A Case Study of Model-Based Testing

Using Microsoft Spec Explorer and Validation Framework

Carleton University
COMP 4905

Andrew Wylie
Jean-Pierre Corriveau, Associate Professor, School of Computer Science
17/04/2012

Abstract

Model-based testing (MBT) is a method of software testing which uses a specification of the behavior of an implementation for the creation of test cases. This report will focus on two different MBT tools used for software testing. They are Microsoft Spec Explorer and Validation Framework. The tools will be examined through the use of a case study of a stopwatch system. The stopwatch program was first implemented, followed by the development of model programs for both Spec Explorer and Validation Framework. Following this, test cases were created and run for both of the MBT tools. This allowed a comparison of the operability of the tools with respect to testing of the implementation. The report includes a description of the problems encountered during development as well as a comparison of the MBT tools. Overall, Spec Explorer allows for the automatic generation and running of test cases, however it does not allow for as comprehensive testing as Validation Framework. In either case, the testing of a software system should not be limited to a single tool, as this does not guarantee full coverage.

Table of Contents

List of Figures

List of Tables

Project Description

Software engineering, in a broad sense, is a collection of techniques, tools, and methodologies which help to develop a high quality software system within certain constraints. Of all of these techniques and methods, one common and important element is that of testing; as a software system doesn't break, but is delivered broken. This report will examine model-based testing (MBT) in particular, and investigate both the extent and ease at which several tools are able to assist in the development process.

The tools investigated are Microsoft Spec Explorer, and Validation Framework (VF). As mentioned, both are MBT tools; though they provide testing abilities using separate approaches. Spec Explorer is a commercial state-based testing tool provided by Microsoft as an extension for Visual Studio. It allows system-level testing of the solution under test (SUT) while taking a black-box approach to testing by comparing only input and output values between the model methods and implementation methods. After a model program is constructed, it can be explored to derive a state machine which represents the possible paths of execution of the program being modelled. At this point, tests can be automatically generated from the model program and run on the implementation to determine conformance. Validation Framework however allows for both system-level testing, and for testing of values during execution of the implementation. This is able to be accomplished as VF runs on top of the implementation and uses observability methods to gather data from the implementation at specific points during execution.

To properly compare the tools, the SUT was first written according to its specification. After the system was developed both test models were then written. Following this, tests were created and run to determine testability provided by the tools.

The system implemented was a stopwatch program that has two separate display modes which allow for the current time to be displayed, or for a timer to be shown. When in timer mode, the user can start, stop, or reset the timer, in addition to being able to freeze or unfreeze the display while the timer is running. These functions are provided by three buttons on the watch; start/stop, mode, and reset/lap. FIGURE 1 along with FIGURE 2 show the system as tested by the tools. A full specification for the system is also available in Appendix A: Stopwatch Specification.

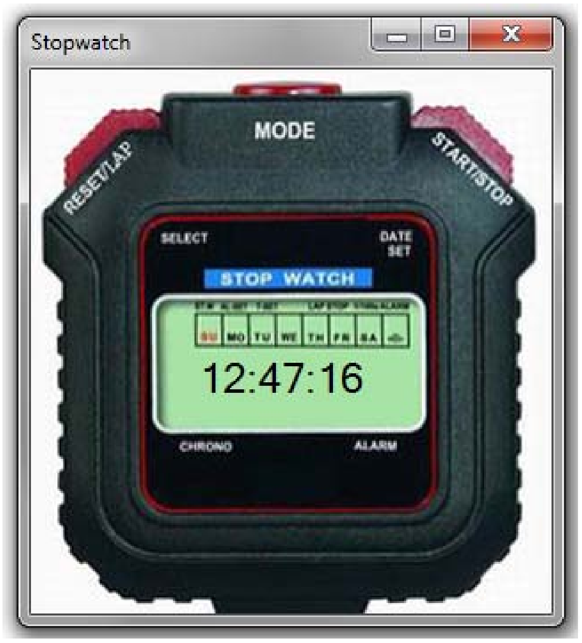


FIGURE 1: Stopwatch system in Date/Time mode.

