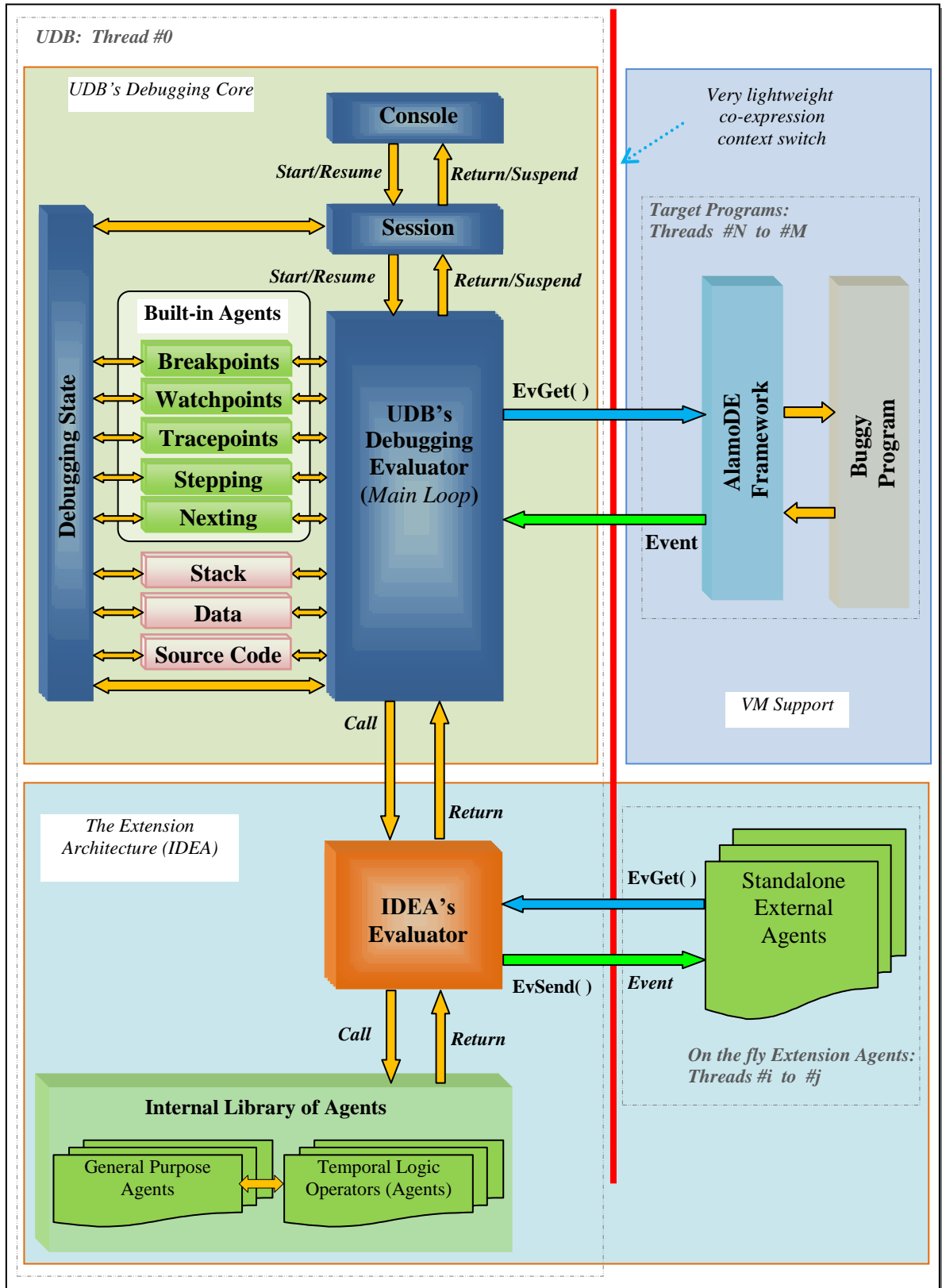


Figure 13.1. Dissertation Contributions

13.2. Discussion

Many debugging techniques such as algorithmic debugging, event grammars, and delta debugging provide automation for specific kinds of bug hunts. Trace-based debuggers such as the ODB [47, 48], TOD[49, 50], and the WhyLine [24, 25] debuggers provide advanced debugging techniques by recording the whole program's history of execution in order to provide an answer for a question that can be asked. In particular, ODB sacrifices most of the standard debugging techniques such as breakpoints, watchpoints, stepping and continuing. It 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. These trace-based debugging approaches encounter some limitations. For instance, the WhyLine debugger is limited for programs that run for a couple of minutes whereas TOD builds a distributed database that stores and indexes every state change during the execution.

In contrast, UDB preserves the debugging techniques found in classical debuggers such as GDB, but it integrates new automatic detection techniques that could be found in trace-based debuggers. These techniques provide the user with answers about the execution in terms of specific behavior. Instead of recording the complete program state and letting the user investigate or ask questions, UDB's approach is to employ agents that monitor the execution of the program and watch for some specific behaviors that may cause a bug. This has the advantage of better scalability and providing answers on the fly.

UDB's agents overcome several of the limitations of standard source-level debuggers. For example, typical source-level debuggers heavily rely on the user's ability to investigate the execution state. If a bug does not crash the program's execution, then a user has to step inside the execution state with anticipated breakpoints and watchpoints. Often, users start with breakpoints that are far before or after the bug's root cause. They end up re-running and single stepping through the program repeatedly; each with different breakpoints and watched variables. In contrast, UDB has the potential to bring more debugging techniques into the source-level debugging session. 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 or a suspicious activity. This has the advantage of better scalability and providing answers on the fly. Often, these agents add indispensable value into the debugging process with moderate impact on the performance of the buggy program. UDB's agents provide capabilities that can be found in trace-based debuggers with the advantage of being small and task-oriented for better scalability.

In general, the slowdown imposed by automatic and dynamic analysis techniques depends on the algorithms used and their implementations within the debugging tool or agent. Compared with the slowdown of many automatic debugging tools, the performance of UDB is reasonable. For example, a suite of debugging agents imposes at most 20 times slowdown on the execution of the buggy program over an uninstrumented execution mode. 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. To place this in perspective, a debugger such as valgrind 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 enable or disable from within a breakpoint based debugging session. This combination of valgrind-style dynamic analysis within an interactive debugger provides more effective debugging.

13.3. Limitations

AlamoDE still has some limitations. First, AlamoDE's instrumentation has no cheap means of filtering an event based on source file name; in particular there is no `E_File` event. For instance, when the source-level debugger has a breakpoint on specific line, it will receive the `E_Line` event whenever that event code and value are satisfied regardless of which source file the execution encountered that event in. This limitation is inherent in the instrumentation. Like most other binary executables, Unicon's binary format has little information about the source code, which must be checked separately from the binary. In particular, the line number event `E_Line` is reported or checked for lines that are about to be executed. However, if the user placed a breakpoint on an empty line, then this event will never get reported by the interpreter of the target program. So, in the implementation of a source-level debugger, the debugger itself checks the current version of source code to validate whether that line is an empty line or not. This mechanism works as long as the executable binary is built from this current version of the source code. However, if the source code is ever modified without rebuilding the executable binary, this may cause confusion for the user.

In general, UDB agents may utilize execution information before the current execution state. This has the advantage that an agent can be built to memorize and analyze what a user might do when debugging with a typical source level debugger. However, those external extension agents that are loaded during the debugging session can only analyze information after their loading/activation time. In general, agents will not be able to record or detect execution properties that were executed when the agent is not present or is disabled. This feature can have one advantage only if the user intentionally wants the agent to detect and analyze a portion of the execution, in this case the user can

manually enable and disable agents between different points; in particular, when the target program is stopped because of a breakpoint, watchpoint, or single stepping.

For DT assertions that may be inserted in the middle of the debugging session and contain references to variables that may have been processed before the insertion time, the debugger will not be able to evaluate such assertions until those variables are executed sometime while the assertion is live. Three options were available. First, lazy evaluation: if any of the assertion's referenced variables is live while the assertion is live, the debugger will put this assertion on the waiting list until all of its relevant data is available. Second, totally ignore the assertion evaluation at that point; hoping that in future hits the data will be available. Finally, stop the execution and provide reasoning about the unsuccessful evaluation. This dissertation took the simplest approach, which is to mark this hit with `Not Valid` and not consider it in the overall evaluation process. Furthermore, the debugger is able to retain assertion relevant information as long as the assertion is enabled. The debugger will not be able to evaluate data that was processed when the assertion was disabled. However, this may have some advantages in some cases where the user is interested in ignoring a portion of the data between two points of execution.

13.4. Future Work

Previous event-based source-level debuggers, such as Dalek [42] encountered performance obstacles. AlamoDE provides usable debugging support proved in the implementation of UDB and its IDEA extension architecture. However, relatively compared with an uninstrumented execution mode, IDEA has room for significant improvement in its performance for extension agents. This slowdown in the processing speed is based on the compulsory overhead associated with the current event's reporting and forwarding mechanism, which is formulated as context switches between the external agents and the debugging core. Performance can be further improved by buffering related events and avoiding extra context switches.

Another potential improvement is to offload the cost of external agents onto additional processor cores. This requires extending Unicon with real concurrency, where different co-expressions can be off loaded onto different processors. Furthermore, the value mask is used as a second filtering mechanism to reduce the number of reported events and further improve the monitoring performance. However, IDEA's debugging core only knows about the event mask of the external agents. Adding support for external agents' value masks would help improve the performance whenever that value mask is in use by the agent. Another debugging context where further work is needed is to debug a long running real time system that is interactive with the user and maybe with other users over the

network. The debugger needs to be plugged into the running program without interfering with its event-driven execution.

Further future work might aim at reducing the number of context switches by adopting a new communication paradigm. At present IDEA's debugging core plays the role of a central server in a star network. A ring-based architecture where each agent forwards events to another agent instead of having a central coordinator would reduce context switches by up to 50%. Another possible architecture is a broadcasting mechanism where the buggy program broadcasts events to all secondary debugging tools. Furthermore, expand UDB/IDEA capabilities to include an optional inter-process communication can support collaborative debugging tools that may share a real time debugging session. It also allows experimentation with foreign agents that may live on different machines and communicate with a network protocol.

Another area of future work may focus on improving the process of debugging using UDB. This can be achieved by adding more agents that utilize automatic debugging techniques for classes of bugs that are difficult to catch using standard techniques such as duplicated control logic, wrong operator, and aliasing-related bugs. Furthermore, UDB's classical debugging features, such as breakpoints and watchpoints, are provided through monitored events and event filtering. Even though these techniques perform well during debugging, improving their performance can be achieved by further implementation of common techniques such as *trapped virtual machine instruction* for breakpoints, and *trapped variable* for watchpoints. Moreover, the increased number of utilized agents associates relatively with maintenance efforts during the debugging session. For example, more agents can easily mean more opened windows for the programmer to manage and organize. This has some relations to the current user interface maintained by UDB that is centered on a command line interface—it is better to have a GUI interface with a mechanism that integrates dynamic agents within the same GUI. These GUI interfaces can be reached by different means such as: 1) building an Eclipse plug-in for UDB, 2) extending DDD to support UDB, or 3) extending Unicon's IDE to include UDB support.

DT assertions are augmented with temporal logic operators. In this approach, assertions are added on the fly during the debugging session and virtually executed in the buggy program source code. Those assertions have capabilities that go beyond the limitations of conditional breakpoints and watchpoints, and the typical in-code assertions, which either of them cannot check the value of a variable that is active in the caller activation record. However, a case study can show how end-users can take advantage of temporal assertions and how fast it allows them to locate the root cause of a

bug (or a suite of bugs). This study can compare these newly introduced debugging techniques to standard techniques found in a typical debugger such as GDB.

