Development of a custom software regression-testing tool: Ensuring a robust system for the management of electricity energy market delivery.

Dr. Tim D. Hunt
The Waikato Institute of Technology,
Hamilton,
New Zealand.
tim.hunt@wintec.ac.nz

Peter Ensor
Realtime Information Limited,
Hamilton, New Zealand.
peter.ensor@realtime.co.nz

The development of a software tool for performing automated regression testing of a critical information and control system is described. The tool was written using Borland Delphi and made extensive use of predefined classes. The software uses ADO to input data from an MSExcel spreadsheet, links to a DLL (the application under test) to perform the tests and outputs results to text files. The software was designed using Rapid Application Development (RAD) technologies between client and developer, making use of frequent demonstrations of implemented functions to verify design progress. The client has used the software to successfully confirm the stability of the product, to scope changes and ensure the robustness of their deliverable system.

1. INTRODUCTION

The importance of adequately testing software has been recognised since software was first written, yet the consumer software market still produces products with a large number of defects resulting in a poor end user experience. When software is used in critical situations, the goal is to achieve software with near zero defects. Software regression testing is one tool that the software engineer has to facilitate achieving this goal.

There are a large number of software testing tools available (for example see the list at: http://www.aptest.com/resources.html) however finding a tool that matches specific testing requirements can be a time consuming and costly process making the implementation of an off the self useful testing system prohibitive to small and medium sized software development companies. Further, when the system under test has no user interface and runs on a single platform it can be more cost effective to develop a custom testing environment. This paper describes the development of a custom regression testing application (the RilTester) for a Hamilton company, Realtime Information Limited (http://www.realtime.co.nz) referred to in this paper as 'the client'. This company provides customized software to the electrical generation and transmission industry.

1.1 Software Testing Theory

Testing practice is often described as either 'Functional' where the software is seen as a 'black box' or as 'Structural' also known as white box testing. Black box testing takes the approach that only the inputs and outputs of the code are tested whereas white box testing uses knowledge of the internal code to design tests that make sure every code pathway is tested. In practice both approaches should be used as they have complementary qualities. A good review of traditional software testing practice is given by Jorgensen (1995).

The 'traditional' testing approach often fails as a result of external commercial pressures reducing the time allocated for testing when the design and coding stages take longer than expected. The advent of Xtreme programming, XP, see an introduction by Beck (1999) has added a new impetus to software testing, with testing taking a central role in the software development process. The testing regime integrated into the Xtreme programming methodology brings testing to the front of the coding process, as apposed to the traditional approach where tests are devised and performed after the software has been written.

Continuous design, Shore (2004), also known as evolutionary or emergent design is a term used to describe a method of software development that doesn't try to guess future functional requirements of a piece of software, but instead implements each

feature as it arises and expects to change the design as new features are requested of the software. It is obvious that when this approach is applied, it is important to retest functions that previously passed tests – this testing approach is known as 'Regression Testing'.

Regression Testing involves the testing of software after a change has been made, to ensure that this change has not had unforeseen implications on previously working features. For example, suppose a program performs the functions: add(a,b) and multiply(a,b); if these functions have been implemented and tested before a third function subtract(a,b) is implemented, good practice requires to not only confirm that the new function subtract(a,b) works but that the other two functions continue to work.

Software that contains a large number of functions that need to be tested under a large number of conditions can quickly require many tests to ensure that all functions continue to perform as expected. Manual testing can be done, but due to the time consuming and boring nature of manual testing, it is hard to ensure that all required tests are performed. An automatic tester can solve this problem by easily rerunning tests on functions that previously passed tests.

When a new fault is detected in the application under test, the specific tests are written to identify this fault and the tests added to the regression test set prior to the fault being fixed. These tests are then used by the software developers to verify that the fault has been fixed.

1.2 Background to the Application Under Test

The electricity network on the island of Tasmania, off the south east coast of Australia is operated independently from mainland Australia, and is responsible for coordinating its own electrical energy needs. With a view to fostering increased competition in Tasmania by connecting into the National Electricity Market (NEM), it was proposed around the mid 1990's to install an undersea High Voltage Direct Current (HVDC) electricity link with the mainland. While the State had sufficient generating plant for coping with the peak demand, it was short on long-term energy storage.

The connection known as Basslink, which is to be commissioned in late 2005, will enable Tasmania to join the national market, which operates on the mainland of Australia. The State will be able to export electricity into Victoria during high demand periods and take advantage of the high prices at those times. Conversely, it will be able to import electricity during the low demand periods when prices are much lower and therefore more favorable.

