View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Central University Of Technology Free State -...

LABVIEW APPLICATION WITH EMBEDDED LUA SCRIPTING FOR A LASER BASED MEASURING MACHINE

A. COETZEE AND T.I. VAN NIEKERK

ABSTRACT

This paper presents the work on the development of software for an industrial laser based measuring machine. The goal being not only for a working application, but also to optimise the development process and ease future maintenance of the software. LabVIEW with its graphical method of programming allows engineers to easily create large software applications to control industrial processes and machines. This software if not properly designed can lead to stability and maintenance problems. The experience gained from developing, maintaining and improving a LabVIEW application for a laser measuring machine, results in the integration of the Lua scripting language into LabVIEW. It is shown how the embedded Lua allows the LabVIEW software application for the machine to be structured for simpler development and maintenance.

Keywords: LabVIEW, Industrial software, Embedded, Lua, Scripting

1. INTRODUCTION

Computers are well suited to solve complex or repetitive problems. This is especially true in the industrial domain where computers have become the standard in control and automation. These industrial computers come in many shapes and sizes, from PLCs (Programmable Logic Controllers) to ARM® micro-controllers. Modern industrial control applications are often large and complex, which require a great deal of custom software to be written. This puts additional strain on the programming knowledge of the engineers responsible for the software. Many engineers are not computer programmers by profession, though they can find themselves in the awkward position of managing a large industrial software project. Many tools have been created to simplify programming for engineers. Such tools include programming environments like MathWorks' MATLAB and National Instrument's LabVIEW. The former uses a dynamic scripting language to simplify numerical computing, while LabVIEW provides an easy-to-learn graphical programming alternative. These tools go a long way to empower engineers to create ever more complex and robust software. Even these sophisticated tools cannot provide the complete solution to the engineer. Many lines of code still need to be written for the software application. As the size increases, so does the need to structure it correctly to assure correct operation. The structure of the software becomes even more important at later stages of the project where requirements might change or be added. A software framework can be used to add structure to an application.

There are frameworks, designed for complex robotic applications, which can be applied to industrial control projects. OpenRDK and Miro are two examples of such frameworks. The frameworks can execute on the Linux operating system, which can, unlike the Windows operating system be configured to provide deterministic response to external events.

These robotic frameworks are, however, not perfectly suited to industrial use and they require substantial programming knowledge to set up and use effectively. They provide structure to industrial applications, but lack the ease of use required by engineers from different disciplines. Another problem faced by engineers is making changes to software while it is executing. Currently the norm is to make software changes, recompile the software and then replace the existing software with the newly built version. This process can lead to lost production time. A small programming error may stop production for a prolonged period. This can largely be avoided by enabling the application to accept certain code changes while executing. An interpreted programming language embedded into the software application allows the engineer to modify machine behaviour without recompiling and replacing current software with a new version. An embedded interpreter also allows the software to be structured for simpler development and maintenance.

This paper shows how the graphical programming language, LabVIEW, can be used to create software to control complex industrial machinery. It also discusses the importance of structuring the LabVIEW code correctly to better accommodate future modifications and additions. Different approaches to structuring LabVIEW applications are shown. This results in the embedding of a scripting language into LabVIEW. The scripting language is chosen based on requirements of LabVIEW and industrial software development. An example application of the developed LabVIEW framework is also shown.

2. LabVIEW for Machine Automation

The programming language LabVIEW was chosen for this research effort because it was designed to be used by engineers and scientist. It offers a unique graphical form of programming that is easily understood by non-programmers. Coupling this with extensive support for hardware, makes LabVIEW a powerful tool for quickly creating large industrial and scientific software applications. The first version of LabVIEW was released in 1988 and was only available for the Macintosh computer[]. Recent versions of LabVIEW support multiple operating systems and computing platforms.

LabVIEW is a domain specific programming language, meaning that it was designed for specific use cases. Although originally intended for the scientific and measurement fields, LabVIEW has also evolved over the years to focus on automation. This makes LabVIEW a good programming language for engineers that need to focus on the problem domain rather than learning to become programming experts.

