Australasian Conference on Robotics and Automation (ACRA), December 2-4, 2009, Sydney, Australia

Realtime Debugging for Robotics Software

Luke Gumbley and Bruce A. MacDonald
University of Auckland, New Zealand
luke.gumbley@gmail.com, b.macdonald@auckland.ac.nz

Abstract

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. A previously theorized debugging construct called a *tracepoint* has been implemented within both a C and a Python debugger. The NetBeans IDE was modified to provide an extensible user interface. A plugin-based visualisation system for rendering trace data has also been implemented. Presently, plugins for the visualisation system have been created for rendering laser and ultrasonic rangefinder data from the Player robot library. Benchmark tests show that although there is still significant room for improvement, in one typical use case the system adds less than 1% overhead.

1 Introduction

A defining characteristic of robotics is a connection to the real world. The development of robotic systems reflects this; considerable effort is spent on real-world issues such as battery life, actuator strength, and sensor reliability. However, the tools used are often derived from normal software development. This treats a robot as merely a computer with interesting peripherals. Consider the utility of standard debugging constructs in a robotic context. A *breakpoint* is a useful way to halt software 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" [13]. Unless the fault has a single cause, visible in the program state at the point of failure, the fault must be replicated to gather enough information for a diagnosis. For computer software, operating in a 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 for the robot to be in a real environment, including pedestrians and vehicles. While the robot is operating it is not possible to halt the controller as this is an unacceptable risk. If the car slows to stop before the controller is interrupted, not only is this a hazard but it prevents the state of the controller being examined until 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 in front of the car, or a vehicle may be attempting to pass. This unexpected sudden shift could cause unpredictable behaviour not relevant to the target fault.

The result of these shortcomings is that alternative robot-aware tools (e.g. simulators, loggers and visualisers) must be used in conjunction with the standard debugger — or the developer must create tools to view the state of the robot while it continues to operate. However, in that case 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 allows discrepancies between the states of the robot and the program. If a simulator is used then there will also be discrepancies between real and simulated behaviour.

An attempt to resolve problems with visualisation discrepancies was made by the "Robotic IDE" project [6]. A proxy server monitored communications between the program code and the robot hardware. Visualisations of monitored robot sensor information were displayed to the user and were an accurate representation of the robot

state. Significant configuration was required. There was also a hard-coded interpreter for the underlying message format, which required ongoing maintenance. Finally, the system could only show information passed along the connection to the robot; internal program logic remained opaque. A more comprehensive and robust solution to the problem of analysing the internal state of a robot is required.

2 State of the art

Yoon and Garcia evaluated the debugging process and suggested a *watchpoint* aid [17]. *Watching* is defined as:

> "Isolating specific variables and keeping track of their changing values as the program runs."

Cheung and Black list "Tracing" as one of seven fundamental debugging techniques [3], and define it 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."

Despite the literature supporting this technique, actual implementations are not widely available. Programmers often simulate the functionality using *Output Debugging*; the insertion of statements into the program code to generate output, which is then analysed to find the source of a fault [3]. This conclusion is supported by a study of 21 novice debuggers; most used "printf" statements rather than the available debugger [11].

Crawford *et al.* claim that the lack of new debugging tools is caused by a focus on the design of interfaces to existing techniques rather than the expansion of the underlying debugging languages [4].

Pop and Fritzon [12] created a debugger for RML, including logging and replay facilities. User code is automatically instrumented within a re-written RML compiler. A data browser application is capable of running complicated post-mortem analyses on logged data, and can move arbitrarily forwards and backwards in time through program execution. No mention is made of issues with concurrency or non-determinism.

A great deal of work addresses debugging of distributed real-time systems. Kortenkamp *et al.* [9] develop an approach for deterministic logging and analysis of distributed systems. Thane and Hansson [15] describe a similar initial theoretical method for deterministic logging but include the ability to replay. Thane *et al.* [16] expand on this approach, describing an improved method including benchmarking results from implementation testing. Burgess *et al.* [2] approach debugging a parallel system by first reducing it to independent event-driven blocks to permit monitoring in real-time, reasoning that an embedded system is always functioning and cannot be debugged cyclically.

The approaches above impose conditions and alterations on system code and must be implemented from the start of development. Additionally, the logging overhead must be considered during hardware design since all require the logging systems to remain present in the final production output so as to avoid the probe effect.

