2010 IEEE International Conference on Robotics and Automation
Anchorage Convention District
May 3-8, 2010, Anchorage, Alaska, USA

Applying regression testing to software for robot hardware interaction

Geoffrey Biggs
Intelligent Systems Research Institute
National Institute of Advanced Industrial Science and Technology (AIST),
AIST Tsukuba Central 2, Tsukuba, Ibaraki 305-8568, Japan
geoffrey.biggs@aist.go.jp

***Abstract*—If robots are to be fully accepted in the homes and offices of the world, it is important that they are guaranteed to be reliable and not to cause damage or harm. This requires testing robot systems and the software that comprises them. But testing robot software has always been a difficult process for developers. Issues of repeatability, safety, access to hardware and the general complexity of robot software are encountered. In industrial robotics, these difficulties are mitigated somewhat by the relatively simple, repeatable tasks and the controlled environment. Robotics for real-world environments, on the other hand, face the full challenges of testing.**

In this paper, we discuss regression testing at a low level of individual software components, particularly those components that are designed to interface with robot hardware. We present a software system for regression testing these components in a fully repeatable fashion as a case study of performing such testing in robotics. The presented system provides an efficient and quick method to monitor changes in the behaviour of software components as they are developed. Developers of robot software can quickly discover undesired changes and correct them.

I. INTRODUCTION

Robotics is a field rapidly moving from its original environment of industrial automation, where it is well-established, to dynamic, human-oriented environments. In these new environments, robots face many significant challenges, from the dynamic, rapidly changing nature of the real world[1] to the inherit uncertainty found there. This is leading to new challenges in the development process of creating robot software [1].

As with any embedded system, or indeed any computer or mechanical system in general, robots require significant testing to ensure they behave as planned under all circumstances. This is particularly important during erroneous conditions. It is relatively simple to test robots located in controlled environments, particularly industrial robots, for all expected conditions and a wide range of unexpected conditions, such as mechanical failures. The remaining unexpected conditions can be controlled by virtue of the environment itself being controlled. Similarly, the safety of surrounding humans can be guaranteed by preventing entry to the controlled environment, as is often the case with the robotic work cell of a factory robot.

[1]If we consider an industrial environment to be controlled to the point of not being real, we can describe human environments as the "real world."

Unfortunately, conditions in the real world cannot be controlled so tightly, and so it is impossible to guarantee that all conditions not tested for will not occur. This presents the field of service robots with a challenge: how to guarantee safety when a complex robot is entered into the dynamic, ever-changing environment of the human world? This is a task perhaps best tackled in a piecemeal fashion. Testing and guaranteeing the entire system as a whole is probably too difficult a task without first guaranteeing the safety of individual components. First, each component of the system should be tested to ensure it conforms to a known set of conditions, particularly with regards to its inputs and outputs. If each component can be guaranteed to only produce a known set of outputs given any input, the problem set of the whole created from the component parts becomes smaller and safety may be easier to guarantee.

In this paper, we consider the use of a technique from software engineering, regression testing, and demonstrate its use on software components for interacting with robot hardware. Regression testing is a well-known testing technique that ensures that new changes do not break existing functionality.

We have created a library that can be used to regression test software written to interface with robot hardware, software that has traditionally been difficult to test in a way suitable for regression testing.

The rest of the paper is laid out as follows. Section III briefly discusses some methods that are currently used to test robot software. Section IV describes the library created to facilitate regression testing of some types of robot software, with its testing facilities described in Section V. Some use cases giving examples of the use of the system are given in Section VI, followed by discussion in Section VII and conclusions in section VIII.

II. REGRESSION TESTING

Regression testing is a well-known technique for functional testing. It is used after a change, such as a new feature or a repair, has been made to the software, and after that change has itself been tested. It aims to ensure that new changes in the software do not break existing functionality [9]. Such testing is necessary because, in complex software, it is very difficult to determine exactly how a change made will affect the rest of the software.