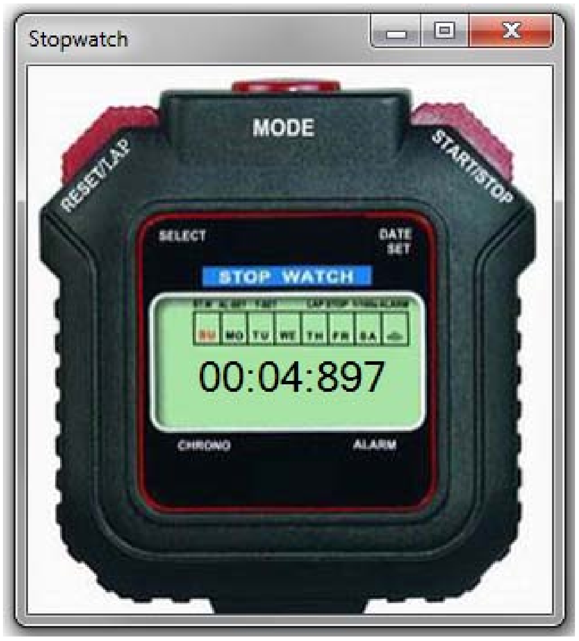


FIGURE 2: Stopwatch System in Timer mode.

Validation Framework

Validation Framework models an implementation using Another Contract Language (ACL) to specify certain behaviours which are then associated with functions carried out by the implementation. When a model method is bound to a function in the implementation, the bound model method is run whenever the corresponding function in the implementation is run. This technique allows VF to test state as Spec Explorer does, as well as allowing tests to be run on variable values of the implementation during runtime. The model program was created with this in mind; first by implementing methods to check state program state, and then adding additional checks to test the runtime values of the displayed time and timer. It is important to note that similar to Spec Explorer, there were some problems encountered during development. However in the case of VF, most problems had to do with the stability of the Validation Framework itself along with the difficulties of testing time data contained in the implementation.

Some of the problems observed with the Validation Framework are as follows:

- The VF project needs to be located on the desktop (can’t have an extended path).
- Observabilities in VF do not properly return lists.
- Static methods in the implementation cannot be bound to.

- The implementation cannot use .Net Framework 4.
- Any bound functions with a ‘void’ return type must have an explicit ‘return’ statement.

Once these issues were identified progress was able to be quickly made on the VF model program. Responsibilities - which are most similar to rules in a Spec Explorer model program - were written for each method of the implementation to cover state testing. Observabilites combined with variables in the model program were then used to collect the state data before a transition was made. The gathered data would be tested against the current model state, and then be transformed according to the applied transition to yield oracle data for testing after the transformation.

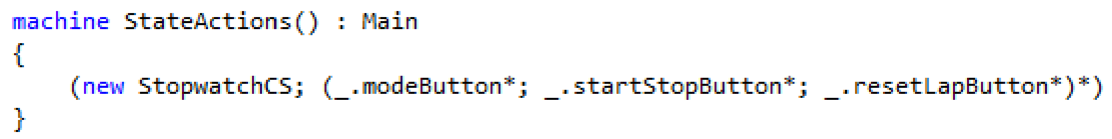
After the state testing code was in place, an invariant was added to the model program for the purpose of testing that the displayed time data was actually correct. As the model program kept track of its own state as an oracle for testing, this was used to determine which of the date or timer information should be displayed. The value retrieved from the implementation for either date or time was then compared to the displayed value. During testing it was found that this process to check the displayed values with the actual values did sometimes give false errors. The issue is that since VF runs on top of the implementation it is testing, any time measurements will be off. This caused errors in comparison when the timer was running, as the data gathered from the observabilities would return the timer value milliseconds apart from each other. In addition to this, in time mode if a second passes as the program is switching between modes an incorrect time will be recorded by the observability due to the processing time taken by switching modes. The first issue was dealt with by simply not testing the timer value if it is running, and the second issue can be dealt with by the setup of test cases. This is possible as the VF test cases must be manually run, so the tester can simply make sure to switch modes only when the time will not change as the mode is being switched.

Spec Explorer

Model-based testing with Spec Explorer consists of a model program to represent the actions and state of the implementation along with a configuration file to describe state paths with respect to the model program. Upon creation, the model program can be either static or instance-based.

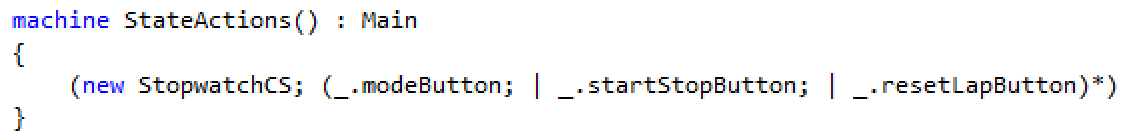
Static models are meant to be paired with a static implementation program, while an instance-based model is to be used when the implementation is also instance-based (Kicillof, 2009). For this reason, the model program for the SUT was first created to be static as the system was planned to be static due to its low complexity. Further into development however, the model program was changed to be instance-based for reasons related to this particular study. When hooking the VF and Spec Explorer models to the implementation, it was found that VF was unable to have bindings to static methods, forcing the implementation, and therefore the Spec Explorer model to be instance-based (unless a test adapter was to be used).