Ho *et al.* [7] explore the design of a "pervasive debugger" which operates at a level below the program being debugged. They suggest simulating distributed systems with a single process at a layer above the debugger, thus providing the potential for deterministic replay.

Jockey is a transparent debug assistant for any Linux application [14]. It is a shared library that instruments any non-deterministic program calls. The return value of the function is stored and the program state checkpointed before control is returned to the target. Overheads are very low, 30% even in extreme cases (where a large amount of I/O is involved) and in many cases are unmeasurable. Practical tests 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 occurs. 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.

Rister *et al.* [13] developed a system for recording and replaying the execution of swarming robot control systems in a simulator. This allowed them to ignore the probe effect. They implemented a comprehensive and easily used system for recording the value history of variables as well as a complete stack trace. 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 permitted 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 such as GDB were "time sensitivity" and "thread sensitivity," i.e. the system was aware of events across a distinct time period (as opposed to a single slice) 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 and is a particular strength of the approach. Disadvantages are the high (90%) overheads involved and the reliance on a simulated space to eliminate the probe effect.

Kooijmans *et al.* [8] detail the debugging during human robot interactions. Robot sensory data is logged and examined to determine trigger events for robot behaviours and human reactions. The problems of recording and presenting multi-modal data simultaneously (i.e. video, rangefinder data, touch sensors) are examined.

Ando *et al.* [1] details a standard for a modular component approach to robotic development, RT-Middleware. Debugging is provided by the "RTCLink" tool, which monitors and logs communications between components.

Moores *et al.* [10] present a robot simulation architecture that enables the user to mark simulated variables for watching or logging, and to attach custom code to be executed upon specified events.

De Sutter *et al.* [5] modified GDB to provide "backtracking" and "dynamic patching." Backtracking enables the user to specify "checkpoints", where the debugged process is halted and a child process forked in which debugging continues. Later the user can return to the earlier checkpoint, 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.

GDB includes a *tracepoint* framework intended for debugging embedded systems. A remote stub of GDB which supports this framework is executed on the debug target and then connected to GDB itself via TCP/IP or serial communications. The main debugger is used to set up a "trace test" consisting of a number of tracepoints set in the code accompanied by *actions* to be executed when hit. Data collected by these actions is stored and retrieved at the conclusion of the test for analysis by the user. Unfortunately there are no remote stubs currently available for GDB that support this framework, although some work has previously been completed.

3 Requirements analysis

Applicability: For a solution to the robot state analysis problem to be useful it must work with a large number of robotic systems. So the system must be cross-platform and have no programming language or robot system specificity.

Real-World Utility: The solution must work in the real world. Many solutions eliminate the probe effect by relying on debugging robots within a simulated world. While this can eliminate some software problems, ultimately a robot designed to operate in the real world must still be debugged and tested in the real world.

Target Software Constraints: The solution must also impose no fundamental changes in existing software or conditions on software being developed. Much of the current work in robot debugging has development driven by debugging needs. By contrast, the majority of developers do not consider debugging requirements when designing and implementing robotic systems.

Usability: Ease-of-use is critical. A developer is unlikely to consider a technique that imposes inconvenient development requirements. The system should take as little developer time to implement and use as possible. Support for the system must be added to a modern IDE.

Accuracy and the Probe Effect: The solution must provide an accurate picture of the internal program state, while having a minimal effect on program execution to minimize the probe effect. It must not change the behaviour of the software being examined or its utility as a debugging tool will be severely compromised.

4 Existing solutions

Source modification: A popular debugging method is to modify the program to output the required data. While straightforward, this solution involves modifying the source code and thus adds significant development overhead. Its primary strength is in ubiquity.

Breakpoints: Information about the internal state of a program can be gained by specifying locations where execution is interrupted and the stack and memory of the program examined. Most IDEs automate these features well. The primary issue is the reaction speed of the user; the target program can remain suspended for seconds or even minutes while the user formulates queries and evaluates the current state.

GDB tracepoints: The current GDB tracepoint framework addresses the issues with breakpoints and user response time very well, but is unable to report results while the target is executing. Instead, a discrete test must be set up and executed. Only once execution is complete may the log can be examined. Additionally there are no available GDBServer stubs that support tracepoints. It has been suggested that this is due to the complexity of the scripting language used to specify logging actions to be taken at tracepoints.

