Development of an Integrated Robotic Programming Environment

Luke Gumbley and Bruce A MacDonald
Department of Electrical and Computer Engineering
University of Auckland, Auckland, New Zealand
lgum002 _at_ ec.auckland.ac.nz, b.macdonald _at_ auckland.ac.nz

Abstract

The paper presents a new Integrated Development Environment (IDE) targeted specifically for use in Robotics programming. The IDE is based on IBM's Eclipse platform and supports the Python scripting language, using the RADAR language extensions developed by PhD student Geoff Biggs. Player and Stage are used as the robot programming tools.

Key issues with existing development processes are identified including the number of different tools that must be managed by the developer, the lack of useful debugging information, and the unsuitability of current debugging methods for coping with real-time systems such as mobile robots.

To resolve these issues, extensions were made to the IDE including a proxy server for gathering debugging data, a tool manager for automating menial tasks including running and monitoring tools, and a number of visualisations to show meaningful debugging information.

The system was tested with two robot programs and shown to streamline the development process considerably. Tests indicate the system is reliable and stable, barring small issues with 3rd party components.

1 Introduction

One of the difficulties facing robotic research and development is the dearth of integrated toolsets for facilitating robot programming. A wide variety of individual tools exist, but the developer must manage them all independently of their actual development environment. The goal of this project is to extend a programming IDE to include integrated robot programming tools in much the same way as most IDEs include an integrated code debugger. The work was undertaken as a final year undergraduate project by Luke Gumbley [2005] and Steve Hsiao [2005].

In the next section we discuss the background to our interest in robot programming environments. Section 3 briefly discusses the requirements and aim. Section 4 analyses current processes for developing and testing robot programs and suggests the improvements needed. Section 5 presents the design choices for a base IDE tool. Section 6 presents the design choices for a robot programming framework. Section 7 presents the extensions made to the base IDE. Section 8 discusses the usability of the resulting extended IDE. Section 9 discusses the reliability of the tool.

2 Background

Robots have become increasingly complex and their controllers increasingly powerful, yet robotic programming tools have not advanced as rapidly [Biggs and MacDonald, 2003, 2005].

Robots must be programmed both at the development stage to create the functionality of the robot, and in the field to customise the robot to applications, environments and tasks. It is important that robots become easier to program so that their potential may be fully realised.

Robot researchers face difficulties developing software systems for robots that are to assist humans in everyday environments. Much of the software is proprietary, there is a lack of open standards to promote collaboration and code reuse, and there is a lack of techniques for bringing the human in to the robot's perceived environment.

The difficulty is the complex interactions robots have in real environments, and the complex sensors and actuators that robots use, including:

- A large number of devices for input, output and storage – far more than the human programmer's senses and effectors, or the few interfaces of a computer.
- Simultaneous and unrelated activity on many inputs and outputs.
- The requirement to operate in real time, in order to interact with real objects.
- Unexpected real world conditions.
- Wide variations in hardware and interfaces, as opposed to the highly standardized desktop computer.

Programmers of robot arms and other complex articulated automatic devices must also deal with non-intuitive geometry. Programmers of mobile robots must deal with varied and unpredictable conditions as the robot moves through its environment.

Standard debugging tools give programmers access only to program data. This makes debugging robot programs difficult because program data is at best an indirect representation of the robot and environment.

We believe robot programming systems must be

tailored more specifically to robotics, paying attention to the varied requirements of robot programs, the typical skills of robot programmers, the interactions between humans and robots, and the predominant programming constructs in robotic applications.

Robot programming systems have three important conceptual components that are of interest to their designers:

- *The programming component,* including designs for programming language/s, libraries and application programming interfaces (APIs), which enable a programmer to describe desired robot behaviour.
- *The underlying infrastructure* including designs for architectures that support and execute robot behaviour descriptions, especially in distributed environments.
- *The design of interactive systems* that allow the human programmer to interact with the programming component, to create, modify and examine programs and system resources, both offline and during execution. The human programmer may also interact with the infrastructure component to examine, monitor and configure resources directly with robots as they perform tasks.

