Development of an automated testing tool for identifying discrepancies between model implementations

[1]Freebairn, A., [1]J. Rahman, [1]S. Seaton, [1]J-M. Perraud, [1]P. Hairsine and [1]H. Hotham

[1]CSIRO Land and Water, E-Mail: Andrew.Freebairn@csiro.au

Keywords: *Model testing; Porting Models.*

EXTENDED ABSTRACT

Software testing is an important and yet time-consuming part of any model development effort. When an existing model is ported to a new platform, developers need to rigorously test the new version for compliance with the existing model.

This paper describes a testing tool (testbench) used to streamline the porting of existing models to a new code base. The tool automates the comparison of the newly ported model against the original, rapidly highlighting errors and giving the developer confidence that the new source code is working as expected. In addition to its utility in porting, the test bench can also support regression testing when further developing existing models.

The test bench takes as input a 'trace file', which contains time series data for every input, parameter, state variable and output of the existing, legacy model. The tool feeds the inputs and parameters from the trace file into the new version of the model and compares values predicted for each state variable and output with those produced from the existing model. If any of the state variables or outputs differ, it is highlighted in a graph. This allows the model developer to quickly see where variables differ and to identify 'patterns' and therefore sources of error in the newly ported model.

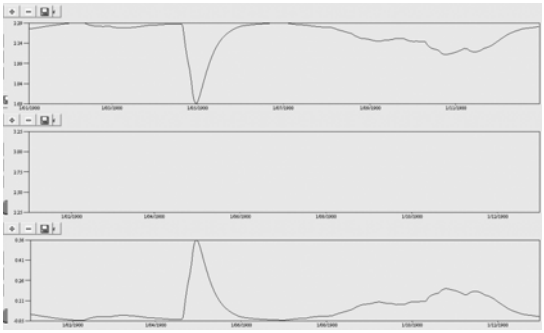

Figure 1: Testbench interface illustrating the expected, modelled and their differences after running.

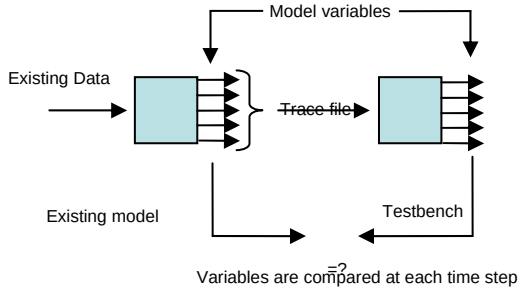

Figure 2: An overview of the porting test system.

The test bench is written using the reflection tools in The Invisible Modelling Environment framework TIME, allowing generalisation to other models. This reduces the time and effort required by developers to utilise the testbench for their specific TIME model.

The development of the test bench is described in the context of porting a water balance model from a C++ Borland framework to a .NET framework.

1. INTRODUCTION

All modellers of systems are subject to periodic changes and upgrades of their modelling environment. Furthermore, the models themselves undergo expansions in functionality, commonly with a requirement to maintain backward compatibility with previous versions. In this paper we describe a tool that automates compliance testing when porting a model into a new framework and discuss key steps to ensure the original requirements are maintained and further improvements of the model are achieved.

Software, including models, are ported between environments to take advantage of new features, be they at a low level, such as dynamic memory allocation absent from older versions of Fortran, or high level, such as the dynamic visualisation capabilities of model development environments such as TIME (Rahman *et al.* 2004).

Moving from one modelling environment to another can result in the following problems.

1. Loss of analytical and end user functionality familiar to existing users
2. Loss of runtime performance
3. Introduced error

A mature approach to testing is essential during software porting to mitigate against these three risks. This paper briefly discusses techniques of end user testing, performance testing and regressing testing for errors. The tool presented here focuses on ensuring that the algorithms of the model are preserved in the new implementation and that errors have not been introduced.

2. SOFTWARE TESTING

A thorough quality assurance program will include:

- testing with users, to ensure that the software meets their needs and is easy to learn and use,
- testing of performance, in terms of a software system's demands on CPU, memory and disk resources under various usage scenarios, and
- testing for correctness, against some objective measure such as a formal specification, or in this case, 'regression' testing against results from another implementation.