One implication of joining the NEM is that the Automatic Generation Control (AGC) that continuously determines the amount of generation required in Tasmania, allocates it to the appropriate generation units and controls the units to their targets, needs to be interconnected with the National Electricity Market Management Company (NEMMCO) control systems. These centralized AGC control systems are located in Sydney and Brisbane and send targets to generation units Australia wide so that they meet the power system demand for the market operation, within specified system control parameters.

The application being developed interfaced between the Australian multi-State AGC system and the principal Tasmanian State AGC. It was named the Pre/Post Processing (PPP) application due to its relationship with the Tasmanian operator's AGC.

Due to the continuous operation of the application and the high visibility and economic implications of any failures, this has led to the requirement of a robust system for the management of electricity generation reserve.

The PPP application was initially coded and tested using Microsoft Visual C++ within a Microsoft(R) Windows environment and then ported to UNIX operating on Alpha servers.

1.3 Testing Environment

The choice of the windows testing environment was based on considerations such as: nature of input data, requirement for a GUI, in-house knowledge and cost. In particular, the use of standard office applications provided efficient test data set construction, maintenance and analysis.

As the majority of applications written by the client are in the windows environment, a general purpose automated regression testing tool would have additional benefits to the company outside of this project.

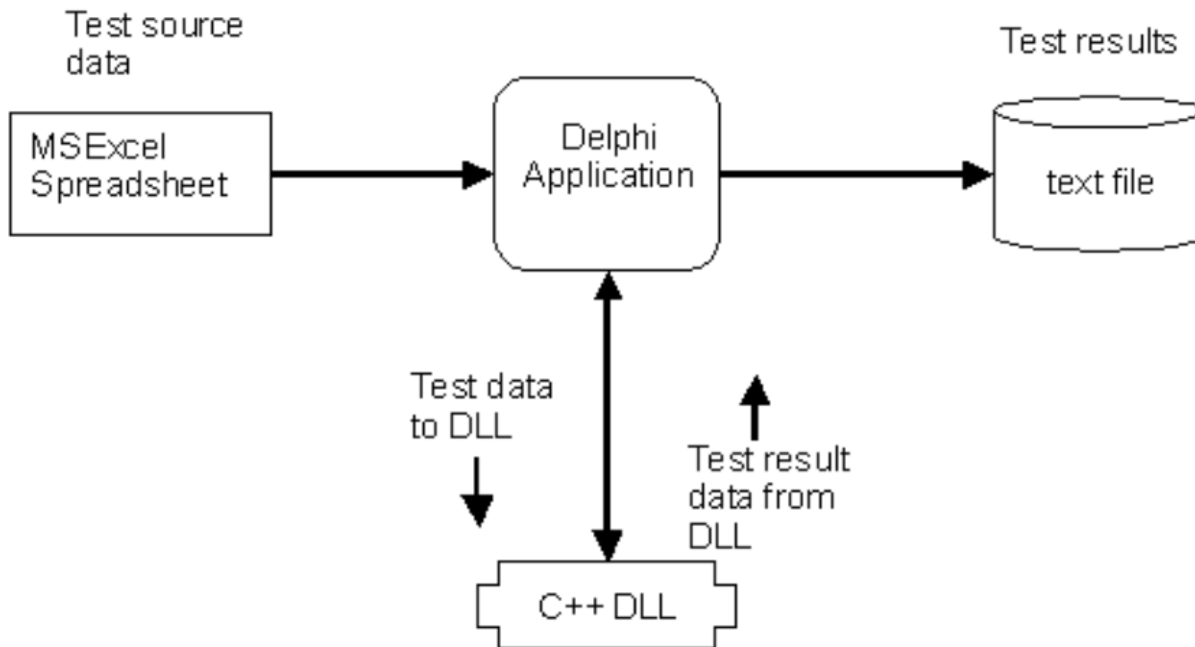

Figure 1. Data flow for test application.

A number of development environments for the regression testing tool were considered including: Visual C++, Visual Basic, C# and Delphi. Delphi was eventually chosen for its GUI strength and match with the skills available at the client who would eventually perform ongoing maintenance of the software.

The PPP software was presented as a Dynamic Link Library (DLL) and input data was provided in an MSExcel spreadsheet. Test results are displayed in the GUI as well as being saved to a text file. Figure 1 shows the flow of input and output data.