It is important that regression testing cover all parts of the software. This means that software dealing with hardware interaction cannot be left out simply because the hardware is not available. Regression testing is typically automated, due to its tedious nature. Without automation, regression testing may not ensure complete coverage of the software features. Any method of regression testing needs to be able to be automated [10].

III. Testing in robot software

There are two commonly-used methods of testing robot software. The first is to run the software on real robot hardware, while the developer or an assistant hovers nearby with their hand on an emergency stop switch. This is not an ideal testing environment. As well as the obvious reasons such as the danger involved in trying new code on a valuable piece of hardware that may be capable of causing damage to itself or injury to people around it, there are other reasons such as the non-repeatability of this method of testing. A bug may appear in one run of the software, but, due to a very slight change in the robot or the environment around it, not appear in the next. Repeatability can only be achieved by a statistical approach of running the software many times and performing a statistical analysis on the combined outcomes. There is less verifiability in such an approach, and it is both time-consuming and difficult to debug software under such circumstances.

Simulators are usually used to avoid the dangers of testing on a real robot, but they can also add some repeatability. The simulated environment and robots can be controlled much more finely than real hardware. Even so, simulation is still a difficult way to test robot software. It may not necessarily guarantee repeatability and it is typically not possible to step through control code as the simulator keeps running. It is, however, possibly the best method we have of testing a complete robot system with the greatest repeatability (given a simulator with sufficient accuracy in its calculations). A large number of simulators have been developed over the years. The Stage [2] and Gazebo [3] simulators are the best known examples of simulators for research, although there are also several simulators in industrial robotics. Examples include the simulators provided by companies such as ABB [4] and KUKA [5] for their industrial robots, and the COSIMIR simulator [6].

When we consider the smaller components that make up a complete robot system, other testing methods become available. Components can be tested in isolation to ensure they conform to all aspects of their described external interface. If the design of the greater system that combines components together is correct, this greatly reduces the likelihood of problems occurring in the system as a whole. This will often involve testing based on use cases from the design phase of the development process. There are established ISO processes for such testing that can be applied [7].

Testing components in isolation is a well-established software engineering practice, and it applies just as well to robot software development as it does in other domains of programming. Components can also be regression tested to compare their new behaviour after modification with previous behaviour. This allows the developer to identify all changes in behaviour that have been caused by changes to the software code, and correct those that are unintended or not desired.

When discussing testing robot software components, it is important to consider the difference between purely software components, such as navigation algorithms, mapping systems and image processing algorithms, and software for interfacing with and controlling robot hardware. The former is relatively simple to test. It does not require the presence of hardware (in most cases, even a simulator is not necessary, just a suitable data set) and it is possible to make the tests one hundred percent repeatable through the use of a pre-defined data set as input. A regression test of such software can be performed by running it over the same data set before and after changes are made, and comparing the outputs. Software for interfacing with hardware, on the other hand, requires that hardware be present and may depend on environmental factors being identical. For example, a software driver for a laser scanner (not a data processing algorithm, but the software responsible for interpreting the scanner's data transfer protocol).

To regression test such software, we need a way to make the output of the hardware perfectly repeatable. Ideally, we should not need the hardware to be present to regression test the software driver, which would allow regression tests to be automated. The following sections discuss a software system that allows this, implemented as part of the flexiport library [8].

IV. Flexiport

Flexiport is a low-level stream-oriented communications library written in C++. It is designed to provide a unified API for communicating with hardware via various different types of connections. Currently, it provides support for serial port (including serial-over-USB), TCP and UDP connection types, three common methods of communicating between computers and hardware. The bulk of the API is unified and the library uses an object-oriented design. This allows different port types to be swapped without changing the code of the software using the library.