13.5. Extensibility to Other Languages

The availability of the virtual machine and its runtime system made the current implementation for all of UDB, IDEA, and DTA economically feasible. The choice of Icon/Unicon as a target implementation is a sponsor requirement. However, the approach can be extended to other languages and debuggers, which is very important to increase the usability of these features. In this regard, a subset of the Alamo framework used by IDEA for Unicon debugging has been implemented for monitoring ANSI C and Python on both Sparc Solaris and Intel Linux. Future work may extend UDB and its IDEA debugging facilities to these languages, or port them to run on other debugging platforms such as JPDA [23]. Another potential future work is to use an instrumentation framework, such as ASM [111], PIN [112], and ATOM [113], as a substitute for AlamoDE. For example, Java may utilize different implementation of DT assertions. Any JPDA based source-level debugger, such as Eclipse, can be extended with similar implementation. The implementation of this extension will be relatively similar to the current implementation since both JPDA and AlamoDE provide an event-driven debugging model. Java debuggers that use instrumentation frameworks (whether it is a third party instrumentation framework such as ASM or a native instrumentation package such as `java.lang.instrument`) can be extended by facilitating those frameworks for DT assertions with different tweaks to the current implementation.

On the other hand, DT assertions' initial implementation within an event-driven source-level debugger and a virtual machine based language does not limit them to this kind of debugger. For example, compiled languages such as C and C++ can take advantage of a third party instrumentation framework. Another way is to extend an existing source-level debugger such as GDB. GDB is already implements breakpoints by inserting illegal instructions and facilitates software implementation for watchpoints. Those can be extended to support automatic data collection for DT assertions. However, DT assertions may increase the number of inserted illegal instructions, which may produce a performance problem. A simpler approach to facilitate DT assertions for C and C++ programs would be to extend the implementation of GDB's front-end debugger DDD [8]. Extending DDD would provide a free GUI interface. DDD already utilizes automatic techniques to obtain and visualize execution data. Extending DDD with DT assertions could be straight forward to adapt from already used techniques. Of course, the implementation will be different.

Appendices

Appendix A: Dynamic Temporal Assertions

This appendix provides sample temporal assertions that can be used in a UDB debugging session.

A.1. Past-Time Assertions

Temporal Intervals of Past-Time temporal assertions start at entering the assertion scope (calling the scope procedure) and end at reaching assertion's source code location for the very first time after entering the scope

A.1.1. Past-Time Temporal Logic Operators

1. `alwaysp() { expression }`
asserts that expression must always hold (evaluate to true) for each, state, temporal interval, and during the whole execution.
2. `sometimep() { expression }`
asserts that expression must hold at least once for each temporal interval, and during the whole execution.
3. `previous() { expression }`
asserts that expression must hold right at the last state before the end of the temporal interval
4. `since() { condition ==> expression }`
asserts that expression must hold right after condition is true up until the end of the temporal interval and for each interval.

A.1.2. Example of Past-Time Assertions

1. `(udb) assert test.icn:50 alwaysp() { x < 10 }`
asserts that always in the procedure that contains line 50, the value of x is less than 10
2. `(udb) assert test.icn:50 previous() { x < y }`
asserts that the last in the procedure that contains line 50, the value of last value x, before the end of the interval, is less than y
3. `(udb) assert test.icn:50 since() { x=0 ==> x<0 }`
asserts that always after x is 0 then x is less than 0 up until the end of the interval which is by reaching the source code of the assertion.

A.2. Future-Time Assertions

Temporal Intervals of Future-Time temporal assertions start at reaching assertion's source code location for the very first time after entering the assertion scope and ends at exiting the assertion scope (returning from the scope procedure). In this kind of temporal assertions, the source code location can be hit more than once before the interval is closed

A.2.1. Future-Time Temporal Logic Operators

1. `alwaysf() { expression }`
asserts that expression *must always hold* (evaluate to true) for each, state, temporal interval, and during the whole execution.
2. `sometimef() { expression }`
asserts that expression *must hold at least once* for each temporal interval, and during the whole execution.
3. `next() { expression }`
asserts that expression *must hold* right at the very first state after the start of the temporal interval
4. `until() { condition ==> expression }`
asserts that expression *must hold* from the beginning of the temporal interval up until condition is true or the end of the temporal interval and for each interval.

A.2.2. Example of Future-Time Assertions

1. `(udb) assert test.icn:50 alwaysf() { x < 10 }`
asserts that always in the procedure that contains line 50, the value of x is less than 10
2. `(udb) assert test.icn:50 next() { x < y }`
asserts that the last in the procedure that contains line 50, the value of last value x, before the end of the interval, is less than y
3. `(udb) assert test.icn:50 until() { x=0 ==> x<0 }`
asserts that always after x is 0 then x is less than 0 up until the end of the interval which is by reaching the source code of the assertion.

A.3. All-Time Assertions

Temporal Intervals of All-Time temporal assertions start at entering assertion's scope and ends at exiting that scope; regardless of the provides source code location

A.3.1. All-Time Temporal Logic Operators

1. `always() { expression }`
asserts that expression *must always hold* (evaluate to true) for each, state, temporal interval, and during the whole execution
2. `sometime() { expression }`
asserts that expression *must hold at least once* for each temporal interval, and during the whole execution

A.3.2. Example of All-Time Assertions

1. `(udb) assert test.icn:50 always() { x < 10 }`
asserts that always in the procedure that contains line 50, the value of x is less than 10
2. `(udb) assert test.icn:50 sometime() { x < y }`
asserts that at least one time in the procedure that contains line 50, the value of x is less than y
3. `(udb) assert test.icn:50 sometime() { x < foo:y }`
asserts that always in the procedure that contains line 50, the value of x is less than last/current value of y from procedure foo

Appendix B: Evaluation and Performance

This appendix provides detailed information about the performance information and the experiments conducted through this dissertation.

B.1. Experimental Programs

During the evaluation of the research conducted in this dissertation, a suite of seven programs are used as monitored targets during the experiments. These programs are:

1. `rsg.icn` stands for random string generator. It generates randomly selected sentences from a grammar. It was written by Ralph E. Griswold.
2. `genqueen.icn` program solve an arbitrary-size n-queens problem. The program solves the non-attacking n-queens problem for (square) boards of arbitrary size. The problem consists of placing chess queens on an n-by-n grid such that no queen is in the same row, column, or diagonal as any other queen. The output is each of the solution boards; rotations not considered equal. It generates all possible solutions for the n-queen problem. It is mostly an algorithmic operation that uses generators. It was written by Peter A. Bigot.
3. `scramble.icn` program reads a document and re-outputs it in a cleverly scrambled fashion. It performs string scanning intensive operations. It was written by Tenaglia.
4. `ichartp.icn` program implements a simple chart parser – a slow but easy-to-implement strategy for parsing context free grammars. It operates in bottom-up fashion. This program was written by Richard L. Goerwitz.
5. `igrep.icn` program is a string search utility that imitates to UNIX `egrep`. It uses the enhanced regular expression supported by `regexp.icn`. It was written by Robert J. Alexander.
6. `miu.icn` program generates strings based on the MIU system. It was originally written by Cary A. Coutant, and modified by Ralph E. Griswold.
7. `pargen.icn` program generates a parser for a context-free language. This program reads a context-free BNF grammar and produces an Icon program that is a parser for the corresponding language. It was written by Ralph E. Griswold.

B.2. Experimental Modes

The following tables shows the average time of the programs described in the appendix C.1. Each of these programs is executed five times and the average of these five runs is calculated. These programs are used in six experiments.

1. No monitoring involved which calculus the average time of each one of these program when it runs in its normal execution without it being monitored by another program
2. Under UDB represents the execution time of the program is measured when it is used under UDB without any command involved. The monitored events are E_Exit, E_Error, and E_Signal
3. Standalone mode represents that each one of these programs is run under the control of a standalone monitor. This monitor is simple enough to count the number of reported events. The same monitor is used under UDB in three different modes
4. Internal-pcall mode represents the target program is running under UDB and the extension agent is used internally in the procedure mode
5. External-pcall mode represents the target program is running under UDB and the extension agent is used in external mode that is utilizing the inter-program procedure calls
6. External-coexpr mode represents the target program is running under UDB and the extension agent is used in the external mode that utilizes the co-expression context switch mode

B.3. Monitored Events

Out of AlamoDE's 121 kinds of events, a suite of four events are monitored in the programs that are presented in the previous section. These events are source code related events. E_Deref is reported when a variable is dereferenced, E_Line is reported when execution changes from one line number into another, E_Pcall is reported when a user procedure is called, and E_Syntax is reported when a major syntax construct is entered or exited. These events are source code related events, which are mostly used debugging purposes. In order to find the actual cost of these monitored events, each of these events is monitored separately in each target program. The average time of the reported events is measured.

Table B.1. rsg Execution Time

#	The rsg: Execution Mode	Real/s	User/s	Sys/s
1	No Monitor	0.060	0.028	0.027
2	Under UDB	0.142	0.081	0.027
3	Standalone Mode	0.271	0.167	0.077
4	Internal -pcall	0.592	0.475	0.076
5	External-pcall	0.639	0.510	0.082
6	External-coexpr	0.691	0.522	0.120

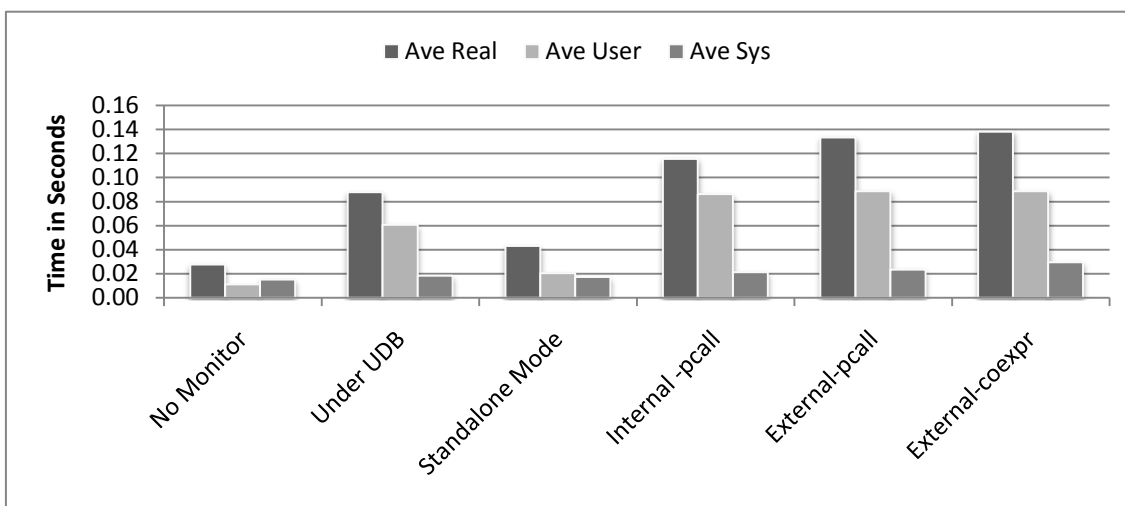


Figure B.1. rsg Average Execution Time

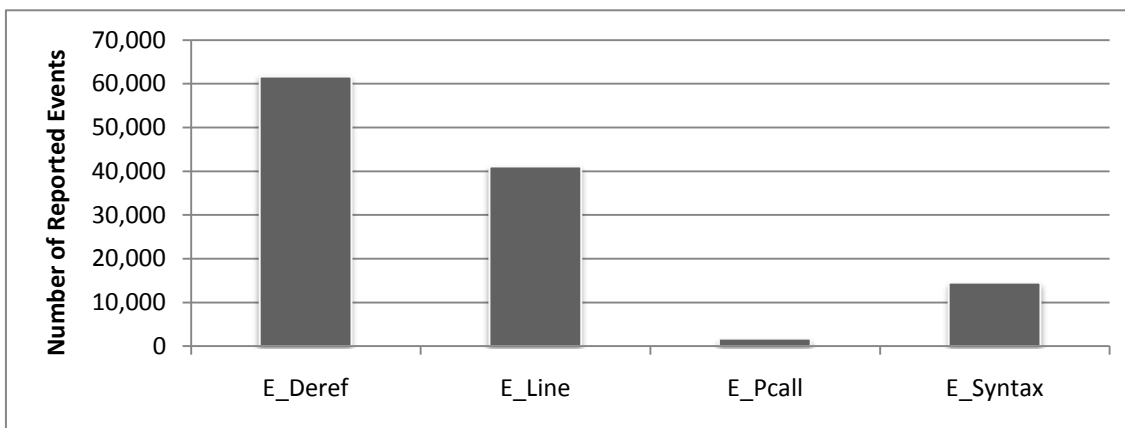


Figure B.2. rsg Reported Events

Table B.2. genqueen Execution Time

#	The genqueen: Execution Mode	Real/s	User/s	Sys/s
1	No Monitor	0.295	0.130	0.026
2	Under UDB	0.356	0.162	0.037
3	Standalone Mode	1.308	0.950	0.317
4	Internal -pcall	3.075	2.694	0.335
5	External-pcall	3.363	2.982	0.330
6	External-coexpr	3.770	3.044	0.629

