

Real-World Robotic Debugging

Luke F. Gumbley

A THESIS SUBMITTED IN FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF ENGINEERING IN COMPUTER SYSTEMS ENGINEERING
THE UNIVERSITY OF AUCKLAND, 28 FEBRUARY 2010

Abstract

With the growth of the robotics industry, the efficiency of robot software development is a concern. Conventional software debugging constructs are insufficient for debugging robotic software due primarily to the assumption of a deterministic, suspendable environment. What is needed is a method to extract and report information about robotic software execution while continuing execution in the real world environment. To provide this ability, a previously theorized debugging construct called a *tracepoint* is implemented within both a C and a Python debugger. Tracepoints are similar to breakpoints, but permit the evaluation of an arbitrary expression. After the expression is evaluated and reported, execution is automatically resumed. The NetBeans IDE is modified to provide a user interface for the new tracepoint functionality. As well as supporting the modified Python and C debuggers, this interface gives an extensible API intended to permit third-party developers to add support for other languages. A plugin-based visualisation system is capable of rendering trace data. Reference visualisations for laser and ultrasonic rangefinder data from the Player robot library are included. Benchmark tests show that the system compares favourably with alternative solutions — typical overheads for the evaluation of a single tracepoint are 100 μ s to 500 μ s.

The tracepoint system provides a method for a robot developer to gather information about the state of a robot control program in real time, without interrupting its operation. The information to be gathered can be specified at runtime, removing the need to re-run a test or recompile code. Full download and deployment instructions are available for Ubuntu Linux, Windows Vista, and Mac OS X.

Dedication

This thesis is dedicated to the memory of my grandfather, Frank Gumbley, who got me interested in electricity in the first place.

Acknowledgements

I'd like to acknowledge my supervisor, Dr. Bruce MacDonald, whose interest in my work has been constant from right back when I was an undergraduate. His feedback and assistance has been invaluable both in the completion of my research and the preparation of this thesis.

Thanks to my parents, Arthur and Faye, and my long-suffering sister Eve, for their love and unstinting support and for putting up with me over the last six months.

Special thanks to my girlfriend Rebecca. I started this thesis when we got together two years ago. Thank you for your willingness to help, and for giving me a great reason to finish! I think of you always.

Last but not least, thank you to the staff and students at the Department of Electrical and Computer Engineering who have helped to make the last eight years so enjoyable. Particular thanks to Allan Williamson for arranging assistance with fees, Bernard Guillemain for encouraging me to apply for a staff position after I finished my degree (and for EE303), Toby Collett and Geoff Biggs for encouraging me to pursue postgraduate study, Grant Sargent for the Shuriken robot and for being my good mate, and finally to the CS406 class of 2009 for a great introduction to lecturing.

Contents

1	Introduction	1
1.1	Robotic Debugging	2
1.2	Background	3
1.3	Objectives	4
1.4	Contributions	4
1.5	Thesis Outline	5
2	State of the Art	7
2.1	Cognitive Analysis	8
2.1.1	Summary	12
2.2	Data Gathering	12
2.2.1	Visualisation	15
2.2.2	Summary	17
2.3	Fault Localisation	18
2.3.1	Summary	20
2.4	Abstraction	21
2.4.1	Simulated Debugging	21
2.4.2	Record and Replay Debugging	22
2.4.3	Summary	25
2.5	Verification	25
2.5.1	Summary	26
2.6	Robotic Middleware	27
2.6.1	Player and Stage	27
2.6.2	ROS	27
2.6.3	RT-Middleware	28
2.6.4	Orca	28
2.6.5	Microsoft Robotics Developer Studio	29
2.6.6	Summary	30
2.7	Implications for Robotic Debugging	30

2.8	Summary	31
3	Problem and Requirements Analysis	33
3.1	Case Studies	33
3.1.1	Responses to Robot Behaviour	33
3.1.2	Blood Pressure Robot	34
3.1.3	Healthcare Assistant	35
3.2	Key Requirements	35
3.2.1	Compatibility	35
3.2.2	Real World	36
3.2.3	Usability	36
3.2.4	Extensibility	36
3.2.5	Performance	37
3.3	Analysis of Existing Solutions	37
3.3.1	Source modification	37
3.3.2	Breakpoints	38
3.3.3	Introspect	39
3.3.4	Visual Studio Tracepoints	39
3.3.5	Microsoft Robotics Developer Studio	40
3.3.6	RLog	40
3.3.7	Time Machine	41
3.3.8	Jockey	41
3.3.9	Robotic IDE	42
3.4	Analysis	42
3.5	Summary	44
4	Concept Design	47
4.1	Concept	47
4.2	Languages	48
4.2.1	Syntax and Formatting	49
4.2.2	Compilation vs. Interpretation	49
4.2.3	Debugging	49
4.2.4	Reflexivity	50
4.2.5	Type Checking	50
4.3	IDE	50
4.3.1	Eclipse	51
4.3.2	NetBeans	55
4.4	Summary	56

5	Implementation	57
5.1	Test Setup	57
5.1.1	Simulated Environment	58
5.1.2	Python/PlayerC	58
5.2	C and C++	60
5.2.1	GDB Overview	60
5.2.2	New GDB Commands	62
5.2.3	Serialization	67
5.2.4	Encoding	67
5.2.5	Type Serialization	67
5.2.6	Pointers	69
5.2.7	NetBeans Plugin	69
5.3	Python	70
5.3.1	Debugging Overview	70
5.3.2	Jpydaemon	71
5.3.3	Serialization	71
5.3.4	NetBeans Plugin	72
5.4	Tracepoint API	72
5.4.1	Class Structure	74
5.5	Tracepoint View	76
5.6	Visualisations	78
5.6.1	Overview	78
5.7	User Workflow	83
5.8	Extensibility	85
5.8.1	Adding Languages	85
5.8.2	Adding Visualisations	87
5.8.3	Adding New Features	87
6	Evaluation	89
6.1	Compatibility	90
6.2	Real world	90
6.3	Usability	90
6.4	Extensibility	91
6.5	Performance	91
6.5.1	Test System	91
6.5.2	Methodology	92
6.5.3	C Benchmarks	94
6.5.4	Python Benchmarks	101

6.5.5	Visual Studio	105
6.5.6	Summary — Performance	105
6.6	Summary	106
7	Conclusions and Future Work	109
7.1	Conclusions	109
7.2	Future work	112
7.2.1	Usability Study	112
7.2.2	Conditional Tracepoints	112
7.2.3	Data Analysis	113
7.2.4	Multiple Expressions	113
7.2.5	Configurable Update Rate	113
7.2.6	Visualisations	113
7.2.7	External Tools	114
7.2.8	Merging Open-Source Work	114
A	Deployment	115
A.1	General requirements	115
A.1.1	NetBeans	115
A.1.2	Java	115
A.1.3	C/C++	116
A.1.4	Python	116
A.2	Linux	116
A.2.1	Instructions	116
A.3	Mac OS X	117
A.3.1	Requirements	117
A.3.2	Instructions	117
A.4	Windows Vista	118
A.4.1	Requirements	118
A.4.2	Instructions	118
B	Benchmark Results	121

List of Figures

2.1	The logbook page depicting the apocryphal “first computer bug”	8
2.2	DDD visualisation of a 2-dimensional array using GNUPlot	16
2.3	DDD visualisation of an array of structures	16
2.4	Default string visualisers in Visual Studio 2008	17
4.1	Excerpt of Eclipse plugin manifest defining breakpoint menu group visibility	53
4.2	Screenshot of Eclipse Run menu showing conflicting breakpoint types . . .	54
4.3	Screenshot of “New Breakpoint...” window in NetBeans	56
5.1	Block diagram of system implementation.	58
5.2	Screenshot of Stage simulator showing test environment.	59
5.3	The <code>playerc_laser_t</code> structure	60
5.4	Screenshot of GDB showing the command-line interface	61
5.5	Screenshot of GDB showing the machine interface	61
5.6	Tracepoint demonstration with GDB CLI.	65
5.7	Tracepoint demonstration with GDB MI.	66
5.8	UML diagram of Tracepoint API class structure	73
5.9	The Tracepoints view.	76
5.10	Tracepoints view showing TraceArrayNode placeholders.	77
5.11	Screenshot of GdbLaserRenderer	80
5.12	Screenshots of GdbSonarRenderer with and without pose information . . .	81
5.13	Excerpt of Tracepoint View before and after pointer array serialization. . .	82
5.14	“New Breakpoint...” option on Debug menu.	83
5.15	Setting initial tracepoint properties.	83
5.16	Tracepoint code annotation	84
5.17	A tracepoint is set.	84
5.18	A debug session running with TracepointRenderer and Tracepoint View updating.	85
6.1	Tracepoint throughput for GDB.	96

6.2	GDB multiple tracepoint benchmark results.	97
6.3	GDB MI tracepoint overheads for structures with varying member counts.	99
6.4	GDB CLI tracepoint overheads for structures with varying member counts.	99
6.5	GDB MI tracepoint execution times with type information always serialized.	100
6.6	GDB MI tracepoint type serialization overheads.	100
6.7	Python throughput benchmark results.	103
6.8	Python multiple tracepoint benchmark results.	104
B.1	Overall GDB benchmark results	123
B.2	Overall Python benchmark results	124

List of Tables

2.1	Support tools for the Comprehension strategy [65]	10
2.2	Support tools for the Isolation strategy [65]	11
2.3	Static simulation information commands for a SystemC debugger [50]. . . .	14
2.4	Dynamic simulation information commands for a SystemC debugger [50]. .	14
2.5	Control commands for a SystemC debugger [50].	14
3.1	Analysis of existing solutions.	43
4.1	Comparison of major IDEs.	52
5.1	CLI and MI tracepoint insertion commands for GDB.	63
5.2	MI tracepoint insertion parameters for pointer serialization.	64
5.3	CLI and MI tracepoint deletion commands for GDB.	64
5.4	CLI and MI tracepoint enablement commands for GDB.	64
5.5	CLI tracepoint information command for GDB.	65
5.6	Type packet header.	68
5.7	Type packet for integral types.	68
5.8	Type packet for array types.	68
5.9	Type packet for structured types.	68
5.10	Member subpacket for struct types.	68
6.1	GDB baseline test results (μ s).	95
6.2	GDB multiple tracepoint test results (Fig. 6.2) compared with baselines (μ s).	97
6.3	GDB pointer serialization test results (μ s).	102
6.4	GDB pointer member serialization test results (μ s).	102
6.5	Python baseline test results (μ s).	102
6.6	Python per-line overhead extrapolation.	105
6.7	Visual Studio tracepoint benchmark results.	105
B.1	Raw GDB Benchmark results (in μ s, part 1).	121

B.2	Raw GDB Benchmark results (in μ s, part 2).	122
B.3	Raw Python Benchmark results (in μ s).	124

1

Introduction

Robotics is a discipline and an industry poised to transform the society in which we live. The applications of practical robotics are increasing at a geometric pace. Medical robots now permit minimally invasive surgeries [45]. Gigantic container robots automatically load and unload ships [41]. Robots have even made it into the home; the iRobot Roomba and the WowWee Robosapien, with global sales in the millions of units [28,63], are now a common sight in department stores around the world. The continuing pace of technology and research in last five years is such that the growth and potential of the robotics industry can be compared to that of the fledgeling computer industry. The recent Computing Community Consortium report “From Internet to Robotics” [16] presents a roadmap for the US robotics industry, and states unequivocally that “Robotics technology holds the potential to transform the future of the country and is likely to become as ubiquitous over the next few decades as computing technology is today.”

One major factor in the current interest in robotic technology is the ageing of the world’s population. By 2050, there will be more than 2 billion people over the age of 60 [10]. The World Health Organisation has stated that “In almost every country, the proportion of people aged over 60 years is growing faster than any other age group, as a result of both longer life expectancy and declining fertility rates” [43]. The question of how society will care for an increasingly elderly population is critical, and robotics may be the answer both indirectly in terms of increasing economic output to fund increased healthcare, and directly in supporting nursing and care for the elderly. A considerable

amount of research is currently dedicated to eldercare robotics worldwide [20, 34, 37, 54].

Considering the importance of the robotics industry both at present and in the years to come, the efficiency of robotic development is a key concern. The question of how to aid a robotic developer is complex. Previous research has shown that programmers can spend between 50% and 80% of their time testing and maintaining software [9]. The US National Institute of Standards and Technology released a report with the same estimate [1], and stated that the major problem as identified by developers was inadequate testing and debugging tools. The same report estimated that software faults cost American industry up to US\$60 billion annually — and yet despite the significant programming elements of robotic development there are few if any tools aimed specifically at aiding robot debugging. A survey of robot development environments (RDEs) by Kramer and Scheutz assigned only one criteria out of thirty-two to debugging support [33]. Within that criteria the highest mark was awarded only for the availability of the type of debugging that can be expected from modern IDEs, though special mention was made of MissionLab (as it contains limited debug automation). While the intent was to provide a broad overview of the capabilities of available open-source RDEs, the construction of criteria is indicative of the prevailing attitude towards debugging in the arena of robotics.

1.1 Robotic Debugging

One of the defining characteristics of robotics is interaction with the real world. The development of robotic hardware platforms reflects this; considerable effort is spent on real-world considerations such as battery life, actuator strength, sensor reliability and so on. When developing robot software, however, the tools and techniques used are often derived from generic software development. This approach treats a robot as merely a computer with some interesting peripherals. Consider the utility of standard debugging constructs in a robotic context. A *breakpoint* is a useful way to halt software while debugging and examine the program state. However, unless the system is running in a simulator which also halts, the real surroundings of the robot will change while its control system is halted. Variable watches and stack traces also require program execution to be suspended. This makes debugging difficult — the pause in execution results in a change in robot behaviour, an example of the “probe effect” [48] (see Section 2.4). In addition, unless the fault has a single cause visible in the program state at the point of failure, the fault must be replicated a number of times to gather enough information for a diagnosis. For computer software, operating in a largely deterministic environment, this is relatively trivial. For a robot, it can be difficult and time-consuming.

The problems with a conventional approach can be demonstrated by considering the example of a robotic driver for a car. During testing it is reasonable to assume that the

robot will be placed in a real environment, including pedestrians and other vehicles. While the car is moving it is not possible to halt the operation of the robot controller for even a brief period as this entails an unacceptable risk of collision. If the car is slowed to a stop before interrupting the controller, not only does this constitute a hazard to other vehicles but prevents examining the state of the controller at a given moment of interest (as the state is only accessible after the robot has come to a complete halt). Moreover, when the controller is restarted the state of the world will have changed; a pedestrian previously out of sensor range may be moving in front of the car, or a vehicle may be attempting to pass. This unexpected sudden change in sensor readings could cause unpredictable behaviour not relevant to the target fault.

The result of these shortcomings is that alternative robot-aware tools (such as simulators, loggers and visualisers) must be used in conjunction with the standard debugger — or the developer must create their own tools to view the state of the robot while it continues to operate. Problems exist with this approach as well; either the robot code itself is altered to permit extraction of information in real-time, or the tools examine the state of the robot without interacting with the actual program. The former approach adds development time and maintenance overhead, while the latter is vulnerable to discrepancies between the state of the robot and the state of the program. If a simulator is used then only the behaviour of the robot within the simulator can be debugged — not the behaviour of the robot in the real world.

1.2 Background

An attempt to improve robotic debugging was previously made with the “Robotic IDE” project [23]. The solution implemented was a proxy server which monitored communications between the program code and the robot hardware. Visualisations of robot sensor information were displayed to the user based on this information and were thus an accurate representation of the state of the robot as it appeared to the program. While this project was successful in that it provided a useful way for the developer to add real-time sensory information to the debugging aids within the IDE, it was lacking in several key areas:

- The system was not robust: A significant amount of configuration was required to set up the proxy server and visualisations. The system had a hard-coded interpreter for the underlying message format, which required ongoing maintenance (shortly after release a protocol change broke the system).
- The system was not complete: Only information passed along the connection to the robot could be shown; internal program logic remained opaque.

- The system was not widely applicable: Although language-independent, the system was specific to the Player robot libraries.

A more comprehensive and robust solution to the problem of analysing the internal state of a robot is required.

1.3 Objectives

The overall objective of this research is to develop a system of tools to aid robot debugging in the real world. Much of the current research into robot debugging is largely theoretical or applicable to only a small range of robotic projects. While a good theoretical basis is essential to a solution, it is often during the implementation phase that many of the problems with theory become apparent. The emphasis with this work is on practicality; the solution must be immediately useful to real robot developers. In order that this objective be met, the system design must allow for as wide a range of target languages and platforms as possible. To confirm that this is the case, two different languages are targeted for an initial implementation, which is tested across Linux, OSX and Windows. The implementation is open source and available for download, accompanied by instructions for deployment and usage.

1.4 Contributions

The major contribution of this research is the design and implementation of an open tracepoint framework for software debuggers. The framework is easy to use, places no restrictions on the developer, and comprehensive benchmarking has demonstrated that the impact on the execution of target software is minimal. Subsidiary contributions include:

- Bugfixes and improvements to the Python debugger for NetBeans
- Addition of a new Base64 encoding library to GDB
- Addition of a suite of new commands to GDB for manipulating tracepoints
- Implementation of an underlying tracepoint API in NetBeans, including:
 - A novel way of representing structured data sourced from different languages
 - A multithreaded data distribution system for consuming this data
 - A generic and extensible base for user-created visualisations
- Implementation of a number of data visualisations in NetBeans

- Modification of the Python debugging plugin for NetBeans to include tracepoint functionality
- Modification of the C/C++ debugging plugin for NetBeans to include tracepoint functionality

A paper on this work has been presented at the 2009 Australasian Conference on Robotics and Automation (ACRA) [24].

1.5 Thesis Outline

Chapter 2 examines research on the cognitive aspects of the debugging process and identifies the workflow of a programmer while debugging. The current state of the art debugging tools and systems are reviewed in the context of four primary debugging activities — Data Gathering, Fault Localisation, Abstraction and Verification. General-purpose debugging tools are considered as well as those that focus on debugging of robotic systems. The chapter concludes with a review of currently available robotic middleware.

Chapter 3 identifies common problems with robotic debugging and the key requirements of a system for examining the internal state of a robot during a debugging session. Three past and present robotics projects are examined as robotic debugging case studies to produce the criteria of an ideal solution. Nine existing solutions are evaluated with respect to their satisfaction of the identified criteria.

Chapter 4 details the concept of the proposed solution to the problems in the previous chapter. The design decisions for an implementation of this concept are outlined and the rationale behind the final decisions is explained.

Chapter 5 covers the implementation of the proposed solution. An overview of the system is initially given before implementation details for each system element are discussed.

Chapter 6 is a comprehensive evaluation of the system. Test programs in both C and Python are benchmarked with and without tracepoints, and with a wide variety of target expressions. The tests are replicated across three different operating systems, and the results are compared to some of the alternative solutions evaluated in Chapter 3. Other evaluations are suggested for future work.

Chapter 7 summarises the results and contributions of the thesis, and summarises whether or not the final system meets the requirements laid out in Chapter 3. Some possibilities for future work to expand and improve the system are also given.

2

State of the Art

Debugging is the process of correcting undesired behaviours (“bugs”) in a system. The use of the term “bug” to describe a fault dates as far back as 1875. In the 15 August 1875 issue of *The Operator*, a magazine for telegraphers, the failure of a quadruplex telegraph is described. The “bug” in this case was an improperly adjusted rheostat. A more famous “computer bug” origin story comes from Grace Hopper in September 1947. A moth had become trapped between the contacts of a relay in the Mark II computer at Harvard University, causing it to fail. The moth was removed and taped into the logbook along with the phrase “first actual case of a bug being found”. A picture of this page can be seen in Figure 2.1. Bugs, and the process of debugging, have thus been a part of the development of any system since well before computers were invented.

In this chapter the current state of the art in debugging and robotics middleware is examined. This review begins with research on the cognitive aspects of debugging — attempts to understand the process of debugging within the mind of the user. The debugging process itself is then examined, with the work split into categories based on four different debugging tasks — Data Gathering, Fault Localisation, Abstraction and Verification. Finally, a brief summary of current robotic middleware is presented.

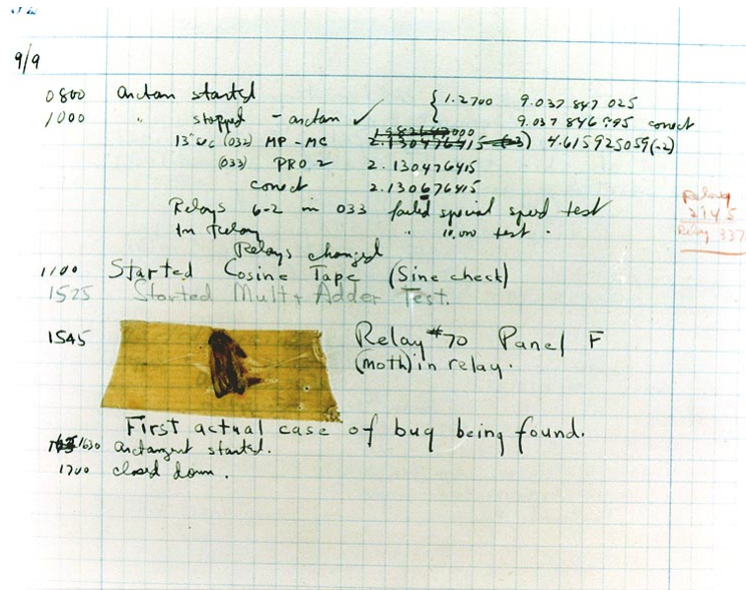


Figure 2.1: The logbook page depicting the apocryphal “first computer bug”.

2.1 Cognitive Analysis

In order to derive methods for assisting a developer while debugging software, it is important to understand the processes of debugging within the developer’s mind. This section therefore investigates research into the cognitive aspects of debugging. Debugging is a singularly demanding mental process. Xu and Rajlich demonstrate this by examining debugging in the context of *Bloom’s Taxonomy of the Cognitive Domain* [64]. Bloom divided cognitive learning activities into six levels of increasing difficulty and complexity. The levels were associated with verbs indicating the types of activities that take place at that level. The levels are:

Knowledge: Acquire, identify, recognize, define, name

Comprehension: Explain, describe, interpret, illustrate

Application: Apply, relate, use, solve, construct

Analysis: Analyze, categorize, contrast, discriminate

Synthesis: Create, design, specify, propose, develop, invent

Evaluation: Validate, argue, judge, recommend, justify

By analyzing a case study involving debugging an improperly implemented web server script, Xu and Rajlich demonstrated that debugging involves activities from all six levels of Bloom’s Cognitive Domain. This indicates that debugging is a difficult cognitive task,

and helps to explain why the process of debugging is difficult to fully automate. In addition to the intellectual challenges of debugging, Cheung and Black suggest that programmers also find it emotionally disturbing, as it requires the admission that their programs are imperfect [11].

Osterlie and Wang suggest that debugging is analogous to any problem-solving situation and advance the concept of *Sensemaking* as an explanation of the process of formulating and then solving the problem [44]. Debugging is described as starting with the question “What is going on?” rather than “Which way do I solve this problem?”. This approach highlights the fact that in debugging the problem is often initially unknown and action must be taken to clarify and describe it before it may be understood. This could be viewed as the initial step in the debugging process — the analysis of program output to determine whether or not a fault is present.

Although this output is usually textual, the information is sometimes rendered graphically. Romero et al. examine the correlation between debugging accuracy and the use of available visualisations [51]. The IDE in use in the study presented a blurred view except for a small unblurred section which users could move around. By monitoring the focus of users during debugging they show that programmers of different skill levels display a markedly different use pattern of visual aids. Novice programmers concentrated almost exclusively on the code, intermediates showed peak visualisation use, and advanced debuggers used the aids sparingly. Differences in “Chunking” ability (see below) are suggested to account for the disparity between intermediate and advanced, in that advanced users more effectively process and retain the information gleaned from one viewing.

Chunking is described by Vessey as the clustering of relevant elements in long-term memory which can then be referred to by a single item in short-term memory [59]. Experience helps to build up useful “chunks” of debugging data, for example common bugs and their resolutions. This mental rapid referencing system permits more efficient bug resolution. Evidence of this ability being of key importance in debugging was gathered in a study of the differences in approach between novice and expert debuggers. Experts were also found to change their approach based on the situation, whereas novices tended to use a more uniform approach.

Yoon and Garcia postulate two complementary overall strategies that programmers use during debugging, referred to as “Comprehension” and “Isolation” [65]. The *Comprehension* strategy is a top-down approach, mentally abstracting and visualising the program structure in an attempt to locate the point at which the program’s design differs from specification. The *Isolation* strategy is an analysis of the error itself — the programmer attempts to locate the specific point in the program at which the error appears and works backwards to determine the ultimate cause of the fault. By analysing the debugging process in this context, they present several goals related to each strategy and a number

Goal	Tools
Requirement Identification/Analysis	Program description chart Design chart Specification list
Program Comprehension	Hierarchical program display Specification list Control flow diagram Data flow diagram Procedure call stack Interprocedural relation chart Cross-reference Visualization, Animation Class browser
Discrepancy Detection	On-line manual Design chart
Information/Clue Gathering	Breakpoints Single stepping Watchpoints State viewing

Table 2.1: Support tools for the Comprehension strategy [65]

of tools that would support each goal. These goals and tools are presented in summary form in Tables 2.1 and 2.2.

Araki et al. and Zeller both explain debugging in similar ways, as a process of generating and refuting hypotheses from available data [4, 67]. Zeller identifies this approach as the scientific method by comparing a program failure to an unexplained natural phenomenon [67]. The programmer must generate a hypothesis about the cause of the failure, and then make predictions based on that hypothesis. Tests are then performed to confirm or refute the validity of the predictions, leading to the hypothesis being accepted or discarded. In this way, the root cause of the fault is narrowed down until the fault itself is located and corrected.

Ko and Myers formulate and discuss the “Interrogative Debugging” paradigm [29]. They contend that rather than assisting programmers to evaluate their hypotheses, debugging tools should focus first on helping programmers construct their hypotheses. This is based on the assertion that every debugging session begins with a “Why?” question, and that programmers often formulate incorrect hypotheses. By making assumptions about program behaviour, a programmer is prone to forget to consider all potential causes of a

Goal	Tools
Symptom Identification	List of symptoms Detailed error messages Warning messages Breakpoints Single stepping Core dump State viewing
Symptom-Cause Identification	Bug library Interprocedural relation chart
Information/Clue Gathering	Breakpoints Single stepping Watchpoints State viewing
Bug Hypothesis Generation	List of current symptoms Bug library Library of bug-symptom associations On-line programming language manual
Bug Hypothesis Verification	Control flow diagram Data flow diagram Breakpoints Watchpoints
Bug Hypothesis Modification and Refinement	Input/Output relation chart Execution history Hypotheses history

Table 2.2: Support tools for the Isolation strategy [65]

symptom. Interrogative debugging is an attempt to allow programmers to ask natural-language questions about the operation of a program and be presented with a number of possible hypotheses. Development and testing of the *Whyline*, an interrogative debugging tool for the Alice programming environment, demonstrated that the tool was useful in a simplistic environment. The Whyline gives an array of possible questions based on a list of system objects and the properties of those objects — “Why didn’t <object>’s <parameter> change to <value>?”. This can be viewed as an intuitive way for the user to filter the source code so that only sections with some bearing on the problem are examined. The authors identify several problems with the system in contemplating the shift to a more complicated real-world programming language (such as Java), the primary issue being the enormous array of potential user questions.

2.1.1 Summary

Debugging begins with the *Sensemaking* process, an analysis of the program output to identify fault symptoms. Programmers then move between the complementary *Comprehension* and *Isolation* strategies, formulating and testing hypotheses about the nature and location of the fault. Information about the program structure and fault symptoms is held within the mind of the programmer in discrete interrelated *Chunks*, permitting rapid recall and analysis of the gathered data. While fully automated debugging is not currently possible due to the cognitive complexity of the task, it is possible to design tools to assist in the gathering and interpreting of program data in order to free the programmer’s mind for more complex reasoning.

2.2 Data Gathering

The term “Data Gathering” in this context refers to the monitoring of a system during a test run. The information gathered in this way is used throughout the debugging process — in the detection of the initial fault, the formulation and verification of hypotheses as to the cause of the fault, and in evaluating the effectiveness of any solution. There are a wide variety of methods a developer could use to gather data from a system both within and without the system itself; for example, the physical behaviour of a robot could be recorded externally and later analysed for aberrant behaviour. Where this did not provide enough information, a debugger could be used to permit the interruption of its internal control program and the examination of the program state. Textual output or log files could be generated for real-time or post-mortem review. These and many other debugging approaches are explored by Cheung and Black [11]. They identify a total of seven different techniques, including Output Debugging, Tracing, Breakpoints, Assertion

Execution, Controlled Execution, Replay and Monitoring. Yoon and Garcia list similar aids under the classification of “Information/Clue Gathering”, however they also identify an additional nineteen debugging aids (see Tables 2.1 and 2.2), many of which are not commonly available [65].

The actual approach taken by a developer is dependent on which methods are possible for the hardware and software platforms in question, as well as the developer’s own preferences and level of experience. As already mentioned in Section 2.1, Romero et al. examined the correlation between debugging accuracy and the use of debugging aids. Their finding was that novice users made little or no use of debugging aids, intermediates the most use of the aids, and advanced users made sparing use of aids and visualisations. Murphy et al. examined the strategies of novice debuggers specifically and discovered that only two of the twenty-one subjects made use of a debugger, whereas more than half used `printf` statements to monitor program flow and variable values [40]. This suggests that for many developers, the use of `printf` is considered more effective than currently available debugging tools. Chmiel and Loui refer to this approach as *Output Debugging* [12].

While techniques such as output debugging are supported directly by the programming language and operating system in use, the use of some techniques is dependent on the presence of an underlying debugger which manages the execution of the program. A wide variety of such debuggers exist, including the GNU C debugger (GDB) and Microsoft’s Visual Studio debuggers. Such debuggers provide data gathering tools like breakpoints, watches, stack traces and memory inspection. The origin of these tools can be traced back to the seminal 1951 paper on software debugging by Gill [22]. It is interesting to note that although more than half a century has passed since this initial work, few additional debugging aids are available in a modern IDE such as Visual Studio.

One notable new data gathering tool in Visual Studio is the *tracepoint*. This takes the form of a “when hit” option for a standard breakpoint. Breakpoints with a specified “when hit” action are referred to as tracepoints and represented by a different glyph next to the target code. The user can specify that a message be printed or a macro executed, as well as whether or not execution should continue or halt. Placing or editing these tracepoints does not require recompilation of the target code. While this feature is available both in the commercial and free (“Express”) editions of Visual C++, the macro function is disabled in the Express edition. The printed message can include arbitrary expressions and special variables, however little control is given over the output format. If a structure is printed, only the first three members are shown, followed by an ellipsis. No type information is given. This ability can be seen as a combination of breakpoints and output debugging — the user can specify information to be output at a point in the program without changing program code and without halting program execution.

lss	list all signals in given hierarchy
lsm	list all modules in given hierarchy
lse	output all events instantiated in modules
lsio	list I/O interface in given hierarchy
lsb	list all bindings of specified channel

Table 2.3: Static simulation information commands for a SystemC debugger [50].

lpt	list all trigger events of all processes
lst	output code line a process is currently pending
lpl	show all processes listening on given event
lsp	output all thread and method processes

Table 2.4: Dynamic simulation information commands for a SystemC debugger [50].

Crawford et al. claim that the general lack of innovation in debugging tools is caused by a focus on the design of interfaces to existing techniques, rather than the expansion of the underlying debugging languages [17]. They outline a new procedural debugging language, GDL, and detail how it can be used to define common debugging tools (watches, program traces etc.). This approach may be thought of as writing a smaller program to examine and modify the execution of a larger in order to isolate faults and test solutions. This approach of writing higher-level debugging programs to automate basic tasks is also taken by Rogin et al. [50]. The major goal of this research was to provide a debugger for SystemC, a C++-based concurrent-process modelling language. GDB was first extended to make it aware of SystemC-specific features such as signals, ports, processes and events. The resulting debugger uses GDB-style syntax but works with SystemC features. Studies were then undertaken at AMD of common debugging activities, and a list of helpful high-level commands was created and then implemented in a mix of shell scripts and C++ libraries. A summary of these commands can be found in Tables 2.3, 2.4 and 2.5. Further work by Rogin et al. expanded on this base by adding limited automation — the user can specify commonly found “debug patterns” that can either be triggered by specific events or manually run by the user [49]. The specified pattern assumes a particular fault and guides the user in gathering the information necessary for a positive diagnosis of the assumed fault.

ebreak	break on specified SystemC event
rcbreak	break on rising edge of given clock
fcbreak	break on falling edge of given clock
pstep	break on next invocation of given process
dstep	break on next simulation delta cycle
tstep	break on next simulation time stamp

Table 2.5: Control commands for a SystemC debugger [50].