Some areas of application for LabVIEW include those shown in Figure 1, ranging from mobile robotics to industrial machine development. LabVIEW, having support for a large range of hardware in the form of drivers and example

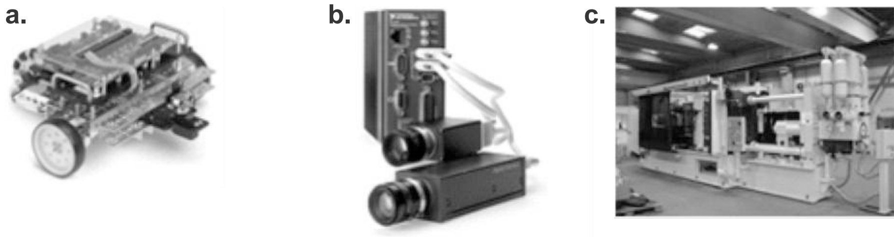

Figure 1: LabVIEW Applications: (a) mobile robotics; (b) machine vision; (c) industrial machines.

applications, allow LabVIEW developers to quickly prototype and try different solutions to engineering and scientific problems.

2.1 Advantages of Graphical Programming

LabVIEW provides a visual form of programming with the flow of data being represented by wires that connect different functions, as shown in Figure 2a. Simulink (Figure 2b) is also an example of a visual programming language and is displayed alongside the LabVIEW diagram for comparison.

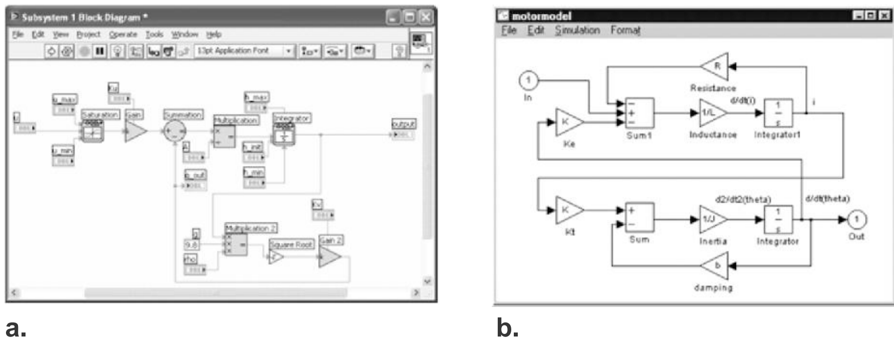

Figure 2: Visual Programming: (a) LabVIEW; (b) Simulink

Visual programming languages are becoming more important for programming tasks that are generally performed by non-programmers. This is especially true for industrial projects where engineers need to automate complicated machines. Diagrams and icons are often more easily understood by the end users than a conventional C program, as it allows them to participate in the creation of the program.

For engineers who are accustomed to modelling complex relationships, using graphical methods, programming in a visual form comes naturally. Other advantages of graphical programming are the ability to quickly see the connections between software components, and visualizing operations that can happen in parallel. In LabVIEW the sequence of operations are controlled by how the data flows between the functions.

2.2 Software Requirements and Flexibility

During the initial planning phase of a software application, the scope of the project needs to be determined. It is an important first step in the development process to establish all the requirements for a particular project. It aims to define all the functionality of the software application before any programming is done. Setting the structure and framework of the application can then be done. Figure 3 shows the identification of requirements as a complex process. The four steps of requirements gathering are connected with complex loops and must be continually performed throughout the project's life cycle. This is further complicated when requirements change during the development stage of the software. As the application nears completion, these changes can be costly in both time and resources to integrate. A software application that has been poorly designed, can suffer from changing requirements, with compromises being taken to implement or change functionality. These compromises can lead to software bugs being introduced into the application, mitigating the effort that has been put in during the planning stages.

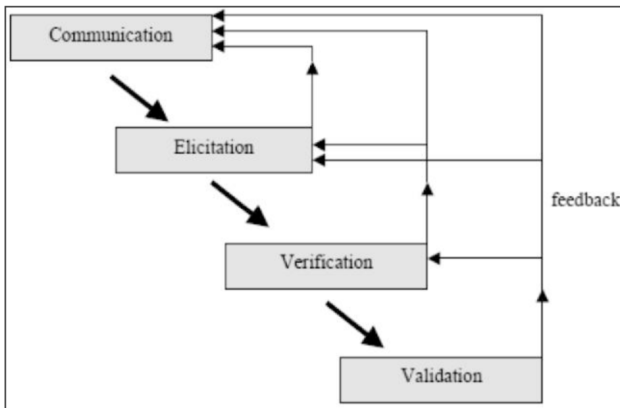

Figure 3: Software Requirements Identification Process

The same happens with software for engineering applications. The problem domain may include multiple contractors from different backgrounds working together. It often becomes very difficult to determine and pin down the exact requirements under time and budget constraints. That may lead to a more reactive development process during the commissioning phases of the project. The engineer may have to change the software application to cater for unexpected differences from the initially defined requirements.