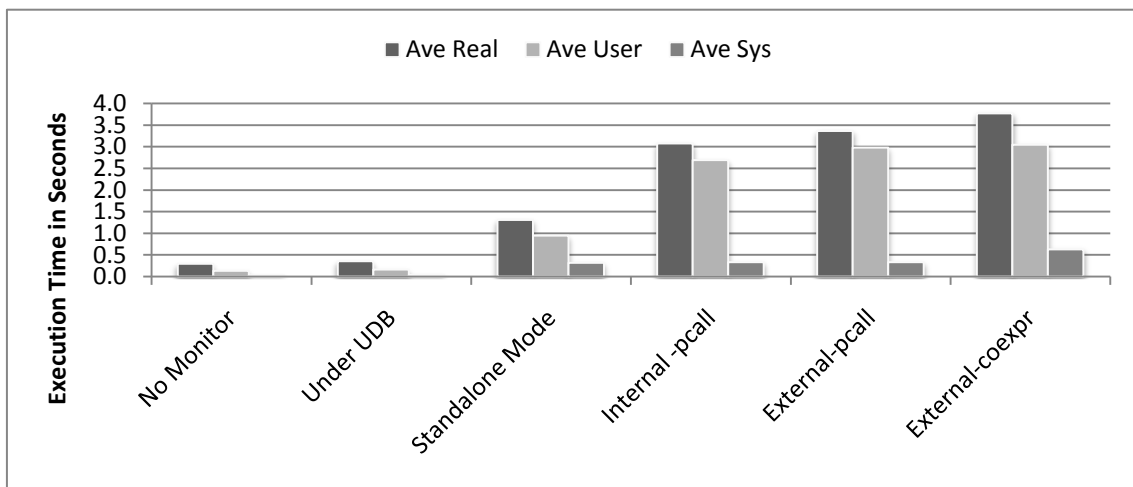
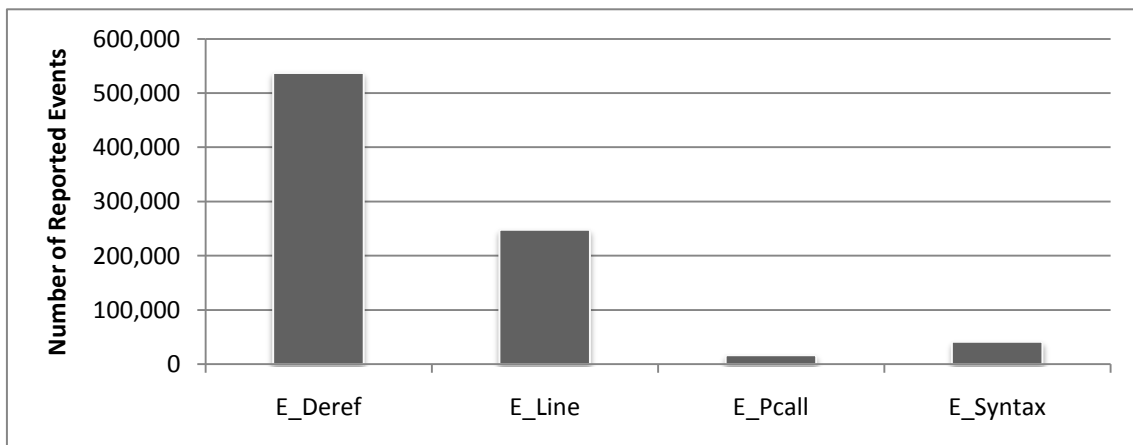
**Figure B.3. genqueen Average Execution Time****Figure B.4. genqueen Reported Events**

Table B.3. scramble Execution Time

#	The scramble: Execution Mode	Real/s	User/s	Sys/s
1	No Monitor	0.249	0.156	0.045
2	Under UDB	0.281	0.189	0.048
3	Standalone Mode	0.528	0.396	0.122
4	Internal -pcall	1.034	0.895	0.120
5	External-pcall	1.140	0.994	0.122
6	External-coexpr	1.235	1.007	0.205

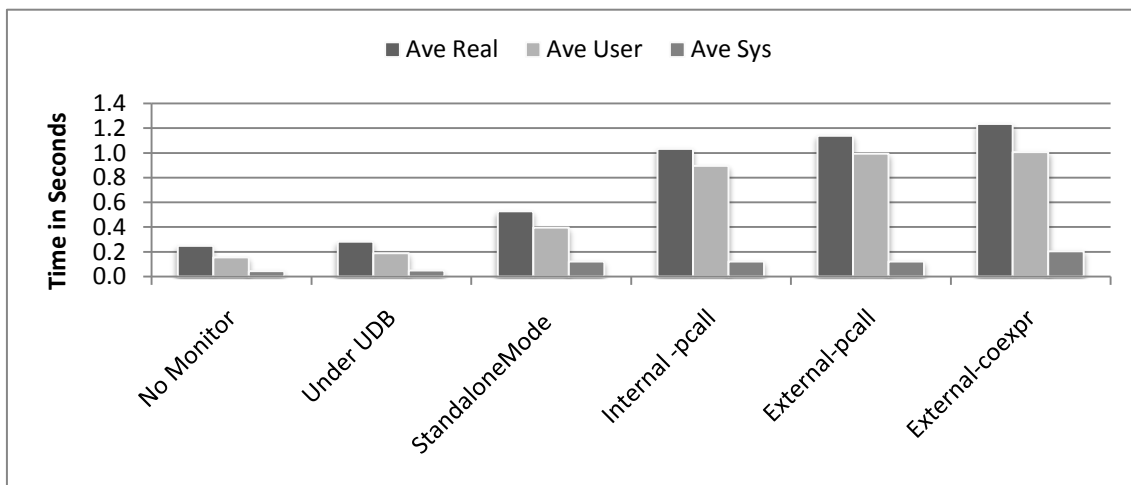
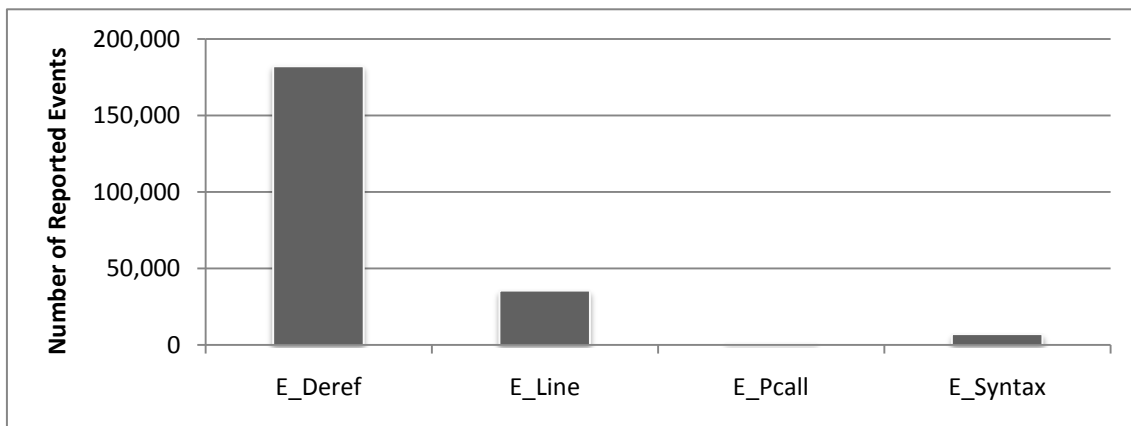
**Figure B.5. scramble Average Execution Time****Figure B.6. scramble Reported Events**

Table B.4. ichtartp Execution Time

#	The ichtartp: Execution Mode	Real/s	User/s	Sys/s
1	No Monitor	0.691	0.653	0.029
2	Under UDB	0.751	0.694	0.033
3	Standalone Mode	6.953	5.346	1.583
4	Internal -pcall	16.394	14.713	1.632
5	External-pcall	17.915	16.222	1.639
6	External-coexpr	19.828	16.522	3.265

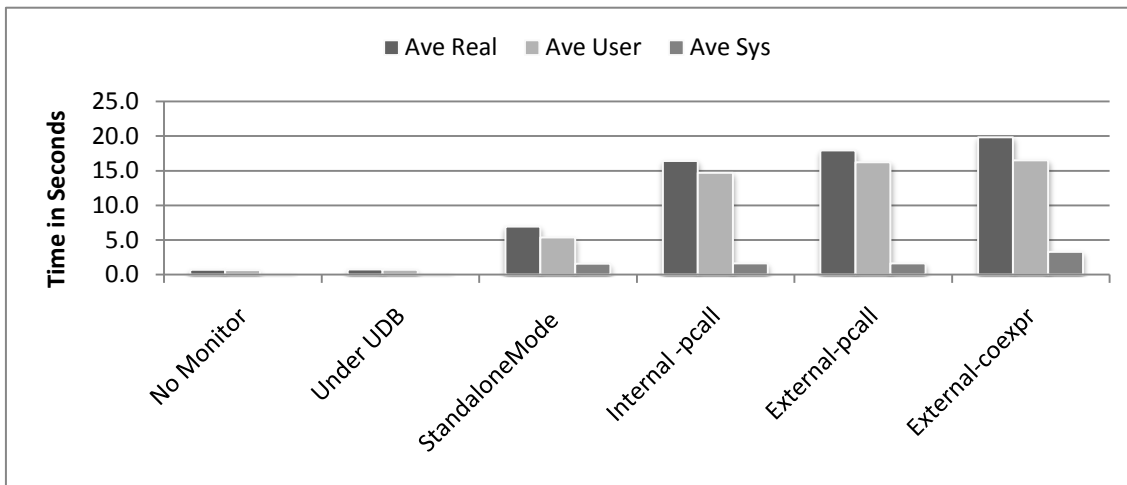


Figure B.7. ichtartp Average Execution Time

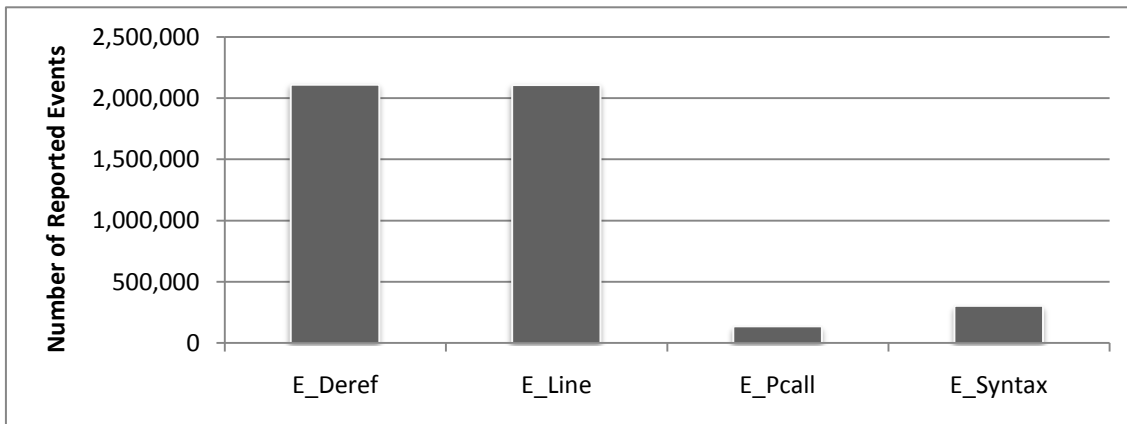


Figure B.8. ichtartp Reported Events

Table B.5. igrep Execution Time

#	The igrep: Execution Mode	Real/s	User/s	Sys/s
1	No Monitor	0.108	0.070	0.015
2	Under UDB	0.131	0.090	0.022
3	Standalone Mode	0.590	0.444	0.141
4	Internal -pcall	1.424	1.251	0.152
5	External-pcall	1.571	1.385	0.159
6	External-coexpr	1.697	1.412	0.265