2.2.1 Visualisation

Visualisations of the data gathered during a debugging session are often used to display information in a more useful form. This is a particularly useful approach for robotic debugging; the sheer volume and different modalities (sound, vision, distance etc.) of information gathered by robot sensors could otherwise be difficult to process. Kooijmans et al. developed the *Interaction Debugger*, a debugging tool to assist in the development of robots meant to interact with human beings [30]. During a test run, data from every robot sensor is collected. The Interaction Debugger permits detailed analysis and annotation of these records across seven different types of data — Sound, Vision, Person Identification, Motion, Body Contact, Distance and Robotic Behaviour. During analysis, the data is examined to determine trigger events for robot behaviours and human reactions to those behaviours. The Robovie robot is used in four case studies to confirm the effectiveness of the tool.

The need for visualisations in robot development is explained by Collett et al. in terms of Human-Robot Interaction (HRI) [14, 15]. One of the main barriers to communication and interaction between humans and robots is said to be due to each having different “perceptual spaces”. These differences lead to a “lack of understanding about the robot’s world view”, and it is primarily this lack of understanding that makes robot software development challenging. To resolve this issue, the use of Augmented Reality (AR) is advocated and a toolkit for the purpose is developed. Augmented reality in this case refers to the layering of robot sensor information over images of the real world environment from which the information is gathered. This permits the human developers to understand the sensor information in context and thus more easily spot errors. An expansion of this system capable of visualising the complicated state information of a Simultaneous Localisation and Mapping (SLAM) algorithm is presented by Kozlov et al. [32].

GDB has a graphical front end called the Data Display Debugger (DDD). This tool provides a graphical user interface to the debugger and permits the visualisation of any information collected during the session. The most powerful visualisation is capable of displaying variables and their interconnections in a block diagram (see Figure 2.3 for a screenshot). In this diagram, the user can dereference pointer values with a double-click. Another visualisation permits the display of numeric data in a plot using the `gnuplot` application. A screenshot of this function can be seen in Figure 2.2. The important thing to note about these functionalities is that they only operate when the debugging session is halted — values are not updated during execution.

Visual Studio .NET also permits the creation of custom debug visualisations. Any object, excluding objects of type `Object` and `Array`, can have a custom visualisation attached. The user must create a class library containing a subclass of `DialogDebuggerVisualizer` and override the `Show` function, as well as include an assembly attribute in the source that

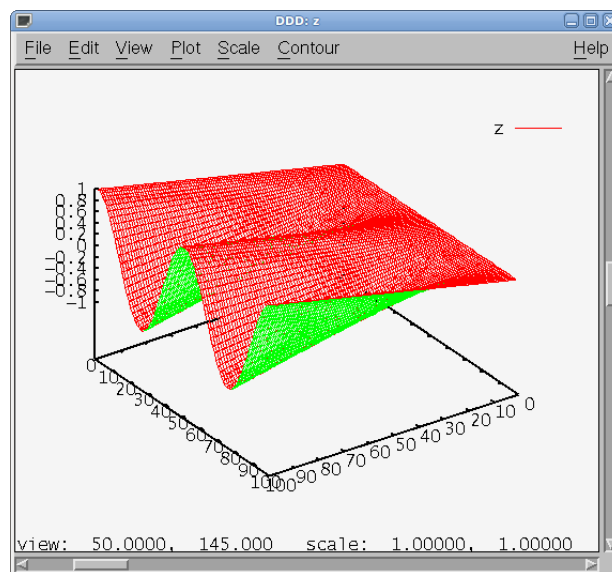


Figure 2.2: DDD visualisation of a 2-dimensional array using GNUPlot

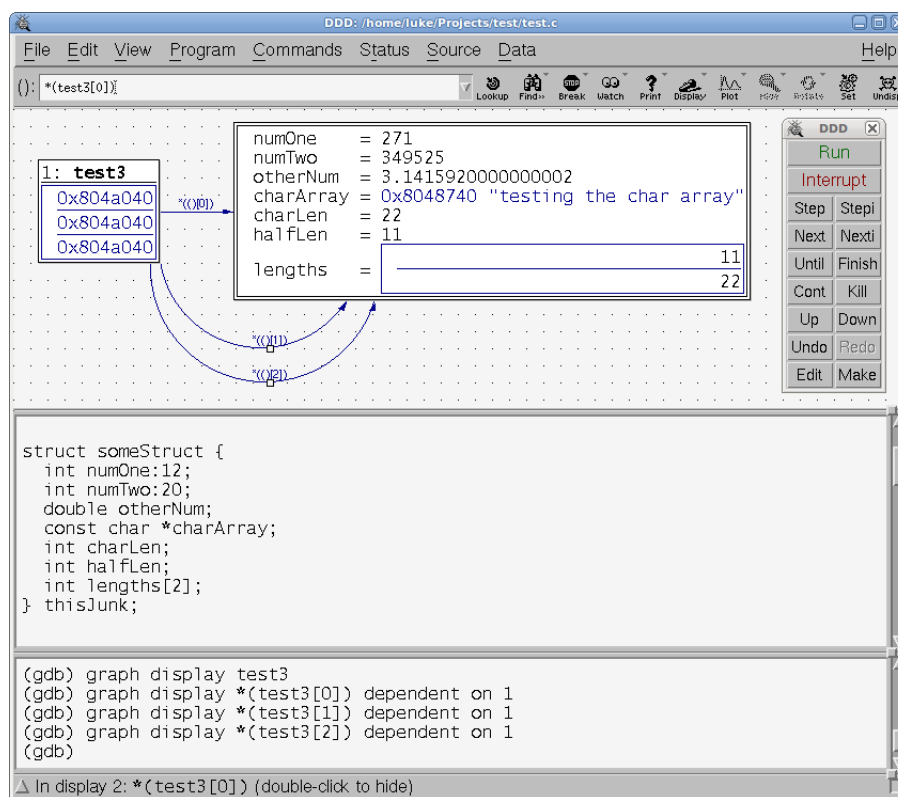


Figure 2.3: DDD visualisation of an array of structures

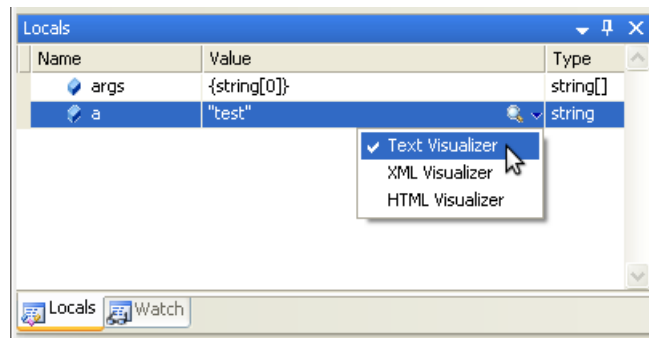


Figure 2.4: Default string visualisers in Visual Studio 2008

indicates which object types can be visualised by the new visualiser. After compilation, the class library is placed in the user's Visualisations directory. The user can now access the visualisation via the small magnifying glass glyph in the Watch or Locals window (see screenshot in Figure 2.4). The visualiser opens in a separate window. Similarly to GDB, these visualisations are not updated on-the-fly, and in fact must be reloaded manually while the system is halted in order to update the displayed value.

2.2.2 Summary

Data gathering is the collection of information about a program during its execution. This data is used by the developer to form hypotheses about program behaviour. Data can be gathered from a program in many different ways. Output debugging is one of the most popular methods, consisting of the inclusion of (often temporary) output statements into program code. Other methods of data gathering include the execution of the program within a debugging tool. This permits the use of breakpoints, where the execution of the program is halted upon reaching a target location. Once halted, the memory of the program can be examined to retrieve information about the program state, for example variable values or the call stack. Tracepoints are a similar concept to breakpoints, available in Visual Studio and used to output information without halting the program.

As the amount of data gathered during a test run can be overwhelming, and consist of many different types of data (particularly in robotics), visualisations are often used. Visualisations present the gathered data in a form where the information represented can be more efficiently assimilated by the developer. Both GDB and Visual Studio provide tools to permit the visualisation of gathered data.

2.3 Fault Localisation

Fault Localisation occurs after a fault has been detected, and is the process of narrowing down the ultimate cause of the displayed symptoms. This usually refers to the gradual elimination of sections of code unrelated to the fault symptom. One of the simplest ways of doing so is to consider only the code that was actually executed during a test run, as this code is absolutely guaranteed to contain the cause of the fault. The subset of a program's code that is executed is referred to as the program's *coverage*. The relative ease with which this information can be gathered makes coverage analysis a highly valuable and often utilised technique for fault localisation.

Zeller details an algorithm for localising faults called “Delta Debugging” [66, 67] as well as an implementation of the algorithm, the *Wynot* tool (from “Worked yesterday, not today”) [66]. The Wynot system is given a set of “circumstances” under which execution of a program results in a fault. These circumstances could be input data, a change in the source code, or particular input from the user. The system automatically (and blindly) removes a portion of the given input and re-runs the test. If the same fault is produced, more data is removed. If no fault is produced, a different alteration is tried. The goal is to find the smallest and simplest subset of the initial circumstance set under which the fault is produced. This process continues until no further data can be removed. One case study involved a bug in the Mozilla web browser. When a particular web page was displayed the browser crashed. The Wynot tool reduced the HTML required to cause the crash from 896 lines to a single line that exhibited the same fault. This permitted rapid identification of the underlying fault in the browser. Matsushita et al. expand on Delta Debugging with the DMET (“Debugging Method”) and DSUS (“Debugging Support System”) tools [38]. These tools take into consideration issues with collaborative development (i.e. where a bug is introduced by separate changes to more than one source file) and provide the ability to resolve errors by automatically identifying and excluding erroneous software revisions.

An equivalent approach to Delta Debugging was taken in the development of the ChipperJ tool by Sterling et al. [55]. ChipperJ is an automated tool for performing “Program Chipping” (i.e. the gradual removal or chipping away of program elements) on Java applications. The system attempts to reduce a program to a minimal subset of its original code while preserving a given symptom. It is syntax aware and applies different minimization strategies depending on the type of code block. The techniques are deliberately very simple. The algorithm has definite drawbacks, including the difficulty of distinguishing the desired symptom from potentially identical symptoms generated as a result of the chipping process. Integration with a debugger for coverage awareness, i.e. knowledge of which statements were actually executed in a test run, has the potential to significantly improve results.

A more directed form of Delta Debugging known as *Program Slicing* is advanced by Weiser [60, 61]. This is a method of debugging by first finding a subset of the program source, through static analysis of the code, that influences the outcome of a statement or the value of a variable. This reduced subset can then be debugged separately to the main program. It is suggested that this is the mental process that developers use when debugging in order to concentrate on statements that are actually relevant to a fault. Weiser presents a method for automatically creating a program slice given specific criteria (referred to as a “window”) that is equivalent to the original in the sense that it will produce the same output when viewed through the same window as was specified. This window consists of variable values as observed at particular points in the code. Thus if the user specifies a window that contains a faulty variable value in normal execution, the generated slice will include only the code required to generate this faulty value.

Further work on slicing is presented by Zhang et al. in which the concept of a *Dynamic* program slice is introduced [69]. This combines the technique of program slicing with coverage analysis. In a dynamic slice, only executed code that influenced the value of a given variable is included. Three different algorithms for performing dynamic slicing (*Data*, *Full* and *Relevant*) are tested and evaluated. A Data slice includes only executed statements on which the value of the target variable was directly (i.e. mathematically) dependent. A Full slice is the next most comprehensive, and includes conditionals that affected the execution of statements in the Data slice. The final and most inclusive slice method, the Relevant slice, takes into account the possibility that statements that were not executed (and thus not in the Full slice) should have been, and includes the conditionals that excluded them. As slicing in general seeks to reduce the input to the developer while still capturing information relevant to the fault, the algorithms are rated by the size of the produced slice (smaller being better) and whether or not the slice contains the faulty code. The only algorithm that consistently contained the fault was the Relevant slice, which during testing was able to decrease target program size to between 12% and 62% of executed statements.

In a compromise between static and dynamic slicing techniques, Wong and Maldonado suggest using static program slices to debug SDL [62]. These slices are augmented with statistical data gathered by recording the coverage of debug runs against whether the output of the run was “correct” or “incorrect”. The programmer can then examine code in order of probability of containing a fault.

Liu et al. also advocate the use of data mining to localise faults [36]. By instrumenting call and branch operations and performing Bayesian analysis on data from one “passing” and one “failing” execution, they attempt to localise logical faults. The approach is novel in that it has been designed to provide useful information with only two program executions. While the technique performs well in individual artificial tests, it does not

provide for debugging programs with multiple faults. It would likely be most useful as one of a number of probabilistic techniques for error localisation, rather than as the sole aid. Dallmeier et al. suggest a similar technique, tracing and comparing program flow in java applications in order to localise faults to particular methods [18]. A tool implementing the technique was created and integrated with the Eclipse development environment, relying on JUnit testing to gather data on program runs. A slightly less rigorous method is used for calculating the ranking of potential problem spots, but as the system relies on multiple “passing” execution runs for base data, the results are still useful. Another flow-comparison method by Zeller utilises Delta Debugging techniques to isolate the error-causing input data as well as more accurately localise the fault [68].

Nikolik discusses a novel approach called “Convergence Debugging” [42]. This approach records the results of conditional statements across a set of test executions (each with different input data). The success or failure of each test is recorded. One failing test is selected as the reference for comparison with the others. For each test a notional distance to the reference test run is calculated using the Pythagorean Theorem, with the number of true and false results of each conditional as co-ordinates in an arbitrary space. This space has $2n$ dimensions, where n is the number of conditionals instrumented. The distances thus calculated represent convergence or divergence between test cases, suggesting related test cases worthy of closer attention. This method is novel primarily due to the focus on the test case rather than the program code.

2.3.1 Summary

Fault localisation is the process of finding the section of code that contains an error. Delta debugging is a technique used to narrow the circumstances in which a fault manifests. Such circumstances can consist of source code alterations as well as input data. Monitoring the coverage of the target program (i.e. recording the statements that were actually executed) during debugging allows for immediate narrowing of the focus of the developer to just that code which could have contributed to a fault. Adding data gathered from multiple test runs permits the developer to look first at the code which is statistically the most likely to have caused a fault. Program slicing is the generation of a subset of program code. Statements are included or excluded in a slice based on the influence they have on the value of a given variable. This technique can be applied statically, without executing the program, or dynamically, incorporating information about program coverage. Program slicing automates to some extent the process of elimination which the developer would otherwise perform mentally.

2.4 Abstraction

Often when a fault appears in a system, the symptom allowing the developer to identify the fault is observed some time after the actual cause. The developer must work backwards from the symptom to identify what caused it. This involves repeatedly generating the fault, moving backwards in the chain of causation each time (known as *Cyclic Debugging*). This can be difficult for a number of reasons. Firstly, the fault can be intermittent. If it occurs unpredictably then it can be difficult to systematically gather information about it. Secondly, the fault could be caused by the interaction of program threads (such as a deadlock where threads compete for the same resources). If this is the case then halting a thread or instrumenting code can affect whether or not the fault exhibits any symptoms at all.

Abstraction in this case refers to the ability to decouple the execution of a test run from the real world — so the test can be re-executed any number of times and still exhibit exactly the same symptoms. There are two main approaches for achieving this abstraction. *Simulated* debugging refers to the execution of the test within a simulated environment. In this case the test code can be paused or halted, or any amount of logging or data output code added, without compromising the virtual execution speed or behaviour of the program. The other approach, referred to as *Record and Replay* debugging, is to attempt to instrument the test program in such a way as to minimally affect its execution. Replay of the test code can then be performed using a simulator fed with the logged data. A key concern in record and replay debugging is the degree to which program modifications (for the purposes of data gathering) affect program execution. This is referred to as the *Probe Effect* [8].

2.4.1 Simulated Debugging

The idea of simulating a computer for debugging purposes is explored by Ho et al. [26]. The authors propose a “pervasive debugger” where the programs to be debugged execute within a virtualized environment, which is itself executing within a debugger. In this way the entire system can be halted and the state of any process within the virtual environment can be examined. This permits both *Horizontal debugging*, where multiple processes can be debugged simultaneously, and *Vertical debugging*, where not only the state of the process but also that of the underlying operating system can be examined.

The use of simulators in robot development is widespread, thanks to the relatively low cost and simplicity of a simulator over a real robot. Rister et al. developed a system for recording and replaying the execution of swarming robot control systems within a simulator [48]. They implemented a comprehensive and easily used system for recording the value history of variables as well as complete stack traces of the system. Debugging is

compiled in to the target code — variables and classes of interest are tagged and a script adds stack tracing code before compilation. Visualisations permit the user to watch the system during a run as well as replaying the events after a run completed. The distinctions made between this system and a traditional debugger like GDB are “time sensitivity” and “thread sensitivity”, i.e. the system was aware of events across a distinct time period and across multiple threads. The idea of “causal splicing” or “causal tracing” is advanced as a method of tracking the events that caused a variable to be a certain value (equivalent to the Dynamic slice proposed by Zhang et al. [69]). Disadvantages are the high (90%) overheads involved. Although the use of a simulator means these overheads will not affect the execution of the program within the simulated environment, this system could not be expected to work as well for recording test runs in the real world. Another debugging-oriented simulator is presented by Moores et al. [39]. A robot simulation architecture was developed that includes the ability for the user to mark simulated variables for watching or logging as well as the capability for the user to attach custom code to be executed upon specified events.

Microsoft Robotics Developer Studio includes the “Visual Simulation Environment”, which permits software development and testing without the presence of an actual robot. Robots are fully simulated in the virtual environment, which permits 3D terrain. The physics simulator is capable of being hardware accelerated with the AGEIA PhysX processor. The software being tested is unaware of the difference between a simulated run and a real-world run. Debugging facilities are limited to the recording of robot movements for later playback — however this does not also record the internal state of the robot, unlike the simulator developed by Rister et al. [48].

2.4.2 Record and Replay Debugging

Distributed systems present a unique debugging challenge. Conventional debugging permits the developer to halt and examine the state of a process. When the system comprises multiple processes running on different systems, examining the state of the entire system at a discrete point in time is much more difficult. Kortenkamp et al. developed a suite of tools for deterministic logging and analysis of distributed systems [31]. The individual elements of a system connect to a central timekeeping server, which calculates the timing offsets for each component. This provides a common time base for all logged data, necessary for later analysis. At key points in the program, the developer adds calls to the provided *rlog* function (similar to *printf*), capable of taking a number of basic C types as a parameter. This call is compiled into the code before a test run. The data gathered in this way is timestamped and logged to a database (one per component). Later, the individual databases are collated taking into account the timing differences between components. The system also includes tools for later analysis of the data. A syntax called *Interval*

Temporal Checking Logic is outlined, capable of applying and testing timing constraints to the logged data. Testing was done in Red Hat Linux. The longest average execution time of a call to `rlog`, 7 ms, was recorded when collecting information about a TCP/IP socket. The shortest was 90 μ s, for logging *null*. This system focuses on post-mortem analysis of a test run and does not include the ability to replay.

Thane and Hansson describe the *Time Machine*, a similar initial theoretical method for deterministic logging but based around later replay of a recorded test run [56]. Thane et al. expand on this approach, describing an improved method including benchmarking results from implementation testing [57]. This work concentrates on embedded systems. Data collection happens at two levels. On one level, the developer manually instruments the code, marking key event locations where non-deterministic interactions occur. The program state is logged immediately before and after these events. System interrupts are automatically logged. These event logs are then used to replay the execution of the system such that precisely the same behaviour is exhibited. During replay, custom breakpoints are transparently inserted into the code at logged event locations so that the replayed state can be modified to simulate the external interactions. Between events the code executes as normal. In this way the system can be debugged as if it is running within a simulator, with data gathered from a run in the real world. While a major goal was source code transparency, the authors acknowledge that in some cases source code modification may be necessary in order to use the system to best advantage. An IDE was modified to expose the replay functionality. While the system has been tested within a number of different environments, the authors list some baseline requirements of the target: that it executes within an emulator or an RTOS with instrumentable hooks, and that a debugger supporting scripted breakpoints is available.

It is important to note that the approaches so far mentioned impose conditions and alterations on system code and must be implemented right from the start of development. This makes them unsuitable choices for debugging existing systems. Additionally, the logging overhead must be considered in the hardware design stage as the logging systems must remain present in the final production output. In this way these systems avoid many of the implications of the probe effect, as the logging code is part of the final system. The disadvantage of this approach is that logging overheads are still present in the system even when the outputs themselves are no longer used.

Saito discusses Jockey, a record/replay debugging tool that can be used with any Linux application [52]. It gathers information in two main ways. First, a shared library is preloaded that shadows operating system function calls. Preloading ensures the Jockey function is found before the actual system call. The function records the parameters before passing them on to the system function itself. On return, the return value is also logged, before finally returning control to the target. Secondly, CPU instructions with non-

deterministic effects are overwritten with a call to a Jockey function and similarly logged. This effectively avoids the need to alter the original source code and attendant developer overhead. On replay, the intercepted function calls and instructions are reproduced with the data collected during the recording. The overheads exhibit considerable variation, topping out in extreme cases at 30% (where a large amount of I/O is involved) but in other cases dropping so low as to be unmeasurable. Practical testing showed that systems like Jockey are most effective for diagnosing faults that “exhibit quickly” — where the detectable symptoms of the fault follow soon after the fault itself. Using the replay technique to diagnose faults that propagate across a number of interconnected systems was cumbersome and not a significant time saving for the developer. Jockey is limited to execution on Linux systems, however it does not specify the target language (as it doesn’t technically work with the target code at all). Similarly to previous systems, Jockey does not support the output of debugging information during a session, instead recording information about a test run for later replay.

Snyder and Blandy developed Introspect, an extension to GDB specifically for debugging embedded systems [53]. Three steps are identified for analyzing a program at runtime — *Specifying*, *Collecting* and *Analyzing* program data. GDB is identified as ideal for *Specifying* and *Analyzing* data as it has strong natural language lexical and symbol analysis capabilities, but not for *Collecting* data (as this would necessitate the transfer of large amounts of data over a relatively slow serial link). Introspect is a combination of extensions to GDB and to the GDBServer protocol for debugging remote targets. GDB is used to specify the data to be collected. This information is passed to the target, running an implementation of the GDBServer stub. The stub handles data collection and logging, and then at the completion of a “trace test”, the data is transferred back to GDB. At this stage the user is free to analyse the data collected at various points during execution as if the program had hit a breakpoint at this stage. The completeness of the snapshot depends on the data gathered, however provision is made for the collection of stack and frame implementation as well as evaluating arbitrary expressions. Although the GDB extensions were included in the main program, the GDBServer stub was never released. There are currently no available GDBServer stubs that support tracepoints. This may be due to the complexity of the scripting language used to specify logging actions to be taken at tracepoints.

De Sutter et al. also modified GDB in order to provide “backtracking” and dynamic patching functionality [19]. With backtracking, the user can specify “checkpoints” at which the state of the program is saved (in reality, the debugged process is halted and a child process forked in which debugging continues). Later the user can return to the earlier “checkpoint” in order to examine program execution more closely, for example just before a terminal error occurs. Dynamic patching permits the user to modify and

recompile a program while it is being debugged, replacing the live version with the revised version and continuing execution. The majority of the functionality of the new GDB was provided by existing GNU tools and the operating system itself, the inference being that implementing these useful tools should be easy in the majority of cases.

2.4.3 Summary

Abstraction is the separation of a test run from any non-deterministic effects that would produce different results on re-execution. This can be achieved either by simulating the system on which the test is running (*Simulated* debugging), or by recording non-deterministic interactions and re-creating them in a deterministic fashion later on (*Record and Replay* debugging). Problems with the latter approach occur when modifications to the program in order to record these interactions affect the performance of the program, altering or removing fault symptoms. This is referred to as the *Probe Effect*. The three stages of debugging via Abstraction are *Specifying*, *Collecting* and *Analyzing* data. *Backtracking* or *Reverse Debugging* refers to the ability of some replay tools to move execution backwards in order to examine previous program states.

2.5 Verification

System faults can be thought of as behaviour that departs from the system specification. Verification is the process of checking that the system meets the requirements of the original specification. A wide variety of methods are used for this purpose, from manual code review through to unit testing. Reviewing the design of the system itself makes sense as this is usually a clearer and less complicated representation than the actual system code. In this way problems with the system specification can be identified without the more expensive and time-consuming code debugging process [62].

Chmiel and Loui studied the debugging habits of students, and advocate more comprehensive teaching of debugging techniques as students learn to program [12]. They suggest individual code review, the creation of “personal checklists” of commonly occurring faults, and the keeping of a comprehensive log of errors. During the study, students discovered that more time spent in the design phase decreased the time spent coding and debugging.

A complementary approach is to formalise the design specification by writing it in a system design language such as the Specification and Description Language (SDL) that can itself be executed and debugged. This is the approach advocated by Wong and Maldonado [62]. Similarly, Liang and Xu suggest “scenario-based” debugging, where the actual behaviour of a program is compared to a specification written by the user in their java-based debugging language [35]. In the case studies, the bug is first localised

manually. A “behaviour specification” for the potentially buggy section of code is then written in order to further narrow down the bug location. The developer contrasts its operation with the original in order to find a bug.

One method for automatically verifying a specification is presented by Reiss [47]. The authors outline the design of CHET, a system for loosely testing programmer obedience to rules on component usage. A programmer writes a Java component and then sets rules for it in the CHET specification language. CHET is then run on a Java program which uses the component. The program is first abstracted into a flow and data representation, which is then tested to ensure it satisfies the component usage rules. Execution time is an issue, with the abstraction taking up to half an hour for the most complex program tested (Jalopy, a source code formatter for Java). Possible speed improvements could come from incremental abstraction during program development.

Hovemeyer and Pugh contend that although there has been considerable research in the area of debug automation, few of these techniques have found widespread adoption [27]. They advocate a less complicated approach, developing a tool called *FindBugs*. This tool is capable of locating simple bugs “one step removed from a syntax error”, such as infinite recursion or null pointer exceptions. It does so through a static analysis of system code, searching compiled bytecodes for bug patterns using the open-source Bytecode Engineering Library (BCEL) developed by the Apache Software Foundation. The sections flagged could be compared to compiler warnings in that not all are in error. The hit rate is estimated at about 50% true faults [13]. This could be seen as an automation of the technique advocated by Chmiel and Loui where the developer keeps a “personal checklist” of common faults to check for before more comprehensive debugging is attempted [12]. Some conclusions reached were that more work needed to be done on integrating new debugging techniques with IDEs, including incremental and background processing to prevent the long computation times of probabilistic bug localisation algorithms.

Unit testing is a commonly accepted method of program verification. The user writes tests based on the system specification. These tests provide inputs to the system and check against the correct outputs as defined in the specification. This is of particular use in collaborative development of large systems, where modifications are made to part of a system that inadvertently affect another part.

2.5.1 Summary

Verification is the process of ensuring that a program meets its design specification. This can involve the creation of a formal specification in a system design language such as SDL. Such a specification can then be used in a number of automatic verification techniques, as well as providing a benchmark against which the behaviour of the actual program can be compared. It is also possible to test the code of a program against a number of constraints

in order to identify usage errors before they manifest as faults in a test run. Unit testing is another common method of verifying that the behaviour of a system is as designed.

2.6 Robotic Middleware

Much of the development effort in creating a robot lies in the writing of the system software. Writing this software to interact directly with a specific robot hardware platform raises two primary questions. First, there is unlikely to be a simulator available for the target platform that presents such a low-level interface. Second, if the targeted hardware platform should change or become unavailable, changes must be made to the system software in order to interact with the new target. For these reasons, many developers prefer to add a third-party layer between the robot and their software. This layer is referred to as robot middleware. In this section some of the more widespread robot middlewares are examined.

2.6.1 Player and Stage

Player/Stage is an open-source system first released in 2000 [21, 58]. It comprises three primary elements:

- *Player*, a server that runs on and abstracts robot hardware.
- *Stage*, a simulator that can be used in place of a real robot.
- Client libraries for connecting to Player from a number of languages.

The Player Server reads from a configuration file specifying the interfaces available to clients (sensors, motion etc.) and the drivers and parameters that provide those interfaces. Drivers exist for hundreds of devices and a wide range of different robot platforms, including the popular MobileRobots Pioneer for which the system was originally developed. These interfaces are exposed to the client via TCP/IP. Many different clients can connect to a single server. In this way, `player_v` can be used to view sensor information from a robot that is being controlled by a different program. Player/Stage also comes with a number of command-line tools. One such is `player_v`, used to connect to a robot and provide an interface for a user to control it directly (as well as view sensory data). Player does not provide any debugging tools beyond the simulator and visualisations.

2.6.2 ROS

The Robot Operating System is a recently released robot programming framework being developed under the auspices of Willow Garage [46]. It is described as a “meta-operating

system” for a robot, abstracting robot hardware as well as providing peer-to-peer communication between system elements (referred to as “Nodes”). The central tenet of the system is promoting open source and code reuse in robotics. To this end the project has a number of internal repositories. Users can upload the nodes they develop or download nodes they wish to improve or reuse. The other major facet of the system is the focus on providing easy communication between distinct nodes. This makes the framework particularly suitable for distributed systems. A central “Master” node co-ordinates communication between nodes using the XML Remote Procedure Call protocol (XMLRPC). Publisher nodes inform the master of the information feeds (“Topics”) they publish. Subscriber nodes request a particular topic from the master and are provided a list of nodes that publish that topic. The two nodes then negotiate a connection medium with each other using XMLRPC. After this, the subscribing node can establish a direct connection to the publishing node to receive the information feed. The loose coupling of nodes permits them to be easily swapped into and out of a running system, minimizing the difficulty of debugging a single element. The system also contains built-in logging and replay capabilities at the node level, a significant asset for debugging.

2.6.3 RT-Middleware

Robot Technology Middleware (RT-Middleware) is an open standard for developing and working with RT-Components, intended to be reusable modules of code for robot development [2,3]. OpenRTM-aist is an implementation of this standard. The standard has been in development since 2002 by the Japanese National Institute for Advanced Industrial Science and Technology (AIST). Each RT-Component is intended as a standalone module with input and output ports, which are connected to the ports of other RT-Components. Communication between components is via the Common Object Request Broker Architecture (CORBA), which abstracts communications as function calls to remote objects. Debugging in OpenRTM-aist is provided by the “RTCLink” tool, capable of monitoring and logging communications between components.

2.6.4 Orca

Orca is similar to ROS and RT-Middleware in that it is a robotics framework designed around the idea of individual interacting *Components* (processes) that combine to form a robotic system. These components, for example hardware abstractions or algorithm implementations, can then be reused in other projects as needed. In Orca1, components exchanged information in the form of *Objects*, with communications provided by the Orca *Transport Mechanisms* (CORBA, TCP/IP etc.) and *Communication Patterns* (server-push, client-push and so on) [6]. A component’s *Interfaces* (each a combination of a

transport mechanism and a communication pattern) were defined in an XML configuration file. Orca2 now makes use of the Ice middleware for defining and providing interfaces between components [7, 25].

The two major points on which Orca differentiates itself from other robotic frameworks are the emphasis on the component-based approach, and the consideration of the commercial aspects of component development [7]. Due to the different licensing used for core components and the main libraries, it is possible to build systems that comprise both closed- and open-source components. This permits developers to sell their components if they wish to do so.