Some aspects of testing can be automated, allowing tests to be undertaken repeatedly to track a systems progression towards or away from some goal.

The tool described here presents an automated way to undertake correctness testing with respect to results from a 'trusted' source.

Before describing the testing framework for porting, brief descriptions of general software testing practices are given. This will illustrate where the testing framework fits within existing practices.

2.1. End user testing

Usability testing requires end user to perform a sequence of tasks that have been previously outlined in an application's specification. The usability of the software is measured by users ability to learn and perform these tasks, the speed at which users are able to complete tasks and the ability of tasks to be completed via differing paths (flexibility, shortcuts for experienced users).

2.2. Performance testing

As part of the non-functional specification of software, performance testing endeavours to determine how the software performs tasks in differing environments (hardware, operating systems, and networks).

2.3. Correctness Testing

Software testing is a Quality Assurance (QA) process that endeavours to verify that application requirements are met and is a means to verify correctness.

The following is an example of how a requirement is tested using an automated test harness. The requirement is that the calculator can add two integers. The test class (CalculatorTest) is part of a test suite that tests each function of the calculator. The key line is the assertion testing (in bold) that the addition of one and one equals two.

```
namespace math
{
        public class Calculator
        {
                private int _result;

                public void Add(int value1, int value2)
                {
                        Result = value1 + value2;
                }

                public int Result
                {
                        get{ return _result;}
                        set{ _result = value;}
                }
        }
}

The test case for the calculator is:
namespace math
{
        using NUnit.Framework;

        [TestFixture]
        public class CalculatorTest
        {
                [Test]
                public void AddIntegers()
                {
                        Calculator cal = new Calculator();

                        cal.Add( 1, 1 );

                        Assert.AreEqual( 2, cal.Result );
                }
        }

}
```

Figure 3: Example of how a specific requirement is unit tested using a test harness.

This illustrates the components of an automated unit test:

1. A component (or unit) to be tested. In this case, the Add method of the Calculator

2. An action to be performed on the unit. In this case, we invoke the Add method, with the line: `cal. Add( 1, 1 )`

3. A check of the outcome of the action. The "Assert" statement which in this case checks that the result equals the expected value of two.

Tests are used to investigate what happens in normal and abnormal conditions. Testing the range from normal to abnormal conditions assesses robustness of the system. An addition to the above example which tests for robustness is to test that an exception is thrown when one or both of the variables that are required to be added are fractions and not integers.

A test is a controlled sequence of operations that produces results that can be evaluated. The extent to which the results match the specifications is a measure of correctness. If the results compare favorably with required outcomes the tests are passed. A test driven approach never results in a formal proof of correctness for a software component. Rather, the level of trust in that component grows based on the range of alternate tests that it has been subjected to.

2.4. Practical benefits of automated testing

The example above makes use of the existing NUnit (NUnit, 2005) framework for software unit testing. Using NUnit, each module, sub-system and whole system of code requires a series of tests and data sets that test its specification. Once tests are written they can, and should, be executed regularly, whenever the source code changes. The use of an automated testing framework such as NUnit, provides advantages such as reliability, repeatability, comprehensiveness, reusability, and improved testing times. The following steps are a brief guide to establishing a robust testing methodology based on automated testing.

Define software specifications which include functional and non-functional requirements.

Develop a test plan. Defining who will, what will and how will the testing be performed.

Develop test cases. Individual tests should be designed to cover each specification.

Execute tests. Using automated tools perform unit and system tests. Use acceptance testing to identify usability issues.

Evaluate results. Automated testing tools will produce a list of functions/areas of code that are incorrect and need fixing. Evaluating the usability test will require testers to prioritize solutions to users' responses.

Respond to evaluation. Make recommendations for changes to the code base based on the test results.

3. TESTING FRAMEWORK

The following sections will address the specific problem of developing a testing framework for porting *models* through the use of a testbench. This testing framework is differentiated from lower level frameworks, such as NUnit, by providing specific support for testing models using time series and spatial data.