Kortenkamp* et al.*: This work focuses primarily on logging and logical and temporal analysis of log files after the completion of a test run [9]. The data collection is done via a printf substitute, *rlog*. No mention was made of cross-platform testing or suitability for multiple languages.

Thane* et al.*: This work concentrates on record-and-replay style debugging of embedded systems [16]. The system carefully considers the probe effect and gives a number of different methods of data collection. 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

	Cross-platform	Language/system non-specific	Simulator not required	No design constraints	Developer efficiency	IDE integration	State accuracy	Minimal probe effect
Source modification	2	2	0	0	0	2	2	0
Breakpoints	2	2	2	2	2	2	0	0
GDB tracepoints	2	0	2	2	0	2	2	0
Visual Studio	0	2	2	2	2	2	2	0
Robotic IDE	2	0	2	2	2	2	1	2
Kortenkamp *et al.*	1	1	2	1	1	0	2	2
Thane *et al.*	2	1	1	1	2	2	1	2
Saito *et al.*	0	2	2	2	2	0	1	2

Table 1: Analysis of existing solutions. The numbers show the extent to which a requirement has been satisfied. A score of "0" means the requirement was not satisfied, "1" partially satisfied , and "2" fully satisfied.

hooks, and that a debugger supporting scripted breakpoints is available.

Ho* et al.*: Ho *et al.* suggest using *pervasive debugging*, where the environment in which the target executes is virtualized [7]. Thus there is no requirement for efficiency as interruptions to the program being debugged are transparent. While this reduces the probe effect to nil, it also means that robot behaviour in the real world may differ from the simulation.

Jockey: The data-gathering method implemented by Jockey instruments any non-deterministic program calls by replacing system calls with an instrumented stub [14]. This effectively avoids the need to alter the original source code and attendant developer overhead, however it restricts the direct availability of program state information to information exchanged with external libraries. 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.

Summary: An analysis of the suitability of existing tools can be seen in Table 1. This analysis shows that although some solutions come close, none are optimal by the criteria laid out in the previous section. A novel solution is proposed instead, based primarily on the work of Cheung and Black in [3] and Yoon and Garcia in [17].

5 Proposed solution

The proposed solution takes the form of an additional construct for a standard debugger called a *tracepoint*. 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. As tracepoints are conceptually similar to breakpoints, a similar user interface will be added to a suitable IDE.

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. No changes whatsoever to program design are necessary. The only requirement 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.

Any debugger which has the ability to place breakpoints and evaluate expressions can be made to support tracepoints. Thus, conceptually, solution is not specific to any one platform or language. The initial implementation is to be for Linux, but as the main components are all cross-platform, support for other operating systems will be available. Debugging information is gathered from the program itself, thus the robot library used is unimportant. The implementation is expected to be fast enough that a simulator is not required to compensate for the probe effect; benchmark testing will be used to confirm this.

5.1 Languages

In order to demonstrate the applicability of the solution to different programming languages, the initial tracepoint implementation is for two languages, C and Python. C was chosen for its ubiquity and the completeness of the GNU C debugger, GDB. Python was chosen as it is a commonly used higher-level language with significantly different syntax and structure. Both languages also continue to be used within our research group for robotic projects, providing a valuable internal user base for feedback and testing.

5.2 IDE

In order that the work be useful and practical, the tracepoint implementation will be added to the NetBeans IDE, an open-source java based IDE sponsored by Sun Microsystems. The previous "Robotic IDE" project focused on Eclipse [6], however in the intervening time Eclipse has developed a number of disadvantages.

- Development with Eclipse has become cumbersome and difficult to standardise. Debugging constructs in particular have entirely separate and conflicting

implementations for each language. This makes the platform extremely unattractive for further work, as a multiple-language implementation is desired.

- Significant difficulty has been encountered in getting bugfixes and changes developed outside IBM approved for inclusion into the Eclipse source tree.
- The Eclipse plugin architecture has a lazy-loading paradigm that means most functionality must be first defined in XML manifests before actually being implemented in Java, a doubling-up of work that also results in a number of use cases that are impossible to implement.

The NetBeans IDE has since reached maturity and now represents a far more attractive platform for development. The IDE is based on an underlying NetBeans Platform which has emphasised modularity in development. This has resulted in a lower bar for patch acceptance and a generally higher standard of interoperability between modules. There is also considerably less reliance on XML as NetBeans relies more on streamlined dependencies to speed load times (rather than the XML based lazy-loading of Eclipse).