Swapping port types at run-time is facilitated by a factory function used to create an instance of a `Port` object and the method of customising that object to the applications' needs (see Listing 1, line 4). Rather than passing in port settings as individual function arguments, the factory function accepts a string containing the port settings as *<option,value>* pairs, similar to command line arguments passed to an executable in a command shell. One of these is the type of port to create, such as "serial" or "tcp," allowing port types to be swapped transparently without code changes.

Flexiport is distributed as a member of the Gearbox project [8]. It is utilised by other libraries in Gearbox.

V. Regression testing with flexiport

The flexiport library provides the low level communications between robot hardware and the software directly respon-

Listing 1: Creating two ports with different configurations. One port is a serial port, the other is a TCP network port.

```
string portOptions =
    "type=serial,device=/dev/ttyS0,timeout=1";
flexiport::Port *port1, *port2;
port1 = flexiport::CreatePort (portOptions);
port2 = flexiport::CreatePort
    ("type=tcp,ip=130.216.217.24,listen");
```

sible for communicating with and controlling it. Ensuring the correctness of this interface, particularly a developer's interpretation of the communications protocols used, means ensuring that the software implementing the interface is correct. We term this software the client software from here on, i.e. a client of the flexiport library and the hardware.

The correctness of this interface can be tested by ensuring that the contents of the communications session on the wire between the software and the hardware is correct. With regards to regression testing, this means ensuring that one communications session is identical (or similar enough, within acceptable bounds) to a previous communications session. With the flexiport library, we can perform regression testing by applying the well-known technique of log files: first recording a communications session that is known to be good, and then replaying that session back through the software component, emulating the hardware device and communications port. The remainder of this section describes the facility in flexiport for recording log files, followed by the facility for handling the more complex task of emulating a communications port.

A. Logging ports

The ability to swap port types transparently presents us with an opportunity to introduce a powerful logging facility to the library. By creating a new port type that wraps around one of the existing port types and logs every action performed with that port, we can record a detailed log of the activity of the communications channel.

This is the purpose of the *LogWriter* port type. This port type encapsulates another instance of the `Port` type. All calls to the standard flexiport API made against this object are passed on through to the internal `Port` object, but first they are recorded in a log file, including their arguments, such as data to be written. Similarly, the result of any calls is recorded before it is returned to the caller. This includes any data received by the port, where applicable.

A log of a communications session actually consists of two separate log files, one for the data written to the port and one for the data read from the port. The files are separate to simplify the format they are written in, as there is no need to specify whether an operation is a read or a write in the files themselves this way.

The log file is written using a simple block-based format, illustrated in Figures 1 and 2. Each block consists of a time stamp, a data size (as an unsigned 32-bit integer), and any

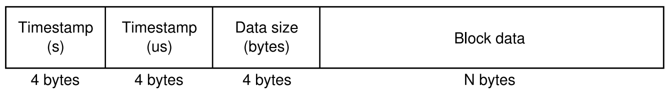

Fig. 1: The format of the log files used by flexiport to record communications sessions.

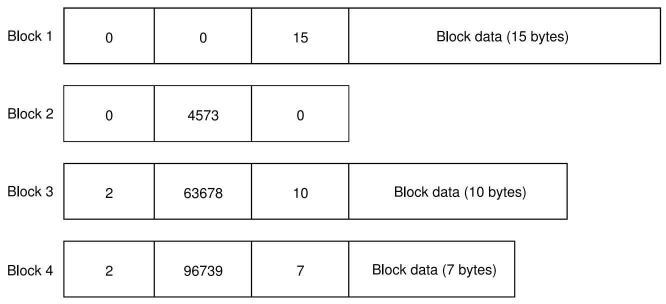

Fig. 2: An example log file showing how the format is used.

data that is present. The time stamp is taken against the time the log file was opened for writing. It is measured in seconds and microseconds and is written as separate 32-bit unsigned integers. 32-bit sized values are used to ensure compatibility across 32-bit and 64-bit architectures (important if the log files are to be disseminated to other library users). It is also unlikely that the 293-billion-year range afforded by using 64-bit values is necessary. These values are changed to network-byte order before writing for the same reason. The data, however, is written in its raw state, as byte-ordering is not important for single-byte values.