There are other components that are not of particular concern to designers of robot programming systems, such as the robots themselves, operating systems, compilers, robot hardware drivers and so on. A few aspects, such as real time operating system performance, will be of concern.

In this paper we describe the development of an IDE to help robot programmers interact with the programming component of a robotic system.

2.1 Related Work

Although the majority of the existing work on robotic programming is focused on the problems surrounding industrial robotic arms, many issues have been identified that are similar to the ones we raise.

Tomas Lozano-Perez [1983] identified sensing, world modeling, flow of control and programming support as key requirements of a robot programming system. His work came to many of the same conclusions as this paper, most significantly that the complexity of robot programs and systems make them very difficult to debug using traditional methods.

PROGRESS is a graphical robot programming system developed by Naylor *et al* [1987], which approaches the problem by converting many textual programming methods to graphic form, and combining them with a graphical simulator. This is described as integrating 'geometric and logic' viewpoints, a process that previously took place within the mind of the programmer. This integrated environment is designed around robotic arms but demonstrates the necessity of bringing code and reality as close together as possible in order to aid the development process.

Johnson and Marsh [1998] came to a similar solution for robotic arms, but did away with text-based programming entirely in favour of a higher level CAD approach with integrated tools such as path-finding and free-space detection.

The ACT system, developed by Mazer *et al* [1991], was based around the development of advanced program tools which were then combined with traditional programming methods to form a more effective programming environment, again, for robotic arms. ACT includes a 3D simulation engine using GL, as well as the ability to link with actual robots and work in real-time.

The FDNet programming environment by Tokuda *et al* [2004] addresses mobile search and rescue robotics using a neural network-like approach. The main issues identified were robot complexity and modularity. Tools developed to cope with these problems were a viewer, editor and logger combined to form a 'human interface.' The system could function in real-time or 'offline' by use of a simulator.

Belousov *et al* [2001] developed an integrated system for remote control and programming of robots over the Internet. Their work addresses issues with communication lag between the developer and the robot and coping strategies, including novel control methods. The development environment includes a 3D representation of the remote robot as well as real-time video and programming facilities.

There is considerable work on mobile robot programming tools, for example the Player and Stage project [2005] provides a good set of tools for programming mobile robot systems, but lacks an integrated IDE to bring them together.

Our goal in this project is to develop an extendable programming IDE for mobile robot systems that can include new developments in robot programming languages, as well as new tools for interaction between robots and humans.

3 Requirements

The project requirements were to design and implement an IDE for Robotics, using an existing IDE tool as a starting point. It was also required that the final IDE support a variety of languages including the Python language, in particular the RADAR extensions designed by PhD student Geoff Biggs [Biggs and MacDonald, 2005]. Finally, the IDE had to be demonstrated by at least one robot programming application.

To be a compelling alternative to existing robot development methods, the new IDE design must address the common issues with robot programming listed in Section 2. The IDE must provide a modern, streamlined development process.

4 Current Methods

The first step was to evaluate the processes currently involved in developing and testing robot programs. Although these processes are individual and differ subtly between developers, there are elements common to all.

All developers require a code editor, a debugger, and either a robot simulator or a robot management tool. Additionally many developers employ some form of visualization or data viewing tool to show data produced or sensed by the robot.

One problem with this approach is that it requires the management of a number of separate tools by the developer. The tools must be loaded and configured before any development can proceed. This is a time-consuming and menial process.

Additionally, the developer is provided with very little in the way of useful debugging information. Most bugs must be diagnosed by writing code specifically to output the state of the program and the robot around the failure point. Once the problem has been resolved, this code becomes redundant and represents wasted development time.

Part of the problem is that robots are complicated systems that operate in real-time. The majority of programming tools assume that it is possible to freeze execution of the program while debugging takes place. As robots operate in the real world, this is often not an option.