While there was some customization of the application to make it available as a dll, the use of a dll was a deliberate decision to provide a standard interface to the regression testing tool. It allowed multiple versions of the dll to be maintained in a compiled form in a revision control system (RCS), along with the source code so that faults that were identified late in the testing cycle could be verified against the earlier version to determine when the fault was introduced to the production code.

2. PROBLEM DOMAIN

2.1 Analysis

Analysis of the problem began with informal discussions between the software developer and the client. Due to the non-commercial nature of the work and the relatively small application size, a formal specification was not developed. Instead, analysis and development progressed with small incremental and iterative steps with frequent demonstrations of functionality to the client – RAD methodologies.

The initial testing approach was to use 'Black Box' testing (Jorgensen, P.C) but it was realized that due to the large number of input variables, the quantity of input data required to test all possible variations, meant that strict black box testing would be impractical. A mix of white and black box testing was employed with white box testing used mainly at the function level and black box testing employed at the system level.

2.2 Details of Application Under Test

The PPP application is written in ANSI C and employs a single large C structure for the maintenance of its internal variables. While the application is event driven, the large majority of the application consists of a single threaded application that is executed every second. It is primarily this thread that was tested using this regression testing application.

In interfacing the PPP application with the RILTester, a set of standard functions were designed that would be 'exported' to the RILTester and would be generic to any application under test.

These 'C' definitions for each of these functions are:

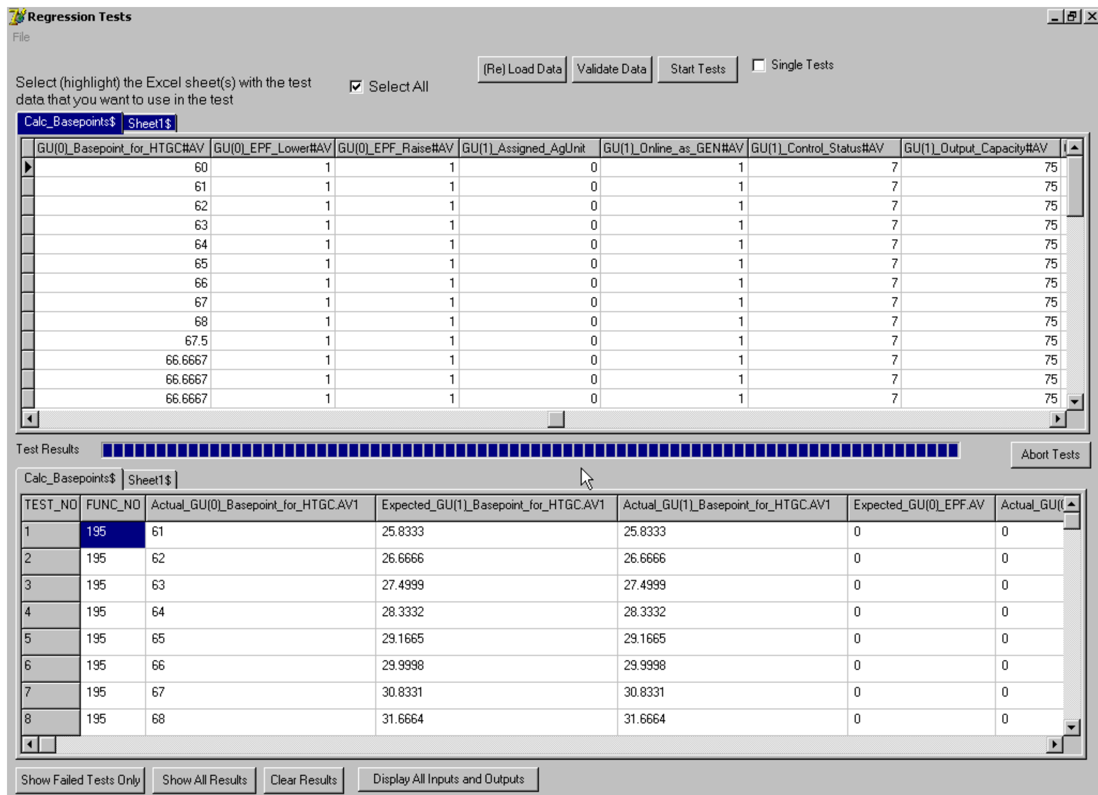

Figure 2. The GUI interface of the RILTester. The top half of the screen shows the input data, and the bottom half shows the test results.

```
__declspec(dllexport) int __stdcall
Test_Initialise();

__declspec(dllexport) int __stdcall
Test_Destroy();

__declspec(dllexport) int __stdcall
Test_Clean();

__declspec(dllexport) float __stdcall
Test_GetPtr(char *LogicName);

__declspec(dllexport) int __stdcall
Test_PutValue(float Index, char *Value);

__declspec(dllexport) int __stdcall
Test_GetValue(float Index, char *Value,
int MaxLength);

__declspec(dllexport) int __stdcall
Test_Run(int TestNumber, double
*ReturnedValue);
```

The functionality of each of these functions is briefly outlined in table 1. The table describes the use of a logic model name. This is a human readable name that is used to identify a particular element of the data structure such as Generating_Unit[1].MW.