The development of the model program was mostly straight forward, with some problems being encountered along the way. To begin with, the model program was written to cover basic state of the implementation. This was done using an enumeration for the timer modes as well as variables to keep track of display related states. Additionally, several of Spec Explorer's built-in conditions were used to enforce that only valid transitions could be traversed between the derived states. At this point in development, the model was explored to find that every transition was listed twice. After some testing it was found that the issue related to the way that the machine was defined for state exploration. The original grammar for the machine is shown in FIGURE 1, which was simplified to give the working grammar shown in FIGURE 2. This change resolved the issue and development was able to continue.



```
machine StateActions() : Main
{
    (new StopwatchCS; (_.modeButton*; _.startStopButton*; _.resetLapButton*)*)
}
```

FIGURE 3: Kleene star used to describe state expansion.



```
machine StateActions() : Main
{
    (new StopwatchCS; (_.modeButton; | _.startStopButton; | _.resetLapButton)*)
}
```

FIGURE 4: Or operation used to describe state expansion.

With the basic model completed and exploration returning a proper state diagram for the system, it was now time to attempt any additional system testing which Spec Explorer may have allowed.

A Datetime variable was introduced into the model with the hope that the generated tests could both check state after each transition as well as test the display. Along with the Datetime variable, a diagnostics Timer object was added to keep track of timer actions from the model. For the model to compile and be explored properly, the newly added types had to also be declared as native types in a modeling assembly (Microsoft, 2012). After this was done, the model was explored and tests were generated. However, it was at this point that it was discovered that code was compiled at the stage of test generation, and that Spec Explorer along with the test framework have no notion of time. As mentioned previously, testing of the implementation is done by comparing method input and output values of the implementation with those produced by the model. The implementation is simply executed using the test data as input, with the model being executed along the same path to generate oracle data. The outcome of the added testing in this case would cause the generated tests to use the current time as oracle data, which would always fail as the tests would be run at a point in time after they were generated. These issues with the model program were solved by the removal of the display checking code, which left the base model program that would successfully monitor and test program state. In the end, the tested aspects of the implementation included watch mode, display mode, and timer mode.

Compare & Contrast

After the development of both the Validation Framework and Spec Explorer model programs, tests were generated and run against the implementation for both systems. For VF, there is no way to automatically generate test cases, so paper test cases were written to match the tests generated by Spec Explorer. The written test cases can be found in Appendix B: Stopwatch Tests.

Both tools succeeded in testing the implementation with respect to a certain base level. State of the system; including the watch mode, display mode, and timer mode were all properly confirmed by tests run from both systems. The Validation Framework model program also included an additional invariant to check that the displayed value of the watch matched with both the calculated state, and with the internal value kept by the implementation. This level of testing was unable to be reached with Spec Explorer due to the issues relating to timekeeping and test generation noted earlier. However, Spec Explorer took overall less time for testing than

Validation Framework as the tests could be directly generated from the model, and run automatically. This was especially helpful when alterations had to be made in the model due to any incompleteness or errors.

Overall, both tools performed adequately to test the specified implementation. Differences between the tools can be seen with respect to their ability to model the system, along with any additional features which they may introduce to speed up the development process. In regards to this, we have seen that Spec Explorer allows for both the automatic generation and running of the tests cases themselves. This is a great benefit over having to manually create, write out, and run test cases as is done with Validation Framework. However, when it comes to modelling the implementation, Spec Explorer is bound to using a state-based approach with formal rules and well-defined state machines to allow this automation. Modelling with Validation Framework can however be done in any way, which can allow for more coverage of the SUT. This point is demonstrated in the case study, as the model created using Validation Framework covers testing of the stopwatch display along with the system state, whereas the Spec Explorer model only covers testing of the system state.

One more difference between the tools has to do with how they handle certain condition checking. This condition checking can be done with Spec Explorer through the use of some of its built in features, though for more complex models there may have to be additional variables introduced to assist in testing. Any variables added to the Spec Explorer model are automatically included as part of the system state, and will increase the state space of the explored model. This occurred during development when a Datetime variable was introduced to allow for display checking, which caused an explosion of states which could not be avoided. Introducing variables to a Validation Framework model however does not have any effect on the function of the model until its use is directly specified in the code.