When considering what to log, the `LogWriter` object was designed with a minimalist approach. It only logs those actions and the data necessary to recreate the data received from and sent to the port by the client software. This is because when regression testing software using flexiport, it is not important to check that, for example, the data received from the hardware during a skip operation (which reads data from the port and dumps it without returning it to the caller) is identical to data skipped in previous communication sessions. It only matters that the data on either side of the skip that *is* used by the client is identical. This both simplifies the implementation of the log recording and playback, and reduces the size of log files.

Client software can provide a log-writing alternative to their usual port type for the *type* option when the port object is created. The factory function that constructs port objects (see Section IV) will construct an instance of the `LogWriter` object, passing in the remaining options as usual, but with the `type` option re-added, this time specifying the type of the internal port object to create. The `LogWriter` constructor then makes its own call to the factory function to construct the internal port, passing in the entire set options. In this way, a user still has full control over the options of the internal port object. For example, the options shown in Listing 1 could be amended to:

`type=seriallog,device=/dev/ttyS0,timeout=1`
creating an identical serial port, but wrapped with a log writer

that will record all activity using that port.

B. Emulating ports

Once a communications session has been recorded, it can be used to emulate the hardware and communication port, and so test if the software driver to interface with the hardware is behaving as expected with respect to previously known behaviour.

The `LogReader` port type is used to provide this functionality. Unlike its log-writing relative, this port type does not need to encapsulate a real port object, as it uses the provided log files for its data source.

While at first, playing back a recorded communications session would seem to be a simple task, it is not as easy as simply reading from the recorded file and comparing data. It is important that, for an accurate test, the timing of when data becomes available for reading is emulated. Driver software often relies on the timing of data for operation, such as the length of time between messages being used to determine when a message has ended.

However, it is also not simply a case of reading a chunk from the log file and attempting to match it against the current read or write operation being performed by the client software. The goal is to ensure that the client software still performs the same overall task correctly and with the same result, not that it performs every operation identically. If what was previously done in three separate read operations is now performed in a single operation, this is acceptable, provided that there is data available from the log file within the timeout of that single read operation and the same total quantity of data is read.

For this reason, the `LogReader` object performs a complex operation involving reading and buffering the data from the log files in overflow buffers as necessary. The algorithm for reading blocks from the log files is shown in Figure 3. It should be noted that the option to test without emulation of timing is available, if timing is not important.

As each read is performed, the data is sent back to the client software. If the client software is expecting something different, due to a change in its code, it will cause an error in the client software and will therefore indicate that the regression testing has failed. It is then up to the developer to determine the cause of the failure, which may be due to an unintended change in the behaviour of the client software, or due to a planned change (in which case, new log files matching the new behaviour must be created for future tests).

Testing of writes requires comparing the data passed in by the client software against that found in the write-data log file. Times are checked to ensure data is not written too early, again because some protocols require accurate timing of reads and writes. However, more flexibility is given for checking of write operations than read operations. Three levels of strictness are available:

1) Writes are not checked at all
2) Writes are checked, but timing is not checked
3) Writes are checked, and timing must be accurate to within a user-provided margin of error.