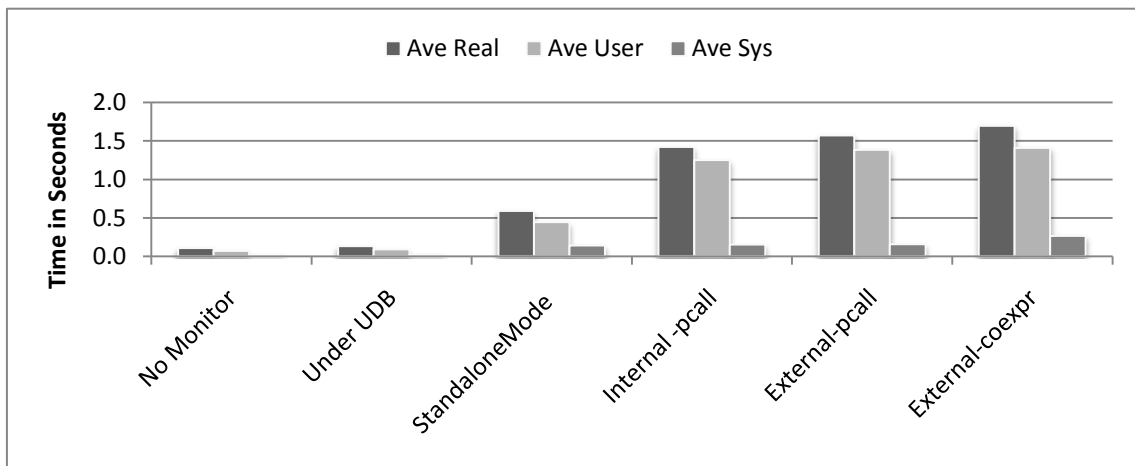


Figure B.9. igrep Average Execution Time

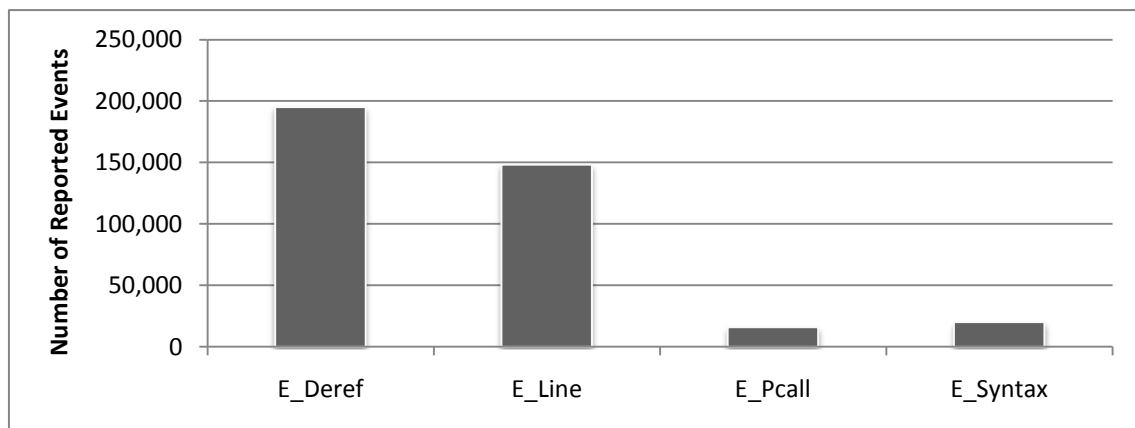


Figure B.10. igrep Reported Events

Table B.6. miu Execution Time

#	The miu: Execution Mode	Real/s	User/s	Sys/s
1	No Monitor	1.340	0.672	0.042
2	Under UDB	1.370	0.715	0.056
3	Standalone Mode	1.592	1.031	0.162
4	Internal -pcall	2.165	1.602	0.158
5	External-pcall	2.234	1.740	0.170
6	External-coexpr	2.406	1.800	0.550

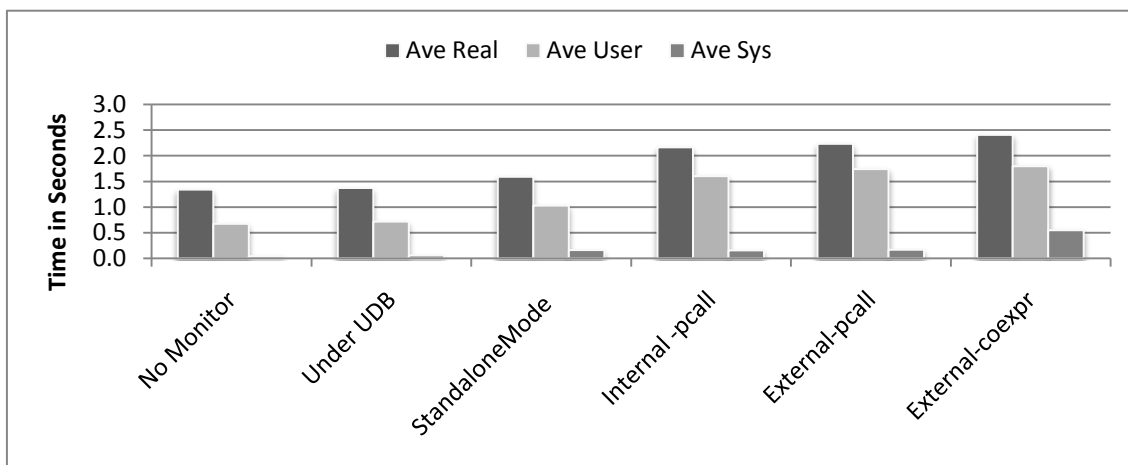
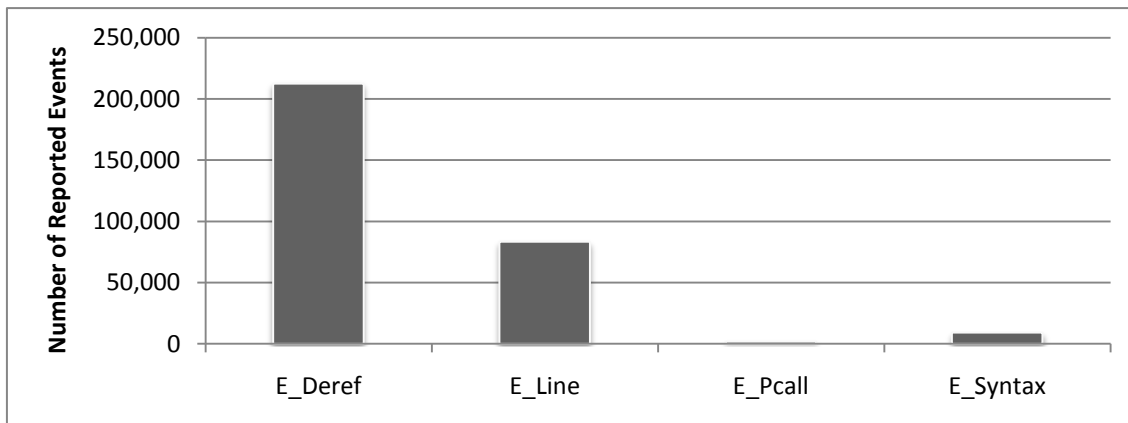
**Figure B.11. miu Average Execution Time****Figure B.12. miu Reported Events**

Table B.7. pargen Execution Time

#	The pargen: Execution Mode	Real/s	User/s	Sys/s
1	No Monitor	0.028	0.011	0.015
2	Under UDB	0.088	0.061	0.018
3	Standalone Mode	0.043	0.021	0.017
4	Internal -pcall	0.115	0.086	0.021
5	External-pcall	0.133	0.089	0.023
6	External-coexpr	0.138	0.089	0.030

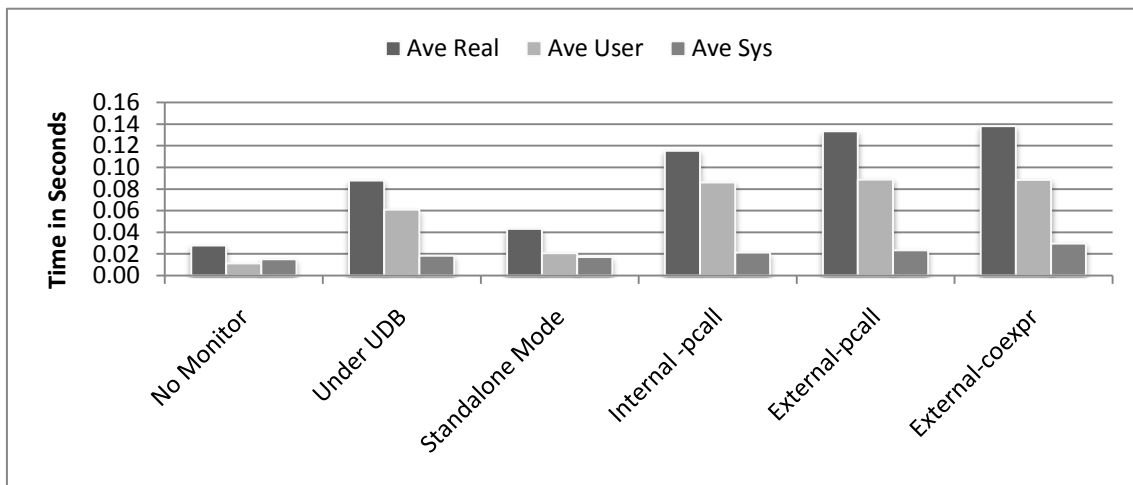


Figure B.13. pargen Average Execution Time

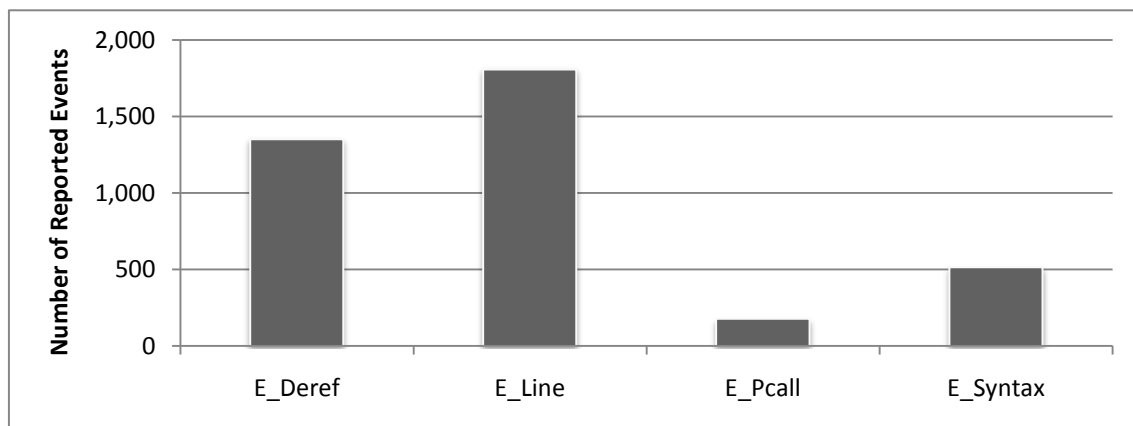


Figure B.14. pargen Reported Events

B.4. Average Monitored Events (E_Deref, E_Line, E_Syntax, E_Pcall)

This appendix shows the average of all experiments provided in Appendix B.3.

