Novel Software Solutions for Automating Biochemical Assays

Rob Harkness

A thesis submitted for the Degree of Doctor of Philosophy

Faculty of Health and Medical Sciences
University of Surrey

November 2009
© Rob Harkness 2009
Abstract

Laboratory Automation is used throughout the pharmaceutical and biotechnology industries to assist research within the drug discovery process. Many software packages are commercially available for automating biochemical assays, such as the ELISA, as part of this process. However, it is often difficult for a scientist to translate their assay into what is essentially a piece of programming logic. Advanced users with an understanding of basic programming are often required. By shifting the development approach, a software product has been created that focuses on how the user can set up an assay as opposed to how the software will automate instrumentation.

A review of existing software in the field of laboratory automation and the scheduling methods that are used has provided a basic platform from which a new product, Overlord2, has been written using the Microsoft .NET framework. A flow chart interface has been selected as the method of describing an assay. This has the distinct advantage of allowing the user to control how their assay will be processed unlike the commercial products that currently exist. A new method of event driven scheduling has been created that uses fully utilizes this new flow chart interface.

A simple underlying architecture has also been created that separates the core functionality into discrete components. This design has significantly improved the development-testing lifecycle. Additionally, this has allowed custom applications, tailored to the users requirements, to be implemented that use a set of common components, a novel concept in the field of laboratory automation.

A software package, Overlord2, has been produced as part of this work using the latest programming technologies. At its core, it uses an instantly recognisable flow chart interface for assay creation. A scientist with limited programming knowledge can automate, with this software, the most common type of assays carried out in the Drug Discovery process.
Acknowledgements

This thesis could not have been written without the help and assistance of many people around me. First of all, I would like to acknowledge the huge contributions from both my tutors, Dr David Povey and Dr Malcolm Crook, in equal measure for first of all convincing me that doing a PhD was a good idea, and for providing endless help along the way.

I would also like to thank PAA for providing me the opportunity and financially assisting my studies. In particular, I would like to thank my colleagues in the Software/Integration group for their contribution to some of the more complex details surrounding the programming aspect of the project and their work with the final product.

And of course, many thanks to Caroline, my wife for providing endless cups of tea and sparing me from the daily chores around the house, which enabled me to complete this thesis outside of work. A special mention to my two children, who appeared on the scene during the course of this PhD. Despite taking up many hours I could have otherwise devoted to this work, they have provided a much welcome relief to the torture of writing such a long document.
## Glossary

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COM</td>
<td>Component Object Model is a technology that has been developed by Microsoft as a method for different software packages to communicate with each other. ActiveX controls use COM as a method of exposing their functionality to a programmer.</td>
</tr>
<tr>
<td>Thread</td>
<td>Within a program, different threads can be created so that computer code can run concurrently. This has the advantage of running the calling program faster than running all the code sequentially. However, there is greater risk of creating software bugs, especially if the threads use common resources.</td>
</tr>
<tr>
<td>Mutex</td>
<td>A mutex is a thread safe object within the .NET framework that is typically used to synchronize threads. The term mutex is short for &quot;Mutually exclusive&quot; which means that when the thread locks this mutex for its use, other threads cannot use it.</td>
</tr>
<tr>
<td>Deterministic scheduling</td>
<td>A method of scheduling where only the optimal solution is calculated for a scheduling problem. The solution is guaranteed to be the best solution however, the time to calculate is often long</td>
</tr>
<tr>
<td>Heuristic scheduling</td>
<td>A method of scheduling using intelligent decisions to determine the order to process activities within the schedule. The solution is achieved quicker than the deterministic approach but it may not be the optimal solution.</td>
</tr>
<tr>
<td>Programming Type</td>
<td>A method of letting the program know the type of variable that is being used (i.e. string, integer, boolean)</td>
</tr>
</tbody>
</table>
CHAPTER 1 Introduction

Biochemical assays can be complicated protocols that involve multiple sample processing or sample analysis steps. Running these experiments, or any type of experiment manually, is often time consuming and prone to errors. To address this, scientists try to use automation within the laboratory. Laboratory Automation is the technique of making laboratory instruments function automatically without a person required to initiate every action. These laboratory systems can be either a single instrument or a larger integrated system consisting of multiple instruments.

Laboratory Automation is used throughout the pharmaceutical and biotechnology industries to aid research into many biological processes. An example is drug discovery, which is an important component of the research and development (R&D) activities of the pharmaceutical and biotechnology industry.

1.1 Drug Discovery

Research & Development is a major area of interest within pharmaceutical and biotechnology companies. A successful R&D program, which includes drug discovery, is the foundation for the long-term future for any of these companies. The costs for R&D are extremely high, but there is no alternative. Investment has to be made in R&D to find the drugs of tomorrow.

1.1.1 History

Drug Discovery R&D, as a process to discover new drugs, has existed for many years. The first evidence of drug discovery can be traced back to the times of the ancient Greeks. Up to this point, our ancestors would treat minor ailments with readily available herbal remedies. These remedies were selected on a trial and error basis and knowledge would be passed from generation to generation, but these were only used on minor ailments. A serious condition was still treated with magic as the general belief was that of a supernatural explanation to a condition. The ancient Greeks were different, they were renowned for applying a more logical thought process to disease. Greek medicine was entirely different and was much closer to the structured approach found...
today. Remedies, that were predominantly herbal, were selected on the basis that they must address an imbalance in the body, as it must be this that was causing the problem.

Unfortunately, during the dark ages (450AD – 1000AD), the knowledge acquired through the work of the ancient Greeks was lost. In Europe, the Christian Church dominated the way of life and the attitude to any illness or disease was that it was God’s will and it should not be interfered with. Luckily, the Arab World continued the work of the ancient Greeks. It was not until the Middle Ages (1300 AD) that Europe once again started to utilize the knowledge gained by the ancient Greeks.

In the early 19th century, it was possible to start isolating compounds from a plant using solvent extraction. At this point, scientists knew the plant contained the compound in question. This advance allowed compounds to be administered in a controlled manner. Although advances were being made at this time, there was still a reliance on using plants that had been discovered by serendipity.

Throughout the 19th century, organic chemistry progressed to a stage where drugs could be synthesized. At first, this experimental work was being carried out predominantly in universities. However by the turn of the century, there were industrial pharmacologists and chemists beginning to specialize in drug discovery working for the pharmaceutical companies. Many of the current major pharmaceutical companies existed in some form by the end of the 19th century. Some of these companies began life as drug suppliers and apothecaries (pharmacists), such as Schering (now part of Merck), Burroughs Wellcome, SmithKline (now part of GlaxoSmithKline) and Eli Lilly. Other companies such as Pfizer, Ciba, Geigy and Sandoz (now Novartis) and ICI (now AstraZeneca) originated from the production of organic chemicals, in particular dyes.

During the early part of the 20th century, the process of drug discovery was still very much associated with an element of luck. There was a lack of understanding of the therapeutic targets, which is critical to the drug discovery process. Penicillin is the best example of luck-based discovery, accidentally discovered by Alexander Fleming in 1928 whilst studying the Staphylococcus bacterium.

The two World Wars also influenced the progress of drug discovery. Resources were reallocated to work on providing improved painkillers and chemical weapons. There were also unexpected discoveries during this period, for example sulphur mustard was used throughout the First World War as a poison gas. The research into using nitrous
mustard as a more potent weapon also found that it was extremely effective for attacking lymphomas in mice. This would later become an effective drug for chemotherapy. There were other important milestones during this time that were not directly linked to wars, such as the isolation of vitamins and hormones to address deficiency diseases, such as diabetes.

The post war era, from 1945 through to the end of the 1960s, is commonly known as the “therapeutic revolution”\(^3\). After the Second World War, substantial investment was made into drug discovery and this in return led to a rapid increase in new drugs appearing on the scene. Many of today’s popular synthetic drugs such as Ampicillin, the first semi synthetic pencillin (1959), Diazepam, also known as Valium (1962) and Ibuprofen (1964), originate from this time period. By the mid 1970s, the rate of new drugs being discovered slowed. This is thought to be caused by the increased number of regulatory controls being applied, as a result of the thalidomide tragedy and a more structured approach being undertaken to discover drugs: to carefully design drugs based upon the knowledge obtained about the disease being targeted. This trend, the rate of new synthetic drugs coming to market, has not changed even today.

As a quality, as opposed to quantity, tactic was being applied, fewer drugs were appearing, but the drugs reaching market were more effective.

The mid 1970s also saw the rise of the biotechnology companies, a direct result of the increased efforts in the field of genetic engineering. This was primarily a result of the discovery of a recombinant DNA technique in 1973, also known as gene splicing. By using this technique, simple human proteins could be created. Genentech, one of today’s largest Biotech companies and now part of Pfizer, developed human insulin using genetic engineering in 1978. The biotechnology (or biotech) industry grew rapidly in the United States as venture capitalists backed new startup companies, often spin-offs from academia. The rise of biotech has led many of the pharmaceutical companies to adopt the same approach to drug discovery. Some of these companies have set up their own internal biotechnology departments, whereas others have looked to form collaborations with these new biotech companies. This has been mutually beneficial as the pharmaceutical (or pharma) company can provide financial backing and in return can license the new products.

New methods for increasing the size of the drug discovery pipeline have constantly been implemented in an. One of these was Combinatorial chemistry which became extremely
CHAPTER 1

popular during the late 1980s. This was a method of synthesizing a large set of compounds using a base subset of smaller compounds in varying combinations. The logic to this was that a larger array of compounds to screen with would enhance the change of finding a potential compound. To handle the vast number of compounds, new methods of screening were required. This led to the adoption, by many of the pharmaceutical companies, of high throughput screening in an effort to process the much larger compound libraries that now existed. Laboratory automation was born out of necessity so that the “high throughput” aspect could be realized.

The last two decades have provided many new targets attributed to certain diseases which in turn have shaped modern drug discovery. However, the most significant milestone during this time period has been the publication of the Human Genome project in 2003\(^4\). This has provided a better understanding of how certain diseases arise in the human body and therefore a wide range of new targets to address.

1.1.2 Current Drug Discovery

The Research and Development process, found at all pharmaceutical and biotechnology companies contains the following sub-processes:

- **Drug Discovery**: the identification of compounds that are suitable for disease therapy.

- **Early Development**: pharmacological profiling and toxicological testing to ensure the compound exhibits the properties required for a modern day drug.

- **Clinical Trials**: tests on humans to ensure the drug suitability matches the data acquired through development. This stage is made up of Phases that go from Phase I to Phase IV. Once Phase IV has been met, the compound can be registered and sold to market.

![Figure 1 - The Research and Development Process](image)
Drug Discovery covers the initial series of stages that are performed as part of the R&D process. These stages include:

- Target Identification or Discovery
- Hit and Lead Identification or Discovery and Screening
- Lead Optimization
- Preclinical Development (ADME)

**Target Identification** involves the search for proteins that are implicated in a certain disease. Work is then carried out to discover compounds that can address the pathway to prevent disease occurring. Genomic and proteomic sequence data, protein structure data, mRNA and protein expression and metabolic data are used to determine the target pathways for the biochemical process. Work in this area has moved from targeting single proteins to determining the cellular pathways to creating these proteins. This is a more suitable model to utilize, as it closer resembles the complex workings of the human body.

To support this work, bioinformatics is used to analyse the function of a target. This involves the analysis of protein sequences, structures, and interactions. Molecular modelling is often used as part of target identification to simulate interaction networks within the cellular pathways. If a protein is linked to a disease, its 3D structure may provide the information to assist in designing drugs to interfere with the action site of the protein.

Currently, only about 500 gene products are targeted by the drugs available in the market. The publication of the human genome in 2003, that contains an estimated 30,000 to 40,000 genes, has presented an opportunity to advance drug discovery. However, this has simultaneously created a bottleneck regarding the choice of targets to support the drug discovery pipeline. The major advances in genomics and proteomics has meant that discovering a potential target is no longer the problem; the issue is determining the targets that are most likely to be “hits”. The main use of bioinformatics in the process has now shifted from target identification to **Lead Optimization**.

Once targets have been selected, compounds are screened against this target as part of the **Hit and Lead Identification** stage. Screening involves the development of assays to test small molecule compounds for activity against the chosen biological target.
These assays are run against compound libraries that the pharmaceutical or biotechnology companies maintain for this purpose. These libraries have been built up over time with chemical compounds that are most likely to show favourable ADME (adsorption, distribution, metabolism and efficacy) properties.

Any candidates from the **Hit and Lead Identification** screening stage are moved through to **Lead Optimization**. In this stage, the targets that were used in the assay are validated to ensure they contribute to the disease state being targeted. The active or "hit" compounds/proteins are optimized which involves altering the structure of the compound to improve the drug like properties such as activity, efficacy and reducing potential side effects.

When the **Lead Optimization** stage is complete, the compound moves onto the **ADME** (efficacy) stage, where various toxicological and safety pharmacological tests are performed to ensure the candidate compound is safe for Phase I clinical trials. Once the compound enters this stage, the Drug Discovery part of the process is complete.

### 1.2 Laboratory Automation

The basis for the success of any pharmaceutical or biotechnology company is its development and implementation of the best technologies available for drug discovery. New technologies, such as improved detection methods and miniaturization are being developed all the time. These must be adopted in some form to continually improve the drug discovery process, to enable the companies to stay ahead of the competition. The sooner a drug can be brought to market, the sooner it can contribute to the revenue stream of the company. Laboratory automation is widely acknowledged as being a crucial technology for assisting the drug discovery process. It is very unusual to find a pharmaceutical or biotechnology company that does not utilize laboratory automation at some stage of the process.

Laboratory automation is typically used throughout the lead identification and optimization stages. It assists the drug discovery process for a number of reasons:

- Increase sample throughput
- Improve quality control (QC)
- Free resources
- Reduce safety risks.
• Automatic data handling

Increasing throughput has always been a major driving force behind adopting automation within the workplace. This applies to car manufacture or packaging consumer products as it does to drug discovery. Simple logic implies that the more assays that can run in a laboratory, the greater the chance of finding a potential drug candidate. All of the pharmaceutical companies are looking at reducing R&D costs at a time when the time and cost of bringing a drug to market is at an all time high. Although R&D spending has increased, the amount of new drugs coming to market remains low. Laboratory automation provides a method of increasing productivity per scientist.

The use of laboratory automation often ensures an improvement in the quality of experimentation, since automation is far more consistent than a human. For example, if an automated liquid dispenser is instructed to dispense 250µl, it may physically dispense 248µl but it consistently dispenses the same amount to every sample. A human may dispense differing amounts where the average volume is good but with poor precision. Automation also eliminates the need for manual controls such as result verification. Repeatability of automation means that once a process is validated, manual checks are no longer required.

Using automation also decreases the risk to the scientist of exposure to harmful reagents. By using automation to aspirate and dispense harmful reagents, such as DMSO, the health risks are reduced significantly. This can also reduce costs since extra safety measures for manual activities are no longer needed. Volumes of waste toxic products are reduced and additional safety training is not required.

Automating the use of an instrument also involves automation of the data handling that the process generates. Data handling can range from handling very simple text files to complex queries and database manipulations using Laboratory Information Management Systems (LIMS).

1.3 Robotics

Automated systems almost always have a robotic element that moves labware around the system. The term "laboratory automation" has become almost synonymous with robotics since it is the robotic element of the system that allows it to become fully automated. It is common for the word 'robotics' to be associated with laboratory
automation since automating any process often involves a mechanical element that moves items around.

The word 'robotics' was first mentioned by Issac Asimov in a series of short stories published in 1942. Asimov also went on to define what is known as the "Three Laws of Robotics", important in the context of robotics but not really relevant to the types of robotic arms used in the laboratory since the laws focus on the artificial intelligence aspect of robotics. The first industrial arm was used in 1961. It was called the PUMA Unimate and was used to complete repetitive tasks on a General Motors assembly line.

Since then, modern industrial arms have increased in capability and performance but the basic design of robot has remained the same. By the mid-1980's the industry had grown very fast, primarily due to the investment by the automotive industry.

By this time, the first robots had started appearing in the laboratory. In the early 1980s, the Zymark Corporation (now part of Caliper LifeSciences) introduced the Zymate 1 robot which was the first robot made specifically for use in the Laboratory (Figure 3). Soon after, PerkinElmer, Inc started offering the MasterLab which was based on the industrial Mitsubishi Movemaster robot. Although not a success, this was a precursor for the types of system used now.
In 1991, Hewlett Packard, Inc started offering the ORCA robot arm as a track based solution, an automated solution that is still offered by Beckman Coulter, Inc today (Figure 5).

Since the early nineties, many different robots have been used as part of laboratory systems. These can be classed into three different categories.

**Laboratory Robot** – These are either cylindrical arms or SCARA type arms. They are usually driven by stepper motors. These tend to lack power, which is an advantage in terms of safety, as no safety screening is required. This means the robot can simply be placed onto a laboratory bench. The disadvantage is that this can compromise the overall reliability and accuracy of the arm.
**Factory Industrial Robot** - A factory robot is a high powered and accurate device for manipulating materials or hardware items. Originating from the manufacturing industry, where reliability is essential, factory robots are now being used in pharmaceutical laboratories because of their accuracy and reliability.

**Liquid Handler Robot** – These are Cartesian robotic arms that move in the XYZ plane and are integrated within the liquid handler workspace. These extend the capability of liquid handlers, themselves a form of Cartesian robot, to allow them to integrate with other devices. There is a limit to the reach of these arms though, and typically only 1-2 instruments can be attached.

### 1.4 The Microtitre Plate

The microtitre plate has become the standard labware for use in automated systems. The plates are essentially arrays of small test tubes, known as wells, within a set footprint. The number of wells can vary, but the most common formats are 96, 384 and 1536. The greater the number of wells, the smaller the volume of each well. For example, a 96 well plate can typically hold 300μl of sample in liquid form, whereas a 1536 plate can hold 10μl per well. Although automated systems that move tubes are available, the most common labware in use is the microtitre plate.
In the 1990s, the dimensions of these plates could vary between manufacturers. This made it extremely difficult, and therefore costly to implement robotic systems that could handle different plate types. In 2003, the Society of Bimolecular Screening (SBS) released a standard that all plate manufacturers should adhere to, setting out a standard set of dimension for all plates. This has substantially reduced the chance of plate misplacement on automated systems. The biggest challenge now is handling the different heights of plates, as the plates can also come in low volume and deep well variations to the standard height plates. When the lids are used, the matter is further complicated, especially with lids that are a tight fit on the microtitre plate.

1.5 Laboratory Information Management Systems

Laboratory Information Management Systems (LIMS) were initially in-house projects with the objective of more structured systems to store the increasing amounts of data being generated by their laboratories. Companies started appearing during the 1970s that would provide laboratories with custom systems. This was ideal for most laboratories, as they didn’t have the time nor the expertise required to set up their own LIMS. Although each laboratory deals with different data, the processes required for moving raw data to a repository has common elements. Using a commercial product was cheaper than using a custom built system, as common components could be used repeatedly. The commercial products were typically provided by manufacturers, ultimately as a way of improving the sales of their instruments kit, although it wasn’t before long that the LIMS package became the core products for these companies.
It is now commonplace to see a LIMS system in a laboratory and consequently interacting with a laboratory automation system. The majority of laboratory automation solutions are either driven by the data from the LIMS or they generate data for the LIMS. Sample tracking is extremely important and consequently nearly all labware is barcoded as a unique means of identifying samples. It is an essential requirement for automation software to link generated data with the barcodes of the samples before passing onto the LIMS.

1.6 Applications

Laboratory automation is used mainly in the Lead Identification and Lead Optimization parts of the drug discovery R&D process. Lead Identification looks to find compounds that are pharmacologically or biologically active. Scientists develop biochemical assays to test for specific activity that would indicate that the compound either inhibits or stimulates a particular biological pathway.

Automation has been used on many different types of biochemical assays. The most common of these assays is the Enzyme Linked ImmunoSorbant Assay (ELISA). This is a technique for detecting the presence of antibodies in a sample within a microtitre plate. After an incubation period for antigen/antibody binding, the excess or unreacted antibody is washed away. A second enzyme-labelled antibody is added, which will specifically bind to the antigen-antibody complex linked to the microtitre plate. The enzyme reacts with the substrate at the final stage of the assay to produce a detectable colour change that can be measured with a spectrometer. This color change is proportional to the amount of antibody present. The diagram below (Figure 11) shows the steps required for a typical indirect ELISA assay.

![Figure 11 - Indirect ELISA method](image)
The benefits of automation are particularly applicable to the ELISA assay as the assay consists of many liquid addition, wash and incubation steps. Run manually, the assay is time consuming and especially prone to error. There is also an increase in throughput and reproducibility that is to be expected on an automated platform. 

The ELISA assay has been automated in many different ways using just small standalone workstations to large automated systems using a tracked based robotic arm.

Figure 12 – An ELISA Workstation with a plate reader manufactured by Tecan. The workstation consists of a pipetting head (8 tip spreadable probes) and a robotic arm. Within the pipetting work area is a shaking incubator and a plate reader, both accessible by the robotic arm. (Reproduced by permission of Tecan AG - www.tecan.com)

All ELISA based systems feature a liquid handling step which is why single liquid handling workstations such as the Tecan Freedom or the Beckman FX can be used. Workstations are typically equipped with an XYZ arm that enables the movement of the pipetting head. These systems will also integrate a plate washer and a reader as part of the deck space. These extra instruments can be from the same manufacturer as the liquid handler: such is the case with Tecan, or other 3rd party devices (Figure 12).

Systems using a robotic arm mounted on the bench or on a track can use large flexible liquid handlers but generally tend to use small dedicated bulk dispensers for reagent addition. These small bulk dispensers are relatively low cost, so if the reagent...
dispensing is identified as a bottleneck, adding an additional instrument is cost effective. The same applies to any instrument on the system. The extra reach of a robotic arm normally means the system can have increased capacity. A separate hotel or incubation carousel can be added onto a workstation but often there is a space limit on how many instruments can be added.

There is an obvious difference between a workstation and a separate robotic arm approach if the hardware components of the system are compared. However, both systems invariably need to be capable of processing plates through the same ELISA assay steps. The common denominator in an automated system is always what controls the process, which is primarily the role of the control software.

ELISA is a primary screening assay, the parameters for which can change on a week-by-week basis. The automation could originally be configured to use two antibodies. Only a week later, the end users may want to use one antibody as the labware being used has already been pre-treated with antibodies. Automation of any primary screening assay requires flexibility. When any new system is designed, the software is a very important component that allows the system to cope with the demands of primary screening in the drug discovery process. As with any software package, flexibility is proportional to complexity. If there is an increase in the number of parameters required to define an assay, it implies that there there is a steeper learning curve for the end-user.

Software used in the context of laboratory automation is required to initiate actions and then check that the action has been completed on each piece of equipment on the automated system. This sequence of actions or tasks may be driven in a number of different ways that use different algorithms, however the basic functionality of ‘start an action’ then wait for the ‘action to complete’ are the key operations of the software. With more complex software, multiple operations can be run simultaneously leading to parallel tasks on the system, where multiple instruments are operating simultaneously.

The following section reviews existing commercial products and the perceived situation of laboratory automation software within the drug discovery industry. Based upon these observations, the argument will be stated for how this software can be improved and the actions required for such improvements to be implemented.
1.7 Existing Commercial Software

The first evidence of a commercial automation software package was the scheduler software that was supplied with the Zymark system as described in the robotic section (Section 1.3). This software focused almost entirely on the interface between the device and the controlling PC. Since then, the software packages have evolved along with the hardware providing a method for the scientist to fully automate the process using many different types of instruments. The following section describes the commercial automation software currently available and how each software provides laboratory automation control functions. Only two products, Beckman SAMI and Thermo POLARA were available to evaluate. The other commercially available packages have been investigated based upon respective company literature and publications of work using the relevant software.