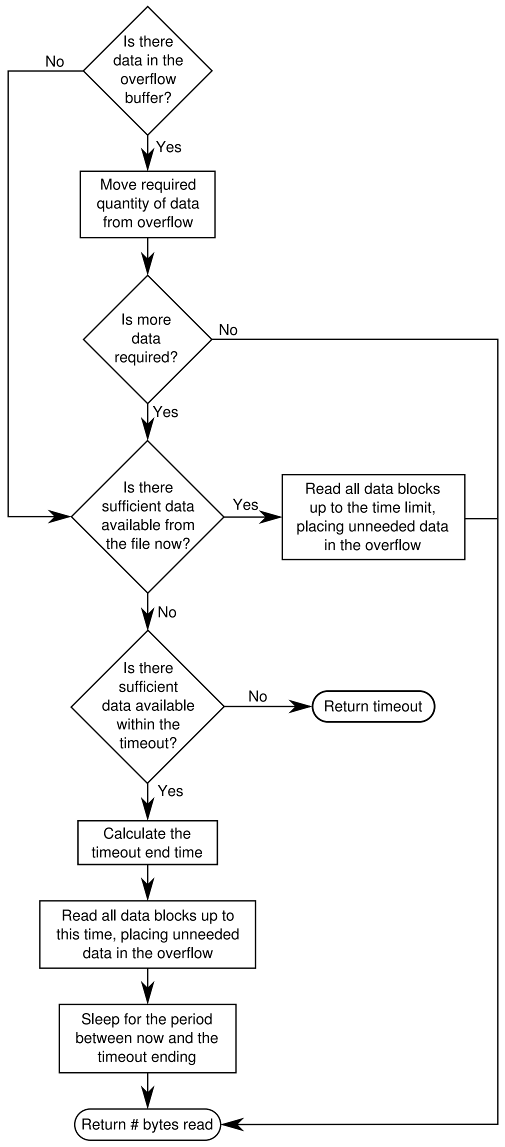
Is there data in the overflow buffer?
No
Yes
Move required quantity of data from overflow
Is more data required?
No
Yes
Is there sufficient data available from the file now?
Yes
Read all data blocks up to the time limit, placing unneeded data in the overflow
No
Is there sufficient data available within the timeout?
No
Return timeout
Yes
Calculate the timeout end time
Read all data blocks up to this time, placing unneeded data in the overflow
Sleep for the period between now and the timeout ending
Return # bytes read

Fig. 3: The algorithm for reading blocks from the log files. Note that memory management of the overflow buffers is not shown.

The reason for option one is that often the commands of a protocol may change while the data returned by the hardware does not, necessitating a change in the behaviour of the client software that sends those commands without changing the behaviour responsible for reading the result. Option two is provided for the case where timing is not important.

VI. Usage examples

This section describes two examples of using flexiport to both bring flexibility in communications to robot software for interfacing to hardware and, more importantly, using flexiport to regression test that software. The first example is of a hardware driver in the Gearbox project that uses flexiport for communications. The second is a more generic solution as part of the Player project [11].

A. Gearbox

Flexiport is a member of the Gearbox distribution of robot software. The hokuyo_aist library from Gearbox uses flexiport for communication with hardware.

Hokuyo_aist is a hardware driver for small-scale laser scanners from Hokuyo, including the URG-04LX, UHG-08LX and UTM-30LX [12]. The URG-04LX provides serial and USB (via USB serial port emulation) communications, while the others provide only the USB option. They can all be communicated with using flexiport's `SerialPort` object.

Because of this, we are able to regression test the hokuyo_aist library. A communications session between the example program provided with the hokuyo_aist library and a URG-04LX model laser scanner was recorded. The log files from this recording are distributed with the library, available for use by users of the library to try the example when hardware is not present and confirm the library is functioning as expected. This is a secondary use of the logging facilities.

This process can be automated within Gearbox's automated testing system, ensuring that the library continues to function correctly after any code changes. This is an important example of how we can ensure robot software quality is maintained.