Table B.8. The Average Monitoring Time of All Events

#	Execution Mode	Real/s	User/s	Sys/s
1	No Monitor	0.396	0.246	0.028
2	Under UDB	0.446	0.284	0.035
3	Standalone Mode	1.612	1.194	0.346
4	Internal -pcall	3.543	3.102	0.356
5	External-pcall	3.856	3.417	0.361
6	External-coexpr	4.252	3.485	0.723

Table B.9. Number of Reported Events

#	Execution Mode	Average Number of Reported Events				
		E_Deref	E_Line	E_Syntax	E_Pcall	Other
1	rsg	61,619	41,032	14,534	1,672	61,619
2	gengueen	182,162	35,568	7,150	1,001	182,162
3	scramble	543,746	248,062	41,266	16,789	543,746
4	ichartp	2,111,315	2,107,136	30,4027	136,194	2,111,315
5	igrep	197,311	148,270	20,088	16,013	197,311
6	miu	212,421	83,539	9,289	1,868	212,421
7	pargen	1,351	1,806	515	178	1,351

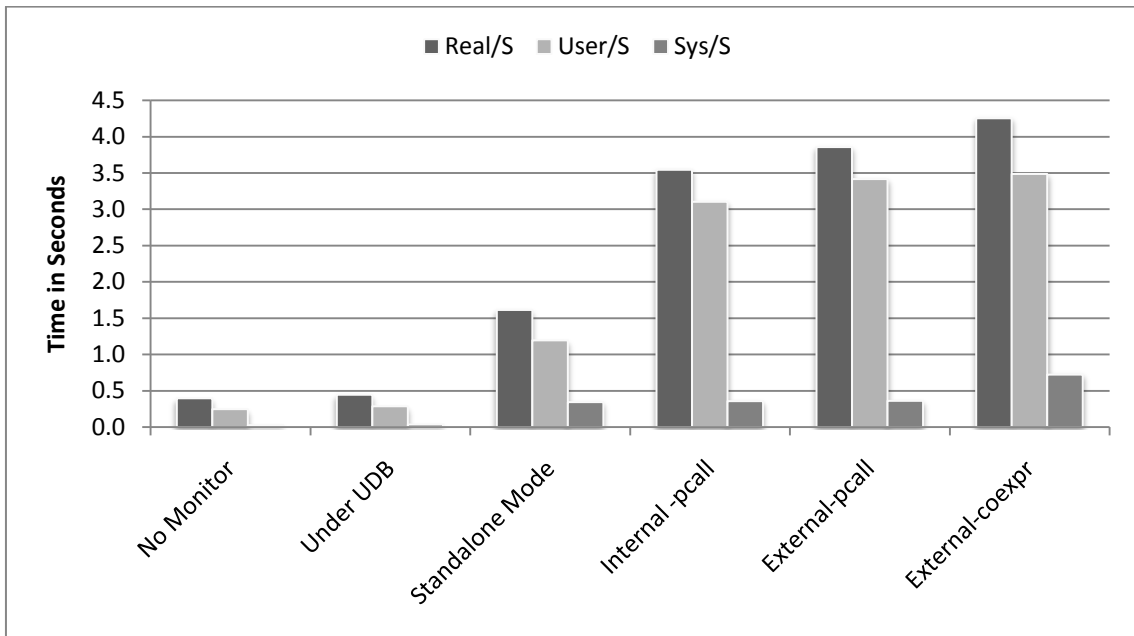


Figure B.15. Average Time of all Events

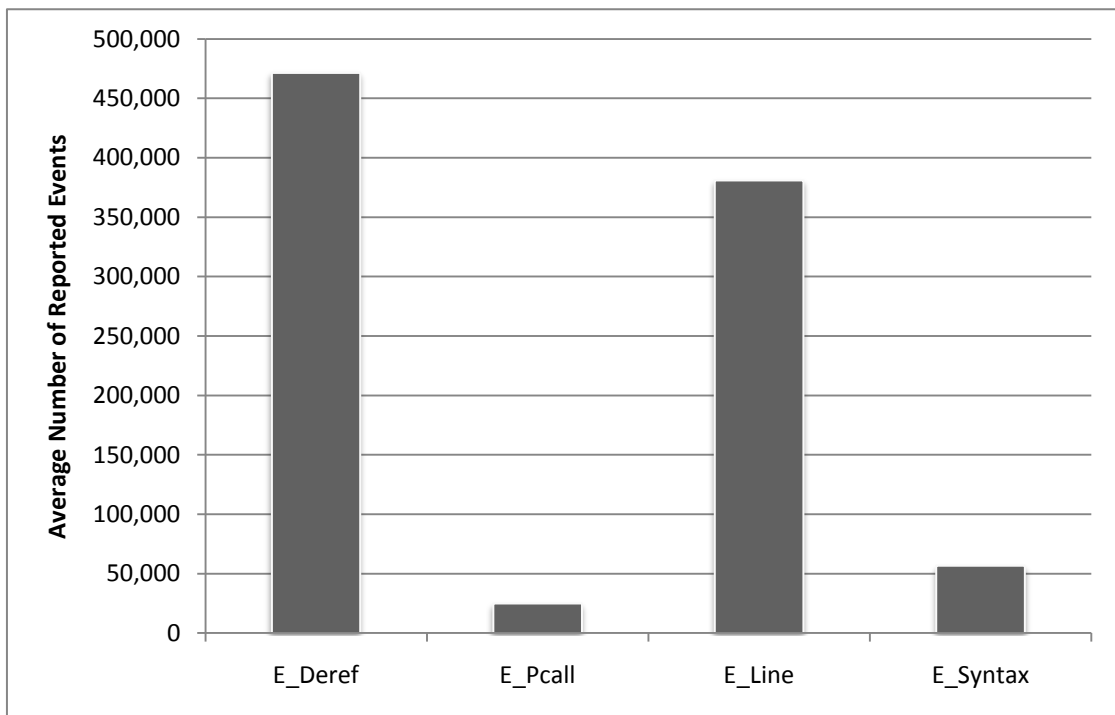


Figure B.16. Average of E_Deref, E_Pcall, E_Line, and E_Syntax Events

Appendix C: UDB Command Summary

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

C.1. 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.

C.2. 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.3. 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.

C.4. Stopping UDB

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

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

C.5. 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**.

C.6. 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**.

C.7. 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 procedure	Removes the breakpoint at the entry to procedure .
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 <code>info break</code> , but marked as <i>deleted</i> .
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] .

C.8. 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	<p>Sets an assignment watchpoint on variable whenever assigned, with an optional condition on the assigned value. watch is an alias to this awatch command.</p> <p>If -silent is provided, the watchpoint does not notify the user at every incident.</p> <p>If count is provided and count > 0, it observes the first count number of incidents.</p> <p>If count is provided and count < 0, the user is able to trace back the last count number of incident's locations and values.</p>
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 <i>assignment</i> watchpoints.
info rwatch	Shows a list of all <i>read</i> watchpoints.
info vwatch	Shows a list of all <i>value change</i> watchpoints.
info twatch	Shows a list of all <i>type change</i> 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.

C.9. Tracepoints

Techniques are used to observe execution behavior 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 manual investigations. 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**.

<code>trace [-silent] [count] operator [behavior [op value]]</code>	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: (+, -, *, /, \, =, ~=, ==, ~==, ===, ~===, <, <=, <<=, >, >=, >>=, ++, --, **, !, ?, []).
<code>trace [] fail</code>	Sets a tracepoint on [] (<i>subscript operation</i>) whenever it is failed .
<code>trace ! suspend</code>	Sets a tracepoint on ! (<i>Bang operator</i>) whenever it is suspended .
<code>trace = fail</code>	Sets a trace point on = whenever it is failed .
<code>trace ~==</code>	Sets a trace point on ~== whenever any of its behaviors is satisfied (occurred).
<code>trace ~== return</code>	Sets a tracepoint on ~== whenever it returns (the operation succeeded because both sides are lexically not equal).
<code>trace ~== return = "ab"</code>	Sets a tracepoint on ~== whenever it returns (the operation succeeded because both sides are lexically equal to "ab").
<code>info tracepoints</code>	Prints a complete list of all tracepoints; <code>info trace</code> and <code>trace</code> are aliases.
<code>info trace [n]</code>	Prints detailed information about the tracepoint with id number [n].
<code>info trace [name]</code>	Prints detailed information about the tracepoint set on [name].
<code>info trace enabled</code>	Prints a complete list of all enabled tracepoints.
<code>info trace disabled</code>	Prints a complete list of all disabled tracepoints.
<code>info trace deleted</code>	Prints a complete list of all deleted tracepoints.
<code>clear trace [n]</code>	Clears all tracepoints, if [n] is provided, it only clears the tracepoint with id number [n].
<code>clear trace [name]</code>	Clears all tracepoints, if [name] is provided, it only clears the tracepoint set on [name].
<code>delete trace [n]</code>	Deletes all tracepoints, if [n] is provided, it only deletes the tracepoint with id number [n].
<code>delete trace [name]</code>	Deletes all tracepoints, if [name] is provided, it only deletes the tracepoint set on [name].
<code>enable trace [n]</code>	Enables all tracepoints; if [n] is provided, it only enables the tracepoint that has the id [n].
<code>enable trace [name]</code>	Enables all tracepoints; if [name] is provided, it only enables the tracepoint set on [name].
<code>disable trace [n]</code>	Disables all tracepoints; if [n] is provided, it only disables the tracepoint that has the id number [n].
<code>disable trace [name]</code>	Disables all tracepoints; if [name] is provided, it only disables the tracepoint set on [name].

C.10. 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.

C.11. 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.

C.12. 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 field 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.

C.13. 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. 1 -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.

C.14. 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.

C.15. 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.

C.16. 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 session	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

C.17. Temporal Assertions

Temporal Assertions-related commands.

assert file:line always() { expr }	expr must hold at all times
assert file:line sometime() { expr }	exper must hold at least once during each interval

<code>assert file:line alwaysp() { expr }</code>	<code>expr</code> must hold at all times
<code>assert file:line sometimep() { expr }</code>	<code>expr</code> must hold at least once during each interval
<code>assert file:line since() { p1 ==> p2 }</code>	since <code>p1</code> holds, <code>p2</code> must hold at all times up to the end of the interval
<code>assert file:line previous() { p }</code>	<code>p</code> must hold at the previous state right before the assertions' location
<code>assert file:line alwaysf() { expr }</code>	<code>expr</code> must hold at all times
<code>assert file:line sometimef() { expr }</code>	<code>expr</code> must hold at least once during each interval
<code>assert file:line until() { p1 ==> p2 }</code>	<code>p1</code> holds until <code>p2</code> holds
<code>assert file:line next() { p }</code>	<code>p</code> must hold at the next state right after the assertions' location
<code>info assert</code>	Prints information about all assertions available in the session
<code>info assert id</code>	Prints information about the assertion number <code>id</code>
<code>info assert id hit</code>	Prints information about the assertion number <code>id</code> and its interval number <code>hit</code> .