5.3 Framework

By contrast to the previous Robotic IDE work, the system design is inherently compatible with any robot platform supported by Python or C. This is because the debugger operates at a lower level than the target source, where previously data was gathered at a higher level. Testing has focused on programs based around the Player system, which is a commonly used open robotic programming and simulation system. This permitted quick setup and testing. The only aspects of the implementation specific to Player are the visualisations for sensor data - however, the underlying framework could be just as easily used for any system.

5.4 Renderer

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 visualistions can still be created, however the system does not provide a superclass with that functionality at present.

6 Implementation

6.1 System overview

Figure 1 illustrates the overall layout of the tracepoint plugin implementation and use in NetBeans, by contrast

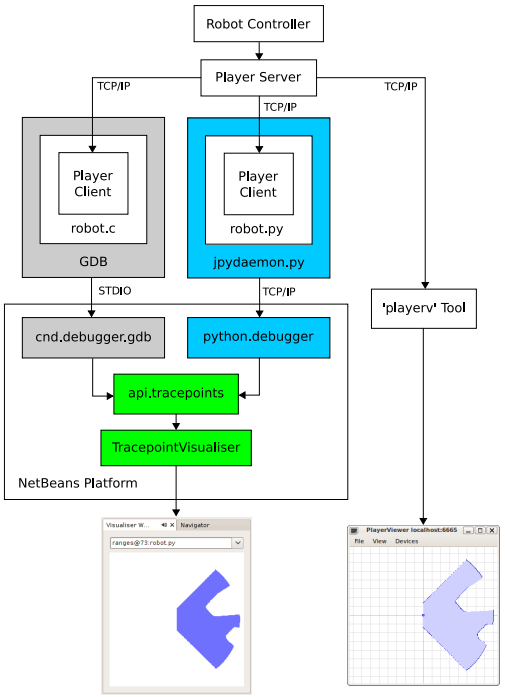

Figure 1: Block diagram of system implementation

with an existing tool (playerv). At the top of the diagram is the robot controller, responsible for controlling the actual robot hardware. It communicates with the Player server, usually running in an onboard computer. The Player server provides a standard abstraction layer over the robot controller, meaning the same software can be used for robots from different manufacturers.

The actual code written by the robot developer can be seen in the diagram as robot.c and robot.py. This is the code to be debugged. Each of these programs links to a Player client library, responsible for communicating to the Player server. The code runs within a debugger, GDB or jpydaemon.py respectively, which is executed and controlled by the user through plugins in the NetBeans IDE (cnd.debugger.gdb and python.debugger). Both the debuggers and the interface plugins for the debuggers have been modified to support the tracepoint construct. A new plugin called “api.tracepoints” has been created to hold the underlying tracepoint API. This contains the functionality common to all client languages. Among other things, these classes provide abstract representations of gathered data, which are consumed by sisualisation plugins (such as the laser visualisation, pictured in figures 1 and 7).

Figure 1 shows that all the information displayed in the visualisation is gathered from the state of the program being debugged. It can clearly be seen that the information shown by the existing playerv tool is gathered independently from the server. This means that,

with playerv, synchronicity with the program state is not guaranteed and fault symptoms such as slow execution, communication faults or dropouts will not be evident.

6.2 Debugger

At present the system works with any robot program written in C or Python executed from the Linux operating system. The debugger is a separate program which can be used either to start the robot program initially, or (in the case of C) to attach to a robot program that is already executing. This work does not currently support the debugging of embedded systems, although such support is possible. GDB in particular permits debugging some supported embedded controllers through custom stubs, but such support was not investigated further in this work.

GDB

The GNU debugger, GDB, supports terminal-based debugging both C and C++ programs. The path to the target executable 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 transparent to the user. Value outputs are similarly human-readable, although GDB has recently added a "machine interface" mode 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 process standard input and output pipes, and only recognises input while the target is not running. As it is possible to silently interrupt the target, the user can still place breakpoints and tracepoints during program execution.

Tracepoints have been implemented in GDB by way of a special breakpoint containing additional information, including an expression to evaluate and additional parameters controlling the serialisation of the result. When any breakpoint is set using GDB, the location of the breakpoint is first translated to a memory location, and then the instruction at that location cached and an operating system interrupt is substituted. When the interrupt occurs, the program halts and GDB is able to examine its state. Ordinarily, the interpreter outputs the breakpoint that was struck and a prompt for further commands. In the case of a tracepoint, not only the tracepoint that was struck but also the result of the expression is output. Target execution is then immediately resumed without performing any further state analysis.