A separate, somewhat anecdotal example comes from the development of the hokuyo_aist library. After its initial release, a bug report was received from one user that it was not functioning as it should for their laser scanner. The behaviour was difficult to describe and the verbose output of the example did not provide enough information to diagnose the problem. Because the bug only occurred on the user's specific revision of a URG-04LX laser scanner, it was not possible for the developers of hokuyo_aist to debug directly on the hardware. As a replacement, the reporter of the bug provided a recording of a communications session in which the bug occurred. The developers were able to replay this log through a hokuyo_aist instance as if using the actual hardware, with the added advantage of being able to step through the code, unconstrained by time as is usually the case when debugging with the actual hardware. The bug was rapidly diagnosed and fixed in much less time than it would have required had the log file not been available (especially given the delays and limitations of email communication). While not regression testing, this illustrates the importance of being able to record and replay the wire-level communications between robot hardware and its interfacing software in the modern world, where software may be used by others around the world. The mailing list exchange discussing this bug is archived online[2], and similar exchanges have since occurred for other bugs.

B. Player

Player [11] is one of the most popular robot software frameworks. It uses interfaces to abstract away the differences between various implementations of device types. One interface is provided for each device type, for example a *ranger* interface for range sensors, and a *position2d* interface for controlling the position and velocity of a robot in 2D space. Drivers provide the implementation for a specific piece of hardware, such as a driver for a SICK laser scanner or a driver for a Pioneer robot. Drivers can communicate with each other using these interfaces to access resources provided by other drivers.

One of the difficulties with developing drivers for Player has been maintaining them. Many of the drivers are orphaned, their original authors having moved on. With Player under constant development and API changes being made when new versions are released, all drivers must be updated regularly and bugs are often found by users that must be fixed. It is the responsibility of the core Player developers to perform these tasks. However, the core developers do not have access to most of the hardware supported by Player. Drivers are often updated without being tested, which can lead to bugs being introduced that are not detected for a long period of time. Some drivers are not updated for this reason (the "why update it if I cannot test it?" mentality), meaning they fail to work or even compile correctly.

The solution, and the inspiration for the work presented in this paper, is to provide a method for regression testing drivers when they have been updated after a change in Player's driver API. One way to do this is to use the flexiport library when writing a driver to provide the communications with the hardware. This approach leads to regression testing that is mostly similar to that described in Section VI-A.

Another method is through the use of Player's generic *opaque* interface. This interface is designed to allow devices which don't match one of the provided interface types to still use Player. A client and a driver send opaque messages, which are simply blocks of raw data, across the opaque interface, and decode them manually at either end. This can be utilised for removing the hardware communications from a driver by reading and writing an opaque interface as if it were a communications port. A separate Player driver, sitting at the other end of the opaque interface, receives data to write and writes it to the hardware port, and reads data from the hardware port which it then sends over the opaque interface. This separates hardware communications and protocol interpretation into two separate drivers, which can even be running on separate Player servers on separate computers.

By providing an opaque driver which uses flexiport to implement the hardware communications, regression testing capability has been added to Player for any driver which uses the opaque interface to access its hardware rather than directly opening a port (currently, this is only a small number of drivers, as the idea is relatively new). We are then able to record a communications session for any of these drivers, store it with the Player distribution, and test drivers at a later date without the hardware present. This driver uses flexiport's port options to allow a Player user to fully configure the port created by the driver at run-time, including using a `LogWriter` or `LogReader` port. See Listing 2 for an example of specifying the port options in the Player configuration file.

[2]See http://sourceforge.net/mailarchive/forum.php?thread_name=18405740.post@talk.nabble.com&forum_name=playerstage-users

Listing 2: A sample player configuration file showing the configuration of two flexiport drivers with different port configurations.

```
driver (
  name "flexiport"
  provides ["opaque:0"]
  portopts "type=serial,device=/dev/ttyACM0"
)

driver (
  name "flexiport"
  provides ["opaque:1"]
  portopts "type=logreader,file=test1"
)
```

This eliminates a major obstacle in the development and maintenance of Player drivers. The core developers are now able to update drivers when necessary and test their behaviour without requiring the hardware to be present.

VII. Discussion

The ability to regression test robot software components is important to ensuring the continued health of created robot software. In the world of open-source robot software, where software may be maintained by developers other than the original author, it is even more important.