1.7.1 Beckman Coulter SAMI

SAMI\textsuperscript{16} was originally created by Sagian Inc, who were eventually acquired by Beckman Coulter. The software initially was written to work with third party equipment, but it is now heavily based around either a Beckman pipetting station or the Beckman ORCA arm. The entire application consists of three sections

\begin{itemize}
\item Planning Editor
\item Planning Scheduler/Gantt Viewer
\item Run Schedule.
\end{itemize}

The planning editor is a sequence editor that defines the activities run on the automated system. The scheduler is resource centric in that the process maps out all of the resources that will be used by the labware during the planning stage. Whilst building an assay in the planning editor, the software will assist the user if they try to add an action that is not possible (e.g. putting a lid on a microplate on a Biotek Washer). SAMI can run methods in simulation to ensure the assay is robust before actually running the process. Although it uses a fairly simple to follow workflow interface, this rapidly becomes convoluted for processes that require a large number of steps.
CHAPTER 1

Figure 13 – The Beckman SAMI Planning Editor which is used to define the steps with the assay that is being automated. (Reproduced by permission of Beckman Coulter - www.beckmancoulter.com)

The scheduler section takes the tasks, defined in the planning editor and attempts to schedule these into the shortest period of time. A good feature of SAMI is its ability to use plate incubation times that have a minimum and a maximum incubation time. This is typical of the majority of biochemical assays, so it is good that this flexibility in the assays is utilized as this can help achieve better schedules.

Beckman claim that they have a "sophisticated timing model" that knows the duration of every plate movement, pipetting operation, and device action. In reality, the only way to run a static schedule reliably is by having real timings of each step in the process. At runtime, SAMI can adapt to certain events such as when it acquires data. At this point, the user must ensure that a "Reschedule" action is added into the process. A true dynamic scheduler would automatically reschedule at this point. It seems an unnecessary requirement of the end-user to have to tell the system when to reschedule.

SAMI will automatically track all plate movements and control storage areas such as Plate Stacks and Incubators. This is a good feature as the software can instruct the user
where to load labware and reduce potential issues mid-run. However, this does reduce flexibility as the user may want to control where their plates are stored, especially if the storage area is an incubator that contains plates from multiple assays.

To use SAMI, the user needs to be technically aware to take full advantage of the full flexibility that is offered. SAMI offers a simple interface for developing assays; however, this can become complex for multiple step assays, such as a full ELiSA assay. SAMI has been in the market for many years and as a result, it does appear to be extremely robust. However, this may explain why the core of SAMI has not changed a great deal over the years and has remained a static scheduler. SAMI is a product that will fit the requirements of a software package that deals with primarily a pipetting station with up to three integrated instruments. This is limiting for larger systems, but since this software package is optimized to support the Beckman pipetting station range, this is not an issue for Beckman.

Beckman provide an SDK for developing new instrument drivers called SILAS. By implementing a certain COM interface, SAMI will automatically allow this new driver to be used within the application. Beckman provide a set of templates to assist developing this application with a full set of documentation.

**1.7.2 Tecan EVOware Plus**

Tecan EVOware Plus, previously known as FACTS, was designed to work with the latest Tecan pipetting workstations (Figure 14). It uses a similar method/process editor as SAMI. However, EVOware Plus uses a dynamic scheduler that allows time constraints to be obeyed but will cope with unforeseen events such as errors or instruments finishing before they are scheduled to finish. The software also has extremely good error recovery mechanisms. The software is customized to work with the Tecan products and becomes complicated when 3rd party equipment is added which seems to be typical of software packages that come from suppliers who are primarily instrument manufacturers.
The predecessor to EVOware, known as FACTS, allowed distributed control. This is done by controlling other instances of FACTS running on different PCs. This is a very powerful feature as it allows large automated systems to be broken down into more manageable automation units\(^1\). This has many advantages such as redistributing CPU load to reduce the chance of the control PC crashing, simplifying the method programming and allowing the automation of instruments on the same system that normally cannot be executed from the same PC. This is a common issue especially with integrations that utilize the instruments native software (e.g. plate readers, liquid handlers). Although no evidence could be found regarding distributed control with EVOware, it is assumed that this software includes this functionality as the “Plus” version of the software is based upon its predecessor.
1.7.3 Thermo CRS POLARA

Thermo CRS POLARA is a software package for automating the Thermo CRS range of robotic arms. POLARA is a dynamic scheduler that allows new plates to be added once a run has begun. POLARA does not schedule processes automatically; the user has to specify a pacing time for new plates to be introduced into the system. As with Beckman SAMI, POLARA is divided into a series of sections, although these do all appear as client windows within the core application.

- Editor
- Gantt View
- Run Section

Before an assay method can be created, a workspace followed by a profile has to be defined. Although the instruments physically connected to the PC are defined in setup, a profile has to be defined by the user indicating what instruments will be used. For example, there may be three liquid dispensers physically on the system, but the workspace will indicate that only one of these is being used, as the assay this user will run, will only utilize one of these devices. This seems rather an unnecessarily complicated procedure to create a method. The terminology used as part of this process is focused towards the programmer (e.g. instances, component classes, profiles etc). This terminology will not be familiar to a chemist or biologist, only to experts familiar in this software, which indicates that Thermo believe only trained personal can get this software up and running effectively.

POLARA utilizes a logical and simple to use drag and drop interface for building assays. The assay steps are listed out in a verbose list showing the steps required for a single assay. The user does not need to worry about adding in the robot moves as these are automatically determined by the software. This is a good feature of the software as it reduces the number of steps that the user has to be concerned with. Robot moves have to be defined for this feature to work, and can only work with Thermo CRS robotic arms. The software to program the robot and link this into POLARA is rather complex and again appears to be only something that Thermo or a trained expert could implement.
Once the assay method has been completed, it can be scheduled to see how it will run. The user has to create a schedule template, and then add in the defined method to indicate that this schedule will use this method. The number of samples for this assay method are set, as the method up to this point has been defined for one sample. Each sample can contain multiple containers, such as an assay and reagent container. Once an assay has been added to a schedule it is referred to as a batch. It is possible to have more than one batch as part of a schedule, and each batch can use a different assay method. This is a powerful feature as it allows the user to queue up batches to use on the automated system. This is extremely useful for maximising the use of the system during overnight runs. To get an optimized schedule, the user has to modify certain parameters available on the schedule. This can be achieved by using ‘Sample Stagger’ and ‘Batch Stagger’. ‘Sample Stagger’ allows the user to optimize the sample processing uniformity by specifying a delay the scheduler must insert between each sample. ‘Batch Stagger’ enables one to optimize batch processing uniformity by specifying a delay the scheduler must insert between batches as POLARA will attempt to interleave batches if feasible. The stagger functionality allows optimized schedules to be
realised: however, once again the user is required to fully understand the software package to use it to its full potential. The concept of staggers and how that results in the best schedule is a concept that only experts will be able to appreciate.

Figure 16 – The POLARA Schedule setup allows the user to modify sample staggers and batch staggers in order to find the optimal schedule for processing the assay.

Once the schedule is set, the batches within it can be run. POLARA displays a window showing where all the labware on the system should be placed. POLARA is fairly rigid in demanding where labware should be placed. This does not cater for scenarios where the user loads a stack of plates into a certain position and wants to tell the software where the plates are located. This is actually a common scenario, especially on open access systems where many users run their plates through the system, and this is a weakness of POLARA.

At a low level, POLARA controls the devices on the system using drivers written in RAPL3, a propriety programming language from CRS that is "C" based. This runs on CROSnnt, a UNIX type operating system that has to be installed on the PC. This is
automatically installed as part of the POLARA installation process. Every time a new
device is installed on a POLARA system, a series of files are required: a COM DLL for
the edit screens in POLARA, and the RAPL3 files for controlling the device at runtime.
This entire process of installing new devices within POLARA seems extremely
complicated. It is quite evident that the software is an evolution from a time where the
system would run on a UNIX system. Although this brings the advantage of using an
established code base, the architecture appears to have become extremely unwieldy.
From a programmer's perspective, writing a driver to use in POLARA is extremely
cumbersome. Although most programmers have the expertise to write the Edit
components, which are COM based in a language such as C++ or VB6, the RAPL3
language will be a new language that needs to be learnt. It is noted that Thermo CRS
provide excellent documentation and code examples for using their Software
Development Kit, however this does not mask the complexity of the architecture.

In summary, POLARA is simple enough to use once the system has been setup by an
expert. However, the scheduling parameters that the user needs to adjust seem an
unnecessary step for the user to comprehend. The software is based upon an old
architecture that dates back to the mid 1990's, and this has been restrictive with regards
to installing existing drivers and writing new ones.

1.7.4 Velocity11 VWorks

Velocity11 VWorks\textsuperscript{20} is different to the other schedulers in that it is event-driven with a
large focus on efficient processing and error handling (Figure 17). It is used in the
industry predominantly for running Velocity11 equipment; however it can be used with 3\textsuperscript{rd}
d party equipment.
The editor uses a process flow that is resource centric, similar to the interface used by the Beckman SAMI software. The VWorks software uses a multi-document interface, as found in many Microsoft applications such as Outlook and Word. This provides a single window to the end user, with many smaller dockable windows. This is a much better approach to having many different sections of the software appear in separate windows, as there is just then a single window for the end user to become familiar with. However, the VWorks main screen does display a lot of information simultaneously, which could be confusing and disconcerting to the novice user.

1.7.5 Caliper LifeSciences iLink Pro

Caliper LifeSciences iLink Pro, formerly CLARA, was written originally by SCITEC before Zymark acquired the company in 1999. Caliper LifeSciences then acquired Zymark in 2003. iLink Pro is a dynamic scheduler that uses a text based method editor. It uses a Gantt chart for schedule visualization much like SAMI and in addition a similar runtime interface (Figure 18). CLARA has powerful interface capabilities for writing 'adaptors', or instrument software interfaces.
1.8 Software Customization

Literature regarding the quality of the software used for laboratory automation is sparse. Recent reviews of the industry have focused on the issues with integrating systems of more than one instrument. Communication between the instruments from different manufacturers is still seen as challenging as is sample tracking and data interactions with LIMS. It is also noted that the type of complex biochemical assays being automated coupled with the hardware selected reduces flexibility\textsuperscript{21}.

It appears that approximately half of laboratories using automation are using off the shelf software products to control the automation. An attractive alternative appears to be modifying or developing add on features to existing products, which accounts for approximately 40\% of installations. There are also a significant number of users developing their own solutions, although this appears to be becoming less common\textsuperscript{22}. The requirement for customization is significant. It is fair to assume that the end-users
that are using off-the-shelf packages are performing simple tasks such as moving a stack of plates through a plate reader or liquid handling device.

What must be studied is how assays are defined in the real world. The literature implies there are two predominant methods of describing an assay, text-based instructions and by process flowchart (Table 1). The example shows how the two methods would describe washing and then adding reagent to 20 different microtitre plates (MTP).

<table>
<thead>
<tr>
<th>Text Based Instructions</th>
<th>Process Flowchart</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Wash MTP Plate</td>
<td><img src="image" alt="Flowchart Diagram" /></td>
</tr>
<tr>
<td>2. Add Reagent</td>
<td>Loop 20 Times</td>
</tr>
<tr>
<td>3. Repeat Steps 1-2 for 19 more plates</td>
<td>Wash MTP</td>
</tr>
<tr>
<td></td>
<td>Add Reagent</td>
</tr>
</tbody>
</table>

*Table 1 – There are two main ways of describing a process, either by text or by flowchart. Both examples in this case are describing the same process.*

Both of these examples have been used in some form as user interfaces. The text-based approach has recently been used as a way of enhancing the Beckman SAMI interface. Kaber et al took a cookbook metaphor approach, which is a similar approach to that which this project is looking to achieve. In this case, the cookbook approach was taken which is very similar to the text-based instruction method. The study states there was a significant improvement over assay development time using the cookbook metaphor interface as opposed to the SAMI interface (See 1.7.1 Beckman Coulter SAMI). Even though the SAMI software is dedicated to using the Beckman equipment, in this case an ORCA robotic arm, the assay development time was significantly improved from a mean time of 254.2 seconds with SAMI to 72.9 with this new interface.

It is interesting to note that the interface developed by Kaber et al converts the text into a process flowchart for a clearer overview of the assay. The author also states "Flow charts of recipe preparation procedures can be found in many basic cookbooks". The flowchart is used in many fields for describing a process and is instantly recognizable to the majority, especially scientists. Flowcharts are used throughout the literature as a...
way of representing the steps within an assay\textsuperscript{24} and have been used by other vendors as a method of defining the assay steps\textsuperscript{25}.

1.9 Thesis Statement

It appears to be usual for automated systems to have a super user who will write the assays for the system. Experience has led groups to create this role for the laboratory systems they have installed. It is recommended to have a dedicated individual who is trained to service the instruments or modify software parameters. However in reality, it is impractical to have only one person who can write assays on the system. The aim of this project is to provide a software package that can be used by all chemists and biologists. This software should encourage staff to use the automation instead of running their assays manually. Clearly, with the available software, this is not the case.

All of the commercial software packages that have been reviewed appear to deliver what is intended: they would not sell licences otherwise and each package has been successful in its own right. However, there are shortcomings that must be addressed if laboratory automation is to be used by all research scientists within the drug discovery process.

The main issue is the continual need for an expert or the software vendor to setup and configure the laboratory automation software. In many cases, this includes developing the methods to execute an assay. It has been stated previously that the primary goal for automation suppliers is to simplify the process of automating an assay, without continual expert support from the supplier\textsuperscript{26}.

There are a number of reasons why this scenario has arisen:

- The graphical representation of the assay, especially for multiple-step biochemical heterogeneous assays, such as ELISA, rapidly becomes convoluted and extremely difficult to follow.
- The graphical user interface is a major factor in the acceptance by customers. Laboratory automation software to date has been complicated by either showing multiple windows simultaneously or using a multi document interface with vast amounts of data on screen at the same time.
- A common theme of all the software reviewed was the terminology which consistently used words found throughout the programming community. These
terms are not familiar to the chemist or biologist in the laboratory and only add a
degree of unease when using the respective software package.

To address some of these concerns, vendors commonly customize the software to fit in
with the workflows at their respective companies. Although the hope is to tackle these
points so the need for customization is reduced, a novel approach to the software
development kit is required. This customization will involve modifying the graphical user
interface and instrument drivers.

It is also essential to consider the interface to other software components used within the
drug discovery process. Although there is a trend for creating small workcells that
perform actions for a subset of the assay\textsuperscript{27}, it is still important for the data to be shared
and for the execution of the assays to be controlled by higher-level software\textsuperscript{28}. The
architecture of the product should be designed in a manner that would allow this to
happen.

A software package will be developed that addresses these issues. The best method of
scheduling will be determined, along with the most suitable development tools to realize
the end product, before developing the software package through the common software
project lifecycle process.

The work has been carried out in association with Process Analysis & Automation Ltd
\textsc{(paa)}.

All manufacturers' trademarks are gratefully acknowledged.
CHAPTER 2 Scheduling

Scheduling is an important component of a laboratory automation software package. The benefit of using laboratory automation is that multiple plates within an experiment or assay can be processed automatically.

To improve efficiency, the schedule must allow multiple plates to run at the same time so the execution is interleaved, which then maximizes throughput. A scientist running a set of plates manually through an assay would run multiple plates at the same time to ensure the assay can be completed within the working day. An automated system should be no different. This leads to a level of complexity that the user has to understand to fix any issues that may occur during a run.

An experiment or assay can be run by breaking it down into a series of tasks. Linking these tasks together completes the experiments. An experiment in its simplest form would be a single task, which for example dispenses 100µl of reagent into a microtitre plate. Gantt charts are commonly used to visualise how tasks in the assay will execute. The chart below shows how this example will process.

<table>
<thead>
<tr>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

Plate 1

![Gantt chart showing task execution](image)

**Figure 19** - Single microtitre plate, single task schedule

There is only one possible way of executing this task. There is no need to consider the different ways of executing this task. If we want to run 5 microtitre plates and dispense 100µl of reagent to each plate, there is still only one possible way of executing the set of tasks. Each task uses the same resource; in this case a liquid dispenser which means the only way of processing the single task for each microtitre plate is one after the other.
Once again there is no need to consider the different ways of executing this task. Automation of this process, running a single task on each microtitre plates, is common. However, scheduling software is not required due to the simplicity of the experimental process. The only way to increase the efficiency of this process is to have multiple instruments. It is only when more than one task is run against each microtitre plate that different ways of execution become apparent. To demonstrate this, take the following relatively simple experiment that contains 2 tasks. In this example, 3 microtitre plates will be run.

- Wash microtitre plate (duration 180 seconds)
- Dispense reagent onto the microtitre plate (duration 60 seconds).

Assuming that each plate will be started in order, there are two ways that this experiment can be run.

Figure 20 - Four microtitre plate, single task schedule
The schedule shown in Figure 21 is a feasible solution. The cycle time, the time difference between starting each plate is 240 seconds. Assuming there are no delays between starting each task, this is also the worst possible solution as the schedule is not utilizing instrument idle time.

The interleaving schedule, as shown in Figure 22, which includes overlapping tasks, will run in a shorter amount of time than the schedule that is running one plate after the other. As soon as the wash task has completed on Plate 1, Plate 2 begins. This reduces the cycle time to 180 seconds. It is desirable to run these series of tasks in the quickest amount of time as it improves the throughput. Throughput is the measure of how quickly micotitre plates or the samples within the micotitre plates can be processed through the experiment. With the 2 plate example, we will get a 60 second time saving. If the automated assay was to use 50 plates, 50 minutes would be saved using the overlapping schedule. Since the entire time to run the assay, also known as the makespan, for the schedule is 152 minutes using the optimised schedule, a 50 min time saving is significant and desirable.

From a scheduling perspective, running each plate, one at a time without any overlapping is a simple process. Each task is run in a sequential manner, simply waiting for the previous task to complete. For the overlapping schedule, we need to take into account that the other plate may be using a resource that we plan to use. Scheduling
within automation essentially is the problem of allocating resources to tasks over hard set and variable time windows. This is a complex operation.\textsuperscript{26}

It is important to adopt a form of scheduling to ensure the required resource is available for a task when the task needs it whilst obeying any time constraints such as Incubation times. The objective is to complete the series of tasks for all plates on the system in the shortest amount of time. To do this, there are three methods of scheduling that can be used, Static, Dynamic and Event driven.

\textbf{2.1 Static Scheduling}

Static scheduling takes the tasks required to run the assays and prepares a pre-emptive plan on how and when to execute each task. By knowing how long each task takes to complete, the scheduling algorithm can, using either a deterministic or heuristic approach, calculate an optimal or near optimal path for executing the tasks in the shortest amount of time.

For the overlapping schedule that was described previously, a static schedule would calculate the start time of each task based upon the time taken for each task. It would come up with the following schedule.

<table>
<thead>
<tr>
<th>Task</th>
<th>Resource</th>
<th>Start Time</th>
<th>Task Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plate 1 – Task 1</td>
<td>Washer</td>
<td>0</td>
<td>120</td>
</tr>
<tr>
<td>Plate 2 – Task 1</td>
<td>Washer</td>
<td>120</td>
<td>120</td>
</tr>
<tr>
<td>Plate 1 – Task 2</td>
<td>Dispenser</td>
<td>120</td>
<td>60</td>
</tr>
<tr>
<td>Plate 2 – Task 2</td>
<td>Dispenser</td>
<td>240</td>
<td>60</td>
</tr>
</tbody>
</table>

\textit{Table 2 – The static schedule timings that would be calculated by a static scheduling algorithm using the example in Figure 22}

To simplify the scheduling requirements, it is acceptable to schedule each plate in the order they are listed. In the example above, the start times for the first plates are calculated first. This sets each task running after the previous task has completed. In reality, on larger experiments, it is sometimes necessary to add wait times between tasks to ensure resources are free at a specific time. This is only determined once the scheduling algorithm has been run. However, for the purpose of these examples, it will be assumed that the tasks for each plate will execute without any delay between each task.
To determine a static schedule, complex algorithms have to be used as there are so many scheduling permutations that have to be evaluated to come up with an ideal scheduling solution. The problem presented in scheduling a laboratory automation process is very similar to what is known as the Job Shop problem. A Job Shop looks how to run a series of operations belonging to a Job through a series of machines. The problem arises due to there being multiple Jobs which means each operation has to be scheduled on the relevant machine whilst obeying the processing order of the operations. These type of schedules are widely acknowledged to be one of the most important and difficult set of problems in scheduling\textsuperscript{30}. The problem is known as NP hard (non-deterministic polynomial-time hard) which means that the time required computing a definitive solution to the Job Shop problem is extremely large. If we have $x$ tasks in total, $x!$ permutations would need to be evaluated to find the best task to run. Once this is calculated, $(x-1)!$ permutations would need evaluating, then $x-2$, $x-3$ until $x = 2$ so that all tasks have been evaluated. Such a method is not a problem if $x$ is less than 5, but if we have 10 tasks, which is still a small schedule, we would be looking at evaluating 3,628,800 scheduling permutations.

An absolute solution is practically impossible for reasonable sized assay schedules (e.g. 10 or more plates with 10 tasks). Different approaches have been made to provide near on optimal solutions to the Job Shop problem using search approaches\textsuperscript{31}. These methods involve finding a solution and then applying intelligent decisions to improve the solution until an acceptable solution is discovered. The solution in this case is not necessarily the best solution; however it will still be a good solution which is a satisfactory result.

The Job Shop approach for creating static schedules has been used in existing laboratory automation software packages, such as Beckman™ SAMI\textsuperscript{32}. A different approach using Discrete Event Systems (DES) modeling has also been used for the Cybio™ Scheduler scheduling software, which is a commercially available software package aimed at working with the Cybio instrument product range\textsuperscript{33}. A model representing the operation or tasks to perform on one plate is created. The parameters for each task, the start time and the completion time are defined as a set of equations. To ensure scheduling does not become NP-hard, the search space is reduced by simplifying the model. The resulting equations can then be used within a Solver program such as the Microsoft Foundation Solver\textsuperscript{34} or GAMS\textsuperscript{35} to provide the optimal plate cycle.
time. What differentiates this work from others is the way the equations formed allow there to be a variable time delay time within certain tasks. This delay time allows the algorithm to adjust this delay time to find the optimal schedule. This delay time is relevant to a practical application such as an ELISA assay as incubation times can often tolerate a 1-2 minute difference in incubation time to what was stated (e.g. Incubate for 30 minutes ± 2 minutes). Utilizing this delay time can ensure that the plate cycle time, the delay time between each plate starting or completing is reduced, therefore increasing system throughput. The DES approach is limited by the fact that each plate must run exactly the same activities. In some cases this is not a problem, however it is common to see a schedule where every “x” number of plates, a specific task has to be performed such as recalibrating an instrument to ensure quality of results. The DES approach should not be used in this case.