Outputting the result of the tracepoint proved problematic. As GDB was originally intended for direct use by the developer, the command and response syntax is 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. 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 and of the value itself. A custom format for type information was created based on the internal GDB type information and is output the first time the tracepoint is hit. The expression result can then be output as a direct memory dump. In order to output binary data in the textual GDB interface, both the type information and the memory dump are encoded with MIME Base64.

In some cases a direct memory dump may not encompass all the data required by the user. In the case of the `playerc_laser_t` structure of Player, used to store laser rangefinder data, the actual laser ranges are stored as a pointer to a dynamically allocated array. In the memory dump of this structure, only the value of the pointer would be present and not the laser ranges themselves. To resolve this problem, when a tracepoint is being set with an expression that would resolve to a structure, the user has the option of specifying that a given structure member of a pointer type be serialized as an array of the target type. This is done by specifying the name of the pointer member, as well as the name of a member of an integral type that specifies the size of the array. When the expression is evaluated, the values of both members are retrieved and used to perform a secondary memory dump of the target area of memory. Both memory dumps are then presented to the user each time the tracepoint is struck.

Modifications to GDB include the new breakpoint type, new commands in both the regular user interface and the machine interface, and a small library for performing MIME Base64 encoding and specialised printing of type information. These alterations are being submitted for inclusion into the standard GDB release.

Bdb and jpydaemon.py

By contrast to C, which uses an external standalone debugger, Python debugging is performed by a python program making use of certain internal Python hooks and the Python debugging framework, contained within the Bdb class. Both Eclipse and NetBeans supply their own debugging script, which in the case of NetBeans is called jpydaemon.

In order to start a debugging session, jpydaemon is executed with the name of the target 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 rather only operates when the target is halted. This means that tracepoints cannot be placed or edited while the target is running.

The Python debugging hooks permit callbacks to debugger-defined functions at every context change (i.e. function call) and then, if requested, at every line of code in the target script. Breakpoints are implemented by checking each context as it is entered for the presence of a breakpoint. If a breakpoint is present, then the per-line hook is requested. Each time the source line changes, it is checked against an array of breakpoints. When the breakpoint matches, a notification is sent back to the IDE and the debugger awaits further instructions.

Tracepoints were implemented in jpydaemon by implementing a new array within the debugger of tracepoint locations (in addition to the existing array of breakpoint expressions). Each time a context or source 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 a python call that permits the execution of arbitrary python commands. The result is then serialized in to an XML-based format (to match the existing jpydaemon protocol).

Serialisation for Python is very different to that performed for C. 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. Member functions are ignored.

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 the previous section, this is a problem. SWIG serialised 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 serialiser 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.

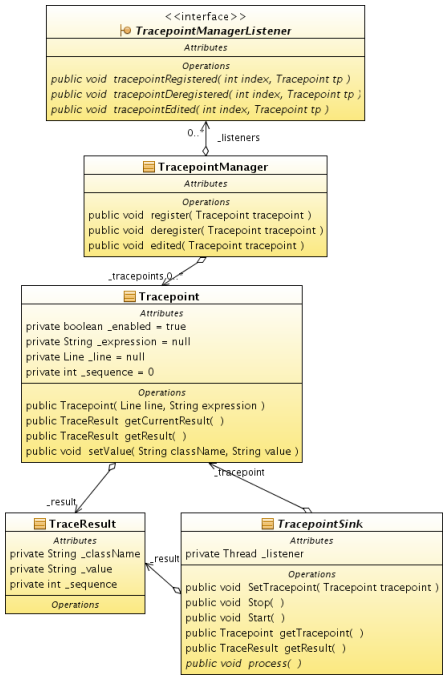

Figure 2: Edited UML of tracepoint class structure

6.3 NetBeans tracepoint API

The "api.tracepoints" module has been added to the NetBeans source tree to contain a general-purpose tracepoint API for use by language-specific implementations. As shown in Figure 2, a generic class *Tracepoint* was implemented which holds the basic data required for the tracepoint — the target file, line number and expression to be evaluated. When a tracepoint is added by the user, the UI creates an instance of this class and registers it with the *TracepointManager*.