2.6.5 Microsoft Robotics Developer Studio

Microsoft Robotics Developer Studio is a collection of frameworks to assist robot development. There are two main ideas on which it is based — “Decentralized Software Services”, DSS, and the Concurrency and Co-Ordination Runtime (CCR). DSS is the middleware layer that goes between the code controlling the robot and the robot itself. CCR is a number of software tools to streamline working with multiple processors, jobs and information sources. In addition to this there is a new language (and tool) called the “Visual Programming Language” or VPL. This permits software design using a graphical interface, dragging and dropping components and connecting them together to form robot control systems. The system also includes a simulator, the Visual Simulation Environment, discussed in more detail in Section

Debugging facilities for Microsoft Robotics Developer Studio are those provided by the user’s IDE, typically Visual Studio. There is one exception — when the Visual Programming Language is used, debugging facilities are only available through a web-based interface. This interface automatically opens when a “Debug Start” is requested, but can also be accessed independently from the VPL GUI. This interface shows pending activities and permits the user both to step through them and to add breakpoints. Information about the system is shown independently of the VPL GUI itself, such as an image of the graphical elements that make up a particular activity. All information about the state of the system can be retrieved through this interface. While these debugging facilities are similar to those found in modern IDEs, there are several ways in which they fall short. The web-based interface is only fully compatible with Internet Explorer (tests with Mozilla Firefox and Google Chrome failed to display correctly). Although this interface does permit remote debugging, it is much more difficult to use than a standard IDE-based debugger. Breakpoints are not persisted between sessions, and can only be set on a “Pending Action” when the system is halted. While they can be enabled and disabled, no conditional breakpoints or hit counting is possible.

2.6.6 Summary

While the frameworks reviewed are only a subset of the available robotic middlewares, some conclusions about the future of robotic development can be drawn based on the characteristics they share. With the exception of Player, the two most important areas targeted by these frameworks are the modularity and reusability of robotic code, and the provision of easy-to-use communication protocols to facilitate distributed systems. Every system is able to be used across a network. Possible reasons for the general ubiquity of Player in robotics are its relative age and simplicity — its been around for long enough that abstractions for many robotic systems are available out of the box. An active user community provides constant improvements to documentation, bug fixes and tutorials. It is interesting to note that despite current research efforts into debugging hard and soft real-time operating systems and algorithms (such as that of Thane et al. [56, 57]), support for these techniques at the middleware level is limited. Another aspect in which middleware is limited is debugging support; only ROS includes significant debugging support. It is clear from the proliferation of middleware that any tool for assisting robot debugging should be middleware-agnostic.

2.7 Implications for Robotic Debugging

Attempting to apply generic software debugging approaches to robotics is problematic in two main ways (as discussed in Section 1.1). Firstly, halting the execution of a robot controller to examine its state can have safety implications as well as potentially affecting the behaviour of the robot. Secondly, reproducing faults requires reliable and consistent reproduction of the circumstances of their appearance, which can be both difficult and time consuming to achieve. In terms of the four categories of debugging activity, the first problem is related to Data Gathering and the second to Abstraction. This suggests that a tool for assisting robot developers should aid either Data Gathering or Abstraction. Interestingly, all the research on robot debugging summarised in this chapter falls into one of these two categories.

Data Gathering and Abstraction are closely related activities. Of primary importance to both is the probe effect (see Section 2.4). In record-and-replay debugging, an Abstraction activity, Data Gathering is used to collect information about non-deterministic interactions with the real world. Data Gathering in general is not so specific, permitting the user to directly specify the data to be collected. The major difference between the two activities is the emphasis by Abstraction on collecting information for later review, whereas Data Gathering permits immediate feedback during a test. For these reasons — flexibility and immediacy — the focus of this work is to be on assisting the Data Gathering activity in robotic debugging.

2.8 Summary

Debugging is the process of correcting undesired behaviours (“bugs”) in a system. It can be seen as the application of the scientific method to software faults. Programmers observe fault symptoms, hypothesise about the root causes and then gather information and modify code to confirm or refute their hypotheses. This process continues until the fault is resolved. Tools to assist in debugging automate basic tasks to permit the developer to focus on higher-level reasoning.

Debugging tools can be placed in four general categories - Data Gathering, Fault Localisation, Abstraction and Verification. Data gathering retrieves information about program execution and presents it to the developer. Fault localisation attempts to direct the developer to probable fault locations given some basic evidence. Abstraction assists in the reliable reproduction of faults by compensating for non-deterministic interactions. Verification compares program code and execution to a pre-defined specification and flags any differences for attention.

Robotic middleware is a layer between robot hardware and software that permits software to be used on different hardware platforms. A number of different robot middlewares exist, the primary goals of which are the promotion of code reusability and modularity. At present no clear standard middleware has emerged.

The two primary debugging activities in robotics are Data Gathering and Abstraction. Of the two, Data Gathering is the primary candidate for tool development as it is a more flexible approach that provides immediate feedback to a developer during a test run.

3

Problem and Requirements Analysis

If Data Gathering is an activity in robotics which has been neglected, it remains a very broad area. To narrow the focus of this work and assist in the design of a tool for robot debugging, this chapter first identifies some of the most common problems in robotic debugging. This is done through the analysis of three robotic research projects previously undertaken by this research group. These case studies, along with current research explored in Chapter 2, are used to build a list of requirements for a tool that would have been of some assistance in debugging the faults experienced during the projects. A number of existing tools are then evaluated by how well they fit these requirements.

3.1 Case Studies

In order to develop a list of requirements for the design of a tool to assist robot debugging, three robotic projects undertaken by this research group are studied in terms of their debugging requirements. The first was a project on human-robot interaction. For the others, the primary area of research is the design and programming of robots to assist in caring for the elderly [37].

3.1.1 Responses to Robot Behaviour

A study was carried out on the psychological response of humans to differences in robot behaviour [5]. A MobileRobots B21R robot was used, with an internal computer running

Ubuntu Linux. The Player robot middleware was used to abstract the control and sensory input of the robot. A rough figure-of-eight course was laid out and participants were told they had to guide the robot around the course as quickly as possible. The robot had been programmed to detect the presence of a human guide using a 180 degree infrared laser rangefinder. Two behaviours were available — a “good” behaviour, where the robot followed the nearest object (assumed to be the guide), and a “bad” behaviour where the robot moved away from the closest object, or in a random direction. Tests were filmed and later the participants were questioned about their feelings and responses to the actions of the robot.

The original code development was with a simulated robot, using the Stage simulation system. Testing with the real robot in advance of the study was limited. One problem that emerged during the study itself was that the laser sensor used to detect obstacles was unable to detect the tables around the edge of the room. This meant that if the user led the robot too close to a table, a collision was quite likely. The robot did not notice these collisions and continued to attempt to move through them. This resulted in damage to the robot and the tables, and the corruption of some results.

3.1.2 Blood Pressure Robot

A shortage of trained nursing professionals has led to the idea of automating some of the menial data-gathering tasks in order to free up the human staff for more difficult work. An initial proof-of-concept implementation was completed some time ago, comprising a MobileRobots “PeopleBot” robot with an Omron M10-IT blood pressure monitor [34]. The robot had a monitor which displayed a rendered face, and a speaker system and voice synthesiser to interact with patients. The monitor and synthesizer were driven by an onboard computer running Debian Linux. The robot sensors were interfaced through an instance of the Player server running on the same machine. Patients were approached by the robot, greeted, and given instructions as to how to attach the blood pressure monitor. The robot then activated the monitor and took the patients’ blood pressure before reporting the results.

In order to determine the psychological response of patients to interacting with a robot, a study was carried out whereby a number of patients were brought in to interact with the robot and have their blood pressure taken. Following this, they were to fill out a questionnaire about what they thought of the experience. Each robotic consultation was videotaped for later analysis. A technician was sometimes on hand in case difficulties were experienced with the robot, but as a technician was not always available, a set of instructions was given to the researchers carrying out the experiments. This enabled them to start up the robot and prepare it for the scripted user interaction.

While the software controlling the robot had logging capabilities coded into it, these

capabilities were far from comprehensive. Problems with the robot, though rare, only became apparent at the point where individual sessions had to be abandoned. Suspicions as to the cause of faults could not be verified without independent testing or adding code to output state information, neither of which were possible between sessions with patients. Interaction with the robot during a session was not possible as it would affect the independent psychological study taking place. What was required was a way to gather data from arbitrary locations in the software during a test session, without interrupting the session. Instructions to help avoid certain failure modes could then be given and the integrity of the remaining sessions preserved until the robot could be decommissioned for patching.

3.1.3 Healthcare Assistant

A study currently being undertaken involves a similar scenario to the blood pressure robot in the previous section. The robot now in use is a Yujin Robotics CAFERO, with two internal computers. The first runs a custom Linux operating system for hardware real-time control. The interface to the robot itself is through the Willow Garage ROS middleware. The other computer runs Windows XP to provide a user interface. The two machines are connected via TCP/IP. The user interacts with the robot via a touchscreen, displaying a menu system written in Adobe Flash. This robot has a far wider range of functionality than in the previous project, including such abilities as face recognition, medicine reminders and entertainment in addition to direct sensor data gathering. Two new sensors are employed; a different type of blood pressure monitor as well as a blood oxygen monitor. The robot is currently being used in a study on the acceptability of robotic healthcare assistants, on-site at an aged care facility. With this robot the problems previously faced are exacerbated by the development and testing systems being located on separate machines to the robot; interface programs are developed and tested externally before being loaded into the robot for final verification without debugging tools.

3.2 Key Requirements

Based on the case studies in Section 3.1 and the work surveyed in Chapter 2, a number of key requirements have been identified. The following sections list these requirements and explain how they were derived and what they mean in terms of the final solution.

3.2.1 Compatibility

The compatibility of a tool is a measure of the extent to which it can be applied to existing robotic systems without modifying those systems. This implies cross-platform operation.

In the third case study the robot contained computers running two different operating systems. Compatibility with multiple programming languages and robot middlewares is also necessary. The first and second projects studied used the Player middleware, but the third used ROS. These differences demonstrate a classic problem with developing tools for robotics — a lack of standardization. A debugging solution able to meet the requirements of all three projects must be able to work with different host hardware, operating systems, sensors, programming languages, and middleware. Many existing tools also require modifications to the source code of the software being debugged (for example many record-and-replay tools require this; see Section 2.4.2). In all three of the case studies, debugging needs were not considered until the systems themselves had already been implemented. A tool that required a rewrite of the target software before it could be used to find bugs would not have been useful.

3.2.2 Real World

The tool must be capable of debugging in the real world. Many techniques for debugging abstraction eliminate the probe effect from consideration by relying on debugging robots within a simulated world (see Section 2.4). Such debugging is sufficient for testing basic algorithms but cannot debug problems with the real robot, just as happened in the first case study. Requiring a simulator for debugging also precludes the use of hardware and middleware for which a simulator is not available.

3.2.3 Usability

The usability of a tool or technique is related to the amount of time the developer must spend on it in order to get useful output. As identified by Kooijmans et al. (see Section 2.2.1), robots gather a great deal of information from the many sensors they employ to observe their surroundings [30]. An individual tool must usually be started to show the output for each sensor (Kooijmans et al. aimed to create a unified tool). During the projects detailed in the previous section the management of such tools constituted a significant overhead for the developers. Tool management was also identified as an issue in previous work on robot debugging [23]. To minimize this overhead, the debugging tool developed should integrate with the user's debugging workflow.

3.2.4 Extensibility

Robotics is defined by enormous variability in hardware and software. Even assuming that a system could realistically be developed which contained built-in support for every robotic platform ever created, the pace of robotic research ensures that new systems will

constantly be emerging and old systems evolving. The first project studied in the previous section was undertaken in 2007, the second in 2008 and the third in 2009. Each used a different robot platform. Although the Player middleware was used for the first two, in between it underwent a significant protocol and interface change. Given these factors, in order to ensure ongoing compatibility, the tool must be extensible to provide for new and unforeseen platforms.

3.2.5 Performance

Data gathering must have a minimal effect on program execution. It must not change the behaviour of the software being examined (i.e. exhibit the probe effect, see Section 2.4)) or its utility as a debugging tool is compromised. It is also unacceptable to affect the performance of the final release system after debugging is complete. Some existing solutions compromise on this point, including the debugging code as part of the final system in order to avoid the probe effect. The three robot projects used output debugging and breakpoints, which have very high and very low efficiency, respectively.

3.3 Analysis of Existing Solutions

Using the criteria outlined in the previous section, this section analyses nine existing tools and techniques for robotic debugging. Each existing solution was evaluated on a per-requirement basis. For each requirement a nominal score between 1 and 5 was given for how well the solution satisfied the requirement. The features of the best solutions are used as a basis of the design of a new tool. These systems were chosen based on the research in Chapter 2. Following the individual analyses, a summary of the findings can be seen in Table 3.1.

3.3.1 Source modification

A popular method of retrieving data from a program is to simply modify the program to output the required data (see Section 2.2). While straightforward to implement and requiring nothing more than a way of outputting data from the target system, this solution involves modifying the source code and thus adds significant development overhead. Its primary strength is in ubiquity; the ability to debug programs using this method is almost always available.

Compatibility $\frac{5}{5}$ Virtually any system is capable of outputting data. While this does involve modifying the source code, it is not a fundamental change to the system.

Real world $\frac{5}{5}$ Source modification can produce outputs in the real world.

Usability $\frac{2}{5}$ Output code must be manually inserted, the system recompiled and then later the code must be removed. This adds considerable development overhead. With the exception of languages that permit reflection, outputs also cannot be changed at runtime, requiring that tests be re-executed.

Extensibility $\frac{3}{5}$ Although source modification is technically extensible, it provides no framework on which a user can build.

Performance $\frac{5}{5}$ As the instrumentation is compiled into the target, performance can be near optimal — depending on the developer. Poorly coded output implementations can have significant overheads.

3.3.2 Breakpoints

Where a debugger is available, information about the internal state and operation of a program can be gained by specifying locations where execution is interrupted and the stack and memory of the program examined (see Section 2.2).

Compatibility $\frac{5}{5}$ The use of breakpoints requires a debugger, which is usually readily available.

Real world $\frac{4}{5}$ Breakpoints can be used as easily in the real world as in simulation, however it is in the real world where their drawbacks become most apparent.

Usability $\frac{5}{5}$ Breakpoints are a familiar and easy to use construct for most developers. It is reasonable to assume that an IDE with debugger integration provides breakpoints, but even without an IDE debuggers themselves provide symbol tables to ease breakpoint insertion (as is the case for GDB). No modification to system source is required.

Extensibility $\frac{3}{5}$ Some debuggers provide the ability to execute arbitrary commands when a breakpoint is struck, but this behaviour is not universal, nor is it consistent between those debuggers that do implement it.

Performance $\frac{1}{5}$ In this area, breakpoints are extremely poor. A breakpoint halts program execution until the user resumes it. If the user is examining the program state, the interruption is in the order of seconds, if not minutes. In a real-world application, this is rarely without consequence, at the very least in terms of program behaviour.

3.3.3 Introspect

Introspect is the name of a GDB extension that provides limited record/replay functionality for remote debugging targets [53] (see Section 2.4.2). The user indicates the data to be monitored, and then executes a test. After execution, the gathered data may be examined by GDB in the usual manner, as if the system had struck a series of breakpoints.

Compatibility $\frac{3}{5}$ Introspect is an extension to GDB and primarily supports only C and C++. It also requires the use of a remote platform that implements the tracing protocol — and no such platforms exist at present.

Real world $\frac{5}{5}$ The design of the system was intended specifically to permit the retrieval of data from embedded systems operating in the real world.

Usability $\frac{3}{5}$ As Introspect is a GDB extension, using it is roughly analogous to setting and clearing breakpoints. However by contrast with breakpoints, no IDEs currently integrate with this functionality — only a command-line interface is available. It is also not possible to change the data being gathered during a test.

Extensibility $\frac{1}{5}$ The system was not designed to be extensible in any way — its capabilities cannot be extended by the end-user.

Performance $\frac{5}{5}$ One of the design goals of the system is minimal disturbance to program execution.

3.3.4 Visual Studio Tracepoints

Microsoft Visual Studio permits the user to specify that execution continues immediately after a breakpoint is struck, as well as permitting the output of arbitrary expressions and the execution of IDE macros. When a breakpoint is modified to use this functionality, it is referred to as a *tracepoint* (see Section 2.2).

Compatibility $\frac{4}{5}$ While Visual Studio tracepoints are supported for every language in the suite, Visual Studio itself is still restricted to the Windows platform.

Real world $\frac{4}{5}$ Tracepoints do not require the use of a simulator, although they are only accessible within the IDE.

Usability $\frac{5}{5}$ The IDE includes options for setting and clearing tracepoints, and converting breakpoints to tracepoints. Tracepoints are clearly indicated within code by glyphs and highlights, and can be edited and changed after placement.

Extensibility $\frac{1}{5}$ While the execution of macros is configurable on a tracepoint hit, this behaviour is only supported by the commercial version of the IDE. Otherwise, expression results are output inline with the program's own output and are not accessible to the developer in any other way.

Performance $\frac{3}{5}$ Visual Studio tracepoints incur a significant overhead, tested in section 6.5.5 and found to be approximately 62 ms per hit.

3.3.5 Microsoft Robotics Developer Studio

Microsoft Robotics Developer Studio (see Section 2.6.5) permits the insertion of standard breakpoints into Visual Programming Language (VPL) code only, via a web interface. All other debugging is provided by Visual Studio.

Compatibility $\frac{2}{5}$ VPL breakpoints may only be set in that language, and only interacted with through Internet Explorer. VPL can only be executed on .NET-compatible platforms.

Real world $\frac{4}{5}$ Although MRDS provides a simulator, VPL breakpoints can be set in real hardware.

Usability $\frac{3}{5}$ The breakpoints are not placed using the IDE. The functionality is instead accessed through a web interface, which makes instrumenting the code quite difficult. They are also not persisted between debugging sessions.

Extensibility $\frac{1}{5}$ VPL breakpoints are not extensible.

Performance $\frac{1}{5}$ Performance is very poor. The difficult interface leads to the program spending even more time halted than with conventional breakpoints.

3.3.6 RLog

The user instruments their code with a call to the *rlog* function [31] (see Section 2.4.2). The function transmits the given data to a central server where it is logged for later analysis.

Compatibility $\frac{3}{5}$ The logging function is in C, although this does not necessarily preclude the use of other languages capable of calling C libraries. The system was also only tested in Linux. It also requires the connection of system components via TCP/IP and the provision of a database for each component.

Real world $\frac{5}{5}$ The system is designed to test distributed software running under real-world conditions.

Usability $\frac{3}{5}$ The developer must manually instrument the code, and any changes to instrumentation require a recompile. No IDE integration or automation was mentioned.

Extensibility $\frac{3}{5}$ The ability for the logger to output data in different ways permits significant reuse of the RLog libraries even if the timestamping and logical analysis functionality is not required. The developer would still need to write their own routines to receive and interpret the data, as no library was provided for this purpose.

Performance $\frac{4}{5}$ Test results from Kortenkamp et al. showed the longest delay was 7 ms on average (for transmitting log data over a TCP socket) [31]. Logging to an SQL database took an average 3 ms per call.

3.3.7 Time Machine

Non-deterministic program interactions are recorded either by specialized hardware or manual software instrumentation [57] (see Section 2.4.2). These logged events are then later used to allow a full replay of the logged test run within a modified IDE.

Compatibility $\frac{2}{5}$ The authors state that the system has been successfully used with a number of different operating systems, hardware platforms, compilers and debuggers. The authors list as requirements a real-time operating system and/or dedicated debugging hardware. Although a target language requirement is not given, logging functions for C are the only ones specified. The test outlined in the paper was in the C language, and ran within the VxWorks RTOS.

Real world $\frac{5}{5}$ The system was designed for real-world execution — it records a real-world interaction for later replay in a deterministic requirement.

Usability $\frac{4}{5}$ If debugging hardware is not available, the user must manually instrument their code with the required logging functions. While this is less than ideal, the replay functionality does have full IDE integration. A screenshot shows the replay abilities and visualisations added to the IAR Embedded Workbench.

Extensibility $\frac{1}{5}$ The system was not designed to be extensible.

Performance $\frac{4}{5}$ Overheads for the presented case study by Thane et al. are given as up to 3% of CPU time during a test run [57].

3.3.8 Jockey

Function calls in the target program with non-deterministic execution times are instrumented by injecting logging code with a preloaded shared library [52] (see Section 2.4.2).

Compatibility $\frac{3}{5}$ The system was written and tested in the Linux environment, however code insertion through shared libraries is equally possible in the Windows environment. Any target language that makes use of shared libraries is supported.

Real world $\frac{5}{5}$ As with most record/replay systems, real-world execution is targeted.

Usability $\frac{5}{5}$ Code instrumentation, logging and replay is completely automatic.

Extensibility $\frac{1}{5}$ Jockey was not designed to be extensible.

Performance $\frac{3}{5}$ The authors state that performance is not a core goal of the system as long as it does not change the behaviour of the program being debugged [52]. Correspondingly, benchmark figures vary — the overheads can be so low as to be unmeasurable, or as high as 30% of the CPU time of the target program. Jockey is not intended to be used outside the debugging process.

3.3.9 Robotic IDE

The “Robotic IDE” was an attempt by Gumbley and MacDonald at exposing the program state by intercepting TCP/IP communications between the robot and its control program [23] (see Section 1.2).

Compatibility $\frac{3}{5}$ While the target language and operating system is nonspecific, only the Player robot middleware was supported. Only TCP/IP was implemented as an information-gathering method.

Real world $\frac{5}{5}$ Both Real-world and simulated interactions could be visualised.

Usability $\frac{3}{5}$ Considerable configuration was required to set up the proxy server. No other interaction with the developer or source code was required. The system was integrated with the Eclipse IDE.

Extensibility $\frac{4}{5}$ Gathered data was distributed using an internal API which was exposed to developers. Alternate visualisations could be added.

Performance $\frac{5}{5}$ An unmeasured (assumed to be negligible) lag was introduced into the network connection between the robot and the control software..

3.4 Analysis

The overall results are shown in table 3.1, ranked by highest overall score. The high ranking of the Source Modification and Breakpoint solutions is corroborated by their use

	Compatibility	Real world	Usability	Extensibility	Performance	Overall score
Source modification	5	5	2	3	5	20
Robotic IDE	3	5	3	4	5	20
Breakpoints	5	4	5	3	1	18
RLog	3	5	3	3	4	18
Introspect	3	5	3	1	5	17
Jockey	3	5	5	1	3	17
VS Tracepoints	4	4	5	1	3	17
Time Machine	2	5	4	1	4	16
MRDS	2	4	3	1	1	11

Table 3.1: Analysis of existing solutions. Each solution is rated out of five for each requirement.

in all three case studies. Neither is an ideal solution, however; Source Modification falls short in terms of usability, while Breakpoints fall short in performance. Neither scored well for extensibility. Their selection as debugging tools in the case studies was primarily based on compatibility; for this requirement they scored highest of all the solutions analysed.

The most usable solutions were Breakpoints, Jockey and Visual Studio Tracepoints. None of these solutions required any modifications to program source code. Breakpoints and Tracepoints include IDE integration, streamlining debugging workflow. The operation of Jockey is completely automatic, involving only an extra parameter when executing the target system.

The best solutions in terms of performance were Source Modification, the Robotic IDE and Introspect. In the case of Source Modification, performance requires additional effort by the debugger, but Introspect and the Robotic IDE were designed specifically to have a minimum impact on the program being debugged. The poorest solutions were based around breakpoints — this is due to the requirement for the developer to manually resume execution.

Extensibility was a requirement not met well by any of the solutions. The Robotic IDE came closest, permitting the addition of custom data visualisations. More than half the solutions provided no framework for extension whatsoever.

3.5 Summary

An analysis of three different robotics projects is used to determine a list of common problems in robotics. Some problems identified in this way are:

- A lack of real-world testing.
- An inability to detect fault conditions before system failure.
- An inability to gather information without modifying code.
- An inability to gather information during testing.
- The use of separate development environments for development and testing.
- A lack of commonality in robot platforms.

By combining the identified issues with the research surveyed in Chapter 2, the key requirements for an ideal robot debugging tool are formulated. They are comprised of Compatibility, Real world capability, Usability, Extensibility and Performance. Nine existing tools are rated using these criteria to measure their suitability for use in robotic debugging. None of the tools are ideal, the best examples being Source Modification, the

Robotic IDE, Breakpoints and RLog. While all solutions score well for their ability to operate in the real world, only the Robotic IDE has significant capacity for extension.

4

Concept Design

This chapter presents the design of a debugging tool which satisfies the requirements identified in the previous chapter. The key design questions of which languages to target in the initial implementation and which IDE to support are addressed in detail along with the rationale for the choices made.

4.1 Concept

The proposed solution is a novel implementation of the Tracepoint debugging aid, combining elements of the Visual Studio tracepoint implementation and the Robotic IDE project. A tracepoint is similar to a breakpoint in that it is tagged to a particular location in the code and takes action when execution reaches that point. However, instead of halting execution, an attached statement is evaluated by the debugger between steps and the result reported to the user as execution continues. Cheung and Black define “Tracing” as follows:

“The tracing technique uses a standard trace facility supplied by the operating system, compiler, or programming environment to display selected information. The trace facility tracks execution flow or object modification and reports relevant changes at defined times.” [11]

A similar debugging activity called *watching* is defined by Yoon and Garcia as:

“Isolating specific variables and keeping track of their changing values as the program runs.” [65]

A limited implementation of this concept has been available in Visual Studio since 2005. The primary problems with this implementation, as identified in the previous chapter, are specificity to the Windows operating system, a lack of extensibility, and significant debugging overheads.

Tracepoints provide an ideal solution for a number of reasons. Like breakpoints, the concept is applicable to any imperative language. Unlike breakpoints, tracepoints have the potential to avoid the probe effect as program execution is not halted. The tracepoint concept also satisfies the real world requirement as they do not require the use of a simulator. Because tracepoints are a function of the debugger, they are placed and edited after compilation and do not affect or even require the presence of the source code. The only necessity is that the target program must be compiled with debugging information attached. This is necessary as tracepoint expressions are evaluated using the debugger’s internal symbol table.

As tracepoints are conceptually similar to breakpoints, a similar user interface can be added to an IDE to promote system usability. As cross-platform operation is a goal, the IDE to be augmented must support multiple operating systems. The IDE must also be modular and itself extensible, to permit extensibility of the overall system. The system should also provide for visualisations for robot sensor information within the IDE, to reduce the number of external tools required by the developer and increase the usability of the system.

The robot middleware used by target software is not a factor for system compatibility, as tracepoints gather data at a lower level. Any open-source debugger which has the ability to place breakpoints and evaluate expressions can be made to support tracepoints. Thus, conceptually, the solution is not specific to any one operating system, middleware or language. This is important in order to satisfy the requirement of compatibility, although individual debuggers will still need to have tracepoint support added.

4.2 Languages

The initial system release targets the C and Python programming languages. The reasons for these choices are as follows:

- They are imperative languages (necessary for the use of tracepoints).
- They are popular, helping to ensure the system is compatible with as many projects as possible on release.

- They are cross-platform, identified as a system requirement in Section 3.2.
- Open-source, cross-platform debuggers are available for both.
- They are significantly different languages

The fact that the two languages are different is significant as one requirement is to provide a debugging tool that is language-agnostic. By ensuring the design concept works with two such different languages, this requirement is satisfied. Some of these aspects in which Python and C differ are discussed below, along with the implications of these differences for system design.

4.2.1 Syntax and Formatting

Significant differences in syntax may affect the definition of expressions to attach to breakpoints. For example, C code is not sensitive to line breaks or whitespace whereas the structure of Python code is sensitive to indentation.

4.2.2 Compilation vs. Interpretation

C is a compiled language, meaning that all source code is translated to machine code that runs directly on the CPU. Python is an interpreted language — although source is compiled to byte code, the byte code is still executed by the Python interpreter. This has significant implications for debugging. To modify a C program after compilation requires a significant degree of knowledge about the platform targeted by compilation, as well as information about how the source was compiled (symbol tables, etc.). Conversely, modifying a Python program is a trivial process.

4.2.3 Debugging

C programs are debugged by an external debugger. This debugger usually executes the target program as a child process in order that it have access to the program's memory. The debugger interrupts execution of the program by sending signals to the program's process. While execution is halted, the debugger can examine the memory of the target program and rewrite portions of the program's machine code in order to provide breakpoint functionality.

Python has a completely different debugging style. The Python interpreter provides hooks into its operation by way of callback functions that can be set to run at different points during interpretation, for example before each source line is executed. The “debugger” consists of just these functions and the code to set them, and is itself written entirely in the Python language.

4.2.4 Reflexivity

Reflection is the ability of a language to rewrite sections of its own code. Python supports reflection through the *metaclass* type. A *class*, defined by the user, is actually an instance of a metaclass, making it mutable. Different instances of a given class can thus have different members. This has many implications, but in the context of this work, the major difficulty is that type information must be considered on an object-by-object basis — in Python, knowing the type of an object does not imply anything about the data contained by the object. This is in sharp contrast to C, where a variable is either a basic type (`char`, `int`, `float` etc.) or a `struct`. Types in C are not mutable — in C, the type and structure of the data contained by a variable is always known.

4.2.5 Type Checking

C is a statically typed language, meaning type constraints are checked at compile-time. Python is dynamically typed, meaning that constraints are checked at run-time. This is because Python variables are untyped in code and so the type cannot be determined until code is actually executed. This creates a significant problem by comparison with C — in C, a given statement executed at a given point in the code will always return a result of the same type. This is not the case in Python. Another side-effect of Python's dynamic type checking is the difficulty in dealing with array types. In C, the type of an array determines the type of the elements. In Python, the *list* type can contain individual elements of any type.

4.3 IDE

While tracepoints can be used outside an IDE by a command-line debugger, the intention is to provide a tracepoint interface within a modern IDE. This streamlines the process of adding and removing tracepoints, and gives many more options for presenting the gathered data to the user. A number of IDEs are currently available, but to be useful for the purposes of this work, the IDE must satisfy a number of requirements, based on the solution requirements as laid out in Section 3.2 and the design in Section 4.1:

Popular To promote the use of the tracepoint system, it must be added to a popular IDE. The popularity of an IDE is also an indicator of its *Usability*.

Open-source The IDE needs to be modified to add tracepoint support, so open-source IDEs are preferred. This also has implications for the *Extensibility* of the system.

Modular Modularity is also a factor for reasons of *Extensibility*.

Cross-platform This is necessitated by the requirement of *Compatibility*.

Support for C/C++ development C is one of the languages chosen for the initial implementation (see Section 4.2), so it must be supported by the selected IDE.

Support for Python development Python is the other language to be initially supported.

A selection of eight IDEs are evaluated by the above criteria to determine their suitability for use in this project. The results of this analysis are presented in Table 4.1. Eclipse and NetBeans are the only two IDEs to satisfy all requirements — KDevelop is excluded from consideration due to its comparatively small userbase. Additionally, KDevelop is written in C++, meaning that cross-platform support requires cross-compilation. Eclipse and NetBeans do not have this problem, as both are implemented in Java. Although the final implementation uses NetBeans, initially Eclipse was selected as the target IDE for two reasons: It had been previously modified in the “Robotic IDE” project, and its Python support is more mature than that of NetBeans (where Python support is still in beta). The reasons for the abandonment of Eclipse and adoption of NetBeans as the target IDE are given in the following section.

4.3.1 Eclipse

Eclipse is an open-source IDE written in Java, originally developed by IBM and now maintained by the Eclipse Foundation. Eclipse supports many different programming languages and, being written almost entirely in Java, can run interchangeably on Windows, Linux and OSX. As of this writing, the current version is Eclipse Galileo, 3.5.0.

Eclipse plugins are based around *extensions* and *extension points*. A plugin can allow its behaviour to be modified by providing *extension points*, and can itself modify the behaviour of other plugins with *extensions*. A plugin comprises an XML manifest which defines the extensions and extension points provided by the plugin as well as parameters for those extensions and fully-qualified references to Java classes within the plugin that implement the actual functionality. The manifest is necessary because Eclipse uses a lazy-loading paradigm for plugins in order to speed startup of the system. Classes within a plugin are not loaded until their functionality is requested through the plugin manifest. While there is provision for a plugin installer that initializes functionality, use of this extension point is strongly discouraged.

One side-effect of this approach is that in order for a class to be loaded as late as possible, much of the functionality of the class is defined within the XML file. For example, to add a menu item, the location and name of the menu item must be added to this file, as

	Websites (1000s) ^a	Free	Open-source	Extensible	Cross-platform	C/C++ support	Python support
Visual Studio	49000	$(\frac{\checkmark}{2})^b$		$(\frac{\checkmark}{2})^c$		✓	$(\frac{\checkmark}{2})^d$
Eclipse	5920	✓	✓	✓	✓	✓	✓
NetBeans	2610	✓	✓	✓	✓	✓	$(\frac{\checkmark}{2})^e$
XCode	367	✓				✓	$(\frac{\checkmark}{2})^f$
KDevelop	210	✓	✓	✓	✓	✓	✓
Code::Blocks	162	✓	✓	✓	✓	✓	
Geany	62.4	✓	✓	✓	$(\frac{\checkmark}{2})^g$	✓	
GNAT Studio	12.6	✓	✓	✓	✓	✓	

Table 4.1: Comparison of major IDEs.