2.3 Laser Measuring Machine

A previous project that contributed to the development of LabVIEW-based software that can better handle changing requirements is the LMM (Laser Measuring Machine). The LMM measures the circumference of a part to an accuracy of ±40μm over a 100mm span in an industrial environment. The machine rotates the part in front of a highly accurate distance laser sensor and reconstructs the data to form a two dimensional cross-section of the measured part. The design view and finished machine can be seen in Figure 4. More than one machine was built. With each machine the structure of the LabVIEW software application was improved. Two software design patterns were employed to provide structure and make the application more reconfigurable. These are the FSM (Finite-State Machine) and the producer-consumer design patterns.

Figure4: Laser Measuring Machine: (a) design; (b) finished machine.

2.3.1 Finite-State Machine

At first the software made use of a FSM design pattern. It is described as having several parts, consisting of states and rules for going from one state to another. A graphical example of a FSM is shown in Figure 5. It shows how software can be divided into sections (or states) that can be activated by certain conditions.

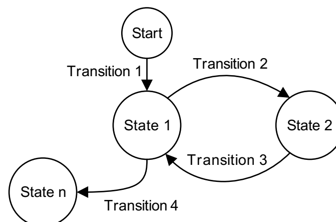

Figure 5: Finite-State Machine

This allows the software to group functionality and simplify the flow of control. The problem encountered with this particular design pattern was an increase in complexity with additions and changes to the software. After several modifications to the machine and its associated software, a decision was made to rewrite the software, using a design pattern that could better handle small modifications without compromising the rest of the application.

2.3.2 Producer-Consumer

In order for the software to respond better to changes, it was decided to switch to a producer-consumer pattern. A part of the software sends a list of instructions to a receiving function, which then executes them in order as shown in Figure 6. These instructions can also take the form of states, creating a queued-state-machine. Separating the logic that queues the states from the states themselves makes it easier to add or change states without affecting the others.

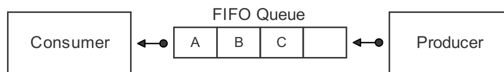

Figure 6: Producer-Consumer Design Pattern

By using the producer-consumer design pattern, the LabVIEW application became more flexible and robust. The method of changing the software on the machine remained replacing the old with a modified version. Adding these modifications to the software involved compiling the LabVIEW source code into an executable on a separate development computer.

3. SCRIPTING LABVIEW

The LMM was in continuous development. With every new machine built, new features were added and old ones improved. The software for the new machines also had to be more flexible. Measurement results had to be communicated to many different SCADA (Supervisory Control and Data Acquisition) systems. The producer-consumer pattern that the LabVIEW software was based on could not adequately handle this increased complexity. To simplify future development and debugging of the software the decision was made to structure the LabVIEW application around an embedded scripting language. Figure 7 illustrates the difference between programming a menu in LabVIEW and the scripting language Lua. The LabVIEW code will become more cumbersome and harder to maintain with additional menu entries. LabVIEW being a compiled language also makes it more difficult to alter the menu after the application has been deployed. The corresponding text based script in Figure 7b is better suited to future alterations and maintenance.

Although it can be changed after the application is deployed, scripted code does execute slower than compiled code such as LabVIEW.