The testing framework developed for porting consists of tools for reading inputs and reconstructing models, mechanisms for supplying data to the model and recording data, ability to measure correctness, and tools for visualizing and browsing results.

The aim of the testbench is to highlight differences between the same variables of the original model and the newly ported model. Visualizing the results at every time step produces a graph that illustrates magnitude and trends. This directs users to possible sources of error.

The testbench is driven by a tracefile which contains time series data for every input, parameter, state variable and output of the existing, legacy model. Each element is named and configured to represent the model's configuration (see Figure 4). The trace has been constructed from a physically based model Macaque (Watson, 1997) (see case study) which delineates hill slopes into Elementary Spatial Units (ESU's). Each unit (ESU, hill slope and world (whole of catchment)) has a set of parameters that is repeated each time step. Figure 4 illustrates one time step, which would be repeated n times, where n equals the simulation period divided by the time step. The catchment illustrated in Figure 4 contains two hill slopes, one which contains two ESUs and the other with one. This information is used when reconstructing the catchment in the testbench.

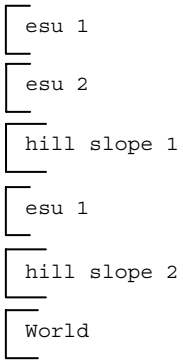

Figure 4: Representation of a trace file configuration.

To ensure that the initial model's stores are stable, the trace file is produced after the first water year has completed.

The testbench uses TIME's introspection for mapping the trace file parameters with those of the new model (see Figure 5). Named variables within the trace file are mapped to variables with the same name or alias in the TIME model. This relationship is then used once the model is executed to play the values into the model and record results. The tracefile parameter list that defines the first time step is used to prime the TIME model.

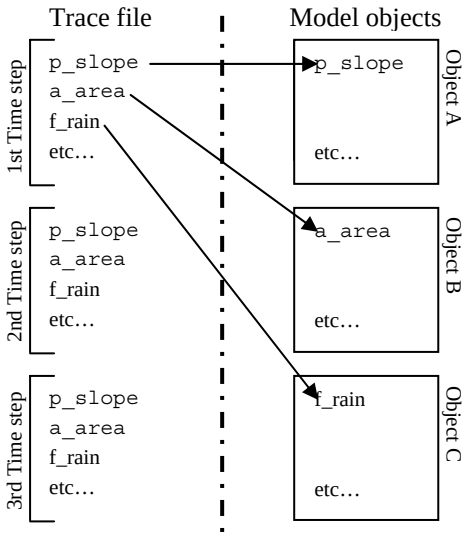

Figure 5: Representing the use of introspection in mapping file elements to model objects.

The comparison of results is measured with a tolerance that ensures that significant differences are highlighted. An example is illustrated in Figure 6.

```
TimeSeries diff = ( TimeSeries )( result - oringinal );
diff.name = result.name;
double originalRange = original.Max - original.Min;
double biggestDiff = Math.Max( Math.Abs( diff.Min ),
            Math.Abs( diff.Max ));

if ( originalRange != 0 )
{
   if(( biggestDiff > 1e-5 ) &&
      ( biggestDiff > 0.0001 * originalRange ))
      diffs.Add( diff );
}
```

Figure 6: Example of identifying significant differences.

Users of the framework are presented with an automatically generated interface (see Figure 7) which is generated by "VisualTime" (Rahman, 2003). The interface consists of a list of parameters from the source (original model), result (new model) and the difference (result – original). Associated with each parameter list is a visual control which is used to view either a time series or spatial map.