Overall, the case study examined the MBT tools using a base level of features provided. As a result of this, the operability between the tools was a trade-off between automation with strict restrictions on the model implementation, and no automation with freedom to create a model program to better fit the implementation. A look into the advanced capabilities of Spec Explorer’s online testing abilities (On-The-Fly tests) as a comparison to the default offline testing (Test Code Generated testing) may be valuable, as this would be more similar to the online testing done by Validation Framework. In addition to this, research into the ability of the tools to deal with larger systems would be interesting, as it would provide more information about their scalability, and allow usage of the more advanced features provided by the tools.

References

Kicillof, N., 2009. *Static vs. Instance-Based Models.* [Online]
Available at: http://blogs.msdn.com/b/specexplorer/archive/2009/11/23/static-vs-instance-based-models.aspx
[Accessed 2012].

Kicillof, N., 2010. *Channel 9 Blog.* [Online]
Available at: http://channel9.msdn.com/blogs/nicok/
[Accessed 2012].

Labiche, Y. & Shafique, M., 2010. *A Systematic Review of Model-Based Testing Tool Support,* Ottawa: Carleton University.

Microsoft, 2012. *NativeType Attribute.* [Online]
Available at: http://msdn.microsoft.com/en-us/library/ee620483.aspx
[Accessed 6 March 2012].

Appendix A **Stopwatch Specification**

The specification for the stopwatch system was taken from the Spec Explorer Video Tutorial series found on the Channel 9 Blog. The specification is shown at 16:55 into Session 2, Part 1, and has also been duplicated below for quick reference. Some adjustments were also made to eliminate any unspecified behaviour.

Two display modes
- Date & Time
- Timer

Three Buttons
- Mode (always enabled)
  - Switches between the Timer and Date & Time modes.
- Start/Stop timer
  - Only available in timer mode.
  - Starts or stops the timer depending on current watch mode.
  - Unfreezes the display on a stop action.
- Reset/Lap timer
  - Only available in timer mode.
  - If the timer is running then the button freezes or unfreezes the display.
  - If the timer is stopped then the button resets the timer to zero.

Appendix B **Stopwatch Tests**

The test cases used to test the implementation are available below. They are the same tests that are created by Spec Explorer test generation, and have also been checked to ensure that they cover all possible paths of execution in the implementation.

The initial state of the implementation is shown in the left column of the first row, with the next action to be triggered listed on the same row on the right column. The resulting state after execution of an action is located in the row below the previous state. As shown in the tables, the current program state is displayed in the order of: Watch Mode / Timer Mode / Frozen Display.

TABLE B-1: Timer Start/Stop Test

State	Action
DateTime / Reset / False	ModeButton()
Timer / Reset / False	StartStopButton()
Timer / Running / False	StartStopButton()
Timer / Stopped / False	StartStopButton()
Timer / Running / False	

TABLE B-2: Reset Timer Test

State	Action
DateTime / Reset / False	ModeButton()
Timer / Reset / False	StartStopButton()
Timer / Running / False	StartStopButton()
Timer / Stopped / False	ResetLapButton()
Timer / Reset / False	

TABLE B-3: Timer Stopped, Toggle Display Test

State	Action
DateTime / Reset / False	ModeButton()
Timer / Reset / False	StartStopButton()
Timer / Running / False	StartStopButton()
Timer / Stopped / False	ModeButton()
DateTime / Stopped / False	ModeButton()
Timer / Stopped / False	

TABLE B-4: Timer Running, Freeze & Unfreeze Display Test

State	**Action**
DateTime / Reset / False	ModeButton()
Timer / Reset / False	StartStopButton()
Timer / Running / False	ResetLapButton()
Timer / Running / True	ResetLapButton()
Timer / Running / False	

TABLE B-5: Stop Running Timer with Frozen Display Test

State	**Action**
DateTime / Reset / False	ModeButton()
Timer / Reset / False	StartStopButton()
Timer / Running / False	ResetLapButton()
Timer / Running / True	StartStopButton()
Timer / Stopped / False	

TABLE B-6: Switch Modes with Frozen Timer Test

State	**Action**
DateTime / Reset / False	ModeButton()
Timer / Reset / False	StartStopButton()
Timer / Running / False	ResetLapButton()
Timer / Running / True	ModeButton()
DateTime / Running / True	ModeButton()
Timer / Running / True	

TABLE B-7: Switch Modes Test

State	**Action**
DateTime / Reset / False	ModeButton()
Timer / Reset / False	ModeButton()
DateTime / Reset / False	

TABLE B-8: Switch Modes with Running Timer Test

State	**Action**
DateTime / Reset / False	ModeButton()
Timer / Reset / False	StartStopButton()
Timer / Running / False	ModeButton()
DateTime / Running / False	ModeButton()
Timer / Running / False	