The main strength of a static scheduler is that it will process each plate in exactly the same way, unless there was an error on the system. As resources are pre-emptively allocated, the execution of tasks can be within a second of the allocated times. Incubation times can run for the exact specified time, which for some cell-based assays is extremely important. Static schedulers also provide a method for users to see how long their assay will run. It allows them to see the effect of modifying task parameters and what effect there will be adding extra devices on the system. For example, maybe the pipetting station is the rate limiting step on the system and the inclusion of an additional instrument will increase the sample throughput. Using a software package that utilizes a static scheduler, it is very simple to see how throughput can be improved by adding on an extra device.

Due to their simplicity, static schedulers do lack flexibility. It is impossible for the assay to react to “events” that happen during a run so that the experiment moves down an alternate route. For example, if after reading a plate it is determined that all of the samples in the assay must be incubated for a longer period, it is not possible with a static scheduler to change task durations mid run. There is also the disadvantage that if a device running a task goes into an error state, it is impossible for the next tasks to be executed until this error state has been addressed, typically by the user. A static schedule cannot deal with such scenarios and often struggles to deal with the time delay generated by this situation, often relying upon the user to remove the tasks that have slipped from the original time schedule.
2.2 Dynamic Scheduling

Dynamic scheduling is an evolution of static scheduling. The advantages of static scheduling are retained however, with a dynamic schedule it is possible for the schedule to deal with “events” that effect the task durations. If a task has been scheduled to run for 10 minutes but takes 5 minutes instead, a dynamic scheduler can use this time gained and start the execution of the next task in the assay. This is because dynamic schedulers can reschedule “on the fly” at runtime.

There is even more need for the scheduling algorithms, described in Section 2.1, to run as fast as possible. In such a scenario, the rescheduling is run in the background to the running processes on a separate process or thread to ensure any rescheduling does not delay the running of instruments. A potential problem arises in that the tasks that need rescheduling will have changed once the rescheduling completes as during this time, certain tasks may have started and possibly completed. The rescheduling needs to consider this. However, this can lead to undesirable schedules where in the first part of the new schedule, tasks are not starting optimally as contingency has to be put in place to accommodate the time the dynamic schedule took to run. The alternative is to have the system wait for the scheduling to complete to ensure synchronization. Of course, with this the system is paused whilst the new schedule is being calculated. Depending upon the size of the assay, and the frequency of rescheduling, these delays can become unacceptable to the end user since throughput will suffer.

2.3 Event Driven Scheduling

Also known as round robin scheduling, this is the method of running an assay when events occur such as device state change. The user instructs the system how to move the plates through the different instruments on the system based upon the implemented rules. The advantage of such a scheduler is that as soon as a task completes, the samples can be moved onto the next stage of the assay immediately. It is irrelevant if a task is quicker or takes longer to run than originally estimated, the assay will continue to run without any need to run a scheduling algorithm. This also allows decisions to be made at runtime without any adverse effect to the entire system. The decision to increase incubation times as described in the static scheduling section is easily handled in an event driven configuration. Due to the increased flexibility of event driven
scheduling, the additional options available in the software means it is also more complicated to use than a static or dynamic scheduler. The steps required to define an assay require more input from the user and even the most basic rules on what to do when an event occurs requires a level of understanding of the process that is greater than that required for a static scheduler.

The same schedule described in Table 2 can be set up to run in event driven mode. The following rules can be used if the process is divided into the following three stages.

<table>
<thead>
<tr>
<th>Stage</th>
<th>Activity</th>
<th>Rules</th>
</tr>
</thead>
</table>
| 1     | Load Plate onto Washer and start running (will run for 180s) | Washer instrument is idle  
Washer has no plate loaded |
| 2     | When washer has finished, move plate to dispenser and start running | Washer is idle  
Washer has a plate loaded  
Dispenser is idle  
Dispenser has no plate loaded |
| 3     | When dispenser has finished unload plate | Dispenser is idle  
Dispenser has a plate loaded |

Table 3 – The event driven rules that would be generated to process the plates in an interleaved manner as shown in Figure 22

To run x number of plates through the schedule, each stage must run x times. Using the 2 plate example previously described, we would expect the stages to run in the following sequence for each plate.

<table>
<thead>
<tr>
<th>Event</th>
<th>Stage</th>
<th>Time of Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Plate 1 – Stage 1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>Plate 1 – Stage 2</td>
<td>120</td>
</tr>
<tr>
<td>3</td>
<td>Plate 2 – Stage 1</td>
<td>120</td>
</tr>
<tr>
<td>4</td>
<td>Plate 1 – Stage 3</td>
<td>180</td>
</tr>
<tr>
<td>5</td>
<td>Plate 2 – Stage 2</td>
<td>240</td>
</tr>
<tr>
<td>6</td>
<td>Plate 2 – Stage 3</td>
<td>300</td>
</tr>
</tbody>
</table>

Table 4 – The times each stage, as described in Table 4, will start in response to an event (i.e. stage or device completing)

As soon as Plate 1 has been moved from the washer to the dispenser, it means the washer is ready to receive the next plate. This pattern then repeats until both plates have been processed through the entire system.
This method of running an automated system is scalable and can run assays, such as ELISA assays with many steps and incubations by simply using more stages. With more complicated assays, there is the possibility for the process to become deadlocked. If each plate only goes to an instrument once, this will never be a problem. However, if the plate can return to a device, a deadlock scenario can arise. Taking the example used in Table 2, if we add in an extra washer step after the plate has been on the dispenser, we have a potential deadlock situation.

If for example Plate 2 is on the dispenser and Plate 1 is on the washer the system is deadlocked. This is because Plate 2 cannot move back to the washer as Plate 1 is occupying it. Plate 1 cannot be moved to solve this conflict as this in turn would need to be moved to the washer, but this is occupied by Plate 2 (See Figure 23).

This deadlock scenario can be resolved by controlling the addition of new plates into the system. In this example, plate 2 would not be introduced to the system until Plate 1 has been used on the washer for the 2\textsuperscript{nd} time. This then ensures the dispenser is free for Plate 2 once it has completed the first washer step, thereby eliminating the deadlock scenario. This solution does slow down the throughput of the system as the degree of concurrency is reduced to avoid resource conflicts. Alternatively, relief nests can be used to empty the conflicting instrument so that the assay can continue. This keeps throughput for the system as high as possible as simultaneous plates can be processed.
although the system will require extra steps to be implemented. The deadlock scenarios can be detected by the software by using Petri nets, which effectively simulates the full process before running it\textsuperscript{36} This eliminates one of the disadvantages of an event driven system whereby it is only known how the system will behave after it has been run.

2.4 Scheduling Choice

The choice of scheduler is not that important to the scientist using the software package for laboratory automation. As long as the software processes each plate through the assay obeying any constraints set, such as incubation time and associated tolerances, the type of scheduler, static, dynamic or event is irrelevant. The reality is that each scheduler is suited to specific assays. A cell based assay that requires exact incubations for compound contact time with the cell requires a static scheduler. This can ensure the resources are available pre-emptively, to avoid the scenario where an incubation step completes, but the robot is busy with another activity. A compound reformatting system that cherry picks a sample may only find out the "picks" at runtime. This type of process will require an event-driven scheduler. For a cherry picking assay, a worklist is supplied specifying barcodes of the source plate.

The choice of scheduler should be one that is a best fit for all types of assay. Based upon the analysis of each scheduling type, an event driven schedule will be adopted within this project. The event driven schedule has the following benefits:

- The assay method can branch into a different process based upon acquired data or some other event.
- Accurate timings of each task do not need to be maintained to schedule the process.
- The system can easily handle instruments taking different amounts of time to run on a day to day basis. This is actually a common occurrence as external factors such as room temperature and physical sample properties (i.e. viscosity, volume) can affect unit operation durations.
- System redundancy can be handled if an instrument breaks down mid-run.
- Device pooling can be handled where multiple instruments of the same type can be treated as one device and the automated system just uses whichever device is free.
It is acknowledged that more effort will be required by the end user to setup a series of rules for exact incubations. However this is a minor inconvenience when taking the benefits into account which justify this selection of scheduling.
CHAPTER 3 Coding Language

The programming language choice for any software project is important. Many points need to be considered such as:

- Does the software provide the functionality required?
- How familiar are the programming team with the language syntax?
- Support for the language.

The software team at paa has primarily used Visual Basic 6 (VB6) as the language to develop software projects. Microsoft Visual Basic (VB) has proved to be a popular programming language, especially in business programming. Take up by end users within the laboratory automation industry appears to be fairly high. This is because of the familiarity many people have with VB for Applications (VBA) through using other Microsoft products such as Excel and Word. The easy to use syntax enables rapid application development (RAD) that has made it suitable for creating small programs that are frequently required in the laboratory.

VB6, however, is now an old product that is no longer supported by Microsoft. It was replaced back in 2002 when Microsoft first introduced Visual Basic .NET. The version available during this period of work was known as Visual Basic 2005. It is a programming language that utilizes the Microsoft .NET Framework, version 2.0 in this case. It is a new form of Windows API that is much more extensive and is now fully Object Oriented (OO) and is a major change to the architecture of Visual Basic. This is because the .NET Framework is substantially more advanced in terms of objects and components compared to the Component Object Model (COM), on which VB was previously based.

Since the language is based upon the Microsoft .NET framework, it is limited to only creating applications that run on Windows. However, the industry predominantly uses the Microsoft Windows operating system with some academic institutions perhaps being the only exception.

Other languages have been used to develop software packages for laboratory automation such as C++ and Ruby. It is noted also that C#, a new language from
Microsoft based upon C, is a viable choice since this also uses the .NET Framework. If the choice is based upon functionality, there are no major differences between VB and C#. However, the group that will be developing the software has had the most experience using Visual Basic. It is therefore logical to select Visual Basic.NET (VB.NET) as the preferred choice, as it reduces the risk of missing shipping dates due to unfamiliarity with the coding syntax.

It was important to ensure that after all the hype surrounding VB.NET, it did offer all of the key functionality required for this new software package. A thorough review of Visual Basic 2005 that uses the .NET Framework 2.0, was performed to ensure that primarily it was now, truly, an Object Oriented language and that threading was possible. However, other functionality was reviewed to ensure this language choice would enable the realisation of the design and architecture of the final product.

An important requirement for the software will be the ability to run tasks asynchronously. Threading is now supported in VB.NET, something that the previous version could not do in a safe manner.

3.1 Object Orientation

Visual Basic .NET represents a significant shift from an essentially procedural-based language (VB6) to one that demands an understanding of Object Oriented (OO) programming concepts. Although it is still possible to create applications for the laboratory that hardly make any use of objects, a thorough understanding of the .NET framework is essential to exploit fully the language.

VB4 was the first version that allowed use of classes in applications, a major improvement at the time. Many people then proclaimed that VB was an OO language. If the standard definition of an OO language is followed, a key requirement of an OO language is Inheritance: however this was not available in VB4. Inheritance is now a key feature of VB.NET. All of the other OO features, such as Abstraction, Polymorphism and Encapsulation are present, as they were in VB6 although they are implemented slightly differently as shown in the examples below.

In Code Example 1, there are two classes ThermoReader and TecanReader. Both are inheriting from the base class PlateReader which has a Read method. The Base class is an abstraction of a Plate Reader, the ability to write a class that represents something.
Public MustInherit Class PlateReader
  Public MustOverride Sub Read()
End Class

Public Class ThermoReader
  Inherits PlateReader
  Public Overrides Sub Read()
    'Code here to run this reader
  End Sub
End Class

Public Class TecanReader
  Inherits PlateReader
  Public Overrides Sub Read()
    'Code here to run this reader
  End Sub
End Class

--- Code Example 1 -- Abstraction and Inheritance

Notice in this example how the base class is constructed. The keyword MustOverride is used for the method Read in the PlateReader base class. When ThermoReader inherits the base class it must implement its own version of the method Read as indicated by the Overrides keyword preceding it.

As both classes are inheriting from the same base class, a method can be created that can run either type of reader, irrespective of the fact that each object is created from a different class, a concept known as Polymorphism.

Dim objThermo As New ThermoReader
Dim objTecan As New TecanReader

RunReader objThermo
RunReader objTecan

Public Sub RunReader(ByVal Reader As PlateReader)
  Reader.Read()
End Sub
CHAPTER 3

**Code Example 2 - Polymorphism**

The ThermoReader class can be extended to include a property that represents where generated data will be stored when a Read has completed even though this property is not in the base class (See Code Example 3).

```vbnet
Public Class ThermoReader
    Inherits PlateReader
    Private mDataPath As String
    Public Overrides Sub Read()
        'Code here to run this reader
        'Saves data to path specified in 'mDataPath'
        End Sub
    Public Property DataPath() As String
        Get
            Return mDataPath
        End Get
        Set(ByVal value As String)
            mDataPath = value
        End Set
    End Property
End Class
```

**Code Example 3 - Encapsulation**

Every object that is instantiated from the ThermoReader class will contain the property DataPath and the value of this property will always be contained within the object. Wherever the object is available, the value of the property DataPath will also be available as it is encapsulated within the object.

### 3.2 Types

VB.NET is very flexible when working with types as it can behave as both a static and dynamic typed language. Turning on Option Strict in VB.NET by entering the keyword at the top of each class or as a global setting for the project and it becomes statically typed. If this option is off, the language behaves as a dynamic typed language. This choice allows the programmer to decide what level of type checking they want at compile-time. If Option Strict is turned on, type errors are highlighted at edit time. For most work, Option Strict should be turned on, to reduce type software bugs. However,
static typing can be restrictive and there are times when dynamic typing makes life easier. An example is when working with COM (Component Object Model) on which VB6 is based. If Option Strict is turned on, late binding is not possible.

All of the types that were included in VB6, apart from Variants and Currency, are present in VB.NET although they are different in structure. For example, the Integer type is now 32 bit and not 16 bit, plus there is support for unsigned Integers. Essentially, everything is an Object, so it is still possible to work with unknown types if required. This replaces Variants in the previous versions of VB. The problem with using Object for unknown types of data is that it is very prone to error as assumptions are made that the data is of a certain type. VB.NET has also introduced Generics to address this issue. Basically, code can be written to handle types before knowing what the type will be. The example below is a very simple class that behaves as a collection.

```vbnet
Public Class GenericCollection(Of itemType)
    Private mCol As New Collection
    Public Sub Add(ByVal item As itemType)
        mCol.Add(item)
    End Sub
    Default Public Readonly Property Item(ByVal Index As Integer) As itemType
        Get
            Return DirectCast(mCol(Index), itemType)
        End Get
    End Property
End Class
```

**Code Example 4 - Generic Collection**

Instead of referring to an explicit type, the placeholder `itemType` is used. When creating an instance of this class, `itemType` is set as a type as shown in the example below that is instantiating the class to handle Strings.

```vbnet
Dim colGen As New GenericCollection(Of String)
```

In Code Example 4, the function `DirectCast` is used. It is strong typed and ensures that the data is or inherits from the expected type. There is also the `CType` function that will convert data to a different type. This is weaker typed than `DirectCast` and useful for
converting for example a String to a numeric type, although the runtime performance is slower.

3.3 Operating Systems

Since the language is based upon the Microsoft .NET framework, it is limited to only creating applications that run on Windows. Web applications are a slight exception, in that they support cross-platform clients, but deployment does require a Windows Web server. The .NET framework is also available in a 64-bit variation. All of the functionality in the 32-bit version of the framework is present; the difference lies behind the scenes in the CLR (Common Language Runtime). There is no real noticeable difference in working in both environments. More often than not, programs compiled in a 32-bit OS (Operating System) such as Windows XP can be run on a 64-bit OS. As well as desktop applications, programs can be written to run on a PDA or Mobile Phone by making use of the .NET Compact Framework, a subset of the main Framework.

3.4 GUI Development Tools

One of the major factors in the take up of any language is how good the IDE (Integrated Design Environment) is. The IDE for Visual Basic .NET 2005 is used by all the languages that are part of Microsoft Visual Studio 2005, these are Visual C#, Visual C++ and Visual J#.
CHAPTER 3

Figure 24 – The Microsoft Visual Studio 2005 IDE that allows VB.NET, C# and C++ window applications to be created in a single development environment.

The IDE has form designers for Windows Forms, traditional Windows applications and Web Forms, using ASP.NET pages. Designing and coding Windows Forms and Web Forms has been made as similar as possible so that both types of applications can share components and much of the complexity of coding Web applications is kept hidden. The support for writing code is a major advance in the IDE. When writing code, suggestions are made and syntax errors and warnings are highlighted. There is a library of reusable code snippets that can be dragged right into the code window which speeds development. It is also possible to define and reuse your own code snippets.

Debugging has always been well supported in Visual Basic and this has been enhanced. One useful addition is the help dialog that will be displayed with trouble-shooting tips and recommendations on what changes need to be made when an error is encountered in the code. All code can be edited at this point and execution resumed. Also included are Tracepoints which are breakpoints that don’t break. Instead, they can report the current program status such as a variable value and let the code continue. This is very useful

45
for programs where having a break in the code would create an unrealistic testing environment such as when waiting for a response from an instrument.

However, it is the new features and tighter integration to data that really make the difference over previous versions. For example, it is now possible to test the methods and properties of a class using the Object Test Bench. An instance of the class can be tested without having to write a test harness for all but the most complicated classes. This is particularly useful for applications that work with instrumentation as each device can be tested, assuming a class has been written for each, before testing the entire program. Although this feature would not be high on the list on why one should use Microsoft Visual Studio, it is these additions to the IDE that makes it a simpler process to create a program.

Visual Studio 2005 is available in various editions, Express, Standard, Professional, Tools for Office, and Team System\textsuperscript{44}. The \textit{Express Edition} which is free to download from the Microsoft website is a cut-down version targeted at novice developers and is an ideal choice for completing a basic evaluation of what the product has to offer. The high end products are the \textit{Team System Editions}. These are supplied individually as \textit{Software Architect}, \textit{Software Programmer} and \textit{Software Tester} packages or the \textit{Team Edition} which is a package of all three versions. The \textit{Team System Editions} are primarily for large development organizations and are not really suitable for writing the types of applications found in the laboratory. The most suitable choice is the \textit{Standard} edition, which is the cheapest version available that has all the functionality required to write a wide range of applications. This version also comes with the Microsoft Developers Network (MSDN) help library and is a very useful resource.

\section{Multi-Tasking}

There are two ways of running code asynchronously in VB.NET, using Delegates\textsuperscript{45} or Threads\textsuperscript{46}. Most Visual Basic programmers will be familiar with the concept of Events since it is an event driven language. For example, when a button is clicked, a click event is raised. Events can be raised from inside a class so that when a particular event occurs, the client that instantiated the object can be notified. Events are still present in VB.NET but are now based upon Delegates\textsuperscript{47}. Using Delegates, it is possible to start a method asynchronously, similar to an event, but it can also have a callback raised upon completion, something which Events cannot do. To run a method through a Delegate, in
this example the Read method in the ThermoReader class, a delegate must first be declared.

Delegate Sub DelegateRead()

An instance of the delegate can then be created using an instance of the ThermoReader class. The signature of this delegate must be the same as the object with which it works. In this example, the Read method of the ThermoReader class has no parameters, so neither does the delegate, DelegateRead.

Dim AsyncRead As DelegateRead = New DelegateRead(AddressOf objThermo.Read)

An AsyncCallback object must also be created that passes in the method that will be called when objThermo.Read has completed. The only proviso is the method has the parameter of the type IAsyncResult.

Public Sub ReadComplete(ByVal lAsyncResult As IAsyncResult)
    MessageBox.Show("Read Complete")
End Sub

Dim asCallBack As AsyncCallback = New AsyncCallback(AddressOf ReadComplete)

**Code Example 5** - Creating an Asynchronous Callback

It is now possible to invoke the delegate which will execute objThermo.Read along with the Asynchronous callback that has just been created.

Dim state As Object = New Object
AsyncRead.BeginInvoke(asCallBack, state)

**Code Example 6 - Invoke Delegate using the Asynchronous Callback**

BeginInvoke will not be a blocked call and any code immediately after this call would be executed. For example, the other reader, TecanReader, could be started so both readers are multi-tasked. When the 'Read' method completes, the Asynchronous Callback is executed as shown in Delegates are probably one of the most radical changes to VB.NET and it is certainly not the easiest concept to understand. However, it is a powerful feature that can make applications run at a faster speed.
CHAPTER 3

The other way to run code asynchronously is to use threads. There are various ways to implement threads, the simplest being the BackgroundWorker component. The best method to gain complete control over a thread is to use the Thread class. This can be implemented using the following code, which again uses the ThermoReader class from the previous examples.

```vbnet
Dim ReadThread As Thread = New Thread(AddressOf objThermo.Read)
ReadThread.Start()
```

*Code Example 7 - Run Method in new thread*

This code will run the *Read* method in a separate thread to the calling code. To then wait until the thread has completed, to synchronize with the calling thread, the following code should be used.

```vbnet
ReadThread.Join()
```

There is much more functionality available that allows complete control over the behaviour of each thread. It should be noted, that although the example shown is pretty simple, writing multithreaded applications that are thread-safe are difficult as threads interact with each other. For example, the *Read* method in the *Reader* class may contain code that interacts with a resource, hardware or software, that can only be interacted with by one caller at a time. If two or more instances of the *Reader* class were then run in separate threads, there could be a scenario where all of the threads are trying to access the resource simultaneously. This often leads to a fatal errors such as memory violations, although the IDE does a good job in handling many of these exceptions.

There are various synchronization techniques available to ensure that code is thread-safe. One way is to use a *Lock*. In Code Example 8, a lock is placed around the code that interacts with Eventlog, which in this example is a resource that writes to some data device. The lock around this code prevents more than one thread accessing this code at any one time. If a thread was running this section of code, another thread which was also running this 'Read' method would wait when it gets to this point until the first thread has completed this section and released the lock.

```vbnet
Public Sub Read()
    Dim WriteLogLock As New Object
    '### Code here to run this reader ###
```

48
CHAPTER 3

### Code in here to write to EventLog object ###
End SyncLock
End Sub

**Code Example 8 – Sync lock the Read Method**

Thread synchronization is very important when developing multi-threaded applications and requires careful planning to ensure it is thread-safe. The main advantage to using threads is the increased speed in running tasks through the application. In the previous code example, provisions are made for running Plate Readers at the same time as opposed to one after the other. When running multiple plates in an assay, through many devices, the time savings can be significant, fully justifying the extra time and effort required to develop a multi-threaded application.

### 3.6 Hardware Interfacing

VB.NET comes with a series of control components that makes it a fairly simple process to integrate hardware. Most devices found in the laboratory are connected to the PC using one of the following types of connection.

- RS232/RS485
- Ethernet
- USB

RS232 and RS485 can be handled using the SerialPort component that is part of the Framework and available in the IDE. The control is then simply dropped onto the form that will be used to send commands, although it is also possible to create an instance of this control without having to place it (sink it) onto a form. For example, if an instrument is used that has a command set that uses the string "HOME" to move the instrument back to the home position, the following code would execute this.

```vbnet
With SerialPort
    PortName = 1 'Set Port
    WriteLine("HOME") 'Send Command to Device
End With
```

**Code Example 9 - SerialPort Component**
Ethernet connections can be made by using the System.Net.Sockets namespace in the .NET Framework. The principles behind using sockets in previous versions of VB are pretty much the same in VB.NET. The main advantage now is the ability to use delegates which gives much more flexibility over the asynchronous control of each socket.