^a Websites figures obtained from Google. Only pages updated in 2009 are included. Figures are intended only as a rough indication of popularity.

^b Although the Express Editions are free, these are missing some of the functionality of the full commercial software.

^c Although some support for extension is given, there is no scope for these extensions to become part of the core Visual Studio system.

^d Full IronPython support is provided for VS 2008 by a third-party plugin, however this plugin is no longer maintained.

^e NetBeans Python support is complete but deficient in many areas and not currently being maintained.

^f Xcode supports python source editing and execution, but not debugging.

^g Although the IDE will run in windows, building and debugging functionality is no longer available.

```
<visibleWhen checkEnabled="false">
  <and>
    <with variable="org.eclipse.core.runtime.Platform">
      <test property="org.eclipse.core.runtime.bundleState"
        args="org.eclipse.debug.ui"
        value="ACTIVE"/>
    </with>
    <with variable="activeContexts">
      <iterate operator="or">
        <equals value="org.eclipse.debug.ui.breakpointActionSet"/>
      </iterate>
    </with>
    <systemTest
      property="org.eclipse.debug.ui.breakpoints.toggleFactoriesUsed"
      value="true">
    </systemTest>
  </and>
</visibleWhen>
```

Figure 4.1: Excerpt of Eclipse plugin manifest defining breakpoint menu group visibility

well as some complicated markup to determine when the option is enabled. The structure and definition of this markup is in constant flux between Eclipse versions, with the result that there are often many different methods of achieving a particular effect (e.g. menu item visibility) — each with their own proponents with the Eclipse community. Some are deprecated, some are documented but unimplemented, and others are buggy. There is no guarantee between versions that an approach will continue to work. This presented a problem for the creation of a tracepoint UI.

Eclipse contains a basic API for adding support for different languages. A particular branch of this API concerns breakpoints. Three different breakpoint types are supported through menu items in the “Run” menu — a *Line Breakpoint*, a *Method Breakpoint*, and a *Watchpoint*. A fourth generic “Toggle Breakpoint” option is also given. However, these breakpoint types are rigidly defined, and no additional types may be created. Additionally, this API is always enabled — in the “Debug” view, the “Run” menu will always contain those four options, regardless of whether or not they are enabled for the current debugger. This has resulted in many plugins implementing their own breakpoint structure and ignoring the built-in methods. Thus there is no dependably common appearance or workflow for adding breakpoint-like markers (such as tracepoints) to code and working with them while code is executing.

Furthermore, when a particular debugging plugin is activated, all of its breakpoint types are accessible regardless of the current editor context. Previous versions of Eclipse permitted extensions to be activated based on the current selection, however this feature was disabled when it was discovered that this was slowing down the user interface. The current version does not permit plugins to be activated based on the current selection. This means that some options which are only valid at certain cursor positions can be selected inappropriately. An example of this is attempting to place a breakpoint on an

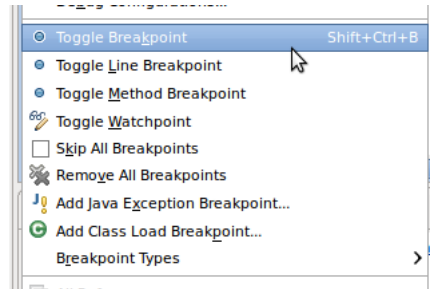


Figure 4.2: Screenshot of Eclipse Run menu showing conflicting breakpoint types

empty line or outside a function. This appears to be possible, but when attempted by the user results in an error message stating that the operation cannot be performed.

The final problem that was exposed with development for Eclipse was a bug with the editor (#19825). Many of the options in the context menu of an editor, including the setting and unsetting of breakpoints, depend on the current caret position. When the user right-clicks on a section of code to open the context menu, the menu options pertain to the code at the current location of the input caret. The problem is that right-clicking the mouse does not alter the position of the caret. In order to perform an operation on a piece of code, the user must first left-click to position the caret, and then right-click to select the operation. This bug was first reported in 2002, and since then at least two patches have been submitted by users to fix it, including one from this project. Unfortunately due to opposition and disinterest from the maintainers of the affected branch of Eclipse, no solution has ever been implemented.

Although by this time a working tracepoint implementation had been completed for the PyDev Python plugin within Eclipse, a combination of these problems led to the abandonment of Eclipse development in favour of NetBeans:

- Required duplication of work between the XML manifest and the actual plugin code
- Functionality constraints imposed by forced lazy-loading
- Lack of clear guidelines in documentation and specifications
- Lack of flexibility and foresight in the underlying debugger API
- Duplication of basic debugging functions between language plugins
- Inability to specify visibility of actions by editor context
- Lack of engagement with the community and response to user concerns

4.3.2 NetBeans

NetBeans is an open-source Java-based IDE being developed under the auspices of Sun Microsystems, the original developer of the Java language and current maintainer of the OpenJDK. The NetBeans IDE is actually a sub-project of the larger NetBeans Platform project, an attempt to provide a general purpose framework (a “Rich Client”) on which a developer can build a powerful multi-windowed application. The current version of NetBeans as of this writing is 6.8M1.

NetBeans plugins are called *modules*. A module consists of a number of Java packages, along with optional service definition files and an XML manifest. Unlike Eclipse, where the manifest contains markup that determines the behaviour of a plugin, the NetBeans manifest is only used to define a virtual file system. This file system stores system resources (e.g. icons, window settings etc.). The extension framework is embodied by the `Lookup` class, based on the JDK 6 `ServiceLoader` class. One module defines a public interface. A second module can then provide an implementation of that interface, which it exposes to the `Lookup` class via a special file in the `META-INF/services` folder. The filename is the fully-qualified name of the interface, and contains the fully-qualified name of the implementing class. Multiple implementations can be specified, one per line. A third module can then use the `Lookup` class to find implementations, or *Service Providers*. Only a reference to the module containing the interface is required; there is no direct dependency on the module containing the service provider. In this way a module can alter the behaviour of existing modules, and permit its own behaviour to be modified. Every aspect of the NetBeans IDE has been created in this way; the IDE can be regarded as a module suite that upgrades the functionality of the underlying NetBeans Platform.

A good example of this system is the method NetBeans gives for adding new breakpoint types. The “New Breakpoint...” option on the “Debug” menu opens a window containing two drop-down lists and an editing area. The left list contains a number of categories; the right list gives the breakpoints available for that category. The area underneath is populated with the appropriate configuration options for the breakpoint type selected. Figure 4.3 gives a screenshot. New breakpoint types are added to this window by way of the `org.netbeans.spi.debugger.ui.BreakpointType` service. To add a new type, the developer must first create a class that implements that interface. The functions required for the interface return a category name, a type name, and a class that provides the editor panel. Then the developer simply refers to it in a special file in `META-INF/services`. This made creation of the new tracepoint type a much more straightforward and standardized process than the equivalent method in Eclipse.

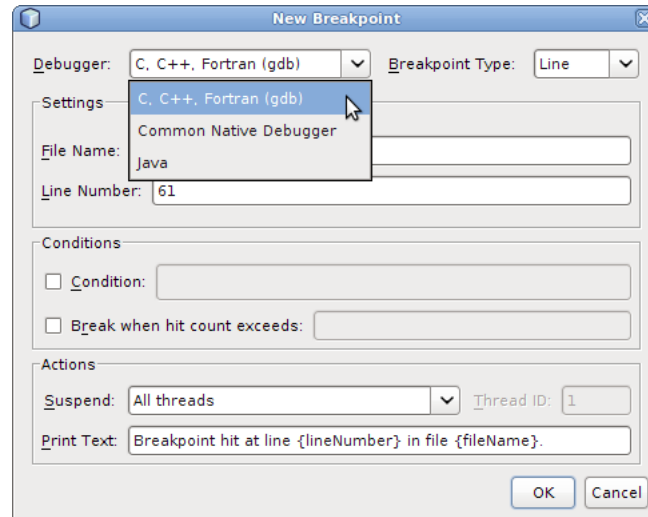


Figure 4.3: Screenshot of “New Breakpoint...” window in NetBeans

4.4 Summary

The selected concept is a novel implementation of the tracepoint debugging tool. A tracepoint is similar to a breakpoint, but does not halt the execution of the program being debugged, instead evaluating an arbitrary expression and reporting the result to the developer. Two languages are targeted for initial support, C and Python. These particular languages are chosen for reasons of compatibility with overall system requirements. Another factor is that they differ in many aspects, including syntax and formatting, execution style (compiled or interpreted), debugging, reflexivity and type checking. Implementing tracepoints for two such disparate languages partially satisfies the requirement of compatibility. A survey of current IDEs is presented to determine which should be targeted for extension. The two most suitable for extension are Eclipse and NetBeans. Although initial development of the IDE plugins targeted Eclipse, difficulties with a complicated extension system and poor engagement by the maintainers with the community led to the abandonment of Eclipse in favour of NetBeans. NetBeans is a free, open-source, cross-platform and extensible IDE implemented in Java.

5

Implementation

This chapter details the implementation of the system, explaining how each system element was created and interfaced. Figure 5.1 illustrates the overall layout of the tracepoint system. At the top of the diagram is the robot controller, responsible for controlling the actual robot hardware. It communicates through middleware to the actual code written by the robot developer. The code runs within a debugger, which is executed and controlled by the user through plugins in the IDE.

Both the debugger and the IDE plugins for the debugger are modified to support the tracepoint construct. A new plugin is created to hold the underlying tracepoint API, containing the functionality common to all client languages. Visualisation plugins consume the data gathered by tracepoints and render it to be displayed within the IDE.

Figure 5.1 shows that, using tracepoints, all the information shown to the developer is gathered directly from the state of the program being debugged. Gathering information directly from the debugger guarantees independence from the specific middleware in use.

5.1 Test Setup

Although the proposed tracepoint framework collects data independently of the robot middleware, for the purposes of rapid development only one middleware is used for testing. The middleware selected for this purpose is the open-source Player/Stage system (see Section 2.6.1). This system was chosen as it contains a built-in simulator as well as client

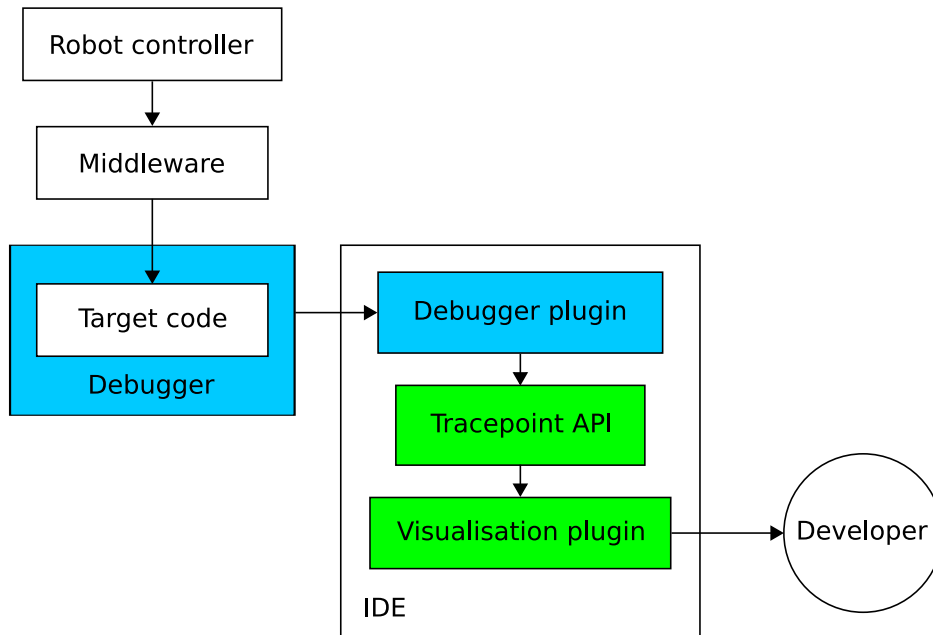


Figure 5.1: Block diagram of system implementation. Unmodified elements are in white, modified elements are in blue and novel elements are in green.

libraries for both of the targeted languages, C and Python. It can also be installed on Linux, Mac OS X and Windows. Our research group also has considerable experience with this middleware, as evidenced by a long history of working with and contributing to it.

5.1.1 Simulated Environment

The basic “Simple” world configuration is used in all testing for this work, defined in the `simple.cfg` and `simple.world` files included with Stage. A simulated Pioneer P2/DX robot navigates the environment, controlled by the “LaserObstacleAvoid” example. The original C++ `laserobstacleavoid.cpp` was ported to C and Python. A screenshot of the world as it appears in the Stage simulator on startup can be seen in figure 5.2. The screenshot shows Stage visualising information from the simulated laser and ultrasonic rangefinders.

5.1.2 Python/PlayerC

The Python client library for Player is actually a set of Python bindings for the C client library. These bindings are automatically generated by the open-source “Simple Wrapper and Interface Generator” system, or SWIG. It was discovered during early testing that

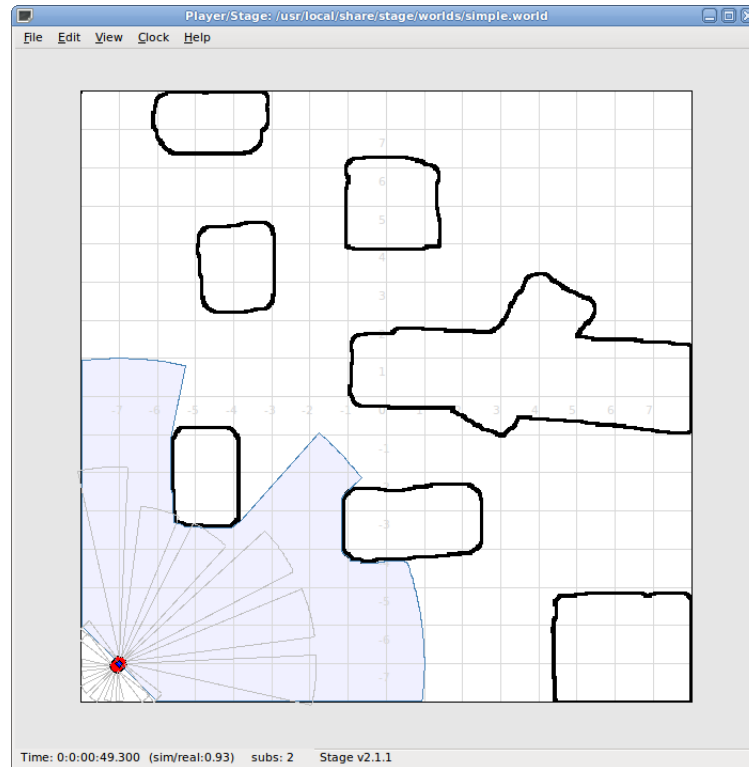


Figure 5.2: Screenshot of Stage simulator showing test environment.

these bindings had not been updated since a major change in the Player protocol (version 2.0). Previous versions of Player used fixed-size packets for transmitting data — for example, a laser packet always contained space for 401 readings, regardless of how many were used (see Fig. 5.3(a)). In the C client library, this was represented by a fixed-size array, `ranges`. From version 2.0 onward, a new variable-size packet was introduced (see Fig. 5.3(b)). The client library now contains a pointer to an array, instead of the array itself.

When SWIG translated this structure member to Python, it originally appeared as an instance of the native Python class `list`, easily subscriptable and accessible. After Player version 2.0, however, `ranges` was serialized as an instance of `SwigObject`, which contains only a hidden address pointing to the C array. The actual laser range information was no longer accessible.

In order to fix this so that the Player could be used to test tracepoints in Python, the script used to generate the SWIG bindings was rewritten. Pointers to core C types such as `int`, `float` and `double` are now represented by a special type generated on-the-fly (`intArray`, `floatArray` etc.). These types contain accessor functions patterned after those available for the native Python `list` class, thereby permitting access to the array members using the subscript operators (`[]`). Full representation as array types was not attempted, as the length of the array is unknown. The length is stored as a different

```
typedef struct
{
    playerc_device_t info;
    double pose[3];
    double size[2];
    int scan_count;
    double scan_start;
    double scan_res;
    int range_res;
    double ranges[PLAYERC_LASER_MAX_SAMPLES];
    double scan[PLAYERC_LASER_MAX_SAMPLES][2];
    double point[PLAYERC_LASER_MAX_SAMPLES][2];
    int intensity[PLAYERC_LASER_MAX_SAMPLES];
} playerc_laser_t;
```

(a) Player 1.6.5

```
typedef struct
{
    playerc_device_t info;
    double pose[3];
    double size[2];
    double robot_pose[3];
    int intensity_on;
    int scan_count;
    double scan_start;
    double scan_res;
    double range_res;
    double max_range;
    double scanning_frequency;
    double *ranges;
    double (*scan)[2];
    player_point_2d_t *point;
    int *intensity;
    int scan_id;
    int laser_id;
    double min_right;
    double min_left;
} playerc_laser_t;
```

(b) Player 2.1.3

Figure 5.3: The `playerc_laser_t` structure

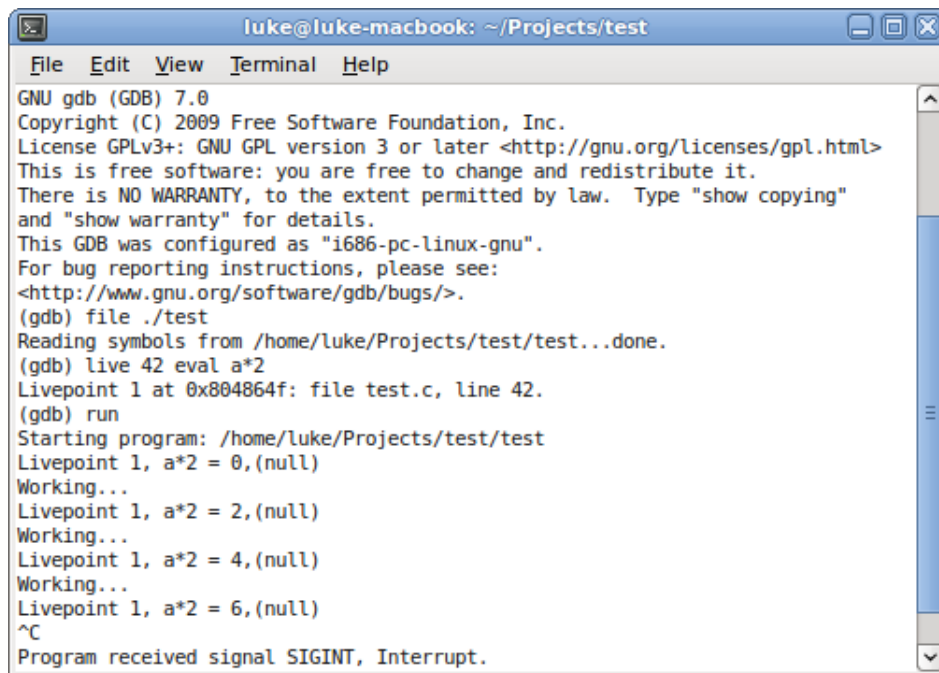
member of the data structure, and although this information could be passed to SWIG, it would have required marking up the entire C client library with this information wherever a variable-length array was used.

This modification was accepted by the Player maintainers and has been present in the main trunk since 23 February 2009 (versions 2.1.3 and upwards).

5.2 C and C++

5.2.1 GDB Overview

The GNU debugger, GDB, supports terminal-based debugging of both C and C++ programs. The path to the target executable (referred to as the *inferior*) is given on startup, and from this executable a symbol table is created. Breakpoints and watches may be added in a natural-language manner, by specifying line numbers and symbol names that are translated into memory locations in a way that is transparent to the user. Value outputs are similarly human-readable. The standard textual interface is referred to as the “command-line interface”, or CLI. Other interfaces are available, including a “machine interface”, or MI, that outputs information in a way more easily parsed by applications that wish to control the debugging process. GDB can only be interacted with via the standard input and output pipes of a process, and only recognises input while the inferior is not running. Screenshots of GDB working in a terminal are shown in Figures 5.4 and 5.5.

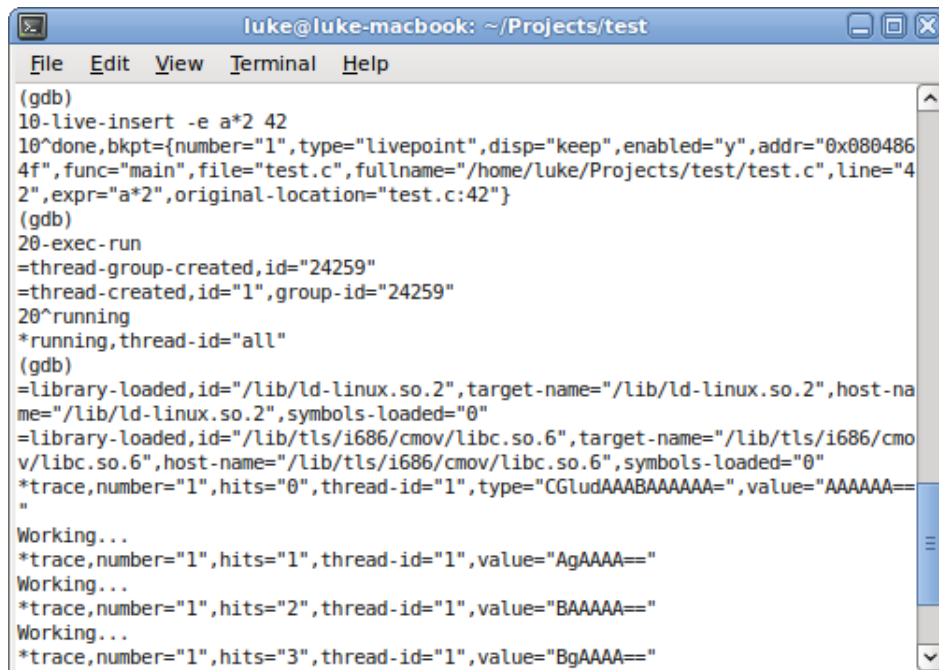


```

luke@luke-macbook: ~/Projects/test
File Edit View Terminal Help
GNU gdb (GDB) 7.0
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
(gdb) file ./test
Reading symbols from /home/luke/Projects/test/test...done.
(gdb) live 42 eval a*2
Livepoint 1 at 0x804864f: file test.c, line 42.
(gdb) run
Starting program: /home/luke/Projects/test/test
Livepoint 1, a*2 = 0,(null)
Working...
Livepoint 1, a*2 = 2,(null)
Working...
Livepoint 1, a*2 = 4,(null)
Working...
Livepoint 1, a*2 = 6,(null)
^C
Program received signal SIGINT, Interrupt.

```

Figure 5.4: Screenshot of GDB showing the command-line interface



```

luke@luke-macbook: ~/Projects/test
File Edit View Terminal Help
(gdb)
10-live-insert -e a*2 42
10^done,bkpt={number="1",type="livepoint",disp="keep",enabled="y",addr="0x0804864f",func="main",file="test.c",fullname="/home/luke/Projects/test/test.c",line="42",expr="a*2",original-location="test.c:42"}
(gdb)
20-exec-run
=thread-group-created,id="24259"
=thread-created,id="1",group-id="24259"
20^running
*running,thread-id="all"
(gdb)
=library-loaded,id="/lib/ld-linux.so.2",target-name="/lib/ld-linux.so.2",host-name="/lib/ld-linux.so.2",symbols-loaded="0"
=library-loaded,id="/lib/tls/i686/cmov/libc.so.6",target-name="/lib/tls/i686/cmov/libc.so.6",host-name="/lib/tls/i686/cmov/libc.so.6",symbols-loaded="0"
*trace,number="1",hits="0",thread-id="1",type="CGludAABAAAAA=",value="AAAAA="
Working...
*trace,number="1",hits="1",thread-id="1",value="AgAAAA="
Working...
*trace,number="1",hits="2",thread-id="1",value="BAAAAA="
Working...
*trace,number="1",hits="3",thread-id="1",value="BgAAAA="

```

Figure 5.5: Screenshot of GDB showing the machine interface

When a breakpoint is placed in GDB, the instruction at the specified code location is cached and replaced by an instruction (on X86 processors, `INT 3` or `0xCC`) that causes a CPU interrupt. This instruction is used as the interrupt instruction must be shorter than the instruction being replaced (`INT 3` is only one byte in length, the smallest possible X86 instruction). In Linux, this causes the inferior to halt with a `SIGTRAP` signal. While the inferior is running, GDB uses the `sigsuspend` function to wait for it to emit a signal. Once a signal is received, GDB calls the `waitpid` function to find out what it was. If the signal was `SIGTRAP`, the current program counter is retrieved and compared against all the stored breakpoints to determine which one was hit. Further action is then taken depending on the breakpoint type.

Breakpoints are represented internally by `struct breakpoint`, defined in the GDB sources in `breakpoint.h`. GDB already contains support for twenty different breakpoint types (although many of these are only for internal use). All of these types use the same structure to store information about the breakpoint, with the type being differentiated by the `type` member, an instance of `enum bptype`. A new `bptype` was added to support tracepoints. When a tracepoint is set by the user, GDB creates a regular breakpoint with the tracepoint type flag. The `cond` member of `struct breakpoint`, normally used to store the breakpoint condition, is instead used to hold the tracepoint expression. The expression is parsed and converted to opcodes when the tracepoint is set, just as a condition would be for a regular breakpoint.

The tracepoint is set and triggered like a normal breakpoint, by replacing the instruction at the specified location with `INT 3`. The difference occurs when the matched breakpoint is processed by the `bpstat_what` function in `breakpoint.c`. This function contains a switch statement that takes action depending on the breakpoint type. If a tracepoint is hit, the `tracepoint_evaluate` function is called, and the debugger is instructed to continue as if it was a conditional breakpoint where the condition was not met. The `tracepoint_evaluate` function takes care of evaluating the tracepoint expression as well as serializing and outputting the result. The expression is evaluated by an internal GDB function called `evaluate_expression`, which takes a pointer to a `struct expression` and returns a pointer to a `struct value`.

5.2.2 New GDB Commands

In order to access the new tracepoint functionality, new commands have been added to both the normal and machine interpreter modes of GDB. Since the Introspect extensions to GDB have already co-opted the *trace* and *tracepoint* keywords, the tracepoints implemented for this thesis are referred to as *livepoints* in GDB syntax. This is to reflect the fact that the tool provides a live report of expression results as they are evaluated, rather than the post-execution functionality of Introspect tracepoints. As GDB permits

user interaction via two different interpreters, two different sets of commands have been implemented to expose tracepoint functionality regardless of the preferred interpreter. These commands are described in the following sections. An example of a CLI interaction can be seen in Figure 5.6 and an MI interaction in Figure 5.7.

Tracepoint Insertion

CLI	live <loc> eval <expr>
MI	<seq>-live-insert -e <expr> <ptropts> <loc>

Table 5.1: CLI and MI tracepoint insertion commands for GDB.

Both the CLI and MI deletion commands take mandatory expression (<expr>) and location (<loc>) fields. The expression field is specified in valid C/C++ syntax and must contain no whitespace. The location field may be specified in a number of different ways. Valid options include a direct address within the inferior code (decimal or hexadecimal formats accepted), a function name, or a token of the form <file>:<line> specifying a filename and line number at which to place the tracepoint. Functions and line numbers are converted to addresses by GDB using the internal symbol table. The tracepoint expression is evaluated and reported *before* the targeted address is executed. Tracepoints are indexed in the order of definition, starting at number 1. The numbering system is shared with all other breakpoint types. Indexes are not reused after deletion.

The MI version of the command takes an optional <seq> field specifying a number to be echoed back when the command completes. The MI command also permits extra options specifying pointer serialization (signified in Table 5.1 by <ptropts>). These options apply when the expression evaluates to a pointer or struct (see Section 5.2.6 for more information). When the expression evaluates to a pointer, the *-s* family of options applies. When the expression evaluates to a structure, the *-sm* family of options applies. Pointers can be serialized as null-terminated strings, as an array with length specified by an arbitrary expression, and (in the case of a pointer which is the member of a structure) as an array with length specified by another member of the same structure. These parameters are specified in Table 5.2 below.

Serialization	Applies to	Parameter
Null-terminated	Pointer	-s :cstr
By expression	Pointer	-s =<expr>
Null-terminated	Structure	-sm <ptr>:cstr
By expression	Structure	-sm <ptr>=<expr>
By member	Structure	-sm <ptr>#<len>

Table 5.2: MI tracepoint insertion parameters for pointer serialization.

The `<expr>` field contains an arbitrary expression. The `<ptr>` field contains the name of a member of the structure which is a pointer. The `<len>` field contains the name of a member of the same structure as `<ptr>`. This member must be of an integer type. Where the tracepoint expression evaluates to a pointer, only one `-s` parameter may be used. Where the tracepoint expression evaluates to a structure, multiple `-sm` parameters may be specified in order to serialize more than one member of a structure. Both the `-s` and `-sm` parameters are optional — where they are not used, pointers are serialized as an integer type containing a memory address.

Tracepoint Deletion

CLI	delete livepoints <id> ...
MI	<seq>-live-delete <id> ...

Table 5.3: CLI and MI tracepoint deletion commands for GDB.

The deletion commands are capable of deleting multiple tracepoints at once. If the user does not specify any indices, all tracepoints are deleted. If one or more indices are specified, only the specified tracepoints are deleted. Indices are not reused after deletion. As with other MI commands, the `-live-delete` command takes an optional `<seq>` field specifying a number to be echoed back on completion.

Tracepoint Enablement

CLI	enable livepoints <id> ...
CLI	disable livepoints <id> ...
MI	<seq>-live-enable <id> ...
MI	<seq>-live-disable <id> ...

Table 5.4: CLI and MI tracepoint enablement commands for GDB.


```
GNU gdb (GDB) 7.0
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
(gdb) file ~/Projects/test/test
Reading symbols from /home/luke/Projects/test/test...done.
(gdb) live test.cc:11 eval i
Livepoint 1 at 0x804853e: file test.cc, line 11.
(gdb) info livepoints 1
Num      Type      Disp Enb Address      What
1        livepoint  keep y  0x0804853e  in main at test.cc:11
          evaluate i
(gdb) run
Starting program: /home/luke/Projects/test/test
Livepoint 1, i = 0
Root is 0.000000
Livepoint 1, i = 1
Root is 1.000000
Livepoint 1, i = 2
Root is 1.414214
Livepoint 1, i = 3
Root is 1.732051
```

Figure 5.6: Tracepoint demonstration with GDB CLI.

These commands handle enabling and disabling tracepoints after insertion. A disabled tracepoint will not be written into the inferior. No interrupt is propagated to the debugger when execution passes a disabled tracepoint. This means that hit counts are not updated for disabled tracepoints. Like the deletion commands, these commands are capable of taking any number of indices — if no indices are specified, the command is applied to all tracepoints. As with other MI commands, the `-live-enable` and `-live-disable` commands take an optional `<seq>` field specifying a number to be echoed back on completion.

Tracepoint Information

CLI	<code>info livepoints <id></code>
-----	---

Table 5.5: CLI tracepoint information command for GDB.