However, regression testing of software for interfacing with hardware components is a non-trivial task. The hardware may not be available when the testing is to be performed, or may not be available at all. Therefore the ability to emulate any hardware in a generic fashion is important. The ideal method for doing this is to record and replay the communications session between the hardware and the interfacing software at the wire-level. This is the niche that flexiport fills for robotics with its logging facilities. Flexiport provides a great benefit to any software that must directly interface with hardware, allowing it to be tested in a repeatable way (not to mention the other benefits it brings in cross-platform capability and a unified API for various port types).

As mentioned in Section VI-A, flexiport also brings benefits in finding and fixing bugs in robot software that has been distributed to other users around the world. The log files can be used to replay a communications session that triggers a bug on some esoteric variation of hardware the developer of the software does not have access to, greatly reducing the time and difficulty involved in fixing the bug.

VIII. Conclusions

Testing in robotics is a difficult practice. Industrial robotics typically features repeatability of the tasks involved and a controlled environment. It benefits from these when it comes to testing, both in simplifying the testing requirements and reducing the exceptional conditions that must be tested for and guaranteed against. Robots operating in the dynamic and unpredictable real world do not have these benefits, and testing the software that must operate in these conditions is correspondingly more difficult. However, we can still apply existing established testing procedures in many ways. One of these techniques is regression testing, which allows developers to confirm that previously-established behaviour still takes place, usually in an automated way.

In this paper, we have described the implementation of a method for regression testing software that interfaces with robot hardware. The method uses a logging facility built into a communications library. It records a communications session between hardware and its driver software. The record of this session can then be replayed later, emulating the hardware and allowing the software to be tested independent of the hardware. Two examples of using this facility have been presented, and an example of the benefits it can bring to the debugging process has also been discussed. The ability to regression test software components is shown to be a benefit to the process of developing robot software.

References

[1] B. A. MacDonald, G. Biggs, T. H. Collett, and Y. H. Kuo, *Software Engineering for Experimental Robotics*. Springer, 2007, ch. 44.

[2] R. T. Vaughan, B. P. Gerkey, and A. Howard, "On device abstractions for portable, reusable robot code," in *Proceedings of the 2003 IEEE/RSJ Intl. Conference on Intelligent Robots and Systems (IROS03)*, vol. 3, Las Vegas, Nevada, October 2003, pp. 2421–2427.

[3] (2010) Player Project - Gazebo. [Online]. Available: http://playerstage.sourceforge.net/index.php?src=gazebo

[4] (2010) The ABB group. [Online]. Available: http://www.abb.com/

[5] (2010) KUKA Automatisering + Robots N.V. [Online]. Available: http://www.kuka.be/

[6] D. Freund, E.; Pensky, "COSIMIR Factory: extending the use of manufacturing simulations," in *Proceedings of the IEEE Intl. Conf. on Robotics and Automation (ICRA '02)*, vol. 3, May 2002, pp. 2805–2810.

[7] Y. K. Chung and S.-M. Hwang, "Software testing for intelligent robots," *Control, Automation and Systems, 2007. ICCAS '07. International Conference on*, pp. 2344–2349, Oct. 2007.

[8] (2010) GearBox Project - Flexiport. [Online]. Available: http://gearbox.sourceforge.net/group__gbx__library__flexiport.html

[9] G. J. Myers, *The Art of Software Testing*, 2nd ed. Hoboken, New Jersey: John Wiley & Sons, Inc., 2004, ch. 6.

[10] E. Dustin, *Effective Software Testing*. Addison-Wesley Professional, 2002, ch. 39.

[11] T. Collett, B. MacDonald, and B. Gerkey, "Player 2.0: Toward a practical robot programming framework," in *Proceedings of the Australasian Conference on Robotics and Automation*, University of New South Wales, Sydney, Australia, December 5–7 2005.

[12] (2010) Scanning range finder URG. [Online]. Available: http://www.hokuyo-aut.jp/02sensor/07scanner/urg.html