It is proposed that these functions would be customized for each new application being placed under test. For example, the logic model variable names would need to be resolved into different data structures for each application.

3. SOFTWARE DEVELOPMENT: THE RILTESTER

3.1 User Interface

The GUI of the RILTester (Figure 2) is split into two main sections. The top section displays the input data that has been imported from the MSExcel spreadsheet. The bottom section shows the results of the testing that are also automatically saved to a comma delimited text file. The spreadsheet can have data on multiple sheets, and Figure 2 shows how the data from each sheet is placed on a different 'tab'. Only data from the selected tabs is used for testing. To reduce the chance of 'bad' data causing a problem during testing, the input data can be validated before the testing starts. The user can choose to display either all the results or just those for tests that have failed.

3.2 Design

The continuous design philosophy (see section 1.1) was employed in that the initial functions speci-

Table 1. List of the functions provided by the PPP DLL for testing purposes.

Function	Description
Test_Initialise()	Initialises the application under test. Sets up memory structures and opens files if required. This function is called once by RILTester during the initialization sequence.
Test_Destroy()	Releases memory and cleans up ready for termination of the dll. This function is called once by RILTester during the termination of the RILTester application.
Test_Clean();	This function reinitializes the memory structure to a known state. This function can be called between the steps in a regression test to place the application under test into a pre-defined known initial state.
Test_GetPtr(char *LogicName);	In order to speed up the updating and retrieving of the individual elements within the data structure. The exact implementation will vary between the different applications under test. In the PPP application, the floating-point value that is returned is a memory offset pointer from the base of the data structure. The integer portion of the float is the offset in bytes while the fractional part represents how the data is formatted in memory. For example, the following fractional parts are used to represent the associated storage format types: 0.1 = float 0.2 = integer 0.3 = time_t 0.4 = DQ (special data type for the PPP application) This function is called once for each element of the data structure that is to be updated or retrieved by RILTester (as defined in the spreadsheet). This function also resolves any array indexes such as Generator_unit[23].MW as part of the determination of the index value.
Test_PutValue(float Index, char *Value);	The values are updated into the application under test's data structure using the index determined above and the value passed as a string. The decision to use a string was adopted, as this would provide the most flexibility as to the data that could be passed.
Test_GetValue(float Index, char *Value, int MaxLength);	This function is similar to the Test_Put_Value() except that it retrieved the value.
Test_Run(int TestNumber, double *ReturnedValue);	Each function to be tested is allocated a test number. In the PPP application, every function was allocated at least one test number. That allowed the low level functions to be tested prior to the testing of the higher-level functions. Some functions were allocated multiple test numbers. This occurred when a constant was passed in the parameter list such as Summate (a, PP_RAISE_ONLY, b, c). In this case one test passed the PP_RAISE_ONLY while another test used the PP_LOWER_ONLY definition. In many cases the tests used hard coded array indexes in the function parameter lists as the test Summate(a[0], b[0], c[0]) would not operate differently than Summate(a[1], b[1], c[1]). A ReturnedValue parameter was passed back to RILTester for those functions that returned a result such as c = abs(b).