The `info` command displays information about the specified tracepoint, including location, enablement, expression and hit count. If the location can be translated to a symbolic location within the inferior, this information is displayed as well. This command is not supported for MI mode.

```

~"GNU gdb (GDB) 7.0\n"
~"Copyright (C) 2009 Free Software Foundation, Inc.\n"
~"License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>\n"
~"This is free software: you are free to change and redistribute it.\n"
~"There is NO WARRANTY, to the extent permitted by law.  Type \"show copying\" \n"
~"and \"show warranty\" for details.\n"
~"This GDB was configured as \"i686-pc-linux-gnu\".\n"
~"For bug reporting instructions, please see:\n"
~"<http://www.gnu.org/software/gdb/bugs/>...\n"
~"Reading symbols from /home/luke/gdb-test/gdb/test..."
~"done.\n"
(gdb)
10-live-insert -e i test.cc:11
10^done,bkpt={number="1",type="livepoint",disp="keep",enabled="y",addr="0x0804853e",
func="main",file="test.cc",fullname="/home/luke/Projects/test/test.cc",
line="11",expr="i",original-location="test.cc:11"}
(gdb)
20-exec-run
=thread-group-created,id="4595"
=thread-created,id="1",group-id="4595"
20^running
*running,thread-id="all"
(gdb)
=library-loaded,id="/lib/ld-linux.so.2",target-name="/lib/ld-linux.so.2",
host-name="/lib/ld-linux.so.2",symbols-loaded="0"
=library-loaded,id="/usr/lib/libstdc++.so.6",target-name="/usr/lib/libstdc++.so.6",
host-name="/usr/lib/libstdc++.so.6",symbols-loaded="0"
=library-loaded,id="/lib/tls/i686/cmov/libm.so.6",target-name="/lib/tls/i686/cmov/libm.so.6",
host-name="/lib/tls/i686/cmov/libm.so.6",symbols-loaded="0"
=library-loaded,id="/lib/libgcc_s.so.1",target-name="/lib/libgcc_s.so.1",
host-name="/lib/libgcc_s.so.1",symbols-loaded="0"
=library-loaded,id="/lib/tls/i686/cmov/libc.so.6",target-name="/lib/tls/i686/cmov/libc.so.6",
host-name="/lib/tls/i686/cmov/libc.so.6",symbols-loaded="0"
*trace,number="1",hits="0",thread-id="1",type="CGLudAAABAAAAA=",value="AAAAAA=="
Root is 0.000000
*trace,number="1",hits="1",thread-id="1",value="AQAAAA=="
Root is 1.000000
*trace,number="1",hits="2",thread-id="1",value="AgAAAA=="
Root is 1.414214
*trace,number="1",hits="3",thread-id="1",value="AwAAAA=="
Root is 1.732051
*trace,number="1",hits="4",thread-id="1",value="BAAAAA=="
Root is 2.000000

```

Figure 5.7: Tracepoint demonstration with GDB MI.

5.2.3 Serialization

Outputting the result of the tracepoint expression proved problematic. As GDB was originally intended for direct use by the developer, the command and response syntax is textual, human-readable and difficult to parse. Although the implementation of the machine interface has improved the situation, variable output is still in human-readable text form, modelled on standard C syntax. This is less than ideal for a number of reasons. One simple example would be the serialization of an integer value. GDB can read the binary value of the integer directly from the memory of the inferior, but to be made human-readable it must be converted from its packed form to a decimal string. Assuming another program is controlling GDB, this string must then be read and re-converted back to a memory representation in order for any operations to be performed on it. Not only is decimal representation far less efficient than binary in terms of size; the conversion operation itself adds unnecessary overhead.

A more efficient method of serialization was required. As the type of an expression result is guaranteed to be the same every time a tracepoint is hit, the decision was made to separate the output of type information from the output of the value itself. Type information is only output the first time the tracepoint is struck. Thereafter only a memory dump of the value is output, and the client program uses a cached copy of the type information to decode it. This prevents unnecessary serialization and deserialization and keeps tracepoint overhead as low as possible.

5.2.4 Encoding

As GDB uses a comma-delimited, CRLF-terminated command and response structure (even in MI mode), outputting raw binary information is not permitted. Base64 encoding was therefore used to allow tracepoint outputs within the name/value pair style used by GDB responses. A new source file and accompanying header (`b64.c` and `b64.h`) were added to GDB to permit this. This Base64 library implements a stream-based Base64 encoder supported by a `b64_buf` structure. This was necessary as discrete Base64 code groups require three input bytes apiece. With the buffer, output can be streamed a byte at a time (necessary for the type serialization code, which is recursive).

5.2.5 Type Serialization

GDB represents values with two main structures, `struct value` and `struct main_type`. The value structure holds a pointer to a type structure as well as a number of fields specifying the address and size of the value in the memory of the inferior. A straight byte address is insufficient as this fails to take into account the possibility that the value is part of a bitfield. The type structure holds all information about the

type. The general type is indicated by the enum `type_code` `code` member, with more detailed information also available, such as endianness, size, name, pointer information and so on. The serialized type information output by the system consists of a header, followed by further information as required for a particular type. Only three types require additional information at present; integral values, arrays, and structures.

<i>TypePacket</i>			
<i>Field</i>	<i>Type</i>	<i>Bytes</i>	<i>Description</i>
Code	int	1	<code>type_code</code> value from GDB
TypeName	str	?	Null-terminated type name
MemberName	str	?	Null-terminated member name (if any)
Size	int	4	Size of type — equivalent to <code>sizeof()</code>

Table 5.6: Type packet header.

<i>IntegralTypePacket</i>			
<i>Field</i>	<i>Type</i>	<i>Bytes</i>	<i>Description</i>
Type	TypePacket	?	Basic type packet
Flags	int	1	Integer flags — signed/unsigned, endianness etc.

Table 5.7: Type packet for integral types.

<i>ArrayTypePacket</i>			
<i>Field</i>	<i>Type</i>	<i>Bytes</i>	<i>Description</i>
Type	TypePacket	?	Basic type packet
ElementType	TypePacket	?	Nested type packet specifying element type

Table 5.8: Type packet for array types.

<i>StructTypePacket</i>			
<i>Field</i>	<i>Type</i>	<i>Bytes</i>	<i>Description</i>
NumMembers	int	2	Number of structure members
Members[]	StructMemberPacket[]	?	Array of member information packets

Table 5.9: Type packet for structured types.

<i>StructMemberPacket</i>			
<i>Field</i>	<i>Type</i>	<i>Bytes</i>	<i>Description</i>
BitPosition	int	4	Position of member within structure in bits
BitSize	int	4	Size of member within structure in bits
MemberType	TypePacket	?	Nested type packet specifying member type

Table 5.10: Member subpacket for struct types.

5.2.6 Pointers

Another difficult problem was how to appropriately serialize a pointer. The pointer type does not imply anything about the actual contents of the target memory. It could be a pointer to a single instance of the target type, to an arbitrary array of the target type, or in the case of a `char*`, to a null-terminated array of the target type. Without input from the user, no assumption can be made. Pointers are therefore serialized by default as an unsigned integer. However, the user may wish to examine the actual contents of the memory referenced by a pointer. In order to allow for this situation, the GDB MI was further expanded to allow for a tracepoint result to include additional memory dumps where the tracepoint result contains a pointer (either a pointer itself, or a structure with a member that is a pointer).

The target location of the memory dump is set by the pointer value; the extent can be specified via one of three methods:

1. Null-terminated
2. Length specified by arbitrary expression
3. Length specified by structure member

The third option is only available when the specified pointer is the member of a structure, as it uses the value of another member of the same structure as a length value. As this does not involve evaluating an expression, it is expected to be faster (depending on expression complexity).

When a pointer has a user-specified serialization, extra fields are output when the tracepoint is struck in addition to the basic “type” and “value” pairs. Similarly to a tracepoint, the pointer serialization type is output the first time the tracepoint is struck. After that, only the serialized value itself is output.

5.2.7 NetBeans Plugin

The NetBeans plugin for working with GDB is contained within the `cnd.debugger.gdb` module. This module handles starting a debug session, configuring GDB for machine-interpreter mode, and interacting with the debugger during a debugging session. While the overall structure of this module remains unchanged, a number of new classes have been added in order to enable tracepoint functionality. The basic requirements laid out in section 5.8.1 were followed. The `GdbTracepoint` class is a subclass of the basic `Tracepoint` class provided by the Tracepoint API. Additional functionality was added to this class to support the pointer serialization options discussed in section 5.2.6 above. The `Base64` class handles the translation of the MIME Base64-encoded data used in tracepoint serialization back to raw bytes.

Deserialization

As GDB returns both encoded type information and raw memory dumps, a two-step deserialization process is required. In the first step, an instance of the `GdbTraceType` class is created by a call to `GdbTraceType.Parse()`. This function takes the raw type data from GDB (after Base64 decoding) and uses it to populate one of the subclasses of `GdbTraceType`. These classes are a Java equivalent to the GDB internal `main_type` structure (see section 5.2.5). They contain the logic for parsing the raw memory dumps returned by GDB when a tracepoint is struck. There is a subclass for each GDB type that requires extended data (see tables 5.7, 5.8, 5.9). Container types, like structures and arrays, are represented by a `GdbTraceType` that itself contains subsidiary `GdbTraceTypes`. Thus nested classes and arrays can be easily decoded with recursive calling.

In the second deserialization step, the raw data is passed to the created `GdbTraceType` instance with a call to `GdbTraceType.Decode()`. This call returns a `Trace` instance (see section 5.4.1), which can be attached to the tracepoint with a call to `Tracepoint.setData()`.

5.3 Python

5.3.1 Debugging Overview

Tracepoints are simpler to implement in Python — it is an interpreted language, so the debugger becomes a part of the program being debugged. This, along with Python's introspection abilities, meant that values being traced are effectively serialized by the inferior itself. This allows for much greater flexibility than with C and GDB.

Debugging in Python uses an interpreter hook (a callback function). The interpreter calls the provided function with the current stack as a parameter whenever one of seven possible events (specified in the `settrace` call) occurs. The main two events are “call” and “line”. If “call” is specified, the callback function is called whenever a function is about to execute. If “line” is specified, the callback function is called before every line of code in the inferior executes. The return value of the callback can be used to specify a new “local” trace function which is called before every line in the current function executes.

Usually (although not always) a Python debugger will call `sys.settrace` with the “call” event specified. When the user places a breakpoint, the function in which it was placed is recorded and this value is checked against the stack every time the callback function executes. When execution is within the function that has the breakpoint, a local callback is specified which then checks the line number to see if the correct line has been reached. When this condition is met, the breakpoint is hit. This is the method employed

by the Python debugger included with NetBeans. Unlike C and GDB, where execution is unaffected until the breakpoint is reached, this method incurs an overhead for every function called, and a further overhead for every line within the target function. While this is less than ideal, no other method exists for debugging Python programs.

5.3.2 Jpydaemon

Jpydaemon.py is the name of the Python debugging program provided with NetBeans. In order to start a debugging session, jpydaemon is executed with the name of the inferior script as a parameter. A TCP/IP connection is then established with the IDE, which sends textual commands to jpydaemon in order to control the debugging session. Jpydaemon's responses are XML-formatted text. Unfortunately, jpydaemon has not been written to execute alongside the target program but instead only processes commands when the inferior is halted.

Tracepoints were implemented in jpydaemon by implementing a new array within the debugger of tracepoint locations (in addition to the existing array of breakpoints). Each time a function or line is checked for the presence of a breakpoint, it is also checked for the presence of a tracepoint. If a tracepoint is present, the attached expression is evaluated using the built-in `eval` function. The `eval` function takes the current stack frame and a string containing python code to be executed. The interpreter executes the code and the function returns the generated value. The result is then serialized and transmitted back to the IDE.

5.3.3 Serialization

In Python the type information of the expression must be output every time the tracepoint is hit, as type information in Python is mutable. First, a list of every member of the object is obtained. For some hard-coded variable types (basic types such as integers, booleans, strings etc.), only the value is serialized. For classes, the serialisation method is recursive, but only down one layer to prevent circular references generating infinite-size packets. Member functions are ignored. Additional difficulties become evident when array types are considered. In C, the element type is defined by the array type. In Python, no such restriction exists. A list can contain objects of any number of different types. Type information must therefore be transmitted for every individual element in an array. Because of this, tracepoint overhead in Python is inevitably higher than in C even before the performance drawbacks of Python as an interpreted language are considered.

Due to the XML-based protocol used by jpydaemon, values are serialized to an XML tree. There are several drawbacks to this approach, primarily issues of speed and efficiency. XML is an extremely inefficient method of communication back to the target IDE. A faster

approach would be to write a library in C to perform serialisation based on the underlying C classes used by the interpreter, however this would have taken far too long in the context of this project.

A further issue for this project was that the Player client libraries for Python were generated from the C++ libraries using SWIG. As Player makes use of pointers to dynamically allocated arrays, (as explained in a previous chapter), a further difficulty arose. SWIG represents pointers as a class with a pointer as the value member. While modifications were made to the SWIG code in Player to permit the use of standard array syntax in Python, as it is not a native list type the serializer cannot cope with it directly. Thus custom code for serialising these objects had to be included in jpydaemon. If player had native Python client libraries (as opposed to a Python wrapper to the C++ client library) this would not have been a problem.

5.3.4 NetBeans Plugin

The NetBeans plugin for Python debugging is called `python.debugger`. Similarly to the plugin for GDB, this handles the configuration and execution of the python debugger, as well as ongoing session management by the user. One significant difference with the GDB plugin is that the python debugger itself is contained within this plugin. All the modifications required to use tracepoints with python are therefore contained within this plugin. As NetBean's Python facilities are not yet mature (the first alpha version was released in April 2008), the official release still has several bugs, such as the inability to forcibly terminate a debugging session. This has been resolved in the version released with this system. New classes have been added to permit the use of tracepoints, as specified in Section 5.8.1.

5.4 Tracepoint API

The tracepoint API implemented within NetBeans is contained within a single entirely novel module called “`api.tracepoints`”. Many of the 28 classes contained within this module are for internal use only; this section concentrates on the classes necessary for implementing tracepoints for a new language. These classes handle editing, storage, management and persistence of tracepoints, the representation and proliferation of the data they gather, and the display of this data to the end-user.

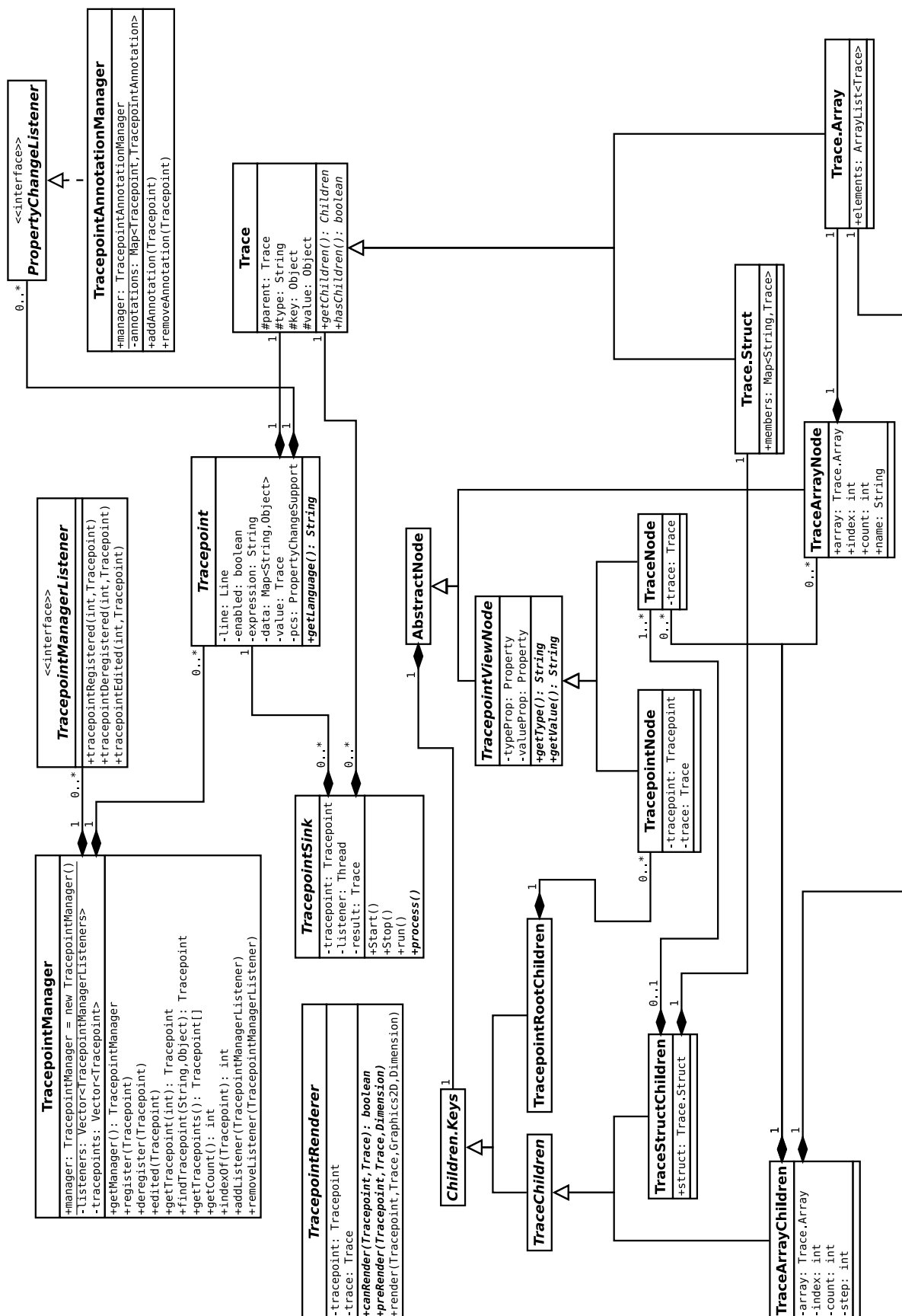


Figure 5.8: UML diagram of Tracepoint API class structure

5.4.1 Class Structure

Tracepoint

The *Tracepoint* class is an abstract base class intended to be subclassed for each target language or debugger. The only requirement of the subclass is that it provides an implementation of the `getLanguage()` function. The return value of this function is used as a filter when searching through the global tracepoint list to ensure that different implementations are dealing only with the correct tracepoint type. The base class provides a full implementation of all other necessary functionality. It contains the target file and line number as well as the expression to be evaluated and a flag signifying whether or not the tracepoint is enabled. It holds the latest value gathered from the inferior and maintains a list of ‘listeners’ that are notified whenever a tracepoint property changes (i.e. target line number). The name of the tracepoint is generated within this class, defined as “expression@filename:linenumber”. Each *Tracepoint* also contains a `_data` hashmap for storing general-purpose data at runtime. This functionality is used when setting tracepoints in GDB, to store the sequence number of the tracepoint creation command.

TracepointManager

Once a *Tracepoint* has been created, it must be registered with the *TracepointManager*, a singleton class instantiated at startup. Access to the singleton is by the static `getManager()` function call. The *TracepointManager* class is thread-safe and contains the global list of *Tracepoints*. The *Tracepoint* list can be searched by language (from `getLanguage()`) or by key/value pair within the `_data` hashmap discussed in the previous section. The *TracepointManager* maintains a list of listeners implementing the *TracepointManagerListener* interface. These listeners are notified when *Tracepoints* are registered, deregistered or edited. Finally, the *TracepointManager* acts as the root node of the *TracepointNode* tree (from the `org.openide.nodes` package within NetBeans). This tree is the primary way the list of *Tracepoints* is displayed to the user within NetBeans.

When the `register` function is called with a new *Tracepoint*, it is added to the global *Tracepoint* list and this list is persisted to the debugger properties cache. The singleton *TracepointAnnotationManager* class is also called to set the code annotation (a small yellow glyph and highlight displayed inline). The *TracepointAnnotationManager* adds itself to the *Tracepoint*’s listeners and alters the annotation to reflect any changes in the *Tracepoint*’s properties (enablement, etc.). Calling `deregister` removes a *Tracepoint* from the list and cache, as well as removing the code annotation.

Trace

The *Trace* class is the general-purpose data abstraction used to represent data gathered when a tracepoint is hit. This class was critical to the overall system and went through several design iterations to improve speed and flexibility and decrease the memory footprint. All Traces have four base variables — a reference to the parent Trace, a string containing the type name, a key Object (used when the Trace is a structure or class member or an array element), and finally an Object containing the Java representation of the Trace's actual value. Of the four, only the type name is guaranteed to be set. The Trace class has two internal subclasses, *TraceStruct* (used to represent container types where members are keyed by a string) and *TraceArray* (used to represent container types where elements are keyed by an index). Trace.Struct and Trace.Array set their value member to null, as the value of these Traces is in the contained elements (rather than the trace itself).

Trace values are generated by the plugin for the target language when a tracepoint is hit, and associated with a Tracepoint with a call to the Tracepoint's `setTrace` function. From here the value can be retrieved with a call to the `getCurrentTrace` function, which returns the current Trace (if any). The `getTrace` function is slightly different. This function waits on the Tracepoint before returning the trace. As the Tracepoint is signalled every time a new Trace is set, this guarantees a new Trace is returned. The Tracepoint maintains a sequence number which is incremented every time a new Trace is set. In this way, Trace consumers can check for the existence of a new Trace without necessarily having to wait.

TracepointSink

To automate the process of consuming Trace objects, the abstract TracepointSink class was created. Subclasses must implement the `process` function, called whenever a new Trace is available. The TracepointSink class is created with a reference to a Tracepoint and on instantiation creates a new thread. This thread first checks the internal Trace sequence number of the Tracepoint. If the number is new, a Trace is immediately passed to the subclass `process` function. If the number is not new, the TracepointSink uses the `getTrace` function to wait for a new one. The `process` function is called inline, so if it takes a while to execute there is a possibility that some Traces may be missed. It is not possible for a TracepointSink subclass to hold up the inferior by processing Traces too slowly.

Tracepoint	Type	Value
*sp@main.c:49	playerc_sonar_t	
▶ info	playerc_device_t	
▶ pose_count	int	16
▶ poses	player_pose3d_t[]	
▶ scan_count	int	16
▶ scan	double[]	
*lp@main.c:51	playerc_laser_t	

Figure 5.9: The Tracepoints view.

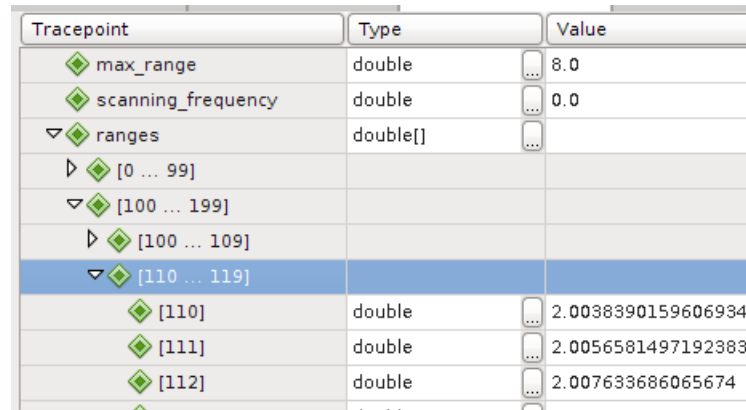
5.5 Tracepoint View

A “Tracepoint View” has been added to the IDE which displays an entry for every tracepoint that the user has added (see Fig. 5.9). This view has three columns, one for the tracepoint name and structure, one for the data type, and one for the current value. Tracepoints are named for their expression, file name and line number (“expression@filename:line”). Before a tracepoint has been struck, both the type and value columns are blank. This view is updated as tracepoints are added and removed, as well as when they are struck during a debugging session. This view uses the OpenIDE Node Tree API provided by NetBeans to display and update tracepoint information. When a tracepoint hit occurs, the type and value columns are filled in. If the expression evaluated to a container type, such as an array or struct, the tracepoint shown in the view becomes expandable. The user can click to the left of the tracepoint name to expand it, and a list of the members of the container is provided. These members in turn can be expanded if they themselves are containers.

Whenever a Tracepoint is struck, the entire node tree is automatically updated live and in place. If the user has expanded certain nodes, as long as those nodes or elements still exist, they will still be expanded. The position and selection of the user within the tracepoint view is unchanged.

OpenIDE Node Tree

The OpenIDE Node Tree is a system provided by NetBeans whereby data can be represented in a way that can be rendered and interacted with by a number of UI components interchangeably. The *AbstractNode* class (not actually abstract) can either be subclassed or used directly for this purpose. Each *AbstractNode* has a display name, an icon, and a number of associated actions (which appear in a context menu when the Node is right-clicked). A *Children* object is also required, which is used to produce child nodes. Child nodes are not actually produced until the user expands the parent.



Tracepoint	Type	Value
max_range	double	8.0
scanning_frequency	double	0.0
ranges	double[]	
[0 ... 99]		
[100 ... 199]		
[100 ... 109]		
[110 ... 119]		
[110]	double	2.0038390159606934
[111]	double	2.0056581497192383
[112]	double	2.007633686065674

Figure 5.10: Tracepoints view showing TraceArrayNode placeholders.

The Tracepoint View is a *TopComponent*, a window that appears in the IDE (similar to the Watches or Call Stack views) which contains an *OutlineView*. The *OutlineView* is a combined Tree and Table view component, capable of displaying tree data with extra columns showing additional information for each node in the tree. The *OutlineView* uses the tracepoint node tree as the basis for the displayed data.

The tracepoint node tree is made up of several different object types. At the root of the tree is the *TracepointManager* (not visible to the user). The second-level Tracepoint nodes are represented by the *TracepointNode* class. Trace values attached to the Tracepoint are represented by *TraceNode* instances. A special case is when the Trace value is a *Trace.Array*. As arrays could potentially have thousands of elements, a placeholder node represented by *TraceArrayNode* is used. This node displays a range of indices. The maximum number of elements an array can expand at a time is ten. For example, if the Trace value attached to a Tracepoint was an array of 100 elements, when the user expanded the TracepointNode there would be ten *TraceArrayNodes*, each showing [0 ... 9], [10 ... 19], and so on. Each of those nodes in turn would expand to ten actual array elements. A screenshot of this behaviour can be seen in Figure 5.10.

The *Children.Keys* class was subclassed to provide children factories for all the new node types. This class stores node children against a list of key objects. When updates occur, new nodes are created and old nodes are deleted based on the new list of keys produced by the *Children.Keys* instance. Subclasses of *Children.Keys* include *TraceArrayChildren*, *TraceStructChildren*, and the *TracepointManager* singleton. *TracepointNode* children are provided by the appropriate *Trace<x>Children* class, acting on the contained *Trace*. The appropriate *Children* class is obtained by a call to *Trace.getChildren()*. This function returns *Children.LEAF* (i.e. no children) for a regular *Trace*, and is overridden by *Trace.Struct* and *Trace.Array* to return an instance of *TraceArrayChildren* or *TraceStructChildren*, as appropriate.

5.6 Visualisations

The tracepoint functionality as described so far permits values to be observed in real-time. The difficulty lies in interpreting the values that are being reported; for example, an array of 360 double-precision floating point values, updating ten times per second, does not easily translate to a laser range map in the mind of a developer. The visualisation functionality of the tracepoint plugin is intended to assist the developer by graphically interpreting the data being reported. In this way, trends or outliers in data can be much more easily identified.

5.6.1 Overview

The basic visualiser is a new window for the IDE called the Tracepoint Render View. This view is provided by the `api.tracepoints` plugin, so is always present. This view presents a drop-down list of tracepoints that is updated as the user adds and removes them. Below this list is a special *JPanel* used as the rendering area. When the user selects a particular tracepoint, a `TracepointSink` is instantiated to consume incoming Trace data. When a Trace is received, the `TracepointSink` checks to see if the view currently has a valid *TracepointRenderer* for the data. If there is no valid renderer, a search of available *TracepointRenderer* service providers is conducted (using the NetBeans Lookup API as outlined in 4.3.2) to find one. When an appropriate `TracepointRenderer` is found, it is cached and used to render the Trace information as it is received. The *JPanel* used as a render target has the internal `paint` function overridden. When a new Trace is received, a redraw of the *JPanel* is forced which in turn calls the rendering functions of the current `TracepointRenderer`. The rendering process runs on a different thread and does not hold up the program being debugged. While this means that there is a potential for tracepoint hits to be missed and not rendered, the assumption is that at that point the renderer has too high a frame rate for the user to be able to perceive it.

While initial work focused on the use of OpenGL as the rendering subsystem for visualisations, the system now supports the Java2D libraries by default. Dependency issues with JOGL, the Java OpenGL libraries, caused some difficulties when the system was packaged for deployment. Although JOGL is part of the official Java standard, some additional libraries must still be installed, which complicates deployment of the IDE. JOGL visualisations can still be created, however the system does not provide a superclass with that functionality at present.

TracepointRenderer

The abstract `TracepointRenderer` class is provided by the Tracepoint API. Subclasses must override three abstract functions. The `canRender` function takes a `Tracepoint` instance

and a `Trace` instance and returns a boolean signifying whether or not the class is capable of rendering the given objects. This function is used in the search for an appropriate renderer. The `preRender` function is provided to permit the renderer to perform calculations (and cache the results) when a new `Trace` is received. Finally, the `render` function actually renders the `Trace` using a provided `Graphics2D` object (taken from the `paint` function of the `JPanel`). Once a subclass has been created within a new module, the fully-qualified classname must be listed in a file located in the `META-INF.debugger.tracepoints` directory and named `org.netbeans.api.tracepoints.TracepointRenderer`. Once the module is installed, even during a debugging session, the renderer is immediately available.

It is also important to note that the `TracepointRenderer` deals only with the `Tracepoint` base class — this means that any developer seeking to add a new renderer needs only add the `api.tracepoints` module as a dependency to their module. A dependency on the target language module is not required, although it can be useful to access more in-depth functionality of the language-specific `Tracepoint` implementation. Examples of this can be found in the `Player laser` and `ultrasonic` renderers implemented in this project and detailed below.

GdbLaserRenderer

This renderer makes use of the `GdbTracepoint` type implemented inside the `cnd.debugger.gdb` plugin supplied with NetBeans. Use of the native GDB `Tracepoint` type is necessary in order to modify the serialization of pointers within the laser data structure. `Player laser` rangefinder data, as previously discussed in sections `x y`, is stored within a `playerc_laser_t` structure. The member of most interest is `ranges`, a pointer to a dynamically allocated array of double-precision range values.

The `canRender` function performs four validity checks on the supplied `Tracepoint` and `Trace`:

- That the `Tracepoint` object is an instance of `GdbTracepoint`
- That the `getLanguage()` function of the `Tracepoint` returns “C”.
- That the `Trace` is an instance of `Trace.Struct`
- That the `Trace` type is `playerc_laser_t`

If these checks are all successful, the renderer returns `true` to signify that it is able to render the `Trace` data.

The `preRender` function is where the functionality of `GdbTracepoint` is used. First, the `ranges` member of the `Trace` is retrieved. If this member is not an instance of



Figure 5.11: Screenshot of GdbLaserRenderer

Trace.Array, then `ranges` is still being serialized as a pointer (see Section 5.1.2). The renderer calls the `setPointerOptions` function and passes “ranges” and “scan_count”, signifying that the `ranges` member of the tracepoint expression result is a pointer to be serialized as an array of length `scan_count` (see diagram of `playerc_laser_t` structure in figure 5.3(b)). The tracepoint is then flagged as edited with the Tracepoint-Manager, which causes the GDB plugin to briefly halt the inferior, delete the tracepoint and re-create it with the new serialization option. The renderer then exits, as without range information the Trace cannot be rendered. The next time the tracepoint is hit, the Trace should have the `ranges` member serialized as an array. If this is the case, `preRender` calculates a *Polygon* based on the range information and caches it within the renderer. When `render` is called, this polygon is drawn to the rendering area.

This demonstrates that it is possible for a renderer to have a degree of influence over the tracepoint itself if more information is required. The automatic step described above obviates the need for the user to specify the range serialization themselves. If the user wishes to render laser range data, but has not specified the correct serialization, the system can take that step itself.

PythonLaserRenderer

The *PythonLaserRenderer* has the same basic code as the *GdbLaserRenderer* class, with one exception. In Python, the laser range data is guaranteed to be serialized before it reaches the renderer, so the automatic step necessary with the *GdbLaserRenderer* is not required. The actual rendering of the Python laser information is exactly the same (as expected, as ultimately the underlying class of the data is exactly the same).

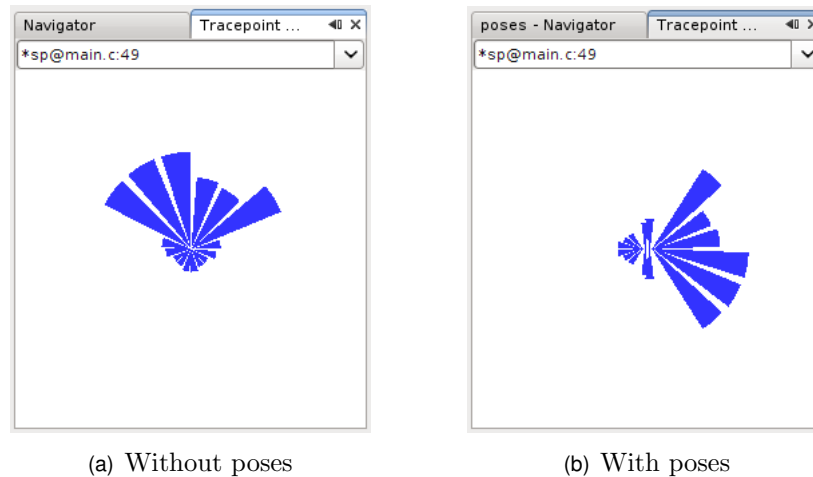
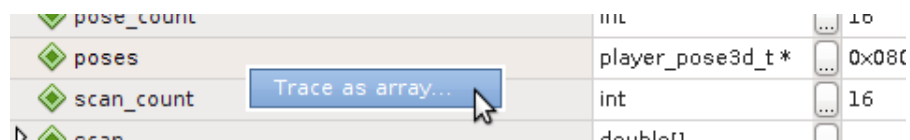


Figure 5.12: Screenshots of GdbSonarRenderer with and without pose information

GdbSonarRenderer

The final proof-of-concept renderer, the *GdbSonarRenderer*, is a demonstration of a multi-mode renderer. This renderer is capable of displaying the `playerc_sonar_t` structure. This represents the readings of a collection of ultrasonic rangefinders on a robot. One interesting feature of the structure is that it optionally contains pose information for the individual sensors. Additional Player library calls are required to populate this information. The render thus has two possible datasets to work with; ranges without poses, and ranges with poses. Accordingly, a multi-modal rendering algorithm was implemented. When pose information is not present, ranges are rendered as individual circle sectors centered at the origin and distributed evenly around 360 degrees. When pose information is present, the sectors are centered at the correct pose and pointing in the correct direction relative to the robot.