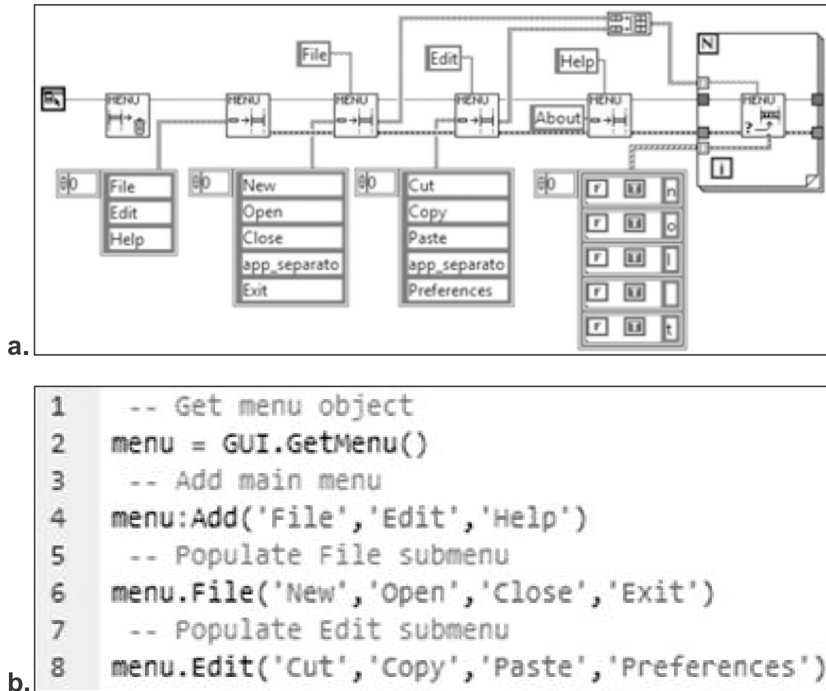

Figure7: Code Comparison: (a) LabVIEW (b) Lua

By writing the performance critical code in LabVIEW and combining it with a more flexible scripting language allows for rapid development and easier maintenance of the LMM software.

3.1 Script Language Evaluation

Instead of using LabVIEW natively to provide some kind of scripting support, an existing scripting language can be embedded into LabVIEW. This method overcomes the complexities of writing and debugging the scripting language itself. This leads to a new problem - how to interface between the graphical nature of LabVIEW and the more conventional text based programming of scripting languages. The embedded scripting language must also add as little complexity as possible and at the same time not affect reliability negatively. The ultimate decision to embed Lua into LabVIEW results from a study of several popular scripting languages that are freely available. The unique graphical syntax and parallel nature of LabVIEW imposes certain requirements and limitations that have to be addressed by the chosen scripting language. The requirements are as follows:

- **Well known:** In order to gain the full benefit of an embedded scripting language in an industrial environment, it must be easy for engineers and technicians to find information on how to program and use it.
- **Mature:** Software that runs on industrial machines must be reliable. It is assumed that a scripting language that has been in active development for several years has had time to address software bugs and instabilities.
- **Support multiple threads:** The scripting language must be callable from multiple threads simultaneously. This characteristic is necessary to support the inherent parallel nature of LabVIEW. The embedded scripting component must ensure safe parallel execution if it is to extend LabVIEW.
- **Cross platform:** LabVIEW programs can execute on a variety of computing platforms and operating systems. The embedded scripting language must be able to deploy to these environments as well.
- **Efficient:** Performance critical code can be created by using LabVIEW. An efficient scripting language embedded in LabVIEW would allow more features to be implemented by using it and ultimately make the application more flexible.
- **Small size:** The language size adds to the cross platform requirement of the scripting language. If it is to be used on embedded platforms with limited resources, the size that the scripting language itself adds to the application becomes important.
- **Easy to embed:** The scripting languages must provide enough features to make it easy to embed into LabVIEW.

The scripting languages that were evaluated include: JavaScript, Python, Ruby and Lua. Table 1 shows the requirements that each of these scripting languages support. JavaScript fulfils most of the requirements, but it is mostly used in web-browsers and it also has a large memory requirement. Python is a general purpose scripting language and is easily embedded into software applications. However, it has a large memory requirement and is not very efficient.

Table 1: Scripting Language Evaluation

Requirement	JavaScript	Python	Ruby	Lua
Well known	X	X	X	X
Mature	X	X	X	X
Support multiple threads	X	X		X
Cross platform	X	X	X	X
Efficient	X			X
Small size				X
Easy to embed		X		X

Ruby, like Python, is a general purpose scripting language. It is used more in web-server scripting applications and is not easily embedded into other software. Lua was designed to be embedded into other software applications. It also supports execution in multiple threads and has a small memory requirement. Lua is ANSI-C compliant and therefore supports all the platforms which LabVIEW can execute on.