4.1 Improvements

To resolve the issues identified with current robotic development methods, two main improvements were targeted. Integration of the wide range of tools used in robot programming is essential. The code editor, debugger, simulator and visualisations must all be accessible from within the IDE. The IDE must also provide accurate and timely debugging information to the developer. This should include both a real-time debugging facility and full log of all robot data and activities.

5 Base IDE

In order to focus on the IDE aspects specific to robotic programming, it is important to base the new IDE on an existing framework. This framework must provide code editing and basic debugging functionality. Frameworks under consideration were judged on five critera:

1. Support for the Python scripting language is essential, so we can use our RADAR real unit extensions to the language. Additionally, Python is a flexible rapid-development language well suited to robotics research.
2. An open-source license is also a requirement. This permits modifications the underlying code of the framework as well as a simplified distribution model.
3. Robotics development takes place across a wide variety of platforms. We felt that this should be reflected by adopting a cross-platform approach to the IDE. This meant that the base framework must be supported across a minimum of two platforms, preferably Linux and Windows.
4. As the project is intended to expand on the base functions of the IDE, the extendability of the chosen framework is also important. Although an open-source platform permits modification to the IDE source code, this is not an optimal solution; ideally the basis for the IDE will support such extensions without source modifications.
5. Finally, the IDE chosen must both (a) be in popular use (to ensure ongoing development and support) and (b) be user-friendly. That is, it must include such modern IDE features as syntax highlighting and project management.

IDEs considered include Eclipse [2005], NetBeans [2005], KDevelop [2005], IDLE [2005], and Wing IDE. As shown by the table in Fig. 1, Eclipse satisfies all the selection criteria used. In particular, Eclipse is the only framework that supports extensions natively.

IDE	Criteria				
	Support Python	Open Source	Cross-Platform	Extendable	User Friendly
Eclipse	✓	✓	✓	✓	✓
NetBeans	✓	✓	✓		✓
KDevelop	✓	✓			✓
IDLE	✓	✓	✓		
Wing IDE	✓	✓	✓		

Fig. 1: Base IDE Selection

5.1 Eclipse

Eclipse is an open-source IDE framework originally developed by IBM. It is written wholly in Java, and so supports operation on a wide variety of operating systems. The entire Eclipse framework has been designed to be extensible through user-developed plug-ins written in Java.

Natively the Eclipse IDE supports only Java development, but the PyDev plugin is available, which enables integrated development in Python [Zadrozny and Totic, 2005]. Eclipse, with the PyDev plug-in, forms the basis of our Robotics development environment

6 Robot programming framework

Our goal is to supplement or replace existing code development methods. So it is necessary to select one or more popular robot programming tools to integrate in the IDE tool, so that the user does not have to develop all robot code from scratch.

The selection of a popular tool is critical to ensure that the IDE can be immediately useful to the maximum number of robot developers. During development a robot simulation environment is useful for debugging. Testing with real robots is also important, so the tools must be able to control some of the robots available in the laboratory. The presence of both simulation and management capabilities within a single tool suite is preferable, simplifying IDE integration and allowing developers to switch between simulated testing and real testing transparently.

The Player/Stage [2005] robot simulation and management suite was eventually chosen. It has a simple but powerful network-based interface, and comprises both simulation and management capabilities.

Our group also has several Pioneer robots and a B21r robot suitable for testing purposes. Both of these robots have drivers written to work with Player and Stage.

6.1 Player and Stage

The Player and Stage robot simulation suite is ideally suited to our IDE development project. It is a combined open-source robot management server and simulation tool. Player is the robot server, and Stage is the simulator.

Robots communicate with the Player server, which provides a unified interface for robot control programs to access. These programs connect to the Player server over

standard network protocols.