Both the range information and the pose information are pointers to dynamically allocated arrays, similar to the `playerc_laser_t` structure as detailed above. To demonstrate that the renderer can change modes while a debugging session is active, only the range information is automatically serialized. Initially, the first mode is seen (Fig. 5.12(a)). After the user manually specifies the serialization of the pose information (Figs. 5.13(a), 5.13(b)), the renderer snaps to the second mode (Fig. 5.12(b)). It can clearly be seen that the ranges are shown in a different order and are no longer evenly spaced.



pose_count	int	10
poses	player_pose3d_t *	0x080
scan_count	int	16
scan	double	16

(a) Before pointer serialization



pose_count	int	10
poses	player_pose3d_t[]	16
scan_count	int	16
scan	double	16

(b) After pointer serialization

Figure 5.13: Excerpt of Tracepoint View before and after pointer array serialization.

5.7 User Workflow

To place a tracepoint, the user would normally first navigate to the source line in question, before selecting the “New Breakpoint...” option from the Debug menu, as in figure 5.14 below:

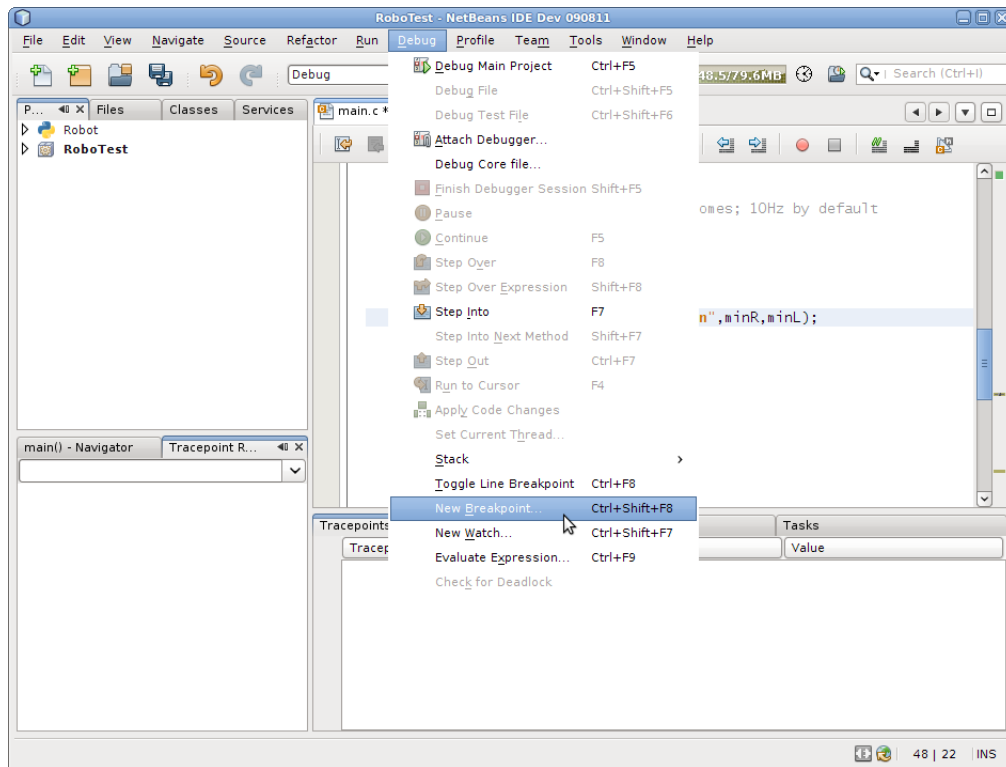


Figure 5.14: “New Breakpoint...” option on Debug menu.

The “New Breakpoint” window will then open. The user selects the correct category from the left drop-down, and then “Trace” from the right drop-down:

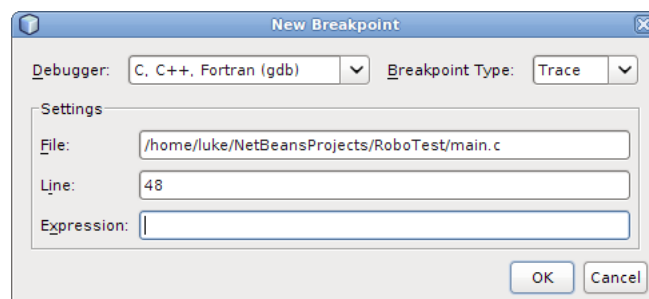


Figure 5.15: Setting initial tracepoint properties.

After entering the expression to evaluate, the user clicks “OK”. The tracepoint is now set and appears in the Tracepoint View. Once the tracepoint has been added, its presence

is shown at the target line in the code with a yellow glyph to the left, and a yellow highlight (see Fig. 5.16). A small tag is also shown next to the scrollbar on the right, indicating the approximate location of the tracepoint in the file. The user can enable/disable the tracepoint and edit its parameters by right-clicking the glyph.

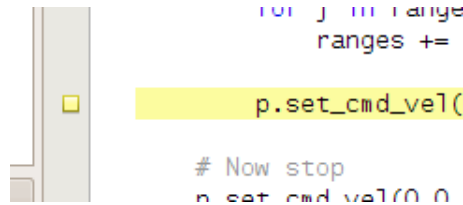


Figure 5.16: Tracepoint code annotation

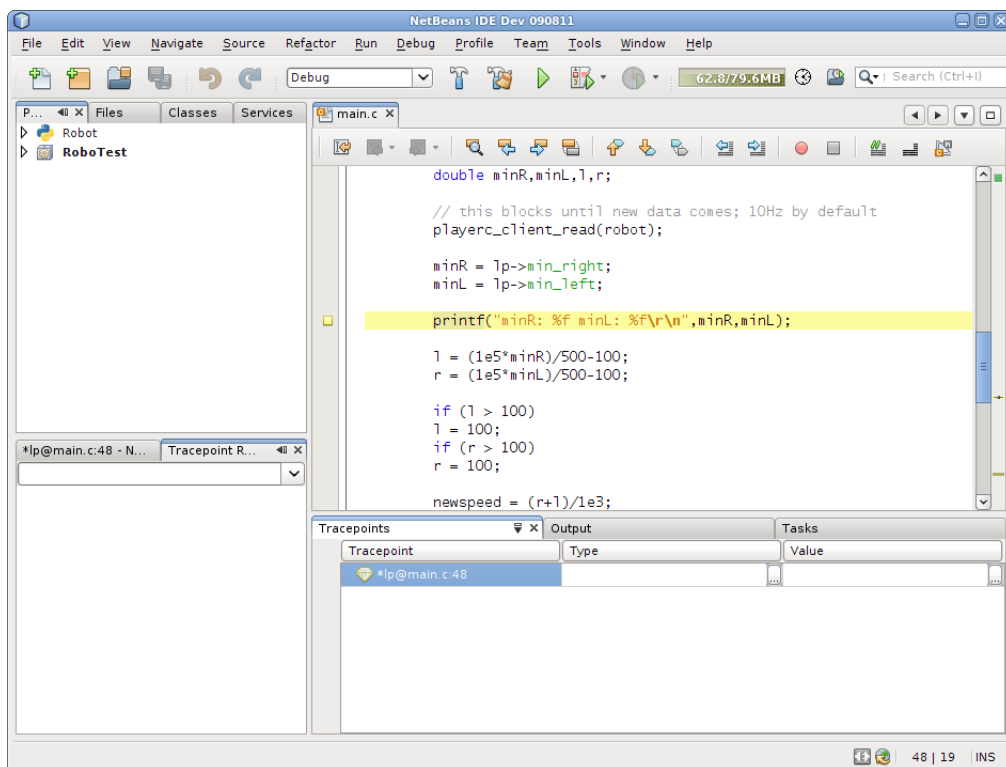


Figure 5.17: A tracepoint is set.

Now the user selects the tracepoint in the Tracepoint Render window on the lower-left, and starts a debug session. A TracepointRenderer starts updating within the Render View, and the Trace information is populated in the Tracepoint view. The user can expand the tracepoint in the Tracepoint View to examine the data being traced:

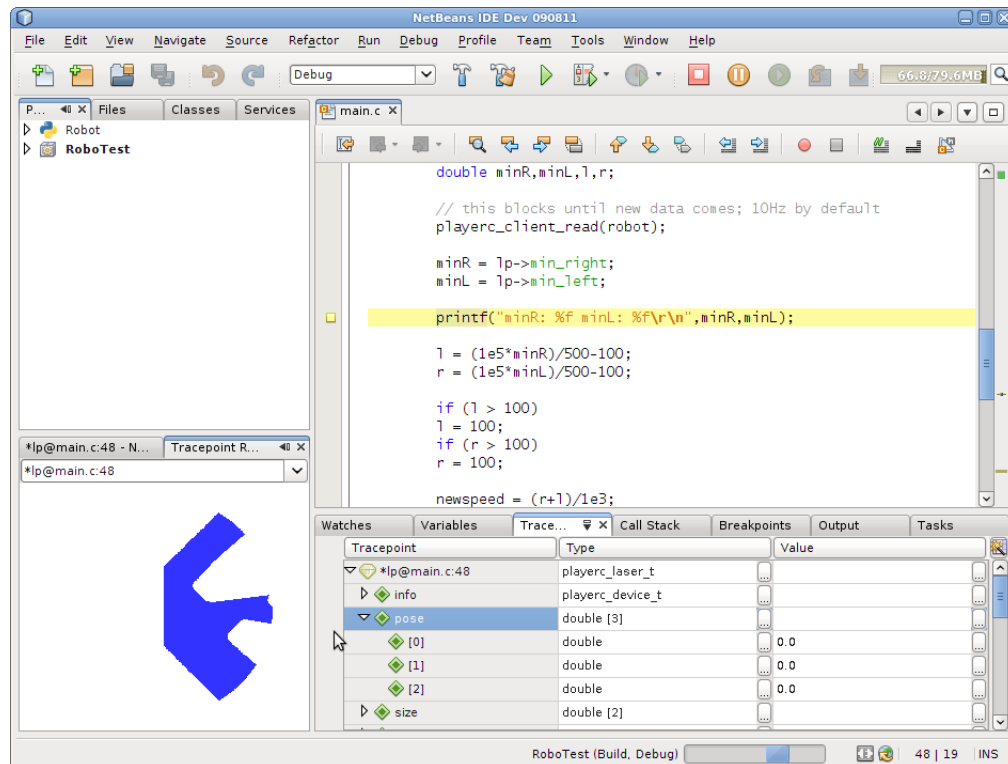


Figure 5.18: A debug session running with TracepointRenderer and Tracepoint View updating.

5.8 Extensibility

5.8.1 Adding Languages

Implementing tracepoints for a particular language, assuming the language is already supported by NetBeans, is a straightforward process. This section details the various steps that need to be taken in order to fully support tracepoints in a new target language.

UI Extensions

Only four steps are required to provide a tracepoint UI for a new language. Note that these steps only provide the user interface; the new tracepoints are settable and visible everywhere in the IDE, however until the underlying functionality is implemented in the language debugger, they are merely placeholders.

1. Subclass Tracepoint, including an implementation of the `getLanguage()` function. This function returns the name of the language targeted by the tracepoint type. While no further functionality is required, language-specific functions can be implemented. For example, pointer serialization within GDB is handled within the `GdbTracepoint` class.

2. Subclass *TracepointType*. This class is itself a subclass of *BreakpointType*, and is the method by which new breakpoint types are inserted into the “New Breakpoint” window. Required functions are `getCategoryDisplayName()` (returns the name of the category where the tracepoint appears), `getTypeDisplayName()` (returns the name of the new type, usually “Trace”), `getCustomizer()` (returns a *JComponent* containing the controls used to edit the tracepoint) and `isDefault()` (signifies whether or not the tracepoint should be the default result for this category). The fully-qualified name of the subclass must also be listed in `org.netbeans.spi.debugger.ui.BreakpointType`.
3. Subclass *AbstractTracepointPanel*. Use of this class is not mandatory, but it does provide a *JComponent* suitable to be returned by a *TracepointType* (see previous step). The panel includes filename, line and expression entries which are automatically populated with the current cursor position. The `finished` function must be implemented by the subclass. This function is called when the user clicks “OK”. It takes a *Line* and a string and returns a boolean signifying whether or not the operation was successful. It is here that the *Tracepoint* subclass should be created and registered with the *TracepointManager*.
4. Subclass *TracepointsReader*. This class provides a skeleton for persisting *Tracepoints* between IDE sessions. The persistence is handled by the *Properties* class exposed by the NetBeans Debugger API. Three functions must be implemented in a *TracepointsReader* subclass. The `getSupportedClassnames` function returns an array of strings, each one the fully-qualified name of a class that the *TracepointsReader* can persist. The `TracepointsReader.write` function takes an *Object* and a *Properties* cache and writes the *Object* to the cache. The base *Tracepoint* class includes a `write` function that takes a *properties* cache and persists the basic *Tracepoint* parameters. This function can be called directly if a *Tracepoint* is to be serialized. The `TracepointsReader.read` function takes a *classname* and a *Properties* cache, and uses this to reconstitute the persisted object. Again, the base *Tracepoint* class has a constructor that takes a *Properties* cache and reads the basic parameters for a *Tracepoint* from it.

Debugger Extensions

The steps required to implement tracepoints in the NetBeans plugin for a language debugger are very straightforward. These steps assume that the debugger already supports tracepoints, but that they have not been added to the IDE plugin yet. All *Tracepoint* API calls are completely thread-safe and can be called from anywhere.

1. Create a class that implements `TracepointManagerListener`. This would usually be an anonymous class inside the main debugger controller. Include a static block in the anonymous class that calls the `addTracepointManagerListener` function in the `TracepointManager` singleton. The three required interface functions each take a `Tracepoint`, and allow the debugger to know when the user adds, removes or edits a tracepoint. First check the parameter to ensure it is of the correct type (i.e. an instance of the `Tracepoint` subclass which is specific to the target language). After this has been confirmed, appropriate action can be taken to instruct the debugger to add or remove the tracepoint.
2. In the function that initializes the debugger (when the user starts a new session), retrieve a list of the tracepoints for the language with a call to the `getTracepoints` function of the `TracepointManager`. This function takes a class as a parameter and returns a list of `Tracepoints` of the specified class. Ensure all the returned tracepoints are added to the debugger (if enabled).
3. When a tracepoint is hit, convert the value of the expression to a `Trace` tree. Call `Tracepoint.setValue` with the new `Trace` tree as a parameter. This will update the `Tracepoint` value throughout the IDE, automatically updating the `TracepointView` and every listening `TracepointSink` (including those within `TracepointRenderers`).

5.8.2 Adding Visualisations

Visualisations can be added to any NetBeans module that depends on the `api.tracepoints` module. Multiple visualisations can be provided by a single module. Any traced type can be visualised; See 5.6.1 for details.

5.8.3 Adding New Features

TracepointSink

The `TracepointSink` class provides an extremely flexible way for any module that depends on `api.tracepoints` to access and consume tracepoint information. The `TracepointRenderer` class on which visualisations are based is little more than a `TracepointSink` combined with a service infrastructure to make it easy for external developers to extend. This class could very easily be used to provide logging, profiling or performance monitoring tools.

New Tracepoint Actions

The menu displayed when a user right-clicks on a tracepoint in the Tracepoint View is extensible through the *TraceNodeAction* class. Subclasses need only provide a name (through the *TraceNodeAction* constructor), and implement two functions. The `showFor` function decides whether the option should be displayed for a particular Tracepoint, *TraceNode* and *Trace*. The `actionPerformed` function is called when the user clicks the menu item.

6

Evaluation

This chapter evaluates whether or not the solution as implemented meets the requirements originally specified in Chapter 3. A discussion of each requirement and how it was met forms the basis of this evaluation. All debuggers affect the performance of the software being debugged, so the amount of overhead incurred by tracepoints is a particular concern. To measure this overhead, benchmark testing was conducted of the modified debuggers (external to the IDE). This determined the period of time for which the target program is halted while the tracepoint is evaluated and reported to the user. Both the Python and C debuggers have been tested across all three targeted operating systems.

The initial system requirements as developed in Section 3.2 comprised five key areas of concern for any tool for robot debugging:

Compatibility The capability of the system to work with a variety of operating systems, languages, and middlewares.

Real world How useful the system is in debugging real-world tests, without the use of a simulator.

Usability How little interaction is required by the developer to produce useful debugging output.

Extensibility The ability for the system to be extended by a third party.

Performance The degree to which the system affects the execution of the program being debugged.

In the following sections, the solution developed in this work is evaluated according to these requirements.

6.1 Compatibility

The system works across three different operating systems, so can be considered fully cross-platform. The tracepoint concept has been implemented in the debuggers of two significantly different programming languages (see Section 4.2). A plugin API for the NetBeans IDE has been developed to support implementation of the concept in further debuggers. By supporting two programming languages at release and providing for the addition of support to more languages, the requirement of support for multiple languages is satisfied.

Although testing was only performed with Player/Stage, the system remains theoretically compatible with any middleware as it permits access to any data accessible by statements in the target language. The only Player-specific elements of the system at release are the reference visualisations. These are intended to provide a demonstration of the visualisation API to permit third-party developers to implement visualisations for other middlewares. This is sufficient to ensure the compatibility of the tracepoint implementation with other middlewares.

6.2 Real world

The system was designed such that data gathering overheads would be low enough to disregard the probe effect without the use of a simulator. The comprehensive benchmarking results presented in Section 6.5.3 and Section 6.5.4 confirm that this is the case. The solution is therefore suitable for debugging in the real world.

6.3 Usability

The requirement of usability was considered during the development of the system in a number of ways. The selection of the Tracepoint concept provides a familiar interface to any developer who has previously made use of breakpoints. The integration of system features within an IDE streamlines the workflow of the user. In an advance on the Robotic IDE project by Gumbley and MacDonald [23], no configuration is required to make use of the system.

To fully verify the usability of the system would require a usability study, surveying the experiences and responses of a variety of developers when using the system. Such a study is outside the scope of this work, although it is suggested as a suitable area for future work (see Section 7.2).

6.4 Extensibility

Every aspect of the system has been designed to be extensible. At a basic level, the source code for the system has been released as open source. More comprehensive extensibility is provided through the implementation of a full tracepoint API for NetBeans to permit the addition of support for further languages and visualisations. The steps required to add new visualisations and languages are summarized in Section 5.8. If IDE extensions are required, NetBeans is itself open-source, modular and extensible.

6.5 Performance

When a developer uses the system to place a tracepoint in their program, a small amount of computation is performed every time the program passes that point. If this interruption takes too long, it could have a perceptible probe effect, changing the behaviour of the program being debugged. To verify that the system is fast enough to render the probe effect negligible, comprehensive benchmarking has been performed, and is detailed in this section. Overall charts combining all benchmark tests performed are available in Appendix B in Figures B.1 and B.2. This appendix also contains the raw results, in Tables B.1, B.2 and B.3.

6.5.1 Test System

Tests were conducted on a late 2008 13" aluminium Macbook (version 5.1). System specifications are as follows:

- Intel P8600 2.4GHz Core 2 Duo processor
- 4Gb DDR4 1066MHz RAM
- Fujitsu MHZ2250BH 250Gb, 5400RPM HD
- Vista Ultimate with service pack 2
- Mac OS X 10.5 Leopard
- Ubuntu 9.04 Jaunty Jackalope, with the 2.6.28-17 Linux kernel.

- CPython interpreter version 2.6.4

6.5.2 Methodology

To get as clean and consistent a picture of debugger overheads as possible, testing was not done within NetBeans. In practice, the debugger and the code being debugged operate separately to the IDE. The only circumstance in which the execution of code could be interrupted is a buffer overflow. This could occur where the debugger is outputting information more quickly than it is being read by the IDE over an extended period. This testing therefore assumes buffer overflows are exceptional situations not likely to occur during normal operation.

A number of test scripts were written for each debugger, both to control the debugger and to provide code to debug. All testing was completed in as clean an environment as possible. All non-essential processes were terminated, screensavers and networking disabled, and in Linux even the GDM (Gnome Display Manager, the GUI service) was stopped. Tests were run in a single terminal — `bash` in Linux, `Terminal` in OSX and both `MSYS` (GDB only) and `cmd` in Vista. All test script output is redirected to a null sink, `/dev/null` on Linux, OSX and `MSYS` and `NUL` in `cmd` — this prevents any screen output overhead from influencing the test results. There was no interaction with the test machine during the test. Tests are set up and executed automatically. One command runs through the entire test series, comprising 21 individual tests for Python and 55 for C.

An individual test involves debugging a test program with a timed loop calculating the square root of the iterator variable. Before the actual timed code, a delay of one second was used to give the system time to complete pending I/O after test setup. One or more tracepoints are placed inside the loop before execution, containing expressions that evaluate to global variables of a variety of different types defined earlier in the program. The loop is executed and timed multiple times, with the number of iterations starting at a minimum of 1000 and increased until the loop duration is between five and ten seconds. The final measured duration of the loop code and the recorded number of iterations are then appended by the debugged code to a results file. From these figures, tracepoint overheads can be measured.

Python Methodology

Benchmarking the python debugger is straightforward as it is part of the same process (and thread) as the program being debugged. Debugger code is called as if by a function call at the tracepoint location. Process-level timing code can therefore be added to the inferior to obtain the CPU time consumed by the debugger without being affected by

system loading. A script called `benchmark.py` was written to control the debugger (`jpydaemon.py`) by providing TCP/IP socket connections and transmitting debugging commands. The timing was provided by the script being debugged, called `test.py` — the `time.clock()` function was used to get the current CPU time of the process before and after the stress code.

Only small changes were required between operating systems. The locations of the test scripts were slightly different in each case. The code for executing a child process also differed — Linux and OSX used a combination of `os.fork` and `os.execlp`, but as `os.fork` is unavailable in Windows the `os.spawnl` function was used instead (equivalent to a combined `fork` and `execl`). One final difference was that Linux provides low-level process CPU affinity control through the `taskset` command. This allowed all operating processes to be switched to CPU 1, while the test was executed on CPU 2. Although this was used for all Linux testing, this difference is not so important for Python due to the use of process-level timing; it is explained in more detail in the section on GDB. Such low-level control is not available in OSX and Vista.

C Methodology

A different procedure must be used for benchmarking GDB than Python. As GDB executes in a separate process to the inferior, any measurements of CPU time consumed by the inferior will not exhibit tracepoint overheads because the inferior is not executing while tracepoints are being evaluated and serialized. The time spent in GDB could have been measured, but this would have involved significant alterations to the GDB source. This also would not have captured the time taken to switch processes from the inferior to GDB and back, potentially a significant overhead. To account for this, while testing GDB the inferior measures the system-wide time before and after the loop. While this means that system loading will affect the final timings, as the benchmarks were conducted in as clean an execution environment as possible on a dual-core machine, the amount of task-switching taking place is assumed to be minimal. GDB testing was performed separately for both control interfaces, the command line interface (CLI) and the machine interface (MI).

The tests themselves were controlled by a GDB script. When GDB is executed, it checks the current directory for a file named `.gdbinit`. If found, the file is assumed to contain a list of GDB commands to be automatically executed on startup. This file was used to set up each test, setting and clearing tracepoints and executing the test code. Both the MI and CLI interpreters make use of this script. Although some tracepoint commands are MI-specific, they can be executed in either interpreter by making use of the `interpreter-exec` GDB command. Only MI-specific commands are executed using this command; all other commands used are the CLI equivalents. This

has no effect on performance during a test run. The test code was contained within a file called `benchmark.c`, and used the `gettimeofday()` function to retrieve a `struct timeval` containing the system time accurate to the microsecond. With GDB there is additional overhead the first time a tracepoint is encountered; this is one reason why tests are run multiple times. After the first test run, type information is no longer being serialized. Tests are therefore not contaminated by this one-off overhead. A specific test, using a benchmarking mode of GDB, is used to assess type serialization overhead independently.

Testing between operating systems was only different in two ways. The `gettimeofday()` function is not available in Vista. In its place, the `GetSystemTimeAsFileTime()` function was used. This function actually has a higher stated resolution than `gettimeofday()` (100 ns as opposed to 1 μ s). The other change was the use of the `taskset` command in Linux. This command forced all processes to CPU 1 before running GDB on CPU 2. When GDB executed the inferior it also ran on the second CPU, as child processes inherit CPU affinity from their parent. This should have guaranteed a pristine execution environment for GDB and the inferior on Linux and the most accurate possible picture of the inherent overheads of the tracepoint system. Neither Vista or OSX provide equivalent low-level affinity control; results in those two operating systems are expected to be more affected by system load.

6.5.3 C Benchmarks

This section presents and explains the individual GDB benchmark tests and gives a detailed analysis of the results and their implications regarding tracepoint performance in C. Two basic tests were conducted to provide a baseline from which tracepoint-specific overheads could be calculated. First, the system was run within the debugger but without any tracepoints inserted. This is referred to as the *Raw* baseline. Execution in this instance is no different from a straight program run. Execution time per iteration on this run was less than 30 ns in all operating systems. The second baseline, referred to in this section as the *Zero* baseline, was a tracepoint with an attached expression of “0”, to gauge a minimum tracepoint overhead. “0” was used as this statement is short, contains no operations and references no variables. The fastest Zero baseline was 76 μ s, for the MI in Linux. The reason for the difference between the Raw and Zero baselines is that in the Raw baseline no context switching is performed, whereas in the Zero baseline the inferior is forced to halt before execution is resumed by the debugger. Due to the extreme disparity between the Raw and Zero baselines (over three orders of magnitude), the contribution of the raw code is ignored in these benchmarks. The results of these baseline tests across all operating systems can be seen in Table 6.1.

	OSX CLI	OSX MI	MSYS CLI	MSYS MI	Vista CLI	Vista MI	Ubuntu CLI	Ubuntu MI
raw	0	0	0.02	0.02	0.02	0.02	0.03	0.02
zero	2792.81	3022.23	747.17	359.56	778.2	317.74	87.77	75.89

Table 6.1: GDB baseline test results (μ s).

Throughput Test

This test inserted a tracepoint with an expression that evaluated to a fixed-length array of characters. The objective was to test how the system performed with equivalent expressions that evaluated into simple types of varying sizes. A number of small-size arrays (1-8 bytes in length) were tested as there was an initial hypothesis that the overhead of base64 serialization would be noticeably affected by the modulus of the data length. After this initial small-scale series, array size doubled with each test until the maximum tested array size, 8192 bytes. The actual content of the arrays was not set as serialization overhead is not dependent on content (only length). The results of this test can be seen in the graph in Figure 6.1.

The first thing to note is that significant measurement inconsistencies are present in the results. These inconsistencies are due to the use of system time as a benchmarking metric (see section 6.5.2). Although this was necessary to gain a truer picture of tracepoint overheads, it allows several non-deterministic factors to become present in the results. Task switching is the primary cause of the measurement noise; as the host operating systems do not have real-time capabilities, the time taken to switch between processes cannot be eliminated or made consistent.

Although a clear overall correlation cannot be seen, particularly at small array sizes, a significant upward trend in execution time can be seen towards the upper limit of tested array sizes in the MI tests. In MI mode, GDB outputs the full contents of the traced value, Base64 encoded — therefore execution time should increase with array length. The low measured correlation between execution time and array length indicates that serialization and transmission of values is a relatively small component of the overall overhead. Consider the Linux MI results. The lowest recorded execution time was 79 μ s (7 bytes), and the highest was 117 μ s (8192 bytes). The Zero baseline took 76 μ s. Thus even the largest array size tested results in an increase of only 41 μ s, just 35% of the total execution time for that test. In other operating systems serialization is even less significant — making up a proportion of execution time smaller than measurement noise.

The CLI results vary between operating systems but generally display an initial increase, followed by a flat period with another slight final rise. In CLI mode, GDB outputs a human-readable representation of the value. With an array that means a varying number of array elements is output, observed during testing at 200 to 300. Repetitive arrays are signalled by an output (in one case) of “<repeats 8191 times>”. CLI output can

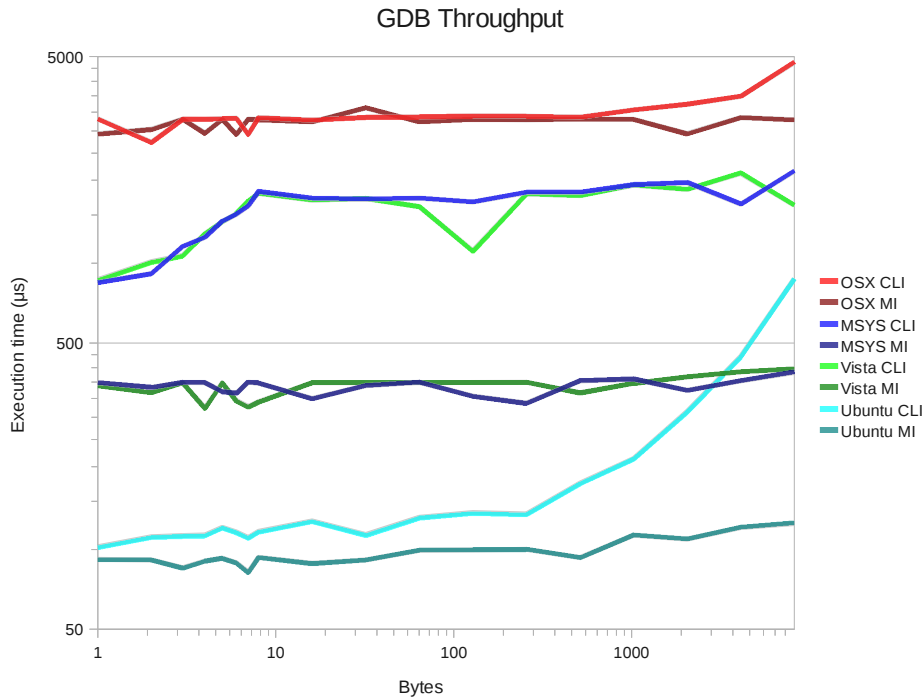


Figure 6.1: Tracepoint throughput for GDB.

therefore remain constant for larger array sizes, as for OSX and Vista, or increase with array size as with Linux. On different operating systems, with different array initialization styles, different CLI outputs may be used.

Multiple Tracepoint Test

This test inserted multiple copies of a tracepoint at the same position. Because every tracepoint is set at the same location, the same instruction in the inferior code would be overwritten. Every time a tracepoint (or breakpoint) is struck, GDB iterates through the list of tracepoints and breakpoints to determine which was struck. Since multiple points can be hit at the same time, GDB runs through the conditions and actions for every matching point before resuming execution. The outcome of all this is that a task switch from the inferior to GDB and back only occurs once for all tracepoints, instead of once for each. This allows the task switching overhead, suggested by the Zero baseline, to be confirmed as all other aspects of a tracepoint strike can be eliminated. The expression for all added tracepoints was “c8192”, corresponding to the largest array from the test in section 6.5.3.