Whenever a tracepoint is hit and the result of the evaluation is received by the IDE, the setValue function is called, passing the classname and the serialized contents. The tracepoint uses this information to create a new instance of TraceResult, stored in its "result" member. All threads currently waiting on the *Tracepoint* instance are notified via notifyAll().

Tracepoint consumers (like visualisations) must instantiate a *TracepointSink*, providing a tracepoint to wait on as well as an implementation of the *process()* function, which is called whenever the tracepoint is hit. When the *TracepointSink* is instantiated a monitor thread is created which alternately calls *wait()* on the provided *Tracepoint* instance and the provided *process()* function.

6.4 Tracepoint IDE user interface

The interface for working with tracepoints has been made as similar to the existing interface for Breakpoints

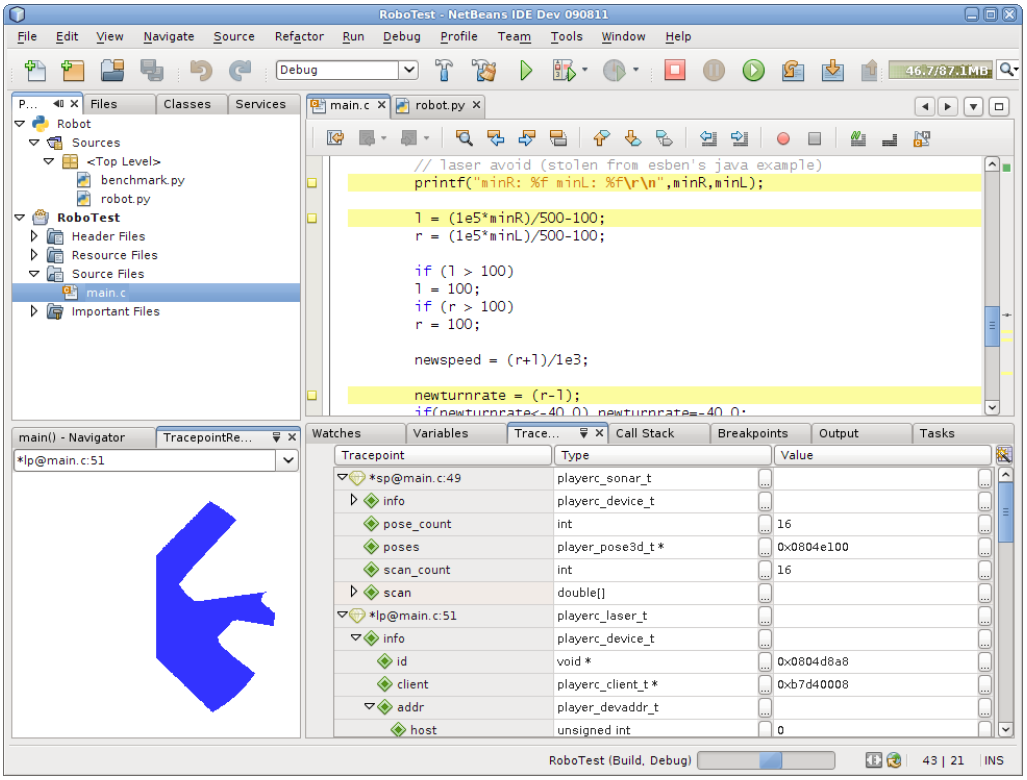

Figure 3: NetBeans during debugging, showing Visualiser, Tracepoint and Tracepoint View

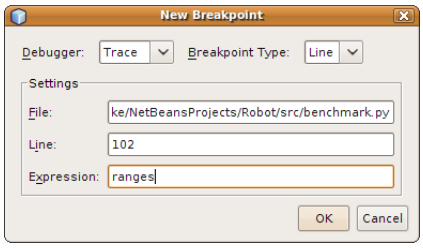

Figure 4: "New Breakpoint" dialog showing user setting a tracepoint

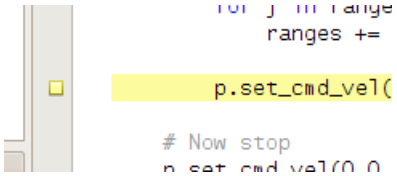

Figure 5: Tracepoint code annotation

as possible. This is in order to provide an experience that users will be familiar with. In practice, the only difference between adding a tracepoint and adding a breakpoint is the necessity of providing an expression to be evaluated when the tracepoint is struck.