Stage provides virtual robots and devices to the Player server, which are accessed by robot control programs in exactly the same way as are real robots. This means that programs can be developed and tested using the Stage simulator, and then switched transparently to control real robots.

7 Eclipse Extensions

The majority of the Robotics IDE functionality was provided through the development of a plug-in for the Eclipse framework. This plugin has three main functions, covered in the following sections. An overall system block diagram is shown in Fig. 2, below.

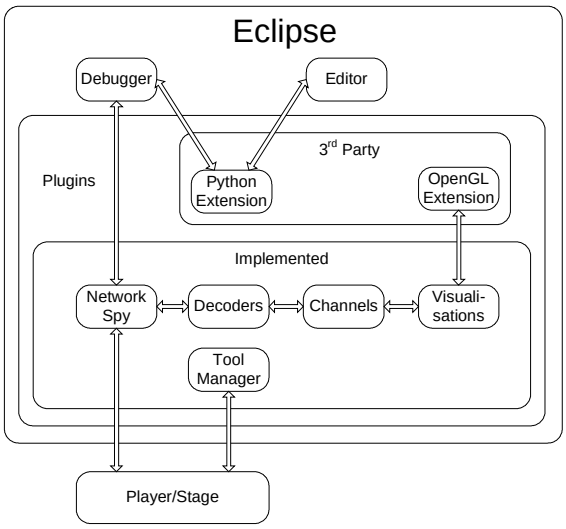

Fig. 2: IDE System Block Diagram

7.1 External Tool Manager

The External Tool Manager was developed in order to automate the menial tasks involved with setting up the development environment. It executes and maintains tools required by the developer that are not a fully-integrated part of the environment.

In the present implementation of the IDE, the Tool Manager is used to manage only one external tool, the Player and Stage suite. This is an open-ended implementation, however; the user may define any number of tools to execute or manage during a development session.

The External Tool Manager is also responsible for configuring the Network Spy (a proxy server covered in more detail in the following section). The Player server listens for connections from robot programs on certain ports. The Tool Manager extracts these ports from a Player configuration file. The ports are modified and written to a temporary file, which is used to run Player. The Network Spy is then invoked and listens on the ports that would normally be used by Player.

7.2 Network Spy

Although limited visualisation and debugging tools exist for Player, they are unreliable as they use a separate network connection. Thus the visualisation will not necessarily be showing the same information that is being 'seen' by the robot program. Buffer overflows and network outages can destroy the synchronisation between the visualisation and the data the robot program receives. These problems will not necessarily affect both connections, so the visualisation tools provided with Player do not always provide a true picture from the point of view of the robot program.

The Network Spy aims to address this issue, illustrated in Fig. 3. It is a versatile proxy server suitable for intercepting network connections and examining the data being exchanged. When the Player server is initialized by the External Tool Manager (as detailed in the previous section), the connection details for the Player Server are extracted and altered to allow the Network Spy proxy to sit transparently between robot programs and the Player server.

The information gained through this interception is accurate; tools displaying information gathered by the Network Spy are guaranteed to be seeing the same data as received by the robot program.

Fig. 3: Network Spy placement

7.2.1 Proxy System

Each port used by the Player server has a single Proxy server thread listening on it and accepting connections. When an incoming connection is accepted (from a robot program), a connection is made to the player server and two client threads are started, one forwarding data in each direction (program to server and vice versa).

Data forwarded through a client connection is passed to a Decoder subsystem which separates the stream out into individual Player packets, shown in Fig. 4. These packets have header information containing the message type (configuration, data etc) and the device type (sonar, laser etc), as well as a data section containing actual sensor data. The packets are then logged and passed to the Stream system (detailed later) for use by the debugging tools within the IDE.