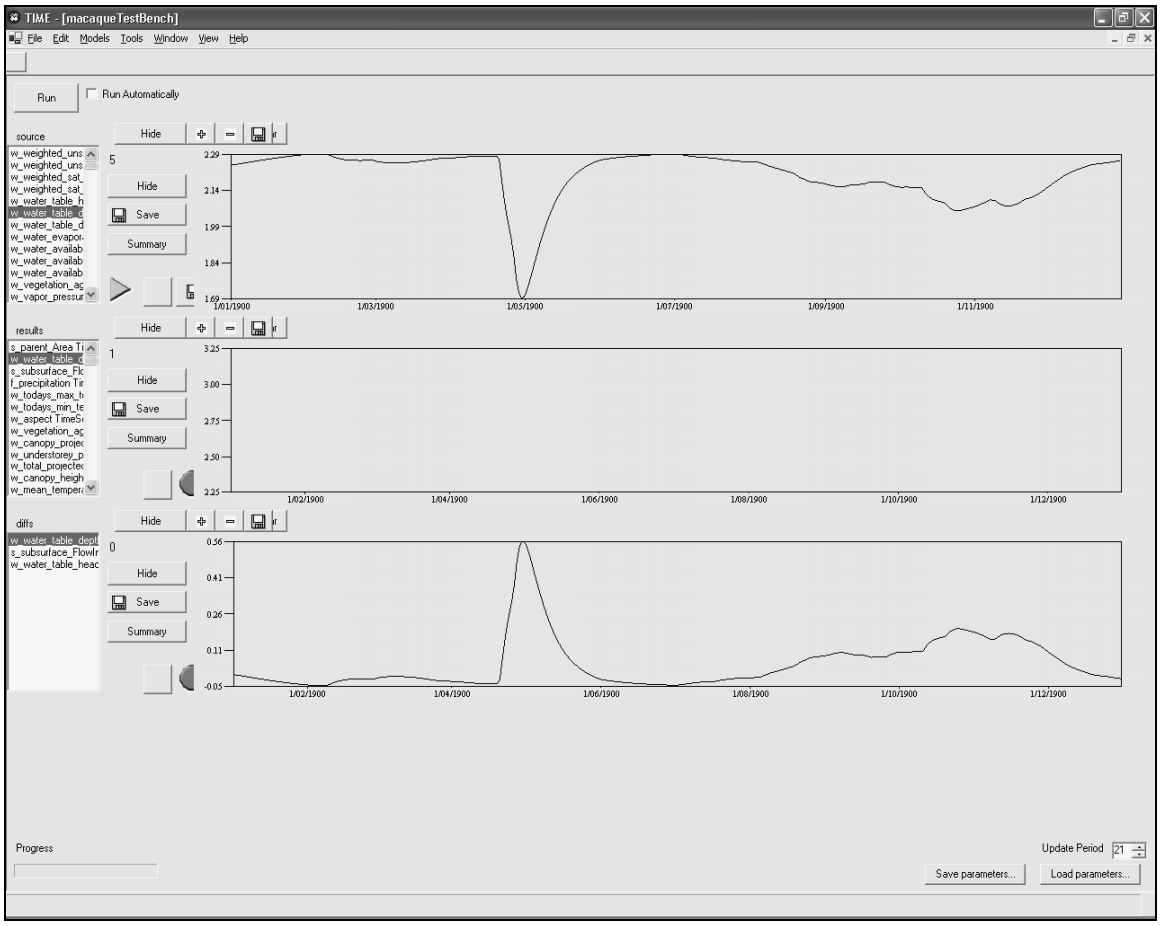

Figure 7: The testbench interface generated by VisualTime illustrating results of a test run.

4. CASE STUDY

The development of the testbench was initiated during a project to port an existing model, Macaque, from a legacy framework Tarsier (Watson, 2003), to a new framework TIME. Macaque is a physically based model, which consists of a large number of variables and hydrological algorithms. To ensure that the porting process did not introduce errors in the modelling behaviours, testing options where explored. The first option considered was to write unit tests for individual functions as well as higher level system tests. This approach would have required a very large effort of analyzing individual functions to extract appropriate test data sets. The second option was to compare the final output of the existing model with the model's output. This simple option could identify errors evident at the model outputs, but had very little diagnostic power to help identify errors within the model. The third option, the development of a testbench, is the option explained. This option was chosen as it allowed a large degree of examination of internal model variables and functions, with much less upfront effort than hand coding individual unit tests. Furthermore, the test bench could be used as a learning tool to explore in detail the behaviour of the hydrological processes being ported.. The test bench highlighted the variables that potentially initiated errors, and because its coverage included all levels of the system, it enabled the development of the core model logic to proceed without a user interface.