Bibliography

- [1]. Boothe, B., "Efficient Algorithms for Bidirectional Debugging." In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (Vancouver, British Columbia, Canada, June 18 - 21, 2000). PLDI '00. ACM, New York, NY, 2000, pp. 299-310.
- [2]. Winterbottom, P., "ACID: A Debugger Built From A Language." In *Proceedings of the Winter 1994 USENIX Conf.*, San Francisco, CA. 1994. pp. 211-222.
- [3]. Jeffery, C. L. "Goal-Directed Object-Oriented Programming in Unicon." In *Proceedings of the 2001 ACM Symposium on Applied Computing* (Las Vegas, Nevada, United States). SAC '01. ACM, New York, NY, 2001. pp. 306-308.
- [4]. Jeffery C. L., Mohammand, S., Pereda, R., and Parlett, R., "*Programming with Unicon*," 2004. Published under the terms of the GNU Free Documentation License, Version 1.2. <http://www.unicon.org>.
- [5]. Stallman, R. M., Pesch, R., Shebs, S., et al, "Debugging with GDB: the GNU Source Level Debugger." Ninth edition, for Version 7.0.5, <http://www.gnu.org/software/gdb/documentation>, 2009.
- [6]. Griswold, R.E., and Griswold, M.T., "*The ICON Programming Language*," Third Edition. Published by Peer-to-Peer Communications, San Jose, CA, 1996.
- [7]. Griswold, R.E., and Griswold, M.T., "*The Implementation of the Icon Programming Language*," Princeton University Press, Princeton, NJ, 1986.
- [8]. Zeller, A., and Lutkehaus, D., "DDD-A Free Graphical Front-End For UNIX Debuggers," *SIGPLAN Not.*, vol. 31, no. 1, 1996, pp. 22-27.
- [9]. Cheung, W.H., Black, J.P., and Manning, E., "A Framework for Distributed Debugging," *IEEE Softw.*, vol. 7, no. 1, 1990, pp. 106-115.
- [10]. Araki, K., Furukawa, Z., and Cheng, J., "A General Framework for Debugging," *IEEE Softw.*, vol. 8, no. 3, 1991, pp. 14-20.
- [11]. Law, R., "An Overview of Debugging Tools," *SIGSOFT Softw. Eng. Notes*, vol. 22, no. 2, 1997, pp. 43-47.

- [12]. Auguston, M., Jeffery, C., and Underwood, S., "A Framework for Automatic Debugging," In *Proceedings of the 17th IEEE International Conference on Automated Software Engineering*, September 23-27, 2002, Edinburgh, UK, IEEE Computer Society Press, pp. 217--222.
- [13]. Kernighan, B.W., and Plauger, P.J., "*The Elements of Programming Style*," Second Edition McGraw-Hill, Inc. New York, NY, USA, 1982.
- [14]. Anonymous, Another Day, "Another Bug." *ACM Queue* vol. 1, no. 6. New York, NY, USA, Sep. 2003, pp. 58-61. DOI= <http://doi.acm.org/10.1145/945131.945156>.
- [15]. Liblit, B., Aiken, A., Zheng, A. X., and Jordan, M. I., "Bug Isolation via Remote Program Sampling." In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation* (San Diego, California, USA, June 09 - 11, 2003). PLDI '03. ACM, New York, NY, 2003, pp. 141-154. DOI= <http://doi.acm.org/10.1145/781131.781148>.
- [16]. Liblit, B., Naik, M., Zheng, A. X., Aiken, A., and Jordan, M. I., "Scalable Statistical Bug Isolation." In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA, June 12 - 15, 2005). PLDI '05. ACM, New York, NY, 2005, pp.15-26. DOI= <http://doi.acm.org/10.1145/1065010.1065014>.
- [17]. Mateis, C., Stumptner, M., and Wotawa, F., "Debugging of Java Programs Using a Model-Based Approach," In *Proceedings of the Tenth International Workshop on Principles of Diagnosis (DX-99)*, Loch Awe, Scotland, 1999. <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.57.7879>.
- [18]. Seward, N. Nethercote, J. Weidendorfer and the Valgrind Development Team, "*Valgrind 3.3 - Advanced Debugging and Profiling for GNU/Linux applications*," Network Theory Ltd, 2008. <http://www.network-theory.co.uk/valgrind/manual>.
- [19]. Nethercote, N., and Seward, J., "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," *SIGPLAN Not.* vol. 42, no. 6, 2007, pp. 89-100.
- [20]. Zeller, A., "Isolating cause-effect chains from computer programs." In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering* (Charleston, South Carolina, USA, November 18 - 22, 2002). SIGSOFT '02/FSE-10. ACM, New York, NY, 2002, pp.1-10. DOI= <http://doi.acm.org/10.1145/587051.587053>.

- [21]. Zeller, A. and Hildebrandt, R., "Simplifying and Isolating Failure-Inducing Input." *IEEE Trans. Softw. Eng.* vol. 28, no. 2. Feb. 2002, pp. 183-200.
DOI= <http://dx.doi.org/10.1109/32.988498>.
- [22]. Zeller, A., "Why Programs Fail: A Guide to Systematic Debugging," Elsevier Inc, New York USA, 2006.
- [23]. Sun Microsystems, Inc. "Java platform debug architecture (JPDA)," 2009. Available at:
<http://java.sun.com/products/jpda/>.
- [24]. Ko, A. J. and Myers, B. A., "Designing the Whyline: A Debugging Interface for Asking Questions about Program Behavior." In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Vienna, Austria, April 24 - 29, 2004). CHI '04. ACM, New York, NY, 2004, pp.151-158. DOI= <http://doi.acm.org/10.1145/985692.985712>.
- [25]. Ko, A. J. and Myers, B. A., "Source-Level Debugging With the Whyline." In *Proceedings of the 2008 International Workshop on Cooperative and Human Aspects of Software Engineering* (Leipzig, Germany, May 13 - 13, 2008). CHASE '08. ACM, New York, NY, 2008, pp. 69-72. DOI= <http://doi.acm.org/10.1145/1370114.1370132>.
- [26]. Ko, A., "Debugging By Asking Questions About Program Output." In *Proceedings of the 28th International Conference on Software Engineering* (Shanghai, China, May 20 - 28, 2006). ICSE '06. ACM, New York, NY, 2006, pp. 989-992.
DOI= <http://doi.acm.org/10.1145/1134285.1134471>.
- [27]. Ko, A. J. and Myers, B. A., "Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior." In *Proceedings of the 30th International Conference on Software Engineering* (Leipzig, Germany, May 10 - 18, 2008). ICSE '08. ACM, New York, NY, 2008, pp. 301-310. DOI= <http://doi.acm.org/10.1145/1368088.1368130>.
- [28]. Ko, A. J. and Myers, B. A., "Finding Causes of Program Output with the Java Whyline." In *Proceedings of the 27th International Conference on Human Factors in Computing Systems* (Boston, MA, USA, April 04 - 09, 2009). CHI '09. ACM, New York, NY, 2009, pp. 1569-1578. DOI= <http://doi.acm.org/10.1145/1518701.1518942>.
- [29]. Nikolik, B., "Convergence Debugging." In *Proceedings of the Sixth International Symposium on Automated Analysis-Driven Debugging* (Monterey, California, USA, September 19 - 21, 2005). AADEBUG'05. ACM, New York, NY, 2005, pp. 89-98.
DOI= <http://doi.acm.org/10.1145/1085130.1085142>.

- [30]. Xie, G., Xu, Y., Li, Y., and Li, Q., "Codebugger: A Software Tool for Cooperative Debugging." *SIGPLAN Not.* 35, 2 (Feb. 2000), 2000, pp. 54-60.
DOI= <http://doi.acm.org/10.1145/345105.345128>.
- [31]. Frost, R., "Jazz and the Eclipse Way of Collaboration." *IEEE Softw.* vol. 24, no. 6, Nov. 2007, pp. 114-117. DOI= <http://dx.doi.org/10.1109/MS.2007.170>.
- [32]. Liblit, B., "*Cooperative Bug Isolation: Winning Thesis of the 2005 ACM Doctoral Dissertation Competition*" (Lecture Notes in Computer Science). Springer-Verlag New York, Inc. 2007.
- [33]. Dionne, C., Feeley, M., and Desbiens, J., "A Taxonomy of Distributed Debuggers Based on Execution Replay." In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques*. August 1996. pp. 203-214.
<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.52.4538>.
- [34]. Auguston, M., "Tools for Program Dynamic Analysis, Testing, and Debugging Based on Event Grammars." In *Proceedings of the 12th International Conference on Software Engineering and Knowledge Engineering (SEKE 2000)*, Chicago, USA, July 6-8, 2000, pp.159-166.
- [35]. Bates, P., "Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior." *SIGPLAN Not.* vol. 24, no. 1, Jan. 1989, pp. 11-22.
DOI= <http://doi.acm.org/10.1145/69215.69217>.
- [36]. Bates, P. C., "Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior." *ACM Trans. Comput. Syst.* vol. 13, no. 1, Feb. 1995, pp. 1-31.
DOI= <http://doi.acm.org/10.1145/200912.200913>.
- [37]. Bourdoncle, F., "Abstract Debugging of Higher-Order Imperative Languages." In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation* (Albuquerque, New Mexico, United States, June 21 - 25, 1993). R. Cartwright, Ed. PLDI '93. ACM, New York, NY, 1993. pp. 46-55.
DOI= <http://doi.acm.org/10.1145/155090.155095>.
- [38]. Bourdoncle, F., "Assertion-Based Debugging of Imperative Programs by Abstract Interpretation." In *Proceedings of the 4th European Software Engineering Conference (ESEC)* (September 13 - 17, 1993). I. Sommerville and M. Paul, Eds. Lecture Notes In Computer Science, vol. 717. Springer-Verlag, London, 1993, pp. 501-516.

- [39]. Ducassé, M., “A Pragmatic Survey of Automated Debugging.” In *Proceedings of the First International Workshop on Automated and Algorithmic Debugging* (May 03 - 05, 1993). P. Fritzson, Ed. Lecture Notes In Computer Science, vol. 749. Springer-Verlag, London, 1993. pp. 1-15.
- [40]. Czyz, J. K. and Jayaraman, B., “Declarative and Visual Debugging in Eclipse.” In *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology Exchange* (Montreal, Quebec, Canada, October 21 - 21, 2007). Eclipse '07. ACM, New York, NY, 2007, pp. 31-35. DOI= <http://doi.acm.org/10.1145/1328279.1328286>.
- [41]. Mishergghi, G. and Su, Z., “HDD: Hierarchical Delta Debugging.” In *Proceedings of the 28th International Conference on Software Engineering* (Shanghai, China, May 20 - 28, 2006). ICSE '06. ACM, New York, NY, 2006, pp. 142-151. DOI= <http://doi.acm.org/10.1145/1134285.1134307>.
- [42]. Olsson, R. A., Crawford, R. H., and Ho, W. W., “A Dataflow Approach to Event-Based Debugging.” *Softw. Pract. Exper.* vol. 21, no. 2, Feb. 1991, pp. 209-229. DOI= <http://dx.doi.org/10.1002/spe.4380210207>.
- [43]. Olsson, R. A., Crawford, R. H., Ho, W. W., and Wee, C. E., “Sequential Debugging at a High Level of Abstraction.” *IEEE Softw.* vol. 8, no. 3, May, 1991, pp. 27-36. DOI= <http://dx.doi.org/10.1109/52.88941>.
- [44]. Abraham, R. and Erwig, M. “Goal-Directed Debugging of Spreadsheets.” In *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing* (September 20 - 24, 2005). VLHCC. IEEE Computer Society, Washington, DC, 2005, pp. 37-44. DOI= <http://dx.doi.org/10.1109/VLHCC.2005.42>.
- [45]. Seidner, R. and Tindall, N., “Interactive Debug Requirements.” In *Proceedings of the Symposium on High-Level Debugging* (Pacific Grove, California, March 20 - 23, 1983). SIGSOFT '83. ACM, New York, NY, 1983. pp. 9-22. DOI= <http://doi.acm.org/10.1145/1006147.1006151>.
- [46]. Mayer, W. and Stumptner, M., “Model-Based Debugging—State of the Art and Future Challenges.” *Electron. Notes Theor. Comput. Sci.* vol. 174, no. 4, May. 2007, pp. 61-82. DOI= <http://dx.doi.org/10.1016/j.entcs.2006.12.030>.
- [47]. Lewis, B., “Debugging Backwards in Time”. In *Proceedings of the Fifth International Workshop on Automated Debugging* (AADEBUG 2003), Ghent, Belgium, Sep. 8-10, 2003.