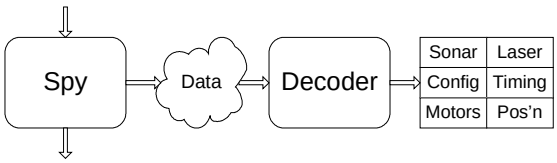

Fig. 4: Network Spy and Decoder

Any changes in the Player network protocol can be easily reflected by small changes to the Decoder system. Similarly, the whole IDE could be expanded to cope with different network-based robot systems by writing a new Decoder to convert the new protocol to the internal IDE format.

7.2.2 Stream System

The Stream system is the backbone of the IDE. It takes data from a single producer, such as the Network Spy Decoder, and copies it to any number of registered consumers, such as the visualisation tools.

Data from the Network Spy is classified by source address, destination address and packet type (a combination of the message type and device type). Thus there could be a sonar data stream being forwarded from the Player server to the robot program. The sonar visualisation tool registers as a consumer of this stream, and so will be sent a copy of all the packets posted to this stream.

Streams are intended to be conceptual rather than physical divisions of data, and so they are not created or managed separately. If a packet is transmitted which does not fit in to any existing Stream, a new Stream is automatically created. It is also possible for consumers to register for streams that do not yet exist. This allows developers to set up visualisations before they start debugging their program. It is also possible to register for Streams using wildcards for selection criteria (source address, destination address, and type identifier). Thus if a developer is only debugging one robot, they can omit all source and destination data and specify only the packet type.

It is important to note that a developer could easily create their own type of Stream and pass data through it to the debug tools. For example, a developer might be interested in average sonar values. The developer needs simply to register as a consumer of a sonar Stream, process the data, and post the average values to a new type of Stream. Half of the stream identifiers are reserved for user-created data (there are 2^{31} user stream identifiers available). Once this stream of data has been created, a user could either view it using an existing visualisation or create a new visualisation to show the data. This process will be detailed in the next section.

7.3 Visualisations

Once the debugging data has been gathered by the Network Spy, the data must be shown to the developer in real-time. Visualisations are provided in the Robotics IDE for showing data. A Visualisation gives a visual interpretation of data on the development station screen.

Visualisations register for any number of data Streams from the Network Spy. When new data on any of these Streams becomes available, the Visualisation is notified and instructed to recreate its image based on the new data.

OpenGL is used for displaying Visualisations. OpenGL is a cross-platform tool well supported across Linux and Windows. Since all rendering code is executed within the Java Virtual Machine, speed is a concern. OpenGL permits complex scenes to be rendered quickly by handling the majority of the actual drawing operations.

Another benefit is that Visualisations may easily be made semitransparent and superimposed by the developer. The advantages of this approach are easily seen; many robot sensors gather the same data by different means (i.e. sonar, laser, infrared rangefinders). Being able to visually compare the output of these sensors gives a marked benefit for debugging.

OpenGL also allows for future expansions using robot simulators that work with three dimensions, such as Gazebo [2005] (Stage is only a 2D robot simulator).

7.3.1 New Visualisations

New visualisations may be created easily by the developer, because of the extensible structure of the Robotics IDE. For each new visualisation, a new subclass of the abstract StreamRenderer class is created, and three abstract functions completed for acquiring the data, refreshing the display lists used by OpenGL, and calling OpenGL to render the lists.

Combined with the malleable Stream system, this allows developers to easily create their own custom tools and further extend the Robotics IDE.

7.3.2 The Robot View

Visualisations are shown within the Robotics IDE in a special Eclipse extension called a 'View'. This is a panel managed by the GUI which can be 'snapped' to any position within the IDE. The position of the View persists between instances of the IDE, so the developer can place the View to their preference and it will be opened there by default. Any number of views may be opened by the developer, each hosting any number of visualisations.

Each View contains a single canvas which is used as an OpenGL rendering target. Visualisations are added to the View by the developer. When the View needs to be redrawn, rendering is initialized and the render functions of each Visualisation will be called in turn.