4.1. The goals for porting

There were two main goals of this project. The first was to develop a new interface that would accommodate a larger user group. The second was to take advantage of new software frameworks and design patterns that enhanced extensibility, maintainability and reusability.

4.2. The steps in porting

The architecture of Macaque was well designed and the main modules cohesive. This made the task of separating the required modules from the original framework relatively easy.

The following points provide an overview of the steps undertaken in the porting process.

1. Familiarization with both the model, the existing framework and the language.
2. Obtaining a trace file of the existing model. Testing the structure of the trace file to ensure that it accurately represents the current model.
3. Understanding existing limitations and utilizing the new frameworks technology and other software engineering techniques to improve both the model and the use of the model.
4. Rewrite the model into the new framework leaving the essential algorithms intact. Basically wrapping them in the new architecture using software engineering patterns. The core science may be ported intact and tested before the interface or persistence layers are developed.
5. Write a mechanism for reading the trace file and reconstructing the model's configuration into the new framework.
6. Use the testbench to compare models.

4.3. Using the testbench in the porting process to identify possible areas of fault

After running the testbench, two additional data sets are produced, the results of the new model and the differences between the old and new. By synchronizing the outputs of each set, users are able to compare results for a selected variable. The magnitude and trends illustrated in the results and differences indicates possible causes of error. An example can be seen in Figure 7, the result graph shows that the variable is constant. This indicates potential error in the code and needs to be investigated. Finding the selected variable in code and seeing if the variables used to calculate it are also in the difference list would be the first step. If one or more of these variables are also found to be incorrect the process is repeated until the problem can be isolated to the code itself and not an input to the calculation. Comparing the code of the models would be the next step. Sometimes the error is a problem with differences in language use but usually it is an error with the intended logic.

4.4. Selection of trace files

A single tracefile is unlikely to exercise every facet of model behaviour (or misbehaviour), although by testing many elementary spatial units within a single whole-of-system tracefile, a wide range of situations can be covered. Nonetheless multiple tracefiles are necessary to gain confidence in the model implementation. The selection of trace files is based on similar guidelines used in the selection of data for use in unit tests. It is not possible to test all possible variations of model configurations, however it is possible to test the extents of use. With knowledge of the behavioural extent of a system, a range of tests can be derived. For example, with a model that models hydrological processes including snow pack and snow melt, separate tracefiles might be selected to test scenarios with no snowpack, part year snowpack and year round snowpack.

5. CONCLUSIONS

The paper has described the motivation, principles and methods of porting a module from one modelling framework to another. The advantages of a testing method are described and illustrated with a case study.

The use of a software testbench is an efficient and robust way of preserving model behaviour during the porting process.

It is recommended that modellers undertaking the porting process start by developing a good understanding of the original model's specifications develop a testing plan and take advantage of new technologies.

6. ACKNOWLEDGMENTS

The testbench has been developed with the assistance of members of CSIRO Land and Water's Environmental Sensing, Predicting and Reporting software engineering team, Fred Watson and Murray Peel.

7. REFERENCES

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994), Design Patterns: elements of reusable object oriented software, Addison Wesley.

Meyer, B. (1997), Object-Oriented Software Construction: (2nd ed) Upper Saddle River, N.J.: Prentice Hall

NUnit (2005), NUnit unit-testing framework, http://www.nunit.org/ Last Accessed August 9, 2005.

Rahman, J.M, S.P. Seaton and S.M. Cuddy (2004), Making frameworks more usable: using model introspection and metadata to develop model processing tool,

Environmental Modelling and Software, 19th March, 2004, pp. 275-284.

Rahman J. M., S. P. Seaton, et al. (2003), It's TIME for a new environmental modelling framework, Proceedings of MODSIM 2003, (4), 1727-1732

Watson, F.G.R., R.A. Vertessy, & R.B. Grayson, (1997), Large scale, long term, physically based prediction of water yield in forested catchments. Proceedings, International Congress on Modelling and Simulation (MODSIM 97), Hobart, Tasmania, 8-11 December, 1997, p. 397-402.

Watson, F.G.R. & J.M., Rahman (2003), Tarsier: a practical software framework for model development, testing and deployment. Environmental Modelling and Software, 19:245-260.