3.7 Data Handling

3.7.1 XML

It is no surprise that the main way to handle data in VB.NET is by using Extensible Markup language, known as XML. The .NET framework is supplied with two base classes, XMLReader and XMLWriter. These are the foundation classes for working with XML, making it very easy to write an application that uses XML documents. For example, to write software to reformat data as it is produced from an instrument such as a 2D Barcode Reader, XML would be by far the best choice as seen in Code Example 10

```xml
<?xml version="1.0" encoding="utf-8" ?>
<Plate Barcode ="A123456">
  <Well Barcode ="DEF123">A1</Well>
  <Well Barcode ="DEF234">A7</Well>
  <Well Barcode ="DEF345">C5</Well>
  <Well Barcode ="DEF456">H8</Well>
</Plate>
```

**Code Example 10 - 2D Barcode XML Data**

Using the Reader class, we can query the attribute of each element in the XML file with the barcode of the tube we want, to return its location. This is a very attractive way to control cherry picking applications. An XML template file, XSL, could be written for the data so that any other application could read and most importantly, understand what the data represented through the .NET XML classes.

3.7.2 Serialization

In any program that uses objects, data is moved around and sometimes it is necessary to save the data. Previously, the best way was to create a load and save method in the
class then write code to read and write all the properties of the class to the required format. To use XML, the Document Object Model available in the Microsoft XML Parser, MSXML, has to be used. With .NET, serialization is available and simplifies the whole process for persistent data. To achieve this, the object is passed as a constructor parameter into a new instance of the Serialization class. The output stream, XML or Text is then defined and the serialize method called. All public properties are then serialized to the defined stream. This functionality is bi-directional so it is possible to deserialise into a new instance of the object. Many assumptions are made in the process and the object has to be simple, although it can handle properties that are collections. If data has to be handled differently, the serialization interface can be implemented within the class.

3.7.3 ADO.NET

In business programming, Visual Basic has one of the largest user bases and one of the primary reasons for this is because of the built-in tools for database connectivity. In VB.NET, ADO.NET (Abstract Data Object) can be used to connect to a variety of databases such as SQL Server, Oracle and MS Access. Nearly all data handling in the .NET environment utilizes XML, and ADO.NET is no different. All ADO datasets are moved around as XML. Any application on any platform that can read XML, can receive data from an ADO.NET enabled application. Another key improvement is the ability to use typed datasets allowing data to be accessed through typed programming. A problem with database programming has been writing SQL statements at design time and then only finding out there is a problem when running the application. Using typed datasets, where the field names are actually properties of the dataset, this problem is removed since the code is checked at compile time.

3.8 Example Application

The following example looks at how a small Windows application can be created to control a Thermo Labsystems Multidrop 96/384. This is a simple device to integrate and one of the most common instruments found in the laboratory.

The first step is to look at the required functionality. Use-case analysis would reveal the following methods:

- SetPort
• Initialize the carrier
• Prime a set volume
• Empty a set volume
• Dispense a set volume to a whole 96 or 384 plate.

Although SetPort has been identified as a method for setting the RS232 port number, it would make more sense for this to be a property. For this example, only dispensing to the entire microtitre plate will be incorporated. The fact that it is possible for the Multidrop to dispense to different columns will be ignored.

A new Windows application project should then be started in Visual Studio. The default form should be renamed to frmMultidrop and this will also be the startup form. A new class file is then added and this is called clsMultidrop. This class is the abstraction of the Multidrop instrument and will contain the methods and properties as described above.

A class for sending commands using RS232 is now created. This class will need a method to send the command and another property to set the port number. In this example, another class file will be created to implement this new class even though it could be written in the existing clsMultidrop class file.

```vbnet
Imports System.IO.Ports
Public Class clsRS232
    Private mSerialPort As SerialPort
    Public Property Port() As String
        Get
            Return mSerialPort.PortName()
        End Get
        Set(ByVal value As String)
            mSerialPort = New SerialPort(value, 9600, Parity.None, 8, StopBits.One)
        End Set
    End Property
    Public Sub Send(ByVal Command As String, Optional ByVal Timeout As Integer = 5)
        Dim sRet As String
        With mSerialPort
            Try
                'Send the command here...
                sRet = mSerialPort.Read(5000)
            End Try
            mSerialPort.Close()
        End With
    End Sub
End Class
```
CHAPTER 3

```
Private mRS232 As clsRS232
Public Enum ee_PlateType
    pt_96 = 1
End Enum

Public Class clsMultidrop
    Private mRS232 As clsRS232
    Public Enum ee_PlateType
        pt_96 = 1
    End Enum
End Class
```

Code Example 11 – RS232 Serial Port Class

The SerialPort object is created using an instance of the class directly from the .NET Framework using the System.IO.Ports namespace, instead of using the SerialPort component. The clsRS232 class shows another feature now available and that is the ability to use constructors with parameters. In previous versions, it was only possible to have a single constructor, New. Constructors can now be overloaded so that when an instance of a class is created, the parameters can be defined in the same line of code.

Now create an instance of the RS232 class in the Multidrop class so that each method can send the relevant command to the Multidrop.
pt_384 = 2
End Enum
Public Sub Initialize()
    mRS232.Send("Q")
End Sub
Public Sub Prime(ByVal Volume As Integer)
    mRS232.Send("P" & Volume.ToString)
End Sub
Public Sub Empty(ByVal Volume As Integer)
    mRS232.Send("E" & Volume.ToString)
End Sub
Public Sub Dispense(ByVal Volume As Integer, ByVal PlateType As ee_PlateType)
    Dim iTimeout As Integer
    'Set Plate Type
    mRS232.Send(If(PlateType = ee_PlateType.pt_96, "TO", "T1"))
    'Formula to calculate how long the Timeout should be
    iTimeout = If(PlateType = ee_PlateType.pt_96, _
        (Volume * 0.07) + 46, _
        (Volume * 0.3) + 49)
    mRS232.Send("V" & Volume.ToString)
    mRS232.Send("D", iTimeout)
End Sub
Public WriteOnly Property Port() As Integer
    Set(ByVal value As Integer)
        mRS232 = New clsRS232(value)
    End Set
End Property
End Class

_Cod Example 12 - Multidrop Class_

This code, encapsulated by clsMultidrop will, when instantiated, control the Multidrop.
To produce a user interface, the form that already exists in the project is used. For this example, one button is added for each function that is available, using input controls to pass in parameters.

### 3.9 Review Conclusion

It should be apparent that Visual Basic .NET is seen as an improvement over its predecessors. The sheer size of the .NET Framework is perhaps overwhelming and it can be difficult to navigate to find the required function. However, this has been addressed in the latest version of VB.NET 2005 with the introduction of the My namespace\(^5\) to provide quick access to the most commonly used functions.

It is important to realize that VB.NET is not the only language that works with the Framework. Visual C#, Visual C++, and Perl also make use of the functionality that is offered. Which language is more efficient or productive is really a matter of personal choice and application specific. For some projects, C# is more suitable than Visual Basic and vice versa as each interacts with the .NET framework in different ways. There is no common consensus on the language that is best, although all programmers seem to have an opinion. The choice of language ultimately comes down to how easy the
syntax is to use and programmer experience. An application developed in VB.NET may require more code than an application developed using C++, but if VB.NET is easier to understand and therefore quicker to code than in C++, any speed advantage may be lost. Once the application is complied and it works as required, no-one will be concerned about the lines of code used or language choice. Apart from a few changes such as structured exception handling, the basic syntax in VB.NET is very similar to VB6. VB.NET should still be seen as an ideal tool for developing applications in the laboratory, for the same reasons that made VB6 suitable, with the added bonus of the .NET Framework at its disposal.

### 3.10 Programming Methodologies

The software development will follow the concepts of Extreme Programming (XP)\(^52\). XP is a method of programming that promotes the concept that coding is the design for the product, and that the code should evolve as the project progresses. The argument for using XP is that it is not possible to design the software to the lowest level without actually coding. XP also encourages constant testing through what is known as NUnit testing\(^53\). NUnit provides a set of test harnesses to develop the code, which is a complete reverse from traditional methods. By writing the test first, and defining the conditions for the test to pass, it is effectively creating the functional requirements of the code. The code is then written to pass this test. Also known as test driven development, keeping this NUnit test block in place, code can be continually written with confidence that the code is being constantly tested. This is especially useful when there is a need to perform regression tests, to ensure new functionality does not break existing code\(^54\). The code separation that has been designed in the overall architecture for the product is ideal for this form of development and ensures the product is developed in the best possible fashion.

Using NUnit, the module testing has been performed as each component is developed. This proved difficult with the GUI, the Overlord Flow and Overlord Main executable, since NUnit can only test classes; it cannot test user interactions. NUnitForms\(^55\) is available as a test for user interactions. However, NUnitForms was deemed unsuitable for testing the main user interaction of dragging and dropping icons onto the Flowchart interface. It was therefore assumed that integration testing would identify user interaction bugs when NUnit module testing had been completed. Once integration
testing is complete, the software will be ready to release in beta form to select customers who are already part of the PaaS customer base.
CHAPTER 4 Software Specification

The most important design considerations for this project are based upon the information gathered in the previous chapters. There are four basic requirements that the new software package must deliver.

- Easy to use user interface (UI) for assay creation
- Process the assay whilst obeying all the assay constraints
- Sample tracking and data management
- Control all the instrumentation.

The following section elaborates on each of these requirements, which will ultimately result in a requirement specification that can be used to design and develop the software package.

4.1 User Interface

The User Interface (UI) is the most important part of the software. It is essential that the software can communicate with instruments and the software can schedule an assay. However, this all becomes irrelevant if the user cannot define a procedure to run the assay. The UI is the link between the system, software and the real world. If this link does not function well, it is guaranteed the software will not perform as originally specified. The UI truly is the measure of acceptance for end-users and needs to be well designed.

A UI is well designed when the program behaves exactly how the user thought it would perform. When designing user interfaces, it is important to constantly imagine how different types of user will interact with the software. Constantly evaluating the design in this way ensures that the UI is accessible to multiple levels of user expertise. The UI should use concepts that are familiar to the user. The UI is going to be where the user defines the assay that the automation needs to run.

The UI for assay development needs to exhibit the following.

- Simple to use interface
• Allow efficient creation of the assay
• Prevent error scenarios
• Flexibility to allow use of all available instrumentation and labware.
• Distributed Control

It is envisaged with the text-based approach that the user will simply type the actions they want to perform into an interface similar to a Word processor. Typing in a keyword such as "Washer" will be automatically recognized at which point the software will prompt the user for the necessary parameters. The user would then continue adding the steps required for the assay by adding to the existing text which will read as shown in Table 1 on page 25.

With the flowchart approach, the user could simply drag and drop an icon representing the instrument onto the chart, at which point the parameters for that instrument can be set. The assay steps could then be built up to a complete assay.

Both approaches satisfy the criteria stated earlier. Both description methods would be easy for a novice user to interact with and good design would allow assay development within a reasonable time. However, the examples up to now have been based upon a simple number of steps. The ELISA assay is typically made up of a larger number of tasks. With the text based approach, the text required to define this assay would be substantial to the extent it could become unmanageable. From a software development perspective, code will be required to translate the natural language text defining the assay into an instruction set that can control the instruments. This approach is acceptable if the Ul is based around common instrumentation; in this case the software will be interacting with 300+ instrument drivers that already exist. Much of the code development time and testing would be dedicated to this interface. The flowchart method does not have any of these problems. With larger assays, the flowchart could simply refer to another flowchart if it was getting too big to manage. There is also no need for translating the interface into an instruction set since the structure of the flowchart holds this information. Flowcharts are already used as a tool for software development for this very reason. For these reasons, the Ul for this project will be based upon a flowchart.

A set of data variables will be needed that are accessible to the user. The flow of a chart would always be dictated by data. If we were writing an assay that moves a set number
of sample plates to the same instrument, we would place these actions into a loop. The number of sample plates dictates the number of times the flow runs around the loop. This value has to be stored somewhere in the system for use in this loop. A global common set of variables will be ideal for storing this data.

### 4.2 Scheduling

It has been decided to use event driven scheduling as the method for processing assays. Taking into account the strengths and weaknesses of each type of scheduling method, event driven scheduling is the most suitable to what we are trying to achieve. Event driven scheduling is the closest to assimilating the decisions a scientist makes when running the assay manually. They can react to an instrument not behaving as expected or taking the experiment down a path based upon results generated at runtime. A software package that works as a person does is clearly going to be a simpler product to use, as it is much clearer to understand. The main disadvantage to this form of scheduling is that it does not know how long tasks take to complete accurately. This is only an issue for incubations. If the experiment requires incubating a plate for an exact amount of time, the software should allow this to be the case. However, the reality is that the almost all incubations have a timing resolution of ±10 seconds at worst. If configured correctly, event driven schedules should be able to ensure incubations complete within this time window by prioritising tasks.

The flowchart will be where the scheduling is defined. Using a combination of flow operators, the method of execution can be defined. Taking into account that the software must have an easy to use UI for assay creation, two key flow commands were identified for controlling the flow of the assay:

- **Loop** – looping a fixed number of times and looping a variable number of times,
- **Decision** – Go down a path of the flowchart based upon the value of a variable or status of an instrument.

These commands are loosely based upon programming concepts. For example, the Decision command could easily be described as an “If..Then” statement or as a “Case” statement. However, it is imperative that although the flowchart is analogous to a programming language, the software should not be presented as such. This would go against one of the key objectives of making the software understandable to a scientist who may have no exposure to programming concepts.
4.3 Data Handling

In the previous version of laboratory automation software, paa developed drivers to access common data formats such as a Comma Separated Variable and Tab Delimited Variable files as well as common databases such as Access, SQL Server and Oracle. Although this was a well-integrated solution, it offered little functionality and the drivers were constantly modified to accommodate user requests for working with their data. It was decided that a more flexible approach would be required.

Although going slightly against a requirement of this project of making the software easy to use, it was decided that having a command available in the flowchart that could execute code snippets would offer the best solution. An advanced user would probably attempt a task like this. By allowing a snippet of Visual Basic script to execute, the software would immediately be able to interact with nearly any available data source. Code examples are available extensively on the Internet. If a common library of code were also included, the work required by the user would be decreased and would improve data integration times. By providing a method for these code snippets to interact with common variables in the product, it would be possible for the assay flow and therefore scheduling to be controlled by external data. For example, the barcode of a sample plate could be used to query a database to determine the experimental method that must be run on this plate. This type of functionality would not be possible using a dynamic scheduler, yet it is exactly the type of scenario that scientists encounter when running a plate manually. Users typically enter the barcode of the plate into the LIMS to find out the work that needs to be performed. By providing a method of automating this sequence of steps, an additional use for a product can be made.

4.4 Instrument Control

The ability to control laboratory instrumentation is obviously an extremely important aspect of the product. There are many different types of instrument available that can be automated such as plate readers (e.g. spectrometers, high content readers), liquid handing stations, incubators, centrifuges & plate sealers. Each of these devices can be controlled in different ways.

- Direct control – The instrument driver sends raw commands to the instrument over Serial RS232/RS485, USB or TCP/IP,
• Instrument SDK – The instrument manufacturer supplies either a .NET assembly or ActiveX component for control of the device.

• Instrument Software – The control software for the instrument has an automation interface that allows the instrument to be controlled by a 3rd party. This interface can be a .Net or COM interface. In older software, DDE or messaging pipes are used. If there is no interface available, it may be possible to run the software by command line and pass in parameters such as a method developed in the native software, which when executed will run the instrument.

Although each instrument presents different functionality, and has different methods of control it will be important that a common interface is developed for each instrument driver so as not to complicate the final product. Automation systems have different instruments on them. If a common “black box” approach can be taken with regards to each instrument, it means the same software product, if configurable, can run the different instruments with the same control software. As stated in Section 1.9, a well designed SDK will ensure that the link to each instrument driver can use common code, which ultimately will make the process of adding instruments simpler to the end user.

### 4.5 Distributed Control

For the majority of automated systems, the equipment is physically attached to the PC with the controlling software installed. However, there are instances where one or more of the instruments are physically connected to another PC. For some instruments that are connected via their native software, in particular plate readers, it is only possible to have one instance of that native software on the PC. If the automated system has two of these instruments, the only way to run both of them is to use an extra PC (Figure 26). A method of distributed control is then required.
Distributed control is also desirable for systems where one robot is feeding plates into two small workcells. Each of these workcells can contain multiple instruments, such as a liquid handler and plate washer as shown in Figure 27.
A method for the main control software to maintain other instances of the control software on different PCs will need to be implemented to handle the two scenarios described.

### 4.6 Requirement Specifications

OVERLORD Workstation is the current product at paa and has an established customer base. To assist in determining what features should exist in a new product, a questionnaire was sent out to existing customers to determine what additional features should be included in a new product. It was important to find out what parts of the existing software the customer liked to ensure useful features are not lost. Using the
information gathered from literature and by querying the customer base, a Functional Requirement Specification (FRS) was developed to meet the project objectives.
CHAPTER 5 Architecture

5.1 Graphical User Interface

It has already been stated that the UI will be based around a flowchart for reasons previously discussed. The flowchart will be the key component of the software package. It will be the mechanism for representing what series of tasks will be carried out to automate a laboratory process. The flowchart will have dual functionality. It is the link between the user and the software application to allow the user to define the process. It is also the representation of what will happen when the user decides to run the flowchart.

The Graphical User Interface (GUI) architecture is an important aspect of the software design. A good architecture is based upon object orientated (OO) principles and structures the application in a way where it is clear what role each section of code undertakes. As with the majority of software projects, code evolves as the project progresses. A solid architecture allows code to evolve without breaking existing structure.

An OO based approach to software development means a separated presentation architecture must be considered. Separated presentation is a commonly used pattern whereby the code that represents the presentation code (UI) is separated from the rest of the application to the extent that the application is unaware of what code is in the presentation layer. This pattern is particularly common with multi-tiered applications but it is also a useful approach for even the simplest single-tier applications.

Separated presentation architecture has the following advantages:

- Clear separation of concerns/responsibility. This allows the view and the model to evolve independently of each other.

- Test Driven – By isolating each major component (UI, presenter/controller, and model) it is easier to write unit tests. This is especially true when using the MVP pattern which only interacts with the view using an intermediate interface.

- A clear separation approach will increase code reuse. For example, the same presentation component can be used to represent the data in other parts of the software (e.g. a preview pane in an "Open Dialog")
Flexibility/Adaptable – By isolating into the separate components your code base is more adaptable to change.

There are two main patterns for GUI architecture that have been considered based upon a separated presentation layer57.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model View Controller (MVC)</td>
<td>The View is responsible for the UI. The Controller is responsible for responding to UI actions and updating the Model. The Model persists the state of the data and notifies the View of any change.</td>
</tr>
<tr>
<td>Model View Presenter (MVP)</td>
<td>The View is responsible for the UI and notifies the Presenter of any change. The Presenter decides upon how to react to the change and manipulates the Model accordingly. The Model will notify the Presenter that it has changed and in turn update the View.</td>
</tr>
</tbody>
</table>

The key difference between the two patterns is the stage where the Model updates. In the MVC pattern, the Model updates the view directly. In the MVP pattern, the Model is not aware of the View so it has to notify the Presenter, which in turn updates the View. This is a distinct advantage for the MVP approach over the MVC pattern with regards to unit testing as the Model can be tested in complete isolation. This is preferable when working within a team of developers, as the level of responsibility is quite clear.

Taking these factors into account, it was decided that the Model View Presenter pattern would be most suitable at this stage of design. The next step of the design was to see how the elements of the software would work with MVP.

The MVP pattern applied to the requirements are met as follows:

- **Model** - A Procedure Object
- **View** - Flow Chart that the user will interact with by dragging commands onto
- **Presenter** - The main application UI that will contain the Flowchart interface.

### 5.1.1 Model

The Model is used in the software to define the laboratory automation processes that will be run by the user. Procedures are represented by the Flowchart but the underlying data is all held with the Overlord Model. The Model is made up of 4 classes:
• **Procedure** – This is the top level object that defines an Overlord Procedure

• **Action** – Represents each block within a procedure that executes a command

• **ActionGroup** – A collection of Actions. The Procedure class inherits from this since the Procedure contains Actions

• **ActionGroups** – A collection of items of the type **ActionGroup**. Each **Action** contains an **ActionGroups** object. This class is only used if an **Action** is a flow type (i.e. Decision, Loop or Multiple)

### 5.1.2 View

The Flow UI Control has been designed to be the View component of a program built using Model View Presenter architecture. The Overlord Model (**Figure 28**), which is a dependency of this control, is the Model part of this definition.

![Model View Presenter Architecture](image)

**Figure 28** – Model View Presenter Architecture showing the relationship between each component and how actions and events propagate so that the user sees the result of their interaction with the view component

The Flow UI will raise events when objects are dropped onto the UI. Depending upon where the object is dropped, an event will be raised. This call is then passed back to the Presenter, which typically is the main GUI. The Presenter should then update the Overlord Model appropriately. The **Overlord Model Procedure** object fires a **Change** event when an action is added or deleted. The **Change** event that is also handled by the Presenter, at this point should pass the Overlord Model Procedure into the Flow UI so that the flowchart is synchronized with the Overlord Model Procedure (**Figure 29**).
The AddAction event, sunk into the Windows form hosting the flowchart component will create a new Action object. This will then be added or inserted into the Procedure (Model) object that is instantiated in the Windows form. The Update event will then need to be invoked from Add or Insert method in the Macro Object. The Update event is also sunk into the Windows form and the code in here should refresh the Flowchart UI (Figure 30).

This form of programming where objects notify other objects of changes is crucial to the implementation of the Model View Presenter architecture.
5.1.3 Presenter

The presenter will be the main executable of the application that shows the flowcharts and links this visual representation into the Model object known as a Procedure. This will be the main section of the software with which the user will interact, so it is very important that this is designed to be visually pleasing and easy to use. Feedback from previous customers (See Appendix A) suggested that a multi-document interface (MDI) would be a significant improvement to the product as would allowing the system variables to be visible on the main GUI. The main UI was developed based upon this information from the paa customer base and experience gained from using the previous product known as Overlord1 (Figure 31).

![Figure 31 - Presenter UI Design that uses a Multi-document interface approach to show multiple flowcharts simultaneously](image)

The UI design emphasises that the flow chart is the core part of the product by being central and the largest controls on the form. By keeping the interface “clean”, the implication to end-users is that the product is simple to use. It is also important that
tools to assist in creating the flowchart, such as the flow actions and the global variables are accessible as well. The UI design keeps all of this information on the same screen, which should allow rapid assay development. A dynamic help section has been added that will show the relevant help topics based upon the users interaction with UI. This additional feature will enhance the users comfort with the product, as it will constantly provide a source of reference.

5.2 Run Architecture

The separated presentation architecture is for applications where the user is inputting data in some form into the program. This is relevant to this project since the user will need to define the assay to be automated. However, the software also needs to be able to run the process that has been defined. The architecture for this section also needed to be designed.

The code to run a procedure should be able to take the information defined in the Model for execution. The user will also need to be aware of which action within the procedure is currently being executed. It is envisaged that the flowchart will be able to provide this information to the user by simply highlighting the active task. These requirements imply that the code to execute a procedure will need to understand the structure of the Model. It will also need to be able to update the Flowchart.
The full architecture has been developed for the laboratory automation software product (Figure 32). It is an extension of the MVP pattern, that is core to the product, which incorporates the functionality to execute procedures as defined by the Overlord Model. The RunEngine will take a procedure object and then process the actions it finds in the flowchart. Since the flowchart depends upon variables, it needs to have access to the global variables that are available in the software. Most importantly, it will also need access to the commands that allow instrumentation to be run. As the run engine uses a Model Procedure, it will have indirect access to updating the Flowchart using the MVP pattern. The addition of the RunEngine to the overall architecture maintains the code separation that is desirable to allow isolated testing.