As expected, in all cases the graph of the results in figure 6.2 shows a linear correlation between the number of tracepoints struck and the execution time. The discrepancies of CLI output as evidenced in section 6.5.3 are not evident here as the same data is being

	OSX	Vista (MSYS)	Vista (CLI)	Ubuntu
CLI intercept	3128	308	546	187
CLI baseline	2793	747	778	88
MI intercept	3117	212	164	67
MI baseline	3022	360	318	76
Intercept average	3123	260	355	127

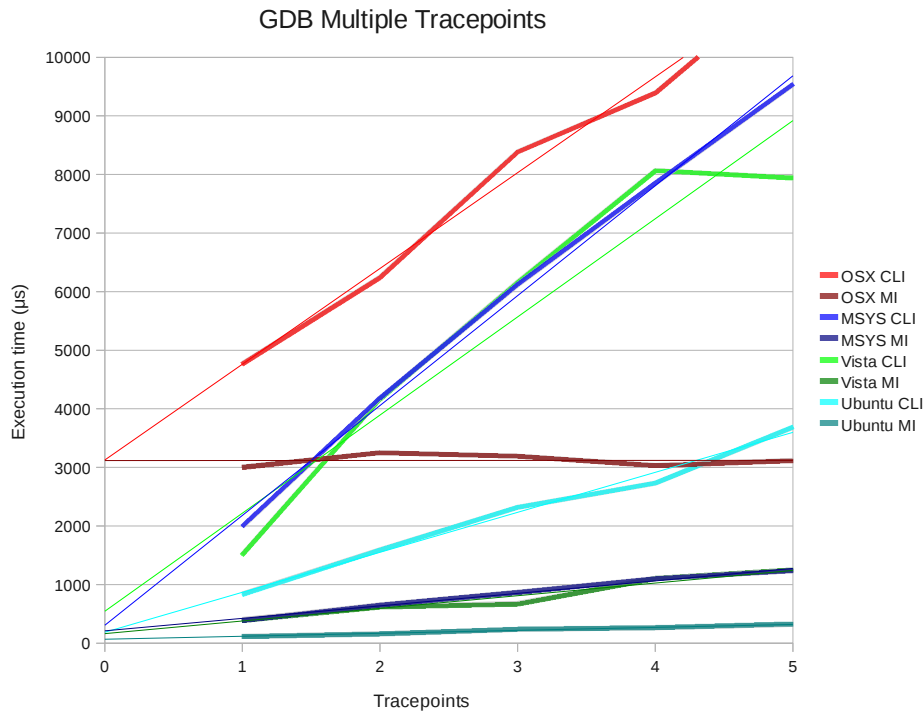
Table 6.2: GDB multiple tracepoint test results (Fig. 6.2) compared with baselines (μs).

Figure 6.2: GDB multiple tracepoint benchmark results.

output each time, with no changes to the program state between outputs. The expectation was also that the task switching and other assorted overheads would be equivalent for the CLI and MI interpreter. This can clearly be seen in the converging trend lines for each operating system, particularly for Mac OS X. The graph shows that task switching takes on the order of thirty times longer for OSX than for Ubuntu. This makes task switching by far the dominant component of tracepoint overhead in OSX. The final prediction was that the extrapolated task switch time would be approximately equal to the Zero baseline, demonstrating the minimal overhead of evaluating and serializing the expression in that test. For the MI interpreter this holds true; for CLI, results are less consistent. The reasons for this CLI inconsistency are likely the same as those detailed in section 6.5.3. Table 6.2 shows a summary of trendline intercepts (from Figure 6.2) and zero baseline results.

Structure Members Test

This test demonstrated the difference between the regular CLI output and the MI output for compound types. After the first time a tracepoint is hit, the MI interface outputs only a memory dump of the value with no attached type information. The CLI interface, conversely, must output type information every time the tracepoint is struck, in human-readable c-style syntax. Therefore the complexity of the serialized type should make no difference to the MI, but a significant difference to the CLI. To test this, a number of structures were created to have equivalent size but different numbers of members. For simplicity, and to make sufficiently large structures to load the system, all struct members were of type `long long int` (64-bit unsigned integers). The variables to be traced were named “sm10”, “sm20” and so on, with the number indicating the number of members in the structure. “sm100” contained one hundred `long long int` members. “sm90” contained 89 `long long ints`, and one array of eleven `long long ints`. In this way the size was kept constant while varying the complexity of the type.

Benchmark results can be seen in figures 6.4 and 6.3. The results have been graphed separately due to the large differences in measured execution time between the CLI and the MI. These graphs show the expected outcome. Execution time for the CLI rises in a fairly linear fashion with the number of members, while the MI displays very little variation. This validates the approach taken with the machine interpreter, of only serializing type information once.

MI Type Serialization Test

This test measured the overhead of MI type serialization. The test was exactly the same as in the previous section, with the exception that a special benchmarking flag was toggled. This forced the MI to output type information every time the tracepoint was struck. Although this overhead is normally only incurred the first time a tracepoint is struck, in some applications this may be a significant consideration. A graph of the resulting execution times can be seen in figure 6.5 ¹. Combining these figures with those from the previous test allows the calculation of the tracepoint serialization overhead for structures of varying complexity. The previous test (Figure 6.3) measures the time taken to serialize the content of the variable. This test (Figure 6.5) measures the time to serialize both the content and the type of the variable. Subtracting Figure 6.3 from Figure 6.5 gives Figure 6.6. The graph shows a steady increase in the MI type serialization overhead, directly correlated with the number of members as expected. OSX summary figures are shown to further demonstrate a key issue with GDB tracepoints in OSX; the task switching overhead is so variable and so high that it is virtually impossible to see any correlation.

¹OSX figures are not visible in this graph as they are off the scale.

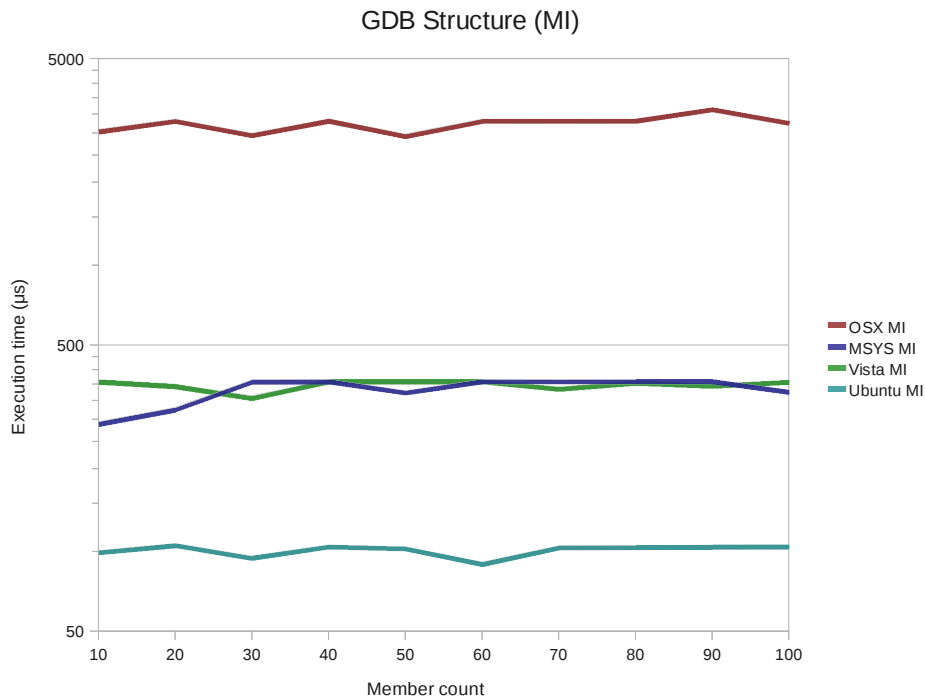


Figure 6.3: GDB MI tracepoint overheads for structures with varying member counts.



Figure 6.4: GDB CLI tracepoint overheads for structures with varying member counts.

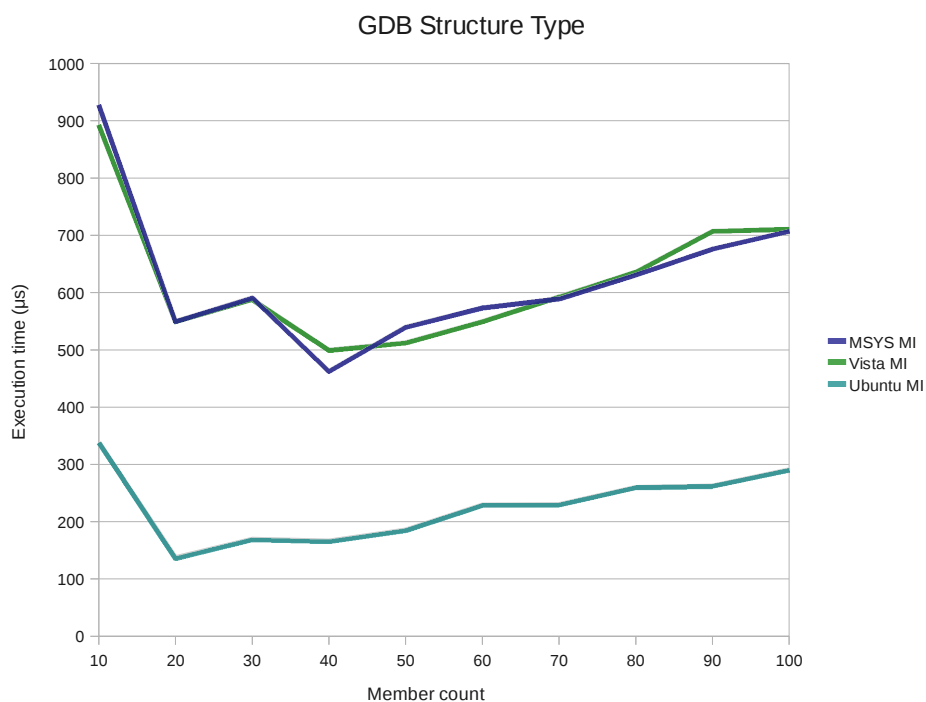


Figure 6.5: GDB MI tracepoint execution times with type information always serialized.

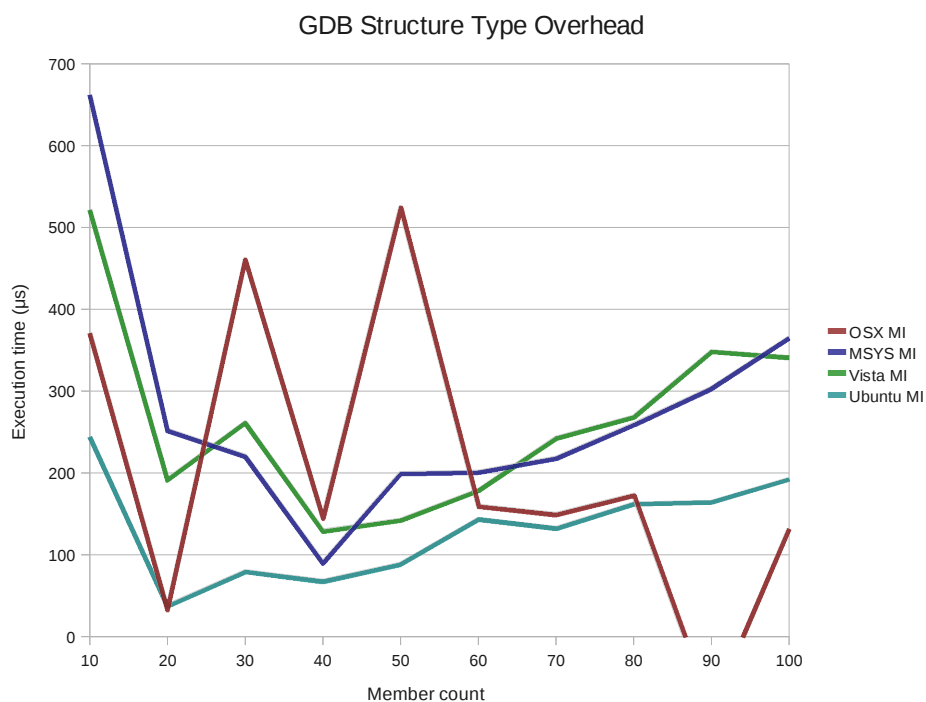


Figure 6.6: GDB MI tracepoint type serialization overheads.

Pointer Serialization Test

Another option for serialization in GDB concerns what to do when the tracepoint result contains a pointer. For an explanation of the options available to the user, see Section 5.2.2. For an explanation of why this is necessary, see Section 5.2.6.

For this test, eight pointers were created (all of type `char*`). Four hold equivalent string constants, each with 64 characters and a terminating null. The other four hold heap allocations of 64 bytes apiece. Of each group of four, one is global and three are within a common structure, named `ptrStruct`. The `ptrStruct` structure also contains a length member, `int length`, set to 64.

Five tests were run:

1. The global buffer is serialized with a constant expression.
2. The global string constant is serialized as a null-terminated string.
3. 1, 2 and 3 of the structure buffers are serialized with a constant expression.
4. 1, 2 and 3 of the structure buffers are serialized with the `length` member.
5. 1, 2 and 3 of the structure string constants are serialized as a null-terminated string.

In tests 3, 4 and 5 of this benchmark, the number of pointers serialized are varied. Since as many pointers in a structure can be serialized as the user requires, this permits the calculation of the cost for each method and suggests which method is the most efficient. The results of these tests are given in Tables 6.3 and 6.4. These results show that where a pointer outside a structure is serialized (as in the first two tests detailed above), serializing as null-terminated is significantly faster. This is most likely due to the fact that in the first test the pointer is initialized with a constant value and never changed, and the contents of the memory pointed to are constant. This means the memory to be dumped is stored in the program code, which is accessible to GDB without reading the memory of the inferior. By contrast, the global buffer does not have a constant value or constant contents, being instead given a heap space allocation by `malloc` at runtime. This conclusion is supported in tests 3-5. In these tests, the three methods are roughly equivalent. In this case, none of the members are initialized, being instead set in a function call at the start of the test (although the memory contents are the same).

6.5.4 Python Benchmarks

The Python benchmarks are much simpler than those for C. Since the benchmarking itself is more accurate (see section 6.5.2), and the serialization less complicated, fewer benchmarks were necessary to fully evaluate system overheads. Performance has also proved

Type	OSX	MSYS	Vista	Ubuntu
expression	3012	621	619	104
string	3036	421	454	99

Table 6.3: GDB pointer serialization test results (μ s).

Type	Members	OSX	MSYS	Vista	Ubuntu
expression	1	3009	456	456	93
expression	2	3007	551	549	112
expression	3	3030	641	632	115
member	1	3016	456	376	102
member	2	3233	493	546	109
member	3	3081	637	637	116
string	1	3053	455	459	103
string	2	3261	548	540	111
string	3	3016	641	642	105

Table 6.4: GDB pointer member serialization test results (μ s).

remarkably consistent across all operating systems. Full benchmark results are available in Appendix B; this section examines and explains the individual Python benchmarks performed. Common baseline tests were performed before any other test, to give an idea of the component of the final figures made up by the actual test code execution. The *Run* baseline timed the benchmarking code without using a debugger, and then the *Debug* baseline timed it within the debugger but with no tracepoints set. The results of these tests can be seen in Table 6.5. Across all operating systems the results in both cases were virtually identical. The *Trace* baseline added a single tracepoint evaluating a zero-length string. The slowest debug baseline was in OSX at 519 ns per iteration. The fastest zero baseline was in Vista, at 193 μ s per iteration. This is a difference of more than two orders of magnitude, meaning the influence of the raw code execution on the test results is negligible. The execution time of the basic code without tracepoints set is therefore disregarded in these results.

Test	Ubuntu	OSX	Vista
run	0.37	0.52	0.27
debug	0.4	0.52	0.26
zero	224.25	223.07	193.62

Table 6.5: Python baseline test results (μ s).

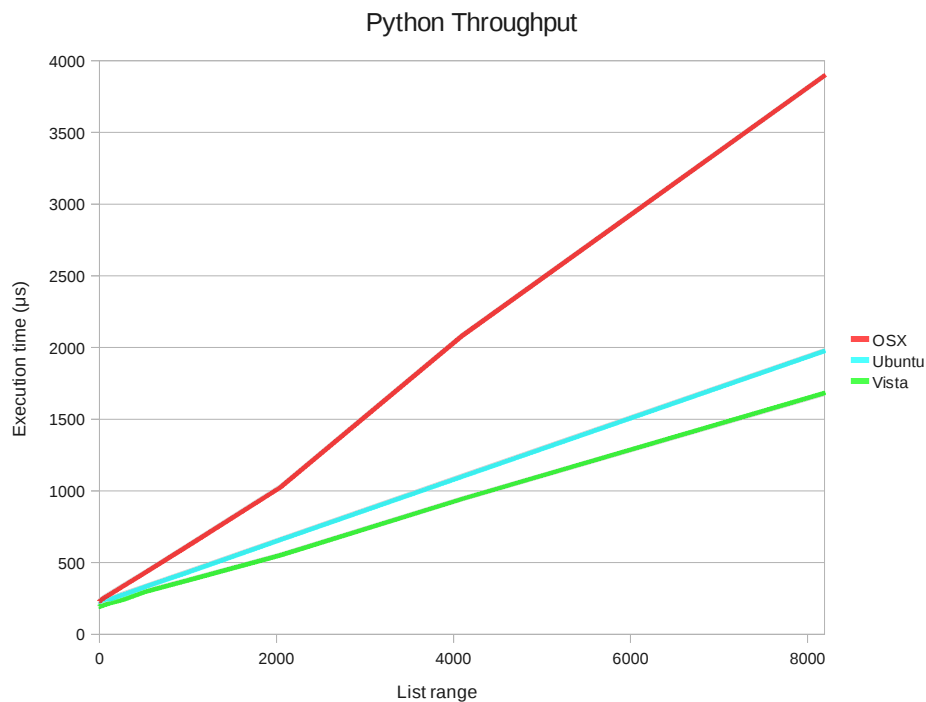


Figure 6.7: Python throughput benchmark results.

Throughput Test

The first Python benchmark is a test where a single tracepoint is placed that evaluates to a `list`. A number of these lists of different sizes are prepared using the `range()` function in order to give an idea of the overhead of serializing different amounts of data. Similarly to the C throughput tests the ranges used are powers of 2, starting at 1 and finishing at 8192. The initial lower readings added in C are not used here. Note that this test is not exactly parallel to the C throughput test — there, the exact size of the transfer could be calculated. In this instance, the `list` elements are integers, not characters, and they are pre-initialized to a set value (equal to their index in the `list`). The serialization is also quite different, wrapped in human-readable XML rather than a raw memory dump. These tests should thus be considered separately to the throughput tests for C. The results of this test are shown in Figure 6.7.

The benchmarks show a very clear, near-perfectly linear increase with array length. This is precisely as expected; serialisation and transmission with python should scale linearly with increasing amounts of data. The results vary between operating systems, with Vista the fastest, followed closely by Ubuntu and with OSX lagging behind significantly at about half the speed of the other two. One possible explanation for this is that Python development in a particular OS follows usage in that OS. Despite official builds of Python being available for OSX, these builds may not yet be fully optimized.

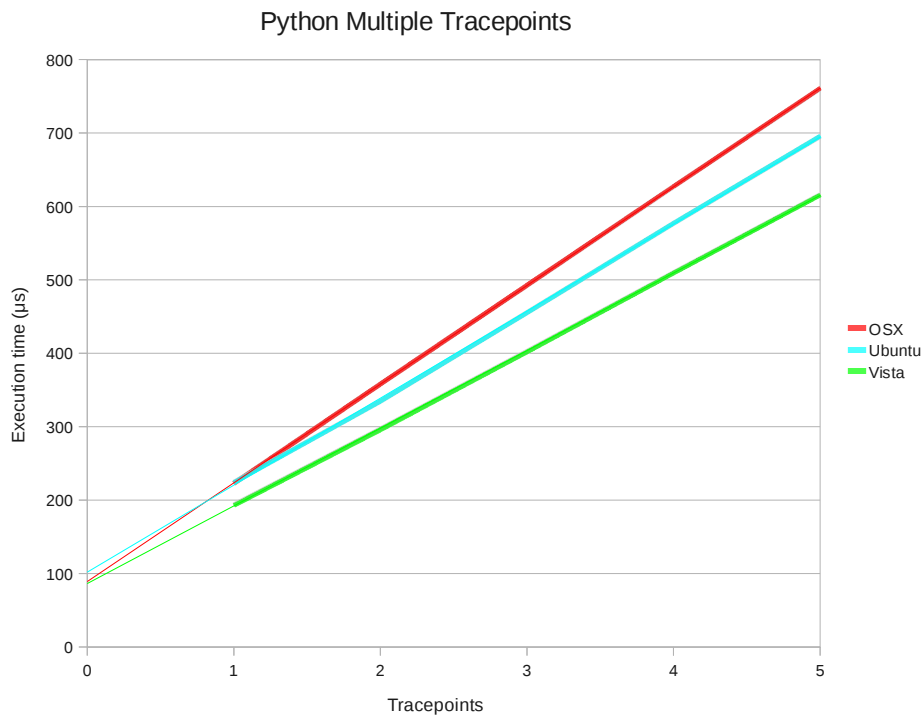


Figure 6.8: Python multiple tracepoint benchmark results.

Multiple Tracepoint Test

This benchmark places multiple tracepoints in the same place to calculate the overall tracepoint overhead. One to five tracepoints were placed, each evaluating an empty string. The results are shown in Figure 6.8. Once again, this test demonstrates a clearly linear relationship between the number of tracepoints placed and the execution time of the test code. When the trendlines are plotted, as in the figure, it can be seen that they do not intersect the Y-axis at anything like the Debug baseline result (where zero tracepoints were placed). This demonstrates the additional overhead incurred by the debugger in order to provide line-level resolution for tracepoint placement. Once a tracepoint is placed in a function, every codeline in the target function must be stepped through. This is what is represented by the execution times at the intercept.

These results can be used to calculate the per-line overhead in a traced function. As a single loop execution contained three lines of Python code, the trendline intercepts of the graph in Figure 6.8 were simply divided by three for each operating system. The results of this calculation are shown in Table 6.6

	Ubuntu	OSX	Vista
Intercept	101.95	89.07	86.52
Per-line	33.98	29.69	28.84

Table 6.6: Python per-line overhead extrapolation (μ s, intercepts from trendlines in Fig. 6.8).

6.5.5 Visual Studio

Visual Studio provides limited tracepoint functionality as described in sections 2.2 and 3.3.4. Tests were carried out using the same benchmarking code as for the C tests in Vista, described in section 6.5.3. The results can be seen in table 6.7 below. The “debug” test was a straight execution of the benchmark code with no tracepoint set. Execution time for this test is comparable to those for GDB and Python debugging sessions in Vista. The “zero” test was the zero baseline (as in section 6.5.3), where a tracepoint was set with a constant expression. Execution time for this test was 62 ms, as opposed to 318 μ s for the equivalent test with GDB MI — making GDB tracepoints almost two hundred times faster.

In all cases where a tracepoint is set in Visual Studio, an overhead of at least 62 ms is incurred. This suggests that when a tracepoint is struck, a large amount of processing takes place regardless of the complexity of the attached expression. Additionally, outputting the expression result may be time consuming as the output window is a GUI component. Although small variations are visible with the amount of data output, overall these are not significant compared to the base overhead.

Test	Time (μ s)
debug	0.081
zero	62416
c1	62431
strPtr	62431
sm100	62400
c8192	62447

Table 6.7: Visual Studio tracepoint benchmark results.

6.5.6 Summary — Performance

This section measured the delays imposed on debugged software by tracepoint evaluation and serialization. Tests were performed on both the Python and C debuggers, external to the IDE, across three operating systems. The same test machine was used for all operating systems. The tests themselves comprised repeated execution of a simple loop containing a

statement on which a tracepoint had been placed. The average execution time for a single loop was calculated over at least ten seconds. For both debuggers the execution time of the loop without a tracepoint placed was many orders of magnitude smaller than that with a tracepoint placed. The contribution of the loop code itself to execution time was thus considered negligible. The C debugger required more comprehensive testing than Python. This was due to its relative complexity. Tests were performed with single and multiple tracepoints, varying the size and composition of the traced type and calculating overheads for type and pointer serialization. Overall the tests showed that the practical overheads have two primary components — the time taken to switch control from the inferior to the debugger and back again, and the size in memory of the data being traced. Operation was fastest in Ubuntu with an average 127 μ s task switching overhead, followed by Windows Vista at 308 μ s, and finally OSX at 3123 μ s. The Python implementation made use of reflection, implementing tracepoint functionality within the program being debugged. This made it a far simpler implementation than with C. Combined with the fact that type information must always be serialized in Python, and the lack of multiple interpreters to test, Python performance was relatively simple to assess. Only two tests were required to characterise data throughput, per-line and per-tracepoint overheads. Performance between systems with increasing data was consistent, with Vista fastest, then Ubuntu, then OSX. Basic overheads for all operating systems were within 10% of the midpoints. These were 94 μ s for a tracepoint hit, plus 31 μ s per line in the target function.

6.6 Summary

This chapter performed an evaluation of the implemented tracepoint tool for robot debugging by evaluating each requirement as specified in Section 3.2. The five requirements evaluated were Compatibility, Real world suitability, Usability, Extensibility and Performance. Tracepoints have been tested with three operating systems and two programming languages. An API has been provided to permit the addition of support for further languages. Data is gathered via a method that makes the system independent of the robot middleware in use. The tracepoint system is thus compatible with a wide range of existing robot software. Benchmarking shows that overheads are low enough to disregard the probe effect without the use of a simulator. As the system permits debugging software in the real world, this requirement is also satisfied. Although the system has been designed with usability in mind, and includes several key features to improve usability, to fully determine whether or not the requirement of usability has been met would require a usability study, which is outside the scope of this work. The system permits extension in several different ways, including the release of source code, the use of an extensible

IDE, and the provision of APIs specifically designed to facilitate third-party extensions. Finally, the performance of the system has been thoroughly characterized. Overheads are strongly use dependent, however under the heaviest tested load, tracepoint evaluation took 11.4 ms. This compares favourably to other systems evaluated in Section 3.3.

7

Conclusions and Future Work

7.1 Conclusions

The robotics industry is in a period of rapid growth, enabled by new technology and fostered by emerging robotic applications such as medicine and heavy industry. With an ageing world population, one area of particular interest in the coming years is the use of robotics to assist in caring for the elderly. The efficiency of the robot development process and the availability of quality tools to aid this process is therefore a key concern for the robotics industry. Despite the heavy software development requirements of robotics, and the importance of debugging in software development, little work has been done dealing specifically with robotic debugging. The objective of this work is to develop a practical tool to assist in debugging faults in robot software.

The key problems when developing robotic software, as discussed in Section 1.1, are a lack of standardisation between different robotic platforms, and execution in the real world. Conventional software techniques are insufficient when considered in the context of these problems. A survey of the state of the art in debugging research reveals that the key activities performed by developers when debugging robotic software are Data Gathering and Abstraction. Due to the greater flexibility and more immediate feedback of the Data Gathering activity, the tool developed in this work primarily supports this activity.

Three case studies, past and present projects of this research group, are examined to identify the shortcomings with existing robot debugging approaches. Along with the

issues identified by research in Chapter 2, the findings are used to construct a set of key requirements for a robot debugging tool. These requirements include:

Compatibility The capability of the system to work with a variety of operating systems, languages, and middlewares.

Real world How useful the system is in debugging real-world tests, without the use of a simulator.

Usability How little interaction is required by the developer to produce useful debugging output.

Extensibility The ability for the system to be extended by a third party.

Performance The degree to which the system affects the execution of the program being debugged.

These requirements are applied to several existing solutions. No existing solution fully satisfies the criteria, although four are close — source code modification, standard software debuggers, the previous robotic IDE project and the RLog tool. A novel solution is proposed, involving the implementation of the tracepoint debugging construct. Tracepoints are similar to breakpoints, but instead of halting execution and waiting for user input, an attached expression is evaluated and the result returned to the user as execution resumes.

The languages targeted for the initial implementation are Python and C, chosen for their ubiquity and for the significant differences between the two languages. That the system supports two such disparate languages validates its applicability to different languages. The test application for the system uses the Player/Stage robot middleware, as it is widely used and includes a simulation platform, helpful for rapid testing. A small change to the SWIG-generated Python bindings for Player is developed to properly translate dynamic arrays to Python. This change has been accepted into the main trunk of the Player project since version 2.1.3.

A survey of existing IDEs gives a choice between the NetBeans and Eclipse platforms for extension. KDE is also considered but rejected as it has a significantly smaller user-base. Netbeans is chosen after initial work with Eclipse is abandoned due to several problems during development, including issues with programming efficiency and interface standardization between languages. NetBeans is modular, cross-platform and open source. This makes it an ideal choice as the basis of the user interface of the tracepoint system.

A modified version of the GNU C debugger, GDB, provides tracepoint functionality when debugging C programs. GDB is a console-based debugger with two command interpreters. The command-line interface, or CLI, is for interacting with a human user. The

machine interface, or MI, is for interacting with a control program such as an IDE. New commands for both interfaces allow the setting and clearing of tracepoints, as well as the specification of serialization options for data. Tracepoint results in the CLI are presented in human-readable C-style syntax, using existing GDB output routines. The MI, on the other hand, outputs memory dumps of expression results. As the GDB terminal protocol is text based, a MIME Base64 buffered encoder is written to serialize this raw information so that it could be sent over the terminal without interfering with the existing protocol.

Tracepoint functionality for Python programs is provided by a modified version of the Python debugger supplied with NetBeans. The debugger is itself a Python program called `jpydaemon.py`. This program is designed to communicate solely with NetBeans over a TCP/IP connection. The communication protocol is based on XML, so tracepoint results in Python are serialized to an XML-based format. Although this causes unnecessary overhead, the use of a different format would require a rewrite of the entire debugger.

A new NetBeans module called `api.tracepoints` holds all the classes necessary to support tracepoint functionality within the IDE. As much of this functionality as possible is included within this module in order to minimize the work required to implement tracepoints for other languages. Modifications to the NetBeans plugins for both the C and Python debuggers (`cnd.debugger.gdb` and `python.debugger` respectively) use the functionality within `api.tracepoints` to permit the use of the modified debuggers within NetBeans. This functionality includes underlying APIs for tracepoint creation and deletion, the abstract representation of data from different languages and classes for propagating and consuming this data. User interface code is also a part of this module, for example generic GUI elements for creating and viewing tracepoints, automatic annotation of code and the basis for providing visualisations of data gathered from tracepoint hits. An external visualisation module containing visualisations for Player/Stage sensor data provides a template for other developers and demonstrates the extensibility of the system.

A series of benchmarking are used to test the performance of the modified debuggers. All benchmarks are based on the repetitive execution and timing of a simple loop containing a square-root operation. GDB benchmarks use the system time for measurement (in order to catch the inherent task switching overhead). The benchmarks are controlled by a GDB command script called `.gdbinit`. This script is automatically executed when GDB starts. Python benchmarks are based on the CPU time of the test process, as both the debugger and the inferior run within the same process. The Python benchmarks use a custom program to control the `jpydaemon.py` debugger via TCP/IP.

The results of the benchmarking process demonstrate that system overheads are predictable and repeatable, with a tracepoint hit taking less than 5 ms even in demanding situations (for example, where the attached expression evaluates to an 8 kilobyte array). Where the serialized result is smaller, on the order of eight bytes, overheads drop to less

than 100 μ s. The high task-switching overheads in OSX are a concern; future work will address the reasons for these overheads. While GDB benchmarking results appear only slightly faster than those of Python, it is important to remember that the GDB results include interference from background processes.

The tracepoint system implemented in this research gives robot developers a method of examining information about the state of a robot control program in the real world without modifying the target source code. The system is compatible with multiple languages, operating systems and robot middleware. To take advantage of the system, only installation on the target OS is required; instructions for doing so are listed in Appendix A. No alterations to existing projects are necessary. Any robot controller written in Python or C can be debugged using tracepoints. The system is thus immediately useful to any robot developer or indeed any developer at all who is working with those languages. An interface to the new functionality has been provided in the form of extensions to the NetBeans IDE. These extensions are themselves extensible if additional functionality is required by the developer. Benchmarking has demonstrated that the system has only a small and quantifiable effect on the behaviour and performance of the program being debugged. The interface was designed for usability, however a full usability study would be needed to confirm this and such a study is beyond the scope of this work. With that one caveat, the tracepoint system meets all the requirements for a useful and practical robotic debugging tool, as identified in Section 3.2.

7.2 Future work

The tracepoint system implemented here provides the minimum functionality required to be useful to a developer. Many extensions are possible which could greatly improve the utility of the base system to a robot developer. In this section, some potential extensions and modifications are proposed as well as some work that would be of benefit to the wider development community.

7.2.1 Usability Study

The system was designed with usability in mind, however in order to verify that it meets this requirement as identified in Section 3.2, a usability study is required. Such a study would also provide valuable feedback about potential improvements to the system.