8 Usability

Fig. 2 shows the overall extended IDE design. Fig. 5 shows an example screen shot of the IDE. The system was tested for usability by undertaking two development projects, one using the new IDE and one using existing development methods. Two simple robot programs were designed and implemented for the Pioneer mobile robot. The time taken to implement and test each program was measured as well as the number of major bugs resolved. The outcome of the tests was that development using the IDE was faster and more streamlined.

8.1 Laser Sentinel – Without IDE

The first program developed was called the 'Laser Sentinel'. This program used a Pioneer mounted with a miniature forward-facing 180 degree laser scanner. The Pioneer was simply programmed to smoothly turn and face the nearest object as reported by the laser scanner.

Development time for this program was 2 hours and 24 minutes. The program had two major bugs; the first was due to the laser scanner readings being read in the wrong order and the second was from not establishing a maximum range for 'interesting' obstacles.

8.2 Sunday Stroll – With IDE

The second program was called 'Sunday Stroll', and was developed using the new Robotics IDE. This program uses a standard Pioneer robot with sixteen sonar sensors mounted facing in all directions around the robot. The robot was programmed to move around its environment turning at random. When the robot got too close to an obstacle, it would stop, turn to face a free direction, and continue moving.

Development time for the 'Sunday Stroll' program was 1 hour and 17 minutes. There were three major bugs. The robot initially tried to turn and move at the same time, but there was never enough room to do so before hitting the obstacle. Sonar sensor numbering was initially confused and resulted in the robot never moving at all. Finally, the robot did not turn sharply enough to make the movement truly random – the small fluctuations averaged out to a straight line, making the movement predictable.

9 Reliability

Testing was undertaken on a computer system with a 2.4GHz Intel Pentium 4 processor and 1GB of DDR400 RAM.

Information gathered from the system logs at the end of the project revealed that over six million packets had been successfully processed by the Network Spy system without fault. The system easily handled the data throughput and visualisation load of six robots running in tandem with both laser and sonar sensors, which drove the rendering requirement up to 120 frames per second.

The longest measured uptime was 36 hours, at which point the system was still running fine with no instabilities or processing errors.

There are still some small faults in the system; but these were traced back to some of the 3rd party plugins in use. It is expected that as these leave beta status and are formally released, the overall stability will improve.

10 Conclusions

Our robot programming IDE extensions to Eclipse have achieved all the original goals and more. The extensible IDE permits developers to increase the functionality and tailor the performance of the tools to their own specifications. As programming and development methods vary significantly between developers, this feature is a compelling advantage over existing methods.

Usability and reliability testing have shown that the system has considerable advantages over existing scattered development methods, with no real disadvantages.

10.1 Future Work

Since the system is extensible, there is no real limit to the amount of work that can be done to extend it. As programming techniques and tools evolve, these changes will need to be integrated into the IDE - however, at present there are three main areas suitable for extension.

Gazebo [2005] simulates robots in three dimensions, unlike the Stage robot simulator, and could be incorporated in the IDE. Gazebo is a part of the Player project, so no changes to the Network Spy system are necessary. The Visualisations are also rendered in 3D, so 3D sensor data can be easily shown with the existing system.

At present, although logging is comprehensive there are no integrated tools to permit a developer to leverage the benefits of the information. A worthwhile addition to the IDE toolset would be a tool for analysing logs, or replaying sensor data to visualisations. This would allow developers to more easily pinpoint errors caused by individual bad packets of data that might not be visible when visualisations are watched at real-time speed.

At present only two visualisations have been implemented, one for viewing Sonar sensor data and one for viewing Laser sensor data. There are a wealth of sensors supported by the Player/Stage suite - adding visualisation support for these would be a simple task.

Acknowledgements

Steve Hsiao worked with Luke Gumbley to design and implement the project.

References

[Belousov, Chellali and Clapworthy, 2001]. Belousov, I.R.; Chellali, R.; Clapworthy, G.J.; Virtual Reality Tools for Internet Robotics. *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference.* Volume 2, Pages 1878-1883.