5.3 Commands

A simple architecture for the commands is critical if an aim of the software package is to provide a configurable system where other programmers can add their own instruments.
or data components into the system. Current automation packages, such as Thermo CRS POLARA require the user to create many different components to integrate a new instrument. An ActiveX interface is required for the edit elements of the instrument. A series of RAPL3 components are also required to handle the runtime communication and event notifications. These components are coded in a CRS proprietary C based language. Although this provides an extremely flexible solution to handle all types of instrument, the steps required to complete this integration are numerous. The entire operation is susceptible to errors, just down to the required number of steps. Multiple language components also increases the learning curve required to install a new instrument in POLARA. With a good design in place, the commands can be implemented as single .NET assemblies. This presents the following advantages:

- Only one assembly needs to be created and maintained.
- A single integrated design environment, such as Microsoft Visual Studio is required to code the assembly.
- Knowledge of a single (popular) language.

5.4 Distributed Control

The distributed control needs to be able to run another instance of the control software residing on a separate PC. A Remote command will be developed that will allow Overlord Procedures to be run on the different PC.
Only a subsection of the entire architecture will be utilized on the remote PC at runtime. Although the full architecture will be used to develop procedures, at runtime the GUI on the remote PC will be not be required as the user will only interact with the Main PC. It is therefore important that the status and any error conditions are returned to the main PC.

The remote command will need to find a method for invoking a Run Engine on a separate PC. The Microsoft .NET framework has been designed to support N-tier architectures where business logic is separated from the user interface and often runs on a separate machine. Within the .NET framework, this functionality is known as remoting that allows objects to be instantiated in different processes. The key aspect is that the different process can exist on a different PC to the client, in this case the Main PC. This setup has many advantages to the development process since the same code developed to run on the Main PC can be run on the remote PC, which in turn reduces the coding and testing cycle.
CHAPTER 6 Coding

6.1 Overlord Model

The Overlord Model is used in the application to define procedures. The model part of the MVP pattern is used to define the procedures which will represent the assay the user wishes to automate. Procedures are represented graphically by the Flowchart but the underlying data is all held within the Overlord Model. The resulting Procedure is the same object that is used to execute the actions at runtime.

The Model is made up of four classes as described in Section 5.1.1. Each class is described in further detail in the follow section.

6.1.1 Procedure Class

The Procedure class inherits from the ActionGroup class since this class is essentially a collection for holding action objects. Only members that are unique to this class are described in this section.

6.1.1.1 Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name (String)</td>
<td>The name of the Procedure. This property gets updated with the name when the Save method is called.</td>
</tr>
<tr>
<td>Author (String)</td>
<td>The name of the person who saved this Procedure last</td>
</tr>
<tr>
<td>Notes (String)</td>
<td>Allows a description of the Procedure to be saved along with the Procedure</td>
</tr>
</tbody>
</table>

6.1.1.2 Methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>Builds the Procedure based upon the specified XML file (Overlord Procedure File)</td>
</tr>
<tr>
<td>Save</td>
<td>Saves the Procedure to the specified XML file (Overlord Procedure File)</td>
</tr>
<tr>
<td>CreateNew</td>
<td>Creates a new procedure with the START and END actions already added.</td>
</tr>
</tbody>
</table>
6.1.1.3 Events

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change (Change Delegate)</td>
<td>If any object is added or removed at any level, the change event will be</td>
</tr>
<tr>
<td></td>
<td>raised to notify the client, the main executable, that the structure of the</td>
</tr>
<tr>
<td></td>
<td>Procedure has changed</td>
</tr>
</tbody>
</table>

6.1.1.4 Example

The following code example demonstrates how the class is used to build a procedure using two instances of the `Action` class.

```vbnet
Imports Overlord.Model
Private WithEvents NewProcedure As New Procedure
With NewProcedure
  .Name = "New Procedure"
  .Notes = "Example Procedure"
  .Author = "A.Programmer"
End With

Dim ActionItem1 As New Action("Test1", Action.ActionType.PluginAction)
Dim ActionItem2 As New Action("Test2", Action.ActionType.PluginAction)
NewProcedure.Add(ActionItem1)
NewProcedure.Add(ActionItem2)
NewProcedure.Save("c:\Procedure.xml")
```

**Code Example 13 – New Overlord Procedure**

A new action can be inserted within existing actions using the following code.

```vbnet
Dim ActionItem3 As New Action("Test3", Action.ActionType.PluginAction)
NewProcedure.Insert(1, ActionItem3)
```
CHAPTER 6

Code Example 14 – Insert action in procedure

This simple code allows the main executable to update a procedure in a logical fashion when the user adds a new action to the flowchart. As part of the MVP architecture, this code will be implemented at the point when the update event is invoked on the flowchart. See Section 5.1.2 for further details on when this event is raised.

6.1.2 Action Class

This class is used to define an Action in an Overlord Procedure. This is the smallest building block within the software application and is essentially what makes up the product. Each action contains the data required to run an activity on a specific command.

6.1.2.1 Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type (ActionType)</td>
<td>The type of Action represented by the enumerate ActionType.</td>
</tr>
<tr>
<td>Command (String [XML])</td>
<td>The Overlord procedure string which contains details about what this Action will do. This string is passed into Overlord drivers at Runtime and is built up at Edit time. The procedure string will be made up of XML data.</td>
</tr>
<tr>
<td>Icon(String)</td>
<td>Full path to the icon that represents this Action. This is used by the Flowchart so that is can graphically represent the action within the chart.</td>
</tr>
<tr>
<td>ID(GUID)</td>
<td>The ID used to determine the Command that should be used when this Action is either edited or executed.</td>
</tr>
<tr>
<td>Groups(ActionGroups)</td>
<td>Collection of ActionGroup objects. This property is only populated if the Action type is flow type such as a Decision or Loop action.</td>
</tr>
</tbody>
</table>

6.1.2.2 Enumerates

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActionType</td>
<td>TagAction = 4</td>
</tr>
<tr>
<td></td>
<td>BlankAction = 5</td>
</tr>
<tr>
<td>FlowMethodType</td>
<td>None = 0</td>
</tr>
<tr>
<td></td>
<td>LoopFlow = 1</td>
</tr>
<tr>
<td></td>
<td>MultipleFlow = 2</td>
</tr>
</tbody>
</table>
6.1.2.3 Example

The following code example demonstrates how the Action class is used to build up a Procedure containing an Action representing a Decision.

```
Imports Overlord.Model
Private WithEvents NewProcedure As New Procedure
With NewProcedure
  .Name = "New Procedure"
  .Notes = "Example Procedure"
  .Author = "A.Programmer"
End With

NewProcedure.CreateNew
```

Figure 34 – With a decision there is a minimum of two different paths that the process can go based upon the evaluation of the condition set. If true, in this case “Action 1” will run, if false, “Action 2” will be run.

A Procedure object should be created which includes the default Start and End Actions.

The ‘Decision’ action to be built needs to contain two ActionGroup objects, each representing the different paths that the procedure can go down. If the Decision is true,
then Action1 should run: if the Decision is false, Action 2 should run. Although the example is showing just one action in each path within the decision, it should also be possible to have a series of actions run if the Decision is true or false. This is why ActionGroups are stored in the ActionGroup collection as opposed to just a single action.

**Figure 35** - The decision command, as shown in Figure 34, is made up of an ActionGroups object that contains two ActionGroup objects, each object representing either True or False.

The following code creates the 'Decision' action.

```vba
Dim SubActionGroupItem As ActionGroup
Dim i As Integer

ActionItem.Flow = Action.FlowMethodType.MultipleFlow

For i = 1 To 2
    SubActionGroupItem = New ActionGroup(True)
    SubActionGroupItem.Actions(i-1).Command = "Action" & i.ToString
    ActionItem.Groups.Add(SubActionGroupItem)
End For
```
Two ActionGroups are created and are added to the Groups property of the Action. By setting the ActionGroup constructor parameter to true, a blank action is created inside the group. This is just to give the flowchart a more appealing visual within the Flowchart.

### 6.1.3 ActionGroup Class

The ActionGroup class is used to contain Actions. An instance of the ActionGroup class is used to group together actions, either as a property of a Flow type action or as the base class to the Procedure class.

The ActionGroup class is based upon the Generic Collection using the Type 'Action'. Only members that are unique to this class are listed.

#### 6.1.3.1 Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actions(ActionGroup)</td>
<td>Read-only property that returns the current ActionGroup</td>
</tr>
</tbody>
</table>

#### 6.1.3.2 Methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>Adds an Action to at the end of the current collection of Actions</td>
</tr>
<tr>
<td>Insert</td>
<td>Inserts an Action before the specified index</td>
</tr>
<tr>
<td>InsertAfter</td>
<td>Inserts an Action after the specified index</td>
</tr>
</tbody>
</table>

The ActionGroup class provides protected methods that can be used to customize its behavior when adding and removing items, clearing the collection, or setting the value of an existing item. Elements in this collection can be accessed using an integer index. Indexes in this collection are zero-based.

#### 6.1.3.3 Example

The following code example demonstrates how the ActionGroup class is used to build a procedure that incorporates a 'Loop' command. The Procedure first of all needs to be defined. The same code listed in Code Example 15 can be used to create the procedure.
As the Start and End Actions are created automatically, all that needs to be added is the Loop Action as shown in Figure 36.

Figure 36 – A loop procedure that will repeat Action 1 a set number of times or until a condition, such as a variable value becoming equal to 1, is met.

The Procedure model will have the following structure as shown in Figure 37.
Figure 37 – The loop action within the procedure shown in Figure 36 will contain a single ActionGroup that in turn contains Action 1

The code to create the action for a Loop action is shown in Code Example 17.

```vbnet
Dim SubActionGroupItem As ActionGroup
Dim i As Integer

ActionItem.Flow = Action.FlowMethodType.MultipleFlow
SubActionGroupItem = New ActionGroup(True)
SubActionGroupItem.Actions(0).Command = "Action1"
ActionItem.Groups.Add(SubActionGroupItem)
```

**Code Example 17** - Create Loop action

6.1.4 **ActionGroups Class**

The **ActionGroups** class is used as a property of the **Action** class. **ActionGroups** contain instances of the **ActionGroup** class. For a Loop action, the **ActionGroup** class will contain a single **ActionGroup**. For a decision, the **ActionGroups** instance will contain as many actiongroup instances as there are decisions on the Decision Action
e.g. If the Decision action has two cases, \([\text{NoPlates}] = 2\) and \([\text{NoPlates}] = 3\), then the total number of decisions is 3 as there are 2 true cases and a default false case.

The ActionGroup class is based upon the Generic Collection\(^{59}\) using the Type 'ActionGroup'. Only members that are unique to this class are listed. However, using this class will reveal it is using the same members found on a Generic collection (e.g. Add, Remove, count, etc).

6.1.4.1 Methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>Adds an ActionGroup to at the end of the current collection of Actions</td>
</tr>
<tr>
<td>Insert</td>
<td>Inserts an ActionGroup before the specified index</td>
</tr>
<tr>
<td>InsertAfter</td>
<td>Inserts an ActionGroup after the specified index</td>
</tr>
</tbody>
</table>
6.2 Commands

A model procedure in the software package will use various commands to execute the assay it is representing. These commands will be running instruments on the system and manipulating data in some fashion. Commands will also be made available within the software to describe how the flow of the model procedure will process. There will also be commands to set and read the system variables that are used throughout a run (See Appendix A of all required commands)

6.2.1 Command Interface

Use case analysis of the required commands in the User Requirement Specification (URS) reveals common functionality across the various commands, irrespective of the fact that each command performs a different activity within a model procedure. A common interface on this key section of the software package will be extremely advantageous as it reduces significantly the coding required to invoke these commands. If each command or command type requires different code, the implementation time increases as does the time spent on debugging. These functions are listed in Table 5.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edit</td>
<td>Displays in a windows form the functionality available on this command. The functions selected will then be saved into an action to use in the model procedure</td>
</tr>
<tr>
<td>Setup</td>
<td>Displays in a window form parameters that need to be set for this command on the system. Contains settings that don’t get changed from one Model Procedure to the next (e.g. The full pathname of an executable to call as part of a command)</td>
</tr>
<tr>
<td>Runtime</td>
<td>The section of code that is passed the action from the Model procedure and executes it. If the command is controlling an instrument, invoking the runtime function would get the instrument to run.</td>
</tr>
<tr>
<td>Initialize</td>
<td>Called at runtime, prior to the main run stating. This method can be used by the command to initiate any communication with the device the command is controlling or setup any data connections</td>
</tr>
</tbody>
</table>

Table 5 - The common functionality that will be found within every Overlord command

Since these methods and properties are common for all commands, a base class was created so that all commands would be seen as the same by the software application.
Using base classes have the following benefits

- Reuse of code.
- Abstraction of complex code that is common to each class that inherits this base class.
- Code consistency as the same interface is used with the objects.

The use of base classes is essential for providing an application that can be configured to use different .NET assemblies, in this case instrument drivers, by the end user. The method of loading assemblies dynamically within a .NET application is known as Reflection. This provides the mechanism within the application for the user to only load the .NET assemblies (commands) that they have connected on their automated system.

```vbnet
Dim assembly As Reflection.Assembly = Nothing

Code Example 18 – Load Command Assembly
```

The code above loads the assembly (instrument command). For simplicity, all Overlord commands are located in a fixed location that is a subfolder in the root application directory. This then allows the application to interrogate this folder and only allow assemblies that inherit from the Command base class to be available to be added into the main application.

```vbnet
Dim commandType As Type = assembly.GetType(cmdlnfo.TypeName, False)

Code Example 19 – Determine Assembly type
```

Once the assembly is loaded and the type confirmed, the command can be instantiated to use within the application. The code for this is shown in Code Example 20.

```vbnet
Dim cmd As Command = CType(Activator.CreateInstance(assembly.GetType(cmdlnfo.TypeName), New Object() {cmdlnfo.ID, cmdlnfo.Port, cmdlnfo.CommandAlias}), Utilities.Command)

Code Example 20 – Instantiate command using Reflection
```

The constructors for the commands are used at this point, as if the command was early bound within the project. Although slightly more code is used to instantiate a single command, it is a far simpler implementation than adding every single command at
CHAPTER 6

design time into the application. As every command will inherit from the same base class, this code will only need to be written once to load any type of commands.

6.3 Command Execution

The most important element of the runtime section of the software is its ability to run instruments at the same time. As discussed previously, the scheduling demands the ability to run multiple assay plates simultaneously within the system. This then allows the automated system to deliver the best throughput possible.

Each instrument will be an instantiated object. Every object encapsulates an abstract representation of the relevant instrument, but will have a common interface. This simplifies the runtime section of the code, as only one implementation needs to be written to handle all of the instrument drivers that will eventually be available. There are alone more than 50 types of plate reader available, so this is an important simplification of the code, not just for code development, but also for maintaining the code base.

To execute an instrument driver, the "Runtime" method will need to be called with the necessary parameters. These values from the procedure model will be passed to the runtime method and executed. This method call is a blocked call. It is necessary to make this call unblocked so that "Runtime" on more than one instrument object can be called simultaneously to allow concurrency.

In the past, developers have had to create their own proprietary mechanism to asynchronously invoke calls on an object. A common method was to have the object spin off a worker thread to process the client's request and immediately return control to the client. The object would later signal the client somehow when the call completed (if the client wanted to know), and the client had to distinguish between multiple method completions. These mechanisms are difficult to develop and test. The developer would need to spend time reinventing the wheel instead of working on more useful aspects of the product.

The .NET mechanism for asynchronous calls is a mainstream facility used throughout the .NET Framework (See Section 3.5). This simplifies tremendously the implementation of executing the instrument devices asynchronously. To simplify the process of writing commands, it is compulsory that the same code is used for synchronous and asynchronous execution. This allows the developer to focus on just
writing the Overlord command. This is important as it is desirable that writing Overlord commands is as simple as possible to increase the chances of adoption by developers.

The main executable should be able to issue multiple asynchronous calls, and manage multiple asynchronous activities in parallel. The main executable should be able to distinguish between different Overlord commands, running asynchronously as they complete their set activity. When component methods have parameters or return values, the main Overlord executable should have a way of getting these parameters or results when the method completes.

Threading was used as the method of running commands simultaneously. A RunCommand class was implemented that was a wrapper for each command to execute its runtime method. The key code within the RunCommand is the Run method which processes the action for this command.

```csharp
Public Sub Run(ByVal action As Action)
    _action = action
    'Reset the signal, so other threads are aware that we are in 'Runtime'
    _waitHandle.Reset()
    'Setting this property will raise the StatusChangedEvent UI
    Me.Status = StatusTypes.Busy
    _runThread = New Threading.Thread(New ParameterizedThreadStart(AddressOf RuntimeThreadStart))
    _runThread.IsBackground = True
    _runThread.Start(action)
End Sub
```

**Code Example 21 – Runcommand Run Method**

The status of the method is returned by using a Mutex that is reset before thread is started and then set upon successful completion. This allows the main application to determine if the command is idle or busy by querying the Mutex, which is a thread safe call. Any error generated by the command is propagated to a single error event on the collection that contains all of the commands. A single error handler can then be used, as the event parameters identify the command and other associated information. Even if multiple commands error simultaneously, the .NET framework ensures these events are queued.
6.4 **Runtime Engine**

This component takes the procedure and runs it using an "Event Driven" schedule. It extracts each action from the procedure and processes it accordingly. For actions that are controlling instruments, this is simply a case of passing the action to the command as actual work is performed within the Run command (See Section 6.3). For the logic commands, decision and loop, the conditions for these commands need to be evaluated within the engine to determine the next set of actions to execute.

The run engine has been structured so that it maps directly to the procedure object with the actions within it. The main method, called "Start" simply iterates over the actions within the procedure as shown in **Code Example 22**.

```vbnet
For actionIndex As Integer = 0 To _actionGroup.Actions.Count - 1
    'Get the current Action
    currentAction = _actionGroup(actionIndex)
    'Get the current Command for the current action
    currentCommand = Core.Commands(currentAction.ID)
    CODE HERE TO DETERMINE COMMAND AND PROCESS ACCORDINGLY
Next
```

**Code Example 22** – RunEngine Start basic structure

Within this code, the command allocated to the actions is determined. Based upon the type of command, the run engine will either send the action to the relevant Run command or process the action internally. Only actions that are using Logic commands and the command to call other procedures are processed internally. All three of these commands result in the current Runengine creating a new run engine to process. However, each runs in a slightly different fashion. For the decision command, the decision conditions are evaluated to determine which branch the decision flow will go. Each branch results in a new ActionGroup, so once the branch has been selected, a new RunEngine is created and the relevant ActionGroup is passed in. An example of how this processed is shown in **Figure 38**.
CHAPTER 6

Decision conditions are evaluated. This Decision has one condition, so 2 branches. The condition in this example is TRUE, so ActionGroup containing ACTION1 is selected.

This Actiongroup is passed into the new RunEngine that is instantiated.

Figure 38 – When a decision is processed within the Runengine, it will evaluate the conditions set within the decision to determine which Actiongroup should be processed. In this example, the decision condition is true, so the Actiongroup containing Action 1 will be processed.

This iterative pattern ensures that no matter how many decisions, or loops and call procedures, are used within a procedure, the same, simple code base for RunEngine is utilized. There is negligible overhead associated with having multiple instances of RunEngines since the code within the class is small.

Dim paramHelper As New Decision.ParameterHelper(currentAction.Parameters)
Dim decisionCase As Integer = 0 'Set to zero
'Check all of the conditions, and run the first one that evaluates...
For iCondition As Integer = 0 To paramHelper.Conditions.Count - 1
    If paramHelper.Conditions(iCondition).Evaluate Then
        'Run this ActionGroup
        decisionCase = iCondition + 1 'Note: Plus one for the else case
        Exit For
    End If
Next
'Run the engine with the determined case
_spawnedEngine.ActionGroup = currentAction.Groups(decisionCase)
_spawnedEngine.Start()

Code Example 23 – RunEngine decision runtime code
The same principles for processing a decision are used to run an action that uses the Loop command or the Call procedure command. The loop action will have one action group within itself (See Section 6.1.3) that needs to be run x number of times, where x equals the number of times the action should loop. The spawned RunEngine is simply called x number of times with this action group. Call procedures are even simpler, as the spawned RunEngine simply runs the procedure specified in the action.

6.5 Error Handling

Each RunCommand wrapper has an error event, "RunError" which is invoked if an exception is raised within the command. This exception could be a message indicating that the instrument is not powered on, or some form of feedback from the device such as a reagent container having no liquid for completing a liquid dispense. This architecture simplifies creating new commands, as any errors can be handled in the default .NET approach where an exception is thrown.

```csharp
For Each cmd As Command In Core.Commands.Values
    AddHandler Core.Commands.RunCommands(cmd.ID).RunError, AddressOf _RunCommand_Error
Next

Code Example 24 – RunError Subscribing
```

When the error state is raised, the command will remain in the “Error” state. If the action is set to "Wait until finished", the RunEngine will effectively be paused as it will only continue once the command becomes idle. If the action is not set to wait until it has finished, the RunEngine will continue processing actions. However, if it comes up to an action that uses the command in an error state, it will pause until that command is “Idle”. This behaviour allows the automated system to continue running, even if a device is in an error state. This functionality is key to allowing system pooling, where the automated system uses whichever device is free out of a pool of similar instruments.

When the error is raised, the user will be required to respond to the error condition. There are two options available for the RunCommands. "Ignore" simply sets the command state back to idle and "Retry" which attempts to run the action again. The user can "Abort" the whole process which will stop the RunEngine. The error state on the command is set to Idle so the RunEngine can complete the Abort process.
Private Sub _RunCommand_Error(ByVal sender As Object, ByVal e As CommandErrorEventArgs)

Dim lockobj As New Object

SyncLock lockobj

' Display a messagebox to the end user so they can select how to proceed
Dim dr As Windows.Forms.DialogResult = MessageBox.Show(_
(e.Exception.Message, e.Command.ToString, MessageBoxButtons.AbortRetryIgnore, MessageBoxIcon.Error)

Select Case dr
    Case Windows.Forms.DialogResult.Abort
        Core.Commands.RunCommands(e.Command.ID).Ignore() 'Ignore the error and Abort the RunEngine
    Case Windows.Forms.DialogResult.Retry
        Core.Commands.RunCommands(e.Command.ID).Retry() 'Retry the command that has just errored
    Case Windows.Forms.DialogResult.Ignore
        Core.Commands.RunCommands(e.Command.ID).Ignore()

' Ignore this error and continue
End Select

End SyncLock

End Sub

**Code Example 25 – RunError Handling**

### 6.6 Common Utilities

A common assembly was generated to be shared amongst all assemblies in the application. This assembly contained the following components:

- Internal Commands,
- Data Layer,
- Base classes,
- System Variables,
- Log File.

All of the commands that are essential to the application, such as the logic commands are stored within this assembly to ensure they are always present. The collection that maintains all of the commands also exists in this assembly and is configured to always load the essential commands in its constructor.

The underlying database to the system is an SQL Server 2005 express database. An XML data source was considered, however the data is exposed and it is imperative that the data is secure. On that basis, a database approach is preferred. There are a set of rich ADO interfaces that provide a seamless and tight integration with SQL Server 2005, which makes this the logical choice for storing the application data. This database is used to store all data within this common utility assembly, such as the commands and the system variables. Helper classes have been developed for the data connection and SQL interactions with the data. These helper classes separate the data from the code, which assisted the unit testing of each component.