- [48]. Lewis, B. and Ducasse, M., "Using Events to Debug Java Programs Backwards in Time." In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Anaheim, CA, USA, October 26 - 30, 2003). OOPSLA '03. ACM, New York, NY, 2003, pp. 96-97.
DOI= <http://doi.acm.org/10.1145/949344.949367>.
- [49]. Pothier, G. and Tanter, É., "Extending Omniscient Debugging to Support Aspect-Oriented Programming." In *Proceedings of the 2008 ACM Symposium on Applied Computing* (Fortaleza, Ceara, Brazil, March 16 - 20, 2008). SAC '08. ACM, New York, NY, 2008, pp. 266-270. DOI= <http://doi.acm.org/10.1145/1363686.1363753>.
- [50]. Pothier, G., Tanter, É., and Piquet, J., "Scalable Omniscient Debugging." In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications* (Montreal, Quebec, Canada, October 21 - 25, 2007). OOPSLA '07. ACM, New York, NY, 2007, pp. 535-552. DOI= <http://doi.acm.org/10.1145/1297027.1297067>.
- [51]. Wu, X., Chen, Q., and Sun, X., "Design and Development of a Scalable Distributed Debugger for Cluster Computing." *Cluster Computing* vol. 5, no. 4, Oct. 2002, pp. 365-375.
DOI= <http://dx.doi.org/10.1023/A:1019708204283>.
- [52]. Graham, S. L., Kessler, P. B., and McKusick, M. K., "gprof: A Call Graph Execution Profiler." *SIGPLAN Not.* vol. 39, no. 4 (Apr. 2004), 2004, pp. 49-57.
DOI= <http://doi.acm.org/10.1145/989393.989401>.
- [53]. Narayanasamy, S., Pokam, G., and Calder, B., "BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging." *SIGARCH Comp. Archit. News* vol. 33, no. 2, May. 2005, pp. 284-295. DOI= <http://doi.acm.org/10.1145/1080695.1069994>.
- [54]. Searle, A., Gough J., Abramson, D., "Automating Relative Debugging," In *the proceeding of the 18th IEEE International Conference on Automated Software Engineering (ASE'03)*, Oct. 6-10, 2003, pp. 356-359.
- [55]. Sobic, R., and Abramson, D., "Guard: A Relative Debugger," *Software: Practice and Experience*, vol. 27, no. 2, (Published Online: Jan 8, 1999 by John Wiley & Sons, Ltd.) Feb.1997, pp. 185-206.
- [56]. Feldman, S. I. and Brown, C. B., "IGOR: A System for Program Debugging via Reversible Execution." In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging* (Madison, Wisconsin, United States, May 05 - 06, 1988). R. L.

Wexelbalt, Ed. PADD '88. ACM, New York, NY, 1988, pp. 112-123.

DOI= <http://doi.acm.org/10.1145/68210.69226>.

- [57]. Albertsson, L., "Simulation-Based Debugging of Soft Real-Time Applications." In *Proceedings of the Seventh Real-Time Technology and Applications Symposium (RTAS '01)* (May 30 - June 01, 2001). RTAS. IEEE Computer Society, Washington, DC, 2001, pp. 107-108.
- [58]. Weaver, V., McKee, S., "Using Dynamic Binary Instrumentation to Generate Multi-Platform Simpoints: Methodology and Accuracy." In *the 3rd EC International Conference on High Performance Embedded Architectures and Compilers (HiPEAC'08)*, Göteborg, Sweden, Jan. 2008, pp. 305-319.
- [59]. Liblit, B., Naik, M., Zheng, A. X., Aiken, A., and Jordan, M. I., "Scalable Statistical Bug Isolation." *SIGPLAN Not.* vol. 40, no. 6, Jun. 2005, pp. 15-26.
DOI= <http://doi.acm.org/10.1145/1064978.1065014>.
- [60]. Arumuga Nainar, P., Chen, T., Rosin, J., and Liblit, B., "Statistical Debugging Using Compound Boolean Predicates." In *Proceedings of the 2007 International Symposium on Software Testing and Analysis* (London, United Kingdom, July 09 - 12, 2007). ISSTA '07. ACM, New York, NY, 2007, pp. 5-15. DOI= <http://doi.acm.org/10.1145/1273463.1273467>.
- [61]. Mehner, K., "Trace Based Debugging and Visualization of Concurrent Java Programs with UML". Paderborn University, Dissertation, 2005.
<http://www.scientificcommons.org/30877459>.
- [62]. Reiss, S. P., "Trace-Based Debugging." In *Proceedings of the First International Workshop on Automated and Algorithmic Debugging* (May 03 - 05, 1993). P. Fritzson, Ed. Lecture Notes in Computer Science, vol. 749. Springer-Verlag, London, 1993, pp. 305-314.
- [63]. Crossno, P., Rogers, D. H., and Garasi, C. J., "Case Study: Visual Debugging of Finite Element Codes." In *Proceedings of the Conference on Visualization '02* (Boston, Massachusetts, October 27 - November 01, 2002). IEEE Computer Society, Washington, DC, 2002, pp. 517-520.
- [64]. Clarke, L. A. and Rosenblum, D. S., "A Historical Perspective on Runtime Assertion Checking in Software Development." *SIGSOFT Softw. Eng. Notes* vol. 31, no. 3, May. 2006, pp. 25-37. DOI= <http://doi.acm.org/10.1145/1127878.1127900>.

- [65]. Burdy, L., Cheon, Y., Cok, D. R., Ernst, M. D., Kiniry, J. R., Leavens, G. T., Leino, K. R., and Poll, E., "An Overview of JML Tools and Applications." *Int. J. Softw. Tools Technol. Transf.* vol. 7, no. 3 Jun. 2005, pp. 212-232.
DOI= <http://dx.doi.org/10.1007/s10009-004-0167-4>.
- [66]. Drusinsky, D., "The Temporal Rover and the ATG Rover." In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification* (August 30 - September 01, 2000). K. Havelund, J. Penix, and W. Visser, Eds. Lecture Notes in Computer Science, vol. 1885. Springer-Verlag, London, 2000, pp. 323-330.
- [67]. Allen, D., "The Perl Debugger." *Linux J.* 2005, 131, Mar. 2005, 8.
- [68]. Beazley, D. M., "*Python Essential Reference*," (3rd Edition) (Developer's Library). Sams, 2006.
- [69]. LaLonde, W. R. and Pugh, J. R.,: *Inside Smalltalk: vol. 1.*" Prentice-Hall, Inc. 1990.
- [70]. Hanson, D. R. "A Machine-Independent Debugger—Revisited." *Softw. Pract. Exper.* vol. 29, no. 10 Aug. 1999, pp. 849-862.
- [71]. Ramsey, N. and Hanson, D. R., "A Retargetable Debugger." In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation* (San Francisco, California, United States, June 15 - 19, 1992). R. L. Wexelblat, Ed. PLDI '92. ACM, New York, NY, 1992, pp. 22-31. DOI= <http://doi.acm.org/10.1145/143095.143112>.
- [72]. Hanson, D. R. and Korn, J. L., "A Simple and Extensible Graphical Debugger." In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Anaheim, California, January 06 - 10, 1997). USENIX Association, Berkeley, CA, 1997, pp. 173-184.
- [73]. Korn, J. L., "*Abstraction and Visualization in Graphical Debuggers*." Doctoral Thesis. UMI Order Number: AAI9944651., Princeton University, Princeton, NJ, 1999.
- [74]. A Graphical Interface to GNU's Debugger (XXGDB), 2003.
<http://linux.maruhn.com/sec/xxgdb.html>
- [75]. Rossi, B., "The Curses (Terminal-Based) Interface to the GNU Debugger--GDB," CGDB Version 0.6.4, 2 April 2007. <http://cgdb.sourceforge.net/>
- [76]. Alekseev, S., "Java Debugging Laboratory for Automatic Generation and Analysis of Trace Data." In *Proceedings of the 25th Conference on IASTED International Multi-Conference: Software Engineering* (Innsbruck, Austria, February 13 - 15, 2007). W. Hasselbring, Ed. ACTA Press, Anaheim, CA, 2007, pp. 177-182.

- [77]. Java Debugging Laboratory, JDLabStudio, version 1.0.0. ” <http://jdlabstudio.sourceforge.net/>.
- [78]. Dotan, D. and Kirshin, A., “Debugging and testing behavioral UML models.” In *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion* (Montreal, Quebec, Canada, October 21 - 25, 2007). OOPSLA '07. ACM, New York, NY, 2007, pp. 838-839.
DOI= <http://doi.acm.org/10.1145/1297846.1297915>.
- [79]. Sterling, C. D. and Olsson, R. A., “Automated Bug Isolation via Program Chipping.” In *Proceedings of the Sixth International Symposium on Automated Analysis-Driven Debugging* (Monterey, California, USA, September 19 - 21, 2005). AADEBUG'05. ACM, New York, NY, 2005, pp. 23-32. DOI= <http://doi.acm.org/10.1145/1085130.1085134>.
- [80]. Sterling, C. D. and Olsson, R. A., “Automated Bug Isolation via Program Chipping.” *Softw. Pract. Exper.* vol. 37, no. 10, Aug. 2007, pp. 1061-1086.
DOI= <http://dx.doi.org/10.1002/spe.v37:10>.
- [81]. Wall, K., and Hagen, W. V., “*The Definitive Guide to GCC*,” Second Edition (Definitive Guide). Springer-Verlag, New York, NY. 2006.
- [82]. Hovemeyer, D. and Pugh, W., “Finding Bugs Is Easy.” In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (Vancouver, BC, CANADA, Oct. 24 - 28, 2004). OOPSLA '04. ACM, New York, NY, 2004, pp. 132-136. DOI= <http://doi.acm.org/10.1145/1028664.1028717>.
- [83]. Ayewah, N., Pugh, W., Morgenthaler, J. D., Penix, J., and Zhou, Y., “Evaluating Static Analysis Defect Warnings on Production Software.” In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (San Diego, California, USA, June 13 - 14, 2007). PASTE '07. ACM, New York, NY, 2007, pp. 1-8. DOI= <http://doi.acm.org/10.1145/1251535.1251536> .
- [84]. InfoEther, “PMD,” Available at: <http://pmd.sourceforge.net/>.
- [85]. Evans, D., “Static Detection of Dynamic Memory Errors.” In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation* (Philadelphia, Pennsylvania, United States, May 21 - 24, 1996). PLDI '96. ACM, New York, NY, 1996, pp. 44-53. DOI= <http://doi.acm.org/10.1145/231379.231389>.
- [86]. Anderson, P. and Zarins, M., “The CodeSurfer Software Understanding Platform.” In *Proceedings of the 13th International Workshop on Program Comprehension* (May 15 - 16,