[Biggs and MacDonald, 2003] Geoffrey Biggs and Bruce MacDonald. A survey of robot programming systems. *Proceedings of the Australasian Conference on Robotics and Automation*, CSIRO, Brisbane, Australia, December 1--3 2003

[Biggs and MacDonald, 2005] GM Biggs and BA MacDonald. A Design for Dimensional Analysis in Robotics. *Proceedings of the third International Conference on Computational Intelligence, Robotics and Autonomous Systems*. 13 - 16 December 2005. Singapore. To appear.

[Eclipse, 2005] The Eclipse Foundation. Available from: http://www.eclipse.org. [Accessed August 2005].

[Gazebo, 2005] About Gazebo [online]. Available from: http://playerstage.sourceforge.net/index.php?src=gazebo [Accessed August 2005].

[Gumbley, 2005] Luke Gumbley. Development of an Integrated Robotics Programming Environment. Final year project report, Department of Electrical and Computer Engineering, University of Auckland, New Zealand. 2005.

[Hsiao, 2005] Steve Hsiao. Development of an Integrated Robotics Programming Environment. Final year project report, Department of Electrical and Computer Engineering, University of Auckland, New Zealand. 2005.

[IDLE, 2005] Python Software Foundation (2005) IDLE – an Integrated DeveLopment Environment for Python [online]. Available from: http://www.python.org/idle/ [Accessed 9th April 2005].

[Johnson and Marsh, 1998]. Johnson, C.G.; Marsh, D.; A Robot Programming Environment Based On Free-Form CAD Modelling. *Robotics and Automation, 1998. Proceedings. 1998 IEEE International Conference.* Volume 1, 16-20 May 1998 Pages 194-199.

[KDevelop, 2005] The KDevelop-Project (2005) KDevelop – an Integrated Development [online]. Available from: http://www.kdevelop.org/ [Accessed 9th April 2005].

[Lozano-Perez, 1983] Lozano-Perez, T. Robot Programming. *Proceedings of the IEEE.* Volume 71, Issue 7, July 1983 Pages 821-841.

[Mazer et al, 1991]. Mazer, E.; Pertin-Troccaz, J.; Lefevre, J.-M.; Faverjon, B.; Ijel, A.; Bellier, C.; Ferrari, B.; Barret, M.; Sellers, P.; Lefebvre, J.M.; Hassoun, M.; Alchami, O.; ACT: A Robot Programming Environment. *Robotics And Automation, 191. Proceedings, 1991 IEEE International Conference* 9-11 April 1991, Pages 1427-1432.

[Naylor et al, 1987] Naylor, A.; Volz, R.; Jungclas, R.; Bixel, P.;Lloyd, K.; PROGRESS—A Graphical Robot Programming System. *Robotics and Automation. Proceedings. 1987 IEEE International Conference.* Volume 4, Mar 1987 Pages 1282-1291.

[NetBeans, 2005] NetBeans (2005) Welcome to NetBeans [online]. Available from: http://www.netbeans.org/ [Accessed 9th April 2005].

[Player/Stage, 2005] Player/Stage project. [online]. Available from: http://playerstage.sf.net. [Accessed

August 2005].

[Tokuda et al, 2004]. Tokuda, K.; Jadoulle, J.; Lambot, N.; Youssef, A.; Koji, Y.; Tadokoro, S.; Dynamic Robot Programming by FDNet: Design of FDNet Programming Environment. *Intelligent Robots and Systems, 2004 (IROS 2004). Proceedings, 2004 IEEE/RSJ International Conference.* Volume 1, 28 Sept-2 Oct 2004 Pages 780-785

[Zadrozny and Totic, 2005] Fabio Zadrozny and Aleks Totic (2005) Pydev. [online]. Available from: http://pydev.sourceforge.net/ [Accessed 9th April 2005].

Fig 5. Example screen shot of the extended Eclipse IDE, showing laser and sonar visualisation “View” within Eclipse at the bottom left.