A "Core" namespace was created to provide access to the common items within the Utilities assembly. This contained the commands collection, the data layers, the logfile components and the system variables. All items within this assembly are shared to ensure there is only ever one instance of each component. This setup permits multiple commands to utilize this Utility assembly and use the available components without having to copy these components and have multiple instances. This can be problematic, especially for multiple connections to the SQL database, so the shared, single instance approach is the most appropriate.

### 6.7 Variables

Overlord2 variables control how procedures execute at runtime and they pass data around between devices on the system and with the outside world. The type of an Overlord variable refers to the type of data it contains. It is important to be able to distinguish the difference between a variable that can store a string and a variable that can store numbers, since we would create an exception in the code if we tried to increment the value of a string. There are four types of variable available.
<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numeric</td>
<td>For variables that store numbers such as the current plate or the volume to dispense.</td>
</tr>
<tr>
<td>Text</td>
<td>For variables that store text such as the data path to write new generated plate data to.</td>
</tr>
<tr>
<td>True/False</td>
<td>Variable that is either true or false. Used typically for variables that indicate if the plates have lids or not.</td>
</tr>
<tr>
<td>Object</td>
<td>This variable type is used on more advanced setups and is used in association with Scripts. It allows Objects that were created in a Script to be persisted across multiple scripts. An example would be an instance of the Excel Application Object. This would mean this object would only need to be created once if we were using multiple times within a procedure.</td>
</tr>
</tbody>
</table>

Table 6 – The types of variable that can be used within Overlord2. The types have been named in a way to make them more understandable to non-programmers.

These variables can be accessed through code as shown by the example below which is setting the Overlord variable [Plate] to equal the value 10

```csharp
Overlord.Utilities.Core.Variables("[Plate]") = 10
```

The code above can be used to either 'get' or 'set' the value, get the type or get name of the variable requested.

The variables will be shared so that a single instance of the variables can be used throughout the main application and more importantly, any command that is being used. Direct access to the variable collection saves having to implement copying routines to pass values into commands. It also eliminates the problem that would arise if two commands simultaneously wanted to increment the same Overlord variable. This does mean that the variable collection has to be thread safe since only one instance of the collection will exist. The principles regarding thread safe code, discussed in section 3.5 were applied in this case.

6.8 Flowchart

The Flowchart is the “View” section of the MVP pattern that is core to this product. The flow chart interface is a graphical representation of the current Overlord Procedure (Model). It is used by the user to build up the procedure representation of an assay using a drag and drop mechanism where icons representing the Flow activity are dragged onto the chart.
A user control using GDI+ was considered as the method of drawing the Flowchart and providing the user a tool to create Flowcharts. GDI+ provides extensive methods for drawing within the .NET environment which allows most developers to draw shapes. However, this process still requires a level of expertise, considerable time and testing to develop an interface that can be interacted with by the user. A 3rd party component from the company MindFusion\(^1\) was prototyped. This component automatically provides methods of drawing common flowchart elements and standard Window functions for UI such as Clipboard actions (cut, copy, paste and undo). Analysis of this product soon revealed its suitability to providing the flowchart elements of the View component.

A user control has been developed, so that multiple instances of the flowchart can be used within a multi-document interface application. The Flowchart.NET component was sunk onto the user control and then customized to provide the functionality, as described in Section 5.1.2. A class within this control called DrawChart contains the code for drawing the flowchart based upon the supplied model. The main method is called DrawActions, which is analogous to the Start method used within the RunEngine (See Section 6.4)

```vbnet
Private Sub DrawActions(ByVal actionGroup As Model.ActionGroup, ByVal StartRow As Integer, ByVal StartCol As Integer, ByVal textv As Boolean)
    For iAction = 0 To actionGroup.Count - 1
        action = actionGroup(iAction)
        DrawAction(action)
        'Draw SubActions
        colLastDrawn = New System.Collections.Generic.List(Of Box)
        iRowOffset = 0
        For iActionGroup = 0 To action.Groups.Count - 1
            subActionGroup = action.Groups(iActionGroup)
            'Create new Instance of DrawChart Class to draw subactions
            objSubDraw = New DrawChart(_FlowChart)
            objSubDraw.DrawActions(subActionGroup, _Row + _StartRow + iColOffset + _StartCol, TextView)
        End For
        Next iActionGroup
    Next iAction
End Sub
```
CHAPTER 6

Code Example 26 – DrawChart Code

The same iterative pattern is applied to map the model to a visual representation. This handles drawing the branches off a decision action and the actions within a loop action.

At runtime, the flowchart needs to indicate which action is active. This is accomplished by highlighting the action that is active. Each action within the model has a StartAction and CompleteAction. These methods are mapped to local methods within each Flowchart Actionbox, so that when StartAction is called from the RunEngine, the relevant Actionbox is automatically highlighted on the Flowchart. This architecture utilizes the delegate capabilities in .NET to allow the RunEngine to automatically update the Flowchart without invoking it directly. When Drawchart creates each ActionBox, it passes in the Model action it is representing. This sets up the events that highlight and unhighlight the ActionBox.

```csharp
Public Property Action() As Model.Action
    Get
        Return _action
    End Get
    Set(ByVal value As Model.Action)
        If value IsNot Nothing Then
            _action = value
            AddHandler _action.ActionStart, AddressOf _action_Highlight
            AddHandler _action.ActionFinish, AddressOf _action_UnHighlight
            AddHandler _action.Updated, AddressOf _action_Updated
        End If
    End Set
End Property
```

Code Example 27 – Subscribing to Model Action events

6.9 Summary

The developed application contains the following four key components

- Flow Chart UserControl – The View component of the MVP architecture that provides a graphical representation of the Overlord Model (Procedure).
• Overlord RunEngine – Runs the selected Overlord procedure - interacts directly with all the other component.

• Overlord Model – Defines the Overlord procedure and all the actions within this procedure.

• Overlord Utilities – All the internal command, data, Base classes, Overlord variables and log file components are stored within this assembly. They are exposed within the Core namespace as shared single instance objects to be used by every assembly used within the application, including external commands to control instruments.

Each of these components delivers the functionality set out in Section 5.2 once integrated into the main executable, Overlord.Main.exe. This is the presenter part of the MVP pattern. This hosts all of the components listed above and binds them together to create the application.
CHAPTER 7 Final product

Starting the Overlord2 program will launch the main screen. All common functionality is available on this main screen to ensure the user can immediately start creating procedures to execute biochemical assays. The flowchart is the central component of the system, and is therefore presented with the largest child window within the MDI interface. All of the other components such as commands and variables surround this window to assist in creating a procedure.

![Figure 39 - The Overlord main screen that appears when the executable is started](image)

On the left hand-side of the window is the Command Palette. This contains the Overlord2 Commands that allow a procedure to be built. These commands are listed into three different categories, Flow, Plug-ins and Devices.
CHAPTER 7

7.1 Configuration

Although simple procedures can be created immediately, most of the time procedures are required that control instrumentation or work with data. The following section describes how Device Commands, for controlling instruments and Plug-in Commands for working with data can be added to Overlord2 and how these can be configured. This section also describes the different options that are available in Overlord2 for error notification and tracing, which are key features for automation software.

7.1.1 Adding Commands

To add a new instrument or data plug-in command into Overlord2, the user must click on the toolbar and select Setup and then click on the Command Setup button.

![Command Setup Button](image)

*Figure 40 – The Command Setup Button that is located on the Overlord2 toolbar*

This will bring up the Command Setup Window which will display Commands that have already been selected. If this is a fresh installation, the display will be empty as shown in Figure 41.

![Command Setup Window](image)

*Figure 41 – The Command Setup Window before any commands have been added*
To add a command, the user must click on the green cross in the top right corner of the window. Overlord2 will then scan for Overlord Drivers in a folder called “Command” which will reside in the same folder as the main executable. As .NET reflection is used to dynamically load the Overlord command .NET assemblies, it is important a consistent location is used to store all Overlord commands; otherwise the process of finding available Overlord commands on the PC will take longer than 10 minutes to scan every folder. Once Overlord2 has finished scanning the command folder, the ‘Add Commands’ dialog will appear listing all the Commands that have been found. This will only list the .NET assemblies that have implemented the base “Command” interface from the Overlord Utilities assembly as described in section 6.2.1.

![Add Command](image)

**Figure 42** - Add Command lists the Overlord drivers that are available on the system so they can be selected and added onto the system.

The user can select the Command to use and then click OK. This command will then appear in the Command Setup Dialog. In the example below in Figure 43, the Biotek ELX405 has been selected.
As the Biotek washer uses RS232 serial communication, the RS232 port of the PC needs to be entered into the Port text box. The port value differs depending upon the communication type of the command. For instance, a command that uses TCP/IP would have the IP address entered in as the port value. The type column indicates the communication method for the relevant command. The Alias name for each command by default is setup with the name of the Command as can be seen in Figure 43. This can be changed though and is typically changed to differentiate between two identical commands on the system. For example if there were two Biotek Washers, the first command would have the Alias “Washer 1”; the second Washer would have the name “Washer 2”.

After adding all of the commands, clicking OK on the Command Setup dialog will save these settings. The new commands will now be available to add into a procedure. Depending upon the type of command, they will either be in the Plug-in or Devices section of the Command Palette.

Each Overlord command will be provided with an associated help file. These files have the same filename as the command, but end with the suffix “.help”. This naming structure then allows the Dynamic help pane on the main screen to update with relevant topics if the command is clicked on the command palette or selected on the Flowchart.
7.1.2 Command Setup

Some commands have setup dialogs available for setting parameter specific to that command. Typical parameters are connection timeout values, the location of data files and the location of 3rd party software executables. Setup also brings up the Teach software for robotic arms and diagnostic software.
To access the Setup section of a command, the user must click on the small arrow that is next to the command in the command palette. This will bring up a small context menu and if the Setup is available, it will be listed as a menu entry. Clicking on this will bring up the Setup dialog for that command.

7.2 Writing Procedures

Procedures are created by dragging Commands across onto the flowchart to build up Actions. The way the procedure runs Actions can then be controlled by using Overlord variables. The following sections describe how all of these components work together to run an Assay and is followed with an example on how a simple procedure can be built up.

7.2.1 Procedure Flowchart

When Overlord2 starts, it opens with an empty procedure that consists of a Start Action, an empty Action and a Finish Action. All procedures contain a Start and Finish Action and these cannot be removed. Commands are dragged in between these two Actions to build up the procedure. A new procedure, if there isn’t already one open, can be created by going to the File Menu and selecting ‘New Procedure’. This will load the following window:
A Command can then be added into the empty Action box or inserted between existing Action boxes. The place where the command will be added will be highlighted so when the mouse button is released, it is clear where the command and resulting Action will be placed. If a Loop command is inserted between the Start action and the Empty Action, the procedure will expand to include this new Action.

Add any command onto an empty box and the procedure remains the same, the only difference being the Action in that place is updated. If a command is added onto a box that already contains an Action, a dialog will appear asking if it is desired to overwrite the existing Action. The basic rules are, dragging onto an arrow will insert the Action, dragging onto a box will overwrite the Action.
Existing Action boxes can also be moved within a procedure or deleted.

### 7.2.2 Overlord Variables

Overlord variables have a very important role in Overlord2. They control how procedures execute at runtime and they pass data around between devices on the system and with the outside world. There are two default variables on every system, [Plates] and [Barcode]. They have the following uses.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Plates]</td>
<td>Represents how many plates are to be run. Typically used to control how many times a Loop action should loop around.</td>
</tr>
<tr>
<td>[Barcode]</td>
<td>Normally used within a Barcode action. The read barcode is stored in this variable. This can then be used in a script to rename a file with the barcode of the plate.</td>
</tr>
</tbody>
</table>

The current list of variables on the system is listed in the Variables pane which is normally in the top right of the main window.
Extra variables can be added by clicking the green cross. The name of the new variable and its type can then be added.

The type of an Overlord variable refers to the type of data it contains. There are four types of variable available.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numeric</td>
<td>For variables that store numbers such as the current plate or the volume to dispense.</td>
</tr>
<tr>
<td>Text</td>
<td>For variables that store text such as the data path to write new generated plate data to.</td>
</tr>
<tr>
<td>True/False</td>
<td>Variable that is either true or false. Used typically for variables that indicate if the plates have lids or not.</td>
</tr>
<tr>
<td>Object</td>
<td>This variable type is used on more advanced setups and is used in association with Scripts. It allows Objects that were created in a Script to be persisted across multiple scripts. An example would be an instance of the Excel Application Object. This would mean this object would only need to be created once if we were using multiple times within a procedure.</td>
</tr>
</tbody>
</table>

**7.2.3 Actions**

Each time a command is dragged onto the procedure flowchart, an action is created. At runtime, each Action will execute a specific task using the Command it was created with. If a robot command is dragged onto the flowchart an edit screen will appear for that Command. If the robot program "Get Plate From Incubator" is selected on the robot command, then at runtime, that program will be executed. Alternatively, if a Loop command is dragged onto the flowchart, an edit screen will appear for setting options on how to loop. If the loop is set to loop 3 times, at runtime, it will do exactly that.
Each Action is similar in that they each have a set of properties. By highlighting an action and then right mouse clicking, the context menu will appear. At the bottom of this menu will be the entry “Properties”. Selecting this will bring up the Action Properties dialog.

![Action Properties](Image)

*Figure 50 – The Action Properties are available for every action within a procedure*

The table below describes each of the properties that are available and what effect they have on a running procedure.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wait until Idle</td>
<td>By default, when an Action is executed at runtime, the procedure execution will wait for the Action to complete. If this property is turned off, the Action will be started, but the procedure will carry on running moving onto the next Action. If the procedures are built up properly, Actions representing Devices (instruments) can be run in parallel which improves the throughput of the system that Overlord2 is running. This functionality is realised by the .NET asynchronous capabilities as detailed in Section 6.3</td>
</tr>
<tr>
<td>Breakpoint</td>
<td>If the Breakpoint property is set, the execution of the procedure will pause when it comes up this Action. This feature is typically used when testing the procedure for the first time. Typically, you turn a breakpoint on, on a Robot Action that was about to move a plate into an area of the Workcell where a collision could occur if the devices was not in the right state (e.g. a pipetting station with its arm in the position where the plate would be loaded)</td>
</tr>
</tbody>
</table>
### CHAPTER 7

<table>
<thead>
<tr>
<th>Enabled</th>
<th>By default, an Action is enabled. If this property is turned off, at runtime it will not be executed.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retry on error</td>
<td>If an error is generated on this Action at runtime and this property is turned on, the Action will automatically run again until an error is actually generated. The number of times it 'Retries' is set in the text box labelled 'Retry Count'.</td>
</tr>
<tr>
<td>Comment</td>
<td>The comment box allows a better explanation of what the Action is doing, to be associated with it.</td>
</tr>
</tbody>
</table>

#### 7.3 Example Procedure

This section will go through step-by-step how a procedure can be created in Ovelord2. This procedure will replicate a cell dosing assay using a variable number of microtitre plates. The resulting procedure will move a set of plates from a Thermo Kendro Carousel onto a Molecular Devices Aquamax using a Mitsubishi RV-2AJ articulated robotic arm.

The procedure will be configured as follows:

- There will be a main procedure that uses sub-procedures that move plates to and from the Aquamax,
- The procedure will allow a variable number of plates to be run,
- The procedure will allow the user to enter which program to run on the Aquamax,
- The Kendro will be setup to automatically retry on error,
• Each robot action will have break points for testing purposes

It is assumed that Overlord2 is already setup with devices described above. It is also assumed that the Mitsubishi has already been programmed with the following four programs

- GPCYT – Get Plate From Kendro Cytomat
- PPCYT – Put Plate onto Kendro Cytomat
- GPAQ – Get Plate from MD Aquamax
- PPAQ – Put Plate onto MD Aquamax

The procedure requires two new Overlord numeric variables. For this assay, these variables will be called ‘CurPlate’ and ‘AquamaxProgram’.

Starting with a new Overlord Procedure, a User Input command is added that asks the user how many plates to be used. The value that the user enters will be stored into [Plates].

![User Input for Number of Plates](image)

Figure 51 - User Input for Number of Plates

Another User Input command is added that asks the user what Aquamax Program they want to run. This value will be stored into the Overlord variable [AquaMaxProgram]. A Loop command should be added after the last User Input action and configured to be a
fixed loop that loops \textbf{[Plates]} times. This is set so that the variable \textbf{[CurrentPlate]} is updated with the current Loop number.

\textbf{Figure 52 - Fixed Loop}

At this stage the procedure contains two User Input commands and an empty Loop. The next step is to fill in the loop with the Actions that will be run on each plate.

\textbf{Figure 53 - Empty Loop}
A Kendro Cytomat Series command from the Devices palette is added into the loop. The option to Unload a plate from the position [CurrentPlate] must now be selected. If the loop is set up to loop 5 times, then the first time through the value of [CurrentPlate] will equal 1. 2\(^{nd}\) time through it will equal 2, 3\(^{rd}\) time through it will equal 3 and so on until the final loop where [CurrentPlate] will equal 5. With this setup, each time the loop is executed, a different plate is unloaded from the Kendro Cytomat in sequential order.

![Kendro Cytomat 2 Edit](image)

**Figure 54** - Setup Cytomat to unload

A Mitsubishi MelfaRX command is inserted to pick up the plate from the Kendro since at this stage, the plate would be sitting on the Kendro Transfer station after being unloaded. The robot program, GPCYT is selected to pickup a plate: Add another Mitsubishi MelfaRX command, using the Mitsubishi program PPAQ to place the plate onto the Aquamax. A Molecular Devices Aquamax command is added to run a program on the Aquamax. This is configured so that it runs a program represented by the variable [AquamaxProgram].

110
At this point in the run, the Aquamax would have finished running the program so the plate can be returned to the Cytomat Incubator. Three commands are required for this. Two Mitsubishi commands are required to pickup the plate, GPAQ, and place onto the Kendro, PPCYT. Finally the plate can be returned into the position within the Kendro it originally came from: a Kendro Cytomat Series command to load the plate into position [CurrentPlate] is added.

The program is now complete and should be saved. This is saved as MAIN - Run Plates on Aquamax.ovp. The Kendro Cytomat actions must be set up to retry automatically upon error plus a breakpoint is required on the first Mitsubishi action. The first Kendro Cytomat action that unloads the plate is selected and the Retry on error property is set so that if there is an error the Cytomat will automatically retry before reporting an error.
(See section 7.2.3 Actions). This is repeated for the other Cytomat action within the loop that loads the plate back into the Cytomat Hotel. The first Mitsubishi command within the loop should also be selected so that the 'Breakpoint' property can be set.

At this stage, all of the Actions are with one procedure. One of the requirements for building up this procedure is to use sub-procedures for moving plates to and from the Cytomat. Building up sub-routines is good practice since it reduces the chance of writing the same blocks of actions multiple times and reduces the number of procedures required as common sub-routines can be reused. Biochemical assays often follow common steps, so this capability will save users time from having to replicate steps each time a new assay is required on the automation platform.

A new Overlord procedure should be created and docked next to the existing procedure.

The Cytomat Action and the Mitsubishi command that unloads a plate from the Cytomat and picks it up is selected. These are cut and pasted into the new procedure and saved as 'Get Plate from Cytomat'. A Call Procedure command is then added into the Main procedure to call this new procedure.

Figure 56 - Create Sub Procedures

The Cytomat Action and the Mitsubishi command that unloads a plate from the Cytomat and picks it up is selected. These are cut and pasted into the new procedure and saved as 'Get Plate from Cytomat'. A Call Procedure command is then added into the Main procedure to call this new procedure.
Another new procedure is created and then the last Mitsubishi action and Cytomat action from the Loop in the main procedure removed. These are cut and pasted into the new procedure and saved as ‘Put Plate on Cytomat.ovp’. Finally a **Call Procedure** command is added into the Main procedure where these two actions were removed that calls this new procedure.

The procedure is now complete with one main procedure, using two sub procedures that moves a variable number of plates onto an Aquamax and runs an Aquamax program that is set by the user at runtime.

**Figure 57 - Get Plate from Cytomat Sub-Procedure**
7.4 Advanced Procedure with Incubations and Data Handling

Incubations can be run in Overlord2 by using the Timers command. Allocating a Timer to a plate and telling it to run for a period of time is effectively the same as if the plate was being run manually using a stopwatch for the incubations. This functionality is key as it keeps the use of the product synonymous with running the assay manually. The user would wait, using a stopwatch, until the required time has elapsed before carrying on with the post incubation steps within the assay. Overlord2 can do exactly the same by telling the Procedure to wait until the timer in question has elapsed before moving onto the next set of Actions.

There are also other ways to monitor timers. Decisions and Loop command can query the state of a Timer. However, for this example the Timer command option "Wait until Complete" will be used.
The previous example from Section 7.3 will be extended so that after the plate is run through the Aquamax it will be incubated for one hour in the Cytomat before processing it through a Molecular Devices SpectraMax using Softmax Pro. Finally, the plate will be moved back into the Cytomat. The plates will also now be bar-coded so a barcode reader is required. The data generated by the SpectraMax will need renaming with the barcode and stored onto a network resource for a LIMS to pickup. The flow diagram below is an illustration of the assay to be automated.

Before modifying the existing procedure, the Metrologic Barcode reader and MD SpectraMax commands will need to be added using Instrument configuration. It is also assumed that the Mitsubishi has been programmed to additionally contain the following three programs.

- GPSM – Get Plate From SpectraMax
- PPSM – Put Plate onto SpectraMax
- BARCODE – Move Plate past barcode reader (always gripping the plate)
The procedure MAIN - Run Plates on Aquamax.ovp should be opened and saved as MAIN – Incubation Demo.ovp

A timer command is added after the step that puts a plate back into the Cytomat. This resulting Action will be telling the procedure that this plate needs to be incubated for one hour. This action sets the Timer represented by [CurrentPlate] and causes it to run for 1hr. Using [CurrentPlate] means that on the first time around the loop when Plate1 is being used, it will be assigned Timer 1. Plate 2 will have Timer 2; Plate 3 will have Timer 3 and so on.

Once all the plates are back in the Cytomat they will each have their own timer, each one elapsing slightly later than the next. If five plates were being run, five timers would have been created. Assuming the first plate was loaded into the Cytomat at 10.00am and the time taken to move a plate to the Aquamax, running the Aquamax and finally moving the plate back to the Cytomat takes 2 minutes, the incubation elapse cycle would look as follows.

<table>
<thead>
<tr>
<th>Plate Timer</th>
<th>Elapase Time (Incubation time of 1hr)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plate Timer 1</td>
<td>11:00 am</td>
</tr>
</tbody>
</table>
CHAPTER 7

<table>
<thead>
<tr>
<th>Plate Timer 2</th>
<th>11:02 am</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plate Timer 3</td>
<td>11:04 am</td>
</tr>
<tr>
<td>Plate Timer 4</td>
<td>11:06 am</td>
</tr>
<tr>
<td>Plate Timer 5</td>
<td>11:08 am</td>
</tr>
</tbody>
</table>

The next part of the Overlord procedure needs to wait for the relevant timer to elapse before moving the plate to the SpectraMax. Another loop is added into the procedure after the existing loop that again loops [Plates] times and uses [CurrentPlate] to represent the current loop number. A Timer command is added inside the loop and ‘Wait for Timer’ selected. This should wait until the timer represented by [CurrentPlate] has completed. The desired effect here will be that the procedure will not move plates to the plate reader until the Incubation has completed.