Tracepoints are added by selecting the "New Breakpoint..." option from the "Debug" menu when the cursor is on the desired line, and then choosing the "Trace" breakpoint type. The file and line number are automatically filled in and the user then enters an expression to be evaluated (see Fig. 4. Any valid, executable python or C expression may be used. 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). 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.

A "Tracepoint View" has been added to the IDE which displays an entry for every tracepoint that the user has added (see Fig. 6). This view has three columns, one for the tracepoint name, 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.

A second view called a "Tracepoint Render View" has

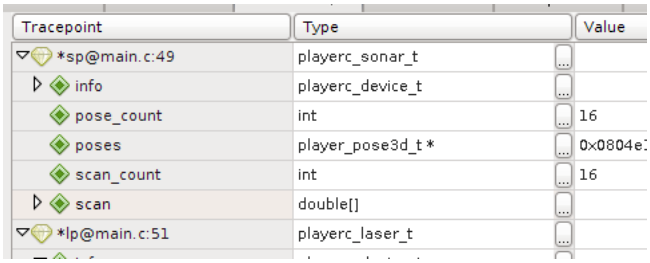

Figure 6: Tracepoint View

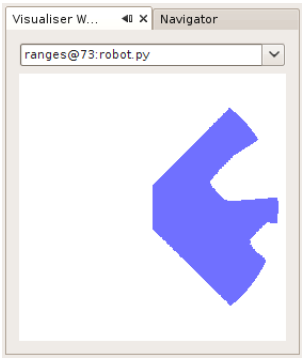

Figure 7: Laser Visualisation

also been added to the IDE (see Fig. 7). This contains a combo box filled with the names of each tracepoint added, as well as an area beneath reserved for the actual visualisation. The user selects the tracepoint they wish to visualise. Whenever this tracepoint is hit, the resulting data will automatically be rendered as appropriate.

When a tracepoint is struck, a search is conducted through the available visualisation services to find one that is capable of rendering the data, given the language and classname. The *TracepointRenderer* resides in a separate thread (courtesy of *TracepointSink*) and "listens" to a tracepoint defined by the dropdown box. In this way, a tracepoint hit and the visualisation of the resulting data are decoupled. The rendering process 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. A sequence number ensures that after a batch of data is received, the Render View is always showing the most recent result.

7 Performance

All debuggers affect the performance of the software being debugged, so the amount of overhead represented by tracepoints is a concern. The current Python implementation of tracepoints has been benchmarked and the results are laid out in Tables 2 and 3. Benchmarking of the C implementation is not yet complete.

In Python, tracepoints incur no overhead for the execution of code outside the context in which the tracepoint is placed (see 6.2) — but within that context, there will be an overhead associated with each line of code executed as well as an overhead when a tracepoint is actually struck.

7.1 Benchmarks

The results of benchmark performance tests are laid out in Tables 2 and 3. Table 2 shows execution with break- and tracepoints set in the benchmark function but outside execution flow. The length of the benchmark function is also altered here to demonstrate the per-line overhead while debugging. Table 3 shows execution with tracepoints set in the benchmark function and hit each time it executes. All times are for one execution of the benchmark code, comprising 1, 2 or 3 lines (boolean "True" expressions) as indicated. Code was executed 10000 times and the average execution time taken.

The benchmark function is shown below:

```
def test:
        True
        True
        True
```

In Python, the expression "True" is executable but does nothing, and so represents a "nop" equivalent. In the function as laid out above, three lines of code are executed. The number of "True" statements was varied in order to determine the tracepoint overhead per line of code. The number of tracepoints was then varied in order to determine the overhead per tracepoint hit.

The "Empty String" and "Laser Scan" columns in Table 3 indicate the type of expression contained within the tracepoints. An empty string was chosen as the baseline benchmark as it would be the fastest to serialize. A laser scan, comprising a list of 360 floating-point numbers, is an anticipated use-case.

The large discrepancy between code that contains no break- or tracepoints can be explained as the difference between the benchmark code being executed instrumenting only context changes, and executed instrumenting every line. This overhead comes from the original debugger code and contains considerable scope for improvement (see 6.2).

The results show that the baseline overhead for including a tracepoint is approximately 25μs per line of code in the target context, with an additional 150μs per tracepoint hit. Serializing and transmitting a typical laser scan requires an additional 500μs.