3.1 LabVIEW Binding

LabVIEW, by being graphical in nature creates some unique challenges for embedding a scripting language. Two things are needed to integrate Lua into LabVIEW: the ability to call the embedded interpreter, and the ability to pass information and retrieve results. The Lua source code is written in the C programming language and exposes a stack-based interface for passing data. LabVIEW provides the 'Call Library Function Node' to interface to externally compiled C code. This provides a means of passing data between the two programming languages, but does not solve the problem of synchronising the function calls between LabVIEW and Lua. Two methods were explored to synchronise LabVIEW and Lua execution:

- Using events
- Using cooperative multi-threading

3.2 Using Events

The first method employed to embed Lua used the Windows and LabVIEW event API to synchronize access to shared memory. The shared memory functioned as the communication channel between the two programming languages. This method was implemented and abandoned in favour of the coroutine based multi-threading approached described in Section 3.4. Figure 8 shows the structure for an event-based embedding of Lua into LabVIEW.

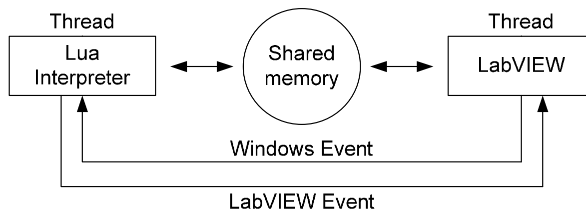

Figure8: Embedding Lua Using Events & Shared Memory

The process is as follows:

1. LabVIEW calls Lua and starts an event listener thread.
2. Lua puts arguments for a function call in shared memory. It notifies LabVIEW by using a LabVIEW event and then waits on a Windows event.

3. LabVIEW receives the event and retrieves arguments from the shared memory. It then continues with the function call, after which it places return values into the shared memory and by using a Windows event, notifies Lua to continue execution.
4. Lua retrieves return values from the shared memory and continues with execution.

Embedding by using events has several disadvantages, which are:

- Windows Events: This ties the application to the Microsoft Windows operating system, which limits the real-time deployment of the software.
- Complexity: The synchronization mechanisms used to communicate between the LabVIEW thread and the Lua thread are difficult to debug.
- Performance: Using events and two separate threads cause the processor to switch between them frequently. This results in high context switching overhead.

3.4 Using Cooperative Multi-Threading

The previous section showed an event-based binding between LabVIEW and Lua. This method was abandoned in favour of using a cooperative multi-threaded approach. The valuable experience gained from the event-based binding approach contributed to the final structure of the code. Many of the low-level data translation routines could be reused, and without implementing the event based approach, the concept of using co-routines would not have been realized. Co-routines were introduced in the early 1960's and represent one of the oldest proposals of a general control abstraction.

Co-routines are similar to functions, but include persistent data and the ability to suspend/resume execution. The flowchart for the co-routine based method of embedding is displayed in Figure 9. The ability to suspend execution in Lua and call-back into LabVIEW is shown by the branch in the flowchart. The call-back is completed by resuming the Lua interpreter, with the return values from LabVIEW on the stack. This is all done within a single thread which means the processor does not incur the context switching overhead. Another advantage of this approach is the ease of debugging as a direct result of the single threaded approach.

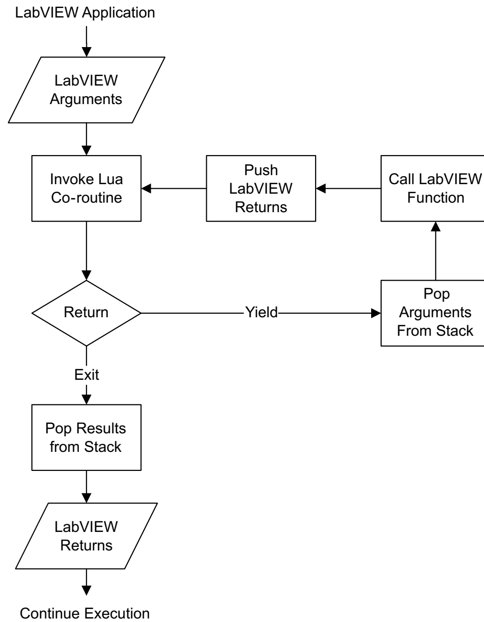

Figure9: Embedding Lua Using Co-routines

4. LASER MEASURING MACHINE

The previously discussed scripting facility for LabVIEW was implemented into a new version of the LMM software application. The improved LMM is shown in Figure 10, where the machine and controller have been separated allowing it to be integrated into an automated production line. The two parts are connected by an umbilical cord that contains all power and control signals. An Ethernet cable connects the control panel with the SCADA system.