![Figure 60 - Add new Loop](image)

The steps to move the plate to the reader and back need to be added after the Timer command. A Call Procedure command is added and the OVP file ‘Get Plate from Cytomat.ovp’, a procedure created in the previous set of examples, is selected.

A Mitsubishi command to read the plate on the barcode must also be included. The Robot program BARCODE is selected. This is followed with a Metrologic Orbit command to take the barcode that was just read and stored it into the Overlord Text variable [Barcode].

Another Mitsubishi command is added using the program PPSM. It is assumed that the SpectraMax plate drawer is already open at this point so the robot can safely place the
plate. This Mitsubishi command is followed with a SpectraMax command using a method that has already been created in Softmax Pro, the software for running the reader in stand-alone mode. This is set so that with each read the data file C:\PlateData\Data.txt is created.

Once the reader has finished, the drawer will open automatically. A Mitsubishi command is added to pick up the plate using the program PPSM. Another Call Procedure command is added after this and the program ‘Put Plate on Cytomat.ovp’ selected.

All of the instrument calls are now complete. The only thing missing is a way of renaming the generated plate data with the barcode of the plate and then moving this onto a network resource. This can be done by using a Script command. This is added after running the SpectraMax. Assuming the command has been setup to use VBScript the following code is used.

```vbscript
Set objFSO = CreateObject("Scripting.FileSystemObject")
objFSO.CopyFile "C:\PlateData\Data.txt", "\LIMS\AssayData\" + _ [Barcode] + ".txt", True
Set objFSO = Nothing
```

*Code Example 28 - Renaming Plate Data File*

The procedure is now complete. This procedure shows how incubations can be added into an Overlord procedure by using timers. The entire assay has effectively been divided into 2 sub processes, *Run plates through Aquamax and Incubate* followed by *Run plates through Spectramax*. A general rule of thumb is that the number of incubations plus one is the number of sub processes required.
7.5 Running Procedures

Procedures are run from the window that they were created in. This section describes how a procedure can be run and also how it can be validated before a run event begins to ensure it is setup correctly. This section also describes how errors are handled in Overlord2 and how a run is logged.

7.5.1 Run

Once a procedure has been created it can be run by simply clicking the Run button on the Ribbon toolbar. The procedure is saved at this point and a message box will appear asking for the procedure to be saved if it is not. Before the run starts, the procedure and all of its sub-procedures are validated to ensure all the Actions can be executed (See Section 7.5.2 Validation). If this passes, the procedure will begin running.

At runtime the Command palette that is normally on the left hand side of the screen disappears, but apart from this the screen will appear the same as it does at Edit time. As each Action is highlighted it is executed and will stay highlighted until the execution is
complete. If a sub-procedure is called, this will be loaded onto the screen and execution will continue through the sub-procedure, as shown in Figure 62. The main procedure will stay open at this point and the Call Procedure action that made the call will remain highlighted so it is clear where execution will resume once the sub-procedure has completed.

Figure 62 - Calling a Sub-Procedure at Runtime

When an Action that is running a device, is executed, the status of this device will be updated in the Devices palette that is typically on the right hand side of the screen, as shown in Figure 63.
The run can be paused at any point by clicking the pause button on the ribbon toolbar. Pausing will stop the execution at its current point and will not execute any subsequent actions. If there is an action currently running or devices that were started asynchronously are busy, these will continue running. In effect, pause just stops the execution of new actions, it doesn’t pause currently running Actions.

To resume the run, the Run button is clicked on the ribbon toolbar. Alternatively, the stepwise button can be clicked. This moves the execution onto the next Action and then pauses the system again. This is useful when running a procedure for the first time for testing to ensure each Action completes as expected before moving onto the next Action. The procedure will also pause when it comes up to any Action that has a breakpoint set (See Section 7.2.3).

**7.5.2 Validation**

Validation checks that the procedure and all of its sub-procedures contain Actions that will run properly at runtime. Validation checks the following:

- Procedures that are being called within Call Procedure commands actually exist,
- Variables that are used in an Action exist on the system,
- That configuration settings have been made for certain devices,
- Syntax errors in VBScripts, VB.NET and C# Code Snippets.
Validation is run by clicking on the Validation button in the Runtime section of the Ribbon toolbar. It will also be run automatically when the Run or Stepwise button is clicked.

If there are issues with any of the Actions in the main procedure or its sub-procedures, these are listed in the Error List palette that typically is located at the bottom of the window.

![Error List](image)

*Figure 64 - Validation Error list*

The list explains what procedures and commands have issues and a description for each problem. Double clicking on the entry in the list will open that procedure if it isn’t already open and the action with the issue, highlighted.

If there are Validation errors, indicated in the list by the red circle with a white cross, the procedure cannot be run and will not run until the issue is resolved. Entries in the Validation Error list can also be Warnings. Validation warnings are listed alongside a yellow triangle with an exclamation mark inside. Warnings simply alert the user to double check that everything is setup as this action may fail if certain conditions are met. An example would be an Action that uses a device that has a dependency on a certain file existing in a certain folder. The warning for this action would simply tell the user to double-check that when the procedure gets to this action, that key file will exist. This wouldn’t be a validation error because maybe that procedure creates this key file at runtime.

### 7.5.3 Error Handling

Although validation does its best to prevent errors, these may still come up during a run. This can be due to many different things such as the following.

- Faulty instrumentation,
- Incorrect configuration (e.g. Device setup up on the wrong COM Port),

---

122
• Data files not generated in the correct format,

• Labware loaded into the wrong position.

Overlord2 will trap all errors that are encountered at runtime and will give the user the chance to respond to the error in three different ways:

• **Ignore** – The error will be ignored for this Command

• **Retry** – The Action will be sent to the command again and depending upon the command will either start from the beginning or will resume from where the error occurred

• **Abort** – The entire run will stop.

There are two different ways that errors are reported and it depends upon what type of command the error is generated in. If the command is a device (e.g. Robot, Plate Reader, Pipetting station etc) then the error will be raised in the Device List palette by highlighting in red the relevant error. If this is the only device in error, the Error palette will be filled in with the details of the error. The Retry and Ignore buttons will also become available giving the user the chance to respond to the error as seen in Figure 65. If there is more than one device in an error state at the same time, clicking on each device will bring up its error details in the Error palette.
If the command is a non-device, a Flow or Plug-in command (e.g. Message box, Script, Loop etc) the error message will be brought up as a Message box with the Abort, Retry & Ignore options available to the user.

All error responses are recorded into the Logfile which is described in more detail in Section 7.5.3.

### 7.5.4 Logfile

Every time a procedure runs, an entry is created in the Overlord Logfile so it can be seen exactly how the run progressed. The logfiles are accessed by clicking on the Logfile button in the Runtime section of the Ribbon toolbar.
Each time a run starts a new Logname is created using a time/date identifier with the format `<dd/mm/yyyy> <hh:mm>`. Every run that has ever been started will appear in the Log Name drop down list and is listed in reverse chronological order, newest first. Selecting the log to be viewed will populate the Log grid which lists all of the activities which occur during a run. There are different activities that happen during a run, known as RunLogTypes. These are listed below.

<table>
<thead>
<tr>
<th>Run Log Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RunStart</td>
<td>When the Run begins</td>
</tr>
<tr>
<td>RunFinish</td>
<td>When the Run completes</td>
</tr>
<tr>
<td>ActionStart</td>
<td>When an action is first executed</td>
</tr>
<tr>
<td>ActionFinish</td>
<td>When the action completes</td>
</tr>
<tr>
<td>ActionError</td>
<td>If there is an Error with the Action</td>
</tr>
<tr>
<td>UserResponseIgnore</td>
<td>The user has responded to the error by selecting Ignore</td>
</tr>
<tr>
<td>UserResponseRetry</td>
<td>The user has responded to the error by selecting Retry</td>
</tr>
<tr>
<td>UserResponseAbort</td>
<td>The user has requested the system to Abort</td>
</tr>
</tbody>
</table>

Figure 66 - Logfile Viewer
The RunLogType column on the LogGrid sometimes lists the name of a variable type, **Numeric**, **Text**, **True/False** or **Object**. This is because the Log also records when a variable value changes.

Each entry in the Log is listed with the time that it occurred. There is also an additional column next to this called Timer ticks. This is for trouble shooting systems where there is a timing issue and a log recording resolution less than 1 second is needed. This functionality is only for advanced users.

There are various drop-down boxes below the Log Name drop-down box. These are all filters which allow the Logfile to only display relevant information. For example if the Kendro Cytomat is selected in the Commands Filter drop down box, only entries in the Log that report usage of the Kendro Cytomat will be displayed. This filter can be reset by selecting ‘All’ in the filter list.

![Log FileViewer](image)

**Figure 67 - Filtered List**
CHAPTER 8 Implementation & Case Studies

8.1 Implementation

The first beta version of Overlord2 was released to market in September 2007. At first this was rolled out on what are classed as small systems. A small system typically consists of a robotic arm or stacker feeding plates to no more than three instruments. Such a system was delivered to a diagnostic company in the United States. This consisted of a Thermo RapidStak plate stacker, a Molecular Devices SpectraMax plate reader and a barcode reader. The official release was November 2007.

There have been over 50 installations of the software on many different forms of laboratory automation platforms, running different biochemical assays. The following section details some of these installations to illustrate the flexibility of the software and how novel software features have improved the implementation of laboratory automation.

8.2 High Content Reader system

The GE Healthcare LifeSciences IN cell 1000 is a high content reader, used for High Content Screening (HCS) and High Content Analysis (HCA) to produce high quality images of cells. HCS uses cells to screen multiple parameters instead of the single parametric nature of a High Throughput Screen (HTS). paa have become the official automation partner for GE LifeSciences and have now installed 12 systems, using Overlord2®.

The IN cell 1000 is a large investment for any pharmaceutical or biotechnology company. Adding automation onto this device gives the following advantages:

- Extension of the working day, by running plates overnight and ensuring that the next plate is analyzed when the instrument has completed the previous plate analysis.

- Improved data integrity, by tracking the sample microplates using barcodes and then renaming IN Cell 1000 data files automatically with the source microplate barcode.
• Reduction in per plate analysis costs because a user does not need to feed the instrument with plates allowing them to undertake more fulfilling and profitable activities

The system that has been supplied to the customers, using Overlord2, typically consists of the following devices

• SCARA Robotic arm with “Last in-First out” stacks
• GE Healthcare LifeSciences IN Cell 1000
• Barcode reader
• Two plate exchange nests

Figure 68 - GE Healthcare Lifescience IN Cell 1000 Robotic System with a SCARA arm that loads microtitre plates onto the plate reader. This system has a capacity of 90 unlied microtitre plates

A typical read time on the IN Cell 1000 will take 30-60 minutes, due to the many data points being acquired. To maximize throughput, it is essential that the rate limiting step is utilized as close to 100% as possible. This is achieved by preparing the next assay plate whilst the current plate is being read. When the read completes, a rapid plate
exchange followed by starting the next run ensures the IN Cell 1000 is idle for less than 15 seconds. Overlord2 allows this type of set of steps to be sequenced due to the asynchronous nature of the commands. While the IN Cell 1000 is running, the robot can be getting the next plate from the stack and loading it onto the transfer nest.

8.3 ELISA System

A cell based ELISA system was delivered in January 2008 to a major pharmaceutical company in the UK. Central to this system was a Staubli TX40 articulated arm. The workcell containing this arm was protected using a light curtain system, to keep the deck space accessible, but at the same time ensuring the user could not enter the working area when the robotic arm was moving. The common instrumentation for an ELISA assay were also part of the system. These included

- Tecan Powerwash -- For removing reagents
- Thermo Multidrop Combi with Multi-valve position -- Reagent dispenser, capable of dispensing up 32 different reagents
- LiCONiC STX500 @ 37°C -- Incubating cell plates
- Barcode Reader -- For tracking plates throughout the system.
The Enzyme-Linked Immuno-Sorbent Assay (ELISA) is a biochemical technique used to detect the presence of an antibody or antigen in a biological sample. An ELISA workflow can be broken down into a number of discrete processes, and each individual ELISA protocol employs some or all of these steps. Each step commonly involves the removal of the previous reagent, the addition of a new reagent and a subsequent incubation period.

The user requirements demanded that the system would need to be able to handle multiple assay permutations. The system was designed to perform the entire workflow of a cell-based microtitre plate (MTP) assay involving multiple wash steps, multiple reagent additions, and varying incubation periods. Each assay could use either 96 or 384-well plates, and typically contained up to 200 plates. It was a core request that assays should be set-up easily, with the system running through the entire assay workflow with no additional user input.

As there were many different parameters that the user needed to enter to start a run, it was considered important, by the customer, that the user interface was as simple to use as possible. It was also necessary to provide a mechanism to allow assays to be
queued so that a series of assays could run overnight. This would then allow the customer to maximize the ROI (return of investment) in the system. As a Model-View-Presenter architecture had been adopted for Overlord2, a new presenter, customized to fit in with the customers' exact needs, could be utilized. Overlord methods would still be generated in the main product, but general day-to-day use was performed using a new graphical user interface for this particular customer called "Batch Manager".

8.4 Dual Robot HTS System

Overlord2 was supplied on the "Eve" system, which is part of the Robot Scientist project at UWA (University of Wales, Aberystwyth). The system is essentially replicating the processes used by the major pharmaceutical companies to do high-throughput drug screening. What makes this system unique is that it automatically analyzes the results using artificial intelligence algorithms to formulate the next assay and associated parameters.

Figure 70 - The "Eve" dual robotic system at the University of Aberystwyth

The automated system runs three main modes of operation:
- reformatting of the chemical compound library to a working copy
- mass screening of the working library
- cherry picking of specific compounds from the working library.

The concept of the research at UWA is to build an 'automated laboratory', so the system needed to be able to switch from one mode to another with minimal human intervention. This is controlled at a high level by software written by UWA, operating in a layer above Overlord2. The architecture of Overlord2 and in particular the ability to instantiate the run engine component made this possible. The drug discovery process that UWA wanted to run on the system is essentially an iterative process that runs an assay, analyzes the results and formulates a new assay based upon these results. Each time, a different assay is executed by the high level software with different parameter values for the relevant Overlord procedure. The Overlord2 framework is unique in offering this functionality: the ability to use select components from the software package, without having to invoke the entire application.

![UWA High Level Software](image)

**Figure 71** – The basic architecture of "Eve" system that controls Overlord2 from a higher level software layer that formulates the experimental parameters to run on the system

The Eve system at UWA was actually made up of 2 robotic arms. The reason for this was mainly to do with the requirement that the system needed to incubate and read assay plates, whilst simultaneously making up new assay plates. For UWA, it was critical that assay plates, stored in the incubator were read precisely every 30 mins on a fluorometer to see if a reaction or a "hit" had occurred. At that point, the plate would be read on a High content reader, in this case, a Molecular Devices IXM reader. However,
new plates would need making up at the same time on the acoustic dispenser, a device capable of dispensing extremely small volumes of reagent, down to 2.5nl. To meet the throughput requirements of the system, a two robotic arm system was proposed that effectively divided the system into two smaller workcells. This was a good design choice as it simplified the process, which is a major factor to ensuring the system is adopted by the customer after the system is delivered. Robot 1 could prepare the assay plates, whilst Robot 2 would perform the incubation and read process. From a software perspective, this would also simplify the implementation, as one instance of Overlord2 could be used for each robot.

![Figure 72 - A bi-directional extension of Overlord2 remoting has been implemented to allow System 1 to request use of System 2 at select points within the procedure running on System 2.](image)

However, there was a complication in that once an assay plate was prepared by Robot 1, it would need to be passed to Robot 2 and introduced into the Read incubate process. This initially would involve Robot 2 picking up the plate from a position that Robot 1 has loaded with the new plate and moving it into the incubator to start its first incubate read cycle. To do this, the process running on Robot 2 performing the read incubate would need to be interrupted so that it could load in the new assay plate, and then continue the read-incubate cycle.
The simplified setup, having one instance of Overlord2 running the mass screening and then another instance of Overlord2, each on different PCs, running the Read incubate process was still the preferred solution. A single Overlord2 procedure to run both the Mass screen and read incubate would have been extremely large and difficult to comprehend. A mechanism to allow System 1 to interrupt System 2 needed to be developed.

A simple method of interrupting two processes is to use a watch file. In this scenario, System 1 would write to the watch file that a plate has been prepared and is ready. System 2 would continually poll this file until a value inside it indicated that the plate was ready. An acknowledge file would also be used so that System 2 could indicate to System 1 that it has moved the plate, allowing System 1 to continue preparing the next assay plate.

There are disadvantages to using watch files, the main issue being file locking. Having two resources accessing a common file is prone to causing locking, whereby simultaneous accessing of the file can cause the application to hang, since the practice is not thread safe, or the wrong value to be read from the file. In this case, this would be a severe error, as it could lead the robot to access the position when the other robot occupies the same space.

The solution was to extend the .NET remoting capabilities of Overlord2 so that Overlord2 could be both a remote listener and remote client. As discussed in Section 5.4, Overlord2 could run a procedure on another instance of Overlord2, running on a different PC. This is achieved by the remote listener, when invoked by the client, creating a new run engine that executes the selected process. The mass screening process on System 1 would have a remoting command which would run a procedure on System 2 that picks up the plate and moves it to the incubator.
Figure 73 – The Bi-directional Remoting architecture that allows System 1 to run processes on System 2 even when System 2 is running a local Overlord procedure

With this architecture, it would not be possible to have the read incubate running at the same time. To achieve this, the remote listener component was integrated into Overlord2. The read incubate cycle could then run. Locking commands were developed that could be used with the read incubate procedure to indicate when System 1 could invoke a procedure on System 2. Using this thread safe approach ensures that both the local and remote run engine on System 2 were never simultaneously active.

This solution addressed the issue of keeping the two procedures, mass screen and read incubate, separate whilst including the capability to interrupt and use the other systems resource where necessary. The remoting architecture finally implemented was bi-directional, allowing System 2 to also control System 1. Although not an original advantage, this was deemed a significant additional feature by UWA as it provided access to plate stacks that could only be reached by Robot 1. This provided far greater capacity than originally required.

The project at UWA proved that it was possible to use Overlord2 within a larger scale automation platform and retain its simplistic procedures, one of the major goals of the project. Modifications can be made to either the Mass screening or read incubate by the user, safe in the knowledge that a change to one procedure, will not affect the running of the other procedure. This architecture is not restricted to the system at UWA: it can be...
used on any system where there is a single instrument, such as a compound store, serving two or more automated platforms. This ensures that, as more platforms are served by the common device, the complexity of the Overlord2 procedures running the entire system are scalable and retain their simplicity.

8.5 Third Party Integration

The Overlord2 Framework, which is made up of the model component of the MVP architecture, the run engine and common utilities was designed to allow staff at paa to rapidly develop new commands to control instruments or manipulate data. The systems described in section 8.3 and 8.4 show how the run engine can be used within a new custom GUI and how existing commands can be extended to solve new problems. All of these examples were implemented by paa. Experience using the framework has proved it to be both powerful and simple to use. However, this opinion can be considered biased since it has been developed by the same person. Although the framework capabilities of other products such as Beckman SAMI and Thermo POLARA were evaluated, and used as benchmark to improve upon, the ultimate measure of success would be the implementation by a 3rd party.

A major automation supplier in the UK decided to use Overlord2 to control a large system they were delivering to one of the major pharmaceutical companies in the UK. This system was designed to allow the compound management group at this pharmaceutical company to create assay-ready plates for screening using acoustic dispensers. The automation supplier wanted to concentrate on delivering the compound management software. It was deemed cost effective by the automation supplier to use an existing product for controlling the automation, as opposed to developing this component themselves. This was a new approach by this particular automation supplier. They had previously not considered using a 3rd party application to control the automation due to the products available lacking the control flexibility and inadequate integration capabilities. After a software evaluation period, Overlord2 was considered mature enough to meet the requirements for this project. This was only achieved by the novel approach taken in the flow chart solution to process development of assays and the MVP architecture adopted.
CHAPTER 9 Conclusion and Future Work

There is continual need within the pharmaceutical and biotechnology industries to deliver potential candidates out of the drug discovery pipeline. Laboratory automation is a tool available to chemists and biologists to assist this process. Nearly all pharmaceutical and biotechnology companies use laboratory automation in some form, be it a simple workstation or a large robotic system using multiple instruments. Laboratory automation has been used for over 25 years and has matured dramatically during this time. However, there is still a fundamental issue with users feeling uncomfortable using the technology available to them. A major contributory factor to the reluctance in using automated systems is the control software. The software currently available to run laboratory automation systems is not simple to use. Advanced training is required to become proficient in creating and then executing biochemical assays. Evidence for this situation can be found at many pharmaceutical and biotechnology companies where selected individuals are identified to be the interface between the software and the scientist.

A novel approach has been taken into developing a flowchart interface that is much more familiar to the user with the aim to eliminate the common problem of only advanced users being able to use the system to its full potential. By using a clear and simple flowchart interface as the basis for defining assays, automation systems using this software are much more accessible to the chemists and biologists in the laboratory.

9.1 Final Evaluation

There is much discussion within the industry on the best method of scheduling an assay. The decision for this software product was to use "event-driven" scheduling. The reasons for this choice has been described in Section 4.2, however it is appreciated that although suitable now, new methods of running an automated assay will appear. Even now, there are new methodologies being presented to tackle this key aspect of running an automated system, one being the use of Petri Nets as a way of running a dynamic schedule64.
The software architecture that has been used allows the user GUI, runtime engine and instrument control to be completely separate. This means that if the scheduling method needs to change in the future due to improved methods or algorithms, this can be accomplished without having to rewrite the control software for the entire product. Only the run engine component would need updating. This could lead to a scenario whereby the user can select the scheduling method of choice or the application could make this choice based upon the assay to be run. From a technical point of view, provided each component had the same interface, there is no limit to the number of different Run Engines that could be made available to accompany the product. By documenting this interface, it should also be possible for a user who is proficient in programming to create a Run Engine. The software can then really be tailored to the way his or her organization works. The MVP pattern separates the code out into assemblies that makes this entirely possible.

Although not an original requirement, this ability for programmers to customize key components of the product is an added bonus. Future work on this software once released to market would centre on this functionality.

The final product can provide the flexibility required to handle the many different assays that are suitable for automation. The flow chart interface is perfect for handling multiple types of homogeneous and heterogeneous biochemical assays. If customization is required, the software package provides a powerful, yet simple to implement SDK. Even if there is a fundamental shift in how laboratory instruments should be automated, the outcome of this work is a software application that can handle most eventualities whilst still being an accessible tool for most bench top scientists.

9.2 Future Work

Future work on this project will ultimately be dictated by the direction of the pharmaceutical and biotechnology industries. For example, the drug discovery process described in Section 1.1.2 covers the discovery of synthetic compounds that show drug like properties to the selected target disease. Since much more target information has made available since the publication of the Human Genome, research has been carried out to find more selective methods of treating the gene that is responsible for the condition. A result of this research has been the use of Biologics. Biologics has come to the forefront as an alternative to using synthetic compounds for certain targets that
involve protein-protein interactions and the targeting of surface features of particular
cells. Biologics uses selective proteins which are formed by genetically engineered cells.
Nearly all pharmaceutical companies have potential candidates in the drug discovery
process that have come from Biologics as do many biochemical firms. Worldwide,
biologic drugs—now a common treatment option for people with conditions such as
multiple sclerosis, diabetes, cancer and rheumatoid arthritis—account for about one of
every eight prescriptions. Laboratory automation software, as a tool to aid drug
discovery, needs to ensure that it can assist these new areas of research as and when
they appear.