Per-line overheads are reasonable and comparable to the original debugger. In the standard use-case, the laser scan adds an overhead of 650μs per hit. This means

	Lines	Time
Run	3	0.4
Debug	3	0.47
Tracepoint	1	265
Tracepoint	2	292
Tracepoint	3	316
Breakpoint	3	317

Table 2: Benchmark results with breakpoints and tracepoints missed (μs).

	Lines	Empty String	Laser Scan
1 Tracepoint	3	470	994
2 Tracepoints	3	614	1647
3 Tracepoints	3	722	2337

Table 3: Benchmark results with tracepoints hit (μs).

that if used with a Player client, which updates at 10Hz, the Laser trace consumes 0.65% of the available time for computation.

8 Ongoing/future work

In order to further decrease the overhead, a system will be implemented whereby the user can set a maximum update frequency for tracepoints from the target program. If a tracepoint fires, further hits will be ignored until a minimum time period has elapsed. This is reasonable as the idea of tracepoints is to supply a display of events in the program to a human being. In the case of the benchmark tests, this would mean that instead of reporting 3-5000 hits per second, the debugger may only report 30.

9 Conclusion

Tracepoints are a flexible, low-overhead solution to the problem of monitoring the state of a robot without interrupting its control systems. They require no modification to the target code and are currently compatible with any C or Python program with more languages to be implemented. By implementing tracepoints in a plugin to the open-source, cross-platform IDE NetBeans, they are available to the widest possible range of robot developers. This also means that the plugin itself may be extended by third parties.

References

[1] Noriaki Ando. RT(robot technology)–component and its standardization - towards component based networked robot systems development. In *2006 SICE-ICASE International Joint Conference*, page 2633, 2006.

[2] P. Burgess. Debugging and dynamic modification of embedded systems. In *Proceedings of HICSS-29 29th Hawaii International Conference on System Sciences HICSS-96*, volume 1, page 489, 1996.

[3] W. H. Cheung and J. P. Black. A framework for distributed debugging. *IEEE Software*, 7(1):106, 1990.

[4] R. H. Crawford, R. A. Olsson, W. Wilson Ho, and C. E. Wee. Semantic issues in the design of languages for debugging. *Computer Languages*, 21(1):17, 1995.

[5] Bjorn De Sutter. Backtracking and dynamic patching for free. In *Proceedings of the Sixth sixth international symposium on Automated analysis-driven debugging - AADEBUG 05 AADEBUG 05*, page 83, 2005.

[6] Luke Gumbley and Bruce A. MacDonald. Development of an integrated robotic programming environment. In *Proceedings 2005 Australasian Conference on Robotics and Automation*, 2005.

[7] Alex Ho. On the design of a pervasive debugger. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging - AADEBUG 05*, page 117, 2005.

[8] T. Kooijmans. Accelerating robot development through integral analysis of human-robot interaction. *IEEE Transactions on Robotics*, 23(5):1001, 2007.

[9] D. Kortenkamp. A suite of tools for debugging distributed autonomous systems. In *Proceedings 2002 IEEE International Conference on Robotics and Automation (Cat No 02CH37292) ROBOT-02*, volume 1, page 169, 2002.

[10] 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.

[11] Laurie Murphy, Gary Lewandowski, Rene McCauley, Beth Simon, Lynda Thomas, and Carol Zander. Debugging. In *Proceedings of the 39th SIGCSE technical symposium on Computer science education - SIGCSE 08 SIGCSE 08*, page 163, 2008.

[12] Adrian Pop. Debugging natural semantics specifications. In *Proceedings of the Sixth sixth international symposium on Automated analysis-driven debugging - AADEBUG 05 AADEBUG 05*, page 77, 2005.

[13] 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.

[14] Yasushi Saito. Jockey: A user-space library for record-replay debugging. In *Proceedings of the Sixth sixth international symposium on Automated analysis-driven debugging - AADEBUG 05 AADEBUG 05*, page 69, 2005.

[15] H. Thane. Using deterministic replay for debugging of distributed real-time systems. In *Proceedings 12th Euromicro Conference on Real-Time Systems Euromicro RTS 2000 EMRTS-00*, page 265, 2000.

[16] H. Thane. Replay debugging of real-time systems using time machines. In *Proceedings International Parallel and Distributed Processing Symposium IPDPS-03*, page 8, 2003.

[17] Byung-Do Yoon and Oscar N. Garcia. Cognitive activities and support in debugging. In *Proceedings Fourth Annual Symposium on Human Interaction with Complex Systems HUICS-98*, page 160, 1998.