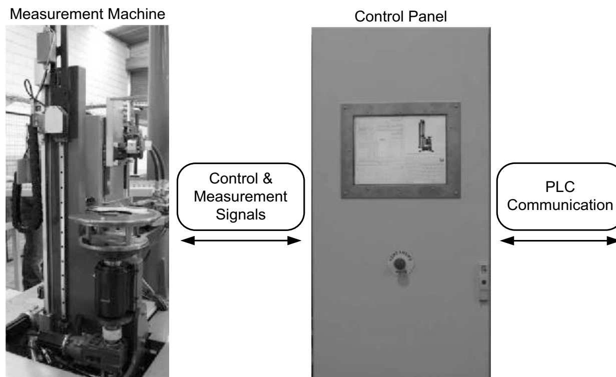

Figure 10: Measurement System

4.1 Software Framework

The control panel in Figure 10 contains the computer that controls the machine's operations. The structure of the software is shown in Figure 11. It displays the various LabVIEW functions that make the machine operational. Central to the software structure is the embedded Lua interpreter. The interpreter executes scripts that synchronise the various LabVIEW functions.

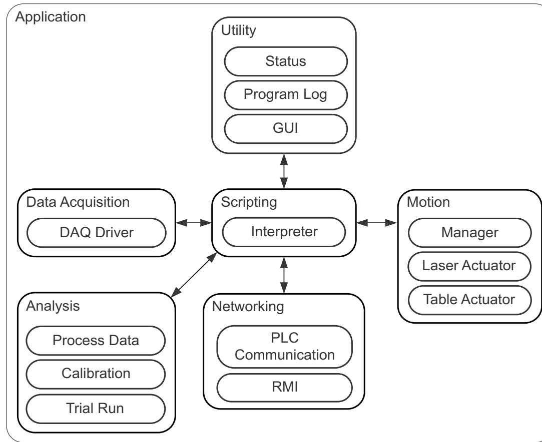

Figure 11: Software Layout

The embedded Lua interpreter simplifies the creation and debugging of the LabVIEW software application. The individual LabVIEW functions can be developed and tested separately and then combined into the final application.

4.2 Measurement & Analysis

The primary purpose of the software running on the LMM machine is to take the analogue signal from the laser position sensor and convert it into usable information. This involves setting up the data acquisition hardware to capture the input signal from the laser correctly and then to process the values. To simplify the design of the software, the data acquisition and analysis functions were exposed to the Lua interpreter.

Figure 12 shows graphically how the Lua interpreter executes the code.

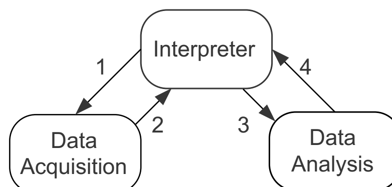

Figure 12: LabVIEW Scripting Example

This piece of Lua code is executed within the LMM LabVIEW software and can be modified after the application is compiled and deployed to the controller. The sequence of operations is:

1. Call the data acquisition function to request data.
2. The raw data is then returned to the interpreter.
3. After the data has been collected the interpreter calls the data analysis function to process it.
4. When the data has been processed the analysis function returns with the results.

The corresponding Lua code as executed on the LMM:

```
1  -- move motion axes
2  TableAxis.VelocityMove(15)
3  LaserAxis.Move(30)
4  -- start the measurement
5  Data = DAQ.AcquireData()
6  -- process the raw data
7  Analysis.Process(Data)
```

The previous script can then be modified by the engineer whilst the machine is in operation to:

```
1  -- move motion axes
2  TableAxis.VelocityMove(15)
3  LaserAxis.Move(30)
4  -- start the measurement
5  Data = DAQ.AcquireData()
6  -- process the raw data
7  Analysis.Process(Data)
8  -- added code to send results
9  Network.Send(Analysis.GetResults())
```

Lines 8 and 9 are added, which program the machine to send the results using the LabVIEW network component to a remote location. By programming the actions of the machine using Lua scripts, the engineer can modify and optimise the machine without recompiling the main LabVIEW application. This translates into shorter cycle-times and reduces costly down-time.