- 2005). IWPC. IEEE Computer Society, Washington, DC, 2005, pp. 147-148.
DOI= <http://dx.doi.org/10.1109/WPC.2005.37>.
- [87]. Pugh, K., Lint for C++. *C Users J.* vol. 11, no. 9, Sep. 1993, pp. 123-127.
- [88]. Flanagan, C., Leino, K. M., Lillibridge, M., Nelson, G., Saxe, J. B., and Stata, R., "Extended Static Checking For Java." In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany, June 17 - 19, 2002). PLDI '02. ACM, New York, NY, 2002, pp. 234-245.
DOI= <http://doi.acm.org/10.1145/512529.512558>.
- [89]. Console, L., Friedrich, G., and Dupré, D. T., "Model-Based Diagnosis Meets Error Diagnosis in Logic Programs" (Extended Abstract). In *Proceedings of the First International Workshop on Automated and Algorithmic Debugging* (May 03 - 05, 1993). P. Fritzson, Ed. Lecture Notes in Computer Science, vol. 749. Springer-Verlag, London, 1993, pp. 85-87.
- [90]. Mayer, W. and Stumptner, M., "Evaluating Models for Model-Based Debugging." In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering* (September 15 - 19, 2008). IEEE Computer Society, Washington, DC, 2008, pp. 128-137. DOI= <http://dx.doi.org/10.1109/ASE.2008.23>.
- [91]. Yilmaz, C. and Williams, C., "An Automated Model-Based Debugging Approach." In *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering* (Atlanta, Georgia, USA, November 05 - 09, 2007). ASE '07. ACM, New York, NY, 2007, pp. 174-183. DOI= <http://doi.acm.org/10.1145/1321631.1321659>.
- [92]. Mayer, W. and Stumptner, M., "Abstract Interpretation of Programs for Model-Based Debugging." In *Proceedings of the 20th International Joint Conference on Artificial intelligence* (Hyderabad, India, January 06 - 12, 2007). R. Sangal, H. Mehta, and R. K. Bagga, Eds. Ijcai Conference on Artificial Intelligence. Morgan Kaufmann Publishers, San Francisco, CA, 2007, pp. 471-476.
- [93]. Shende, S., Cuny, J., Hansen, L., Kundu, J., McLaughry, S., and Wolf, O., Event and State-Based Debugging in TAU: A Prototype. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools* (Philadelphia, Pennsylvania, United States, May 22 - 23, 1996). SPDT '96. ACM, New York, NY, 1996, pp. 21-30.
DOI= <http://doi.acm.org/10.1145/238020.238030>.

- [94]. Anonymous, "Automating C/C++ Runtime Error Detection with Parasoft Insure++," <http://www.parasoft.com/jsp/products/>.
- [95]. Sastry, D. C. and Jagadeesh, J. M., "Better Development with Boundschecker." *Computer* vol. 29, no. 6, Jun. 1996, pp. 107-109. DOI= <http://dx.doi.org/10.1109/MC.1996.507643>.
- [96]. Anonymous, "IBM Rational Purify"; <http://www-306.ibm.com/software/awdtools/purify>.
- [97]. Roy, G., "mpatrol," <http://sourceforge.net/projects/mpatrol/>.
- [98]. Perens, B., "Electric Fence," <http://perens.com/FreeSoftware/>.
- [99]. Anonymous, "Diversity Analyzer," 2004; <http://www.vidakquality.com/>.
- [100]. Wang, T. and Roychoudhury, A., "Hierarchical Dynamic Slicing." In *Proceedings of the 2007 International Symposium on Software Testing and Analysis* (London, United Kingdom, July 09 - 12, 2007). ISSTA '07. ACM, New York, NY, 2007, pp. 228-238. DOI= <http://doi.acm.org/10.1145/1273463.1273494>.
- [101]. Sridharan, M., Fink, S. J., and Bodik, R., "Thin Slicing." In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA, June 10 - 13, 2007). PLDI '07. ACM, New York, NY, 2007, pp. 112-122. DOI= <http://doi.acm.org/10.1145/1250734.1250748>.
- [102]. Xu, B., Qian, J., Zhang, X., Wu, Z., and Chen, L., "A Brief Survey of Program Slicing." *SIGSOFT Softw. Eng. Notes* 30, 2 (Mar. 2005), 2005, pp. 1-36. DOI= <http://doi.acm.org/10.1145/1050849.1050865>.
- [103]. Tip, F., "A Survey of Program Slicing Techniques," *Journal of Programming Languages* (JPL), vol. 3, Chapman & Hall, 1995. pp. 121-189. <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.43.3782>.
- [104]. Saito, Y., "Jockey: A User-Space Library for Record-Replay Debugging." In *Proceedings of the Sixth International Symposium on Automated Analysis-Driven Debugging* (Monterey, California, USA, September 19 - 21, 2005). AADEBUG'05. ACM, New York, NY, 2005, pp. 69-76. DOI= <http://doi.acm.org/10.1145/1085130.1085139>.
- [105]. Ducassé, M. "Coca: An Automated Debugger for C." In *Proceedings of the 21st International Conference on Software Engineering* (Los Angeles, California, United States, May 16 - 22, 1999). ICSE '99. ACM, New York, NY, 1999, pp. 504-513. DOI= <http://doi.acm.org/10.1145/302405.302682>.

- [106]. Templer, K. and Jeffery, C., "A Configurable Automatic Instrumentation Tool for ANSI C." In *Proceedings of the 13th IEEE International Conference on Automated Software Engineering* (October 13 - 16, 1998). IEEE Computer Society, Washington, DC, 1998, pp. 249 - 258. DOI= <http://doi.ieeecomputersociety.org/10.1109/ASE.1998.732663>.
- [107]. Jeffery, C., Zhou, W., Templer, K., and Brazell, M., "A Lightweight Architecture for Program Execution Monitoring." In *Proceedings of the 1998 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering* (Montreal, Quebec, Canada, June 16 - 16, 1998). A. M. Berman, Ed. PASTE '98. ACM, New York, NY, 1998, pp. 67-74. DOI= <http://doi.acm.org/10.1145/277631.277644>.
- [108]. Jeffery, C., "*Program Monitoring and Visualization: An Exploratory Approach*," Springer-Verlag New York, Inc., 1999.
- [109]. Hanson, D. R., "Variable Associations in SNOBOL4," *Software: Practice and Experience*, vol. 6, no. 2, (Published Online: Oct. 27, 2006, by John Wiley & Sons, Ltd.), 1976, pp. 245-254. <http://dx.doi.org/10.1002/spe.4380060210>.
- [110]. Jeffery, C., Auguston, M., and Underwood, S., "Towards Fully Automatic Execution Monitoring." In *Proceedings of Radical Innovations of Software and Systems Engineering in the Future*. LNCS Springer. 2002, pp. 29-41. 2004. <http://www.springerlink.com/content/chwcn2wyjv35xfg8>.
- [111]. Bruneton, E., "ASM 3.0: A Java Bytecode Engineering Library," 2007; <http://asm.ow2.org/>.
- [112]. Bungale, P. P. and Luk, C., "Pinos: A Programmable Framework for Whole-System Dynamic Instrumentation." In *Proceedings of the 3rd International Conference on Virtual Execution Environments* (San Diego, California, USA, June 13 - 15, 2007). VEE '07. ACM, New York, NY, 2007, pp. 137-147. DOI= <http://doi.acm.org/10.1145/1254810.1254830>.
- [113]. Srivastava, A. and Eustace, A., "ATOM: A System for Building Customized Program Analysis Tools." In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation* (Orlando, Florida, United States, June 20 - 24, 1994). PLDI '94. ACM, New York, NY, 1994, pp. 196-205. DOI= <http://doi.acm.org/10.1145/178243.178260>.
- [114]. Shapiro, E. Y. *Algorithmic Program Debugging*. MIT Press, Cambridge, Massachusetts. 1983.

- [115]. Albertsson, L., "Simulation-Based Debugging of Soft Real-Time Applications." In *Proceedings of the Seventh Real-Time Technology and Applications Symposium (RTAS '01)* (May 30 - June 01, 2001). RTAS. IEEE Computer Society, Washington, DC, 2001, pp. 107.
- [116]. Nichols, B., Buttlar, D., and Farrell, J. P., "*PThreads Programming: A POSIX Standard for Better Multiprocessing*," (O'Reilly Nutshell), Sep. 1996.
- [117]. John R. Levine. "Linkers and Loaders." Academic Press, San Diego, CA, USA, 2000.
- [118]. Webopedia, <http://www.webopedia.com/TERM/A/agent.html>
- [119]. Al-Sharif, Z., and Jeffery, C., "Language Support for Event-Based Debugging." In *Proceedings of the 21st International Conference on Software Engineering and Knowledge Engineering (SEKE 2009)*, Boston, July 1-3, 2009, pp. 392-399.
- [120]. Al-Sharif, Z., and Jeffery, C., "A Multi-Agent Debugging Extension Architecture." In *Proceedings of the 21st International Conference on Software Engineering and Knowledge Engineering (SEKE 2009)*, Boston, July 1-3, 2009, pp. 194-199.
- [121]. Al-Sharif, Z. and Jeffery, C., "An Agent-Oriented Source-Level Debugger on Top of a Monitoring Framework." In *Proceedings of the 2009 Sixth International Conference on Information Technology: New Generations - Volume 00* (April 27 - 29, 2009). ITNG. IEEE Computer Society, 2009, pp. 241-247.
DOI= <http://dx.doi.org/10.1109/ITNG.2009.305>.
- [122]. Al-Sharif, Z. and Jeffery, C., "An Extensible Source-Level Debugger." In *Proceedings of the 2009 ACM Symposium on Applied Computing* (Honolulu, Hawaii). SAC '09. ACM, New York, NY, 2009, pp. 543-544. DOI= <http://doi.acm.org/10.1145/1529282.1529397>.
- [123]. Drusinsky, D., Shing, M., and Demir, K., "Test-Time, Run-Time, and Simulation-Time Temporal Assertions in RSP." In *Proceedings of the 16th IEEE International Workshop on Rapid System Prototyping* (June 08 - 10, 2005). RSP. IEEE Computer Society, Washington, DC, 2005, pp. 105-110. DOI= <http://dx.doi.org/10.1109/RSP.2005.50>.
- [124]. Koymans, R. "Specifying Real-Time Properties with Metric Temporal Logic." *Real-Time Syst.* vol. 2, no. 4, (Kluwer Academic Publishers Norwell, MA, USA), Oct. 1990, 1990, pp. 255-299. DOI= <http://dx.doi.org/10.1007/BF01995674>.
- [125]. Drusinsky, D., Michael, B., Shing, M., "A Framework for Computer-Aided Validation," *Innovations in Systems and Software Engineering*, vol. 4, no. 2, June, 2008, pp. 161-168.

- [126]. Drusinsky, D. and Shing, M., "Monitoring Temporal Logic Specifications Combined with Time Series Constraints." *Journal of Universal Computer Science*, vol. 9, no. 11, 2003, pp. 1261-1276. http://www.jucs.org/jucs_9_11/monitoring_temporal_logic_specification.
- [127]. Golan, M., "A Very High Level Debugging Language." Doctoral Thesis. UMI Order Number: UMI Order No. GAX94-10113., Princeton University, Princeton, NJ. 1994.
- [128]. Walker, K. and Griswold, R. E., "An Optimizing Compiler for the Icon Programming Language." *Softw. Pract. Exper.* vol. 22, no. 8, (John Wiley & Sons, Inc. New York, NY, USA), Aug. 1992, pp. 637-657. DOI= <http://dx.doi.org/10.1002/spe.4380220803>.
- [129]. Wilder, M. D., "The Uniconc Optimizing Unicon Compiler." In *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA, October 22 - 26, 2006). OOPSLA '06. ACM, New York, NY, 2006, pp. 756-757. DOI= <http://doi.acm.org/10.1145/1176617.1176710>.
- [130]. Wilder, M., and Jeffery, C., "Towards Fast Incremental Hashing of Large Bit Vectors." In *Proceedings of the 41st Annual Hawaii International Conference on System Sciences* (HICSS), 2008, pp. 474.