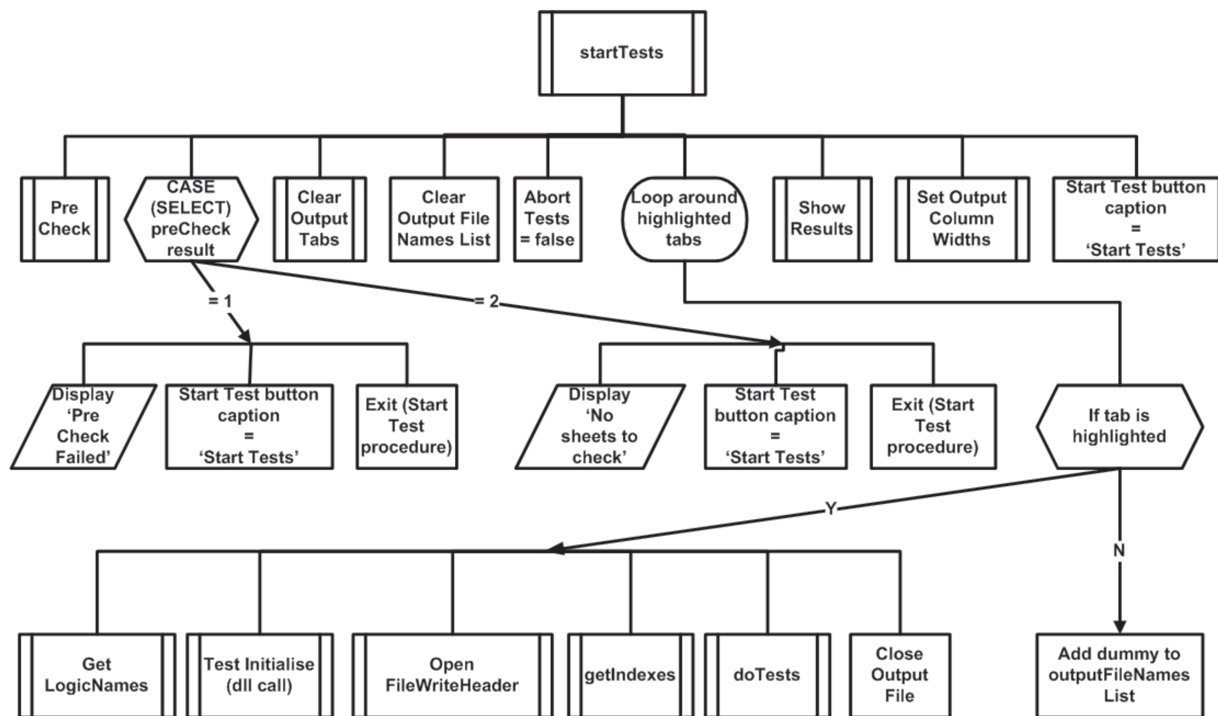

Figure 3. Structure diagram of the main process called when the 'Start Tests' button is pressed.

fied by the client were developed, and then each additional function was implemented when requested by the client. There was a critical moment in the evolution of the software when the initial simple design was unable to cope with the addition of new functions. At this point a major rework of the code as well as design documentation was performed before it was possible to continue to add additional functionality.

Although the application is likely to receive future developments, the current design and implementation makes extensive use of Delphi Classes, and the high level design follows a procedural flow. Figure 4 shows the structured diagram for the code that runs when the 'Start Tests' button is pressed. The top down design used the divide and conquer approach, where major tasks were written as separate functions. As well as allowing for code reuse, the practice of using multiple functions facilitated the testing of the RILTester with each function returning a value to indicate if the function had completed as expected. The use of 'stubs' was used during development to allow the overall design to be tested before every function was completed.

3.3. Major Implementation Decisions

3.3.1 Extracting Data from the MSExcel Spreadsheet

The client required all input data to be supplied in an MSExcel file, using multiple sheets. The first row of each sheet contained the 'logic model' name and subsequent rows the actual data. As well as the input data, each row contained the expected output data. The actual input variables and required output data on each sheet was not known at 'design time' and represented only a small sample of the possible variables. The number and name of the input sheets was also an unknown at design time. These unknowns presented a design issue for displaying data to the user in the GUI.

The Delphi ActiveX Data Object (ADO) connection (TADOConnection) was used to connect to the MSExcel spreadsheet. This allowed the spreadsheet to be treated as a database, with each sheet treated as a separate 'table' and the first row of each sheet giving the 'field' names. The input data could then be accessed using standard SQL. TADOConnection provides the GetTableNames method, which provides the number and names of the input sheets i.e. tables. The Delphi TPageControl class is used to provide a tabbed sheet on the Del-

phi form and a tab is added for each table. As each tab is created, an instance of the Delphi TADOQuery class and an instance of the TDBGrid class are placed on each new tab, so that each set of data can be accessed and displayed separately. Lists are used to store the names of the instances so that they can be referred to by their place in the list at design time.

A special character was reserved for use in a spreadsheet cell (e.g. '*') that was interpreted by the RILTester to skip the updating of a variable in the application under test. This was required due to a number of variables needing to be initialized (and therefore be identified in the input section of the spreadsheet) but should not be updated during subsequent tests. This is particularly significant for applications that have a dynamic response that requires reviewing. It has the additional benefit that by not initializing all input data for subsequent tests, array overruns could be detected where they corrupt significant data.