5. CONCLUSION

Engineers are often put in the position of writing or maintaining software. These software applications can be large and complex, requiring in-depth computer programming knowledge to create and edit correctly. In this study, the continued improvement of the software for the laser measuring machine resulted in the embedding of a scripting language into LabVIEW. Well known scripting languages were evaluated with Lua being chosen.

First an event-based method was developed. This was replaced by an approach using cooperative multi-threading. Lua coroutines provide a simplified method for embedding Lua into LabVIEW, whilst also making the resulting solution as portable as LabVIEW. The Lua scripting facility allows LabVIEW software to better respond to changing requirements. It also makes structuring and debugging the software simpler. This was demonstrated in a new version of the software for a LMM used in an automated production line. The resulting LabVIEW application is more modular, easier to maintain and can accept small modifications online with minimal impact on production. It is therefore concluded that by embedding the Lua scripting language into LabVIEW, that the development and maintenance of industrial LabVIEW software is simplified.

5. REFERENCES

J. Vernon. (2011, October) Programmable Logic Control. [Online]. http://www.control-systems-principles.co.uk/ whitepapers/programmable-logic-control.pdf

Jon Conway and Steve Watts, Software Engineering with LabVIEW.: Pearson Education, 2003.

Roy Miller, Managing Software for Growth: Without Fear, Control, and the Manufacturing Mindset.: Addison-Wesley Longman Publishing Co., Inc., 2003.

Daniele Calisi, Andrea Censi, Luca Iocchi, and Daniele Nardi, "OpenRDK: a framework for rapid and concurrent software prototyping," in Proceedings of the International Workshop on System and Concurrent Engineering for Space Applications (SECESA), Prague, 2008.

H Utz, S Sablatnog, S Enderle, and G Kraetzschmar, "Miro - middleware for mobile robot applications," Robotics and Automation, IEEE Transactions on, vol. 18, pp. 493--497, December 2002.

Wang Zi-niu, Li Song, and Wang Yan, "Researching of Real-Time Versions and Testing Its Performance of CNC System Based on RT-Linux," in Proceedings of the 2009 International Conference on Networking and Digital Society - Volume 02, Washington, DC, USA, 2009, pp. 174--177.

Brian Powel. (2007, June) LabVIEW Performance, The Early Years. [Online]. http://openmeas.blogspot.com/ 2007/06/labview-performance-early-years.html

Lorrie Cranor and Ajay Apte, "Programs worth one thousand words: visual languages bring programming to the masses," Crossroads, vol. 1, no. 2, pp. 16--18, December 1994.

Paul Biggar, Edsko de Vries, and David Gregg, "A practical solution for scripting language compilers," in SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing, New York, NY, USA, 2009, pp. 1916--1923.

Hemant Jain, Padmal Vitharana, and Fatemah M Zahedi, "An assessment model for requirements identification in component-based software development," SIGMIS Database, vol. 34, no. 4, pp. 48--63, November 2003.
Michael Sipser, Introduction to the Theory of Computation.: Course Technology, 2005.

David Flanagan, JavaScript: The Definitive Guide.: OReilly Media, Inc., 2006.
Hans P Langtangen, A Primer on Scientific Programming with Python.: Springer Publishing Company, Incorporated, 2009.

Dave Thomas, Chad Fowler, and Andy Hunt, Programming Ruby 1.9: The Pragmatic Programmers Guide.: Pragmatic Bookshelf, 2009.

Roberto Ierusalimschy, Luiz H de Figueiredo, and Waldemar Celes, "Passing a Language through the Eye of a Needle," Queue, vol. 9, no. 5, pp. 20:20--20:29, May 2011.

Ana L Moura and Roberto Ierusalimschy, "Revisiting coroutines," ACM Trans. Program. Lang. Syst., vol. 31, no. 2, pp. 6:1--6:31, February 2009.