7.2.2 Conditional Tracepoints

At the moment, a tracepoint is always evaluated when expression passes the point at which it is set. It is conceivable that the user may only wish to trace data in some cases.

This could also improve performance in situations where a large amount of data would be serialised by the trace expression. The only concern is that it adds the evaluation of another expression, which could have performance implications if a conditional tracepoint is placed at a frequently executed location.

7.2.3 Data Analysis

Statistics about the frequency of tracepoint hits could be useful to developers curious about the speed of their code. Tracepoints could potentially be used to gather both performance and coverage information. The ability to graph or visualise these results would also be a significant feature, adding time sensitivity to the system and permitting the developer to analyse peaks or troughs in program throughput or load.

7.2.4 Multiple Expressions

Although at present it is possible to place multiple tracepoints at the same position in the target code, depending on the debugger this could be an inefficient approach. One possible way to get around this problem would be to allow the specification of multiple expressions for a single tracepoint. When the parent tracepoint is hit, all the attached expressions would be evaluated.

7.2.5 Configurable Update Rate

In order to further decrease overhead, a system could be implemented whereby the user can set a maximum frequency for tracepoint hits. When a tracepoint is hit, further hits could be ignored until a minimum time period has elapsed. This could be reasonable in many cases as one use of tracepoints is to supply a display of program events to a human being. In that case there would be little point in updating the display at more than 10Hz-20Hz. As some code locations can be executed many thousands of times per second, this would represent a significant decrease in system overheads, particularly serialization.

7.2.6 Visualisations

The visualisations created in this work are proof-of-concept, intended to demonstrate the capabilities of the framework. In order to improve the utility of the system, a library of visualisations could be built up to cover not only robot-specific data such as laser or ultrasonic range information, but basic types. For example, a floating-point value could be analysed statistically or graphed over time. Boolean flag values could be represented as white or black squares on a grid.

7.2.7 External Tools

The information provided by tracepoints is not necessarily only useful within the IDE. The system could be used to gather data for an external system status monitor, for example. GDB can be used to attach to a running process, and as long as that process was compiled with debugging information, a tracepoint could gather arbitrary information from anywhere within the code. In another example, rather than adding logging capabilities to system source code, a number of code locations and expressions could be specified in a tracepoint logging tool.

7.2.8 Merging Open-Source Work

This work has resulted in a number of bugfixes and improvements both to GDB and to core NetBeans modules (most notably the `python.debugger` module). These improvements should be merged into the GDB and NetBeans projects in order to ensure that tracepoints become a standard on which other open-source developers can build and improve. Some barriers exist to this, at present — the new functionality is not yet fully documented in the source, and some code may not comply with the style guidelines laid down by project maintainers.



Deployment

A.1 General requirements

This section details the requirements for installation that apply across all three supported operating systems. Any extra OS-specific requirements are given before installation instructions in each of the relevant OS sections.

A.1.1 NetBeans

All deployment types require the pre-packaged NetBeans platform. This platform is based on NetBeans release 6.8, milestone 1. It has been compiled to include the basic Player visualisations, as well as C and Python development plugins. Further plugins can be installed once the system is up and running. The pre-packaged installation can be downloaded from the University of Auckland Robotics Research Group website, <http://robotics.ece.auckland.ac.nz>.

A.1.2 Java

As NetBeans is a java-based application, a Java Virtual Machine is necessary to execute it. Any version of Java above 1.5.0 is acceptable, although the use of 1.6.0 is recommended. The JVM is available from <http://java.com>. If the ability to work with the NetBeans sources is required, the Java SE Development Kit for Java 1.5.0 is necessary. This can

be downloaded from http://java.sun.com/javase/downloads/index_jdk5.jsp. Although compilation with JDK 6 is possible, NetBeans does not recommend it. To compile, Apache Ant 1.7.1 is required, available from <http://ant.apache.org>. Optionally, the Mercurial distributed version control system integrates with NetBeans. Mercurial can be downloaded at <http://mercurial.selenic.com/>.

A.1.3 C/C++

In order to use tracepoints with C, The modified GDB sources must be downloaded and compiled. They are based on the 7.0 version of GDB, and are also available for download from the University of Auckland Robotics Research Group website, <http://robotics.ece.auckland.ac.nz>. Compilation was tested with GNU Make 3.81 and GNU GCC 4.3.3.

A.1.4 Python

To use tracepoints in Python, only an installation of Python is required. Python can be downloaded from <http://www.python.org>. Only version 2.6 of the interpreter has been tested with the system, but the use of other versions is theoretically possible.

A.2 Linux

These installation instructions were compiled and tested with Ubuntu Jaunty Jackalope on an Intel-based 13" Macbook. No additional software is required beyond the basic Ubuntu installation.

A.2.1 Instructions

1. Open a terminal
2. Install GCC and make: `sudo apt-get install gcc make`
3. Untar the GDB source tree: `tar -xjf gdb.tar.bz2 ~/`
4. Change to the top directory of the tree: `cd ~/gdb-7.0`
5. Configure the source tree for build: `./configure`
6. Rename the existing GDB binary:
`sudo mv /usr/bin/gdb /usr/bin/gdb_old`

7. Create a soft link to the new GDB binary:

```
sudo ln -s ./gdb /usr/bin/gdb
```

8. Unzip the NetBeans package: `unzip NetBeans-6.8m1-trace.zip ~/`

9. Start NetBeans: `~/NetBeans-6.8m1-trace/bin/netbeans`

A.3 Mac OS X

These installation instructions were compiled and tested with MacOSX Leopard on an Intel-based 13" Macbook.

A.3.1 Requirements

In order to compile GDB, the XCode developer tools must be installed. These tools are available for download from <http://developer.apple.com/mac/>. Download requires a free Apple Developer Connection registration. Ensure the correct version is downloaded; 3.2.1 for OS X 10.6 Snow Leopard, 3.1.4 for OS X 10.5 Leopard or 2.5 for OS X 10.4 Tiger.

A.3.2 Instructions

1. Install XCode, ensuring that the “Unix Command-Line Tools” option is selected.
2. Open a terminal
3. Untar the GDB source tree: `tar -xjf gdb.tar.bz2 ~/`
4. Change to the source directory: `cd ~/gdb-7.0`
5. Configure the source tree for build: `./configure --disable-werror`
6. Build the modified GDB: `make`
7. Change to the “gdb” subdirectory: `cd gdb`
8. Set “setgid” permissions bit: `sudo chmod u+s ./gdb`
9. Set GDB group to procmod: `sudo chgrp procmod ./gdb`
10. Rename the existing GDB binary:
`sudo mv /usr/bin/gdb /usr/bin/gdb_old`
11. Create a soft link to the new GDB binary:
`sudo ln -s ./gdb /usr/bin/gdb`

12. Unzip the NetBeans package: `unzip NetBeans-6.8ml-trace.zip ~/`
13. Start NetBeans: `~/NetBeans-6.8ml-trace/bin/netbeans`

A.4 Windows Vista

These installation instructions were compiled and tested with Windows Vista SP2 on an Intel-based 13" Macbook. Vista has the most involved installation process; the instructions must be carefully followed. These instructions use MinGW for cross-compilation; Although using Cygwin is theoretically possible, it was not tested and is not recommended.

A.4.1 Requirements

In order to compile GDB with tracepoints, MinGW and MSYS are required. MinGW is the Minimalist GNU for Windows. It is available from <http://www.mingw.org>. The system includes versions of the GNU build tools that have been ported to produce Windows executable files. MSYS is a minimal Bourne shell implementation for Windows, designed to work with MinGW and assist in the cross-compilation of programs originally written for Linux. A number of files must be downloaded from the MinGW SourceForge page at <http://sourceforge.net/projects/mingw/files/>. A list of these files is given here:

- Automated MinGW Installer 5.1.6: `MinGW-5.1.6.exe`
- MSYS Base System 1.0.11: `MSYS-1.0.11.exe`
- MSYS Base System (Core): `msysCORE-1.0.11-bin.tar.gz`
- MSYS Supplementary Tools (DTK 1.0.1): `msysDTK-1.0.1.exe`
- GCC Version 4: `gcc-full-4.4.0-mingw32-bin-2.tar.lzma`

7-Zip is an open-source archive manager recommended for dealing with `.tar`, `.bz2`, `.gz` and `.lzma` files. It is available for download from <http://www.7-zip.org>.

A.4.2 Instructions

At all points in these instructions you should choose the default options, including installation directories.

1. Install MinGW (`MinGW-5.1.6.exe`)
2. Install MSYS (`MSYS-1.0.11.exe`)

3. Install MSYS supplementary tools (`msysDTK-1.0.1.exe`)
4. Open the MSYS Core archive with 7-Zip (`msysCORE-1.0.11-bin.tar.gz`)
5. Extract the archive to `C:\MinGW\1.0`
6. Open the GCC archive with 7-Zip
(`gcc-full-4.4.0-mingw32-bin-2.tar.lzma`)
7. Extract the archive to `C:\MinGW`
8. Extract the modified GDB source to a convenient location
9. Run an MSYS shell from the start menu shortcut
10. Change to the directory where GDB was extracted (e.g. `cd /C/Projects/gdb`)
11. Configure GDB for compilation: `./configure`
12. Make GDB: `make`
13. Copy the `gdb.exe` file from the `gdb` subdirectory (e.g.
`C:\Projects\gdb\gdb\gdb.exe`)
14. Paste the new `gdb.exe` into the `C:\MinGW\bin` directory
15. Unzip the NetBeans package to a convenient location
16. Execute `netbeans.exe` from the `bin` subdirectory

B

Benchmark Results

Test	OSX CLI	OSX MI	MSYS CLI	MSYS MI	Vista CLI	Vista MI	Ubuntu CLI	Ubuntu MI
raw	0.004	0.004	0.021	0.022	0.021	0.021	0.025	0.024
zero	2793	3022	747	360	778	318	88	76
c1	3034	2690	814	363	833	355	97	88
c2	2514	2788	874	350	961	336	105	87
c3	3023	3023	1087	364	1011	363	106	82
c4	3020	2705	1171	363	1205	297	107	87
c5	3039	3021	1330	338	1325	363	114	89
c6	3049	2680	1407	334	1414	315	109	85
c7	2678	3023	1504	364	1564	300	105	79
c8	3063	3014	1687	362	1663	312	110	89
c16	3006	2962	1600	320	1577	363	120	85
c32	3068	3308	1587	355	1597	363	107	87
c64	3080	2964	1602	364	1496	363	123	94
c128	3103	3025	1552	326	1048	364	128	95
c256	3098	3010	1677	308	1656	364	126	95
c512	3072	3029	1677	369	1634	335	162	89
c1024	3256	3018	1781	374	1777	361	198	107
c2048	3407	2693	1813	342	1718	380	288	103
c4096	3633	3065	1527	369	1957	396	448	113
c8192	4772	3007	1991	395	1513	405	839	117
c8192x2	6246	3249	4178	654	4172	626	1596	161
c8192x3	8385	3191	6115	868	6162	678	2318	236
c8192x4	9391	3035	7847	1104	8065	1102	2730	262
c8192x5	11374	3119	9532	1236	7940	1245	3683	320

Table B.1: Raw GDB Benchmark results (in μ s, part 1).

Test	OSX CLI	OSX MI	MSYS CLI	MSYS MI	Vista CLI	Vista MI	Ubuntu CLI	Ubuntu MI
sm10	3177	2780	1566	265	1502	371	166	94
sm20	3094	3017	2119	297	2119	358	191	99
sm30	3121	2697	2686	371	2677	327	263	90
sm40	3537	3022	3225	372	3241	371	299	98
sm50	3833	2681	3809	340	3887	370	355	97
sm60	3817	3018	4714	372	4696	371	401	86
sm70	3710	3016	5569	371	5507	350	419	98
sm80	4102	3020	5632	372	5647	368	458	98
sm90	4223	3305	6334	372	6224	359	535	98
sm100	4351	2968	6568	342	6630	370	582	98
bsm10	4561	3151	7956	927	7535	892	599	338
bsm20	3084	3050	2137	549	2090	549	193	137
bsm30	3385	3157	2676	591	2682	588	279	169
bsm40	3535	3167	3246	463	3240	500	287	166
bsm50	3680	3205	3923	540	3733	513	364	185
bsm60	3813	3178	4731	573	4681	549	402	229
bsm70	3709	3165	5522	589	5398	593	415	230
bsm80	4120	3193	5694	631	5663	636	448	260
bsm90	4227	3214	6334	676	6256	707	519	262
bsm100	4341	3100	6692	707	6630	710	575	290
bufPtr	4067	3012	10358	621	9734	619	621	104
strPtr	3143	3036	2938	421	2989	454	150	99
sbufPtr1a	3877	3009	10109	456	9875	456	288	93
sbufPtr2a	3792	3007	10078	551	10109	549	320	112
sbufPtr3a	3878	3030	10062	641	9766	632	321	115
sbufPtr1b	3796	3016	10078	456	10093	376	318	102
sbufPtr2b	3869	3233	9391	493	7956	546	317	109
sbufPtr3b	3788	3081	10093	637	10093	637	351	116
sstrPtr1	3886	3053	10093	455	9734	459	293	103
sstrPtr2	3779	3261	10015	548	10062	540	297	111
sstrPtr3	3782	3016	10124	641	10109	641	320	105

Table B.2: Raw GDB Benchmark results (in μ s, part 2).

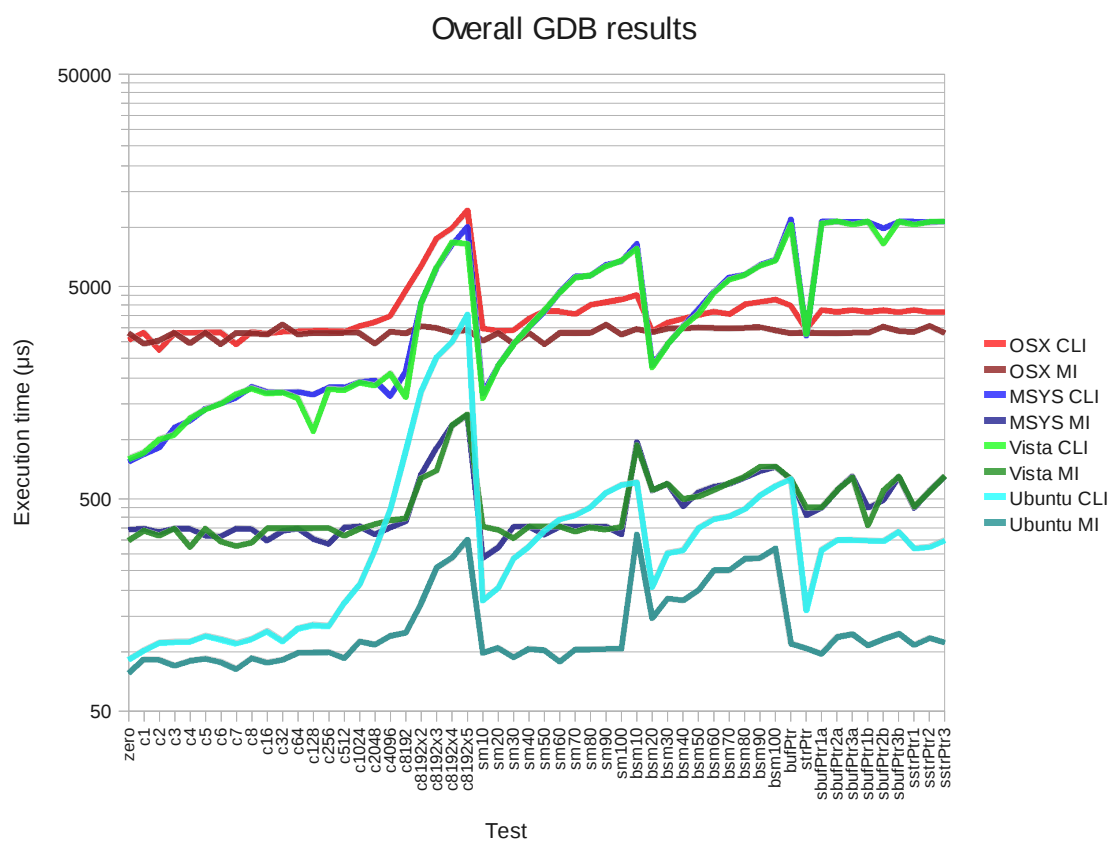


Figure B.1: Overall GDB benchmark results

Test	Ubuntu	OSX	Vista
run	0.37	0.52	0.27
debug	0.40	0.52	0.26
c1	232	230	197
c2	231	233	196
c4	232	232	200
c8	231	233	197
c16	234	238	199
c32	238	247	203
c64	246	262	210
c128	249	285	223
c256	278	335	242
c512	333	431	300
c1024	442	628	382
c2048	662	1029	553
c4096	1103	2082	945
c8192	1978	3897	1680
str1	224	223	194
str2	335	358	297
str3	455	492	403
str4	577	627	510
str5	696	761	616

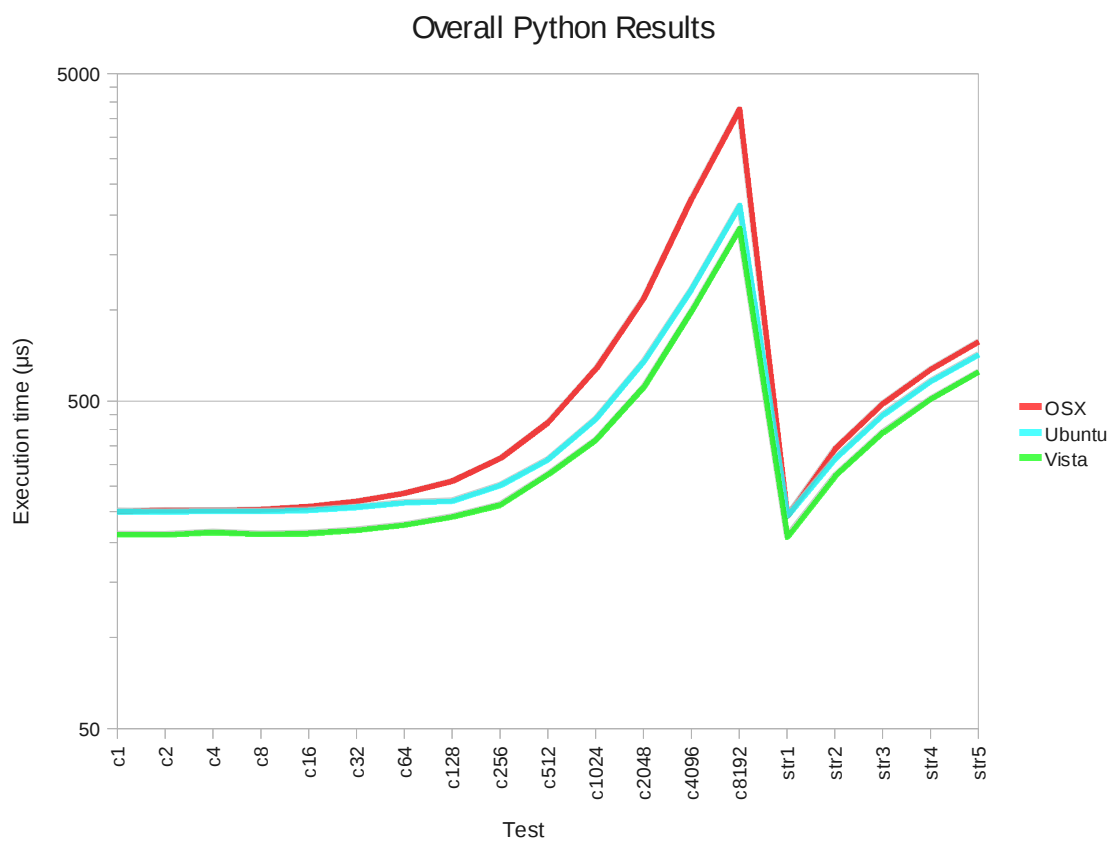
Table B.3: Raw Python Benchmark results (in μs).

Figure B.2: Overall Python benchmark results

Bibliography

- [1] The economic impacts of inadequate infrastructure for software testing. Technical report, U.S. National Institute of Science and Technology, 2002.
- [2] N. Ando, T. Suehiro, K. Kitagaki, T. Kotoku, and Woo-Keun Yoon. Rt-middleware: Distributed component middleware for rt (robot technology). In *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, page 3555, Aug. 2005.
- [3] Noriaki Ando, Takashi Suehiro, Kosei Kitagaki, and Tetsuo Kotoku. Rt(robot technology)-component and its standardization - towards component based networked robot systems development. In *2006 SICE-ICASE International Joint Conference*, page 2633, 2006.
- [4] K. Araki, Z. Furukawa, and J. Cheng. A general framework for debugging. *IEEE Software*, 8(3):14, 1991.
- [5] E. Broadbent, B. MacDonald, L. Jago, M. Juergens, and O. Mazharullah. Human reactions to good and bad robots. In *Proceedings of the 2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3703–3708, San Diego, California, 2007.
- [6] A. Brooks, T. Kaupp, A. Makarenko, S. Williams, and A. Oreback. Towards component-based robotics. In *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems, 2005*, pages 163–168, Aug. 2005.
- [7] Alex Brooks, Tobias Kaupp, Alexei Makarenko, Stefan Williams, and Anders Orebäck. Orca: A component model and repository. 30:231, 2007.
- [8] P. Burgess, M.J. Livesey, and C. Allison. Debugging and dynamic modification of embedded systems. In *Proceedings of HICSS-29: 29th Hawaii International Conference on System Sciences*, volume 1, page 489, 1996.
- [9] F. Calzolari, P. Tonella, and G. Antoniol. Dynamic model for maintenance and testing effort. In *Proceedings. International Conference on Software Maintenance*, page 104, 1998.

- [10] Margaret Chan. *The World Health Report 2008: Primary Health Care Now More Than Ever*. World Health Organization, Geneva, 2008.
- [11] W.H. Cheung, J.P. Black, and E. Manning. A framework for distributed debugging. *IEEE Software*, 7(1):106, 1990.
- [12] R. Chmiel and M.C. Loui. An integrated approach to instruction in debugging computer programs. In *33rd Annual Frontiers in Education, 2003. FIE 2003.*, volume 3, 2003.
- [13] Brian Cole, Daniel Hakim, David Hovemeyer, Reuven Lazarus, William Pugh, and Kristin Stephens. Improving your software using static analysis to find bugs. In *Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications - OOPSLA '06*, page 673, 2006.
- [14] T. H. J. Collett and B. A. MacDonald. Developer oriented visualisation of a robot program. In *Proceeding of the 1st ACM SIGCHI/SIGART conference on Human-robot interaction - HRI '06*, page 49, 2006.
- [15] T.H.J. Collett and B.A. MacDonald. Augmented reality visualisation for player. In *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.*, page 3954, 2006.
- [16] Computing Community Consortium. From internet to robotics. <http://www.us-robotics.us>, May 2009.
- [17] R Crawford, R Olsson, W. Wilson Ho, and C. Wee. Semantic issues in the design of languages for debugging. *Computer Languages*, 21(1):17, 1995.
- [18] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. Lightweight bug localization with ample. In *Proceedings of the Sixth sixth international symposium on Automated analysis-driven debugging - AADEBUG'05*, page 99, 2005.
- [19] Bjorn De Sutter, Bruno De Bus, Michiel Ronsse, and Koen De Bosschere. Backtracking and dynamic patching for free. In *Proceedings of the Sixth sixth international symposium on Automated analysis-driven debugging - AADEBUG'05*, page 83, 2005.
- [20] S. Dubowsky, F. Genot, S. Godding, H. Kozono, A. Skwersky, Haoyong Yu, and Long Shen Yu. Pamm - a robotic aid to the elderly for mobility assistance and monitoring: a “helping-hand” for the elderly. In *Robotics and Automation, 2000. Proceedings. ICRA '00. IEEE International Conference on*, volume 1, pages 570–576, 2000.

- [21] B. P. Gerkey, R. T. Vaughan, K. Stoy, A. Howard, G. S. Sukhatme, and M. J. Mataric. Most valuable player: a robot device server for distributed control. In *Proceedings of the 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 3, pages 1226–1231, 2001.
- [22] S. Gill. The diagnosis of mistakes in programmes on the EDSAC. *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences*, 206(1087):538–554, 1951.
- [23] Luke Gumbley and Bruce MacDonald. Development of an integrated robotic programming environment. In *Proceedings of the 2005 Australasian Conference on Robotics and Automation*, Sydney, Australia, 2005.
- [24] Luke F. Gumbley and Bruce A. MacDonald. Realtime debugging for robotics software. In *Proceedings of the 2009 Australasian Conference on Robotics and Automation*, Sydney, Australia, 2009.
- [25] M. Henning. A new approach to object-oriented middleware. *IEEE Internet Computing*, 8(1):66, Jan-Feb 2004.
- [26] Alex Ho and Steven Hand. On the design of a pervasive debugger. In *Proceedings of the Sixth sixth international symposium on Automated analysis-driven debugging - AADEBUG'05*, page 117, 2005.
- [27] David Hovemeyer and William Pugh. Finding bugs is easy. *ACM SIGPLAN Notices*, 39(12):92, 2004.
- [28] iRobot. About our robots. <http://www.irobot.com/sp.cfm?pageid=74>, January 2010.
- [29] Andrew J. Ko and Brad A. Myers. Designing the whyline. In *Proceedings of the 2004 conference on Human factors in computing systems - CHI '04*, page 151, 2004.
- [30] T. Kooijmans, T. Kanda, C. Bartneck, H. Ishiguro, and N. Hagita. Accelerating robot development through integral analysis of human-robot interaction. *IEEE Transactions on Robotics*, 23(5):1001, 2007.
- [31] D. Kortenkamp, R. Simmons, T. Milam, and J.L. Fernandez. A suite of tools for debugging distributed autonomous systems. In *Proceedings 2002 IEEE International Conference on Robotics and Automation (Cat. No.02CH37292)*, volume 1, page 169, 2002.

- [32] Alex Kozlov, Bruce MacDonald, and Burkhard Wünsche. Covariance visualisations for simultaneous localisation and mapping. In *Proceedings of the 2009 Australasian Conference on Robotics and Automation*, Sydney, Australia, 2009.
- [33] James Kramer and Matthias Scheutz. Development environments for autonomous mobile robots: A survey. *Autonomous Robots*, 22(2):101, 2007.
- [34] Tony Kuo, Elizabeth Broadbent, and Bruce MacDonald. Designing a robotic assistant for healthcare applications. In *Proceedings of the Health Informatics New Zealand Conference and Exhibition 2008*, Rotorua, New Zealand, 2008.
- [35] Donglin Liang and Kai Xu. Debugging object-oriented programs with behavior views. In *Proceedings of the Sixth sixth international symposium on Automated analysis-driven debugging - AADEBUG'05*, page 133, 2005.
- [36] Chao Liu, Zeng Lian, and Jiawei Han. How bayesians debug. In *Sixth International Conference on Data Mining (ICDM'06)*, page 382, 2006.
- [37] B. A. MacDonald, W. H. Abdulla, E. Broadbent, M. J. Connolly, K. J. Day, N. M. Kerse, M. J. Neve, J. Warren, and C. I. Watson. Robot assistant for care of older people. In *Proceedings of the 5th International Conference on Ubiquitous Robots and Ambient Intelligence*, Seoul, South Korea, 2008.
- [38] M. Matsushita, M. Teraguchi, and K. Inoue. Effective testing and debugging methods and its supporting system with program deltas. In *Proceedings International Symposium on Principles of Software Evolution*, page 282, 2000.
- [39] B. T. Moores and B. A. MacDonald. A dynamics simulation architecture for robotic systems. *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*, pages 4532–4537, April 2005.
- [40] Laurie Murphy, Gary Lewandowski, Renée McCauley, Beth Simon, Lynda Thomas, and Carol Zander. Debugging. In *Proceedings of the 39th SIGCSE technical symposium on Computer science education - SIGCSE '08*, page 163, 2008.
- [41] Graeme Nelmes. Container port automation. *Field and Service Robotics*, 25:3, 2006.
- [42] Borislav Nikolik. Convergence debugging. In *Proceedings of the Sixth sixth international symposium on Automated analysis-driven debugging - AADEBUG'05*, page 89, 2005.
- [43] World Health Organisation. Ageing. <http://www.who.int/topics/ageing/en/>, January 2010.

- [44] Thomas Osterlie and Alf Inge Wang. Debugging integrated systems: An ethnographic study of debugging practice. In *2007 IEEE International Conference on Software Maintenance*, page 305, 2007.
- [45] Jaydeep H Palep. Robotic assisted minimally invasive surgery. *Journal of Minimal Access Surgery*, 5(1):1–7, 2009.
- [46] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Y. Ng. Ros: an open-source robot operating system. In *2009 IEEE Workshop on Open Source Software in Robotics*, 2009.
- [47] Steven P. Reiss. Specifying and checking component usage. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging - AADEBUG'05*, page 13, 2005.
- [48] Benjamin D. Rister, Jason Campbell, Padmanabhan Pillai, and Todd C. Mowry. Integrated debugging of large modular robot ensembles. In *Proceedings 2007 IEEE International Conference on Robotics and Automation*, page 2227, 2007.
- [49] Frank Rogin, Erhard Fehlaue, Christian Haufe, and Sebastian Ohnewald. Debug patterns for efficient high-level SystemC debugging. In *2007 IEEE Design and Diagnostics of Electronic Circuits and Systems*, page 1, 2007.
- [50] Frank Rogin, Erhard Fehlaue, Steffen Rülke, Sebastian Ohnewald, and Thomas Berndt. Nonintrusive high-level SystemC debugging. page 131, 2007.
- [51] P. Romero, B. du Boulay, R. Lutz, and R. Cox. The effects of graphical and textual visualisations in multi-representational debugging environments. In *Proceedings of the 2003 IEEE Symposium on Human Centric Computing Languages and Environments*, page 236, 2003.
- [52] Yasushi Saito. Jockey. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging - AADEBUG'05*, page 69, 2005.
- [53] Michael Snyder and Jim Blandy. The heisenberg debugging technology. In *Proceedings of the 1999 Embedded Systems Conference*, San Jose, California, 1999.
- [54] M. Spenko, H. Yu, and S. Dubowsky. Robotic personal aids for mobility and monitoring for the elderly. *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, 14(3):344, 2006.
- [55] Chad D. Sterling and Ronald A. Olsson. Automated bug isolation via program chipping. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging - AADEBUG'05*, page 23, 2005.

- [56] H. Thane and H. Hansson. Using deterministic replay for debugging of distributed real-time systems. In *Proceedings 12th Euromicro Conference on Real-Time Systems. Euromicro RTS 2000*, page 265, 2000.
- [57] H. Thane, D. Sundmark, J. Huselius, and A. Pettersson. Replay debugging of real-time systems using time machines. In *Proceedings International Parallel and Distributed Processing Symposium*, page 8, 2003.
- [58] R. T. Vaughan, B. P. Gerkey, and A. Howard. On device abstractions for portable, reusable robot code. In *Proceedings of the 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 3, pages 2421–2427, Oct. 2003.
- [59] Iris Vessey. Expertise in debugging computer programs: An analysis of the content of verbal protocols. *IEEE Transactions on Systems Man and Cybernetics*, 16(5):621, 1986.
- [60] Mark Weiser. Program slicing. In *Proceedings of the 5th International conference on Software engineering*, pages 439–449. IEEE Press, 1981.
- [61] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446, 1982.
- [62] W Wong, T Sugeta, Y Qi, and J Maldonado. Smart debugging software architectural design in SDL. *Journal of Systems and Software*, 76(1):15, 2005.
- [63] WowWee. Company overview and history. <http://www.wowwee.com/en/company/overview-history>, January 2010.
- [64] Shaochun Xu and V. Rajlich. Cognitive process during program debugging. In *Proceedings of the Third IEEE International Conference on Cognitive Informatics, 2004.*, page 176, 2004.
- [65] Byung-Do Yoon and O.N. Garcia. Cognitive activities and support in debugging. In *Proceedings Fourth Annual Symposium on Human Interaction with Complex Systems*, page 160, 1998.
- [66] A. Zeller. Automated debugging: are we close? *Computer*, 34(11):26, 2001.
- [67] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, 2006.
- [68] Andreas Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the tenth ACM SIGSOFT symposium on Foundations of software engineering - SIGSOFT '02/FSE 10*, page 1, 2002.

-
- [69] Xiangyu Zhang, Haifeng He, Neelam Gupta, and Rajiv Gupta. Experimental evaluation of using dynamic slices for fault location. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging - AADEBUG'05*, page 33, 2005.