3.3.2 Connecting to the DLL

The DLL under test presented seven functions that the RILTester needed to access. This is achieved in Delphi by declaring the functions as external and giving the DLL name (and in this case the 'mangled' name of the function). The DLL also used pointers to reference variables and structures that had to be replicated in the RILTester.

3.3.3 Displaying and Saving Results

As for the input data, the quantity of output data was unknown at design time and the method used to display this data was similar to the display of input data. Once the number of input sheets was determined, the correct number of tabs on the output control (TPageControl) was created. All output data was written to a text file with comma separate values (CSV) format. An instance of the Delphi class TStringList was used to both read the CSV file using the LoadFromFile method and store the data in memory. The contents of the TStringList were then displayed on the form using instances of the TStringGrid class.

The provision of a CSV file allowed the dynamic response of the application under test to be viewed using graphing tools.

3.3.4 Performance Issues

Due to the possibility of a large number of tests, ways of improving the time take to perform the tests were investigated.

3.3.5 Writing to Disk

There exists a conflict between reducing the time for testing and providing a safe guard against loosing all data if the system fails. The initial design wrote test results to the text file after each row of input data. This meant that if a testing session was left to run unattended, say over night, a system crash would not mean the loss of results already obtained. However disk access is a known potential bottleneck in processing through put and so a compromise, between these conflicting requirements was implemented by writing to disk after a set number of tests, initially set to 1000.

3.3.6 Look Up Values

The initial implementation required the RilTester to query the DLL for a pointer before each new value was sent to the DLL i.e. for each row of test data the pointers were requested from the DLL for each value. Due to approximately five hundred logic model variable names being used, the lookup time to resolve this name into a memory point was significant. To remove this step, when the pointers were first obtained from the DLL, they were stored in the RilTester for subsequent use. This enabled highly repetitive calls to be made to the PutValue() and GetValue() functions with minimal overhead.

4. RESULTS

The RILTester was used in practice in two modes:

The first was during the development cycle where data was introduced to a particular function as that function was being developed. This permitted functions to be written in isolation of the remainder of the application, particularly before the application under test's interface had been constructed. The developer would write specific tests to test various scenarios based on the knowledge of the code and the part of the function under test. Once the micro-functionality was verified, the test cases were saved to another tab within the spreadsheet to be used subsequently when the function had been completed.

This method verified that as further micro-functionality was developed within the function, the func-

tionality that had already been proven was not corrupted.

The second method that was employed was the back-box testing where the System Analyst would construct test cases based on the Business Functional Requirements and the System Software Design documentation.

4.1 Benefits to Students

This work was integrated with the delivery of the course PR614 Programming (Interactive), part of the Diploma in Information and Communications Technology (Level 5) delivered at Wintec. Students were set a major assignment based on the requirements for this application. It was explained to the students that the tutor had taken on the task of developing this software and that they were going to assist by developing their version in parallel. The course followed a Problem Based Learning approach and details of the outcomes and lessons learnt will be reported elsewhere.

A future benefit of the application would be to have the students design their application as a dll that could be tested by the examiner with a number of test cases to verify the robustness and accuracy of the students' coding.

5. SUMMARY AND CONCLUSIONS

The development of a custom testing application to enable regression testing of a critical electricity supply infrastructure system has been described. Borland Delphi was chosen for the development environment as it provided the ability to easily integrate the requirements of GUI design, ability to connect to external data sources and software to test, as well as being one of the skill sets of those responsible for future on going maintenance. The continuous design philosophy was employed resulting in a top down procedural approach with extensive use of predefined classes that proved to be a successful and rapid solution to the client requirements. The software design was documented using structured diagrams that allowed for rapid implementation of additional requirements as they became necessary.

The client has used the software to run extensive regression testing of mission critical software for the management of electrical generation reserve. The RILTester interfaces to the application under test using a set of generic functions, so allowing the RILTester to be used to perform regression testing on future software applications created by the client.

ACKNOWLEDGMENTS

The authors would like to thank Hydro Tasmania and Realtime Information Limited for permission to publish this work.

REFERENCES

Beck, K. (1999) "Extreme Programming Explained". Addison-Wesley Pub Co, ISBN 0201616416.

Jorgensen, P.C (1995) "Software testing: A Craftsman's Approach". CRC Press, ISBN 084937345X.

Shore, J (2004) "Continuous Design". IEEE Software, 21(1):20-23.