The tools for qualifying the acceptance of a user interface should also be investigated,
developed and then implemented as a method of improving the usability of the final
product. Although NUnit testing has been used throughout the development cycle, it is
not suitable for testing UI interactions. Test suites such as NUnitForms are available;
however, they are still restricted to testing the code within a windows form. They cannot
test the user interactions such as "drag and drop", which are key activities for building
procedures in Overlord2. A tool that can test and then objectively evaluate the UI will
allow the interface to be compared in a qualitative manner to other products and provide
metrics for further development of the product in the future.
OVERLORD Customer Feedback Questionnaire

Thank you for taking the time to fill in this questionnaire and give us your feedback on OVERLORD. As a valued user, we welcome your input in order to continually improve OVERLORD, and make future releases more suitable for your needs.

We welcome constructive criticism as much as praise, but please be as specific as you can with your answers. Whether you like a particular feature, or think that something could be improved, try to tell us why so we can carry these ideas into future versions.

Name (optional): Withheld

Company (optional): Forensic Science Service

☑ Please tick the box if we can contact you to further discuss your feedback.

Editing

Think about your experiences with creating a new macro and editing existing macros, then answer the following questions.

| Does the range of commands available allow you to create the functionality that you require? If not, what other commands would be useful? |
| Open two macros at a time |
| To call a procedure you haven't yet created |
With regard to names, icons, and location on screen, do you find it easy to locate identify the commands you require? If not, why?

OK

<table>
<thead>
<tr>
<th>Questions</th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do you use drag and drop from the command list to add commands into the macro?</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Do you Left hand mouse click then menu to access commands to add into a macro?</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Do you Right hand mouse click then menu to access the commands to add into a macro?</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>

Are there any commands in particular you find useful?

A lot of calling proc and variable setting

With regard to command edit screen, including those for functions (e.g. Set Variable) and instruments, do you find them intuitive and easy to follow? If not, why?

Very simple (which is good)

How do you think the edit screens for creating Overlord macros could be improved for ease of use for the general user?

141
More than one open at a time.
Expand the info in the command, Like windows explorer. Then you could view more than one command at a time

Do you have any other comments with regards to creating Overlord macros/procedures? What would make it simpler to setup an assay?
Delete and copy more than one command at a time

Runtime
The following questions are regarding how OVERLORD works when it is running a macro

Do you find the functions that ask the users questions and show user messages easy and intuitive to use? If not, why? How could these be improved?
Yes the image and command we have is very useful.
When it crashes or stalls, more information would be good

Currently when a command is active it is highlighted in yellow. Does any other information need to be displayed at this point? If so what?
All the parameters could be displayed in the status bar

Do you find the status of system easy to follow at runtime? If not, why?
Yes

Are there any changes to the on-screen display at runtime that you think would be useful?

Put break points in for troubleshooting

Currently when there is an error with an instrument, the options Abort, Retry and Ignore are displayed with a description of the error. Do you find this provides enough functionality to recover from any error or is there more options you would like to see?

Pause and then allow changes. So you can start from the failed point with all variables intact

Any further comments?

Please use the space below, and additional sheets if necessary, to tell us any other comments you have about ANY aspect of OVERLORD and what features you would like to see in the next version of OVERLORD
OVERLORD Customer Feedback Questionnaire

Thank you for taking the time to fill in this questionnaire and give us your feedback on OVERLORD. As a valued user, we welcome your input in order to continually improve OVERLORD, and make future releases more suitable for your needs.

We welcome constructive criticism as much as praise, but please be as specific as you can with your answers. Whether you like a particular feature, or think that something could be improved, try to tell us why so we can carry these ideas into future versions.

Name (optional): Withheld

Company (optional): GSK

☐ Please tick the box if we can contact you to further discuss your feedback.

Editing

Think about your experiences with creating a new macro and editing existing macros, then answer the following questions.
APPENDIX A

<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Does the range of commands available allow you to create the functionality that you require? If not, what other commands would be useful?</td>
<td>Yes</td>
</tr>
<tr>
<td>With regard to names, icons, and location on screen, do you find it easy to locate identify the commands you require? If not, why?</td>
<td>Yes</td>
</tr>
<tr>
<td>Do you use drag and drop from the command list to add commands into the macro?</td>
<td>Yes</td>
</tr>
<tr>
<td>Do you Left hand mouse click then menu to access commands to add into a macro?</td>
<td>No</td>
</tr>
<tr>
<td>Do you Right hand mouse click then menu to access the commands to add into a macro?</td>
<td>Yes</td>
</tr>
<tr>
<td>Are there any commands in particular you find useful?</td>
<td></td>
</tr>
</tbody>
</table>

145
With regard to command edit screen, including those for functions (e.g. Set Variable) and instruments, do you find them intuitive and easy to follow? If not, why?

Yes, would be nice to have access to the variable definition window from this screen.

How do you think the edit screens for creating Overlord macros could be improved for ease of use for the general user?

I find the manner for adding and inserting lines tedious i.e. it would be easier if I didn’t have to have an empty box to drop a command into. I would rather be able to drag & drop the command between two existing lines, or if a line is inserted a menu of command to add should popup.

Do you have any other comments with regards to creating Overlord macros/procedures? What would make it simpler to setup an assay?

Option to have one dialog to allow the setting of multiple variables rather than several dialogs. Also the use of other vb controls such a checkbox and option buttons would be good.

### Runtime

The following questions are regarding how OVERLORD works when it is running a macro

Do you find the functions that ask the users questions and show user messages easy and intuitive to use? If not, why? How could these be improved?
Currently when a command is active it is highlighted in yellow. Does any other information need to be displayed at this point? If so what?

I'd like more detailed information on what is going on and perhaps display of variable values e.g. loop 1 acquiring Magellan data to file data.

Do you find the status of system easy to follow at runtime? If not, why?

Are there any changes to the on-screen display at runtime that you think would be useful?

I'd like user and expert views. The user would pick method, enter variables and then have just a status window without the current flow diagram. This would allow simpler use by general user. The expert view would have all the information perhaps with more diagnostic information.

Currently when there is an error with an instrument, the options Abort, Retry and Ignore are displayed with a description of the error. Do you find this provides enough functionality to recover from any error or is there more options you would like to see?
<table>
<thead>
<tr>
<th>APPENDIX A</th>
</tr>
</thead>
</table>

No experience.

Any further comments?

Please use the space below, and additional sheets if necessary, to tell us any other comments you have about ANY aspect of OVERLORD and what features you would like to see in the next version of OVERLORD.
Primarily use Scheduler as main program and OVERLORD as a slave. It would be a great feature if Scheduler could communicate with OVERLORD and edit OVERLORD methods on another computer from in Scheduler. Not really an OVERLORD feature but desirable.

On a similar note however it would be neat if OVERLORD could open up other software for changes in methods.
OVERLORD Customer Feedback Questionnaire

Thank you for taking the time to fill in this questionnaire and give us your feedback on OVERLORD. As a valued user, we welcome your input in order to continually improve OVERLORD, and make future releases more suitable for your needs.

We welcome constructive criticism as much as praise, but please be as specific as you can with your answers. Whether you like a particular feature, or think that something could be improved, try to tell us why so we can carry these ideas into future versions.

Name (optional): Withheld

Company (optional): Ionix Pharmaceuticals Ltd

Please tick the box if we can contact you to further discuss your feedback.

Editing

Think about your experiences with creating a new macro and editing existing macros, then answer the following questions.

Does the range of commands available allow you to create the functionality that you require? If not, what other commands would be useful?
On the whole, the commands are sufficiently useful that most functionality can be accommodated, although the addition of a SELECT CASE statement would be a nice enhancement. From experience, this is a more useful way of controlling functionality based on user input, and it would be nice to be able to incorporate this functionality without having to resort to VBScripts. It also avoids the need for having a numbers of nested IF...THEN...ELSE statement.

With regard to names, icons, and location on screen, do you find it easy to locate identify the commands you require? If not, why?

It is reasonably easy to find the commands you need, although having all the commands listed in a single column is less than ideal as it leads to a lot of scrolling up and down the list to find the one you are after. Despite the split-screen look of the Commands/Macro interface there isn't the ability to resize the division. This would allow you to increase the width of the Commands panel so that you could have multiple columns visible (ie. 3 columns of 10 icons, rather than 1 column of 30 icons).

<table>
<thead>
<tr>
<th>Question</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do you use drag and drop from the command list to add commands into the macro?</td>
<td>Yes</td>
</tr>
<tr>
<td>Do you Left hand mouse click then menu to access commands to add into a macro?</td>
<td>Yes</td>
</tr>
<tr>
<td>Do you Right hand mouse click then menu to access the commands to add into a macro?</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Are there any commands in particular you find useful?
The VBScript command is very useful as it allows you to provide the functionality that would be quite longwinded to implement otherwise. VBScripts tend to be used to implement SELECT CASE statements and also for condensing the number of Macro lines taken up with setting variables, e.g., it is often more convenient to have one VBScript that sets 10 variables rather than having 10 Set Var statements listed in a Macro. It also allows comments to be added as to why variables are being set to specific values. The ability to Push & Pull variables is also useful.

With regard to command edit screen, including those for functions (e.g., Set Variable) and instruments, do you find them intuitive and easy to follow? If not, why?

The majority of the command edit screens are fine, however there some could be improved. For instance the Equation and String Manipulation screens are usable, but would be enhanced if a Comments box was available so that the reasons for the calculation/manipulation could be provided. Whilst this doesn’t add to the ease of writing the Macro, it makes it significantly easier to figure out what is going when editing a macro if you haven’t used it for some time, or one that has been written but someone else. Also, the ability to launch a Netlord command and a Wait Until Finished as a single step rather than two separate ones would be nice whilst still retaining the ability to use them independently.

How do you think the edit screens for creating Overlord macros could be improved for ease of use for the general user?
As mentioned above, the ability to add comments. A more consistent interface for some of the instrument edit screens (ie, for the Evolution P3 it is possible to browse for the file you want to run, but on for MultiPROBE it is only possible to select from files in a single (hardcoded?) directory). Also, the ability to see the entire filename rather than just the first ~20 characters would make picking the correct file much easier, especially when the target files have long names. Incorporating a Write to Log feature, and the ability to specify a target file, into the Barcode Reader screen would also be a nice enhancement, rather than relying on additional VBScript to accomplish this task.

Do you have any other comments with regards to creating Overlord macros/procedures? What would make it simpler to setup an assay?

Perhaps the biggest improvement would be the ability to have more than one Macro open at a time. This could be further enhanced by a feature that would allow you to select multiple steps in a Macros and Cut, Copy or Paste them to other locations either within the same Macro, or between different ones. Similarly, the ability to drag & drop commands from one place in a Macro to another, rather than copy & pasting them would be a significant improvement. It also seem remarkably easy to make a mess of the Macro when inserting/deleting lines within nested structures as the Macros rearranges itself. The edit interface might be improved if the Commands panel was replaced with function-specific toolbars (eg. Instruments, Program Flow, Input/Output etc). Also, the availability of keyboards shortcuts in addition to mouse gestures would be nice. An additional feature that may be useful would be a Password Project option that could be used to prevent finished Macros from being inadvertently edited. Perhaps slightly more advanced, syntax highlighting/checking in the VBScript editor would be nice.

**Runtime**

The following questions are regarding how OVERLORD works when it is running a macro
Do you find the functions that ask the users questions and show user messages easy and intuitive to use? If not, why? How could these be improved?

Generally these dialog boxes are okay, although the ability to create custom forms would be useful. These would provide a means of displaying more information to the user (eg, and image rather than a lengthy description), and also a way of restricting or validating the nature of inputs the user can supply (selecting from a list of option buttons rather than relying on a correctly entered value or string). Although, admittedly, this is likely to make writing the Macro more complicated.

Currently when a command is active it is highlighted in yellow. Does any other information need to be displayed at this point? If so what?

This feature works nicely, although the information supplied in the status bar is rather subtle. Perhaps it could be enhanced with an additional Status panel. This could be used to display sequential information about the steps that are being, or have been, executed. In addition, a progress bar would be a nice feature and provides a quick visual indication as to how much of the Macro has been processed (especially useful if loops involved as it isn’t always immediately apparent which iteration is being processed), and also be used to display any error messages that arise.

Do you find the status of system easy to follow at runtime? If not, why?

For processes happening on the Master PC, it is reasonably easy to follow the system at runtime. However, when the system is executing procedures on a Slave PC via Netlord, then the screen tends to be less informative. Maybe the solution is for the Master PC to have the ability to display the Macros that the slave PC is running and highlight the step that is being processed. For instances where both the Master and Slave PCs are running simultaneously, maybe the possibility of having a split screen displaying the progress of both Macros could be employed.
Are there any changes to the on-screen display at runtime that you think would be useful?

See answers in the above two boxes that address this issue.

Currently when there is an error with an instrument, the options Abort, Retry and Ignore are displayed with a description of the error. Do you find this provides enough functionality to recover from any error or is there more options you would like to see?

For errors that occur within Overlord, the message boxes tend to provide enough information that allows you to find the cause of the problem. For errors that occur on instruments under the control of Overlord, things aren’t so useful. For instance, if our Evolution P3 suffers a problem, Overlord simply waits on the icon without giving any indication that there is a problem. Trying to stop the Macro using the traffic lights doesn’t seem to work and you are unable to get to the WinPREP software to see what the problem is. This results in using Task Manager to kill Overlord and the associated programs running.

Any further comments?

Please use the space below, and additional sheets if necessary, to tell us any other comments you have about ANY aspect of OVERLORD and what features you would like to see in the next version of OVERLORD
OVERLORD Customer Feedback Questionnaire

Thank you for taking the time to fill in this questionnaire and give us your feedback on OVERLORD. As a valued user, we welcome your input in order to continually improve OVERLORD, and make future releases more suitable for your needs.

We welcome constructive criticism as much as praise, but please be as specific as you can with your answers. Whether you like a particular feature, or think that something could be improved, try to tell us why so we can carry these ideas into future versions.

Name (optional): Witheld

Company (optional): UCB-PHARMA

☑ Please tick the box if we can contact you to further discuss your feedback.

Editing

Think about your experiences with creating a new macro and editing existing macros, then answer the following questions.

Does the range of commands available allow you to create the functionality that you require? If not, what other commands would be useful?

- A DDE interface for easy connection with EXCEL or ACCESS.
- To be able to add ACTIVEX technology to integrate other software.
- A "case of "structure like in basic language.

With regard to names, icons, and location on screen, do you find it easy to locate...
<table>
<thead>
<tr>
<th>Question</th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do you use drag and drop from the command list to add commands into the macro?</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Do you Left hand mouse click then menu to access commands to add into a macro?</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Do you Right hand mouse click then menu to access the commands to add into a macro?</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Question</th>
</tr>
</thead>
<tbody>
<tr>
<td>Are there any commands in particular you find useful?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Question</th>
</tr>
</thead>
<tbody>
<tr>
<td>With regard to command edit screen, including those for functions (e.g. Set Variable) and instruments, do you find them intuitive and easy to follow? If not, why?</td>
</tr>
<tr>
<td>Some bugs appear concerning SET VARIABLE when we try to play with EQUATION.</td>
</tr>
</tbody>
</table>

How do you think the edit screens for creating Overlord macros could be improved
for ease of use for the general user?

To increase number of columns (for example when we have to imbricate a lot of IF THEN structures.

Do you have any other comments with regards to creating Overlord macros/procedures? What would make it simpler to setup an assay?

Parallel processing for multithreading.

Runtime

The following questions are regarding how OVERLORD works when it is running a macro

Do you find the functions that ask the users questions and show user messages easy and intuitive to use? If not, why? How could these be improved?

To be able to enter Boolean variables (for example for YES /NO question).

Currently when a command is active it is highlighted in yellow. Does any other information need to be displayed at this point? If so what?

See next case.
Do you find the status of system easy to follow at runtime? If not, why?

We would like to see variable values at runtime (for example "plate number") to know the status of the protocol.

Are there any changes to the on-screen display at runtime that you think would be useful?

to be able to see the complete protocol preview in a separate windows and not only the running subprogram.

Currently when there is an error with an instrument, the options Abort, Retry and Ignore are displayed with a description of the error. Do you find this provides enough functionality to recover from any error or is there more options you would like to see?

A complete description of the error for all instruments should be displayed.

It will be very useful to be able for each instrument to program a automatic retry number, so system will not be stopped when an erratic error occurs.

Any further comments?

Please use the space below, and additional sheets if necessary, to tell us any other comments you have about ANY aspect of OVERLORD and what features you would like to see in the next version of OVERLORD.
To increase number of variable and/or to have a functionality to associate a file containing variable names to be associated with a protocol (sometimes, we have to use a variable with a name which doesn't correspond to the function of the variable)

OVERLORD Customer Feedback Questionnaire

Thank you for taking the time to fill in this questionnaire and give us your feedback on OVERLORD. As a valued user, we welcome your input in order to continually improve OVERLORD, and make future releases more suitable for your needs.

We welcome constructive criticism as much as praise, but please be as specific as you can with your answers. Whether you like a particular feature, or think that something could be improved, try to tell us why so we can carry these ideas into future versions.

Name (optional): Withheld

Company (optional): GlaxoSmithKline

Please tick the box if we can contact you to further discuss your feedback.
Editing

Think about your experiences with creating a new macro and editing existing macros, then answer the following questions.

Does the range of commands available allow you to create the functionality that you require? If not, what other commands would be useful?

Could do with some more standard programming features e.g. case statements. Multiple levels of if then statements are laborious, difficult to debug and you rapidly run out of program space.

Remove 50 line restriction. Remove re-use of variables. Provide some sort of parameter passing between macros. Allow arrays of variables. Allow arrays of

With regard to names, icons, and location on screen, do you find it easy to locate identify the commands you require? If not, why?

Not bad although there are some bugs with the drag and drop of icons. Maybe separate program commands from drivers

<table>
<thead>
<tr>
<th>Do you use drag and drop from the command list to add commands into the macro?</th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do you Left hand mouse click then menu to access commands to add into a macro?</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Do you Right hand mouse click then menu to access the commands to add into a macro?</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Are there any commands in particular you find useful?
APPENDIX A

With regard to command edit screen, including those for functions (e.g. Set Variable) and instruments, do you find them intuitive and easy to follow? If not, why?

How do you think the edit screens for creating Overlord macros could be improved for ease of use for the general user?

Do you have any other comments with regards to creating Overlord macros/procedures? What would make it simpler to setup an assay?

Runtime

The following questions are regarding how OVERLORD works when it is running a macro.

Do you find the functions that ask the users questions and show user messages easy and intuitive to use? If not, why? How could these be improved?

These are pretty good. The system forces the user to follow a set sequence which means they cannot forget a key step.
Currently when a command is active it is highlighted in yellow. Does any other information need to be displayed at this point? If so what?

Do you find the status of system easy to follow at runtime? If not, why?

Not really, see below. Some sort of realtime display that could be customised would be good. I have done this with a separate VB6 DLL in the past. Seeing the Overlord macro displayed can be confusing for the user and is only really useful during debugging.

Are there any changes to the on-screen display at runtime that you think would be useful?

Information box that does not stop program execution, ideally one that could monitor and display a variable while the program is running.

Ability to hide code display in background and display user controls only e.g. simply RUN, STOP and PAUSE.

Currently when there is an error with an instrument, the options Abort, Retry and Ignore are displayed with a description of the error. Do you find this provides enough functionality to recover from any error or is there more options you would like to see?

Opening up the relevant instrument manual control screen if available would be good so that you could adjust or reset the instrument.
APPENDIX A

Any further comments?

Please use the space below, and additional sheets if necessary, to tell us any other comments you have about ANY aspect of OVERLORD and what features you would like to see in the next version of OVERLORD

Finish the Help file.

Keep an updated list of bugs and fixes on the website

Fix existing bugs, especially fatal ones that crash you out of Overlord.

User forum or some way of exchanging tips and tricks with other Overlord users.

Access/Excel driver

Driver commands / Serial communications log for debugging

Stack/Hotel driver maintaining a record of where plates are.

OVERLORD Customer Feedback Questionnaire

Thank you for taking the time to fill in this questionnaire and give us your feedback on OVERLORD. As a valued user, we welcome your input in order to continually improve OVERLORD, and make future releases more suitable for your needs.
APPENDIX A

We welcome constructive criticism as much as praise, but please be as specific as you can with your answers. Whether you like a particular feature, or think that something could be improved, try to tell us why so we can carry these ideas into future versions.

Name (optional): Withheld

Company (optional): Rijk Zwaan

☐ Please tick the box if we can contact you to further discuss your feedback.

Editing

Think about your experiences with creating a new macro and editing existing macros, then answer the following questions.

Does the range of commands available allow you to create the functionality that you require? If not, what other commands would be useful?

The range of commands is enough to create a new macro.

With regard to names, icons, and location on screen, do you find it easy to locate identify the commands you require? If not, why?

This is also all right.

| Do you use drag and drop from the command list to add commands into the macro? | Yes | No |
| Do you Left hand mouse click then menu to access commands to add into a macro? | Yes | No |
| Do you Right hand mouse click then menu to access the commands to add into a macro? | Yes | No |
Are there any commands in particular you find useful?

Most of the things are useful.

With regard to command edit screen, including those for functions (e.g. Set Variable) and instruments, do you find them intuitive and easy to follow? If not, why?

Everything is easy for people with a little bit of computer experience.

How do you think the edit screens for creating Overlord macros could be improved for ease of use for the general user?

It could be improved a lot when you create different screens, more like windows. What I mean is that you can change several things in different screens. The problem now is that you’ve to open a file, before you can change it. When you open another file you lose your old file and you have to open the first file again. It should be easier when you create something like a pop up.

Do you have any other comments with regards to creating Overlord macros/procedures? What would make it simpler to setup an assay?

For comment look above.

Runtime

The following questions are regarding how OVERLORD works when it is running a macro
Do you find the functions that ask the users questions and show user messages easy and intuitive to use? If not, why? How could these be improved?

They're easy to use.

Currently when a command is active it is highlighted in yellow. Does any other information need to be displayed at this point? If so what?

Nothing

Do you find the status of system easy to follow at runtime? If not, why?

No, it isn't easy to follow. I think it's easier for the user when you see two different screens, one with the action on the moment and one screen for the bigger view from a run. The second screen is a screen where you can see your main run, the meaning is that you can see at which point of the process you're running.

Are there any changes to the on-screen display at runtime that you think would be useful?

For comment look above.
Currently when there is an error with an instrument, the options Abort, Retry and Ignore are displayed with a description of the error. Do you find this provides enough functionality to recover from any error or is there more options you would like to see?

For us it's more useful that we also see the log-file in the screen. We can see where the problem happened. Maybe it's an option that you can look in overlord for the mistake, when you fixed it, overlord run further.

Any further comments?

Please use the space below, and additional sheets if necessary, to tell us any other comments you have about ANY aspect of OVERLORD and what features you would like to see in the next version of OVERLORD

We will help you all the time with testing your new system at our robot system if it is necessary.
REFERENCES


REFERENCES

"Navigate the .NET Framework and Your Projects with "My" Namespace"


"MVP: Model-View Presenter The Taligent Programming Model for C++ and Java" –


"Generic Collections" - http://msdn2.microsoft.com/en-us/library/ms132398.aspx -
Accessed June 2006

Accessed September 2007


A. Davies – "Automated workflow optimisation and assay development strategies for High